[[Ch06 – Application design and implementation]]

# == Application design and implementation

Like all databases, CockroachDB responds to requests from application code. How an application requests and uses data has a huge bearing on application performance and scalability. In this chapter, we'll review how an application should work with CockroachDB – including best practices for coding CockroachDB requests and transactional models.

Because CockroachDB is PostgreSQL protocol-compatible, any language that supports PostgresSQL can be used with CockroachDB. And in general, the programming idioms and best practices of PostgreSQL apply to CockroachDB. However, because CockroachDB behaves differently at a server level than PostgreSQL, there are some differences in programming styles between CockroachDB and PostgreSQL.

Although you can work with CockroachDB using pretty much any programming language in common use, in this chapter, we'll constrain our discussion to these four languages: Go, Java, Python and JavaScript.

In Chapter 3, we showed how to install language drivers for each of these languages. Please refer back to Chapter 3 for instructions on driver installation, or refer to the CockroachDB documentation footnote:[ https://www.cockroachlabs.com/docs/stable/hello-world-example-apps] for more detailed guidelines, including guidance on how to install drivers for other languages or for alternative drivers.

## === CockroachDB programming

### ==== Performing CRUD operations

We provided basic "Hello world" examples for each language back in Chapter3. Let's extend those examples to perform some non-trivial "CRUD" operations – Create, Read, Update, Delete.

Programming drivers differ in terms of vocabulary, but they generally adopt a similar grammar. The fundamental operations in a database program are:

\* The driver establishes a \*connection\* object representing a connection to the database server. In this chapter we’ll be creating individual connections, but applications will often use a \*connection pool\* to manage multiple reusable connections instead.

\* The connection object is used to create \*statements\*, that represent commands that can be submitted to the databases

\* Some statements return \*Result Sets\* that can be used to iterate through tabular output returned by SELECT statements, DML statements that include a RETURNING clause and some other statements that return results.

Here we see this basic pattern in Java:

[source, java]

----

package helloCRDB;

import java.sql.\*;

public class example1 {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String connectionURL = "jdbc:" + args[0];

String userName = args[1];

String passWord = args[2];

Connection connection = DriverManager.getConnection(connectionURL,

userName, passWord);

Statement stmt = connection.createStatement();

stmt.execute("DROP TABLE IF EXISTS names");

stmt.execute("CREATE TABLE names (name String NOT NULL)");

stmt.execute("INSERT INTO names (name) VALUES('Ben'),('Jesse'),('Guy')");

ResultSet results = stmt.executeQuery("SELECT name FROM names");

while (results.next()) {

System.out.println(results.getString(1));

System.out.println(results.getString("NAME"));

}

results.close();

stmt.close();

connection.close();

} catch (Exception e) {

e.printStackTrace();

System.exit(0);

}

}

}

----

We create a single \*Connection\* object and a single \*Statement\* object, then the statement those to execute multiple SQL commands. When we execute a query, we create a \*ResultSet\* object that we can use to iterate through results. Finally, we close all these objects.

Note that we can retrieve column values from the ResultSet object by position or by name – both styles are illustrated in the above example.

Below we see similar logic for Python. The \*cursor()\* method of the connection object creates a cursor object that can be used to execute a statement or navigate through a result set.

[source, python]

----

import psycopg2

import sys

def main():

if ((len(sys.argv)) !=2):

sys.exit("Error:No URL provided on command line")

uri=sys.argv[1]

connection = psycopg2.connect(uri)

cursor=connection.cursor()

cursor.execute("DROP TABLE IF EXISTS names")

cursor.execute("CREATE TABLE names (name String NOT NULL)")

cursor.execute("INSERT INTO names (name) VALUES('Ben'),('Jesse'),('Guy')")

cursor.execute("SELECT name FROM names")

for row in cursor:

print(row[0])

cursor.close()

connection.close()

main()

----

Here we do the same thing in a NodeJS JavaScript program:

[source,javascript]

----

const CrClient = require('pg').Client;

async function main() {

try {

if (process.argv.length != 3) {

console.log(`Usage: node ${process.argv[1]} CONNECTION\_URI`);

process.exit(1);

}

const connection = new CrClient(process.argv[2]);

await connection.connect();

await connection.query('DROP TABLE IF EXISTS names');

await connection.query('CREATE TABLE names (name String NOT NULL)');

await connection.query(`INSERT INTO names (name)

VALUES('Ben'),('Jesse'),('Guy')`);

const data = await connection.query('SELECT name from names');

data.rows.forEach((row) => {

console.log(row.name);

});

} catch (error) {

console.error(error.stack);

}

process.exit(0);

}

main();

----

We've used the "async/await" style for handling asynchronous database requests. You can also use callbacks or promises if that is your programming style. The node-postgres driver documentationfootnote:[ <https://node-postgres.com/features/queries>] contains examples of using each of these programming styles.

Finally, let us look at how we'd perform the same task in Go:

[source, golang]

----

package main

import (

"context"

"fmt"

"os"

"github.com/jackc/pgx"

)

func main() {

if len(os.Args) < 2 {

fmt.Fprintln(os.Stderr, "Missing URL argument")

os.Exit(1)

}

uri := os.Args[1]

conn, err := pgx.Connect(context.Background(), uri)

if err != nil {

fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)

}

execSQL(\*conn, "DROP TABLE IF EXISTS names")

execSQL(\*conn, "CREATE TABLE names (name String NOT NULL)")

execSQL(\*conn, "INSERT INTO names (name) VALUES('Ben'),('Jesse'),('Guy')")

rows, err := conn.Query(context.Background(), "SELECT name FROM names")

if err != nil {

fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)

}

defer rows.Close()

for rows.Next() {

var name string

err = rows.Scan(&name)

fmt.Println(name)

}

}

func execSQL(conn pgx.Conn, sql string) {

result, err := conn.Exec(context.Background(), sql)

if err != nil {

fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)

os.Exit(1)

}

fmt.Fprintf(os.Stdout, "%v rows affected\n", result.RowsAffected())

}

----

We created the +execSQL+ function in the Go example to modularize the repetitive error checking involved in the initial SQL statements, though in production code, we would perform error checking independently for each query.

### ==== Cursors

A \*cursor\* is an object that allows you to scroll through the results of a query rather than retrieving all the data in one hit. Cursors are a preferred means of dealing with large amounts of data since they avoid the necessity of holding the complete result set in memory and allow you to abort query processing if you actually only want the first few rows.

For instance, let's say we have a web application that displays blog posts ordered by time. The key query might look something like this:

[source,sql]

----

SELECT post\_timestamp, summary

FROM blog\_posts ORDER BY post\_timestamp DESC

----

We have a covering index on POST\_TIMESTAMP, and this index stores the SUMMARY column, so we can retrieve rows efficiently in order. We display our data a page at a time, so we might code a nodeJs routine something like this (to render the first page with ten items):

[source, javascript]

----

const sql = `SELECT post\_timestamp, summary

FROM blog\_posts ORDER BY post\_timestamp DESC`;

// let rows=await connection.query(sql);

const data = await connection.query(sql);

for (let i = 0; i < 10; i++) {

console.log(data.rows[i]);

}

----

The problem with this code is that we need to pull all of the contents of the table across the network just to display the first ten rows.

A cursor lets us pull data on demand so that we don't need to pull the second and subsequent pages of data until or unless needed.

To use cursors in NodeJS pg driver, we have to install the associated \*pg-cursor\* package:

[source, javascript]

----

const Cursor = require('pg-cursor');

----

Having done that, we can then define a cursor object and pull rows from it on-demand:

[source, javascript]

----

const cursor = await connection.query(new Cursor(sql));

cursor.read(10, (err, rows) => {

console.log(rows);

if (err) {

throw err;

}

});

----

The response time implications are significant: for a 5 million row table, the cursor implementation returned the first ten rows in 13ms, compared to 14,556 ms for the vanilla query implementation. <<nodejsFetch>> compares the two approaches.

[[nodejsFetch]]

.Comparison of NodeJs fetch options

image::images/nodejsFetch.png[nodejsFetch]

.Cursors vs. LIMIT

\*\*\*\*

While cursors allow you to pull data from a query on-demand, LIMIT restricts the absolute amount of data requested from the database on the server-side.

The two implementations can result in an identical outcome and often similar performance characteristics. However, if you know for certain that the number of rows to be returned is limited, then using LIMIT is by far the more effective strategy. With LIMIT, the optimizer knows in advance that only a subset of rows will be returned and can pick a better plan. For instance, the optimizer might choose to use an index to avoid a sort \*only\* if it knows that not all of the data in the result set will be returned.

However, you can't "paginate" with LIMIT – there's no efficient way to request second and subsequent pages of information. For this reason, the use of cursors is preferred when possible.

\*\*\*\*

JDBC Result set objects do not give you the option of fetching all rows in one operation (e.g., there's no equivalent to the Python \*fetchall()\* function). However, that doesn't mean that the Java driver is pulling rows across the network one at a time. Under the hood, the JDBC driver retrieves rows in batches whose size is controlled by the \*setFetchSize()\* method of the Statement object. By default, fetchSize is set to 0, which results in \*all\* the rows being pulled into the application before the first row can be processed.

We can adjust the fetchSize if we want to pull only a few rows in each batch as followsfootnote:[Note that the setFetchSize() call has no effect if setAutoCommit is set to true]:

[source, java]

----

Statement stmt = connection.createStatement();

stmt.setFetchSize(100);

results = stmt

.executeQuery("SELECT post\_timestamp, summary "

+ " FROM blog\_posts "

+ " ORDER BY post\_timestamp DESC ");

for (int ri = 0; ri < 10; ri++) {

if (results.next())

System.out.println(results.getString("SUMMARY"));

}

----

You don't have to change your loop logic when you change setFetchSize, but under the hood, the PostgreSQL Driver will pull rows in batches of \*setFetchSize()\* size. <<setFetchSize>> shows that this can be very effective if we want to optimize for fetching the first few rows.

Chart, bar chart

Description automatically generated

[[setFetchSize]]

.Reducing setFetchSize to improve fetch time for first rows

image::images/setFetchSize.png[setFetchSize]

.Cursors in Go and Python

\*\*\*\*

In the pgx Go driver, rows are accessed sequentially as the Next() function is called on the results. However, all the rows are moved into program memory when the query() method is called, so even if you only access the first row, you will pull all rows across the network.

The Python psycopg2 driver provides methods access to the entire result set (fetchall()), a selection of rows (fetchmany()) or a single row (fetchone()). However, regardless of the method called, the entire result set is always transferred from the database to the application.

These restrictions are characteristics of the drivers themselves and are not specific to CockroachDB – they apply equally to PostgreSQL applications.

For these drivers, you may want to use LIMIT or otherwise prevent massive result sets being transferred unnecessarily. The CRDB team is working with the authors of these packages with an aim to improve their efficiency.

\*\*\*\*

### ==== Prepared and parameterized statements

Most SQL operations are parameterized – the same statement is run multiple times with different input parameters. For instance, we might have a lookup program that retrieves rider names for a specified ride id as follows:

[source, sql]

----

**SELECT** u.**name** **FROM** movr.rides r

**JOIN** movr.**users** u **ON** (r.rider\_id=u.id)

**WHERE** r.id='ffc3c373-63ec-43fe-98ff-311f29424d8b'

----

We would, of course, execute this SQL many times, each time specifying a different value for the Rider Id.

When coding a generic lookup function, it seems natural enough to append the parameter to the SQL statement using string concatenation. For instance, in Java we might be tempted to do something like this:

[source, java]

----

private static String getRiderName(String riderId) throws SQLException {

Statement stmt = connection.createStatement();

String sql = " SELECT u.name FROM movr.rides r "

+ " JOIN movr.users u ON (r.rider\_id=u.id) "

+ " WHERE r.id='"

+ riderId + "'";

ResultSet rs = stmt.executeQuery(sql);

rs.next();

return (rs.getString("name"));

}

----

However, as natural as this might seem, it represents an extremely poor practice that has both performance and security downsides.

Most significantly, this code is vulnerable to \*SQL Injection\*. For instance, imagine the application could somehow be persuaded to pass the following string to the function:

[source, java]

----

riderName = getRiderName(

"ffc3c373-63ec-43fe-98ff-311f29424d8b' UNION select credit\_card from movr.users order by 1,name 'n");

----

The resulting SQL statement would become:

[source, sql]

----

**SELECT** u.**name** **FROM** movr.rides r

**JOIN** movr.**users** u **ON** (r.rider\_id=u.id)

**WHERE** r.id='ffc3c373-63ec-43fe-98ff-311f29424d8b'

**UNION** **select** credit\_card **from** movr.**users** **order** **by** 1,**name**

----

And the function would now return credit card numbers as well as rider names.

Of course, the application should prevent such a string from being entered at the user interface layer, but creating the vulnerability in the database code is very poor practice.

The solution is to use \*prepared\* or \*parameterized\* statements. For instance, in the Java example above, we would declare a \*preparedStatement\* as follows:

[source, java]

----

*getRiderStmt* = *connection*.prepareStatement(

"SELECT u.name FROM movr.rides r "

+ " JOIN movr.users u ON (r.rider\_id=u.id) "

+ " WHERE r.id=?");

----

The "?" indicates a placeholder for a parameter (sometimes called a \_bind variable\_). We can call the prepared statement by setting the parameter and executing the statement:

[source, java]

----

getRiderStmt.setString(1, riderId);

ResultSet rs = getRiderStmt.executeQuery();

rs.next();

return (rs.getString("name"));

----

As well as avoiding SQL injection, preparedStatements generally execute faster because CockroachDB can more easily recognize the SQL as one that has already been parsed and can avoid some of the overhead involved with examining what would otherwise appear to be a brand new statement.

Formally "preparing" statements is a Java practice. In other languages, it's sufficient to simply call a SQL statement with placeholders and provide the values in the call. For instance, in JavaScript:

[source, javascript]

----

const sql = `SELECT u.name FROM movr.rides r

JOIN movr.users u ON (r.rider\_id=u.id)

WHERE r.id=$1`;

const results = await connection.query(sql, ['ffc3c373-63ec-43fe-98ff-311f29424d8b']);

console.log(results.rows[0].name);

----

In Python:

[source, python]

----

sql = """SELECT u.name FROM movr.rides r

JOIN movr.users u ON (r.rider\_id=u.id)

WHERE r.id=%s"""

cursor.execute(sql,('ffc3c373-63ec-43fe-98ff-311f29424d8b',))

row=cursor.fetchone()

print(row[0])

----

And in Go:

[source, go]

----

sql := `SELECT u.name FROM movr.rides r

JOIN movr.users u ON (r.rider\_id=u.id)

WHERE r.id=$1`

rows, err := conn.Query(ctx, sql, "ffc3c373-63ec-43fe-98ff-311f29424d8b")

rows.Next()

var name string

err = rows.Scan(&name)

fmt.Println(name)

----

### ==== Connection Pools

In a microservices architecture, we create small routines to perform small tasks. If the service requires database access, then it might seem natural to supply each microservice request with a dedicated connection. This has a clear advantage over a single shared connection since it allows for concurrent requests. For instance, imagine that we have a simple web service that we call whenever a new ride is commenced in our Uber-busting ride-sharing app. We might code the database logic for it as follows:

[source, javascript]

----

async function newRide(city, riderId, vehicleId, startAddress) {

const connection = new pg.Client(connectionString);

await connection.connect();

const sql = `INSERT INTO movr.rides

(id, city,rider\_id,vehicle\_id,start\_address,start\_time)

VALUES(gen\_random\_uuid(), $1,$2,$3,$4,now())`;

await connection.query(sql, [city, riderId, vehicleId, startAddress]);

await connection.end();

}

----

We don't want to single-thread these requests, so we've given each call its own connection. Unfortunately, creating a connection has a non-trivial overhead. When the database access is very simple, the time taken to create and dispose of the connection might dominate overall response time. But we can't run every request through the same connection because that would restrict concurrent queries.

The solution is to use \*Connection Pools\*. A connection pool is a set of connections that can be reused by the application. You avoid the overhead of constantly creating and destroying connections, and you can control the maximum amount of concurrency hitting the database.

In NodeJS, we'd create the pool as follows:

[source,javascript]

----

const pool = new pg.Pool({

connectionString,

max: 40

});

----

We now can change our routine so that it gets connections from the pool:

[source, javascript]

----

async function newRidePool(city, riderId, vehicleId, startAddress) {

const connection = await pool.connect();

const sql = `INSERT INTO movr.rides

(id, city,rider\_id,vehicle\_id,start\_address,start\_time)

VALUES(gen\_random\_uuid(), $1,$2,$3,$4,now())`;

await connection.query(sql, [city, riderId, vehicleId, startAddress]);

await connection.release();

}

----

<<connectionPool>> illustrates how the two approaches compare for performance. With 40 concurrent requests, a connection pool implementation outperformed the unique connection approach by about 700%.

[[connectionPool]]

.Using connection pools to improve concurrency

image::images/connectionPool.png[connectionPool]

The amount of benefit you get from connection pools will vary depending on the amount of work performed in each connection and the amount of concurrent activity the application issues. However, it's almost always advisable to use a connection pool in preference to a single connection used by all threads or allocating each thread with its own transitory connection.

.Connection Pools and blocked connections

\*\*\*\*

Most connection pool implementations will block requests for new connections if all the pooled connections are in use. Therefore, it's important to configure a sufficient number of connections in the pool for the anticipated concurrency. A common rule of thumb is to configure four connections for every core in the entire cluster. For instance, if you have a three-node cluster with eight cores in each node, you might configure 3\*8\*4=96 connections. However, bear in mind that this is just a guideline – the optimal number will depend heavily on the duration of each connection and the amount of idle time each connection experiences as the application performs non-database tasks.

It's also critically important to release connections when not in use. For instance, in the NodeJS example, the \_connection.release()\_ statement at the end of our function is very important.

\*\*\*\*

In Java, there are a variety of connection Pool optionsfootnote:[For instance, see <https://www.baeldung.com/java-connection-pooling>]. Here's an example using the Hikari frameworkfootnote:[https://github.com/brettwooldridge/HikariCP]:

[source,java]

----

import com.zaxxer.hikari.\*;

import java.sql.\*;

public class ConnectionPoolDemo {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String connectionURL = "jdbc:" + args[0];

String userName = args[1];

String passWord = args[2];

HikariConfig config = new HikariConfig();

config.setJdbcUrl(connectionURL);

config.setUsername(userName);

config.setPassword(passWord);

config.addDataSourceProperty("ssl", "true");

config.addDataSourceProperty("sslMode", "require");

config.addDataSourceProperty("reWriteBatchedInserts", "true");

config.setAutoCommit(false);

config.setMaximumPoolSize(40);

config.setIdleTimeout(3000);

HikariDataSource hikariPool = new HikariDataSource(config);

----

This example creates a connection pool with 40 connections using arguments passed in on the command line. Once the pool is created, a connection can be obtained from the pool as follows:

[source,java]

----

Connection connection = hikariPool.getConnection();

----

In the Go pgx driver, we can use the pgxpool package to create and use a connection pool:

[source, go]

----

ctx := context.Background()

config, err := pgxpool.ParseConfig(uri)

config.MaxConns = 40

pool, err := pgxpool.ConnectConfig(ctx, config)

defer pool.Close()

----

We can acquire a connection from the pool as follows:

[source, go]

----

connection, err := pool.Acquire(ctx)

----

Psycopg2 for Python includes a built-in connection pool which we can easily configure as follows:

[source, python]

----

import psycopg2

from psycopg2 import pool

def main():

if ((len(sys.argv)) !=2):

sys.exit("Error:No URL provided on command line")

uri=sys.argv[1]

pool= psycopg2.pool.ThreadedConnectionPool(10, 40, uri)

# min connection=10, max=40

----

And we can connect to the pool as follows:

[source, python]

----

connection = pool.getconn()

----

### ==== Bulk inserts

It's very common for an application to insert multiple rows of data in a single logical operation.

When you have an array of values to insert, it can seem natural to simply insert the values in a loop, as in this python example:

[source, python]

----

for value in arrayValues:

cursor.execute("INSERT INTO insertTestP1(id,x,y) VALUES ($1,$2,$3)",value)

----

It's very inefficient to insert large amounts of data one at a time – each insert will require a network round trip, and there may be transactional implications if we want the entire batch to be inserted in a single truncation (by taking longer, the chance of a transaction conflict and subsequent retry will be magnified).

SQL allows multiple VALUES to be included in a single operation, for instance:

[source, sql]

----

**INSERT** **INTO** insertTest(id,x,y)

**VALUES** (3,'x',1) ,

(4,'y',2) ,

(5,'x',5)

----

So we could, if necessary, dynamically construct an INSERT statement to insert an array of data in a single operation. For instance, in Python, the following code will generate and execute an INSERT statement to insert an array of arbitrary length:

[source, python]

----

sql="INSERT INTO insertTestP(id,x,y) VALUES"

valueCount=0

for value in arrayValues:

if valueCount>0:

sql=sql+","

sql=sql+"(%d,'%s',%d)" % value

valueCount+=1

cursor.execute(sql)

----

In the psycopg2 extras package, there is an execute\_extras helper function that simplifies the coding required:

[source, python]

----

from psycopg2 import extras

<snip>

extras.execute\_values(cursor,

"INSERT INTO insertTestP1(id,x,y) VALUES %s",

arrayValues)

----

The performance improvements obtained with array inserts are dramatic.

[[arrayInsert]]

.Improvement obtained by inserting rows in an array

image::images/arrayInsert.png[arrayInsert]

JDBC includes \*addBatch\* and \*executeBatch\* methods that allow you to prepare inserts one at a time and then submit all the inserted values in a single operation. This avoids the need to concatenate a huge VALUES list and allows us to use formal parameters.

Here's an example of the \*addBatch\* and \*executeBatch\* methods:

[source, java]

----

String sql="INSERT INTO insertTest(id,x,y) VALUES (?,?,?)";

PreparedStatement InsertStmt = connection.prepareStatement(sql);

**for** (**int** arrayIdx = 1; arrayIdx < arrayCount; arrayIdx++) {

InsertStmt.setInt(1, idArray.get(arrayIdx));

InsertStmt.setString(2, xArray.get(arrayIdx));

InsertStmt.setInt(3, yArray.get(arrayIdx));

InsertStmt.addBatch();

}

InsertStmt.executeBatch();

----

We use \*setInt\* and \*setString\* methods to supply values to the prepared statement as usual, but instead of executing, we use \*addBatch\* to add them to the batch of rows to be inserted. When we are ready, we call \*executeBatch\* to add all the rows in a single operation.

The pg NodeJS library does not include any direct support for batch inserts. However, we can use the \*pg-format\* package to create SQL statements that contain multiple VALUES from an array:

[source, javascript]

----

const pg = require('pg');

const format = require('pg-format');

async function main() {

const connection = new pg.Client(connectionString);

const sql = format('INSERT INTO insertTestP2(id,x,y) VALUES %L,

arrayData);

await connection.query(sql);

----

The Go pgx library does not currently support bulk inserts directly. You would need to construct dynamic SQL with multiple VALUES entries, as shown earlier for Python. However, some users have written helper functions to expedite dynamic SQL generation for bulk insertsfootnote:[https://github.com/jackc/pgx/issues/764#issuecomment-685249471].

### ==== Projections

In relational database parlance, "projection" refers to the selection of a subset of columns from a table (or \_attributes\_ from an \_entity\_). In practice, a projection is represented by the list of columns in a SELECT clause.

While SELECT accepts a wildcard projection "\*", this should almost never be used in production code since it results in unnecessary transport of columns from the database to the application. Using "\*" can seem like a handy programming shortcut, but it can have severe performance penalties when processing large result sets.

For instance, let's say we are retrieving a list of user ids and blog post dates to populate a dashboard or to perform some other real-time diagnostic. The following code might seem acceptable:

[source, java]

----

ResultSet results = stmt.executeQuery(

"SELECT \* FROM blog\_posts");

while (results.next()) {

java.sql.Timestamp postTimestamp =

results.getTimestamp("POST\_TIMESTAMP");

Integer userid = results.getInt("USERID");

plotPost(userid, postTimestamp);

}

----

However, a couple of coding seconds saved in omitting the column names costs the application dearly. Every time this code is executed, it retrieves not only the user id and timestamp, but also the potentially very large blog post text. As a result, each network packet can hold less data, and the number of network round trips is magnified. If we add a projection:

[source, java]

----

results = stmt.executeQuery(

"SELECT userid,post\_timestamp FROM blog\_posts");

----

Then elapsed time is reduced dramatically. <<projections>> illustrates the elapsed time savings for a ten million row result set from a remote cluster.

[[projections]]

.Improvement obtained by adding a projection to a query

image::images/projections.png[projections]

Chart, bar chart

Description automatically generated

Of course, the absolute time saved will depend on the total row size versus the size of the projection and the network latency between the application and the server. Furthermore, this degradation only kicks when we pull more rows from the database that can fit in a single network packet. For single-row retrievals, the overhead is negligible.

### ==== Client side caching

The best way to optimize a database request is not to send it at all. No matter how carefully we optimize the database – adding indexes, memory, fast disks, etc. –database requests are blocking operations that can never be made as fast as local computation. For most applications, database accesses are the slowest operations performed and the most critical component of application response time.

One of the most effective ways of avoiding unnecessary database calls is to cache frequently accessed static data in application code. Avoid asking the database over and over again for the same data unless there's a chance that the data will change.

For instance, let’s say that we have a function to return a users name given a userId:

[souce, golang]

----

func getUserName(userId string) string {

conn, err := pool.Acquire(context.Background())

defer conn.Release()

if err != nil {

fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)

}

sql := `SELECT name FROM movr.users WHERE id=$1`

rows, err := conn.Query(context.Background(), sql, userId)

defer rows.Close()

if err != nil {

fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)

}

if !rows.Next() {

return "Invalid userId"

} else {

var name string

rows.Scan(&name)

return (name)

}

}

----

It’s relatively simply to extend this function with a client side cache. We just need to declare an initialize a map structure:

[source, go]

----

var userCache map[string]string

userCache = make(map[string]string)

----

Now in our function we check this map to see if we can find the user’s name. Only if the name does not exist in the cache do we go do the database:

[source, golang]

----

func getCachedUserName(userId string) string {

name, nameFound := userCache[userId]

if !nameFound {

conn, err := pool.Acquire(context.Background())

defer conn.Release()

if err != nil {

fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)

}

fmt.Println("cache miss")

sql := `SELECT name FROM movr.users WHERE id=$1`

rows, err := conn.Query(context.Background(), sql, userId)

defer rows.Close()

if err != nil {

fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)

}

if !rows.Next() {

return "Invalid userId"

} else {

rows.Scan(&name)

userCache[userId] = name

}

}

return (name)

}

----

The performance improvements obtained by not going to the database are greater than any tuning of the databases accesses themselves, since we can never make a database access a zero-cost activity. However, bear in mind that:

\* Caches consume memory on the client program. In many environments, memory is abundant and the tables considered for caching relatively small. However, for large tables and memory-constrained environments the implementation of a caching strategy could actually degrade performance by contributing the memory shortages in the application layer or client.

\* If the table being cached is updated during program execution, then the changes may not be reflected in your cache unless you implement some sophisticated synchronization mechanism. For this reason, local caching is best performed on static tables.

## === Managing Transactions

Transactions provide an important mechanism for ensuring that related modifications succeed or fail as a unit. We discussed the internals of CockroachDB transactions back in Chapter 2.

The basics of programming transactions are common across a wide variety of SQL databases and even some non-SQL systems. A transaction is commenced with a BEGIN statement. Multiple SQL statements are executed within the transaction scope, and then all the changes are made permanent with the COMMIT statement. If an error is encountered during the transaction, all of the transaction's work can be abandoned with a ROLLBACK statement.

Here's a relatively simple transaction consisting of an INSERT followed by an UPDATE – implemented in NodeJS JavaScript:

[source, javascript]

----

async function takeMeasurement(locationId, measurement) {

let success = false;

const measurementTime = new Date();

const connection = await pool.connect();

try {

await connection.query('BEGIN TRANSACTION');

await connection.query(

`INSERT INTO measurements

(locationId,measurement)

VALUES($1,$2)`,

[locationId, measurement]

);

await connection.query(`UPDATE locations

SET last\_measurement=$1, last\_timestamp=$2

WHERE id=$3`,

[measurement, measurementTime, locationId]);

await connection.query('COMMIT');

success = true;

} catch (error) {

console.error(error.message);

connection.query('ROLLBACK');

success = false;

}

connection.release();

return (success);

}

----

### ==== Transaction Retry errors

In databases that default to lower levels of transaction isolation (MySQL, for instance), this transaction would almost always succeed, perhaps failing only if there was a database outage. However, in SERIALIZABLE transaction isolation (which is the default in CockroachDB and an option in other databases), there is a very good chance of transaction failure. If a concurrent transaction modifies the same LOCATION table row between the time our transaction commences and the time we attempt to modify that row, then we will encounter a \_TransactionRetryWithProtoRefreshError: WriteTooOldError\_ (We'll call this a \*TransactionRetry\* error for the sake of brevity).

<<transactionRetry>> illustrates a typical sequence of events in two concurrent transactions that would lead to a transactionRetry error.

[[transactionRetry]]

.Transaction Retry error scenario

image::images/transactionRetry.png[transactionRetry]

Diagram

Description automatically generated

The chance of receiving a TransactionRetry error depends on the chance of two transactions colliding on the same row. For our above example, the percentage of retries varied from less than 1% if there were 10,000 distinct locations to more than 75% when there were just ten locationsfootnote:[The simulation ran 100 concurrent threads randomly executing a transaction 10 times a second].

However, whatever the possibility of encountering a transaction Retry error, the possibility exists, and your application code should be able to cope with these expected error scenarios.

==== Implementing transaction retries

The relatively obvious way to handle retries errors is to do exactly what the error code suggests – retry the transaction. When the TransactionRetry error is encountered, issue a ROLLBACK command to discard the work done so far in the transaction and try the transaction again.

Here are some modifications to our JavaScript NodeJS method to retry the transaction when necessary:

[source, javascript]

----

async function takeMeasurementWithRetry(locationId, measurement, maxRetries) {

const connection = await pool.connect();

const measurementTime = new Date();

let retryCount = 0;

let transactionEnd = false;

while (!transactionEnd) {

retryCount += 1;

if (retryCount >= maxRetries) {

throw Error('Maximum retry count exceeded');

} else {

try {

await connection.query('BEGIN TRANSACTION');

await connection.query(

`INSERT INTO measurements

(locationId,measurement)

VALUES($1,$2)`,

[locationId, measurement]

);

await connection.query(`UPDATE locations

SET last\_measurement=$1, last\_timestamp=$2

WHERE id=$3`,

[measurement, measurementTime, locationId]);

await connection.query('COMMIT');

transactionEnd = true;

} catch (error) {

if (error.code == '40001') { // Rollback and retry

connection.query('ROLLBACK');

const sleepTime = (2 \*\* retryCount) \* 100

+ Math.ceil(Math.random() \* 100);

console.warn('Sleeping for ', sleepTime);

await sleep(sleepTime);

} else {

console.log('aborted ', error.message);

transactionEnd = true;

}

}

}

}

connection.release();

return (retryCount);

}

----

If this method encounters an error '40001' – the retry transaction code – it issues a ROLLBACK, waits for a short time and then tries the transaction again.

In this implementation, the sleep time increases exponentially as the number of retries increases. This is done to avoid a situation in which transactions "thrash" on a resource. This exponential backoff strategy tends to reduce the load on a busy system, but it can result in some very high transaction waits for "unlucky" transactions. Furthermore, when we retry transactions, there is no guarantee that updates will succeed in the order in which they are originally submitted. Transactions that are submitted first may only succeed after transactions submitted at a later date.

### ==== Automatic transaction retries

The logic shown in the previous section can be implemented in any languagefootnote:[See <https://www.cockroachlabs.com/docs/stable/transactions#client-side-intervention-example> for a generic implementation]. However, some drivers implement this logic for you transparently:

\* The GoLang dbtools library includes a transaction retry handler for Go transactions. You pass a set of operations to the transaction handler which will automatically retry transactions with a configurable retry limit and delayfootnote:[ <https://pkg.go.dev/github.com/arsham/dbtools>].

\* Many Object Relational Mapping frameworks – SQLAlchemy for Python, for instance – will automatically retry transactions for you transparently.

### ==== Avoiding transaction retry errors with FOR UPDATE

Performing transaction retries has some significant downsides. Firstly, they are wasteful since work in the transaction that is done prior to the retry is discarded. Secondly, they introduce a delay in transaction processing that is unpredictable or even unnecessary. It's hard to know how long to sleep between transaction retries, and exponential backoffs can lead to some extreme waits. Finally, transaction retries result in non-deterministic behaviors. Transactions will not necessarily be applied to the database in the order in which they are submitted by the application, and even under identical workloads, differences in outcomes will be observed.

The alternative to the transaction retry approach is to "lock" the rows required at the beginning of the transaction with a FOR UPDATE statement. FOR UPDATE is a blocking statement, and once it returns, your transaction has the update rights over the rows concerned.

Here's our sample code with the FOR UPDATE logic:

[source, javascript]

----

async function takeMeasurementForUpdate(locationId, measurement) {

let success = false;

const connection = await pool.connect();

const measurementTime = new Date();

try {

await connection.query('BEGIN TRANSACTION');

await connection.query(`SELECT id FROM locations

WHERE id=$1

FOR UPDATE`, [locationId]);

const insertReturn = await connection.query(

`INSERT INTO measurements

(locationId,measurement)

VALUES($1,$2) RETURNING (measurement\_timestamp)`,

[locationId, measurement]

);

await connection.query(`UPDATE locations

SET last\_measurement=$1, last\_timestamp=$2

WHERE id=$3`,

[measurement, measurementTime, locationId]);

await connection.query('COMMIT');

success = true;

} catch (error) {

console.error(error.message);

connection.query('ROLLBACK');

}

connection.release();

return (success);

}

----

By locking the LOCATIONS row as soon as the transaction begins, we avoid any chance of a transaction retry being issued. However, in a production implementation, it is probably advisable to include a transactionRetry error handler in any transaction.

.Optimistic vs. pessimistic transaction design

\*\*\*\*

The two patterns for transactions we've looked at here – retry handling versus FOR UPDATE locking – have historically been referred to as \*optimistic\* versus \*pessimistic\* transaction models.

In the optimistic transaction model, we feel it is unlikely that there will be a conflicting update that will cause a transaction to abort. Therefore, we don't "pre-lock" data and rely on transaction retries to handle any conflicts that might occur.

In the pessimistic model, we are quite worried about transaction conflicts, so we pre-emptively lock rows that might come into conflict.

Neither model is superior – it really does depend on how likely row-level transaction conflicts are. Don't choose one or the other based on your emotional disposition – think carefully about the likelihood of conflicts – benchmark if necessary - and act accordingly.

\*\*\*\*

### ==== Retry savepoints

<https://www.cockroachlabs.com/docs/stable/advanced-client-side-transaction-retries.html>

### ==== Time travel queries

### ==== Deadlocks

### ==== Nested transactions

### ==== Transaction priorities

### ==== Other options for avoiding conflicts

Both transaction retries and FOR UPDATE locking introduce delays into your application which are undesirable. A transaction retry burns up time during the rollback and sleep following a retry error, while FOR UPDATE will block other transactions that seek to update the contended resource.

The core problem to be avoided is contention on a single row. These "hot spots" should be avoided in database design. If the contention exists for multiple columns then placing each column in it's own \*column family\* for instance. We might also want to consider splitting up

* Keep transactions short
* Update highly contentious rows first
* Partition the data

==== tracking retries

### === Working with ORM Frameworks

While many applications work directly with CockroachDB via SQL statements embedded in application code, other applications use frameworks that avoid the direct use of SQL and instead leverage automated mapping of database tables to program objects. In this section, we'll introduce some of the most popular and provide an example of their use.

#### ==== SQLAlchemy for Python

#### ==== Django for Python

#### ==== Java Hibernate

#### ==== Java JOOQ

#### ==== GORM for GoLang

#### ==== TypeORM for NodeJS