Enabling the Next Generation of Multi-Region Applications with CockroachDB

Rebecca Taft
Arul Ajmani
Marcus Gartner
Andrei Matei
Cockroach Labs

Aayush Shah
Irfan Sharif
Alexander Shraer
Adam Storm
Cockroach Labs

Oliver Tan
Nathan VanBenschoten
Andy Woods
Cockroach Labs

Peyton Walters
University of Pennsylvania

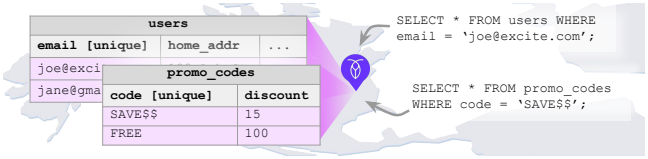
ABSTRACT

A database service is required to meet the consistency, performance, and availability goals of modern applications serving a global user-base. 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 of the new features.

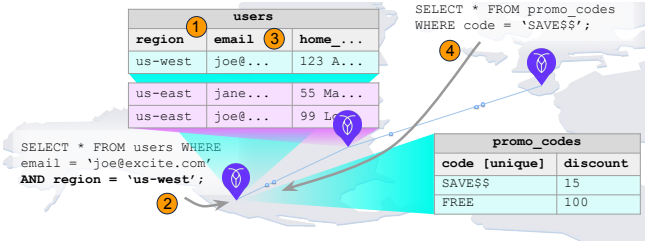
ACM Reference Format:
Rebecca Taft, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Oliver Tan, Nathan VanBenschoten, Andy Woods, and Peyton Walters. 2018. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

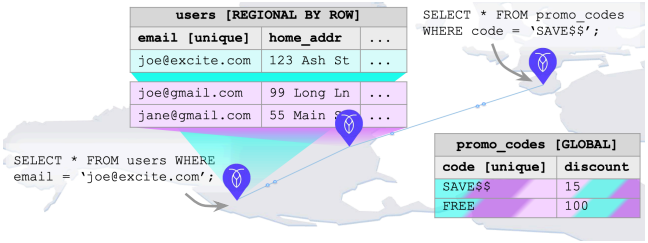
Today's economy is increasingly dominated by multi-national companies that operate around the world. The global nature of these companies is placing new demands on their technology stack and exposing architectural flaws. The database layer is no exception, and developers are finding that the requirements of global applications cannot be met by traditional databases where nodes are confined to a single geographic region without compromising on performance, availability, or compliance. High cross-region latencies [13] cause a severe performance penalty when data is served from remote regions. Natural disasters, electrical grid blackouts, hardware and software failures, and misconfigurations have caused data center and region-wide failures that brought down online services, 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 [48] place strict requirements on where data can and cannot reside.



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



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



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

Figure 1: Converting a single-region application to multi-region

Consequently, companies are turning to multi-region database technologies. Ensuring low latency, high availability, and compliance with regulations using traditional multi-region commercial offerings, however, is extremely challenging. 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. These professionals

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

must select geographic regions in which to deploy databases as well as reason about the performance and availability implications of data placement and configuration decisions, both during normal operation and in the presence of failures. 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, the 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 [26, 54] or lower consistency levels [38], forcing developers to find workarounds and handle data anomalies at the application level [55].

In this work, we introduce new multi-region abstractions: region (geographic area of operation), table locality (expected access pattern), and survivability (level of availability in case of failures). These abstractions are supported in CockroachDB with simple SQL commands, making it easier for developers to build global applications. Additionally, because these concepts are first-class citizens in CockroachDB, we can leverage them to improve database performance. A region-aware database can make data placement and replication decisions that provide high performance while meeting availability requirements. A locality-aware SQL optimizer will choose to access data locally whenever possible. Different transaction protocols are optimized for different table access patterns; awareness of these patterns enables choosing the best protocol.

Example. 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, 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 doesn't 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.

1.1 Multi-region support in CockroachDB

As illustrated above, an application can exhibit multiple distinct workloads and access patterns. These patterns and the desired geo-distribution characteristics of the workload need to be taken into account when modeling application data. With CockroachDB (abbrev. CRDB), an application developer makes the following choices:

- Specify **regions** where data should be placed (e.g., close to concentrations of active clients and/or based on domiciling regulations). `REGION` represents a geographic region that contains one or more `ZONE`s (typically availability zones), which in turn contain nodes in the cluster. Each table is configured to use a subset (potentially all) of the regions.
- Configure **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.
- Choose a **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).

CRDB allows users to make these choices using a new declarative SQL syntax that integrates seamlessly with existing DDL statements in the SQL standard. Based on these selections, CRDB automatically places and configures replicas. It leverages awareness of regions and geo-partitioned data as well as novel locality-aware SQL optimizations to enforce constraints and minimize data access latency. Cross region replication supports low-latency stale reads from all regions. Finally, CRDB selects the appropriate transaction protocol to use based on the table locality. `GLOBAL` tables use a novel *global transaction* protocol that supports low-latency consistent reads from all regions. Together, these innovations make querying data efficient and obviate the need for developers to change their application. In the movr example, it is possible to retain the performance, flexible schema design, and operational simplicity of a single-node deployment by simply selecting appropriate table localities (Fig. 1c). Power-users can further hand-tune their configurations beyond the ones we currently support out-of-the-box.

Modifications to database and table regions or any of the other table characteristics described above can be made by simple declarative commands. Under the hood, the database will automatically reconfigure replicas and move data as needed to achieve the desired configuration. In summary, the contributions of this paper are:

- A new declarative SQL syntax that dramatically simplifies configuration of multi-region database deployments by distilling a vast design space of possible configurations into the 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 are needed and only minimal DDL changes are needed, and the resulting performance exceeds that of previous approaches to multi-region support (Section 7).

[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.

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. Listing 1 shows a sample of the supported commands.

```
-- Show cluster regions.
SHOW REGIONS;

-- Create database and alter database regions.
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";

-- Set survivability goals.
ALTER DATABASE movr SURVIVE REGION FAILURE;
ALTER DATABASE movr SURVIVE ZONE FAILURE;

-- Create tables and alter table locality.
CREATE TABLE east_coast_users ( ... )
  LOCALITY REGIONAL BY TABLE IN PRIMARY REGION;
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;
```

Listing 1: Multi-region SQL syntax

2.1 Region Management

Conceptually, a multi-region cluster is any cluster with nodes in two or more geographic regions. In practice, regions are simply strings (e.g., "us-east1") assigned by the user to each node at startup with the `locality` command line flag (availability zone and other locality attributes are assigned similarly). The cluster regions are the union of all node regions. New nodes can be added and existing nodes can be removed from the cluster at any time, so the regions in a CRDB cluster are dynamic.

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 (see Listing 1).

The `PRIMARY` region does not have more capabilities than other regions, as all regions in CRDB can host leaseholder (i.e., primary) replicas. It merely serves as the default region for data placement when an alternative region has not been specified.

Under the hood, the database regions are maintained in a special `ENUM` SQL data type called `crdb_internal_region`. This type is updated whenever regions are added or removed from the database, and serves as the source of truth for other components of the database to know which regions are available.

2.2 Survivability goals

By default, CRDB automatically guarantees `ZONE` survivability, provided the database has nodes in three or more zones. This ensures base-level survival with minimal impact on read and write latency. However, many companies require stricter survival conditions. As such, CRDB provides users the option of a `REGION` survivability goal (see Listing 1) that adjusts replica placement to ensure the database will remain fully available for reads and writes, even if an entire region goes down. This added survival comes at a cost: write latency will be increased by at least as much as the round-trip time to the nearest region. Read performance will be unaffected.

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` (see Listing 1). 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's 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 originating location of `UPDATE`s modifying the row. This is disabled by default since it could lead to thrashing for some workloads, though we have plans to make the feature adaptive to reduce the possibility.

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 and rehoming allow developers to migrate single-region applications to multiple regions without modifying them to be aware of regions. The optimizer ensures that all constraints are enforced and querying the table is efficient (see Section 4.2). However, some applications may already include the concept of data locality, or may already have a logical partitioning column. These applications can still use `REGIONAL BY ROW` to take advantage of benefits such as automatic zone configurations for partition placement, but can define the `crdb_region` column as a computed column based on the existing logical partitioning column(s) of the application. For example:

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

This approach may be preferable since CRDB can guarantee execution of a given query will be limited to a single region – and thus ensure predictable performance – even if only the determinant column (`state` in this case) is specified in its `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 in the database – at the expense of increased write latencies – by providing a new "global" transaction model. `GLOBAL` tables are therefore useful for read-mostly tables whose rows cannot be partitioned by locality. Reference data that is rarely updated and needs to be read from all regions is a common use case for `GLOBAL` tables.

When strongly consistent reads are not required, then stale reads (see Section 5.3) on `REGIONAL` tables also provide region-local latencies and are generally preferable to a `GLOBAL` table as they do not increase write latencies. However, stale reads are not possible for a subset of transactions. For example, a read-write transaction cannot use stale 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 [53, 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]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.

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're 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 [43] 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 in a multi-region cluster (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 [33]. Listing 2 shows some of the control knobs.

```
// 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 2: 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 don't 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. 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. 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. The `PLACEMENT RESTRICTED` policy does not affect `GLOBAL` tables and cannot be configured in conjunction with `REGION` survivability.

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, and does not require additional checks.
(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 [45].

Although enforcing global uniqueness can increase the latency of some `INSERT`s and `UPDATE`s, 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 the uniqueness of `email` 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 avoid visiting remote nodes and result in low latency for many queries, including both `SELECT`s and `UPDATE`s.

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 [20]. To better inform the optimizer's cost model, we plan to use the measured latency between regions.

5 LOW-LATENCY STALE READS

To support low-latency reads from all regions, we have added several enhancements to the replication layer. Building on the follower reads functionality already supported in CRDB (Section 5.1), we added support for non-voting replicas (Section 5.2) and support for 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 [53, 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 provide a means of balancing read traffic across the replicas.

Being a serializable timestamp-based MVCC system, a read in CRDB with timestamp T needs to reflect all overlapping writes with timestamps $T' \leq T$. So, in order to enable the follower reads functionality, 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. In principle, 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. This is why condition 1) above is not trivial: unless special measures are taken, a (follower) read at timestamp T could be invalidated by a future write at a lower timestamp $T' \leq T$, breaking transaction serializability.

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 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 $\leq T$.

The closed timestamp is tracked at the level of each Range. Thus, different Ranges independently close MVCC timestamps at different offsets from present time. By default, leaseholders close timestamps that trail their wall-clock by 3 seconds. 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 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* ([53, Section 3.4]) on commit.

While the closed timestamp subsystem prevents new writes at or below the current closed timestamp, it does not guarantee no intent value (and thus also exclusive lock) exists at MVCC timestamps below the closed timestamp of their Range (such locks must have been written before their respective timestamp was closed). This means that, 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 scalability and locality from fault-tolerance, CRDB introduced the concept of *non-voting replicas*. These replicas receive Raft log entries (and thus also closed timestamps) and can serve follower reads. They don't, however, vote in consensus decisions and hence don't 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

The lag in publication of closed timestamps means that follower reads operate on a stale MVCC snapshot (a consistent prefix of updates). Stale read-only transactions come in two forms, configured through a special `AS OF SYSTEM TIME` query modifier: exact staleness and bounded staleness.

5.3.1 Exact staleness reads. Exact staleness reads 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'
```

A query reading a stale snapshot is less likely to interact with other queries in the system. Still, such a query might conflict with an exclusive lock held by a long-running transaction, in which case it behaves like a regular query and blocks waiting for the lock to

be released. This is in contrast to bounded staleness reads, which try to avoid blocking on locks.

To be served by a follower replica, the read timestamp of an exact staleness read must be below the Range's closed timestamp and below the timestamp of any conflicting intents.

5.3.2 Bounded staleness reads. Bounded staleness reads 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, compared to other reads. In this phase, 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 closed timestamp among the touched Ranges, and further ratcheting that down below the minimum timestamp of any conflicting intent.

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 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 read timestamp. Such strongly-consistent reads can execute as part of read-only or read-write transactions, and are expected to usually not block on the writers' locks, as we explain below.

6.1 Background: Present-time transactions

CRDB guarantees linearizability for reads and writes at the level of a single key, in addition to providing serializability for transactions. Linearizability ensures that operations appear to take place in some total order consistent with their real-time order. Preserving real-time order means that 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 normally 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 (this check may block on the completion of the write, as we explain below). If the values have changed, the reader must restart; in any case, the upper bound of the uncertainty interval doesn't 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 [53, Section 3.3].

6.2 Future-time transactions

Future-time writes complicate things in that they initially remain invisible to readers that read at present-time. To preserve linearizability (and *read your writes*) the coordinator delays the completion of a write operation (i.e., its acknowledgement to the client) until its hybrid logical clock 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 [25, 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 span of time where a lock can be observed by a reader and cause it to block.

From a reader's perspective, future-time writes are no exception to the uncertainty interval rules: 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 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 hybrid logical

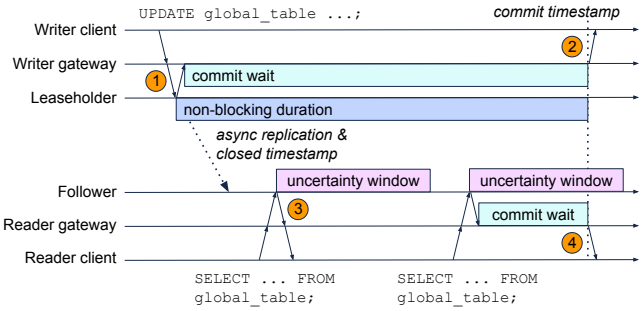


Figure 2: A global transaction (top) and two consistent (non-stale) follower reads (bottom). Time advances from left to right.

clock 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.

Because present-time is closed, in the absence of conflicting writes, present-time reads can be served immediately by any replica. In cases of contention on the same key, if the writing transaction has already committed and the local replica has removed the relevant locks by the time the writes enter the reader's uncertainty window (the expected common case), the reader does not block on locks and instead is delayed by at most `max_clock_offset`. Using modern clock synchronization techniques [1, 27, 36], this can be driven well below cross-region network latency. This is in contrast to techniques that use locking-based approaches [17, 53], leasing-based approaches [11, 41], or invalidation-based approaches [30, 31] to negotiate the atomic visibility of writes and to provide linearizable reads. Each of these use communication to coordinate between reads and writes, leading to high read tail latency in the presence of read/write contention. However, if the writing transaction runs for long enough that the upper-bound of readers' uncertainty intervals reach its write timestamp while its locks are still held at the local replica, then the readers block on its locks.

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 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 in to how far in the future the leaseholder closes MVCC time. This latency is 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 latency depends on the *survivability goal* and is typically in the range of $2-5ms$ for `ZONE` survival and $20-30ms$ for `REGION` survival. The Raft full replication latency accounts for the time it takes a committed log entry to reach all members of the Raft group. This latency 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 transactions'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 coordinating by this node, allowing for a stale read. This is possible with any transaction in CRDB (see

[53, 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. As most of the novel technical contributions are in support of `REGIONAL BY ROW` and `GLOBAL` tables, we focus our efforts on evaluating the performance 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. Due to lack of space, we do not explicitly evaluate the survivability goals, as the ability of CRDB to remain available under large scale failures was demonstrated in prior work [53, Section 6.2]. All experiments are run on Google Cloud Platform (GCP) [28] and use `ZONE` survivability.

7.1 Performance of `REGIONAL BY ROW`

`REGIONAL BY ROW` tables are ideal for data with access affinity and work transparently with existing schemas. Experiments here 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) for 10 minutes in every region with a uniform key distribution. 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 (including legacy CRDB), optimizing tables with locality of access entails manual partitioning – an inflexible scheme potentially requiring both schema and application modifications. It results in predictable performance, however, and can serve as our baseline (*Baseline*).

7.1.1 Region-Aware Optimizations. To evaluate locality optimized search (LOS) and auto-rehoming, we consider three `REGIONAL BY ROW` 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. 3a captures a YCSB-B workload with 100%, 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. We omit *Unoptimized* from the figure – it observes high latencies (150-200ms) by fanning out to all regions for every request since it does not know where the row is located. *Default* maintains local latencies for both reads and writes, using locality optimized search 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.

7.1.2 Uniqueness constraint checks. Section 4.1 describes how uniqueness constraint checks can be omitted if `crdb_region` is computed over the unique columns. This helps avoid an additional region hop for `INSERT`s or `UPDATE`s to the primary key. We demonstrate this in Fig. 3b 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 `INSERT`s for *Computed* since it avoids uniqueness constraint checks, unlike *Default* (the three spikes correspond to the three pairwise inter-region RTTs). It 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.

	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 [10].

7.1.3 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.

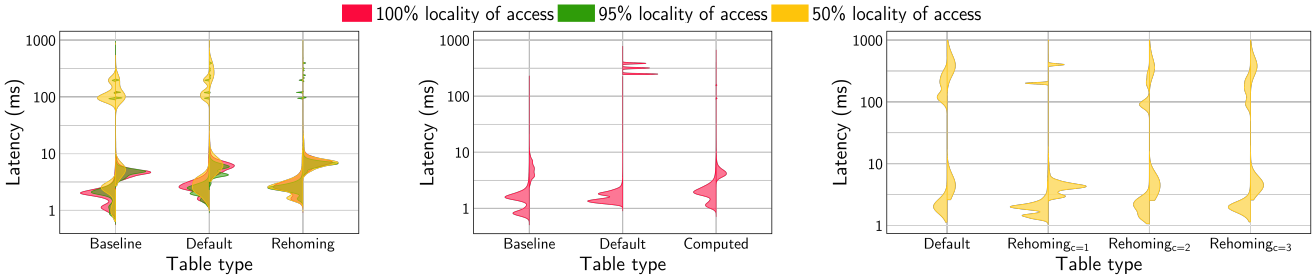
Fig. 3c shows the results of the experiment. When uncontended ($Rehoming_{c=1}$), we observe a single local latency band for reads and writes as all remote data is re-homed to the client's region. With increased contention ($Rehoming_{c=\{2,3\}}$), it is less likely for the remote data to be homed in the local region, resulting in thrashing. At the limit we approach *Default*, where remote data accesses always cross a region boundary.

7.2 Performance of `GLOBAL` Tables

The `GLOBAL` table locality is ideal for tables that must be read with low latency from multiple regions but which can tolerate writes with higher latency. This section evaluates the performance of `GLOBAL` tables on a workload without locality of access.

7.2.1 The Workload. In this experiment, we run a CRDB cluster with nodes located in 5 regions, us-east, us-west, europe-west, asia-northeast, and australia-southeast. Each region hosts 10 clients. CRDB nodes and clients are run in GCP on n2-standard-4 instances. The round-trip times between regions are summarized in Table 1. We use the YCSB-A [24] benchmark with 1:1 ratio of reads to writes. Each client performs single-key reads and writes with keys drawn from a Zipf distribution. The skewed distribution creates hot keys, which is exacerbated by a lack of access locality in clients' workload. All five regions are added to the database and us-east is designated as `PRIMARY`. 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.

7.2.2 Table configurations. To explore the relationship between `max_clock_offset` (the maximum tolerated HLC clock skew between any two nodes in the cluster) and commit wait time in `GLOBAL` tables, we test `GLOBAL` tables on deployments of CRDB configured with different clock skew bounds. First, we test 250*ms*, the default value currently used in CRDB Dedicated [19], we then repeat with 50*ms* (we will likely be able to reduce the maximum clock offset configuration in CRDB Dedicated to 50*ms* within the next year, pending further experimentation and testing), and finally with 10*ms*, a value close to what Google Spanner achieves with custom hardware [25].



(a) Locality optimized search and automatic re-homing with different access localities.

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

(c) Latency profiles of auto-rehoming under contention

Figure 3: Violin plots[42] for the latency distribution of reads and writes. The left and right half of each plot corresponds to reads and writes respectively; the lower and upper halves to local and remote latency regimes.

We compare the performance of the GLOBAL tables against several baselines. The first is the duplicate indexes pattern [34], CRDB's previous approach for low-latency consistent reads from all regions, now superseded by GLOBAL tables. This method creates a separate secondary index per region that contains every column. One replica of each index is pinned to its designated region as the leaseholder. Since the SQL optimizer prefers serving read requests from nearby leaseholders, reads in each region are served using the local index. In exchange for low-latency reads, the pattern incurs high write amplification and write latency, as each index is separately replicated and writes must update all indexes.

We we also compare against two REGIONAL BY TABLE variants in the PRIMARY region – one where read requests return the latest data (i.e., reading from the leaseholder) and another where bounded-staleness reads are used (i.e., reading stale data locally).

The cumulative distribution of latencies from running the workload on all table configurations is presented in Fig. 4. 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. The results show that GLOBAL tables optimize for fast read performance in exchange for slower write performance when compared to REGIONAL tables. Compared to duplicate indexes, they provide similar read and write performance in the common case, while providing more reliable worst-case performance.

7.2.3 Read performance. The duplicate indexes, stale reads, and all three GLOBAL variants provide comparable median read performance. Given the skewed distribution of the workload, reads do not hit contention in the common case. As a result, these variants are all able to serve reads locally, leading to single-digit millisecond response times. The REGIONAL BY TABLE variant performs worse for reads in the common case. All strong reads must be routed to the leaseholder, so 80% of reads are not served locally and we see a step in latency at each quantile (one per region).

In the tail, beyond the 90th percentile, read latency for duplicate indexes, stale reads, and all three GLOBAL variants diverges. Bounded staleness reads provide reliable tail latency because they forgo consistency, allowing them to serve all reads locally. Reads on GLOBAL tables incur some penalty as they begin to observe read-write contention and require a commit wait. Configurations with less precise clock synchronization bounds see the impacts of read-write contention at earlier percentiles because reads have larger

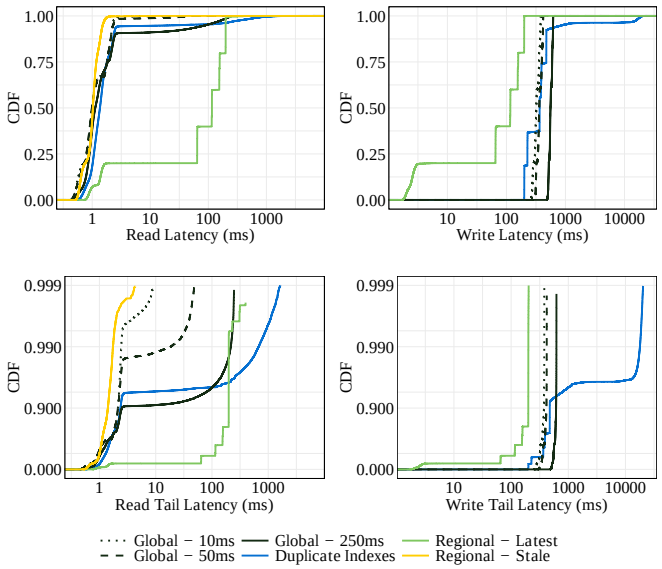


Figure 4: CDFs of reads and write latencies with different schema configurations, with stale reads as a baseline. Global tables are shown with three settings for `max_clock_offset`.

uncertainty intervals, so a larger fraction observe writes in their uncertainty intervals and must commit wait. The impact is also more severe because the maximum commit wait delay of a read on a GLOBAL table is equal to the max_clock_offset. While commit wait leads to an increase in read latency in the tail, the impact of read-write contention has a tight upper bound which is a function of clock synchronization and not of communication latency. Reads on duplicate indexes also incur a penalty when read-write contention occurs. In this case, contention materializes as reads waiting on intents (i.e., locks) written by conflicting write transactions that are running an atomic commit protocol across regions. Reads must wait until conflicting writes complete, leading to the so-called "cloggage problem" [2]. This leads to an earlier and wider tail, with some reads waiting for over one second for a conflicting write to commit. While this effect is demonstrated here through the duplicate indexes pattern, it is present in any technique where communication over WAN is used to coordinate between conflicting reads and

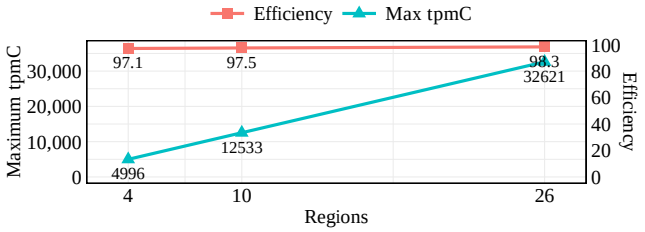


Figure 5: Scalability of multi-region TPC-C

writes to ensure strong consistency (e.g., the experimental results of Megastore [11] and Quorum Leases [41]).

7.2.4 Write performance. All variants differ in their median write latency. The `REGIONAL BY TABLE` variant performs the best, as the latency of a write depends on the latency from the SQL gateway to the leaseholder. As a result, we see a similar pattern to the one we saw for reads, with a step in latency at each quintile, and the first 20% benefiting from region-local latency. The median write latency with both `GLOBAL` and duplicate indexes is higher. All writes to the `GLOBAL` tables must perform a commit wait, which delays them by about $300 - 500ms$, depending on the value of `max_clock_offset`. `max_clock_offset` is factored in to how far in the future writes to `GLOBAL` tables are performed, so it impacts the duration of the commit wait. Meanwhile, with duplicate indexes, all writes to the table must first be routed to the leaseholder of the primary index, after which they are fanned out to the secondary indexes in each region. In the common case, these multiple wide-area network hops lead to a similar write latency to that of `GLOBAL` tables.

However, the write performance on `GLOBAL` tables and duplicate indexes diverges drastically in the tail. Like with reads, the tail latency of writes to a `GLOBAL` table has a tight upper bound. Write latency is a function of clock synchronization and communication latency, but there is no synchronous per-region fan out involved. As a result, the latency of writes to a `GLOBAL` table is slow but reliable. This is true even in the presence of write-write contention, because contending writers are able to commit wait concurrently. Write tail latency with duplicate indexes is much worse. Beyond the 95^{th} percentile, latency spikes and grows to more than 10 seconds. This is because the write must communicate with each of the regions in the database. Even if done in parallel, this fan-out must wait for the slowest index write to complete. Furthermore, in the presence of write-write contention, contended writes must queue behind one another and wait for earlier writes to complete before proceeding.

7.3 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, one per availability zone, and 100 warehouses. We start with 4 regions in North America, then add 6 in Europe and Asia, finally using all but 2 of the 28 GCP supported regions (limited by hardware shortages).

As shown in Fig. 5 the performance (measured in transactions per minute) 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), allowing data pertaining to every region's warehouses to stay local.

For the 10 region experiment, p50 latencies varied from $27.3ms$ to $37.7ms$ and p90 latencies from $109.1ms$ to $285.2ms$ across regions, showing that requests do not cross regions in the common case (only the 10% of new-order transactions accessing remote warehouses require cross-region communication). 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 $125.8 - 268.4ms$.

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

7.4 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.

7.4.1 Converting applications to use multiple regions. We convert movr [21], 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 [22] and YCSB [46] schema. The reduced syntax makes it less error prone, all without requiring any DML statement on the application side.

7.4.2 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 helped reduce 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've 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).

One piece of constructive feedback we've 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.

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 [3]. Some systems [7, 39, 44] optimize for latency by using asynchronous replication strategies with weak consistency models, at the risk of data corruption and security vulnerabilities [55]. Reads from local replicas in Aurora global databases [9] 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 [8] and Cosmos [38] 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 [23] support stale reads that are consistently ordered but may not reflect the most up-to-date state of the database. Spanner [18] 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.

Other systems take advantage of locality of access to avoid cross-region communication during reads and writes. FlightTracker [51] pins data to the region where it is accessed so that quorums are region-local. SLOG [49] 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 [11] and CRDB's deprecated *duplicate indexes* [53] provide consistent, low-latency reads from all regions at the cost of higher write latencies and significant write amplification. Similarly, Citus [17] *reference tables* provide fast reads in all regions that are consistent thanks to two-phase commits. Moraru et al. [41] 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, and do not depend on a select subset of replicas. This results in higher availability and lower tail-latencies.

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, 6, 15, 47, 50, 57]. Prior work has also sought to devise data placement strategies that reduce cloud infrastructure costs [37, 56] and adhere to policy constraints [12, 29]. 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 [16, 35, 40, 44, 54]. Another common approach is to deploy multiple separate instances of a database, each potentially 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 [14, 52]. Usually, these systems forgo cross-shard transactions, indexes, and global uniqueness constraints, and they 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 are designed to 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 declarative 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 DML or application changes, and only a few DDL changes. The abstractions perform better than prior approaches to building multi-region applications.

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 need not specify anything other than their schema and queries; the database should infer the regions and access patterns directly from the workload. There are many challenging problems that must be solved to realize this vision, but 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 [32], 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, 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. [n.d.]. It's Time to Move on from Two Phase Commit. http://dbmsmusings.blogspot.com/2019/01/its-time-to-move-on-from-two-phase.html.

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

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

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

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

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

[10] 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.

[11] 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).

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

[13] 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/.

[14] 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.

[15] 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.

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

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

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

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

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

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

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

[23] 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.

[24] 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.

[25] 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

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

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

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

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

[36] 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.

[37] 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.

[38] 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.

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

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

[41] 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

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

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

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

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

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

[47] 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.

[48] 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).

[49] 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.

[50] 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.

[51] 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.

[52] 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

[53] 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

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

[55] 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.

[56] 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.

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