Enabling the Next Generation of Multi-Region Applications with CockroachDB

Conference Paper · June 2022

DOI: 10.1145/3514221.3526053

12 authors, including:

Nathan VanBenschoten

3 PUBLICATIONS **113** CITATIONS

SEE PROFILE

Arul Ajmani
Cockroach Labs

2 PUBLICATIONS **2** CITATIONS

SEE PROFILE

Marcus Gartner
Cockroach Labs

1 PUBLICATION **2** CITATIONS

SEE PROFILE

Andrei Matei

2 PUBLICATIONS **113** CITATIONS

SEE PROFILE

Enabling the Next Generation of Multi-Region Applications with CockroachDB

Nathan VanBenschoten
Arul Ajmani, Marcus Gartner
Andrei Matei, Aayush Shah
Cockroach Labs

Irfan Sharif, Alexander Shraer
Adam Storm, Rebecca Taft
Oliver Tan, Andy Woods
Cockroach Labs

Peyton Walters
University of Pennsylvania

ABSTRACT

A database service is required to meet the consistency, performance, and availability goals of modern applications serving a global userbase. Configuring a database deployed across multiple regions such that it fulfils these goals requires significant expertise. In this paper, we describe how CockroachDB makes this easy for developers by providing a high-level declarative syntax that allows expressing data access locality and availability goals through SQL statements. These high-level goals are then mapped to database configuration, replica placement, and data partitioning decisions. We show how all layers of the database, from the SQL Optimizer to Replication, were enhanced to support multi-region workloads. We also describe a new Transaction Management protocol that enables local, strongly consistent reads from any database replica. Finally, the paper includes an extensive evaluation demonstrating that CockroachDB's new declarative SQL syntax for multi-region clusters is easy to use and supports a variety of configuration options with different performance tradeoffs to benefit a variety of workloads. We also show that throughput scales linearly with the number of regions, and the new Transaction Management protocol reduces tail latency by over 10x compared to prior approaches.

CCS CONCEPTS

• **Information systems → Data management systems**; **Parallel and distributed DBMSs**; *Database query processing*; *Database transaction processing*; *Distributed database transactions.*

KEYWORDS

multi-region database, geo-distributed database, global transaction, locality optimized search, survivability

ACM Reference Format:
Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, Andy Woods, and Peyton Walters. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526053

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3526053

1 INTRODUCTION

Today's economy is increasingly dominated by multi-national companies. Their global nature is placing new demands on their technology stack and exposing architectural flaws. The database layer is no exception, and developers are finding that the requirements of global applications cannot be met by traditional databases confined to a single geographic region without compromising on performance, availability, or compliance. High cross-region latencies [16] cause a severe performance penalty when data is served from remote regions. Natural disasters, hardware and software failures, and misconfigurations have caused data center and region-wide failures, and have made it clear that relying on a single data center to store and serve application state is likely to result in service unavailability or data loss. Finally, privacy regulations like GDPR [54] place strict requirements on where data can and cannot reside.

Consequently, companies are turning to multi-region database technologies. Ensuring low latency, high availability, and compliance with regulations using most multi-region commercial offerings, however, is extremely challenging. The challenges stem from the fact that these offerings do not provide useful abstractions that make these concepts easy to reason about and simple to deploy. Instead, they require database administrators and application developers to become experts in multi-region database concepts and tuning. For example, deploying a geo-distributed database typically requires reasoning about the performance and availability implications of data placement and configuration decisions, both during normal operation and in failure modes. Developers also often need to modify their applications in sophisticated ways to efficiently use a geo-distributed database. First, if the database is not region-aware, developers have to include this awareness in their application, or else suffer cross-region latencies on every query. Second, some vendors only support a limited form of transactions [30, 61] or lower consistency levels [44], forcing developers to find workarounds and handle data anomalies at the application level [62].

In this work, we introduce new multi-region abstractions that are built into CockroachDB (abbrev. CRDB) as first-class citizens:

- **Regions** – geographic regions where data should be placed (e.g., close to concentrations of active clients and/or based on domiciling regulations). Each `REGION` contains one or more `ZONE`s (i.e., availability zones), which in turn contain nodes in the cluster. Each table may use any subset of the regions.
- **Table locality** – the expected access pattern for a table, denoting whether rows in the table will be accessed primarily from a single region (`REGIONAL`) or from all (`GLOBAL`). This controls which portions of the geographically distributed database are optimized for access from a given region while maintaining the abstraction of a single logical database.

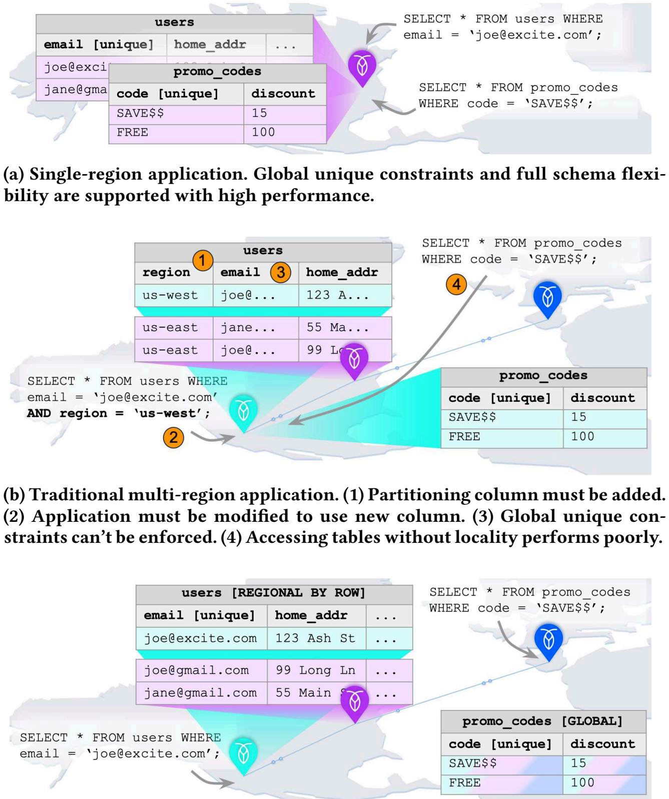
- **Survivability goal** – the types of failures that can occur without rendering the data unavailable (we currently support surviving either a ZONE or full REGION failure).

These abstractions are supported in CRDB with a new SQL syntax that integrates seamlessly with existing SQL DDL statements, making it easier for developers to build global applications. Additionally, because these concepts are first-class citizens, CRDB can leverage them to optimize performance while meeting availability requirements. For example, locality awareness enables the optimizer to efficiently support global uniqueness constraints in geo-partitioned REGIONAL tables and plan queries that avoid visiting remote regions. CRDB also utilizes different transaction management protocols that are optimized for different table access patterns, including a novel protocol for GLOBAL tables that supports low-latency consistent reads from all regions. To use a multi-region CRDB cluster, developers need not change their application but only select the appropriate regions, table localities, and survivability goals for their database. Power-users can further hand-tune configurations beyond the ones we currently support out-of-the-box.

In summary, the contributions of this paper are:

- New SQL syntax that dramatically simplifies multi-region database deployments by distilling a vast design space of possible configurations into selection of regions, survivability goals, and table locality. Sections 2 and 3 describe the SQL syntax and resulting database configuration decisions, respectively.
- Innovations in the SQL Query optimizer such as support for global uniqueness constraints and efficient queries on geo-partitioned data (Section 4).
- A detailed description of our implementation of serializable read-only transactions on historical data that operate locally on any replica's state (Section 5).
- A novel global transaction protocol that enables serializable transactions that observe the latest written data for each key from *any* replica with local latency (Section 6).
- We evaluate CRDB with two industry standard benchmarks (TPC-C and YCSB) that have been modified to support a multi-region workload. We show that no DML changes and only minimal DDL changes are needed, and the resulting performance exceeds that of previous approaches (Section 7).

1.1 A Motivating Example



users		
email [unique]	**home_addr**	**...**
joe@exci		
jane@gma		

promo_codes	
code [unique]	**discount**
SAVE$$	15
FREE	100

```
SELECT * FROM users WHERE
email = 'joe@excite.com';
```

```
SELECT * FROM promo_codes
WHERE code = 'SAVE$$';
```

(a) Single-region application. Global unique constraints and full schema flexibility are supported with high performance.

users		
region	**email**	**home_addr**
us-west	joe@...	123 A...
us-east	jane...	55 Ma...
us-east	joe@...	99 Lo

promo_codes	
code [unique]	**discount**
SAVE$$	15
FREE	100

```
SELECT * FROM promo_codes
WHERE code = 'SAVE$$';
```

```
SELECT * FROM users WHERE
email = 'joe@excite.com'
AND region = 'us-west';
```


(b) Traditional multi-region application. (1) Partitioning column must be added. (2) Application must be modified to use new column. (3) Global unique constraints can't be enforced. (4) Accessing tables without locality performs poorly.

users [REGIONAL BY ROW]		
email [unique]	**home_addr**	**...**
joe@excite.com	123 Ash St	...
joe@gmail.com	99 Long Ln	...
jane@gmail.com	55 Main S	...

promo_codes [GLOBAL]	
code [unique]	**discount**
SAVE$$	15
FREE	100

```
SELECT * FROM promo_codes
WHERE code = 'SAVE$$';
```

```
SELECT * FROM users WHERE
email = 'joe@excite.com';
```

(c) Multi-region application with CockroachDB. Tables designated as REGIONAL or GLOBAL. No other changes from single-region required.

Figure 1: Adapting an application to be multi-region

To make the challenges addressed by this paper concrete, consider a ride-sharing application from a fictional company called movr. Fig. 1a shows two tables from movr's database schema. Fig. 1b shows some of the challenges associated with converting them to multi-region using a traditional DBMS, as the company expands its operation within the US and internationally. Sharding can allow for low-latency access and data domiciling support for the users table, but the schema must be modified to add a partitioning column since no natural partitioning column exists in this case. The application logic and DML must also be modified to use this new column. Furthermore, the database can no longer enforce the uniqueness of email addresses without compromising on performance and compliance[1]. Moreover, while partitioning is a viable (but problematic) option for users, it does not make sense for the promo_codes table, which has no locality of access. With traditional approaches, there is no way to perform low-latency reads of the promo_codes table from all regions while also guaranteeing strong consistency. Finally, depending on the chosen replication strategy, the database could lose data and/or availability if a region suffers an outage.

Fig. 1c shows that these problems are addressed by CRDB. movr can retain the performance, flexible schema design, and operational simplicity of their single-node deployment by simply selecting appropriate table localities.

2 ABSTRACTIONS AND DECLARATIVE SQL

Multi-region capabilities have historically required developers and operators to control underlying primitives like replica placement imperatively. CRDB has extended SQL to allow users to declaratively set database regions, survival goals, and table localities. This section describes these abstractions and the corresponding SQL.

2.1 Region and Zone Management

Conceptually, a multi-region cluster is any cluster with nodes in two or more geographic regions, where each region has one or more zones. In practice, regions and zones are simply strings assigned to each node at startup with the locality command line flag:

[1] Most databases enforce uniqueness constraints with unique indexes. However, partitioning columns must be part of the index key. Therefore, enforcing a unique constraint that does not include the partitioning columns requires forgoing partitioning and losing the associated performance and domiciling benefits.

```
cockroach start \
  --locality=region=us-east-1,zone=us-east-1b # ...
```

The cluster regions are the union of all node regions, and a region's zones are the union of all node zones in the region. Nodes can be added or removed from the cluster at any time, so the regions and zones in a cluster are dynamic. This information is persisted in CRDB and can be retrieved via the SHOW REGIONS command.

A single multi-region CRDB cluster can have several databases, each using a different subset of the cluster regions. To create a multi-region database, users need only choose a PRIMARY region and optionally specify additional regions. For example:

```
CREATE DATABASE movr PRIMARY REGION "us-east1" REGIONS "us-west1",
    "europe-west1";
ALTER DATABASE movr ADD REGION "australia-southeast1";
ALTER DATABASE movr DROP REGION "us-west1";
```

Database regions are maintained in a special ENUM SQL data type called crdb_internal_region, updated whenever regions are added or removed from the database. It serves as the source of truth for other CRDB components to know which regions are available. All regions in CRDB can host leaseholder (i.e., primary) replicas. A PRIMARY region merely serves as the default region for data placement when an alternative region has not been specified.

2.2 Survivability goals

By default, CRDB guarantees ZONE survivability, provided the database has nodes in three or more zones. This ensures base-level fault-tolerance with minimal impact on read and write latency. CRDB additionally offers REGION survivability that ensures availability for reads and writes, even if an entire region goes down. This comes at a cost: write latency is increased by at least the round-trip time to the nearest region. Read performance is unaffected.

Survivability goals are set as follows:

```
ALTER DATABASE movr SURVIVE REGION FAILURE;
ALTER DATABASE movr SURVIVE ZONE FAILURE;
```

2.3 Table Locality Configuration

Every table in a multi-region database has a *table locality* setting, which is REGIONAL BY TABLE, REGIONAL BY ROW, or GLOBAL. This is specified with straightforward SQL:

```
CREATE TABLE west_coast_users ( ... ) LOCALITY REGIONAL BY TABLE
    IN "us-west1";
CREATE TABLE users ( ... ) LOCALITY REGIONAL BY ROW;
ALTER TABLE promo_codes SET LOCALITY GLOBAL;
```

Rows in REGIONAL tables are optimized for low-latency reads and writes from a "home" region (configured at either the table or row level), while rows in GLOBAL tables are optimized for low-latency reads from all regions, at the expense of slower writes.

2.3.1 Regional by Table.

This locality represents the case where all rows in the table are primarily accessed from the same home region, and therefore there is no need for partitioning across regions (data may still be split across nodes within the same region). REGIONAL BY TABLE in the PRIMARY region is the default locality for all tables in a multi-region database if not otherwise specified.

2.3.2 Regional by Row.

In tables with REGIONAL BY ROW locality, individual rows are optimized for access from different regions. This setting divides a table and all of its indexes into partitions, with each partition optimized for access from a different region. That region is specified at the row level in an ENUM column of type crdb_internal_region (see Section 2.1), which constrains its possible values to the set of configured database regions.

Automatic Partitioning. By default, the partitioning column is a system-provided hidden column called crdb_region, which defaults to the region in which an INSERT request originated. Users can also manually update the column (as a hidden column it is invisible to SELECT * queries but is accessible by name).

It is also possible to change the value of this column automatically (called *automatic rehoming*) based on the origin of UPDATEs modifying the row. In effect, rows are re-partitioned to the regions in which they are most recently written to. This is disabled by default since it could lead to thrashing for some workloads. In the future, we plan to make the feature adaptive in order to minimize rehoming, e.g., by using cost-based modeling [3, 6, 7, 57].

Although the crdb_region column is created automatically, it is equivalent to a normal SQL column that simply uses a default value computed from a built-in function provided by CRDB:

```
ALTER TABLE users ADD COLUMN crdb_region crdb_internal_region
    NOT VISIBLE NOT NULL DEFAULT gateway_region();
```

If automatic rehoming is enabled, the column is created with an additional ON UPDATE rehome_row() clause.

Computed Partitioning. Automatic partitioning allows migrating applications to multiple regions without modifying them to be aware of regions. However, some applications may already include the concept of data locality or have a logical partitioning column. These applications can still use REGIONAL BY ROW, e.g., to take advantage of automatic zone configurations for partition placement, but can define crdb_region as a computed column based on existing column(s). For example:

```
crdb_region crdb_internal_region AS (CASE WHEN state = 'CA'
    THEN 'us-west1' ELSE 'us-east1' END) STORED
```

This allows CRDB to perform queries within a single region, ensuring predictable performance, whenever the determinant column(s) (state in this case) are specified in the WHERE clause.

Users can further customize the partitioning column to have any name, constraint, and default value of their choosing, as long as it has type crdb_internal_region. For example, a user might add a foreign key constraint with an ON UPDATE CASCADE clause to ensure rows in a child table stay collocated with their parent.

2.3.3 Global.

Tables with GLOBAL locality optimize for low-latency, strongly consistent reads from every region, at the expense of increased write latencies (Section 6). They are useful for read-mostly data that cannot be partitioned by locality. A common use-case is rarely updated reference data that needs to be read from all regions.

When strongly consistent reads are not required, stale reads on REGIONAL tables (Section 5.3) also provide region-local latencies, without increasing write latencies. However, stale reads without cross-region coordination are not always possible. For example, a read-write transaction cannot use such reads, because it must ensure consistency between its reads and its writes to enforce serializable isolation. Similarly, foreign key validation requires strongly consistent reads of the parent table when the child is updated. GLOBAL tables are essential for good performance in these cases. In particular, users of REGIONAL BY ROW tables expect region-local latency, but a transaction writing to a REGIONAL BY ROW table and reading other tables is only guaranteed to be local if the other tables are GLOBAL. As a result, a common pattern is to have a REGIONAL BY ROW facts table referencing multiple GLOBAL dimension tables.

2.4 Schema changes

Whenever a new multi-region table is created or converted from one locality type to another or a multi-region database is altered, an online schema change is initiated. As described in [60, Section 5.4], CRDB performs schema changes with no downtime.

2.4.1 Adding/Dropping regions to/from a database. Adding or dropping a region is equivalent to adding or removing a value in the `crdb_internal_region` ENUM. Dropping a region involves added complexity to validate that no row in a REGIONAL BY ROW table has its region value set to that region.[2] During validation, the value of the region being dropped is marked as READ ONLY on the ENUM, meaning no query can write that value. By marking the region value as READ ONLY, validation can occur without disrupting foreground traffic. If validation succeeds, the region is successfully removed. If it fails, the operation is rolled back. This mechanism ensures all-or-nothing semantics for dropping regions.

2.4.2 Altering table localities. Altering to a REGIONAL BY TABLE or GLOBAL table simply implies a metadata operation followed by a zone configuration change (see Section 3.2). Altering to a REGIONAL BY ROW table additionally requires prefixing each index with the hidden region column. This operation is implemented by building a new index with the new column set, and once it is backfilled, swapping it with the old. As with other schema changes in CRDB, this process can be completed while the table is online.

2.5 A note on usability

An important feature of CRDB's multi-region abstractions, in addition to making complex concepts easier to reason about, is that they prevent users from mistakenly using anti-patterns by combining lower level building blocks in such a way that they will perform poorly. For example, queries over partitioned tables will likely perform poorly if secondary indexes are not partitioned in the same way as the primary index. This partitioning is done automatically in REGIONAL BY ROW tables, so users cannot make this error.

Today, users could still get poor performance if they select regions far from most clients or choose the wrong table locality for their workload. In future work, we hope to make these features even easier to use by letting the database automatically detect optimal regions and table localities based on the workload.

3 PLACEMENT CONFIGURATION

The high-level concepts and SQL described in the previous section specify policies governing data placement across the cluster. These policies are enforced over *Ranges* using lower level building blocks called *zone configurations* that existed in older versions of CRDB. This section introduces these primitives and explains how they are used to support survivability goals and table localities.

3.1 Background: Ranges and replicas

Data in CRDB is logically stored in a monolithic, ordered key-value store. The keyspace is divided into contiguous Ranges, each replicated using a separate Raft [49] group. Read leases, implemented on top of Raft, allow avoiding consensus round-trips for reads. Raft leaders are usually also *leaseholders* for the Range. Since leaseholders can serve reads locally, a client's proximity to a leaseholder dictates read latencies for that Range (with exceptions for GLOBAL tables (Section 6) and stale reads over REGIONAL tables (Section 5.3)). Writes, on the other hand, have to be acknowledged by a quorum of voting replicas and hence voter placement affects write latencies. Non-voting replicas (see Section 5.2) do not affect write latency.

3.2 Background: Zone configurations

Users can specify placement constraints on individual schema objects (databases, tables, and indexes) through zone configurations [37]. Listing 1 shows some recently introduced control knobs for the number and placement of voting and non-voting replicas.

```
// The difference between num_replicas and num_voters
// determines the number of non-voting replicas.
num_voters = <int>
num_replicas = <int>

// constraints applies to both voting and non-voting
// replicas. It fixes a replica count per-region,
// allowing the remainder to be placed freely.
// voter_constraints is similar but for voters only.
constraints = {
    +region=<string>: <int>,
    +region=<string>: <int>,
    +region=<string>: <int>,
    ...
}
voter_constraints = {+region=<string>: <int>, ... }

// lease_preferences pins the leaseholder to a specific
// region, allowing for consistent reads from within.
lease_preferences = [[+region=<string>]]
```

Listing 1: Zone configuration fields

CRDB guarantees that replicas will be spread across independent failure domains (i.e. localities) while satisfying constraints. It also tries to maximize the number of localities targeted; candidates are assigned a diversity score such that nodes that do not share localities with already placed replicas are ranked higher.

Zone configurations grant users fine-grained control over their data. The syntax is burdensome, however, when translating higher level requirements (e.g., "the table must survive whole region failures", or "the database must be able to serve stale reads locally in any of its regions") into low-level configuration primitives.

3.3 Automatic zone configurations

Table localities described in Section 2 are automatically translated into zone configurations that dictate data placement. REGIONAL BY TABLE and GLOBAL tables are assigned one zone configuration per table, and REGIONAL BY ROW tables have one zone configuration per partition (i.e., per region). The specific configuration depends on the home region and survivability goal.

3.3.1 Home region. We define the *home region* of a table or partition to be the region where all leaseholders of its Ranges are placed.

[2] Because the region column serves as a partitioning key for REGIONAL BY ROW tables, this validation is inexpensive and does not require scanning every row.

For GLOBAL tables, this is the PRIMARY region of the database, while for REGIONAL BY TABLE and REGIONAL BY ROW, it matches the region of the table or row, respectively. The home region informs leaseholder and voter placement; the number of voters in the home region varies based on the survivability goal.

3.3.2 Zone survivability. Databases configured to survive zone failures are automatically set up to have 3 voting replicas for every Range, all in the home region. Within the region, CRDB ensures that replicas are spread across availability zones. Since all voting replicas are constrained to the home region, achieving consensus on writes does not require crossing region boundaries.

To support low-latency reads from other regions, one non-voting replica is placed in each non-home region. A database with N regions configured with ZONE survivability will have 3 voting replicas and $(N - 1)$ non-voting replicas.

3.3.3 Region survivability. Databases can only be configured to survive a region failure if they contain at least 3 regions. With REGION survivability, we use 5 voters with 2 in the home region. This ensures that the home region has two possible candidates for the leaseholder, which allows for fast reads from within the home region with minimal disruption even if one node fails. A database with N regions configured with REGION survivability will have $max(2 + (N - 1), num\_voters)$ replicas with at least 1 replica in each region to ensure stale reads can be served from all regions.

3.3.4 Placement Restricted. ZONE survival databases can optionally be configured using the PLACEMENT RESTRICTED modifier to support data domiciling requirements, such as those required by GDPR [54]. When configured, no replicas (voting or non-voting) belonging to REGIONAL BY TABLE or REGIONAL BY ROW tables are placed outside the home region. As a result, reads (stale or otherwise) cannot be served locally from regions other than the home region. PLACEMENT RESTRICTED does not affect GLOBAL tables and cannot be configured in conjunction with REGION survivability. The lack of REGION survivability with PLACEMENT RESTRICTED is the main restriction around its use, and can only be overcome by direct user modification of the underlying zone configurations.

4 LOCALITY-AWARE SQL OPTIMIZATION

Effective use of REGIONAL BY ROW tables requires an awareness of regions during query planning to avoid cross-region latencies. This section describes how the SQL optimizer accounts for locality during planning and also supports enforcing global unique constraints.

4.1 Enforcing Unique Constraints

Nearly all databases rely on indexes to enforce unique constraints instead of performing expensive full table scans. Therefore, partitioned databases can usually only enforce uniqueness at the partition level. This may be unacceptable for an application; in the movr example (see Fig. 1), they relied on the database enforcing global uniqueness of email. Forgoing partitioning, even if only for the unique column, is also undesirable since it would hamper performance and might not be compatible with domiciling requirements.

In REGIONAL BY ROW tables, indexes are implicitly partitioned by region, but CRDB can enforce globally unique constraints that do not include the partitioning column. This is achieved with constraint checks that run after an INSERT or UPDATE statement as part of the same transaction. To perform the checks, the optimizer uses an efficient join algorithm executing one point lookup in the partitioned unique index for each region containing data. To avoid incurring cross-region latencies, the optimizer omits these checks whenever it is safe to do so. Users can also help in several ways:

1. Use a UUID type for the unique column and let the system generate values with DEFAULT gen_random_uuid(). Since the probability of UUID collisions with this function is negligible, no uniqueness checks are performed by default (they can still be enabled with a cluster setting).
2. Explicitly include crdb_region in the uniqueness constraint definition. This is the best approach if an application only requires uniqueness for a column per region. For example, UNIQUE (crdb_region, col) creates an explicitly partitioned unique index, guaranteeing that col is unique per region.
3. Define crdb_region as a computed column dependent on the unique column(s). In this way, the relevant unique columns become part of the partitioning scheme, and hence uniqueness within a partition implies global uniqueness [51].

Although enforcing global uniqueness can increase the latency of some INSERTs and UPDATEs, it allows CRDB to maintain the integrity of global UNIQUE constraints while keeping all data for a given row in a single region. As we describe next, it also enables querying a unique index with region-local latency.

4.2 Locality Optimized Search

Locality Optimized Search (LOS) is an optimization that is possible when a user is searching for a row that is known to be unique, but its specific location is unknown. For example, SELECT * FROM users WHERE email = 'some-email' does not specify the region where 'some-email' is located, but it is guaranteed to return at most one row since email is known to be unique. CRDB takes advantage of this fact by searching for the row in the local region first. If the row is found, there is no need to fan out to remote regions, since no more rows will be returned. Assuming data is generally accessed from the same region where it was originally inserted (or later rehomed to), this strategy can result in low latency for many queries, including both SELECTs and UPDATEs.

LOS can be generalized to any finite number of rows as long as the maximum number is known. This is useful when there is a LIMIT clause or the WHERE condition uses an IN expression with a unique column rather than an equality expression. LOS can also be used for joins in which rows from the left side of the join are used to look up into a partitioned index on the right side. If the maximum number of results for each lookup is known, the join may be able to avoid visiting remote nodes.

In upcoming releases, we plan to further improve the locality awareness of the optimizer and make better cost-based decisions about when to apply these optimizations. For example, we may be able to make use of foreign-key relationships between REGIONAL BY ROW tables and GLOBAL tables to infer a query's target region and avoid visiting other regions [23]. To better inform the optimizer's cost model, we plan to use the measured latency between regions.

To our knowledge, CRDB is the only DBMS available today that natively supports global UNIQUE constraints over partitioned tables that exclude the partitioning column(s). Thus, it is also the only DBMS that can leverage these constraints for query optimization.

5 LOW-LATENCY STALE READS

To support low-latency reads from all regions, we enhanced CRDB's replication layer. Building on the already-supported follower reads functionality (Section 5.1), we added support for non-voting replicas (Section 5.2) and new forms of stale reads (Section 5.3).

5.1 Background: Follower reads

As reviewed in Section 3.1, the leaseholder of a Range is the only replica allowed to serve up-to-date reads and writes. Non-leaseholder replicas can serve read-only queries on a sufficiently old MVCC snapshot. These operations, called *follower reads*, were previously introduced in [60, Section 3.5]; this section expands on them.

Follower reads provide two benefits. First, they reduce latency for reads of data from geographical locations that are distant from that data's leaseholder but near one of its follower replicas. Second, they balance read traffic across the replicas. Follower reads are used for explicitly requested stale reads (Section 5.3), for long-running transactions, and for *global transactions* (Section 6).

CRDB is a serializable timestamp-based MVCC system, so a read with timestamp T needs to reflect all overlapping writes with timestamps $T' \leq T$. Therefore, a non-leaseholder replica (i.e. a *follower*) can perform a read at a given timestamp T iff:

(1) No future writes can invalidate the read retroactively. The follower needs a guarantee that no new writes will be committed to the Range at MVCC timestamps $T' \leq T$.
(2) It has all the data necessary to serve the read. The follower needs to have applied the prefix of the Raft log that can contain writes at MVCC timestamps $T' \leq T$.

CRDB transactions are assigned a read timestamp and a write timestamp (a provisional commit timestamp) when they start. A transaction's read timestamp identifies the MVCC snapshot that the transaction will read. The provisional commit timestamp dictates the MVCC timestamps of values written by the transaction, thus controlling which other reading transactions will observe those values. A transaction can run for an arbitrarily long time, so its provisional commit timestamp can get arbitrarily old unless the transaction encounters conflicts with other readers, writers, or with closed timestamps (see below). Non-conflicting transactions can commit out of timestamp order. Thus, to ensure condition 1) above, a mechanism is needed to prevent a (follower) read at timestamp T from being invalidated by a future write at a timestamp $T' \leq T$.

5.1.1 Closed timestamps. The two conditions are ensured through the *closed timestamps* mechanism. A closed timestamp is a promise made by the leaseholder that it will not accept new writes at or below that MVCC timestamp (thus "closing" it). These promises are serialized into the Range's replication stream by piggy-backing onto Raft commands. When a follower applies a command carrying a closed timestamp T, it knows that there will not be further commands writing at or below T. Going forward, the follower can start serving follower reads for timestamps $\leq T$.

Closed timestamps are tracked independently by each Range. By default, leaseholders close timestamps that are 3 seconds old. This is recent enough to facilitate follower reads with minimal staleness but not so recent so as to interact with most read-write transactions.

Since new writes below a closed timestamp are not allowed, long-running read-write transactions (i.e. transactions that have been running for long enough that their provisional commit timestamp has been closed by some Ranges they are writing to by the time those writes are served by their respective leaseholder) are forced to increase their provisional commit timestamp, which necessitates a *Read Refresh* ([60, Section 3.4]) on commit. However, provisional writes that have already been evaluated by a Range before that Range closes a timestamp are unaffected by the closing, even if their provisional timestamps are below the closing timestamp. This allows long-running transaction to commit without validating prior writes, but this also means that a closed timestamp does not guarantee that no intent value (and thus also exclusive lock) exists at lower MVCC timestamps. As a result, while reading at a timestamp below the closed timestamp, a follower might run into an exclusive lock. If this happens, the read blocks while it is redirected to the leaseholder to engage in conflict resolution. As such, the condition for a read r to be served on a follower is that r's timestamp T_r is $\leq T_{closed}$ and is $< T_{intent}$ for any intent stored on a key read by r.

5.2 Non-voting replicas

The ability to serve low-latency reads from follower replicas in a Range provides a strong motivation to spread the replicas as wide as possible. However, spreading replicas across many distant regions has a cost in terms of consensus latency, as a majority of the replicas are required to vote on each write to the Range.

To decouple read and write latency, CRDB recently introduced the concept of *non-voting replicas*. These replicas receive Raft log entries (and thus also closed timestamps) and can serve follower reads. They do not, however, vote in consensus decisions and hence do not impact write latency. For tables in multi-region databases, CRDB places a voting or non-voting replica in every region so that clients in all regions can benefit from low-latency follower reads.

5.3 Stale reads

A long-running transaction leverages follower reads when its timestamp becomes older than the closed timestamps of ranges it reads from. CRDB also lets clients directly invoke stale, read-only transactions. Such transactions come in two forms, configured through a special `AS OF SYSTEM TIME` query modifier: exact staleness and bounded staleness. Stale reads operate on a stale MVCC snapshot, corresponding to the read timestamp. The snapshot reflects a prefix of updates and is transactionally-consistent (includes all of a transaction's writes or none of them). Conceptually, these transactions are ordered (w.r.t. other transactions) based on their timestamp. Unlike normal transactions, stale reads do not use uncertainty intervals (see Section 6.1) and thus their timestamp cannot change.

5.3.1 Background: Exact staleness reads. Exact staleness reads were introduced previously in [60, Section 3.5]. They accept a static, user-specified read timestamp. The transaction will read an MVCC snapshot corresponding to that exact timestamp. For example:

```
SELECT * FROM t AS OF SYSTEM TIME '2021-01-02 03:04:05'
SELECT * FROM t AS OF SYSTEM TIME '-30s'
```

If the requested timestamp is old enough, i.e., below the Range's closed timestamp and below the timestamp of any conflicting intents, the read can be served from a follower replica. Otherwise, the query is redirected to the leaseholder. In the future, we might explore an adaptive approach where we attempt to wait locally before communicating with the leaseholder, especially across regions.

5.3.2 Bounded staleness reads. Bounded staleness reads are a feature introduced recently. They use a dynamic, system-determined timestamp, subject to a user-provided staleness bound. Compared to exact staleness reads, the flexibility to choose the timestamp dynamically increases the chance that the query is served by a nearby replica without blocking and with minimal staleness. As a result, bounded staleness reads can improve read availability and provide more reliable latency. For example:

```
SELECT * FROM t AS OF SYSTEM TIME
    with_min_timestamp('2021-01-02 03:04:05')
SELECT * FROM t AS OF SYSTEM TIME with_max_staleness('30s')
```

In exchange for this dynamism, bounded staleness reads are marginally more expensive than exact staleness reads. The implementation of bounded staleness reads involves an extra timestamp negotiation phase, where the system is given a read set and determines the highest timestamp at which these keys can be served by nearby replicas without blocking. As described in Section 5.1.1, intents can exist below the closed timestamp, so this negotiation involves taking the minimum of the closed timestamps and the timestamps of any conflicting intents across the touched Ranges. If this calculation yields a timestamp that is below the query's staleness bound, the transaction either is routed to the leaseholder, using the staleness bound as a read timestamp, or returns an error (depending on a user-specified parameter).

6 GLOBAL TRANSACTIONS

`GLOBAL` tables are meant for read-heavy data replicated across multiple regions, with little or no locality of access. To minimize cross-region communication and coordination, `GLOBAL` tables leverage time delays rather than communication to resolve conflicts between readers and any concurrent writers, and enable strongly-consistent low-latency reads that can be served locally by any replica.

To achieve this, `GLOBAL` tables rely on a novel transaction management protocol we call *global transactions.* Intuitively, these transactions "write into the future" by scheduling their writes to take effect at a future timestamp, as well as generate future-time closed timestamps (see Section 5.1.1). This scheduled time is chosen such that by the time it becomes "present-time", the transaction has likely released its locks, replication has propagated the updated data, and present hybrid logical clock (HLC) time is already closed on all replicas. This allows any replica, not just the leaseholder, to serve present-time reads locally using a regular transaction timestamp. Such strongly-consistent reads can execute as part of read-only or read-write transactions, and do not usually block on writers' locks. The price for these globally fast consistent reads is that transactions that write to `GLOBAL` tables have increased commit latency.

6.1 Background: Uncertainty Intervals

CRDB guarantees serializability for transactions and linearizability for reads and writes at the level of a single key. Linearizability ensures that operations appear to take place in some total order consistent with their real-time order. In other words, a read operation r, invoked after a write w (on the same key) completes, observes the value written by w or newer. To achieve this, CRDB relies on loose clock synchronization and the concept of an *uncertainty interval* — a time window following a read's timestamp within which the reading transaction cannot make real-time ordering guarantees. The duration of uncertainty intervals is configured as the maximum tolerated clock skew between any two nodes, `max_clock_offset`. When reading from a leaseholder, a reader that encounters a provisional or committed write to the same key within its uncertainty interval is forced to ratchet up its timestamp and perform an *uncertainty refresh* – checking whether the values previously read by the transaction remain unchanged at the newer timestamp. If the values have changed, the reader must restart; in any case, the upper bound of the uncertainty interval does not change.

Uncertainty intervals ensure that the relative order of timestamps used by conflicting transactions that touch the same keys respects real-time order. Leaseholders use these timestamps to enforce serializability by blocking reads on the completion of writes to the same key with a timestamp equal to or lower than the read. Leaseholders also advance the timestamp of writes above the timestamp of any previously served reads or refreshes on the same key, preventing writes from invalidating a read's result after it completes. More details can be found in [60, Section 3.3].

6.2 Future-time transactions

Normal transactions in CRDB start with a timestamp assigned from the transaction coordinator's HLC. As the transaction proceeds this timestamp may be ratcheted up, but never exceeds present time by more than `max_clock_offset`. To serve strongly-consistent reads from non-leaseholder replicas, CRDB now also supports future-time transactions. To our knowledge, it is the first DBMS to do so.

Future-time writes are initially invisible to present-time readers. To preserve linearizability (and *read your writes*) the coordinator delays completion of a write operation (i.e., its acknowledgement to the client) until its HLC advances beyond the transaction's commit timestamp. At that point, no other clock in the system lags the commit timestamp by more than the maximum tolerated clock skew, hence every new read is guaranteed to observe the write through the uncertainly interval mechanism described above. This period of time, called *commit wait*, is a variation of a similarly-named stage present in Google Spanner [28, Section 4.1.3]. Unlike in Spanner, the transaction coordinator does not wait for all other clocks to advance beyond the commit timestamp, only for its local clock to do so. CRDB performs this wait concurrently with releasing locks, instead of holding locks for the duration of the commit wait as in Spanner's case. This is key to minimizing the amount of time a lock can be observed by a reader and cause it to block.

Because present-time is closed on `GLOBAL` tables, in the absence of conflicting writes, present-time reads can be served immediately by any replica. In cases of contention on the same key, however, the uncertainty interval rules apply: a read operation r encountering a write w (to the same key) with a higher timestamp but within r's uncertainty interval must observe the written value by bumping its read timestamp to w's timestamp and performing an uncertainty refresh. However, unlike with present-time writes, the existence of a future-time write w does not guarantee that all other clocks in the system are within `max_clock_offset` from w's timestamp. As a result, if the system were to allow r to observe the value written by w and immediately complete, a subsequent read r' performed on a node with a slower clock may fail to observe w in violation of linearizability (in any total order of operations, w appears before r since r returns its written value, and r appears before r' because of

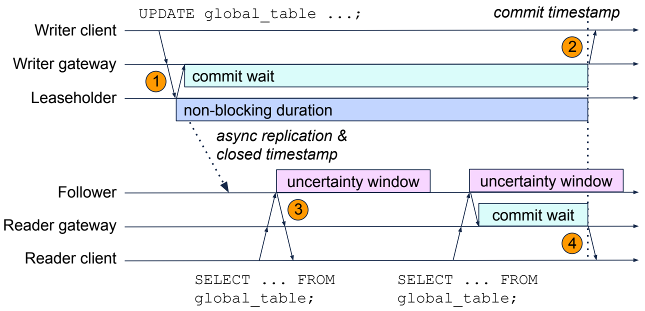


Figure 2: A global transaction (top) and two consistent (non-stale) follower reads (bottom). Time advances from left to right.

real-time order, but r' fails to observe w). The solution is similar to that of writes – if w was written with a future timestamp, r must not only perform an uncertainty refresh using w's timestamp, but must also commit wait before completing, until the local HLC of r's transaction coordinator advances beyond w's commit timestamp. This ensures that when r completes, all observed values are within the uncertainty interval of every node in the system, and any newly invoked read is also guaranteed to observe them.

The delay length for a reader of a `GLOBAL` table encountering conflicting writes depends on whether the writes are committed. If a writing transaction has run for long enough that its locks are still held at the local replica by the time the writes enter the reader's uncertainty window, then the reader blocks on its locks. However, if the writing transaction has already committed and its locks have been removed (the expected common case), the reader is delayed by at most `max_clock_offset` due to the commit wait described above. Using modern clock synchronization techniques [1, 31, 40], `max_clock_offset` can be driven well below cross-region network latency. In contrast, approaches to negotiate the atomic visibility of writes and to provide linearizable reads that are locking-based [20, 60], leasing-based [14, 47], or invalidation-based [34, 35] use communication to coordinate between reads and writes, leading to high read tail latency in the presence of read/write contention.

Fig. 2 depicts the typical flow of a global transaction and its interaction with present-time follower reads. A writing client (on the top) communicates with a transaction coordinator (gateway) which, in turn, communicates with the leaseholder of the relevant Range (1). The write is assigned a future MVCC timestamp and replicates to all replicas. The commit is only acknowledged to the client after commit wait, i.e., when the write's timestamp has become "current" w.r.t. the coordinator's local clock (2). A reading client (on the bottom) performs two reads, in two different transactions. They are both served by a nearby follower replica. The first read runs quickly and doesn't see the recently-written value because its timestamp is below the write timestamp (3). The timestamp of the second read is also lower than the write's, but this time the write falls within the reader's uncertainty window and forces the reader to observe the value. The read bumps its timestamp (which now becomes a future timestamp), performs an uncertainty restart and then commit waits until the timestamp becomes current at its coordinator (4).

6.2.1 Closing timestamps in the future. To enable strongly-consistent present-time reads from any follower, the leaseholder must close time far enough in the future that when the closed timestamp notification propagates to all follower replicas, the timestamp should still be ahead of present time. Consequently, this propagation latency is factored into the chosen closed timestamp, and can be estimated as the sum of Raft consensus latency, L_{raft}, and Raft full replication latency, $L_{replicate}$. The former accounts for the time it takes a Raft group to vote and achieve consensus on a new log entry, typically 1 RTT to the nearest quorum of voting replicas from the leaseholder. This RTT depends on the *survivability goal* and is typically in the range of $2-5ms$ for `ZONE` survival and $20-30ms$ for `REGION` survival. $L_{replicate}$ accounts for the time it takes a committed log entry to reach all members of the Raft group, and is roughly equivalent to the one-way delay between the leaseholder of a Range and its furthest follower. In a multi-region cluster, this is typically $100-125ms$.

In order to serve strongly-consistent reads and commit wait only when a conflicting write is observed within a read's uncertainty interval, all timestamps within the interval should be closed; this ensures that no new writes can appear within the interval. Hence, the size of uncertainty intervals must also be factored in. Added together, a leaseholder must close time at least $L_{raft} + L_{replicate} +$ `max_clock_offset` in the future. Note that this duration directly impacts a writer's potential commit wait time, but does not affect commit wait duration for readers, which is always capped at `max_clock_offset`, the size of the reader's uncertainty interval.

While this provides a good estimate, replication and processing delays occasionally occur. If a read's uncertainty interval is not fully closed, the read is redirected to the leaseholder. We intend to make this policy adaptive, so that the read could choose to wait for a sufficiently large closed timestamp to reach the local replica.

Finally, we explain how CRDB assigns a timestamp to global transactions. Initially, the timestamp is assigned by the transaction coordinator. Each Range in CRDB maintains a closed timestamp target, calculated based on the estimate described above if it is part of a `GLOBAL` table. When a write is sent to the leaseholder of a Range, the transaction's timestamp is advanced immediately past the closed timestamp target of the Range. The target is then attached to the write's log entry as the next closed timestamp. The adjusted timestamp is passed back to the transaction coordinator. If multiple Ranges are involved, the final commit timestamp will be the maximum returned timestamp, across all Ranges. If the commit timestamp is bumped during the transaction's lifetime, global transactions use the usual CRDB mechanism of refreshing their read sets. Conflicts between global transactions are also handled using the regular CRDB blocking mechanisms.

6.2.2 Write and Read Availability. Committing global transactions requires a quorum of voting replicas to be available. Strongly-consistent read availability depends on the replica serving the read being in regular communication with the leaseholder, which in turn must be connected to a quorum of voting replicas to publish closed timestamps. Partitioned replicas may still serve stale reads.

6.2.3 Behavior under clock skew. The single-key linearizability property of global transactions relies on clocks in the cluster being synchronized within `max_clock_offset`, the size of transaction uncertainty intervals. If a node's clock is slow enough, the timestamp of a previously-committed write could be outside the uncertainty interval of a read transaction coordinated by this node, allowing for a stale read. This is possible with any transaction in CRDB (see

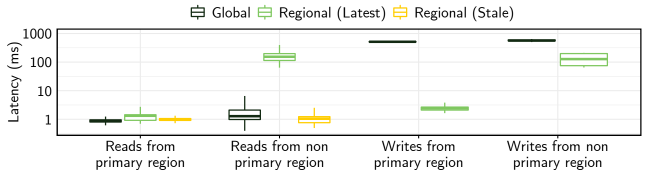


Figure 3: Transaction latency for REGIONAL and GLOBAL tables.

	UE	UW	EW	AN	AS
us-east1	-	63	87	155	198
us-west1		-	132	90	156
europe-west2			-	222	274
asia-northeast1				-	113
australia-southeast1					-

Table 1: Inter-region round-trip times in milliseconds [11].

[60, section 4.3]). Similarly to other kinds of transactions, isolation does not rely on clock synchronization, and therefore CRDB's serializability guarantees are not impacted by clock skew.

7 EVALUATION

This section evaluates the multi-region capabilities of CRDB. We first quantify the tradeoffs between REGIONAL and GLOBAL tables. Then we perform a deeper dive into the performance of REGIONAL BY ROW and GLOBAL tables, as most of the novel technical contributions are in support of those table types. We also test the scalability of the system with increasing numbers of regions, and evaluate the ease of use of the new SQL constructs. All experiments are run on Google Cloud Platform (GCP) [32] and use ZONE survivability.

7.1 Tradeoffs between REGIONAL and GLOBAL

To quantify the performance tradeoffs for REGIONAL and GLOBAL tables, we consider the following types of transactions:

(1) *Global*: fresh reads and writes in a GLOBAL table
(2) *Regional (Latest)*: fresh reads and writes in a REGIONAL table
(3) *Regional (Stale)*: reads in a REGIONAL table w/ bounded-staleness

We use max_clock_offset $250ms$, the default value currently used in CRDB Dedicated [22]. The REGIONAL table uses locality REGIONAL BY TABLE IN PRIMARY REGION.

7.1.1 The Workload. We run a CRDB cluster with nodes located in 5 regions. Each region hosts 3 CRDB nodes and 10 clients, which are all run on GCP n2-standard-4 instances. The round-trip times between regions are summarized in Table 1. We use the YCSB-A [27] benchmark with 1:1 ratio of reads to writes. Each client performs single-key reads and writes with keys drawn from a Zipf distribution. All five regions are added to the database and us-east is designated as PRIMARY, which holds leaseholders for the REGIONAL BY TABLE and GLOBAL tables. Each table is populated with 100k keys. Finally, each client sends 50k queries to a collocated CRDB node in a closed loop, for a total of 2.5 million requests per experiment.

7.1.2 Results. Fig. 3 shows the latency of each transaction type when requests originate from the PRIMARY region, where all leaseholders are located, as well as when requests originate from a non-PRIMARY region. Boxes represent the interquartile range (IQR), and whiskers are length $1.5 * IQR$. As can be seen, GLOBAL tables enable fast reads ($< 3ms$) from anywhere, at the expense of slower writes ($500 - 600ms$). REGIONAL tables support fast reads and writes ($< 3ms$) from the leaseholder's local region, at the expense of slower remote reads and writes ($100 - 200ms$). If stale reads are acceptable, remote reads of REGIONAL tables can be served in $< 3ms$.

7.2 Performance of REGIONAL BY ROW

To evaluate REGIONAL BY ROW tables, we use nine n1-standard-4 GCP instances deployed across three regions (us-east1, europe-west2, and asia-northeast1), running YCSB-B (95% reads, 5% updates) or YCSB-D (95% reads, 5% inserts) with a uniform key distribution for 10 minutes. We configure clients with a "locality of access" value, corresponding to the percentage of operations accessing rows that were originally homed in the client's local region.

In other systems [19, 39, 46, 50, 60, 61] (including legacy CRDB), optimizing tables with locality of access entails manual partitioning – an inflexible scheme potentially requiring both schema and application modifications. Since the manual partitioning method uses the primary key to derive the partition, it results in predictable performance and can serve as our baseline (*Baseline*).

7.2.1 Region-Aware Optimizations. We evaluate the utility of locality optimized search (LOS) and auto-rehoming for REGIONAL BY ROW tables with three variants:

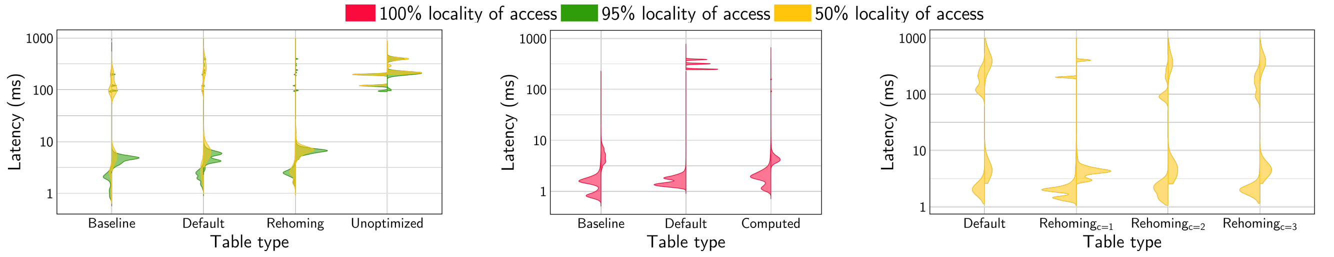
(1) *Unoptimized*: without LOS or auto-rehoming
(2) *Default*: with LOS but no auto-rehoming
(3) *Rehoming*: with both LOS and auto-rehoming

Fig. 4a captures a YCSB-B workload with 95% and 50% locality of access, with clients accessing a disjoint set of keys. YCSB-B only performs updates to non-key columns, so there's no need for uniqueness checks. Since *Unoptimized* does not use LOS, it fans out to all regions on every operation (read or write), exhibiting high latencies (150-200ms). *Default* maintains local latencies for both reads and writes, using LOS to avoid region hops until necessary. It is only slightly slower than *Baseline* which can skip the local search step for remote accesses. The uncontended access lets *Rehoming* re-home all remote rows into the local region, effectively staying in the local latency regime.

7.2.2 Uniqueness constraint checks. Section 4.1 describes how uniqueness constraint checks can be omitted if crdb_region is computed from the unique columns. This helps avoid an additional region hop for INSERTs or UPDATEs to the primary key. We demonstrate this in Fig. 4b running YCSB-D per region with 100% access locality. *Computed* computes crdb_region from the primary key (making it part of of the partitioning scheme), *Default* defaults crdb_region to the region where the INSERT originated, and *Baseline* is a manually partitioned table. We observe local latency INSERTs for *Computed* since it avoids uniqueness constraint checks, unlike *Default* (the three spikes correspond to the three pairwise inter-region RTTs). *Computed* is identical to *Baseline* but with better ergonomics (no schema/application changes needed) – an ideal option when the region can be inferred from existing columns.

7.2.3 Performance with contention. To evaluate automatic rehoming under contention, we run YCSB-B from all regions with 50% locality of access with all remote accesses targeting a shared range of keys. We vary the number of contending clients ($c = \{1, 2, 3\}$), and compare against *Default* where data is not re-homed.

Fig. 4c shows the results. When uncontended ($Rehoming_{c=1}$), we see a single local latency band for reads and writes as all remote data is re-homed to the client's region. With increased contention



(a) Locality optimized search and automatic rehoming with different access localities.

(b) Impact of uniqueness constraint checks and computed regions.

(c) Latency profiles of auto-rehoming under contention

Figure 4: Violin plots [48] for the latency distr. of reads (SELECT) and writes (UPDATE in (a) and (c), INSERT in (b)). The left and right half of each plot corresponds to reads and writes respectively; lower and upper halves to local and remote accesses.

(*Rehoming*$_{c=\{2,3\}}$), the remote data is less likely to be homed in the local region, causing thrashing. At the limit we approach *Default*, where remote data accesses always cross a region boundary.

7.3 Performance of GLOBAL Tables

To evaluate the performance of GLOBAL tables, we use the same workload described in Section 7.1.1, but we also test deployments of CRDB configured with different values for max_clock_offset (the maximum tolerated HLC clock skew between nodes) in order to explore its effect on the latency of global transactions. We test the following values of max_clock_offset:

(1) *250ms*: CRDB Dedicated's current default (See Section 7.1)
(2) *50ms*: CRDB Dedicated's potential default in the future
(3) *10ms*: possible with custom hardware like Spanner's [28]

7.3.1 Baselines. Our primary baseline is *Duplicate Indexes* [38], CRDB's previous approach for low-latency consistent reads from all regions. This method creates a separate secondary index per region containing every column. One replica of each index is pinned to its designated region as the leaseholder, and reads in each region are served using the local index. This approach is similar to the one used by Megastore [14] and Quorum Leases [47].

We also include the *Regional (Latest)* and *Regional (Stale)* experiments from Section 7.1 as additional baselines for reference.

7.3.2 Results. The cumulative distribution of latencies is presented in Fig. 5. Results are separate for reads and writes, and each result is presented twice, first with a focus on the full latency distribution and second with a focus on tail latency.

Read performance. Below the 90%-ile, read latencies are low ($< 3ms$) for all configurations with the exception of *Regional (Latest)*, for which all reads must be routed to the leaseholder, so 80% of reads are remote and we see a step in latency at each quintile (one per region). Low latencies are expected for the other configurations, as the skewed distribution of the workload enables most reads to avoid contention, thus allowing replicas to serve reads locally.

In the tail, read latencies diverge. The *Regional (Stale)* configuration provides reliable tail latency ($< 5ms$) because stale reads are served locally. Reads on GLOBAL tables incur a penalty as they begin to observe read-write contention and require a commit wait. With bigger max_clock_offset, reads have bigger uncertainty intervals, so a bigger fraction observe writes in their uncertainty intervals and must commit wait. The duration of the wait can also increase since it is bounded by max_clock_offset. Reads on duplicate indexes also incur a penalty when read-write contention occurs. In this case, contention materializes as reads waiting on conflicting write transactions to complete their atomic commit protocol across regions. This reliance of duplicate indexes on WAN communication to guarantee consistency leads to unbounded tail read latency. In contrast, GLOBAL tables rely on clock synchronization for consistency, and therefore read latency is bounded by max_clock_offset.

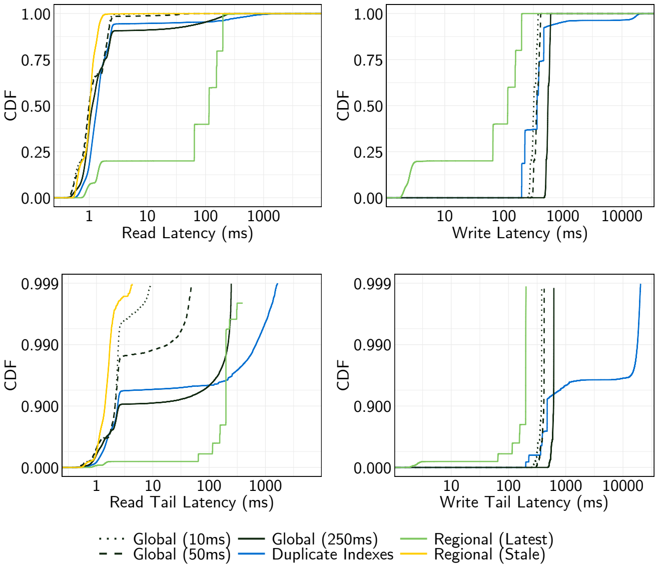


Figure 5: CDFs of reads and write latencies with different schema configurations, with stale reads as a baseline. Global tables are shown with three settings for max_clock_offset.

Write performance. Write latency varies significantly between configurations. REGIONAL performs the best, with a step in latency at each quintile, and the first 20% benefiting from region-local latency. Write latency for GLOBAL tables is higher ($250 - 600ms$, depending on max_clock_offset), as writes perform commit wait.

With duplicate indexes, all writes are routed to the primary index leaseholder, then fanned out to the secondary indexes in each region. In the common case (below the 90%-ile), these multiple WAN hops lead to a similar write latency to that of GLOBAL tables ($200 - 500ms$). In the tail, however, latency for writes to duplicate indexes spikes to more than 10 seconds. This is because transactions must wait for the slowest index write in the furthest remote

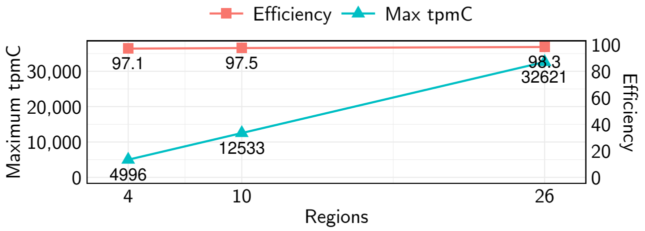


Figure 6: Scalability of multi-region TPC-C

DDL statements required for multi-region operations						
	movr		TPC-C		YCSB	
Operation	**Bef.**	**Aft.**	**Bef.**	**Aft.**	**Bef.**	**Aft.**
New multi-region schema	28	12	44	18	5	1
Converting single-region schema	28	14	44	20	5	1
Adding a region	15	1	20	1	2	1
Dropping a region	9	1	11	1	2	1

Table 2: DDL statements needed for multi-region schema operations before (Bef.) and after (Aft.) the new syntax

region to complete. Furthermore, in the presence of write-write contention, contended writes must queue and wait for earlier writes to complete before proceeding. In contrast, GLOBAL write latency is bounded by $L_{raft} + L_{replicate} +$ max_clock_offset (Section 6.2.1) and contending writers are able to commit wait concurrently.

7.4 Scalability

We evaluate the scalability of multi-region CRDB by running TPC-C against a cluster spread across 4, 10 and 26 regions on GCP. Each region uses 3 n1-standard-4 machines and 100 warehouses. We start with 4 regions in North America (a cross-continental configuration), then add 6 in Europe and Asia (a global cluster with only 3 regions in each continent), and finally employ all but 2 of the 28 GCP supported regions, limited only by hardware shortages (a global cluster with the maximum number of regions in each continent).

As shown in Fig. 6 the performance scales linearly as regions are added to the cluster, staying above 97% efficiency (as defined by TPC-C) in all configurations. The TPC-C schema was updated to leverage multi-region abstractions. The items table is configured to use the GLOBAL table locality as its data is never updated after the initial import. The remaining eight tables use the REGIONAL BY ROW locality with region computed from the warehouse ID.

For the 10 region experiment, p50 latencies varied from 27.3*ms* to 37.7*ms* and p90 latencies from 109.1*ms* to 285.2*ms* across regions, showing that requests do not cross regions in the common case (only the 10% of new-order transactions that access remote warehouses do so). We also verified that PLACEMENT DEFAULT (with non-voters in all regions) did not increase latency compared to PLACEMENT RESTRICTED; p50 latencies for the latter varied from $26.2 - 35.7ms$, while p90 latencies varied from $125.8 - 268.4ms$.

7.5 Ease of use

This section shows that it is possible to convert a single-region application to a high-performing multi-region application and make additional configuration changes with minimal effort.

7.5.1 *Converting applications to use multiple regions.*

We convert movr [24], the application used as an example in earlier sections, to a multi-region application across 3 regions. The DDL changes to do so are captured in Table 2. promo_codes maps to the GLOBAL configuration, the rest map to REGIONAL BY ROW. To convert the application to be multi-region, we need 12 DDL statements when creating a fresh schema (1 CREATE DATABASE, 1 for each of the 6 tables with the specific LOCALITY, and 5 for computed columns translating city to a crdb_region enum). Only 2 additional statements are needed to convert a single-region movr application (regions are added using ALTER DATABASE ... ADD REGION).

Doing this manually with earlier CRDB versions required 28 statements (for a new schema or to convert an existing one) to manually specify partitioning, zone configurations, and duplicate indexes for each table, all to achieve the same functionality. Adding or dropping a new region with the new syntax requires only a single statement. Table 2 shows similar results for the TPC-C [25] and YCSB [52] schema. The reduced syntax makes it less error prone, all without requiring any DML changes on the application side.

7.5.2 *User Feedback.*

To gain additional insight, we consulted with users of CRDB who have adopted the new multi-region abstractions. They noted that the abstractions reduced complexity, making operations easier and making it easier to educate engineers on the team. Some had tried to build similar abstractions previously at the application layer, and appreciated that this was no longer needed. We have seen adoption from all company sizes (e.g., growth, commercial, enterprise), industries (e.g., finance, insurance, entertainment, logistics, marketing), and business models (e.g., B2B, B2C).

An example of a real workload that is using these abstractions is a personalized assistant application that uses CRDB to store global IoT device and user data. It has three regions across the US and Asia. Devices stay in their region, and need to write events fast (using REGIONAL BY ROW with ZONE survival). Meanwhile, users move around, and need fast reads everywhere (using GLOBAL tables).

One piece of constructive feedback we have received is the need to support additional data domiciling use cases beyond those supported by PLACEMENT RESTRICTED. We are continuing to refine our abstractions in this area. In future releases we plan to allow users to express more complex failover relationships between regions and link data to multiple eligible regions. This will enable applications requiring data domiciling to take advantage of features such as non-voting replicas and the ability to survive region failures.

7.6 Summary

Section 7.1 confirms that GLOBAL tables are ideal for read-mostly workloads requiring fast reads from anywhere, while REGIONAL tables are best when a workload has high locality of access or stale reads are acceptable. Section 7.2 shows that REGIONAL BY ROW tables perform as well as or better than the baseline partitioned tables on SELECT and UPDATE queries thanks to locality optimized search and auto-rehoming, but may incur cross-region latencies on INSERTs if enforcing globally unique constraints. Section 7.3 demonstrates that GLOBAL tables and the baseline duplicate indexes both support low-latency reads at the expense of slower writes, but tail latency is bounded for GLOBAL tables, with a tighter bound for smaller max_clock_offsets, while tail latency for duplicate indexes is unbounded. Section 7.4 demonstrates that throughput scales linearly as CRDB scales from 4 to 26 regions. Finally, Section 7.5 shows that deploying a multi-region CRDB cluster requires few DDL changes, and describes constructive feedback from users.

8 RELATED WORK

Geo-distributed consistency. The universal trade-offs between consistency, availability, and latency in distributed databases are exacerbated when data is distributed geographically [2]. Some systems [8, 45, 50] optimize for latency by using asynchronous replication strategies with weak consistency models, at the risk of data corruption and security vulnerabilities [62]. Reads from local replicas in Aurora global databases [10] can wait for the replica to catch up to writes from the same session, however, these reads are not strongly consistent because they are not guaranteed to see earlier writes from other sessions [5]. Dynamo [9] and Cosmos [44] allow for strongly consistent reads and writes, but only within the same region. In contrast to the systems mentioned, CRDB provides strongly consistent reads and writes in all regions by default.

To provide low-latency reads from replicas, other systems such as PNUTS [26] support stale reads that are consistently ordered but may not reflect the most up-to-date state of the database. Spanner [21] supports two types of low-latency, stale reads that can sometimes avoid cross-region communication: bounded staleness reads, where reads see values that are not more stale than the given bound, and exact staleness reads, where reads see a version of data at a specific timestamp in the past. Similar to Spanner, CRDB supports both bounded and exact staleness reads. While CRDB guarantees that stale reads observe a consistent prefix of updates, other studies proposed to make probabilistic consistency guarantees, serving an operation locally after a wait period that minimizes inconsistencies with high probability, for example based on the expected rate of new updates [12] or their replication latency [13].

Other systems take advantage of locality of access to avoid cross-region communication during reads and writes. FlightTracker [58] pins data to the region where it is accessed so that quorums are region-local. SLOG [55] utilizes the locality-of-access of data to provide strongly-consistent transactions that avoid cross-region communication, and it dynamically remasters data to different regions as access patterns change over time. CRDB's `REGIONAL BY ROW` tables allow for low-latency, region-local transactions for workloads that have locality-based access patterns.

Both Megastore [14] and CRDB's deprecated *duplicate indexes* [60] provide consistent, low-latency reads from all regions at the cost of higher write latencies and significant write amplification. Similarly, Citus [20] *reference tables* provide fast reads in all regions that are consistent thanks to two-phase commits. Moraru et al. [47] propose Paxos quorum leases which use existing communication patterns in Paxos-based systems to allow a subset of replicas (lease holders) to perform low-latency, strongly consistent local reads. To this end, all lease-holders must be included in every write quorum and any lease-holder failure stalls writes until the lease expires. CRDB's global transactions, which power `GLOBAL` tables, allow local consistent reads from *any* replica, without sacrificing availability. Writes to `GLOBAL` tables succeed when *any* quorum of voting replicas respond, not just a select subset.

Geo-distributed data placement. The placement of data in geo-distributed databases is critical to minimize latency, balance load, and comply with data location regulations. Several automatic data placement strategies have been proposed with these goals in mind [4, 7, 18, 29, 41, 43, 53, 56, 57, 63, 65]. Prior work has also sought to devise data placement strategies that reduce cloud infrastructure costs [42, 64] and adhere to policy constraints [15, 33]. Other systems, including previous versions of CRDB, require users to imperatively configure the placement and type of each replica. The declarative SQL abstractions presented in this paper allow users to describe their multi-region needs, letting the system take care of tedious replica type and placement decisions.

Some systems support user-configured row-level partitioning where a shard key prefixes each index entry and determines the geographic placement of data [19, 39, 46, 50, 61]. Another approach is to deploy multiple instances of a database, each optimized for different access locality, and defer request routing, data sharding and placement, global schema management and other orchestration to upper layers of the stack [17, 59]. Usually, these systems forgo cross-shard transactions, indexes, and global uniqueness constraints, and often require that applications are aware of the details of the partitioning scheme to ensure efficient data placement and query execution. CRDB's `REGIONAL BY ROW` tables support application-defined partitioning schemes, but can also implicitly partition tables and indexes across regions, and automatically place rows where they are accessed. To our knowledge, CRDB is the first database to support global uniqueness constraints on row-level partitioned tables.

9 CONCLUSION AND OUTLOOK

This paper described the multi-region abstractions in CRDB that enable any developer to build a high performance global application. To use CRDB's multi-region abstractions, developers need only specify regions of operation and expected access patterns per table (and optionally, domiciling restrictions and availability requirements under failure). These abstractions are exposed through intuitive SQL syntax, and supported by advances in the query optimizer as well as the replication and transaction layers. We showed that converting a single-region application to multiple regions requires no application changes and minimal DDL changes. The abstractions perform better than prior approaches to multi-region support.

The abstractions described in this paper are just the first step toward truly democratizing multi-region application development. We envision a future in which users only need to provide their schema and queries; the database should infer the regions and access patterns from the workload. Additionally, the cost of operating a multi-region cluster today is prohibitive for some, as it requires statically allocating compute and storage resources in multiple regions. Future solutions must be affordable.

We believe the path forward involves supporting multi-region applications in a serverless database– a pay-as-you-go system in which users need not concern themselves with the number or location of servers needed to serve their workload. We recently introduced a multi-tenant single-region serverless offering [36], and we are actively working to add multi-region support. Multi-tenant multi-region serverless will not only make it possible for users to avoid selecting regions and therefore reduce operational complexity, it will also drastically reduce the cost of operating a multi-region database due to the ability to share infrastructure costs across users. Many challenges remain, though, such as how to track access patterns with minimal overhead, and we encourage the research community to join us in finding solutions. We look forward to working together to enable the future of multi-region applications.

REFERENCES

[1] [n.d.]. chrony. https://chrony.tuxfamily.org/.

[2] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.

[3] 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.

[4] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: Automatic physical design metamorphosis for distributed database systems. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3573–3587.

[5] Steve Abraham. 2020. Building globally distributed MySQL applications using write forwarding in Amazon Aurora Global Database. https://aws.amazon.com/blogs/database/building-globally-distributed-mysql-applications-using-write-forwarding-in-amazon-aurora-global-database/.

[6] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 739–753.

[7] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Habinder Bhogan. 2010. Volley: Automated data placement for geo-distributed cloud services. (2010).

[8] Amazon. [n.d.]. Amazon RDS Read Replicas | Cloud Relational Database | Amazon Web Services. https://aws.amazon.com/rds/features/read-replicas/.

[9] Amazon. [n.d.]. Global Tables: How It Works - Amazon DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/globaltables_HowItWorks.html.

[10] Amazon. [n.d.]. Using Amazon Aurora global databases. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-global-database.html.

[11] ATT Center for Virtualization at Southern Methodist University. 2021. Google Cloud Inter-region latency and throughput. https://datastudio.google.com/u/0/reporting/fc733b10-9744-4a72-a502-92290f608571/page/70YCB.

[12] Xiao Bai, Flavio P. Junqueira, and Adam Silberstein. 2013. Cache Refreshing for Online Social News Feeds. In *Proceedings of the 22nd ACM International Conference on Information Knowledge Management* (San Francisco, California, USA) *(CIKM '13)*. 787–792.

[13] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2012. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proc. VLDB Endow.* 5, 8 (2012), 776–787.

[14] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. (2011).

[15] Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Compliant Geo-distributed Query Processing. (2021).

[16] BigBitBus. 2018. What is your ping, Google Cloud and Amazon AWS? https://www.bigbitbus.com/2018/05/07/What-Is-Your-Ping-AWS-And-Google-Cloud/.

[17] AirBnb Engineering Blog. [n.d.]. How we partitioned Airbnb's main database in two weeks. https://medium.com/airbnb-engineering/how-we-partitioned-airbnb-s-main-database-in-two-weeks-55f7e006ff21.

[18] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2018. Adapting to Access Locality via Live Data Migration in Globally Distributed Datastores. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 3321–3330.

[19] Citus. [n.d.]. Choosing Distribution Column — Citus 10.2 Documentation. https://docs.citusdata.com/en/v10.2/sharding/data_modeling.html.

[20] Citus. [n.d.]. Concepts — Citus 10.2 documentation. https://docs.citusdata.com/en/v10.2/get_started/concepts.html#type-2-reference-tables.

[21] Google Cloud. [n.d.]. Timestamp bounds | Cloud Spanner | Google Cloud. https://cloud.google.com/spanner/docs/timestamp-bounds.

[22] CockroachCloud. [n.d.]. https://www.cockroachlabs.com/product/cockroachcloud.

[23] CockroachDB. [n.d.]. https://github.com/cockroachdb/cockroach/issues/69617.

[24] CockroachDB. [n.d.]. https://github.com/cockroachdb/cockroach/blob/4021c7342f4ebb794f4176e2d3e10b913e5eeb58/pkg/workload/movr/movr.go#L308-L489.

[25] CockroachDB. [n.d.]. https://github.com/cockroachdb/cockroach/blob/4021c7342f4ebb794f4176e2d3e10b913e5eeb58/pkg/workload/tpcc/tpcc.go#L563-L720.

[26] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1277–1288.

[27] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[28] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, Berkeley, CA, USA, 251–264. http://dl.acm.org/citation.cfm?id=2387880.2387905

[29] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 163–174.

[30] DataStax Documentation. [n.d.]. Apache Cassandra Lightweight Transactions. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useInsertLWT.html.

[31] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 81–94.

[32] Google Cloud. 2021. Google Cloud Compute Engine. https://cloud.google.com/compute.

[33] Sudarshan Kadambi, Jianjun Chen, Brian F Cooper, David Lomax, Raghu Ramakrishnan, Adam Silberstein, Erwin Tam, and Hector Garcia-Molina. 2011. Where in the world is my data? *Proceedings of the VLDB Endowment* 4, 11 (2011), 1040–1050.

[34] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 201–217.

[35] 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.

[36] Andy Kimball. 2021. How we built a forever-free serverless SQL database. (October 2021). https://www.cockroachlabs.com/blog/how-we-built-cockroachdb-serverless/

[37] Cockroach Labs. [n.d.]. Configure Replication Zones. https://www.cockroachlabs.com/docs/stable/configure-replication-zones.html.

[38] Cockroach Labs. [n.d.]. Duplicate Indexes Topology. https://www.cockroachlabs.com/docs/v20.2/topology-duplicate-indexes.

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

[40] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. 2020. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 1171–1186.

[41] 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.

[42] Guoxin Liu and Haiying Shen. 2017. Minimum-cost cloud storage service across multiple cloud providers. *IEEE/ACM Transactions on Networking (TON)* 25, 4 (2017), 2498–2513.

[43] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 2018. NashDB: an end-to-end economic method for elastic database fragmentation, replication, and provisioning. In *Proceedings of the 2018 International Conference on Management of Data*. 1253–1267.

[44] Microsoft. 2021. Consistency levels in Azure Cosmos DB | Microsoft Docs. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels#strong-consistency-and-multiple-write-regions.

[45] MongoDB. [n.d.]. Replication — MongoDB Manual. https://docs.mongodb.com/manual/replication/.

[46] MongoDB. [n.d.]. Sharding — MongoDB Manual. https://docs.mongodb.com/manual/sharding/.

[47] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2014. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2671001

[48] National Institute of Standards and Technology DataPlot. 2020. Violin Plot. https://www.itl.nist.gov/div898/software/dataplot/refman1/auxillar/violplot.htm.

[49] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 305–319.

[50] Oracle. [n.d.]. Oracle Sharding. https://www.oracle.com/a/tech/docs/sharding-wp-12c.pdf.

[51] Glenn Norman Paulley. 2001. *Exploiting functional dependence in query optimization.* Citeseer.

[52] Peyton Walters. [n.d.]. https://github.com/pawalt/cockroach/blob/d838a72967b0a1518ccf7933814b6141960658a7/pkg/workload/ycsb/ycsb.go#L239-L298.

[53] Fan Ping, Jeong-Hyon Hwang, XiaoHu Li, Chris McConnell, and Rohini Vabbalareddy. 2011. Wide area placement of data replicas for fast and highly available data access. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*. ACM, 1–8.

[54] General Data Protection Regulation. 2016. Regulation EU 2016/679 of the European Parliament and of the Council of 27 April 2016. *Official Journal of the European Union* (2016).

[55] 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.

[56] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[57] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. 2015. Take me to your leader!: online optimization of distributed storage configurations. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1490–1501.

[58] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. 2020. FlightTracker: Consistency across Read-Optimized Online Stores at Facebook. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 407–423.

[59] Alexander Shraer, Alexandre Aybes, Bryan Davis, Christos Chrysafis, Dave Browning, Eric Krugler, Eric Stone, Harrison Chandler, Jacob Farkas, John Quinn, Jonathan Ruben, Michael Ford, Mike McMahon, Nathan Williams, Nicolas Favre-Felix, Nihar Sharma, Ori Herrnstadt, Paul Seligman, Raghav Pisolkar, Scott Dugas, Scott Gray, Shirley Lu, Sytze Harkema, Valentin Kravtsov, Vanessa Hong, Wan Ling Yih, and Yizuo Tian. 2018. CloudKit: Structured Storage for Mobile Applications. *Proc. VLDB Endow.* 11, 5 (2018), 540–552. https://doi.org/10.1145/3187009.3164138

[60] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[61] Vitess. [n.d.]. The Vitess Docs | Sharding. https://vitess.io/docs/reference/features/sharding/.

[62] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 5–20.

[63] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. 1997. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)* 22, 2 (1997), 255–314.

[64] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. 2013. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 292–308.

[65] Victor Zakhary, Faisal Nawab, Divy Agrawal, and Amr El Abbadi. 2018. Global-Scale Placement of Transactional Data Stores.. In *EDBT*. 385–396.