PolyBase: Adapting to Data Affinity Changes in Geo-Replicated Database via Row-Level Consensus-Group Affiliation Re-Assignment

Chaoyi Ruan*
USTC
rcy@mail.ustc.edu.cn

Yingqiang Zhang
AZFT
yqzhang1987@gmail.com

Juncheng Zhang
USTC
xzwj@mail.ustc.edu.cn

Cheng Li
USTC
chengli7@ustc.edu.cn

Xiaosong Ma*
MBZUAI
xiaosongma@acm.org

Hao Chen
AZFT
cighao@gmail.com

Jie Zhou
AZFT
jarry.zj@gmail.com

Feifei Li
AZFT
ricosfeifei@gmail.com

Xinjun Yang
AZFT
yangjimmy@gmail.com

ABSTRACT

Transaction performance in geo-replicated databases heavily relies on the request location: when not issued by the primary region, transactions are forced to involve costly wide-area communication. While existing systems distribute primary roles across regions, such assignment typically occurs at the shard level, making it difficult to align with geographically dispersed access to individual records.

This paper introduces PolyBase, a pioneering architecture to address such misalignment, leveraging the widely adopted Paxos-based log replication mechanisms. It enables flexible *row-level consensus group affiliation*, which runs on an *unchanged Paxos protocol*, but *dynamically re-assigns* database rows between Paxos log replication groups, whose leaders become the primary region, enjoying faster writes and up-to-date versions for reads. With carefully designed data structures and protocols, PolyBase significantly reduces wide-area RTTs without compromising transaction or log replication consistency or reliability guarantees. We implemented PolyBase with optimized re-assignment policies and integrated it into two popular databases (RocksDB and MySQL). Our evaluation on AWS, using a production e-commerce workload and microbenchmarks confirms that PolyBase offers significantly higher transaction throughput and lower average/tail latency compared to baselines.

PVLDB Reference Format:
Chaoyi Ruan, Yingqiang Zhang, Juncheng Zahng, Cheng Li, Xiaosong Ma, Hao Chen, Jie Zhou, Feifei Li, Xinjun Yang. PolyBase: Adapting to Data Affinity Changes in Geo-Replicated Database via Row-Level Consensus-Group Affiliation Re-Assignment. PVLDB, 18(3): 702 - 714, 2024.
doi:10.14778/3712221.3712236

*The work was partially carried out during the authors' appointment at QCRI, and Chaoyi's appointment at MBZUAI.

Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712236

1 INTRODUCTION

Geo-replicated databases [3, 10, 17, 40] rely on state machine replication enabled by consensus protocols such as Paxos [23, 47], and offer fault tolerance and performance advantages. They have been building blocks for conventional Internet services such as e-commerce as well as emerging ones such as smart vehicles.

For highly scalable and low-latency processing, the common practice in implementing geo-replicated databases is combining data sharding and the leader-based consensus protocols such as Paxos. They partition large tables (usually by key ranges) into shards, and naturally adopt data shards as the granularity for state machine replication, tying each shard to a dedicated consensus replication group. Data has a unique *primary region*, usually the same region assigned to be the group's *leader*. With such an architecture, transactions (including read-only ones) are much faster when executed in the region that owns consensus group leadership of the data involved (more details in Section 2).

Unfortunately, such shard-level leadership management cannot be coordinated with the individual movement in data regional affinity, meaning most requests for the target data are sent to a specific region in a time duration. In Alibaba, two major scenarios highlight this problem. The first lies in its classic business of e-commerce, where user traffic could be redirected to less-utilized regions to mitigate workload spikes caused by best-selling products or promotion events. The second arises with its new services supporting smart vehicles, which were born to be cloud-native, providing crucial real-time data processing, such as traffic information, navigation services, charging guidance, and vehicle/user history data. As vehicles cross regional boundaries, the associated service cloud changes accordingly and instantly. To ensure a consistent and reliable user experience, cloud databases must seamlessly handle queries when users move across different geographical regions.

Under both scenarios, a small part of a data shard, whose consensus group is led by Region A, could suddenly receive sustained accesses primarily from Region B. Currently, there are no solutions to this *intra-shard, fine-grained access locality misalignment*, even when the sharding granularity can be reduced to megabytes, and

Paxos groups can be merged to manage many shards. These "foreign" accesses need to pay higher latency, through mechanisms such as request forwarding or 2PC, going through the region assigned to lead the entire shard. This is unacceptable to new applications like autonomous driving, where even minor delays can have critical safety implications. Such coarse one-leader-per-shard assignment also makes it harder for the database to adapt to workload skewness or worse, dynamic shifts in workload skewness.

The above scenarios demand dynamic and fine-grained consensus group assignment and multi-primary capability. While recent Paxos protocol innovations (*e.g.*, WPaxos [2] and EPaxos [46]) aim to achieve dynamic leadership, they follow the traditional replicated state machine model, involving themselves only at the commit stage for log replication. Hence they fundamentally lack the coordination mechanisms (such as locks) required by multi-primary DBs during transaction processing. At the database level, there are also recent optimizations re-assigning leadership to regions receiving dynamically shifting data accesses. However, existing solutions fail to transparently, generally, and efficiently mitigate the problem, as they retain the coarse granularity of consensus group leader assignment (Akkio [4] and MGR [33]), or impose additional constraints/requirements, such as centralized coordination and pre-known write sets (DynaMast [1]), deterministic transaction execution (SLOG [40]), or database schema modification (CockroachDB [44]).

In this paper, instead, we propose a novel architecture, PolyBase, to eliminate misalignment between the dynamic fine-granule data affinity changes and the coarse-grained shard-level consensus group assignment by allowing *dynamic finest-grained row-level consensus-group assignment* for database tables. This decouples the strong binding between data shards and consensus groups, allowing each row of data to freely commit its transaction to the nearest leader at the 'local" region (or individually *colored*, to put the concept in a visual way). When a row/record of a DB table is found consistently accessed by transactions from another region, according to certain locality management policy, it can be individually reassigned/recolored to another consenus group, led by the other region, making subsequent accesses "local" again.

We argue that the efficiency and flexibility of using row-level consensus group affiliations outweigh the cost of storing per-row color metadata. Less obvious, though, is that this approach actually *simplifies* replication group setup itself, as regardless of the total data size or the number of shards, fully replicated data across N regions needs only N consensus groups, one led by each region. Additionally, the color attribute operates transparently, enabling automatic, fine-grained locality annotations without requiring DB administrators or developers to account for locality migration or sharding configurations. This is especially appealing given the mixture of end-user mobility and cross-datacenter load balancing.

PolyBase enables row-level consensus-group affiliation management and primary region appointment without using centralized managers or imposing additional constraints such as deterministic transaction executions. Rather than modifying a consensus protocol directly, we leverage mature consensus protocols for log replication, treating group affiliation reassignment operations as standard database operations. This also allows PolyBase to work atop different leader-based consensus protocols. By building its distributed row-level color management on top of existing log replication schemes,

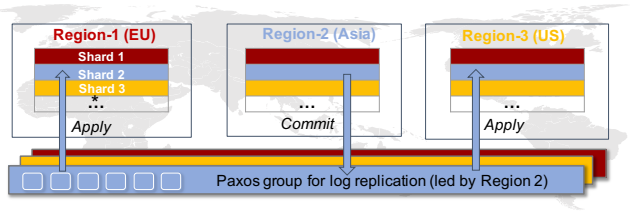

Figure 1: Geo-replicated DB w. Paxos-based log replication

PolyBase contains its complexity within a few additional plug-in modules to existing DBs and introduces no changes to their transaction consistency. In addition, we adopt special care in supporting database integrity and designing access hotness- and correlation-aware policies for optimized row recoloring.

Finally, our PolyBase prototype extends Rocks-DB [15] and MySQL [35], using Paxos for demonstration. Our in-depth evaluation used 18 nodes distributed in 6 AWS regions across 3 continents, running a production shopping cart workload as well as widely-used benchmarks such as TPC-C and YCSB+T. The results confirm that PolyBase smoothly adapts to dynamic request affinity changes and by its agile, fine-grained row recoloring, delivers significantly higher transaction throughput and lower average/tail latency than offered by the best baselines we could find (both open-source and commercial). In particular, in tests simulating real-world request mobility, PolyBase outperforms its closest contender by 1.5×-6.9× in aggregate throughput and 48%-86% in tail latency.

2 BACKGROUND AND MOTIVATION

2.1 Geo-Replicated Database and Paxos Setup

Figure 1 illustrates the common architecture of modern geo-replicated databases, such as Spanner [10], F1 [43], TiDB [19], CockroachDB [48], and GeoGauss [52]. The data (tables) are split into multiple shards (key ranges) for scalability. Geo-replicated databases use consensus protocols such as Paxos [23] and Raft [32] for fault tolerance. In a typical setup, the nodes carrying the same shard also form a per-shard consensus group. While our design applies to both protocols, we focus on Paxos for clarity and brevity.

Each consensus group appoints a leader or primary node in one region, which doubles as the *primary region* for the replicated DB, with other regions as *secondary* ones. We use the terms "primary" and "leader" interchangeably in this paper. Only the primary has the right to write and always possess the latest version of data, while all nodes can serve read requests. Each physical node can be the primary and secondary for different shards simultaneously. In Figure 1, *e.g.*, Region 2 is the primary region for shard 2 and a secondary for shard 1 and 3, with corresponding Paxos groups color-coded. The primary node commits a geo-replicated write transaction by replicating the write-ahead log entry via Paxos (1 RTT), tagged with a unique *log sequence number*, following the replicated state machine. Secondary regions (followers in Paxos) synchronize with the primary by applying the logs to their local database. We call databases that have their data sharded, fully replicated, and protected by multiple consensus groups as *"multi-primary geo-replicated databases"*.

Compared to *local accesses* within the same region (with a latency typically below 10 milliseconds), cross-region operations are

Table 1: Comparison with multi-primary geo-replicated databases, most of which handle dynamics in data affinity

	Aurora Global [3]	**Spanner** [10]	**DynaMast** [1]	**SLOG/Detock** [31, 40]	**PolyBase**	**WPaxos** [2]
Nature	DB	DB	DB	DB	DB	Consensus protocol
Supported Transaction Type	General	General	A priori write set	Determinisitc	General	OCC/Deterministic + single object
Management granularity	Shard	Shard	Shard	Row	Row	Row
Extra WAN RTTs for coordination excluding replication costs	Forward to primary (1RTT)	2PC (2.5 RTT)	Send RPC to central site (1RTT)	SLOG: global ordering. Detock: wait for complete set from other regions.	Except for migration, RTTs are local	Except for migration, RTTs are local
Extra component for consensus group affiliation change	-	Movedir service	Central site selector	Sequencer/Scheduler	-	-
Consensus group affiliation metadata management	-	Directory	Centralized management	Cache + distributed index; Asynchronous update	Embedded with Row	Extra owner set
Changing policy	-	Background; Locality-based	Before transaction; Cost Model based	Background; Migration after 3 updates	In TXN/background; Hotness/Correlation based	Locality-based

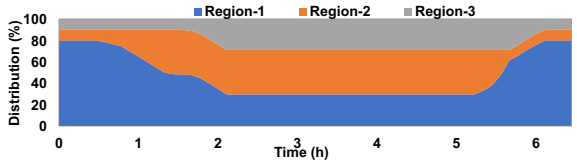

Figure 2: Traffic distribution across 3 public cloud regions from Alibaba Cloud during a 6-hour shopping event

expensive (from 10s to 100s of milliseconds). Clients usually interact with the nearest node in the local region. However, with shards mostly sized at the MB level or larger, cross-region (remote) data accesses are inevitable.

For remote write requests, each write query is dispatched to its primary node (1 RTT). Transactions accessing data records spanning shards led by multiple regions have to use a two-phase commit (2PC, at 2.5 RTTs) to ensure data atomicity, which introduces additional communication overhead. For remote reads, client requests in the primary region directly retrieve the latest data locally. For those in secondary regions, the default process retrieves the latest version from the primary region (1 RTT). Users willing to tolerate data staleness could opt for follower read [11], performing local snapshot read instead. Unless otherwise noted, our discussion/evaluation concerns the default latest read mode.

In summary, co-locating DB query execution with the Paxos leadership is crucial for the performance of industrial geo-replicated DBs due to the significant disparity between the local and wide-area network latency. It is especially important to execute transactions within a single region to avoid the expensive 2PC process.

2.2 Existing Dynamic Leadership Methods

Frequent Data affinity changes. Existing studies [9, 20] show that, despite high data locality, access points often shift across regions due to user mobility and large-scale, planned workload migrations, like smart vehicle movement and traffic re-balance for spikes in shopping applications. For example, our study, based on a 6-hour trace from Alibaba Cloud, confirms such patterns (Figure 2). When a shopping event commenced, the load on Region 1 surged, prompting load balancers to redistribute traffic to other regions. After 2.1 hours, the load was evenly distributed across regions and stayed for a few more hours. Similarly, Akkio [4] observed that up to 50% of Facebook's requests during peak periods were processed in regions different from their origin, and such shifts occur daily.

Current shard-based geo-replicated databases, while popular, face two major challenges with dynamic modern workloads. Firstly, they experience significant cross-region latency when clients and primary data locations are mismatched, as traffic and users frequently migrate between regions. Secondly, their shard-based structure lacks the flexibility needed for dynamic fine-grained data affinity changes in application since Paxos manages data at the shard level and cannot move individual data's leader solely.

Limitations of existing methods. We summarize characteristics of representative multi-primary, geo-replicated databases in Table 1. The first 4 columns are DB designs capable of changing the primary region of target data to accommodate dynamic workloads. However, they suffer serious limitations, as shown in the table.

First, they may require additional services for consensus group changes. DynaMast employs a central coordination site, while SLOG and Detock use deterministic transactions with added sequencers and scheduling layers, both costly for wide-area workloads.

Second, even with inter-group migration, existing solutions still easily incur extra cross-region communication during transactions. Aurora Global and Spanner process cross-region transactions with 2PC, requiring 2.5 RTTs. DynaMast mandates an RPC to its "global site selector" for transaction pre-checks, adding 1 RTT. SLOG uses single-leader global Paxos for transaction ordering and Detock employs all-to-all broadcasts for transaction set synchronization, both leading to scalability issues. In addition, Aurora, with its single primary architecture, needs 1 RTT for request forwarding from secondary regions to the primary.

Third, industrial products like Spanner and Aurora manage data at the shard or database level, lacking flexibility for specific application access patterns. While several other DBs in Table 1 offer row-level management, DynaMast struggles with long cross-region RTTs in geo-replicated setups and defaults to shard-level implementation. Both SLOG and DynaMast employ extra distributed index/cache for managing metadata regarding consensus group affiliations, necessitating asynchronous updates on these extra data structures, thereby increasing the risk of data inconsistency.

Finally, current row-level dynamic solutions restrict transaction types. DynaMast requires a prior write set, while SLOG and Detock only support deterministic transactions. This contrasts with the more general models supporting interactive transactions, adopted by mainstream database application developers [37] and tools [36].

Besides multi-primary DBs, there are proposals on multi-leader or leaderless Paxos protocols. For instance, WPaxos [2] adapts the conventional Paxos protocol to allow a non-leader node to steal

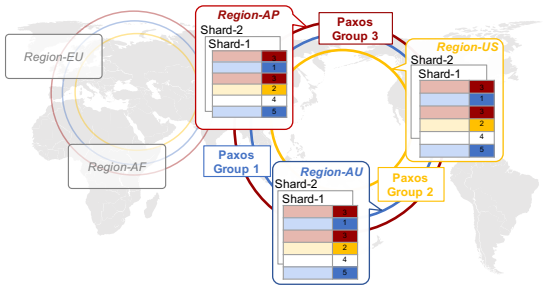

Figure 3: PolyBase system model

the leadership for an individual data object, making it faster in reaching consensus by allowing the region seeing more accesses to lead its log replication. EPaxos [46], leaderless protocol, allows any member node to opportunistically propose/commit non-conflict requests, but still requires a second phase for conflict requests, suffering WAN latency. As shown in Figure 1, the interaction between Paxos variants and DBs occurs at the transaction commit stage, only to determine if the logs can commit globally. In multi-writer scenarios, therefore, they cannot use objects' ownership to hold off another region's concurrent attempt for leadership change. If another node steals the ownership, the original owner's updates are invalidated, leading to abort. Thus, such protocols alone can only support optimistic transaction control (OCC) modes or deterministic transactions (abort and retry upon conflicts), neither feasible in supporting interactive database transaction processing.

In contrast, our proposed PolyBase integrates unchanged Paxos with database transactions, enabling row-level Paxos-group affiliations reassignment and embedding metadata directly in rows. Its recoloring mechanism reduces wide-area RTTs and supports unrestricted transactions. Paxos safeguards recoloring by ensuring correctness through a serial update history linked via Paxos logs.

3 POLYBASE OVERVIEW

3.1 System Model

Geo-replication Paxos group setting. With a *replication factor* of R, PolyBase replicates the database across R geographical regions (out of N total regions) while sharding data within each region for scalability. Modern cloud databases often span dozens of regions, with R typically set between 3 and 5. Figure 3 gives a sample setup, where $N = 5$ and $R = 3$.

Departing from the traditional "one Paxos group per shard" model, PolyBase constructs only R Paxos groups, each led by one of the R regions, as illustrated with different colors. This design not only simplifies management but also reduces overhead by minimizing the number of Paxos groups. With each group handling more data, it enables efficient message batching [6], lowering per-transaction communication costs and increasing throughput. Actually, we observe that a highly optimized Paxos group could handle the entire database instance's traffic within a region in production (30K write transactions/s). If indeed one group becomes a bottleneck, we could easily scale out by creating more groups on demand (*i.e.*, $2R$ groups, two in each color, for the setup in Figure 3). When multiple groups are used per region, the data is partitioned among

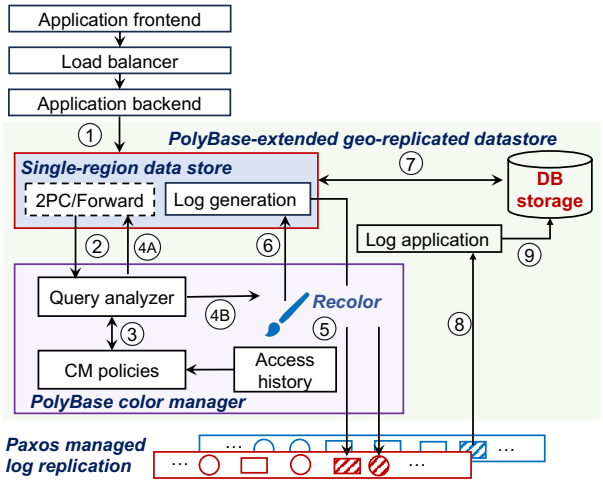

Figure 4: PolyBase Architecture: Building atop Paxos groups

groups by key range split. Without the loss of generality, our discussion stays with the case where we have R groups in total.

Row-coloring for Paxos group assignment By decoupling Paxos group setup with sharding, PolyBase acquires the liberty to enable the ultimate fine-grained data locality management, by coloring individual database rows according to the Paxos group it is currently assigned to. As shown in Figure 3, this is done by appending a per-row metadata field denoting the appropriate Paxos group ID (`GID` for short). With a replication factor of R, there are only R colors. When access locality change is sensed, PolyBase could independently "recolor" a row, assigning it to its new "local region", without physical data movement.

Our main intuition is that with either users' physical mobility or request-redirection for load balancing, a locality change perceived by PolyBase is likely to sustain for some time, at least hours or days. With such independent, fine-grained, and lightweight recoloring, database rows get to have their Paxos group affiliation continuously aligned with the majority of transactions accessing them (hotness) at the moment. Note that for transactions accessing multiple rows with different hottest regions, we may resort to 2PC. However, the idea is that frequently co-accessed data rows will likely have the same color, reducing the number of such multi-color transactions.

One related issue is the efficiency of large, read-only scans, which benefits from shard-level Paxos group assignments. However, in line with the mainstream industry practice, PolyBase provides snapshot isolation regarding scans, leveraging MVCC (Multi-Version Concurrency Control). This significantly reduces the impact of multi-colored data shards, as scans operate simply on the local data snapshot once the Paxos log application has caught up with the specific version. The only additional overhead from the row-level color assignment is to wait for the Paxos-replicated logs to be locally applied. Section 6.4 demonstrates that this overhead is quite minor, especially compared to the wide-area RTT of cross-region.

3.2 Software Architecture

As shown in Figure 4, PolyBase extends a geo-replicated database running common DBs such as MySQL [35] and RocksDB [15]. At the top, the database supports one or more applications (user-facing

Data	Color label (cLabel) (64bit)							
	GID (8-bit)	aGID (8-bit)	aSeq (32-bit)	reset (1-bit)	cSeq (10-bit)	reset (1-bit)	InTransit (1-bit)	Extend (3-bit)
...	2	NULL	NULL	0	3	0	False	...

Figure 5: PolyBase cLabel structure

or internal), typically with an application-specific load balancer dispatching requests to geographically dispersed regions. Underneath the database, multiple Paxos groups are dedicated to log replication.

PolyBase augments the database transaction path with several key components. A *color manager* dynamically assigns row-level Paxos group affiliations (Section 4.2), requiring enhanced *cross-group dependency tracking* for per-object linearizability and a *pre-key* protocol to maintain data integrity during concurrent key insertions and affiliation changes (Section 4.4). Furthermore, PolyBase employs configurable policies for recoloring decisions (*i.e.*, when and what to recolor) based on access hotness and correlations (Section 5.1), optimizing remote transaction performance (Section 5.2).

4 DYNAMIC ROW COLORING

4.1 Row-Level Color Annotation

Row-level Paxos group assignment management requires consistent storage metadata that delineates the relationship between data records and Paxos groups. Rather than using an extra Paxos group to reliably replicate standalone metadata, PolyBase embeds the relevant metadata field within each row, called *cLabel (color label)*.

Figure 5 demonstrates a DB row extended by appending the compact 64-bit cLabel. This structure contains several fields relevant to a row's color management, whose usage will be detailed in Section 4.2. First, the `GID` field (8-bit) identifies the consensus group overseeing a row, which dictates the row's current color. Next, the 8-bit *appliedGID* (`aGID`) and the 32-bit *appliedSequence* (`aSeq`) fields record the group ID and the sequence number of the last applied log to this row, where each log has a unique sequence number within Paxos. As we implement the recoloring process using Paxos itself, the action is propagated across regions using log replication and application as well. This pair of fields ensures that when a row changes color, transaction logs from the previous color group have all been applied, and the row is up to date.

Next, the 10-bit *colorSeq* (`cSeq`) field records the number of inter-group affiliation re-assignments (color changes) of the row, to ensure the color group initiating the row-recoloring is up to date in terms of the row's color-change history itself. Both the `aSeq` and `cSeq` fields reset to zero upon overflow, with an auxiliary `reset` bit to detect wrap-around. Finally, the `inTransit` bit acts as a *local lock*, which is not replicated and coordinates conflicting concurrent requests from the same region during the row's color change. These add up to 61 bits, leaving 3 free bits in cLabel for future extensions (*e.g.*, extending the maximum number of groups). We consider the 64-bit per-row metadata overhead more than justified for performance and flexibility, considering typical modern record sizes of 100s-1000s of bytes [8, 13].

4.2 Recoloring Protocol

The challenge of row-level Paxos group affiliation change (recoloring) lies in achieving consensus on the new color, with per-object linearizability among the recovering events guaranteed while accommodating independent log applications from different Paxos groups and minimizing the recoloring overhead.

Recoloring workflow. We illustrate the process of a sample row's color change in Figure 6, with a formal description in Algorithm 1. This row (record A) is currently blue, meaning only the blue group generates logs for its update, with log application up to the log sequence number (`aSeq`) of 238. Also, the `cSeq` field shows that the row has witnessed 10 recoloring operations.

When a transaction at the red region accesses that blue row and PolyBase considers the workload locality change significant enough (policies being described in Section 5) to warrant a recolor, the red group leader locks the cLabel with the `inTransit` bit locally. Upon lock granting, the leader issues a color change request (`req_cc`) embedding the cLabel to the blue group leader via RPC (line 2-3).

As shown in Figure 6, the bulk of the recoloring work happens in Region 2, which leads the blue group. Here the proper node in the blue region (responsible for the shard in question) first checks (line 9) the row's cLabel, to make sure that the row is (1) indeed blue right now and (2) has not missed previous color changes (the request bearing the same `cSeq` as in the local row). If the check fails, the node returns an error, otherwise enters the recoloring process.

It starts by locking the row in the blue region, locally setting its `inTransit` bit. This local locking procedure coordinates conflicting concurrent requests from multiple remote regions, *e.g.*, Region 1 and 3. To prevent livelock or deadlock, we adopt the common exponential back-off strategy in restarting requests after a lock timeout. Again, once permitted, the RPC continues to create a special color change log (`CC_Log`, of rectangular shape in the Figure 4), as opposed to the normal transaction log (`TXN_Log`, of round shape). This log entry records the blue-to-red GID change (line 11), plus corresponding dependencies used for data consistency (to be detailed in discussing Figure 7). The node appends the log for global replication using the blue Paxos group, commits the recolor transaction, and unlocks its local row by resetting `inTransit` (line 12-13). Note that as the validated current "owner" region, it is guaranteed to have the most recent update to this row, in its local `aSeq` field. When it receives the Paxos ACKs from the other two regions, it wraps up its recoloring operation with a response RPC back to the requesting (red) region, embedding its `aSeq`.

In the background, each region simultaneously employs many threads to apply local DB changes by the replicated logs of the three Paxos groups (line 18-23), as illustrated in the middle part of Figure 6. Here a `CC_Log` would result in a local color change, with the `cSeq` incremented. The red region, as the requester of the recolor operation, needs to ensure that the transaction log application of the row in group `aGID` has caught up to the `aSeq` noted in the blue region response RPC, to avoid operating on stale data. Afterwards, it resets `inTransit` to unlock its local row, proceeds to execute the transaction that triggered the recoloring in the first place, and replicates `TXN_Log` globally.

The recolor operation as portrayed in the figure generally takes two wide-area RTTs, with each wide-area communication step (RPC and Paxos alike) taking 0.5 RTT. While a 2PC-based recolor implementation is also possible, the expensive coordination will be excessive for our row-level color management. Our approach leveraging the existing Paxos log replication infrastructure is much

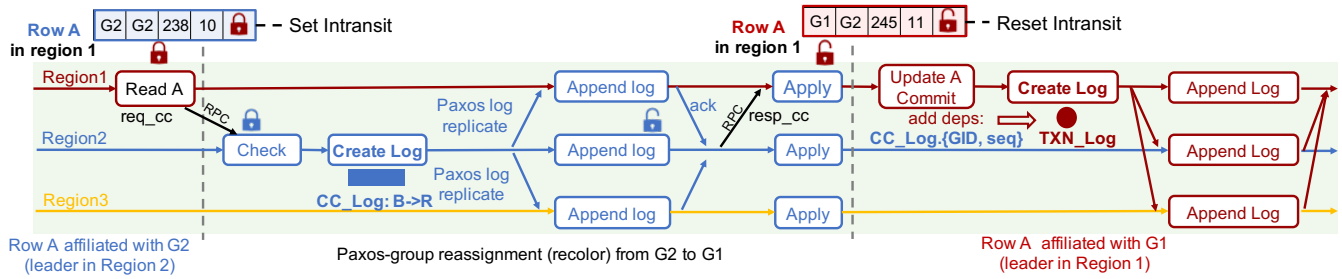

Figure 6: PolyBase recolor operation timeline, with Row A re-assigning group affiliation from Region 2 (blue) to Region 1 (red)

```
1  fun Recolor(row_key, cur_GID, next_GID, cSeq):
2      row = DB.Get(row_key), Lock(row.cLabel.InTransit)
3      res = req_cc(row_key, cur_GID, next_GID, cSeq)
4      if res.succ():
5          WaitSeqUpdate(res.GID, res.Seq)
6      UnLock(row.cLabel.InTransit)
7  fun OnRecolor(row_key, cur_GID, next_GID, cSeq):
8      row = DB.Get(row_key), cLabel = row.cLabel
9      if <cur_GID, cSeq> == <cLabel.GID, cLabel.cSeq>:
10         Lock(cLabel.InTransit)
11         CC_Log = {row_key, cur_GID, next_GID, deps:{cLabel.aGID,
               cLabel.aSeq}}
12         seq = PaxosList[cLabel.GID].replicateLog(CC_Log)
13         UnLock(cLabel.InTransit)
14         return {Code::succ, cLabel.GID, Seq}
15     else:
16         return {Code::error} //version fall-behind or lock-timeout
17 fun ApplyLog(Log):
18     if Log is CC_Log:
19         WaitIndexUpdate(Log.deps.aGID, Log.deps.aSeq)
20         row = DB.Get(Log.row_key)
21         row.cLabel.{GID, cSeq, aSeq, aGID} = {Log.next_GID, cSeq+1,
               Log.seq, Log.cur_GID}
22     else: // Log is TXN_Log:
23         ApplyTransaction(Log)
```

Algorithm 1: Row-level group affiliation recolor

lighter in comparison. With this sample recolor scenario, it is anticipated that most of the subsequent accesses to this row will be issued from the red region, enjoying the local latency in both reads and writes. Additionally, optimizations such as batching multiple operations and prefetching data in advance further minimize latency and reduce overall costs, as detailed in Sections 5 and 6.1.

Cross-group dependency tracking. Row-level recolor operations embedded in `CC_Log` complicate log application at the Paxos follower side. Please note again that at each region, there are multiple Paxos groups (red, blue, and orange in our example) performing independent log applications to the local DB. When a row gets recolored, its affiliation changes from one group to another, with the recoloring event becoming one synchronization point between the log sequences of the source and the destination groups (the parallel log application "tracks" in two colors).

Using our running example, this problem is illustrated by Figure 7 from the viewpoint of Region 3 (leading the orange group), where it

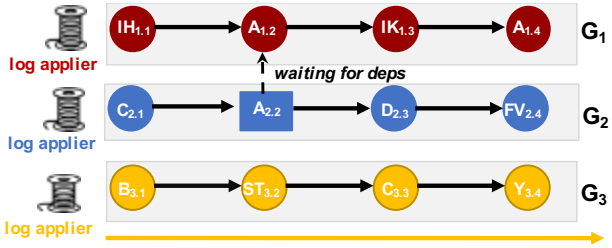

Figure 7: Sample DAG for recolor-aware log application, where rectangles/circles represent `CC_Log` and `TXN_Log` entries. Letters denote the key of row(s) accessed.

simultaneously applies logs from the three Paxos groups. Again the shape of the log entry indicates the event type: normal transactions (round) or recolor operations (rectangular). As row A was recolored from blue to red, only the blue group bears the `CC_Log` ($A_{1,2}$). The first transaction updating A in the red group, however, could only happen after this recoloring, creating a cross-group dependency in log application, as indicated by the dashed edge.

Such a dependency edge is added in the transaction that triggered the recolor. At the right end of Figure 6, the red region executes the transaction after recoloring and adds related dependency information in its `TXN_Log`. In this case, it writes down the `GID` and the sequence number of the `CC_Log`, as notified by the response RPC from the blue group. With such dependency carried in the log entry, a follower region that did not participate in the recolor operation (orange here) learns the cross-group ordering of log entries and ensures that $A_{2,3}$'s application follows that of $A_{1,2}$ (line 19).

It is rather straightforward to prove that the log "nodes" and their local or cross-group dependency edges form a directed acyclic graph (DAG): the edges (local or cross-group) always indicate logical precedence, therefore if there is a cycle, two log entries in the same group (color) would have contradicting ordering. This structure establishes a partial order for all log entries across groups, thereby preserving per-row linearizability (more discussions in Section 4.3).

Failure handling. For PolyBase 's fault tolerance, two cases need to be considered. First, if failures occur prior to recolor or after it has been persisted (i.e., `CC_Log` entry successfully applied to update the `GID` bit), then we offer the same level of fault tolerance as provided by Paxos for that record, as it manages `CC_Log` replication just like that of ordinary `TXN_Log`. If otherwise (when a failure happens in between those), we claim that crashed `req_cc` RPC will not leave the row colors in an inconsistent state. There are further two possibilities. If `req_cc` fails to complete, then the old color persists

and an error message will be returned to the recolor RPC initiator. However, if `req_cc` proceeds to the end, then the `CC_Log` will be written and guarantees that the color change will persist.

Another issue is color handling when an entire region (*e.g.*, the red) is down. In this case, PolyBase goes through Paxos re-election to find a new group leader, *e.g.*, the blue region. Before the red region recovers, PolyBase adopts a lazy approach that does not actively recolor the red rows. Instead, any surviving region accessing a red row would be told to consider it blue and perform the recolor on demand. As region-wide failures are rare and typically short, this avoids unnecessary large-scale recolor operations.

4.3 Correctness in Log Replication

The correctness of our proposal is based on the observation that our row-level consensus group assignment does not alter the linearizability [18] guarantee offered by the underlying consensus protocol, in this case Paxos. More specifically, the object/row-level linearizability delivered by such protocols for geo-replicated database ensures that all replica nodes observe the same sequential order of operations on each row. The correctness of our proposal equals proving that for a given row, the history collectively built by Paxos groups involving dynamic assignment forms a linear order.

To do so, we first define the new history under the context of our row-level assignment. Given a row r, whose affiliation can be assigned within n Paxos groups $G = (g_0, g_1, \ldots, g_{n-1})$, while geo-located users issue a set of operations $O = (o_0, o_2, \ldots, o_k)$ that mutate r. During O's execution, there are m affiliation reassignments, each by a color change request (`req_cc`), generating a sequence of recolor events $E =< e_0, e_1, \ldots, e_{M-1} >$.

Definition 1 (History). *The history of* r is defined as $H = (h_0, h_1, \ldots, h_m)$, where a history segment h_i is generated and maintained by a single Paxos group $g \in G$, concluded by a recolor event e_i that changes r's affiliation from g to g' (and then initiates h_{i+1}).

We then formally define the correctness of PolyBase's row-level Paxos group assignment in the following theorem.

Theorem 1. *Given a row* r*, its history* H *is semantically equivalent to a linearizable order on its operation set* O*.*

Theorem 1's proof builds on the following two lemmas. Lemma 1 asserts that at any given time, for each row, there exists exactly one consensus group whose leader can handle its write operations.

Lemma 1. *For any operation* $o \in O$*, there exists a single* $h \in H$ *such that* o *appears in* h*.*

Proof. There are three cases to consider regarding the occurrence of recolor operations within H:

Case 1: When no recolor happens, the statement holds trivially since the row has been owned by a single Paxos group and follows conventional log replication/application.

Case 2: A recolor is attempted, but does not complete due to failure. Note that the recolor log (`CC_Log`) itself is also replicated by Paxos, which in this case would not reach the consensus to persist the recolor operation. According to the recolor protocol given in Figure 6, r would remain locked when the failure occurs, shielding it from further updates. In this case, the recolor operation would not appear in H at all, with r remain in the color of its source Paxos group g upon recovery. The history H therefore appears the same as in Case 1 above, with all operations located in a single segment, recorded by g.

Case 3: When a recolor operation is successfully executed, the source Paxos group leader g logs the recoloring (`CC_Log`) and replicates it to the majority of nodes. During this transition, all concurrent requests on r are blocked, ensuring that no operations in O are executed until the recoloring process completes. Once the `CC_Log` entry is applied at the region leading g (r's "old primary region"), the lock on r is released. At this point, regions still perceiving r as belonging to g reject writes due to the metadata mismatch, while any write operation retrieving r's updated `GID` directs requests to the new Paxos group g'. When all nodes have applied the `CC_Log` and updated r's color metadata, subsequent operations in O are handled exclusively by g' until the next recoloring. Thus, each operation appears in the appropriate history segment of either g or g', ensuring consistency across segments.

□

Lemma 2 asserts that for any row r, any of its recolor event e connects the history segments managed by the recolor source and destination Paxos groups to form a linear order.

Lemma 2. *For any pair of* r*'s adjacent history segments,* h_i *and* h_{i+1}*,* $o_{h_i,|h_i|-1} < o_{h_{i+1},0}$*, where the former is the last operation in* h_i*, while the latter the first of* h_{i+1}*. ("<" here denotes the "preceding" relationship.)*

Proof. Let g_{src} and g_{dst} be the source and target Paxos groups of the recolor event e, respectively. By the design of our recoloring protocol (see Section 4.2), e is the last operation of h_i, *i.e.*, $e = o_{h_i,|h_i|-1}$. It generates a recolor log (`CC_Log`), which changes r's `GID` from g_{src} to g_{dst}. Write operations issued by the region leading the group g_{dst} could only happen after this log is applied. Following this, there must be a read operation o_{read} executed prior to $o_{h_{i+1},0}$ to fetch the most updated `GID` value of r to determine that it is now colored to be affiliated with g_{dst}. Therefore, we have $o_{read} < o_{h_{i+1},0}$. Because of data dependencies on r's *GID*, we also have $o_{h_i,|h_i|-1} < o_{read}$. By transitivity of happen-before relationship, $o_{h_i,|h_i|-1} < o_{h_{i+1},0}$. As a result, such recolor events at the end of each history segment enforce a strict order between the operations within the adjacent pairs of history segments, chaining them to form a global linear order of operations. □

Together, the two lemmas above ensure that PolyBase also forms a total order on logged user operations with regard to any row (objects), thus delivering the same linearizability guarantee as a single Paxos group without violating Paxos semantics.

4.4 Supporting Database Integrity

Geo-replicated databases extended with PolyBase's recoloring protocol preserve ACID transactional semantics. However, PolyBase's row-level Paxos group affiliation introduces challenges for enforcing data integrity, particularly for constraints like unique, monotonically increasing primary keys (e.g., ascending IDs used in orders, invoices, or tracking numbers). In Spanner-style geo-replication, consensus group assignment is tied to key-range sharding, with new insertions handled by the region leading the shard's consensus group. In contrast, PolyBase assigns consensus groups at the row level, decoupling them from sharding. This allows any region to

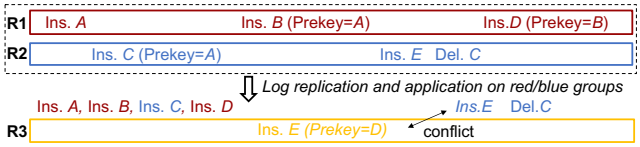

Figure 8: Duplicated insertions due to lost hidden dependency

insert new keys, risking cross-region duplication or violations of uniqueness and monotonicity constraints.

Pre-Key protocol. To this end, we adopt a *pre-key* protocol inspired by established locking mechanisms, such as MySQL's next-key locking [34], but tailored to maintaining data integrity in a geo-replicated database. When inserting a new key, a region must locate the row with the preceding key (*pre-key*) in the primary index and recolor it to its own region. As explained in Section 4.2, this recoloring effectively "locks" the open key range between the pre-key and the key following it. This ensures synchronization during key insertions and prevents inconsistencies across regions.

Augmented pre-key with deleted keys. Handling concurrent insertions and deletions is more complex, as implicit data dependencies between Paxos groups may be lost, leading to potential duplicate insertions. Consider the case in Figure 8: three regions, each leading one Paxos group, generate three streams of replicated transaction logs to insert and delete rows with primary keys A, B, C, D, and E. Although these keys are in ascending order, the transactions may not be. In this case, Region 1 (red) inserts A. Next, Region 2 (blue) inserts C and recolored C's pre-key A as blue. After that, Region 1 proceeds to insert key B, finds A as its pre-key, and recolors A back to red. Region 2 then continues to insert key E, with C (already blue) as the pre-key, then deletes C. When Region 1 replays the log and sees that C was deleted, it uses B (still red) as the correct pre-key to insert key D.

At the same time, Region 3 (orange) is slower in applying the logs from Region 2, unaware when attempting to insert key E that Region 2 has already done so. It identifies D as the pre-key, recolors it to orange, and successfully inserts E again. Such dupliated insertion only becomes apparent when Region 3 later applies Region 2's log with the already committed insertion of E.

Such problems arise because the deletion of C breaks the implicit keys' dependency in different regions (from red D to blue C), manifested in the independent log application of another region. PolyBase fixes this by requiring each region to include, as dependencies in its key insertion log, any mark-deleted keys between the pre-key and the new key (*i.e.*, B and the deleted C when Region 1 inserts D). This ensures that Region 3 cannot apply the red log to insert D until it first processes the dependent blue logs, forcing it to see Region 2's insertion of E before making its own attempt. This is facilitated by the widely-adopted design in modern commercial DBs to not explicitly delete a key but to mark it with a tombstone, thereby preserving dependencies.

4.5 Overhead Discussions

Recolor operation. PolyBase's recolor overhead primarily depends on the wide-area RTT, with each operation incurring about 2 RTTs (Figure 6): our measurement shows that with RTT at 30ms/60ms, PolyBase's has recolor latency of 64.26ms/127.63ms. The recoloring of multiple rows is further batched (Section 6.1), amortizing and minimizing the overhead.

DB transactions with concurrent key insertions. When two regions insert keys concurrently, the conventional geo-replicated DBs rely on a global central coordination module for transaction ordering and conflict avoidance, requiring 1 RTT for request forwarding. In PolyBase, if the target previous keys are local, the data accesses of the corresponding transaction remain local. In the worst case, recoloring remote previous keys adds 2 RTTs.

Dependency tracking. PolyBase's dependency tracking runs in the background with minimal impacts on ongoing transactions. This involves microsecond-level log application and local version checks, both handled locally. In setups with wide-area RTTs (30ms to 200ms), this overhead is negligible.

Overall trade-off. Note that PolyBase trades the aforementioned 2-RTT recolor latency for much cheaper local executions for *multiple subsequent transactions*, as opposed to the 2.5-RTT per-transaction remote accesses paid by conventional DBs. Therefore, when either we have higher RTTs (more expensive remote accesses) or a higher ratio of requests undergoing recolor (more workload mobility), PolyBase obtains more savings despite higher network overhead spent on recoloring itself. In our use cases, such as cross-region data center load balancing or smart vehicle transitions, high regional data affinity with necessary yet infrequent recolor operations allows most subsequent transactions to remain local with an average latency close to 1 RTT (used for replication). The worst case occurs when a row's accesses switch back to Region A right after it meets the hotness requirement to be recolored to Region B. With our motivating use cases, this is unlikely to be other than a rare coincidence.

5 COLORING POLICY

5.1 Criteria for Recoloring

PolyBase recolors individual DB rows for better affinity and latency by placing them in the Paxos group led by the region most likely to access them, which is driven by access hotness and correlation.

Access hotness. Access hotness, measured by recent access count, indicates the potential gain of recoloring a row. The major challenge here lies in aggregating access history globally in a decentralized way. Again, PolyBase leverages Paxos logs double as an access history database. But, it cannot afford to scan the logs to derive per-row hotness metrics on demand when recoloring. Instead, PolyBase adopts an 8-bit Morris counter [26, 30] per row for approximating access count or frequency. This is managed by each Paxos group's log replay thread in a HashMap-based list – 'Heat List'. The counter increments with every write to the row and halves when the time between successive updates exceeds a *decay_time* threshold. Edge cases like execution forwarding are also handled. Specifically, requests in Region 1 are forwarded to Region 2 and then executed; these accesses should be counted for Region 1. For this, we incorporate a `forward_source` field within the log schema to indicate such forwarding.

Access correlation. While the hotness optimization is reactive, PolyBase proactively prefetch row's affiliation likely to be accessed in a region based on observed data association patterns. Inspired by cache prefetching techniques [50], PolyBase employs a lightweight

log-based prefetching layer, which stores the recently updated rows in a hash table with an *update timestamp vector*. The parameter *rsize* specifies how many correlated rows are recorded.

Among the "hot" rows, PolyBase deploys a lightweight data mining algorithm [50] to identify row groups with correlated access patterns by comparing rows' update timestamp vectors. Here, there are two key parameters: *lookahead*, the maximum time distance between correlated rows' timestamp vector, and *freq*, the minimum update frequency a row must meet to enter the mining process. The algorithm has an $O(n \log n)$ complexity, where parameter n denotes the number of rows mined. The prefetching layer is all in-memory and we follow a previous work [50] to limit the prefetching layer size to 5% of available memory. By predicting forthcoming data accesses, PolyBase can "color prefetch" correlated rows when recoloring. Given the common access skewness, the correlation mining could yield further gains in some cases, as detailed in Section 6.5.

Parameter tuning. *decay_time* determines how quickly hotness diminishes, with smaller values risking premature devaluation of hot rows and larger values retaining stale rows. *lookahead* defines the temporal window for detecting correlated rows, where smaller values may miss associations and larger ones risk prefetching irrelevant rows. Optimal values depend on workload patterns: bursty workloads favor smaller settings, while consistent patterns benefit from larger ones. Currently, PolyBase uses static values for these parameters, but future work may explore dynamic tuning.

5.2 Transaction Execution Optimization

Finally, we describe PolyBase's workflow in transaction processing, as outlined in Figure 4. The client requests are firstly sent to servers within their vicinity (①). The request processing begins with query analysis (②), which examines relevant color information based on the cLabel of involved rows. The color manager consults the PolyBase recolor policies (supported by access history data) to decide whether recolor operations are needed, following the appropriate workflow (5.1, ③), before proceeding to DB query execution. The execution strategy hinges on the color distribution of accessed data. PolyBase can recolor remote data (2 RTT) to match the local region if it meets the criteria for hotness or correlation (④B, ⑤, ⑥). If all the accessed data belongs to a single remote region, the transaction is forwarded to that region (1 RTT). In rare multi-row/color transactions where rows have different hottest regions, PolyBase directly uses 2PC (2.5 RTTs, ④A) instead of recoloring, as recoloring all data to one region could negatively impact future transactions on those rows. Notably, if the transaction involves only local accesses (the local region owns all data), the local database executes the query directly (⑦). Ultimately, PolyBase replicates the transaction logs and applies them globally (⑧, ⑨) to ensure a consistent state.

6 EVALUATION

6.1 Implementation Details

PolyBase is a general proposal compatible with various databases, as popular RocksDB and MySQL integration demonstrated. The PolyBase color manager has 8000 lines of C++ code, with an additional 1000-2000 lines for RocksDB/MySQL. We retained the transaction and storage logic in both DBs. For MySQL, we revised its SQL layer and log module for the color manager, while for RocksdDB, we developed a logical log like MySQL's binary log. For both, cLabel is embedded as an invisible field. Log modules are also augmented to record and process recolor-induced dependencies.

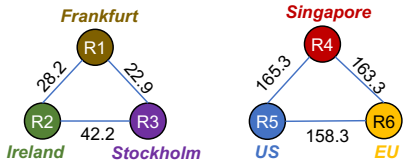

Figure 9: Multi-region group setup with RTT latency (*ms*)

To further reduce recoloring cost, PolyBase batches concurrent recolor requests when possible. For a query accessing multiple rows, we divide these rows into sub-groups based on their colors and consolidate the recolor for the sub-group into a single RPC.

PolyBase introduces no changes to Paxos, and Paxos does not directly access the PolyBase's metadata or DB engine. Though PolyBase defines new log content for group affiliation reassignment, it uses Paxos *solely for log replication*, making it also compatible with other leader-based consensus protocols (*e.g.*, Raft [32]). We implement a `log_replicate(logs)` function, which invokes Paxos logic for replicating `CC_Log` and transaction logs. Therefore, one could replace Paxos with other leader-based consensus protocols like Raft with minor changes.

PolyBase system runs above unchanged Paxos, thus inheriting Paxos' fault tolerance and per-key linearizability guarantee: it functions despite the failure of f out of $2f + 1$ replicas or client crashes. It offers the same transactional semantics as its baseline DB. Finally, it also guarantees *read freshness*, which means once a transaction is committed, all future transactions will see its changes.

6.2 Experimental Setup

Testbed configuration. We use AWS EC2 with *three instances per region across six regions* to form two 3-region groups: one European and one intercontinental. Figure 9 summarizes their region locations and inter-regional latency. Each instance has a 16-core CPU, 64GB DRAM, and ample SSD storage, running CentOS-7.

Baselines. PolyBase aims at dynamic workloads running on geo-replicated databases. For an apples-to-apples comparison, we selected three dynamic multi-primary systems: CockroachDB (CRDB) [44], MySQL Group Replication multi-primary (MGR) [33], and DynaMast [1]. Among them, CRDB dynamically re-partitions data based on workload patterns. The majority of our tests use its open-source version, which only offers partition-level management, while its commercial version (denoted as "CRDB-c") supports row-level data primary region adjustment. MGR rotates leaders for load balancing, similar to Mencius [7]. As for DynaMast, the latest dynamic multi-primary architecture, there is no implementation available and we crafted our own by strictly following its original design and enabling general transaction support.

Each system is configured with synchronous geo-replication for strong consistency. Note that in our target scenarios, the difference in query execution performance due to data structure design is dwarfed by the large cross-region RTT, ranging from 30ms to 200ms.

Finally, we compare MySQL-based PolyBase with the Amazon Aurora Global Database (Aurora) [3], a popular multi-region cloud service also derived from MySQL.

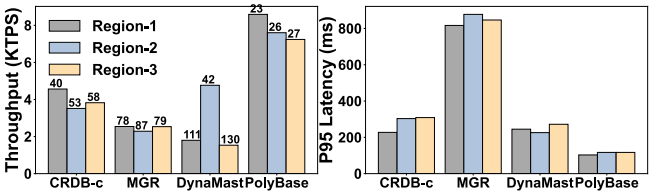

Figure 10: Real workload results (numbers above bars give average latency in ms)

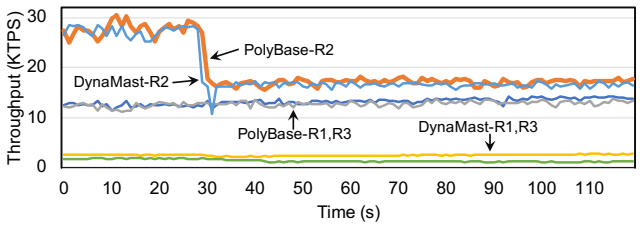

Figure 11: Adaptation to high-level load balancing

Workloads and datasets We test with the popular KV transactional benchmark YCSB+T [13],the widely-used OLTP benchmark TPC-C [8], and a real shopping cart workload collected from our production environment. As MGR's KV support is built on MySQL, we tested YCSB+T by adapting SysBench [21]. Our YCSB+T dataset contains 30 million rows (20GB). Our TPC-C workload has 200 warehouses (20GB), focusing on update-intensive New-order transactions. The shopping cart workload consists of 75% read, 8% insert, and 17% update transactions, accessing a 70GB dataset in 48 tables. In PolyBase, one node only deploys one shard.

Recoloring parameters . We set *decay_time* to 60s, following Redis [39]. We empirically chose a *lookahead* of 5, a *min_freq* of 10, and saved approximately 100K (*rsize*) correlated rows. The mining table size n is set to 1250 rows, as in prior work [50].

6.3 Major Use Scenarios

Baseline real workload. We begin with a baseline scenario where requests possess a natural affinity to users' residence locations. The tests run the real production shopping cart workload within the European group (R1-R3). Requests are distributed to regions by the user ID primary key, which simulates the typical locality around end users' base location, driven by 200 client threads per region.

Figure 10 shows PolyBase's clear advantage in both throughput and latency (average and tail): by implicitly and gradually recognizing each user's "home region", it colors his/her frequently accessed row accordingly and benefits from local accesses. Here we paid to use CRDB's commercial version ("CRDB-c"), which indeed significantly outperforms its open-source counterpart and the other baselines in most cases. PolyBase, however, roughly doubles the CRDB-c throughput, with half average/tail latency.

As we give per-region results here, in most cases R1 performs the best, due to its relatively low latency to both R2 and R3. Interestingly, for DynaMast R2 offers the highest throughput, as it happens to host the global site selector, saving one RTT for requests there.

Load balancing upon request burst. Next, we examine one major source of cross-region requests based on real-world scenarios from Alibaba Cloud when one region undergoes a major workload burst, due to regional events or promotions. Modern cloud infrastructure

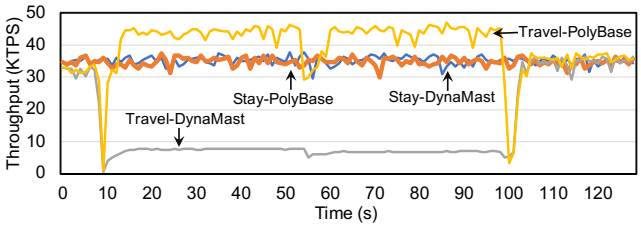

Figure 12: Adaptation to user mobility

allows easy detection of such bursts and quick load balancing action to spread the extra load to multiple nearby regions. Tests in Figure 11 simulate such a scenario by increasing the number of client threads at R2 to 300, while the other two stay at 100. The surge is identified with a configurable throughput threshold (25 KTPS here) and time threshold (30s), then handled by diverging 1/3 of the overflow to R1 and R3 each.

The per-region throughput timelines of PolyBase behave as expected: the R1/R3 throughput steadily increases, finally reaching around 9.24% above their base level before load balancing, consistent with the load increase. The gradual increase reflects the effect of each row warming up to meet the hotness criterion before its recoloring. Requests diverted to R1/R3 enjoy mostly local data accesses due to recoloring, despite paying one RTT per transaction.

For DynaMast, we make the test more friendly by putting its global site selector in R2, thus its R2 throughput matches that of PolyBase before load-balancing. However, after requests are spread, its R2 throughput comes down while R1/R3 throughput does not significantly pick up, with a huge gap in between, as all requests still need to be checked with the global site selector (R2). The penalty is especially harsh for reads, which compose 80% of the shopping cart workload. Here the R2 reads enjoy high throughput (no RTTs in checking with the site selector, no logs to replicate), while the R1/R3 reads pay one RTT. PolyBase's R1/R3, on the other hand, knows each row's color by simply checking its cLabel, making reads diverted to data recolored to a region truly "local".

User mobility. Another source of cross-region access is user mobility, for which we simulate a scenario with users taking road trips following customer demand in Alibaba Cloud. Here we compare two groups of users from the same home region (R2 again), with the "stay" group doing staycations, while the "travel" group taking three types of vacation itineraries: round trip to R1, round trip to R3, and a multi-city trip of visiting R3 and R1, before returning to R2. Figure 12 reports the aggregate throughput for the two groups, again comparing PolyBase and DynaMast.

Not surprisingly, DynaMast and PolyBase remain similar throughput for the "stay" group. When the "travel" group departs (at 10s), both experience a throughput dip as requests become cross-region. PolyBase quickly recovers by recoloring user data to the new regions, even exceeding its original throughput since R2 has higher latency to both R1 and R3. In contrast, DynaMast stabilizes at lower throughput as traveling users continue relying on the site selector in R2. Later, PolyBase briefly dips again when a third of the 'travel' group moves from R3 to R1 but promptly recovers.

Crash Recovery. Another scenario important to any application is fault tolerance in the event of regional failures. Here we check a demanding scenario requiring intercontinental recovery, using

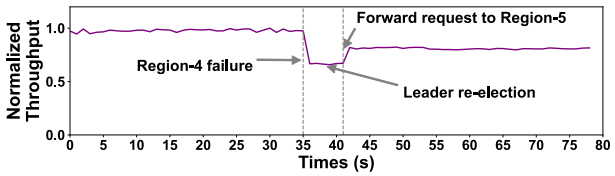

Figure 13: PolyBase recovery upon region-wide failure

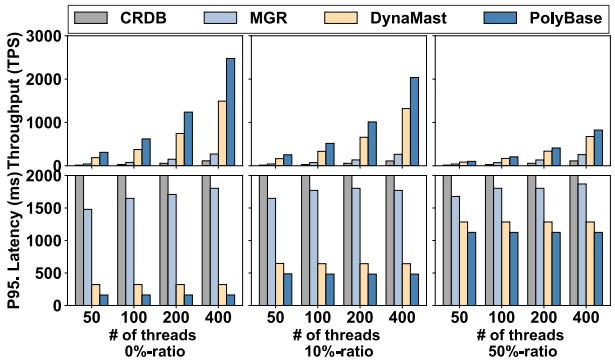

Figure 14: YCSB+T performance (CRDB tail latency truncated, up to 5000ms)

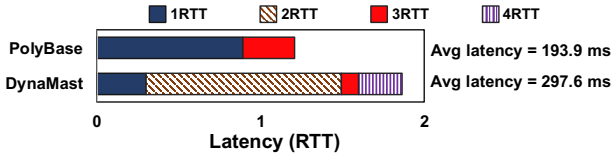

Figure 15: Per-request RTT count breakdown (10% remote ratio, 50 threads)

regions R4-R6, each with 100 client threads. Figure 13 portraits PolyBase's aggregate throughput over time, with R4 shut down at around the 35-second mark, resulting in a 1/3 throughput loss. In about 5 seconds, the underlying Paxos consensus group led by R4 re-elects a new leader (R5), with the latter taking over requests originally routed to R4. Subsequently, PolyBase's throughput rebounds rapidly, albeit stabilizing at a lower overall level, as cross-region request forwarding results in higher per-transaction latency. As discussed in Section 4.2, all the rows colored to R4 will now be considered as having the R5 color (with cLabel updated upon access).

6.4 Benchmark Performance

YCSB+T. With PolyBase's advantages showcased in the previous set of experiments, we now examine its behavior in unfriendly settings, starting with a set of YCSB+T tests with varying ratios of transactions that require remote data access, conducted within the intercontinental region group. Here each client transaction contains 10 interactive updates, issued to its nearest region. Simulating regional locality, each client has an exclusive data access range. In addition, certain data ranges are shared among all clients to emulate random and dynamic data traffic across regions. We manipulate such traffic by setting the ratio of these inter-region transactions within the workload at three levels: 0, 10%, and 50%. Note that the set of data initially assigned to the Paxos group led by the local region is not identical to its client-side exclusive range. More importantly, the "cross-region accesses" evaluated are indeed

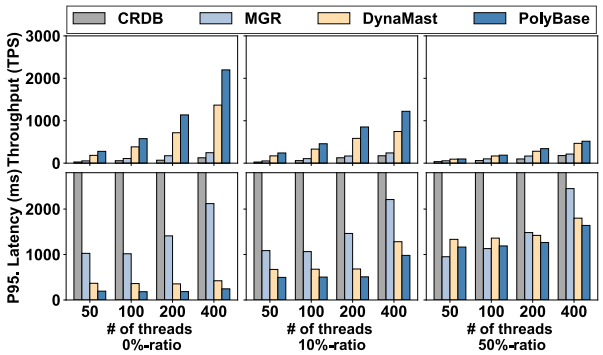

Figure 16: TPC-C performance (CRDB tail latency truncated, up to 7500ms)

Table 2: TPC-C throughput comparison w. Aurora

	R4	R5	R6
Aurora	450.1	2.3	2.3
PolyBase	1590.6	1410.7	1421.2

random, unlike the dynamic yet consistent region switches assessed in the previous tests. They instead present a worst-case scenario for PolyBase, where it finds little reason or benefit from recoloring.

Figure 14 reports results with escalating concurrency (number of threads) and varying cross-region transaction ratios. As expected, PolyBase significantly outperforms other baselines, improving throughput up to 1.2-21.8× and reducing P95 latency by 1.14-30.5×. CRDB and MGR, lacking effective data regional affinity detection, maintain consistent but low performance across all test cases. In contrast, both DynaMast and PolyBase exploit data access locality to promote local executions, though their performance also degrades significantly as the cross-region transaction ratio grows.

DynaMast, due to its design of having a single global "site selector", experiences heavier performance decline, with PolyBase wins by 1.2-1.7× in throughput and 1.1-2.0× in P95 latency. In particular, forcing all requests to go through the global site selector to query the data primary region, DynaMast hurts local transactions, which do not access remote data at all (the 0% cross-region ratio case).

Figure 15 zooms into one test case, showing the breakdown of queries by the number of wide-area RTTs and the average latency for PolyBase and DynaMast. Both systems require one RTT per transaction for global log replication via Paxos. However, PolyBase eliminates wide-area RTTs for 90% of requests through adaptive recoloring, making these requests local. In contrast, DynaMast's centralized primary site management increases RTT counts. Although it uses row-level reassignment, two-thirds of its "local" accesses incur an additional RTT to confirm ownership via a remote global site selector. The latency gap widens further with asynchronous log replication (Figure 2), as the baseline RTT is removed.

TPC-C. Figure 16 presents similar test cases but with TPC-C. The results are also quite similar to that of YCSB+T: compared to DynaMast, the strongest among the baselines, PolyBase brings a throughput improvement of 61%, 64%, and 28% at the three cross-region ratio levels respectively, with corresponding latency reduction.

Global cloud database comparison. We compare PolyBase with the Aurora Global Database (MySQL version), aligning regions (R4-R6) and resources for a fair cross-region behavior comparison. Here,

we focus on their cross-region relative behavior. Both systems use asynchronous replication, with Aurora restricting writes to a single primary region (R4). Table 2 shows Aurora's R5/R6 throughput is under 1% of R4's, while PolyBase achieves more consistent cross-region performance, albeit with higher throughput in R4 due to its lower latency to R5/R6.

Scan workload. Finally, we evaluate scan performance by adapting YCSB-E to create a workload quite unfriendly to PolyBase. In the test, 100 threads in R1 execute scans (each with a length of 5000 records, around 30% non-local), while another 20 threads in R2+R3 issue writes to impose a heavy log application overhead on R1.

When scanning mixed-color data, PolyBase incurs a latency of 11.00 ms, a modest 13% increase over pure local scans (9.68 ms with no concurrent updates). This slowdown, due to waiting for Paxos log application from other regions under snapshot isolation (see Section 3.1), is minor given the average 30 ms cross-region RTT. Overall, PolyBase improves transaction latency globally, as the 13% latency increase for the mixed-color scans is easily offset by the latency savings from write transactions that would have been cross-region.

6.5 Other Optimizations

This series of tests further evaluates the performance impact of PolyBase's internal optimizations.

Global hotness-aware scheduling. Firstly, we evaluate PolyBase hotness-aware recoloring, using YCSB+T with 100 client threads each in R4-R6. Every transaction has 5 queries, with 10% of queries and 40% of transactions access remote-owned data. Unlike tests in Section 6.4, the 10% remote access overlaps with local accesses in the "home region," requiring conservative recoloring. Without hotness-aware recoloring, PolyBase is hyper-sensitive and indiscriminately recolors each row to the region accessing it. Hotness-aware recoloring reduces the median latency by 63%, the P95 tail latency by 27%, and the number of recolor operations by 90%.

Correlation-aware proactive recoloring. We then assess PolyBase "color prefetching" using access correlation mining with a YCSB+T workload following a Zipfian-0.99 distribution, where 80% of accesses target 36% of the data. In this setup, 36% of "hot" rows are paired and accessed by transactions with two queries. The test runs on R4 and R6. Our results indicate that without correlation-aware recoloring, the average latency is approximately 500 ms, around 3 RTTs, which means one recoloring (2 RTTs) plus log replication (1 RTT). Proactive recoloring improves overall throughput by up to 30.8%, roughly removing one RTT with recoloring by association, even when the two regions have 100% overlapping hot datasets.

Deterministic transaction processing. We further check PolyBase's performance under deterministic transactions, comparing with two state-of-the-art geo-replicated deterministic databases, SLOG [40] and Detock [31]. For fairness, we adopted a batching request submission model in PolyBase, where the write sets of transactions can be known ahead of the batch's execution. Our results show that PolyBase outperforms SLOG and Detock by up to 3.35× and 2.05×, respectively, due to their design inefficiencies. SLOG relies on a single global Paxos module, while Detock struggles with scalability limitations and extended wait times due to its heavy, graph-based concurrency control.

7 RELATED WORK

In addition to prior work detailed in Section 2, we discuss other relevant works in the following categories.

Dynamic sharding. Numerous systems utilize data migration to adapt to dynamic workload locality, but either without transaction support (Akkio [4], Tuba [5]) or by analyzing workload traces offline (Schism [12]). Some systems dynamically configure replica location and Paxos roles for partitioned databases (Sharov et al. [42]), while others support row-level primary region changes for geo-distributed setups (L-Store [25], CRDB [48]). However, the CRDB's approach can cause excessive migrations and require client involvement. In contrast, PolyBase is a fully geo-replicated database with transaction support for dynamic workloads, using lightweight metadata updates for reassignment instead of moving large physical data copies across regions.

Consensus protocols. To eliminate the single leader bottleneck in Paxos, multileader and leaderless solutions were proposed. Multileader solutions, like M^2Paxos [38], ZooNet [24] and SwiftPaxos [41], enhance parallelism for non-conflict operations' performance. WPaxos and DPaxos [28] adapt to dynamic workloads by allowing leaders to utilize geographically localized quorums. Leaderless Paxos protocols like EPaxos [46] and Tempo [14] allow replicas to commit non-conflicting operations opportunistically. Mencius [7] employs a round-robin fashion for load balance, suffering long latency for cross-region conflicting operations. Unlike the Paxos protocols, which only handle log replication at the database commit stage, PolyBase dynamically adjusts Paxos group affiliations to fully support transaction processing and reduce latency without modifying the Paxos protocol itself.

Geo-distributed database. Many approaches aim to reduce cross-region latency in geo-distributed transactions. They range from leveraging the optimistic concurrency control (OCC) protocol to speed up transactions [16, 22, 29, 52], combining concurrency control with replication into one RTT [27, 51], overlapping the stages of 2PC and replication during transaction execution [49], to adopt a deterministic transaction model but with limited application scopes [45]. In contrast, PolyBase targets geo-replicated DBs and our recoloring protocol transforms most geo-distributed transactions into local ones with large performance gains, without losing support for general transactions.

8 CONCLUSION

PolyBase builds bridges atop a federation of isolated "Paxos islands". This innovative approach to geo-replicated databases fosters new optimization avenues by *shifting Paxos affiliation across multiple Paxos groups at the finest granularity.* Through flexible and adaptable management, PolyBase significantly improves transaction performance and decouples geo-replication from sharding, thereby revolutionizing the efficiency and effectiveness of geo-replicated databases, when adapting to data affinity changes.

ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their insightful feedback. This work was supported in part by the National Key R&D Program of China under Grant No.2024YFB4505701. Cheng Li is the corresponding author.

REFERENCES

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive dynamic mastering for replicated systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1381–1392.

[2] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2019. WPaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 211–223.

[3] Inc Amazon Web Services. 2023. Amazon Aurora Global Database. https://aws.amazon.com/rds/aurora/global-database/. "[accessed-Sep-2024]".

[4] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the Shards: Managing Datastore Locality at Scale with Akkio.. In *OSDI*. 445–460.

[5] Masoud Saeida Ardekani and Douglas B Terry. 2014. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 367–381.

[6] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, et al. 2021. Log-structured protocols in delos. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 538–552.

[7] Catalonia-Spain Barcelona. 2008. Mencius: building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*.

[8] TPC Benchmark. 2022. TPC-C. http://www.tpc.org/tpcc/. "[accessed-Sep-2024]".

[9] Francesco Calabrese, Mi Diao, Giusy Di Lorenzo, Joseph Ferreira Jr, and Carlo Ratti. 2013. Understanding individual mobility patterns from urban sensing data: A mobile phone trace example. *Transportation research part C: emerging technologies* 26 (2013), 301–313.

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[11] CRDB. 2023. CRDB Follower Reads. https://www.cockroachlabs.com/docs/stable/follower-reads. "[accessed-Sep-2024]".

[12] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. (2010).

[13] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+ T: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 223–230.

[14] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient replication via timestamp stability. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 178–193.

[15] Facebook. 2023. RocksDB. https://github.com/facebook/rocksdb. "[accessed-Sep-2024]".

[16] Hua Fan and Wojciech Golab. 2019. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1471–1484.

[17] Alibaba Group. 2023. PolarDB Global Database Networks. https://www.alibabacloud.com/help/en/polardb-for-mysql/latest/global-database-networks. "[accessed-Sep-2024]".

[18] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[19] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[20] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 145–161.

[21] Alexey Kopytov. 2022. Sysbench. https://github.com/akopytov/sysbench. "[accessed-Sep-2024]".

[22] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 113–126.

[23] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[24] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. 2016. Modular composition of coordination services. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 251–264.

[25] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*. 1659–1674.

[26] Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (1978), 840–842.

[27] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts.. In *OSDI*. 517–532.

[28] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*. 1221–1236.

[29] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1279–1294.

[30] Jelani Nelson and Huacheng Yu. 2022. Optimal bounds for approximate counting. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 119–127.

[31] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[32] Diego Ongaro and John Ousterhout. 2015. The raft consensus algorithm. *Lecture Notes CS* 190 (2015), 2022.

[33] Oracle. 2023. MySQL Group Replication. https://dev.mysql.com/doc/refman/5.7/en/group-replication.html. "[accessed-Sep-2024]".

[34] Oracle. 2023. MySQL Next Key Locking. https://dev.mysql.com/doc/refman/8.0/en/innodb-next-key-locking.html. "[accessed-Sep-2024]".

[35] Oracle. 2023. MySQL Server. https://github.com/mysql/mysql-server. "[accessed-Sep-2024]".

[36] Oracle. 2023. Processing SQL Statements with JDBC. https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html. "[accessed-Sep-2024]".

[37] Andy Palvo. 2023. What Are We Doing With Our Lives? Nobody Cares About Our Research on Concurrency Control. https://www.cs.cmu.edu/~./pavlo/slides/pavlo-keynote-sigmod2017.pdf. "[accessed-Sep-2024]".

[38] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2016. Making fast consensus generally faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 156–167.

[39] Redis. 2024. What is Redis. https://github.com/redis/redis. "[accessed-Sep-2024]".

[40] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1747–1761.

[41] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. 2024. {SwiftPaxos}: Fast {Geo-Replicated} State Machines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 345–369.

[42] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. 2015. Take me to your leader! online optimization of distributed storage configurations. (2015).

[43] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. 2013. F1: A distributed SQL database that scales. (2013).

[44] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[45] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.

[46] Sarah Tollman, Seo Jin Park, and John Ousterhout. 2021. EPaxos Revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 613–632.

[47] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 1–36.

[48] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, et al. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *Proceedings of the 2022 International Conference on Management of Data*. 2312–2325.

[49] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*. 231–243.

[50] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. 2017. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*. 66–79.

[51] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. 2018. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 1–37.

[52] Weixing Zhou, Qi Peng, Zijie Zhang, Yanfeng Zhang, Yang Ren, Sihao Li, Guo Fu, Yulong Cui, Qiang Li, Caiyi Wu, et al. 2023. GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.