A Demonstration of Multi-Region CockroachDB

Arul Ajmani
Aayush Shah
Cockroach Labs

Alexander Shraer
Adam Storm
Rebecca Taft
Cockroach Labs

Oliver Tan
Nathan VanBenschoten
Cockroach Labs

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 fulfills 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. CockroachDB also enables many types of queries on the multi-region database to perform as well as they would in a single-region deployment, due to enhancements to the SQL optimizer, transaction, and replication layers. This paper showcases these features with a comprehensive demonstration scenario tracking a ride-sharing company's journey as they expand their application globally.

PVLDB Reference Format:
Arul Ajmani, Aayush Shah, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, and Nathan VanBenschoten. A Demonstration of Multi-Region CockroachDB. PVLDB, 15(12): 3610 - 3613, 2022.
doi:10.14778/3554821.3554856

1 INTRODUCTION

Developers today 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 [1] 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 [4] 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, which mandates a deep understanding of fundamental distributed systems trade-offs. 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 [2, 6] or lower consistency levels [3], forcing developers to find workarounds and handle data anomalies at the application level [7].

This paper demonstrates new multi-region abstractions, first introduced in our previous publication [5], that were recently added into CockroachDB (abbrev. CRDB) as first-class concepts: *region* (geographic area of operation), *survivability* (expected availability in the presence of failures), and *table locality* (expected access pattern). These abstractions are supported in CRDB with simple declarative SQL commands and exposed at the schema level, making it easy for developers to build global applications. The system then leverages these first-class concepts to make data placement and replication decisions that provide high performance while meeting availability and data domiciling requirements. The SQL optimizer is locality-aware and will choose to access data locally whenever possible. An enhanced transaction protocol is optimized for different table access patterns.

1.1 Abstractions

CRDB extended SQL to allow users to declaratively set database regions, survival goals, and table localities. This subsection describes these abstractions (see Section 3 for the corresponding SQL).

1.1.1 Region. A multi-region CRDB cluster is any cluster with nodes in two or more geographic regions, where a region has one or more availability zones. Regions and zones are assigned to each node at process startup with the `locality` command line flag:

```
cockroach start \
  --locality=region=us-east1,zone=us-east1-b # ...
```

A single multi-region cluster can have several databases, each using a subset of the cluster regions. Users choose a `PRIMARY` region and optionally specify additional regions to create a multi-region database. All regions in CRDB can host leaseholder (i.e., primary) replicas. The `PRIMARY` region serves as the default region for data placement when an alternative region has not been specified.

1.1.2 Survivability Goal. By default, CRDB guarantees `ZONE` survivability, provided the cluster is comprised of 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 which ensures availability for reads and writes, even if an entire region goes down. This can be configured at the cost of increased write latency. Read performance remains unaffected.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554856

1.1.3 *Table Locality.*

Every table in a multi-region database has a *table locality* setting, which can be either `REGIONAL BY TABLE` (the default), `REGIONAL BY ROW`, or `GLOBAL`.

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

2 PERFORMANCE OPTIMIZATIONS

In this section, we describe several mechanisms that allow CRDB to efficiently support geo-distributed workloads using locality-awareness and optimizations throughout the stack.

2.1 Globally `UNIQUE` Constraints

Nearly all databases rely on unique indexes to enforce unique constraints on the index key columns. In a partitioned database, the partition key must be part of the index key, so a unique index can only guarantee uniqueness of the other columns at the partition level. Some use-cases, however, require global uniqueness. In such cases, forgoing partitioning for the unique index is undesirable since it would hamper performance and might not be compatible with domiciling requirements. In `REGIONAL BY ROW` tables, all indexes are implicitly partitioned by region, and the CRDB optimizer injects uniqueness checks on `INSERT` or `UPDATE` statements, as part of the same transaction. These checks execute one point lookup of the unique index in each region. To avoid cross-region latencies, the optimizer removes these checks when it is correct to do so.

An optimization that becomes possible with globally unique keys is Locality Optimized Search. When a user is searching for rows that are known to be unique but their location is unknown, we search in the local region first. If the rows are found (or, if enough rows are retrieved to satisfy a `LIMIT` clause), there is no need to fan out to remote regions. 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 `SELECT`s and `UPDATE`s. To our knowledge, CRDB is the first geo-distributed partitioned DBMS to support enforcement of global `UNIQUE` constraints, or leverage them in the query optimizer.

2.2 Reads from Local Replicas

In order to serve reads with low latency from any region, and improve read-scalability, CRDB supports reads from any replica. Each transaction is assigned a read timestamp that determines the MVCC snapshot read by the transaction; in order to serve the read from a local replica, the read timestamp needs to be "closed" at the replica. This means that the replica has seen all updates up to the timestamp and no new updates are possible with a lower timestamp. The leaseholder of a range determines and periodically communicates the latest closed timestamp to its followers. Even though new writes that can affect the read are not allowed, read-write conflicts with in-progress transactions that have a lower commit timestamp and performed writes before their timestamp was closed may still occur; the read can only be served locally if there are no such conflicting writes.

Follower replica reads are useful in several scenarios. When the timestamp of a long-running transaction becomes old enough, its reads are redirected and served by local replicas. An application can also use the `AS OF SYSTEM TIME` clause to specify either a specific timestamp that is likely to be closed (which can be automatically chosen by CRDB with `follower_read_timestamp()`), or a bound on maximum acceptable staleness (e.g. `with_max_staleness('10s')`) which gives CRDB more flexibility to try to pick a timestamp that can be served locally.

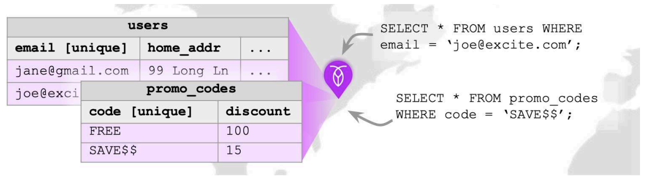
While follower replicas can reduce read latency and aid read-scalability, they can hamper writes since the leaseholder needs to replicate updates to a quorum of replicas. To mitigate this problem, similarly to most Paxos-based systems, CRDB allows increasing the number of followers without increasing the quorum size by distinguishing between voting and non-voting followers.

2.3 Global Transactions

Finally, CRDB supports a novel transaction management protocol that allows local-replica reads to be strongly consistent, even for short-lived transactions. When a table is defined as `GLOBAL`, writes are assigned a future write timestamp and leaseholders close timestamps in the future. Transactions are assigned read timestamps just like before, but can now be served locally since present time is likely to be closed. Writers block until the chosen write timestamp becomes present time, after committing and releasing locks but before returning to their caller. When conflicts occur, because writers are scheduled into the future, readers rarely wait for conflicting writes to commit. Instead, the write's outcome is commonly already determined, so to coordinate the write's visibility, readers need only block for a duration bounded by the tolerated clock skew. These delays, called *commit-wait*, guarantee that a read r starting after a write w commits observes w, which is essential for single-key linearizability provided by CRDB (assuming clock-skew is bounded). Just like with normal transactions, isolation (serializability) does not depend on timing assumptions. This protocol uses time delays, rather than communication, to coordinate between readers and writers. Since writers always block, it trades off write latency, but in return achieves fast and predictable read latencies from any region. This makes `GLOBAL` tables most appropriate for read-heavy data with little or no locality of access.

3 DEMONSTRATION

For our demonstration scenario, we will 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 global uniqueness of email addresses without compromising on performance and data domiciling. Moreover, while partitioning is a viable (but problematic) option for `users`, it does not make sense for the `promo_codes` table,



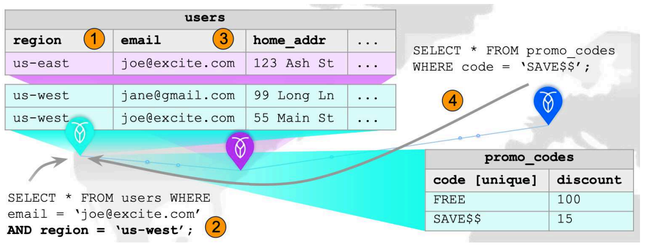
users		
email [unique]	home_addr	...
jane@gmail.com	99 Long Ln	...
joe@exci		

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

```
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.



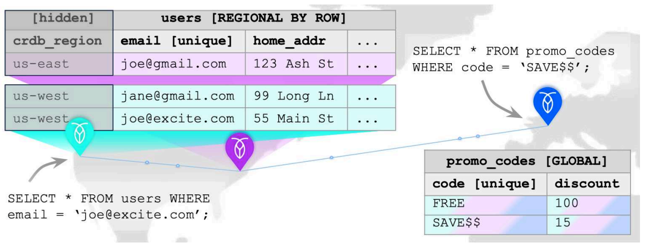
users			
region	email	home_addr	...
us-east	joe@excite.com	123 Ash St	...
us-west	jane@gmail.com	99 Long Ln	...
us-west	joe@excite.com	55 Main St	...

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

```
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.



[hidden]	users [REGIONAL BY ROW]		
crdb_region	email [unique]	home_addr	...
us-east	joe@gmail.com	123 Ash St	...
us-west	jane@gmail.com	99 Long Ln	...
us-west	joe@excite.com	55 Main St	...

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

```
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

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.

3.1 Adapting movr to use multi-region CRDB

Starting from movr's single region deployment, our demonstration will show how to easily expand to multiple regions with CRDB, while retaining the performance, flexible schema design, and operational simplicity of their single-node deployment (see Fig. 1c).

To make the cluster multi-region, we can simply add nodes to the cluster in the new regions, setting their locality flags to indicate their location (see Section 1.1.1). To see the added regions, run:

```
> SHOW REGIONS FROM CLUSTER;
     region    |                 zones
---------------+----------------------------------------
  europe-west1 | {eur-west1-a,eur-west1-b,eur-west1-c}
  us-east1     | {us-east1-a,us-east1-b,us-east1-c}
  us-west1     | {us-west1-a,us-west1-b,us-west1-c}
```

To enable the use of the new multi-region abstractions, a DBA need only make an easy configuration change at the database level. The following commands add all three regions to the database and set "us-east1" as the the default home for leaseholder replicas:

```
> ALTER DATABASE movr SET PRIMARY REGION "us-east1";
  ALTER DATABASE movr ADD REGION "us-west1";
  ALTER DATABASE movr ADD REGION "europe-west1";
```

These commands take a few seconds to complete because they trigger a number of automatic configuration changes: First, we have modified the database descriptor to indicate that it is a multi-region database with zone survivability (the default):

```
> SELECT regions, survival_goal FROM [SHOW DATABASES]
  WHERE database_name = 'movr';
              regions              | survival_goal
-----------------------------------+----------------
  {europe-west1,us-east1,us-west1} | zone
```

Additionally, we have modified all of the tables in the database to have the default table locality:

```
> SELECT table_name, locality FROM [SHOW TABLES];
  table_name  |               locality
--------------+--------------------------------------
  promo_codes | REGIONAL BY TABLE IN PRIMARY REGION
  rides       | REGIONAL BY TABLE IN PRIMARY REGION
  users       | REGIONAL BY TABLE IN PRIMARY REGION
```

Finally, we have applied a zone configuration at the database level to place all voting replicas for this database in the primary region, and one non-voting replica in each other region:

```
> SELECT raw_config_sql FROM [SHOW ZONE CONFIGURATION
  FOR DATABASE movr];
                raw_config_sql
-------------------------------------------------
  ALTER DATABASE movr CONFIGURE ZONE USING
      num_replicas = 5,
      num_voters = 3,
      constraints = '{+region=europe-west1: 1,
        +region=us-east1: 1, +region=us-west1: 1}',
      voter_constraints = '[+region=us-east1]',
      lease_preferences = '[[+region=us-east1]]'
```

Non-voting replicas are important as they allow the non-primary regions to serve follower reads without increasing write latency. Without follower reads, a client issuing the query SELECT * FROM promo_codes; will only experience low latency (< 10 ms) if they are in the primary region, since consistent reads must be served by the leaseholder for REGIONAL tables.

With follower reads, the same query results in low-latency for clients in any database region thanks to the non-voting replicas:

```
> SELECT * FROM promo_codes AS OF SYSTEM TIME
  follower_read_timestamp();
  code  |          description
--------+-------------------------------
  10off | ride global and get 10% off
Time: 3ms
```

The REGIONAL BY TABLE locality may not be the best option for some tables. REGIONAL BY ROW is a better choice for the users table as it allows fast consistent reads and writes from a region specified at the row level, matching each user's region of residence.

```
> ALTER TABLE users SET LOCALITY REGIONAL BY ROW;
```

This command triggers a number of automatic actions. First, we add a hidden column called `crdb_region` which encodes the optimal region for reads and writes of each individual row. We can see this new column with introspection:

```
> SELECT create_statement FROM [SHOW CREATE TABLE users];
                  create_statement
--------------------------------------------------------
  CREATE TABLE users (
    id UUID NOT NULL DEFAULT gen_random_uuid(),
    name STRING NOT NULL,
    email STRING NOT NULL,
    home_addr STRING NOT NULL,
    crdb_region crdb_internal_region NOT VISIBLE NOT
      NULL DEFAULT default_to_database_primary_region(
      gateway_region())::crdb_internal_region,
    CONSTRAINT users_pkey PRIMARY KEY (id ASC),
    UNIQUE INDEX users_email_key (email ASC)
  ) LOCALITY REGIONAL BY ROW
```

The new column is added using new `NOT VISIBLE` syntax which means that the column won't show up by default in `SELECT *` queries (keeping it out of the way of users). If it is unspecified on `INSERT`, `crdb_region` will get a default value matching the region of the SQL gateway node. We also partition the table, and all of its indexes, by this hidden column. Zone configurations are assigned to each partition so that they reside in the desired region.

Since all indexes are implicitly partitioned by the `crdb_region` column, the indexes can no longer enforce global `UNIQUE` constraints, and we need to check that any mutation to a table will not violate a constraint. This check is added by the optimizer as a "post query" to be executed after the mutation, and causes the transaction to fail if any rows are returned:

```
> EXPLAIN INSERT INTO users (name, email) VALUES (
  'Craig Roacher', 'craig@cockroachlabs.com');
                          info
---------------------------------------------------------
  • root
  ├── • insert into: users (id, name, email, crdb_region)
  │   └── • values (gen_random_uuid(), 'Craig Roacher',
  │             'craig@cockroachlabs.com', 'us-east1')
  └── • constraint-check: error if rows
      └── • semi join (lookup users@users_email_key)
          └── • scan buffer
```

In addition to maintaining the consistency of customers' `UNIQUE` constraints, an important side benefit of enforcing these constraints is that we can use them for locality optimized search. This optimization applies if we are searching for keys that we know to be unique but do not know where they are located. We first search the local region, and if all the requested keys are found locally, we do not need to fan out to remote regions since we know that we have found the only rows with those keys. The optimization is implemented using a limited `UNION ALL`, allowing the operator to short-circuit if the limit is reached by the first child:

```
> EXPLAIN SELECT * FROM users WHERE
  email = 'craig@cockroachlabs.com';
                            info
---------------------------------------------------------
  • index join (users@users_pkey)
  └── • union all
      │ limit: 1
      ├── • scan: users@users_email_key
      │     [/'us-east1'/'craig@cockroachlabs.com']
      └── • scan: users@users_email_key
            [/'europe-west1'/'craig@cockroachlabs.com']
            [/'us-west1'/'craig@cockroachlabs.com']
```

In our demonstration, attendees can simulate running this query from each region, and see that it is significantly faster in the region where the row is located, since we can avoid visiting other regions.

Another table in movr's schema that is well suited to `REGIONAL BY ROW` locality is the `rides` table, since rides usually start and end in the same region. `rides` has an additional complication, which is that it includes a foreign key reference to the `promo_codes` table. Inserting a row with a non-null promo code requires validating that the code exists in `promo_codes` to maintain referential integrity. To ensure serializability, this read must be consistent; it cannot be a stale follower read. Since all of movr's promo codes can be used from any region, this requires a cross-region hop in most cases. The following insert results in high latency in two out of three regions:

```
> INSERT INTO rides (id, revenue, promo_code) VALUES (
  DEFAULT, 36.12, '10off')
INSERT 1
Time: 131ms
```

The solution is to make `promo_codes` a `GLOBAL` table, which will ensure low latency for the previous insert from all regions:

```
> ALTER TABLE promo_codes SET LOCALITY GLOBAL;
```

`promo_codes` is a good candidate for `GLOBAL` locality, as it is infrequently updated, but frequently read from all regions.

With only a few simple commands to alter database regions and table locality, movr has a multi-region application that maintains the semantics and performance of the single-region application.

REFERENCES

[1] 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/
[2] DataStax Documentation. [n.d.]. Apache Cassandra Lightweight Transactions. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useInsertLWT.html.
[3] 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.
[4] 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).
[5] Nathan VanBenschoten, Arul Ajmani, et al. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *Proceedings of the 2022 ACM International Conference on Management of Data (SIGMOD '22).*
[6] Vitess. [n.d.]. The Vitess Docs | Sharding. https://vitess.io/docs/reference/features/sharding/.
[7] 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.