**https://leanpub.com/aprimeronsql/read**

**Some of The Most Important SQL Commands**

* **SELECT** - extracts data from a database
* **UPDATE** - updates data in a database
* **DELETE** - deletes data from a database
* **INSERT INTO** - inserts new data into a database
* **CREATE DATABASE** - creates a new database
* **ALTER DATABASE** - modifies a database
* **CREATE TABLE** - creates a new table
* **ALTER TABLE** - modifies a table
* **DROP TABLE** - deletes a table
* **CREATE INDEX** - creates an index (search key)
* **DROP INDEX** - deletes an index

**C:\Users\compaq>sqlite3**

SQLite version 3.13.0 2016-05-18 10:57:30

Enter ".help" for usage hints.

Connected to a transient in-memory database.

Use ".open FILENAME" to reopen on a persistent database. Use ‘.tables’ to see list of all tables for a given database.

sqlite> .open xyz.db

sqlite> .mode column

sqlite> .headers on

sqlite> CREATE TABLE mydata (

...> id INTEGER,

...> language VARCHAR(20),

...> author VARCHAR(25),

...> year INTEGER);

sqlite> INSERT INTO mydata VALUES (1, 'Fortran', 'Backus', 1955);

sqlite> INSERT INTO mydata VALUES (2, 'Lisp', 'McCarthy', 1958);

sqlite> INSERT INTO mydata VALUES (3, 'Cobol', 'Hopper', 1959);

sqlite> SELECT \* FROM mydata;

1|Fortran|Backus|1955

2|Lisp|McCarthy|1958

3|Cobol|Hopper|1959

**Constraints –**

sqlite> CREATE TABLE mydatacopy (

...> id Integer NOT NULL,

...> language VARCHAR(20) NOT NULL,

...> author VARCHAR(25) NOT NULL,

...> year INTEGER NOT NULL,

...> standard VARCHAR(10) NULL);

**Selective fields INSERT –** Unlike earlier,inserting data in arbitrary order –

sqlite> INSERT INTO mydatacopy (id, language, author, year, standard)

...> VALUES (1, 'prolog', 'Colmerauer', '1972', 'ISO');

sqlite> INSERT INTO mydatacopy (id, language, author, year)

...> VALUES (2, 'Perl', 'Wall', '1987');

sqlite> INSERT INTO mydatacopy (id, year, standard,language, author)

...> VALUES (3, '1964', 'ANSI', 'APL', 'Iverson');

sqlite> SELECT \* FROM mydatacopy;

1|prolog|Colmerauer|1972|ISO

2|Perl|Wall|1987|

3|APL|Iverson|1964|ANSI

**Primary key constraint –** forces fields to be unique

sqlite> CREATE TABLE mydatacopy2 (

...> id Integer NOT NULL PRIMARY KEY,

...> language VARCHAR(20) NOT NULL,

...> author VARCHAR(25) NOT NULL,

...> year INTEGER NOT NULL,

...> standard VARCHAR(10) NULL);

**Unique key constraint -** A **unique key** like a primary key is also used to make each record inside a table unique. Once you have defined the primary key of a table, any other fields you wish to make unique is done through this constraint. For example, in our database it now makes sense to have a unique key constraint on the language field. This would ensure none of the records would duplicate information about the same programming language.

sqlite> CREATE TABLE mydatacopy3 (

...> id Integer NOT NULL PRIMARY KEY,

...> language VARCHAR(20) NOT NULL UNIQUE,

...> author VARCHAR(25) NOT NULL,

...> year INTEGER NOT NULL,

...> standard VARCHAR(10) NULL);

### Differences between a Primary Key and a Unique Key

1. A primary key field cannot take on a NULL value, whereas a field with a unique constraint can. However, there can be only one such record since each value must be unique due to the very definition of the constraint.
2. You are allowed to define only one primary key constraint but you can apply the unique constraint to as many fields as you like.

**Dropping Tables** – We created few tables above. We’ll keep the last one and delete others.

sqlite> DROP TABLE mydata;

sqlite> DROP TABLE mydatacopy;

sqlite> DROP TABLE mydatacopy2;

**Creating New Tables from Existing Tables –** Now we have only mydatacopy3 table which has no records as yet. Let’s put some records –

sqlite> INSERT INTO mydatacopy3 (id, language, author, year, standard)

...> VALUES (1, 'Prolog', 'Colmerauer', '1972', 'ISO');

sqlite> INSERT INTO mydatacopy3 (id, language, author, year)

...> VALUES (2, 'Perl', 'Wall', '1987');

sqlite> INSERT INTO mydatacopy3 (id, year, standard, language, author)

...> VALUES (3, '1964', 'ANSI', 'APL', 'Iverson');

sqlite> CREATE TABLE mydata AS SELECT \* FROM mydatacopy3

...> ;

sqlite> SELECT \* FROM mydata;

1|Prolog|Colmerauer|1972|ISO

2|Perl|Wall|1987|

3|APL|Iverson|1964|ANSI

**Modifiying Tables –**

ALTER TABLE mydata ALTER author varchar(30)

Above command for column modification doesn’t work in SQLITE. The work around is to rename this table, then create a new table (but with name of original table) with desired modified column, then move data from renamed table to this table. Doesn’t work in PostgreSQL either.

**Table information in SQLITE –**

sqlite> .schema mydata

CREATE TABLE mydata(

id INT,

language TEXT,

author TEXT,

year INT,

standard TEXT

);

**Writing Basic Queries**

**sqlite> SELECT language, year FROM mydata;** # selecting multiple columns

Prolog|1972

Perl|1987

APL|1964

**sqlite> SELECT language, year FROM mydata ORDER BY year;** #same query with order

APL|1964

Prolog|1972

Perl|1987

**sqlite> SELECT language, year FROM mydata ORDER BY year DESC;** #same query, desc order

Perl|1987

Prolog|1972

APL|1964

**sqlite> SELECT language, year FROM mydata ORDER BY 1**; #1 means language. 2 means year

APL|1964

Perl|1987

Prolog|1972

**sqlite> SELECT language, year FROM mydata WHERE standard = ‘ANSI’;** #conditions with WHERE

APL|1964

**sqlite> SELECT language, year FROM mydata WHERE year > 1970 ORDER BY author;**

Perl|1987

Prolog|1972

**sqlite> SELECT language, author, year FROM mydata WHERE year > 1970 AND standard IS Null;**

Perl|Wall|1987

**sqlite> SELECT language, year FROM mydata WHERE year BETWEEN 1980 AND 1990;**

Perl|1987

Tcl|1988

You can also use **NOT BETWEEN** to exclude the values between your range.

**Manipulating Data**

**Inserting Null –**

**sqlite> INSERT INTO mydata VALUES (4, 'Tcl', 'Ousterhout', '1988', NULL);**

**sqlite> SELECT \* from mydata;**

1|Prolog|Colmerauer|1972|ISO

2|Perl|Wall|1987|

3|APL|Iverson|1964|ANSI

4|Tcl|Ousterhout|1988|

**Inserting Data into a Table from another Table –**

**sqlite> CREATE TABLE mydata2 (language VARCHAR(20), standard VARCHAR(10));**

**sqlite> INSERT INTO mydata2 SELECT language, standard FROM mydata WHERE standard IS NOT NULL;**

**sqlite> SELECT \* FROM mydata2;**

Prolog|ISO

APL|ANSI

**Updating Existing Data –**

**sqlite> UPDATE mydata SET year = 1972, standard = 'ANSI' WHERE language = 'Forth';**

**sqlite> SELECT \* FROM mydata;**

1|Prolog|Colmerauer|1972|ISO

2|Perl|Wall|1987|

3|APL|Iverson|1964|ANSI

4|Tcl|Ousterhout|1988|

5|Forth|Moore|1972|ANSI

**Deleting Existing Data from Tables –**

**sqlite> DELETE FROM mydata WHERE language IS 'Forth'; # or language = ‘Forth’**

**sqlite> SELECT \* FROM mydata;**

1|Prolog|Colmerauer|1972|ISO

2|Perl|Wall|1987|

3|APL|Iverson|1964|ANSI

4|Tcl|Ousterhout|1988|

In above examples, it was implicitly assumed that all fields would contain single value. For example, all the languages have only one author. What if a language has more than one author? We may try to insert comma separated value or insert values with ‘&’ but this will cause more problems if we were to make queries. We can’t even create fields such as ‘author1’ or ‘author2’ as we don’t beforehand how many authors a language might have. To solve all these problems we split the tables as describe below –

**Splitting the Tables -**

Figure: a table holding author details

| **author\_id** | **author** | **language\_id** |
| --- | --- | --- |
| 1 | Colmerauer | 1 |
| 2 | Wall | 2 |
| 3 | Ousterhout | 4 |
| 4 | Iverson | 3 |
| 5 | Kemeny | 5 |
| 6 | Kurtz | 5 |

Figure: a table holding programming language details

| **id** | **language** | **year** | **Standard** |
| --- | --- | --- | --- |
| 1 | Prolog | 1972 | ISO |
| 2 | Perl | 1987 | (null) |
| 3 | APL | 1964 | ANSI |
| 4 | Tcl | 1988 | (null) |
| 5 | BASIC | 1964 | ANSI |

Now we create following tables -

**sqlite> CREATE TABLE lang (id INTEGER NOT NULL PRIMARY KEY,language VARCHAR(20) NOT NULL,**

**...> year INTEGER NOT NULL, standard VARCHAR(10) NULL);**

**sqlite> CREATE TABLE auth (author\_id INTEGER NOT NULL, author VARCHAR(25) NOT NULL,**

**...> language\_id INTEGER REFERENCES lang(id)); #notice REFERENCES**

sqlite> INSERT INTO lang (id, language, year, standard) VALUES(5,'BASIC', 1964,'ANSI');

sqlite> INSERT INTO auth (author\_id, author, language\_id) VALUES(5,'Kemeny', 5);

sqlite> INSERT INTO lang (id, language, year, standard) VALUES(1, 'Prolog', 1972, 'ISO');

sqlite> INSERT INTO lang (id, language, year) VALUES(2, 'Perl', 1987);

sqlite> INSERT INTO lang (id, language, year, standard) VALUES(3, 'APL', 1964,'ISO');

sqlite> INSERT INTO lang (id, language, year) VALUES(4, 'Tcl', 1988);

sqlite> INSERT INTO auth (author\_id, author, language\_id) VALUES(6,'Kurtz', 5);

sqlite> INSERT INTO auth (author\_id, author, language\_id) VALUES(1,'Colmerauer', 1);

sqlite> INSERT INTO auth (author\_id, author, language\_id) VALUES(2,'Wall', 2);

sqlite> INSERT INTO auth (author\_id, author, language\_id) VALUES(3,'Ousterhout', 4);

sqlite> INSERT INTO auth (author\_id, author, language\_id) VALUES(4,'Iverson', 3);

**sqlite> SELECT COUNT(\*) FROM mydata;** #counting records

4

**sqlite> SELECT COUNT(standard) FROM mydata;**

2

**Column Aliases -**

**sqlite> SELECT id, language, author creator FROM mydata;**

1|Prolog|Colmerauer

2|Perl|Wall

3|APL|Iverson

4|Tcl|Ousterhout

sqlite> .mode column

sqlite> .header on

# above 2 commands ensures we see columns name. Be default, SQLITE doesn’t show them

sqlite> SELECT id, language, author creator FROM mydata; #repeated command

id language creator

---------- ---------- ----------

1 Prolog Colmerauer

2 Perl Wall

3 APL Iverson

4 Tcl Ousterhout

### Order of execution of SELECT queries

A query is not evaluated from left to right, there is a specific sequence in which its various parts are evaluated as given below.

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
6. ORDER BY clause

**Like operator –**

For matching we are provided with two wildcard characters to use with *LIKE*.

|  |  |
| --- | --- |
| 1) % (Percent) | Used to match multiple characters including a single character and no character |
| 2) \_ (Underscore) | Used to match exactly one character |

**sqlite> SELECT \* FROM mydata WHERE language LIKE 'p%';**

id language author year standard

---------- ---------- ---------- ---------- ----------

1 Prolog Colmerauer 1972 ISO

2 Perl Wall 1987

**sqlite> SELECT \* FROM mydata WHERE language LIKE 'P%';**

id language author year standard

---------- ---------- ---------- ---------- ----------

1 Prolog Colmerauer 1972 ISO

2 Perl Wall 1987

**sqlite> SELECT \* FROM mydata WHERE language LIKE '%erl';**

id language author year standard

---------- ---------- ---------- ---------- ----------

2 Perl Wall 1987

**sqlite> SELECT \* FROM mydata WHERE language LIKE '\_\_l';**

id language author year standard

---------- ---------- ---------- ---------- ----------

3 APL Iverson 1964 ANSI

4 Tcl Ousterhout 1988

**sqlite> SELECT \* FROM mydata WHERE language LIKE '\_\_L';**

id language author year standard

---------- ---------- ---------- ---------- ----------

3 APL Iverson 1964 ANSI

4 Tcl Ousterhout 1988

Per tutorial, wildcard matching with % and \_ is case sensitive (in Postgres it is indeed) but above examples show otherwise. \_ can be used per character basis. This means, as shown above, if we are to match 3 character word ending with ‘l’, we’ll use 2 \_ followed by letter ‘l’.

**Mathematical Calculation**

**sqlite> SELECT language, (year%10) remain FROM mydata;**

language remain

---------- ----------

Prolog 2

Perl 7

APL 4

Tcl 8

**sqlite> SELECT language, year - (year%10) decade FROM mydata;**

language decade

---------- ----------

Prolog 1970

Perl 1980

APL 1960

Tcl 1980

**String Operation**

**sqlite> SELECT language, 'The '||((year/10)\*10)||'s' decade FROM mydata;**

language decade

---------- ----------

Prolog The 1970s

Perl The 1980s

APL The 1960s

Tcl The 1980s

sqlite>

**Literal values -**

**sqlite> SELECT language, year, 'AD', 44 FROM mydata;**

language year 'AD' 44

---------- ---------- ---------- ----------

Prolog 1972 AD 44

Perl 1987 AD 44

APL 1964 AD 44

Tcl 1988 AD 44

#works differently in Postgres. It does fill the values like above but doesn’t use these values as column name. Instead it uses ?column? as placeholder column name.

**Aggregation and Grouping –** Let us add few more records and run some queries –

**sqlite> INSERT INTO mydata (id, language, author, year, standard) VALUES(5, 'For**

**tran', 'Backus', 1957, 'ANSI');**

**sqlite> INSERT INTO mydata (id, language, author, year, standard) VALUES(6, 'PL/**

**I', 'IBM', 1964, 'ECMA');**

**sqlite> SELECT \* FROM mydata;**

id language author year standard

---------- ---------- ---------- ---------- ----------

1 Prolog Colmerauer 1972 ISO

2 Perl Wall 1987

3 APL Iverson 1964 ANSI

4 Tcl Ousterhout 1988

5 Fortran Backus 1957 ANSI

6 PL/I IBM 1964 ECMA

**sqlite> SELECT COUNT (DISTINCT year) FROM mydata;**

COUNT (DISTINCT year)

---------------------

5

**sqlite> SELECT COUNT (DISTINCT standard) FROM mydata;**

COUNT (DISTINCT standard)

-------------------------

3

Note that the **DISTINCT** clause did not count NULL values as truly distinct values.

**sqlite> SELECT MIN(year) FROM mydata;**

MIN(year)

----------

1957

**sqlite> SELECT language, MAX(year) year FROM mydata;**

#seems doesn’t work in Postgres.

language year

---------- ----------

Tcl 1988

**Grouping Data –**

**sqlite> SELECT language, standard FROM mydata WHERE standard IS NOT NULL GROUP BY standard, language;**

language standard

---------- ----------

APL ANSI

Fortran ANSI

PL/I ECMA

Prolog ISO

#seems Group By works differently in Postgres

Note –You cannot group by a column which is not present in the *SELECT* list. You must specify all the columns in the grouping clause which are present in the *SELECT* list.

#Better and Revised Explanation – The `GROUP BY` clause must appear right after FROM or (optionally) WHERE clause. Followed by the `GROUP BY` clause is one column or a list of comma separated columns. This clause may be followed by `ORDER BY` clause. Also, the column(s) mentioned in `GROUP BY` clause must be present in SELECT clause. The general syntax is like below –

SELECT column\_1, aggregate\_function(column\_2)

FROM table\_name

WHERE some\_condition (this is optional)

GROUP BY column\_1

ORDER BY column\_1 or aggregate\_function(column2);

ORDER OF EXECUTION – FROM->WHERE -> GROUPBY->SELECT

**sqlite> SELECT standard, count(\*) FROM mydata GROUP BY standard;**

standard count(\*)

---------- ----------

2 #notice this record

ANSI 2

ECMA 1

ISO 1

**sqlite> SELECT standard, count(standard) FROM mydata GROUP BY standard;**

standard count(\*)

---------- ----------

0 #notice this record and compare with above

ISO 1

ANSI 2

ECMA 1

Notice the first record in above result. This is the result for NULL values.

**The ‘Having’ Clause –**

sqlite> SELECT language, standard, year FROM mydata GROUP BY standard, year, language HAVING year < 1980;

language standard year

---------- ---------- ----------

Fortran ANSI 1957

APL ANSI 1964

PL/I ECMA 1964

Prolog ISO 1972

**Understanding Joining –** Recall we have created 2 tables ‘auth’ and ‘lang’.

**sqlite> SELECT \* FROM auth;**

author\_id author language\_id

---------- ---------- -----------

5 Kemeny 5

6 Kurtz 5

2 Wall 2

3 Ousterhout 4

4 Iverson 3

1 Colmerauer 1

**sqlite> SELECT \* FROM lang;**

id language year standard

---------- ---------- ---------- ----------

1 Prolog 1972 ISO

2 Perl 1987

3 APL 1964 ISO

4 Tcl 1988

5 BASIC 1964 ANSI

No**w we run following queries -**

**sqlite> SELECT author, language FROM auth, lang WHERE language\_id = id;**

author language

---------- ----------

Kemeny BASIC

Kurtz BASIC

Wall Perl

Ousterhout Tcl

Iverson APL

Colmerauer Prolog

**sqlite> SELECT author, language FROM auth JOIN lang ON language\_id = id;**

Above query produces the same result as earlier one.

**Resolving ambiguity in join columns** – In our example the join condition fields had distinct names - *id* and *language\_id*. But what if in our languages table (*lang*) we kept the key field’s name as *language\_id*. This would create an ambiguity in the join condition, which would become the confusing *language\_id = language\_id*. To resolve this, we need to qualify the column by prepending it by the table name it belongs to and a .(period).

**sqlite> SELECT author, language FROM auth JOIN lang ON auth.language\_id = lang.language\_id;**

**Self Joins –** Relationship between 2 columns in the same table.

**sqlite> CREATE TABLE inflang (id INTEGR PRIMARY KEY,language VARCHAR(20) NOT NULL,**

**...> influenced\_by INTEGER);**

**sqlite> INSERT INTO inflang(id, language) VALUES (1, 'Fortran');**

**sqlite> INSERT INTO inflang(id, language, influenced\_by) VALUES (2, 'Pascal',3);**

**sqlite> INSERT INTO inflang(id, language, influenced\_by) VALUES (3, 'Algol',1);**

**sqlite> SELECT \* FROM inflang;**

id language influenced\_by

---------- ---------- -------------

1 Fortran

2 Pascal 3

3 Algol 1

Now, running our self join query -

**sqlite> SELECT l1.language, l2.language AS influenced FROM inflang l1, inflang l2 WHERE l1.id = l2.influenced\_by;**

language influenced

---------- ----------

Algol Pascal

Fortran Algol

Notice the use of table aliases to qualify the join condition columns as separate and the use of the AS keyword which renames the column in the output.

**Subqueries -** A subquery, simply put, is a query written as a part of a bigger statement. Think of it as a *SELECT* statement inside another one. The result of the inner *SELECT* can then be used in the outer query.

**sqlite> SELECT author FROM auth WHERE language\_id IN (SELECT id FROM lang WHERE language = 'Tcl');**

author

----------

Ousterhout

There are basically 2 types of subqueries. First one scalar subquery which returns only a single column of a single row. The query we ran just above is an example of scalar subquery. Other type of subquery is Table subquery which returns a table in itself. Example is below –

**sqlite> SELECT author, language FROM auth a, (SELECT id, language FROM lang WHERE year > 1980)n WHERE a.language\_id = n.id;**

author language

---------- ----------

Wall Perl

Ousterhout Tcl

Using subqueries in INSERT statements – Let’s first insert a record in ‘lang’ table –

**sqlite> INSERT INTO lang (id, language, year, standard) VALUES(6, 'Pascal', 1970, 'ISO');**

**sqlite> SELECT \* FROM lang;**

id language year standard

---------- ---------- ---------- ----------

1 Prolog 1972 ISO

2 Perl 1987

3 APL 1964 ISO

4 Tcl 1988

5 BASIC 1964 ANSI

6 Pascal 1970 ISO

**sqlite> INSERT INTO auth (author\_id, author, language\_id) VALUES(7, 'Wirth', (SELECT id FROM lang WHERE language = 'Pascal'));**

**sqlite> SELECT \* FROM auth;**

author\_id author language\_id

---------- ---------- -----------

5 Kemeny 5

6 Kurtz 5

2 Wall 2

3 Ousterhout 4

4 Iverson 3

1 Colmerauer 1

7 Wirth 6

-----------------------------------------------------

**Additional bits of SQL -**

**sqlite> SELECT language, standard FROM lang WHERE standard = 'ISO' OR standard**

**IS Null;**

language standard

---------- ----------

Prolog ISO

Perl

APL ISO

Tcl

Pascal ISO

In above query standard = Null doesn’t work.

**WHERE EXISTS** –

**sqlite> SELECT year FROM lang WHERE EXISTS(SELECT author FROM auth WHERE language\_id = lang.id AND language\_id > 4);**

year

----------

1964

1970

#WHERE EXISTS has to be understood properly. More examples needed. Note – Here WHERE and EXISTS are two different keywords.

**Any and All –** Doesn’t work in SQLITE

**Inserting multiple records -**

**sqlite> INSERT INTO auth (author\_id, author, language\_id) VALUES(11,’A’,21),(12,’B’,22);**

**ifnull() function - # not for Postgres**

sqlite> SELECT language, standard, ifnull(standard,'NA') FROM lang;

language standard ifnull(standard,'NA')

---------- ---------- ---------------------

Prolog ISO ISO

Perl NA

APL ISO ISO

Tcl NA

BASIC ANSI ANSI

Pascal ISO ISO

**PostgreSQL**

Installing PostgreSQL was painful. Documentation is unclear. After installing, ‘createdb mydb’ command didn’t work. Then I realized I had to set environment varialble ‘path’. Did that but then faced with another problem. It asked for password which I supplied correctly but somehow it didn’t work. Spent a lot of time looking for solution then found one blog post which suggested that I needed to set another environment variable called ‘PGUSER’. Link for the [blog post](http://bobbyong.com/blog/installing-postgresql-on-windoes/).

C:\Users\Mayank>createdb mydb

C:\Users\Mayank>dropdb mydb

C:\Users\Mayank>psql

-- for comment

Alternatively, you can do following to create database.

C:\Users\Mayank>psql

--snipped output---

postgres=# \l #list out databases

postgres=# CREATE DATABASE mydb; #caps lock isn’t necessary

CREATE DATABASE

postgres=# \l #to see if mydb has been created.

postgres=# \c mydb #to connect to mydb database

mydb=#

You can delete the database by issuing DROP DATABASE mydb; command

Or you can do this –

**C:\Users\Mayank>psql mydb** # mydb should exist

mydb=# \h or \q # for help or to quit respectively

Finding list of tables –

mydb=# SELECT table\_name FROM information\_schema.tables; #all tables of mydb database

mydb=# SELECT table\_name FROM information\_schema.tables WHERE table\_schema = ‘public’;

above command will give the list of tables created by users.

mydb=# \z; #this also gives the list of user created tables with some additional info abt them.

mydb=# \dt; #this also gives the list of user created tables with some additional info abt them.

mydb=# \d+ mydata; #schema of table mydata

mydb=# \d mydata; #schema of table mydata (stripped down details)

mydb=# SELECT column\_name, data\_type, character\_maximum\_length FROM information\_schema.columns WHERE table\_name = 'mydata'; #schema of table mydata

**Creating user** –

Createuser trainee #through command line

CREATE USER trainee #inside psql

postgres=# \du #list of users

You can’t login as a different user when trying to login locally. However, there is a workaround for that –

C:\Users\Mayank>psql -U trainee -d mydb -h 127.0.0.1 -W

#-U for username, -d for database name, -h for connecting through a network interface and #–W for prompting for user password. [Source for this workaround](https://www.digitalocean.com/community/tutorials/how-to-use-roles-and-manage-grant-permissions-in-postgresql-on-a-vps--2)

Password for user trainee: #enter password for user trainee

psql (10.0)

WARNING: ---snipped output---

mydb=> #notice that prompt is now different.

**SOME DATA TYPES from PostgreSQL**

|  |  |
| --- | --- |
| **NAME** | **DESCRIPTION** |
| character [(n)] | Fixed length character string |
| character varying [(n)] | Variable length character string |
| data | Calender data (y, m, d) |
| boolean |  |
| inet | IPv4 or IPv6 addresses |
| integer | 4 byte signed integer |
| json | Textual JSON data |
| jsonb | Binary JSON data |
| real | Single precision 4 byte floating point number |
| serial | Autoincrementing 4 bytes integer |
| text | Variable length character string |
| point |  |
| date |  |
|  |  |
|  |  |
|  |  |

**COPY command and Copying data from CSV file**

First create a CSV file through command line.

Note – I wrongly assumed that ‘cat’ is a Windows command. It isn’t. It doesn’t work on Windows Command Prompt. The reason it worked in my case was that I had Cygwin installed on my system and it was on Path. Since ‘cat’ is a Unix utility, Command Prompt found it in Path and was able to execute it. Try ‘type’ for Windows.

**c:\miniconda3>cat > test.csv**

1,a

2,b

3,c

Now, we insert the content of CSV file in the table like below –

**postgres=# CREATE TABLE test (id INTEGER, name VARCHAR(10));**

CREATE TABLE

**postgres=# COPY test FROM 'c:\miniconda3\test.csv' DELIMITER ',';**

COPY 3

**postgres=# SELECT \* FROM test;**

id | name

----+------

1 | a

2 | b

3 | c

(3 rows)

**Copy Command and Copying Data From TERMINAL**

**postgres=# COPY test FROM STDIN DELIMITER ',';**

Enter data to be copied followed by a newline.

End with a backslash and a period on a line by itself, or an EOF signal.

>> 4,d

>> 5,e

>> \.

COPY 2

**postgres=# SELECT \* FROM test;**

id | name

----+------

1 | a

2 | b

3 | c

4 | d

5 | e

(5 rows)

**Copy database content into a CSV file**

**postgres=# COPY test TO 'c:\miniconda3\test1.csv' USING DELIMITERS ‘,’;**

COPY 5

**Copy database content into a Text File**

**postgres=# COPY test TO 'c:\miniconda3\test2.txt' using DELIMITERS** ;

COPY 5

1|a

2|b

3|c

4|d

5|e

**VIEWS** –

**postgres=# CREATE VIEW test AS SELECT language, author FROM mydata;**

CREATE VIEW

**postgres=# SELECT \* FROM test;**

language | author

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

Prolog | Colmerauer

Perl | Wall

APL | Iverson

Tcl | Ousterhaut

Fortran | Backus

PL/I | IBM

(6 rows)

To delete view, use **DROP VIEW**.

Also, view ‘test’ depends on table ‘mydata’. If you try to remove ‘mydata’, a warning will be raised.

**postgres=# DROP TABLE auth, mydata, lang;**

ERROR: cannot drop desired object(s) because other objects depend on them

DETAIL: view test depends on table mydata

HINT: Use DROP ... CASCADE to drop the dependent objects too. ^

**postgres=# DROP TABLE auth, mydata, lang CASCADE;**

NOTICE: drop cascades to view test

DROP TABLE

**Note** –

**postgres=# SELECT current\_date;**

current\_date

--------------

2018-05-15

(1 row)

Z’

**postgres=# SELECT (5+4)/2;**

?column?

----------

4

(1 row)

**postgres=# SELECT (5+4)/2.;**

?column?

--------------------

4.5000000000000000

(1 row)

**postgres=# SELECT CHAR\_LENGTH('ABCDE');** -- CHARACTER\_LENGTH can also be used

char\_length

-------------

5

(1 row)

Let us create some tables -

**postgres=# SELECT \* FROM weather;**

city | temp\_lo | temp\_hi | prcp | date

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

San Francisco | 46 | 50 | 0.25 | 1994-11-27

San Fransisco | 43 | 50 | 0 | 1994-11-29

Hayward | 37 | 54 | | 1994-11-29

(3 rows)

**postgres=# SELECT \* FROM cities;**

name | location

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

San Fransisco | (-194,53)

(1 row)

**JOIN –**

**postgres=# SELECT \* FROM weather INNER JOIN cities ON city = name;**

city | temp\_low | temp\_hi | prcp | date | name | location

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

SF | 46 | 50 | 0.25 | 1994-11-27 | SF | (-194,53)

SF | 43 | 57 | 0 | 1994-11-29 | SF | (-194,53)

(2 rows)

^

**postgres=# SELECT \* FROM weather LEFT OUTER JOIN cities ON city = name;**

#all rows from left table

city | temp\_low | temp\_hi | prcp | date | name | location

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

SF | 46 | 50 | 0.25 | 1994-11-27 | SF | (-194,53)

SF | 43 | 57 | 0 | 1994-11-29 | SF | (-194,53)

Hayward | 54 | 37 | | 1994-11-29 | |

(3 rows)

**postgres=# SELECT \* FROM weather RIGHT OUTER JOIN cities ON city = name;**

all rows from right table only (cities)

city | temp\_low | temp\_hi | prcp | date | name | location

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

SF | 46 | 50 | 0.25 | 1994-11-27 | SF | (-194,53)

SF | 43 | 57 | 0 | 1994-11-29 | SF | (-194,53)

(2 rows)

**postgres=# SELECT \* FROM weather FULL OUTER JOIN cities ON city = name;**

city | temp\_lo | temp\_hi | prcp | date | name | location

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

Hayward | 37 | 54 | | 1994-11-29 | |

SF | 46 | 50 | 0.25 | 1994-11-27 | SF | (-194,53)

SF | 43 | 50 | 0 | 1994-11-29 | SF | (-194,53)

(3 rows)

**FOREIGN KEY CONSTRAINT**

In above case, suppose we want to add a record in ‘weather’ table which must have have a matching city in ‘cities’ table. If that is not the case, the record shouldn’t be added. We do this as following –

CREATE TABLE cities (

city varchar(80) primary key,

location point

);

CREATE TABLE weather (

city varchar(80) references cities(city),

temp\_lo int,

temp\_hi int,

prcp real,

date date

);

Now try inserting an invalid record:

**INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');**

ERROR: insert or update on table "weather" violates foreign key constraint "weather\_city\_fkey"

DETAIL: Key (city)=(Berkeley) is not present in table "cities".

**Aggregate function and WHERE**

**postgres=# SELECT city FROM weather WHERE temp\_low = max(temp\_low);**

ERROR: aggregate functions are not allowed in WHERE

**postgres=# SELECT city FROM weather WHERE temp\_low = (SELECT max(temp\_low) FROM weather);**

city

---------

Hayward

(1 row)

It is important to understand the interaction between aggregates and SQL's WHERE and HAVING clauses. The fundamental difference between WHERE and HAVING is yuthis: WHERE selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas HAVING selects group rows after groups and aggregates are computed. Thus, **the WHERE clause must not contain aggregate functions;** it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, the **HAVING clause always contains aggregate functions**. (Strictly speaking, you are allowed to write a HAVING clause that doesn't use aggregates, but it's seldom useful. The same condition could be used more efficiently at the WHERE stage.)

**postgres=# UPDATE weather SET temp\_hi = 54, temp\_low = 37 WHERE city = 'Hayward';**

UPDATE 1

postgres=# DELETE FROM weather WHERE city = 'Hayward';

DELETE 1

**WINDOW FUNCTIONS**

**postgres=# SELECT \* FROM dept;**

empno | dept | salary

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

1 | Sales | 15000

2 | Purchase | 20000

3 | Sales | 16000

4 | Purchase | 19000

(4 rows)

**postgres=# SELECT dept, empno, salary, avg(salary) OVER (PARTITION BY dept) FROM dept;**

dept | empno | salary | avg

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

Purchase | 2 | 20000 | 19500.000000000000

Purchase | 4 | 19000 | 19500.000000000000

Sales | 1 | 15000 | 15500.000000000000

Sales | 3 | 16000 | 15500.000000000000

(4 rows)

A window function call always contains an OVER clause directly following the window function's name and argument(s). This is what syntactically distinguishes it from a normal function or nonwindow aggregate. The OVER clause determines exactly how the rows of the query are split up for processing by the window function. The PARTITION BY clause within OVER divides the rows into groups, or partitions, that share the same values of the PARTITION BY expression(s). For each row, the window function is computed across the rows that fall into the same partition as the current row.

**INHERITS**

**postgres=# CREATE TABLE city(name text, population real, altitude int);**

**CREATE TABLE**

**postgres=# CREATE TABLE capitals (state char(2)) INHERITS (city);**

**CREATE TABLE**

**postgres=# \d capitals**

Table "public.capitals"

Column | Type | Collation | Nullable | Default

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

name | text | | |

population | real | | |

altitude | integer | | |

state | character(2) | | |

Inherits: city

**postgres=# INSERT INTO capitals VALUES ('Lucknow', 123456, 123, 'UP');**

INSERT 0 1

**postgres=# SELECT \* FROM capitals;**

name | population | altitude | state

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

Lucknow | 123456 | 123 | UP

(1 row)

**postgres=# SELECT \* FROM city;**

name | population | altitude

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

Lucknow | 123456 | 123

(1 row)

**Modifying Tables**

You can:

• Add columns

• Remove columns

• Add constraints

• Remove constraints

• Change default values

• Change column data types

• Rename columns

• Rename tables

\*Adding column -

ALTER TABLE products ADD COLUMN description text;

The new column is initially filled with whatever default value is given (null if you don't specify a DEFAULT clause).

You can also define constraints on the column at the same time, using the usual syntax:

ALTER TABLE products ADD COLUMN description text CHECK (description <> '');

\*Droping columns

ALTER TABLE products DROP COLUMN description;

Whatever data was in the column disappears. Table constraints involving the column are dropped, too. However, if the column is referenced by a foreign key constraint of another table, PostgreSQL will not silently drop that constraint. You can authorize dropping everything that depends on the column by adding CASCADE:

ALTER TABLE products DROP COLUMN description CASCADE;

\*To set a new default for a column, use a command like:

ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;

Note that this doesn't affect any existing rows in the table, it just changes the default for future INSERT commands.

\*To remove any default value, use:

ALTER TABLE products ALTER COLUMN price DROP DEFAULT;

This is effectively the same as setting the default to null. As a consequence, it is not an error to drop a default where one hadn't been defined, because the default is implicitly the null value.

\*Removing a constraint

ALTER TABLE products DROP CONSTRAINT some\_name;

\*To convert a column to a different data type, use a command like:

ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);

This will succeed only if each existing entry in the column can be converted to the new type by an

implicit cast.

\*To rename a column:

ALTER TABLE products RENAME COLUMN product\_no TO product\_number;

\*To rename a table:

ALTER TABLE products RENAME TO items;

**SUBQUERY EXPRESSION –**

**EXISTS (subquery)**

**expression IN (subquery)**

**expression NOT IN (subquery)**

**expression operator ANY (subquery)**

**expression operator SOME (subquery)**

**expression operator All (subquery)**