[[Ch04 – CockroachDB SQL]]

**== CockroachDB SQL**

The language of CockroachDB is SQL. While there are some command line utilities, all interaction between an application and the database are mediated by SQL language commands.

SQL is a rich language with a long history – we touched upon some of that history in Chapter 1. A full definition of all SQL language features would require a book in its own right, and would be almost instantly out of date, since the SQL language evolves with each release.

Therefore, this chapter aims to provide you with a broad overview of the SQL language used in CockroachDB without attempting to be a complete reference. We’ll take a task oriented approach to the SQL language, covering the most common SQL language tasks with particular reference to unique features of the CockroachDB SQL implementation.

A complete reference for the CockroachDB SQL language can be found in the [CockroachDB documentation set](https://www.cockroachlabs.com/docs/stable/index.html) . A broader review of the SQL language can be found in the OReilly book “SQL in a Nutshell”. book “SQL in a Nutshell”.

We’ll mainly use the MOVR sample dataset in this chapter to illustrate various SQL language features. We learned how to install sample data in Chapter 2; to recap, we create the MOVR database by issuing the command + cockroach workload init movr+ command:

[source, bash]

----

$ cockroach workload init movr

I210510 01:31:40.067523 1 workload/workloadsql/dataload.go:146 [-] 1 imported users (0s, 50 rows)

I210510 01:31:40.084391 1 workload/workloadsql/dataload.go:146 [-] 2 imported vehicles (0s, 15 rows)

I210510 01:31:40.186933 1 workload/workloadsql/dataload.go:146 [-] 3 imported rides (0s, 500 rows)

I210510 01:31:40.269703 1 workload/workloadsql/dataload.go:146 [-] 4 imported vehicle\_location\_histories (0s, 1000 row

s)

I210510 01:31:40.355601 1 workload/workloadsql/dataload.go:146 [-] 5 imported promo\_codes (0s, 1000 rows)

I210510 01:31:40.372375 1 workload/workloadsql/workloadsql.go:113 [-] 6 starting 8 splits

I210510 01:31:40.489789 1 workload/workloadsql/workloadsql.go:113 [-] 7 starting 8 splits

I210510 01:31:40.587835 1 workload/workloadsql/workloadsql.go:113 [-] 8 starting 8 splits

----

# === SQL Language compatibility

CockroachDB is broadly compatible with the PostgreSQL implementation of the SQL:2003 standard. The SQL:2003 standard contains a number of independent modules and no major database implements all of the standard.

CockroachDB varies from PostgreSQL in a couple of areas:

* CockroachDB does not currently support stored procedures, events or triggers. In PostgreSQL, these stored procedures allow for the execution of program logic within the database server, either on-demand or in response to some triggering event.
* CockroachDB does not currently support User Defined Functions
* CockorachDB does not support PostgreSQL XML functions.
* CockroachDB does not support PostgreSQL FullText indexes and functions.

CockroachDB supports most of the PostgreSQL datatypes, however, CockroachDB generally does not distinguish between various scalar datatypes with different precision. For instance, in PostgreSQL, the datatypes smallint, integer, bigint and so on all have distinctly different storage within the database. In CockroachDB these types are all synonums for the INT type which can store up to a 64bit number.

# === querying data with SELECT

Although we need to create and populate tables before querying them, it is logical to start with the SELECT statement since many features of the SELECT statement appear in other types of SQL – subqueries in UPDATEs for instance – and for data scientists and analysts, the SELECT statement is often the only SQL statement they ever need to learn.

The SELECT statement is the workhorse of relational query and has a complex and rich syntax. The CockroachDB SELECT statement implements the standard features of the standard SELECT, with just a few CockroachDB-specific features.

[[SelectStatement]]

.Select Statement

image::images/SelectStatement.png[Delete Statement]

A screenshot of a computer

Description automatically generated with medium confidence

### ==== The SELECT list

The simplest possible SQL statement consists of nothing but a SELECT statement together with scalar expressions. For instance:

[source,sql]

----

**SELECT** **CONCAT**('Hello from CockroachDB at ',

**CAST** (**NOW**() **as** **STRING**)) **as** hello

----

The SELECT list includes a comma separated list of expressions that can contain combinations of constants, functions and operators. The CockroachDB SQL language supports all the familiar SQL operators. A complete list of functions and operators can be found at <https://www.cockroachlabs.com/docs/stable/functions-and-operators.html>

### ==== The FROM clause

The FROM clause is the primary method of attaching table data to the SELECT statement. In it’s most simple incarnation, all rows and columns from a table can be fetched:

[source,sql]

----

**SELECT** \* **FROM** rides

----

Table names may be aliased using the +AS+ clause or simply by following the table name with an alias. That alias can then be used anywhere in the query to refer to the table. For instance, the following are all equivalent:

[source, sql]

----

**SELECT** **name** **FROM** **users**;

**SELECT** u.**name** **FROM** **users** u;

**SELECT** **users**.**name** **AS** user\_name **FROM** **users**;

**SELECT** u.**name** **FROM** **users** **AS** u;

----

==== JOINS

Joins allow the results from two or more tables to be merged based on some common column values.

The \*inner join\* is the most common type and default join operation. In this join, rows from one table are joined to rows from another table based on some common (“key”) values. Rows which have no match in the both tables are not included in the results. For instance, the following query links vehicle and ride information in the MOVR database:

[source, sql]

----

SELECT v.id,v.ext,r.start\_time r.start\_address

FROM vehicles v

LEFT OUTER JOIN rides r

ON (r.vehicle\_id=v.id);

----

Note that a department which had not been involved in a ride would not be included in the result set.

The ON clause specifies the conditions which join the two tables – in the above query the columns +vehicle\_id+ in the +rider+ table was matched with the +id+ column in the vehicles table. If the join is on an identically named column in both tables, then the USING clause provides a handy short cut. Here we join +users+ and +user\_ride\_counts+ using the common +name+ column:

[source,sql]

----

SELECT \*

FROM users u

JOIN user\_ride\_counts urc

USING (name)

----

The \*Outer join\* allows rows to be included even if they have no match in the other table. Rows which are not found in the outer join table are represented by NULL values. LEFT+and RIGHT determine which table may have missing values. For instance, the following query prints all the users in the +users+ table, even if some are not associated with a promo code:

[source, sql]

----

SELECT u.name , upc.code

FROM USERS u

LEFT OUTER JOIN user\_promo\_codes upc

ON (u.id=upc.user\_id);

----

The RIGHT anti-join reverses the outer join. So this query is identical to the previous query, since the +users+ table is now the “right” table in the join:

[source,sql]

----

**SELECT** **DISTINCT** u.**name** , upc.code

**FROM** user\_promo\_codes upc

**RIGHT** **OUTER** **JOIN** **USERS** u

**ON** (u.id=upc.user\_id);

----

### ==== Anti-joins

It is often required to select all rows from a table which do not have a matching row in some other result set. This is typically implemented using a subquery and the IN or EXISTS clause. The following examples illustrate the anti-join using the EXISTS and IN operators.

Each example selects employees who are not also customers.

[source, sql]

----

**SELECT** \*

**FROM** **users**

**WHERE** id **NOT** **IN**

(**SELECT** id **FROM** employees)

----

This query returns the same results, but using a correlated sub-query:

[source, sql]

----

**SELECT** \*

**FROM** **users** u

**WHERE** **NOT** **EXISTS**

(**SELECT** id

**FROM** employees e

**WHERE** e.id=u.id)

----

### ==== CROSS JOINS

CROSS JOIN indicates that every row in the left table should be joined to every row in the right table. Usually, this is a recipie for disaster, unless the one of the table has only one row or is a laterally correlated subquery (see the section on correlated subqueries later in this chapter).

### ==== Set operations

SQL implements a number of operations that deal directly with result sets. These operations, collectively referred to as “set operations” allow result sets to be concatenated, subtracted or overlaid.

The most common of these operations is the UNION operator, which returns the sum of two result sets. By default, duplicates in each result set are eliminated. By contrast the UNION ALL operation will return the sum of the two result sets, including any duplicates. The following example returns a list of customers and employees. Employees who are also customers will only be listed once:

[source, sql]

----

**SELECT** **name**, address

**FROM** customers

**UNION**

**SELECT** **name**,address

**FROM** employees;

----

INTERSECT returns those rows which are in both result sets. This query returns customers who are also employees:

[source, sql]

----

**SELECT** **name**, address

**FROM** customers

**INTERSECT**

**SELECT** **name**,address

**FROM** employees;

----

EXCEPT returns rows in the first result set which are not present in the second. This query returns customers who are not also employees:

[source, sql]

----

SELECT name, address

FROM customers

EXCEPT

SELECT name,address

FROM employees;

----

All set operations require that the component queries return the same number of columns, and that those columns are of a compatible datatype.

### ==== Group operations

Aggregate operations allow for summary information to be generated, typically upon groupings of rows. Rows can be grouped using the GROUP BY operator. If this is done, the select list must consist only of columns contained within the GROUP BY clause and aggregate functions.

The most common aggregate functions are shows in <<AggregateFunctions>>.

[[AggregateFunctions]]

.Aggregate Functions

|=======

|AVG | Calculate the average value for the group.

|COUNT| Return the number of rows in the group.

|MAX | Return the maximum value in the group.

|MIN | Return the minimum value in the group.

|STDDEV| Return the standard deviation for the group.

|SUM | -Return the total of all values for the group.

|=======

The following example generates summary salary information for each department:

[source,sql]

----

**SELECT** u.city,**SUM**(urc.rides),**AVG**(urc.rides),**max**(urc.rides)

**FROM** **users** u

**JOIN** user\_ride\_counts urc

**USING** (**name**)

**GROUP** **BY** u.city

----

### ==== Subqueries

A subquery is a SELECT statement which occurs within another SQL statement. Such a “nested” SELECT statement can be used in a wide variety of SQL contexts, including SELECT, DELETE, UPDATE and INSERT statements.

The following statement uses a subquery to count the number of rides who share the maxium ride length:

[source,sql]

----

SELECT COUNT(\*) FROM rides

WHERE (end\_time-start\_time)=

(SELECT MAX(end\_time-start\_time) FROM rides );

----

Subqueries may also be used in the FROM clause wherever a table or view definition could appear. This query generates a result which compares each ride with the average ride duration for the city:

[source, sql]

----

SELECT id, city,(end\_time-start\_time) ride\_duration, avg\_ride\_duration

FROM rides

JOIN (SELECT city, AVG(end\_time-start\_time) avg\_ride\_duration

FROM rides

GROUP BY city)

USING(city) ;

----

### ==== Correlated subquery

A correlated subquery is one in which the subquery refers to values in the parent query or operation. The subquery returns a potentially different result for each row in the parent result set. We saw an example of a correlated sub-query when performing an “anti-join” earlier in the chapter.

[source, sql]

----

**SELECT** \*

**FROM** **users** u

**WHERE** **NOT** **EXISTS**

(**SELECT** id

**FROM** employees e

**WHERE** e.id=u.id)

----

### ==== Lateral subquery

When a subquery is used in a join, the LATERAL keyword indicates that the subquery may access columns generated in preceding FROMtable expressions. For instance, in the following query, the +LATERAL+ keyword allows the subquery to access columns from the +users+ table:

[source, sql]

----

**SELECT** **name**, address, start\_time

**FROM** **USERS** **CROSS** **JOIN**

**LATERAL** (**SELECT** \*

**FROM** rides

**WHERE** rides.start\_address = **users**.address ) r;

----

This example is a bit contrived, and clearly, we could construct a simple join that perfomed this query more naturally. Where +LATERAL+ joins really shine is in allowing subqueries to access computed columns in other subqueres within a FROM clause. See <https://www.cockroachlabs.com/blog/using-lateral-joins-in-the-cockroachdb-20-1-alpha/> for a more serious example of lateral subqueries.

### ==== Common Table expressions

SQL statements with a lot of subquries can be hard to read and maintain, especially if the same subquery is needed in multiple contexts within the query. For this reason, SQL supports \*Common Table Expressions\* using the WITH clause. <<CommonTableExpression>> shows the syntax of a Common Table expression.

[[CommonTableExpression]]

.Common Table Expression

image::images/common\_table\_expression.png[Common Table Expression]

Graphical user interface, application, Teams

Description automatically generated

In it’s simplest form, a Common Table Expression is simply a named query block that can be used wherever a table expression can be used. For instance, here we create a Common Table Expression +riderRevenue+ with the WITH clause, then refer to it in the FROM clause of the main query:

[source,sql]

----

**WITH** riderRevenue **AS** (

**SELECT** u.id, **SUM**(r.revenue) **AS** sumRevenue

**FROM** rides r **JOIN** "users" u

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

**GROUP** **BY** u.id)

**SELECT** \* **FROM** "users" u2

**JOIN** riderRevenue rr **USING** (id)

**ORDER** **BY** sumrevenue **DESC**

**----**

The RECURSIVE clause allows the Common Table Expression to refer to itself potentially allowing for a query to return an abirtrarily high (or even infinite) set of results. For instance this query returns the factorial of numbers 0 to 9:

[source,sql]

----

WITH RECURSIVE cte (n, factorial) AS (

VALUES (0, 1) -- initial subquery

UNION ALL

SELECT n+1, (n+1)\*factorial FROM cte WHERE n < 9 -- recursive subquery

)

SELECT \* FROM cte;

----

The MATERIALIZED clause forces CockroachDB to store the results of the Common Table Experssion as a temporary table rather than re-executing it on each occurance. This can be useful if the Common Table Expression is refered to multiple times in the query.

### ==== ORDER BY

The ORDER BY clause allows query results to be returned in a sorted order. <<OrderBy>> shows the ORDER BY syntax

[[OrderBy]]

.Order By

image::images/order\_by.png[OrderBy]

Graphical user interface

Description automatically generated

In the simplest form, ORDER BY takes one or more column expressions or column numbers from the SELECT list:

In this example we sort by column numbers:

[source,sql]

----

**SELECT** city,start\_time, (end\_time-start\_time) duration

**FROM** rides r

**ORDER** **BY** 1,3 **DESC**

**----**

And in this case by column expressions:

[source,sql]

----

**SELECT** city,start\_time, (end\_time-start\_time) duration

**FROM** rides r

**ORDER** **BY** city,(end\_time-start\_time) **DESC**

**----**

You can also order by an index. In the following example, rows will be orderd by city and start\_time, since those are the columns specified in the index:

[source,sql]

----

**CREATE** **INDEX** rides\_start\_time **ON** rides (city ,start\_time);

**SELECT** city,start\_time, (end\_time-start\_time) duration

**FROM** rides

**ORDER** **BY** **INDEX** rides@rides\_start\_time;

----

The use of ORDER BY INDEX guarantees that the index will be used to directly return rows in sorted order, rather than having to perform a sort operation on rows returned otherwise. See Chapter 8 for more advice on optimizing statements that contain an ORDER BY

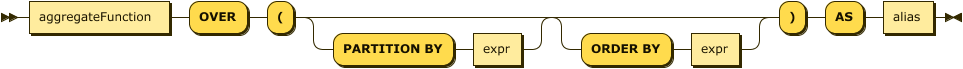
### ==== WINDOW FUNCTIONS

Window functions are functions that operate over a subset – a “window” of the complete set of the results.

[[WindowFunction]]

.Order By

image::images/window\_function.png[WindowFunction]



PARTITION BY and ORDER BY create a sort of “virtual table” that the function works with. For instance, in this query lists the top ten rides in terms of revenue, with the percentage of the total revenue and city revenue displayed:

[source, sql]

----

**SELECT** city, r.start\_time ,revenue,

revenue\*100/**SUM**(revenue) **OVER** () **AS** pct\_total\_revenue,

revenue\*100/**SUM**(revenue) **OVER** (**PARTITION** **BY** city) **AS** pct\_city\_revenue

**FROM** rides r

**ORDER** **BY** 5 **DESC**

**LIMIT** 10

----

There are some aggregation functions that are specific to Windowing functions. RANK() ranks the existing row within the relevant window and DENSE\_RANK() does the same while allowing no “missing” ranks. LEAD and LAG provide access to functions in adjacent partitions.

For instance, here we print the top 10 rides, with their overall rank and rank within their city displayed:

[source,sql]

----

**SELECT** city, r.start\_time ,revenue,

**RANK**() **OVER**

(**ORDER** **BY** revenue **DESC**) **AS** total\_revenue\_rank,

**RANK**() **OVER**

(**PARTITION** **BY** city **ORDER** **BY** revenue **DESC**) **AS** city\_revenue\_rank

**FROM** rides r

**ORDER** **BY** revenue **DESC**

**LIMIT** 10;

----

==== Other SELECT clauses

The LIMIT clause limits the number of rows returned by a SELECT while the OFFSET clauses “jumps ahead” a certain number of rows. This can be handy to paginate through a result set though it is almost always more efficient to use a filter condition to navigate to the next subset of results, since otherwise each request will need to reread and discard an increasing number of rows.

# === Creating tables and indexes

In a relational database, data can only be added to pre-defined tables. These tables are created by the CREATE TABLE statement. Indexes can be created to enforce unique constraints or to provide a fast access path to the data. Indexes can be defined within the CREATE TABLE statement or by a separate CREATE INDEX statement.

The structure of a database schema forms a critical constraint on database performance, and also on the maintainability and utility of the database. We’ll discuss the key considerations for database design in Chapter 5. For now, let’s create a few simple tables.

A very simple CREATE TABLE is shown below. It creates a table +mytable+ with a single column +mycolumn+. The +mycolumn+ column can store only integer values.

[source, sql]

----

CREATE TABLE mytable

(

mycolumn int

)

----

We use CREATE TABLE to create a table within a database. The CREATE TABLE statement must define the columns that occur within the table, and can optionally define indexes, column families , contraints, and partitions associated with the table. For instance, the CREATE TABLE statement for the +rides+ table in the +movr+ database would look something like this:

[source, sql]

----

**CREATE** **TABLE** public.rides (

id UUID **NOT** **NULL**,

city **VARCHAR** **NOT** **NULL**,

vehicle\_city **VARCHAR** **NULL**,

rider\_id UUID **NULL**,

vehicle\_id UUID **NULL**,

start\_address **VARCHAR** **NULL**,

end\_address **VARCHAR** **NULL**,

start\_time **TIMESTAMP** **NULL**,

end\_time **TIMESTAMP** **NULL**,

revenue **DECIMAL**(10,2) **NULL**,

**CONSTRAINT** "primary" **PRIMARY** **KEY** (city **ASC**, id **ASC**),

**CONSTRAINT** fk\_city\_ref\_users

**FOREIGN** **KEY** (city, rider\_id)

**REFERENCES** public.**users**(city, id),

**CONSTRAINT** fk\_vehicle\_city\_ref\_vehicles

**FOREIGN** **KEY** (vehicle\_city, vehicle\_id)

**REFERENCES** public.vehicles(city, id),

**INDEX** rides\_auto\_index\_fk\_city\_ref\_users

(city **ASC**, rider\_id **ASC**),

**INDEX** rides\_auto\_index\_fk\_vehicle\_city\_ref\_vehicles

vehicle\_city **ASC**, vehicle\_id **ASC**),

**CONSTRAINT** check\_vehicle\_city\_city

**CHECK** (vehicle\_city = city)

);

----

This CREATE TABLE statement specified additional columns, their nullability, primary and foreign keys, indexes and contraints upon table values.

<<CreateTableReference>> summarizes the syntax of the CREATE TABLE statement.

Graphical user interface, text, application

Description automatically generated

[[CreateTableReference]]

.Create Table Statement

image::images/CreateTableReference.png[Create Table Statement

The eelevent clauses in <<CreateTableReference>> are listed in <<CreateTableOptions>>.

[[CreateTableOptions]]

.Create Table options

|=======

|IF NOT EXISTS |Specifies that the command should fail silently if the table already exists.

|column\_def |The definition of a column. This includes the column name, datatype, nullability. Constraints specific to the column can also be included here, though it’s better practice to list all contraints separately.

|Index\_def |Defnition of an index to be created on the table. Same as CREATE INDEX but without the leading +CREATE+ verb.

|table\_contraint | a constraint on the table, such as PRIMARY KEY, FOREIGN KEY or CHECK. See below for constraint syntax.

|LIKE |The Like statement causes a table to be created with the same definition as the referenced table name

|=======

### ===== Column Definitions

A column definition consists of a column name, datatype, nullability status, default value and possibly column level constraint definitions. At a minimum, the name and datatype must be specified:

[source, sql]

----

column\_name datatype

[default default\_value] [not null]

[PRIMARY KEY|FOREIGN KEY REFERENCES tables(columns)]

[CHECK *check\_contraint*]

----

Although constraints may be specified directly against column definitions, they may also be independently listed below the column definitions. Many practitioners prefer to list the constraints separately in this manner because it allows all contraints, including multi-column constraints, to be located together.

### ===== Computed Columns

CockroachDB allows tables to include \*computed columns\* that in some other databases would require a view definition.

[source, sql]

----

column\_name AS expression [STORED]

----

For instance, this table definition has the firstName and lastName concatenated into a fullName column:

[source,sql]

----

CREATE TABLE people

(

id INT PRIMARY KEY,

firstName VARCHAR NOT NULL,

lastName VARCHAR NOT NULL,

dateOfBirth DATE NOT NULL,

fullName STRING AS (CONCAT(firstName,' ',lastName) ) STORED,

)

----

Computed columns cannot be context-dependent. That is to say, the computed value must not change over time or be otherwise non-deterministic. For instance, the computed column would not work, since the +age+ column would be static rather than recalculated every time. While it might be nice to stop ageing in real life, we probably want the +age+ column to increase as time goes on.

[source, sql]

----

**CREATE** **TABLE** people

(

id **INT** **PRIMARY** **KEY**,

firstName **VARCHAR** **NOT** **NULL**,

lastName **VARCHAR** **NOT** **NULL**,

dateOfBirth **timestamp** **NOT** **NULL**,

fullName **STRING** **AS** (**CONCAT**(firstName,' ',lastName) ) **STORED**,

**age** **int** **AS** (**now**()-dateOfBirth) **STORED**

);

----

### ===== Datatypes

The base CockroachDB datatypes are shown in <<CockroachDBDatatypes>>

. See <https://www.cockroachlabs.com/docs/stable/data-types.html> for more details.

[[CockroachDBDatatypes]]

.CockroachDBDatatypes

[options="header"]

|=======

|Type|Description|Example

|ARRAY|A 1-dimensional, 1-indexed, homogeneous array of any non-array data type.|{"sky","road","car"}

|BIT|A string of binary digits (bits).|B'10010101'

|BOOL|A Boolean value.|true

|BYTES|A string of binary characters.|b'\141\061\142\062\143\063'

|COLLATE|The COLLATE feature lets you sort STRING values according to language- and country-specific rules, known as collations.|'a1b2c3' COLLATE en

|DATE|A date.|DATE '2016-01-25'

|ENUM|New in v20.2: A user-defined data type comprised of a set of static values.|ENUM ('club, 'diamond', 'heart', 'spade')

|DECIMAL|An exact, fixed-point number.|1.2345

|FLOAT|A 64-bit, inexact, floating-point number.|1.2345

|INET|An IPv4 or IPv6 address.|192.168.0.1

|INT|A signed integer, up to 64 bits.|12345

|INTERVAL|A span of time.|INTERVAL '2h30m30s'

|JSONB|JSON (JavaScript Object Notation) data.|'{"first\_name": "Lola", "last\_name": "Dog", "location": "NYC", "online" : true, "friends" : 547}'

|SERIAL|A pseudo-type that combines an integer type with a DEFAULT expression.|148591304110702593

|STRING|A string of Unicode characters.|'a1b2c3'

|TIME

TIMETZ|TIME stores a time of day in UTC.

TIMETZ converts TIME values with a specified time zone offset from UTC.|TIME '01:23:45.123456'

TIMETZ '01:23:45.123456-5:00'

|TIMESTAMP

TIMESTAMPTZ|TIMESTAMP stores a date and time pairing in UTC.

TIMESTAMPTZ converts TIMESTAMP values with a specified time zone offset from UTC.|TIMESTAMP '2016-01-25 10:10:10'

TIMESTAMPTZ '2016-01-25 10:10:10-05:00'

|UUID|A 128-bit hexadecimal value.|7f9c24e8-3b12-4fef-91e0-56a2d5a246ec

|=======

Note that other datatype names may be aliased against these CockroachDB base types. For instance the PostgreSQL types BIGINT and SMALLINT are aliased against the CockroachDB type INT.

In CockroachDB datatypes may be cast – or converted – by appending the datatype to an expression using “::”. For instance:

[source,sql]

----

**SELECT** revenue::**int** **FROM** rides

----

### ===== Primary Keys

As we know, a primary key uniquely defines a row within a table. In CockroachDB, a primary key is mandatory since all tables are distributed across the cluster based on ranges of their primary key. If you don’t specify a primary key a key will be automatically generated for you.

It’s common practice to define a auto-generating primary key using clauses such as AUTOINCREMENT. The generation of primary keys in distributed databases is a significant issue, since it’s the primary key that is used to distribute data across nodes in the cluster. We’ll discuss the options for primary key generation in the next chapter but for now, we’ll simply note that you can generate a randomized primary key values using the +UUID+ datatype with the +gen\_random\_uuid()+ function as the default value:

[source,sql]

----

CREATE TABLE people (

id UUID NOT NULL DEFAULT gen\_random\_uuid(),

firstName VARCHAR NOT NULL,

lastName VARCHAR NOT NULL,

dateOfBirth DATE NOT NULL

);

----

This pattern is considered best practice to ensure even distribution of keys across the cluster. Other options for autogenerating primary keys will be discussed in Chapter 5.

### ===== Constraints

The +CONSTRAINT+ clause specifies conditions which must be satisfied by all rows within a table. In some circumstances, the +CONSTRAINT+ keyword may be omitted, for instance when defining a column constraint or specific constraint types such PRIMARY KEY or FOREIGN KEY. <<ConstraintReference>> shows the general form of a constraint definition.

[[ConstraintReference]]

.Contraint Statement

image::images/tableConstraint.png[Constraint Statement]

A picture containing text, black, dark, close

Description automatically generated

A UNIQUE constraint requires that all values for the column or column\_list be unique.

PRIMARY KEY implements a set of columns which must be unique and which can also be the subject of a FOREIGN KEY constraint in another table. Both PRIMARY KEY and UNIQUE constraints require the creation of an implicit index. If desired, physical storage characteristics of the index can be specified in the +USING+. The options of the USING INDEX clause have the same usages as in the CREATE INDEX statement.

NOT NULL indicates that the column in question may not be NULL. This option is only available for column constraints, but the same effect can be obtained with a table +CHECK+ constraint.

CHECK defines an expression which must evaluate to true for every row in the table.

We’ll discuss best practies for creating constraints in Chapter 5.

Sensible definition of contraints can help ensure data quality, and can provide the database with a certain degree of self-documentation. However, some constraints have significant performance implications, we’ll discuss these implications in Chapter 5.

I

## === Indexes

Indexes can be created by the CREATE INDEX statement, or an INDEX definition can be included within the CREATE TABLE statement.

We talked a lot about indexes in Chapter 2, and we’ll keep discussing indexes in the schema design and performance tuning chapters. Effective indexing is one of the most important success factors for a performance CockroachDB implementation.

<<CreateIndexStatement>> illustrates a simplistic syntax for the CockroachDB CREATE INDEX statement.

A screenshot of a computer

Description automatically generated with low confidence

[[CreateIndexStatement]]

.Create Index Statement

image::images/createIndex.png[Create Index Statement]

We looked at the internals of CockroachDB indexes in Chapter 2. From a performance point of view, CockroachDB indexes behave much as indexes in other databases – providing a fast access method for locating rows with a particular set of non-primary key values. For instance, if we simply want to locate a row name and date of birth we might create the following multi-column index:

[source, sql]

----

**CREATE** **INDEX** people\_namedob\_ix **ON** people

(lastName,firstName,dateOfBirth);

----

If we furthermore wanted to ensure that no two rows could have the same value for name and date of birth we might create a unique index:

[source,sql]

----

**CREATE** **UNIQUE** **INDEX** people\_namedob\_ix **ON** people

(lastName,firstName,dateOfBirth);

----

The STORING clause allows us to store additional data in the index which can allow us to satisfy queries using the index alone. For instance, this index can satisfy queries that retrieve phone numbers for a given name and date of birth:

[source, sql]

----

**CREATE** **UNIQUE** **INDEX** people\_namedob\_ix **ON** people

(lastName,firstName,dateOfBirth) **STORING** (phoneNumber);

----

### ==== Inverted Indexes

An inverted index can be used to index the elements within an array, or the attributes within a JSON document. We looked at the internals of inverted indexes in Chapter 2.

For example, suppose our +people+ table used a JSON document to store the attributes for a person:

[source,sql]

----

CREATE TABLE people

( id UUID NOT NULL DEFAULT gen\_random\_uuid(),

personData JSONB );

INSERT INTO people (personData)

VALUES('{

"firstName":"Guy",

"lastName":"Harrison",

"dob":"21-Jun-1960",

"phone":"0419533988",

"photo":"eyJhbGciOiJIUzI1NiIsI..."

}');

----

We might create an inverted index as follows:

[source,sql]

----

**CREATE** **INVERTED** **INDEX** people\_inv\_idx **ON**

people(personData);

----

Which would support queries into the JSON document such as this:

[source,sql]

----

**SELECT** \*

**FROM** people

**WHERE** personData @> '{"phone":"0419533988"}';

----

However, inverted indexes index every attribute in the JSON document, not just those that you want to search on. Therefore, you might find it more useful to create a calculated column on the JSON attribute and then index on that computed column:

[source, sql]

----

ALTER TABLE people ADD phone STRING AS (personData->>'phone') STORED;

CREATE INDEX people\_phone\_idx ON people(phone);

----

### ==== Hash sharded indexes

If you are working with a table that must be indexed on sequential keys, you should use hash-sharded indexes. Hash-sharded indexes distribute sequential traffic uniformly across ranges, eliminating single-range hotspots and improving write performance on sequentially-keyed indexes at a small cost to read performance

[source,sql]

----

**CREATE** **TABLE** people

( id **INT** **PRIMARY** **KEY**,

firstName **VARCHAR** **NOT** **NULL**,

lastName **VARCHAR** **NOT** **NULL**,

dateOfBirth **timestamp** **NOT** **NULL**,

phoneNumber **VARCHAR** **NOT** **NULL**,

serialNo SERIAL ,

**INDEX** serialNo\_idx (serialNo) **USING** **HASH** **WITH** **BUCKET\_COUNT**=4);

----

## === Create table AS select

The +AS select+ clause of CREATE TABLE allows us to create a new table which has the data and attributes of a SQL SELECT statement. Columns, contraints and indexes can be specified as for an existing table, but must align with the data types and number of columns returned by the SELECT statement. For instance, here we create a table based on a join and aggregate of two tables in the MOVR database:

[source, sql]

----

**CREATE** **TABLE** user\_ride\_counts **AS**

**SELECT** u.**name**, **COUNT**(u.**name**) **AS** rides

**FROM** "users" **AS** u **JOIN** "rides" **AS** r

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

**GROUP** **BY** u.**name**

**----**

## === ALTERING Tables

The +ALTER TABLE+ statement allows table columns or contraints to be added, modified renamed or removed, as well as allowing for constraint validation and partitioning. <<AlterTable>> shows the syntax

[[AlterTable]]

.Alter Table Statement

image::images/alterTable.png[Alter Table Statement]

A close-up of a calculator

Description automatically generated with medium confidence

Altering table structures online is not something to be undertaken lightly, although CockroachDB provides a highly advanced mechanisms for propagating such changes without impacting availability and with minimal impact on performancefootnote:[See https://www.cockroachlabs.com/docs/stable/online-schema-changes]

## DROPPING tables

# Inserting data

We can load data into a new table using the +CREATE TABLE AS select+ statement discussed earlier, using the INSERT statement inside a program or from the command line shell, or by loading external data using the +IMPORT+ statement. There are also non-SQL utilities that insert data – we’ll look at these in Chapter 7.

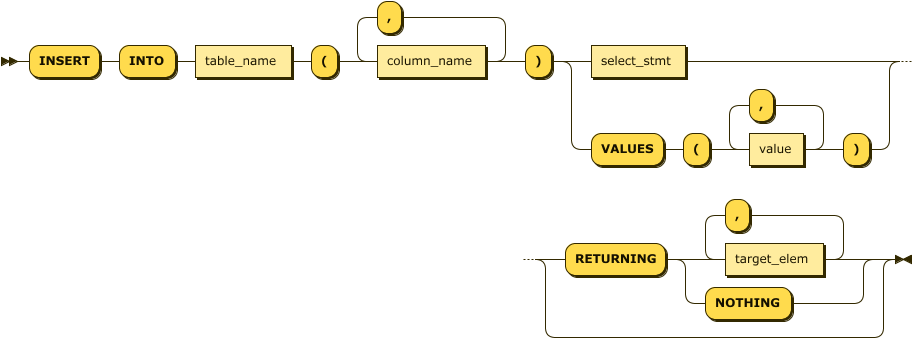
## The Insert statement

The venerable INSERT statement adds data to an existing table. <<InsertStatement>> illustrates a simplified syntax for the INSERT statement.

[[InsertStatement]]

.Insert Statement

image::images/insertStatement.png[Insert Statement]



INSERT takes either a set of values or a select statement. For instance, in the following example we insert a single row into the +people+ table:

[source, sql]

----

**INSERT** **INTO** people (firstName, lastName, dateOfBirth)

**VALUES**('Guy', 'Harrison', '21-JUN-1960');

----

The VALUES clause of the INSERT statement can accept array values, inserting more than one row in a single execution.

[source, sql]

----

**INSERT** **INTO** people (firstName, lastName, dateOfBirth)

**VALUES** ('Guy', 'Harrison', '21-JUN-1960'),

('Michael', 'Harrison', '19-APR-1994'),

('Oriana', 'Harrison', '18-JUN-2020');

----

There are alternative ways to insert batches in the various program language drivers and we’ll show some examples in Chapter 7.

A SELECT statement can be specified as the source of the inserted data:

[source, sql]

----

**INSERT** **INTO** people (firstName, lastName, dateOfBirth)

**SELECT** firstName, lastName, dateOfBirth

**FROM** peopleStagingData;

----

## IMPORT/IMPORT INTO

The IMPORT statement imports the following types of data into CockroachDB:

* Avro
* CSV/TSV
* Postgres dump files
* MySQL dump files
* CockroachDB dump files

IMPORT creates a new table, while +IMPORT INTO+ allows an import into an existing table.

The files to be imported should exist either in a cloud storage bucket – Google Cloud Storage, Amazon S3 or Azure Blob storage – from a HTTP address or from the local filesystem (“nodelocal”).

We’ll discuss the various options for loading data into CockroachDB in Chapter 7. However, for now, let’s create a new table +CUSTOMERS+ from a CSV file:

[source, sql]

----

root@localhost:26257/defaultdb> IMPORT TABLE customers (

id INT PRIMARY KEY,

name STRING,

INDEX name\_idx (name)

)

CSV DATA ('nodelocal://1/customers.csv');

job\_id | status | fraction\_completed | rows | index\_entries | bytes

---------------------+-----------+--------------------+------+---------------+--------

659162639684534273 | succeeded | 1 | 1 | 1 | 47

(1 row)

Time: 934ms total (execution 933ms / network 1ms)

----

For a single node demo cluster, the +nodelocal+ location will be somewhat dependent on your installation, but will often be in an +extern+ directory beneath the CockroachDB installation directory.

# === modifying data with UPDATE

The UPDATE statement changes existing data in a table.

<<UpdateStatement>> shows a simplified syntax for the UPDATE statement.

[[UpdateStatement]]

.Update Statement

image::images/UpdateStatement.png[Update Statement]

A screenshot of a computer

Description automatically generated with low confidence

### FROM clause

### RETURNING

## UPSERT

# === DELETE

DELETE allows data to be removed from a table. <<DeleteStatement>> shows a simplified syntax for the DELETE statement.

[[DeleteStatement]]

.Delete Statement

image::images/DeleteStatement.png[Delete Statement]

Graphical user interface, text, application

Description automatically generated

Most of the time, a DELETE statement accepts a +WHERE+ clause and not much else. For instance, here we delete a single row in the +people+ table:

[source, sql]

----

**DELETE** **FROM** people

**WHERE** firstName='Guy'

**AND** lastName='Harrison';

----

## TRUNCATE

# Transactions

## BEGIN

## COMMIT/ROLLBACK

## SAVEPOINTS

## Select for Update

## As of system time

# Advanced DDL

In the next chapter, we’ll look at some of the considerations for data modelling and schema design. Some of the following commands are more relevant in that context. However, let’s look at some of the more advanced commands that we can use to define data and indexing.

## Schemas

## Sequences

## Change feeds

## Partitions

## TYPES

# administrative commands

CockroachDB supports commands to maintaine authentication of users and their authorities to perform database operations. It also has a job scheduler that can be used to schedule backup and restore and timed Data definition changes. Other commands support the maintenance the cluster topology

## Cluster maintanence

### Configure zone

### SET CLUSTER SETTING

### SET LOCALITY

### SHOW RANGES

### SPLIT AT

### SURVIVE

## Scheduling and backups

### Backup

### create schedule for backup

### SHOW/CANCEL/PAUSE JOBS

## Security

### create/alter/drop user

### create/alter/drop role

## General Administration

### Show statistics

### SHOW TRACE FOR SESSION

### SHOW TRANSACTIONS

### SHOW/CANCEL SESSION

### EXPLAIN

# Summary