[[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 PostreSQL 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 that 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 contrain our discussion to these four languages: Go, Java, Python and JavaScript.

In Chapter 3, we showed how to isntall 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 in guidelines, including guideance on how to install drivers for other languages or for alternative drivers.

### === Performing CRUD operations

We provided basic “Hello world” examples for each language back in Chapter32. Let’s extend those examples to perform some non-trivial “CRUD” operations – Create, Read, Update, Delete.

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

\* The driver establishes a \*connection\* object representing a connection to the database server.

\* 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 execte 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 an cursor object that can be used either 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 for a JavaScript program running in NodeJS:

[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();

----

A few things about the Javascript code above:

\* We’ve used the async/await style of 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 these programming styles.

\* Unlike previous examples, we’ve pulled all data from the query into an object rather than iterating through a cursor. See the section on Cursors below for a further discussion.

Finally, lets 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 repeditive error checking involved in the initial SQL statements, but generally we would perform error checking independently on 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.

Some languages – Java for instance – only support processing query output with cursors. The Java ResultSet class is an implementation of a cursor. However, other languages provide the ability to use a cursor, or to retrieve all results in one action.

For instance, lets 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 which 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 our python routine something like this (to display the first page):

[source, python]

----

cursor.execute("""SELECT post\_timestamp, summary

FROM blog\_posts ORDER BY post\_timestamp DESC """)

for x in range(0, 9):

row=cursor.fetchone()

print(row)

----

This implementation pulls the rows from the database one row at a time. However, it’s just as easy to use fetchall() and page through the results:

[source, python]

----

cursor.execute("""SELECT post\_timestamp, summary

FROM blog\_posts ORDER BY post\_timestamp DESC """)

rows=cursor.fetchall()

for x in range(0, 9):

print(rows[x])

----

The two implementations

**SELECT** measurement\_timestamp, measurement

**FROM** public.timeseries\_data **ORDER** **BY** measurement\_timestamp

#### ==== NodeJS

#### ==== GoLang

#### ==== Python

#### ==== Other languages

### === Coding Best Practices

In this section, we’ll look at how best to code around the various CockroachDB driver APIs. Regardless of what driver you are using, you’ll want to avoid unnecessary database requests, creating SQL injection vulnerabilities, reducing network round trips and in other ways, avoid inefficient interactions with the database.

#### ==== Parameterised statements

#### ==== Array processing

#### ==== Connection pools

#### ==== Projections

### === 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

### === Transaction Handling

CockroachDB is a transactional system – even if you don’t explicitly leverage transactions you will experience the effects of CockroachDB transaction handling.

In this section we’ll discuss how to program against the transactional system, and how to handle transaction retries and other transactional handling patterns.

#### ==== Optimistic vs Pessimistic strategy

#### ==== Automatic transaction retries

#### ==== Client side intervention

#### ==== Time travel queries

#### ==== Deadlocks

#### ==== Nested trasnactions

#### ==== Transaction priorities