Python for PostgresSQL developers

# PostgresSQL Data Types.

PostgresSQL has the following data types available.

|  |  |
| --- | --- |
| Data Type Syntax | Explanation |
| char(size) | Where size is the number of characters to store. Fixed-length strings. Space padded on right to equal size characters. |
| character(size) | Where size is the number of characters to store. Fixed-length strings. Space padded on right to equal size characters. |
| varchar(size) | Where size is the number of characters to store. Variable-length string. |
| character varying(size) | Where size is the number of characters to store. Variable-length string. |
| text | Variable-length string. |

Numeric data types

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Storage Size | Description | Range |
| Smallint | 2 bytes | small-range integer | -32768 to +32767 |
| integer - | 4 bytes | typical choice for integer | 2147483648 to +2147483647 |
| bigint - | 8 bytes | large-range integer | 9223372036854775808 to 9223372036854775807 |
| decimal | variable | user-specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| numeric | variable | user-specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| real | 4 bytes | variable-precision, inexact | 6 decimal digits precision |
| double precision | 8 bytes | variable-precision, inexact | 15 decimal digits precision |
| serial | 4 bytes | autoincrementing integer | 1 to 2147483647 |
| bigserial | 8 bytes | large autoincrementing integer | 1 to 9223372036854775807 |

Binary data types

|  |  |  |
| --- | --- | --- |
| Name | Storage Size | Description |
| bytea | 1 or 4 bytes plus the actual binary string | variable-length binary string |

## Which data type to use?

### Integers.

* Smallint. Use only if space is at a premium, for example embedded systems.
* BigInt. BigInt has a performance penalty compared to Int.
* Int. For everything else.

### Numeric.

* Provides scale and precision
* Scale. Number of digits to the right of the decimal point.
* Precision. Total number of digits in a number.
* Be clear on what you use and why.
  + The precision should be large enough to provide the ability for the application to handle larger numbers at a future time. Example, handling amounts in thousands today and millions tomorrow.
  + The scale has to be sufficient, for example if you have an accounting application that needs to store monetary values with a fraction of the smallest currency account, for example using a scale of 3 or 4 rather than the two needed for USD pennies.
  + Be mindful of rounding and truncation in the decimal fraction and inadvertent NaN’s.
  + Avoid floating point data types for currency appplications. Floating point is designed for performance, not accuracy. In currency applications, accuracy is the more meaningful choice.
* Declarations

numeric(precision,scale)

* + Maximum number declarable is 1000
  + Max scale is 100
  + Has a special value NaN which means Not a Numer.

numeric(precision)

* + This is effectively an integer.

An example of the numeric data type:

SELECT 100 \* (0.08875)::numeric;

---

8.875

SELECT 100 \* (0.08875)::numeric(7,2);

---

9.0

SELECT (100 \* 0.08875)::numeric(7,2);

---

8.88

### Numbers – Floating Point.

* Uses the IEEE 754 standard for floating point representation
* Not exact. Unexpected behavior is possible including:
  + Overflow/Underflow
  + Equality imprecision.
* Constants
  + ‘NaN’, ‘Infinity’,’-Infinity’
* Types
  + Real => 1E-37 <=> 1E+37
  + double precision => 1E-308 <=> 1E+308
  + float(1) <=> float(24) = real
  + float(25) <=> float(53) = double precision

An example of using floats vs. the numeric data type.

|  |
| --- |
| \timing  CREATE TABLE floats (x double precision);  CREATE TABLE numerics (x numeric(15, 15));  INSERT INTO floats  SELECT random() FROM generate\_series(1,1000000);  INSERT INTO numerics  SELECT \* FROM floats;  CREATE INDEX floats\_idx ON floats (x);  CREATE INDEX numerics\_idx ON numerics (x);  SELECT \* FROM floats WHERE x >= 0.7;  -- avg 280ms  SELECT \* FROM numerics WHERE x >= 0.7;  -- avg 120ms |

* Generally better to use numeric rather than float.
* Floating Point usage is application specific
  + Reading data from a thermometer, for example.
  + When you have too many rows for larger numeric data types
  + Don’t requires a specific precision.
* You should understand the ramifications of your choice before making it.

Serial Types.

|  |  |  |
| --- | --- | --- |
| Name | Storage Size | Range |
| smallserial | 2 bytes | 1 to 32767 |
| serial | 4 bytes | 1 to 2147483647 |
| bigserial | 8 bytes | 1 to 9223372036854775807 |

### Serial Data Type.

* Serial is not really a data type, but it is a very useful convenience.

|  |
| --- |
| CREATE TABLE awesome (  id serial  ); |

|  |
| --- |
| CREATE SEQUENCE awesome\_id\_seq;  CREATE TABLE awesome (  id integer NOT NULL DEFAULT nextval(‘awesome\_id\_seq’)  );  ALTER SEQUENCE awesome\_colname\_seq OWNED BY awesome.id; |

### Monetary Data Type.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Storage Size | Description | Range |
| money | 8 bytes | currency amount | -92233720368547758.08 to +92233720368547758.07 |

* Stores monetary amounts based on a the ‘lc\_monetary’ setting.
* Output based on lc\_monetary. E.g.
  + ‘$1000.00’
* The reality.
  + Don’t use the monetary type.
  + Store currency as Integer or Numeric types.
  + Money is based on a database wide environment setting.
  + This setting can change widely between instances.
  + You cannot control the environment swetting for specific database columns.
  + Money is not a standard SQL data type. Postgres includes this for the convenience of users who are importing data from other database systems.

### The Boolean type.

|  |  |
| --- | --- |
| Name | Size |
| boolean | 1. byte |

These are all equivalent

– TRUE, ‘t’, ‘true’, ‘y’, ‘yes’, ‘on’, ‘1’

– FALSE, ‘f’, ‘false’, ‘n’, ‘no’, ‘off’, ‘0’

All case-insensitive, preferred TRUE / FALSE

### Datetime Data Type.

|  |  |
| --- | --- |
| Data Type Syntax | Explanation |
| date | Displayed as 'YYYY-MM-DD'.  timestamp |
| timestamp without time zone | Displayed as 'YYYY-MM-DD HH:MM:SS'. |
| timestamp with time zone | Displayed as 'YYYY-MM-DD HH:MM:SS-TZ'.  Equivalent to timestamptz. |
| time | Displayed as 'HH:MM:SS' with no time zone. |
| time without time zone | Displayed as 'HH:MM:SS' with no time zone. |
| time with time zone. | Displayed as 'HH:MM:SS-TZ' with time zone. Equivalent to timetz. |

* Format can be adjusted using the following:
  + Command: SET <datestyle>
  + Modify postgressql.conf – ‘DateStyle’ parameter
  + Environment variable: PGDATESTYLE

Examples of using the datetime data type in PostgresSql.

|  |
| --- |
| postgres=# BEGIN;  postgres=# SELECT now();  now  -------------------------------  2013-08-26 12:17:43.182331+02  postgres=# SELECT now();  now  -------------------------------  2013-08-26 12:17:43.182331+02  postgres=# SELECT clock\_timestamp();  clock\_timestamp  -------------------------------  2013-08-26 12:17:50.698413+02  postgres=# SELECT clock\_timestamp();  clock\_timestamp  -------------------------------  2013-08-26 12:17:51.123905+02 |

Note that the now() function doesn’t change until the transaction ends, whereas clock\_timestamp

changes each time you call it.

### Intervals.

* YEAR
* MONTH
* DAY
* HOUR
* MINUTE
* SECOND
* YEAR TO MONTH
* DAY TO HOUR
* DAY TO MINUTE
* DAY TO SECOND
* HOUR TO MINUTE
* HOUR TO SECOND
* MINUTE TO SECOND

Intervals allow us to select datetime intervals very easily.

|  |
| --- |
| postgres=# SELECT now() - interval '3 days';  ?column?  -------------------------------  2013-08-23 12:23:40.069717+02 |

Extracting datetime fields.

|  |
| --- |
| postgres=# SELECT extract(DAY FROM now());  date\_part  -----------  26  postgres=# SELECT extract(DOW FROM now());  date\_part  -----------  1 |

Converting between timezones.

|  |
| --- |
| postgres=# BEGIN;  BEGIN  postgres=# SELECT now();  now  -------------------------------  2013-08-26 12:39:39.122218+02  postgres=# SELECT now() AT TIME ZONE 'GMT';  timezone  ----------------------------  2013-08-26 10:39:39.122218  postgres=# SELECT now() AT TIME ZONE 'GMT+1';  timezone  ----------------------------  2013-08-26 09:39:39.122218  postgres=# SELECT now() AT TIME ZONE 'PST';  timezone  ----------------------------  2013-08-26 02:39:39.122218 |

# PostgresSQL operators and functions

Operators in PostgresSQL are reserved words and symbols used in a PostgresSQL statements WHERE clause to perform operations such as comparisons and arithmetic operations.

Operators are used to specify conditions in a SQL statement and to serve as conjunctions for multiple conditions in a statement.

The operators are grouped into the following categories

* Arithmetic operators
* Comparison operators
* Logical operators
* Bitwise operators

### Arithmetic operators.

In the following table, assume that a contains the value of 3 and b contains the value of 5.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| + | Addition –Adds values on either side of the operator | a + b gives 5 |
| - | Subtraction - Subtracts right hand operand from left hand operand | a - b will give -1 |
| \* | Multiplication - Multiplies values on either side of the operator | a \* b will give 6 |
| / | Division - Divides left hand operand by right hand operand | b / a will give 1 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | b % a will give 1 |
| ^ | Exponentiation - This gives the exponent value of the right hand operand | a ^ b will give 8 |
| |/ | square root | |/ 25.0 will give 5 |
| ||/ | Cube root | ||/ 27.0 will give 3 |
| ! | factorial | 5 ! will give 120 |

### Comparison operators

In the following table, assume that a contains the value of 3 and b contains the value of 5.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (a = b) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| <> | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |

### Logical operators

|  |  |
| --- | --- |
| Symbol | Operator & Description |
| AND | The AND operator allows the existence of multiple conditions in a PostgresSQL statement's WHERE clause. |
| OR | The OR operator is used to combine multiple conditions in a PostgresSQL statement's WHERE clause. |
| NOT | The NOT operator reverses the meaning of the logical operator with which it is used. Eg. NOT EXISTS, NOT BETWEEN, NOT IN etc. This is negate operator. |

### PostgresSQL bit string operators.

Bitwise operators operate on bits and do bit by bit operations.

The truth table for the bitwise AND (&) and the bitwise OR (|) is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| p | q | p & q | p | q |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

Now if we apply bitwise operators to A and B, we get the following:

A&B = 0000 1100

A|B = 0011 1101

~A = 1100 0011

Note that the (~) symbol is the NOT operator.

The bitwise operators are as follows:

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| & ~ | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |
| # | bitwise XOR. | A # B will give 49 which is 0100 1001 |

# Postgres Functions

PostgreSQL functions, also known as Stored Procedures, allow you to carry out operations that would normally take several queries and round trips in a single function within the database. Functions allow database reuse as other applications can interact directly with your stored procedures instead of a middle-tier or duplicating code.

Functions can be created in a language of your choice like SQL, PL/pgSQL, C, Python, etc.

Here is the syntax for creating a PostgresSQL function.

|  |
| --- |
| CREATE [OR REPLACE] FUNCTION function\_name (arguments)  RETURNS return\_datatype AS $variable\_name$  DECLARE  declaration;  [...]  BEGIN  < function\_body >  [...]  RETURN { variable\_name | value }  END; LANGUAGE plpgsql; |

Where,

function-name specifies the name of the function.

[OR REPLACE] option allows modifying an existing function.

The function must contain a return statement.

RETURN clause specifies that data type you are going to return from the function. The return\_datatype can be a base, composite, or domain type, or can reference the type of a table column.

function-body contains the executable part.

The AS keyword is used for creating a standalone function.

plpgsql is the name of the language that the function is implemented in. Here, we use this option for PostgreSQL, it Can be SQL, C, internal, or the name of a user-defined procedural language. For backward compatibility, the name can be enclosed by single quotes.

Example

The following example illustrates creating and calling a standalone function. This function returns the total number of records in the COMPANY table. The COMPANY table has the following records –

|  |
| --- |
| testdb# select \* from COMPANY;  id | name | age | address | salary  ----+-------+-----+-----------+--------  1 | Paul | 32 | California| 20000  2 | Allen | 25 | Texas | 15000  3 | Teddy | 23 | Norway | 20000  4 | Mark | 25 | Rich-Mond | 65000  5 | David | 27 | Texas | 85000  6 | Kim | 22 | South-Hall| 45000  7 | James | 24 | Houston | 10000  (7 rows) |

|  |
| --- |
| CREATE OR REPLACE FUNCTION totalRecords ()  RETURNS integer AS $total$  declare  total integer;  BEGIN  SELECT count(\*) into total FROM COMPANY;  RETURN total;  END;  $total$ LANGUAGE plpgsql; |

Now we can call this function from our database interface and have it return the results directly.

|  |
| --- |
| testdb=# select totalRecords();  totalrecords  --------------  7  (1 row) |

## Some useful functions.

### Aggregate functions.

* PostgreSQL COUNT Function − The PostgreSQL COUNT aggregate function is used to count the number of rows in a database table.

|  |
| --- |
| testdb=# SELECT COUNT(\*) FROM COMPANY ;  count  -------  7  (1 row)  testdb=# SELECT COUNT(\*) FROM COMPANY WHERE name='Paul';  count  -------  1  (1 row) |

* PostgreSQL MAX Function − The PostgresSQL MAX aggregate function allows us to select the highest (maximum) value for a certain column.

|  |
| --- |
| testdb=# SELECT MAX(salary) FROM COMPANY;  max  -------  85000  (1 row) |

* PostgreSQL MIN Function − The PostgresSQL MIN aggregate function allows us to select the lowest (minimum) value for a certain column.

|  |
| --- |
| testdb=# SELECT MIN(salary) FROM company;  min  -------  10000  (1 row) |

* PostgreSQL AVG Function − The PostgreSQL AVG aggregate function selects the average value for certain table column.

|  |
| --- |
| testdb=# SELECT AVG(SALARY) FROM COMPANY;  avg  ------------------  37142.8571428571  (1 row) |

* PostgreSQL SUM Function − The PostgresSQL SUM aggregate function allows selecting the total for a numeric column.

|  |
| --- |
| testdb# SELECT SUM(salary) FROM company;  sum  --------  260000  (1 row) |

* PostgreSQL ARRAY\_AGG Function − The PostgresSQL ARRAY aggregate function puts input values, including nulls, concatenated into an array.

|  |
| --- |
| testdb=# SELECT ARRAY\_AGG(SALARY) FROM COMPANY;  array\_agg  ---------------------------------------------  {20000,15000,20000,65000,85000,45000,10000} |

# PostgresSQL type conversion

Type conversion allows the user to convert a variable from one data type to another. A simple example here converts a String constant to an Integer.

|  |
| --- |
| SELECT  CAST ('100' AS INTEGER); |

Note that if you attempt to convert something that won’t convert, for example, attempting to convert ‘abc’ to an integer, then PostgresSQL will raise an error.

|  |
| --- |
| SELECT  CAST ('abc' AS INTEGER);  [Err] ERROR: invalid input syntax for integer: "abc"  LINE 2: CAST ('abc' AS INTEGER); |

Other errors include attempting to cast to a non-existent data type. For example:

|  |
| --- |
| SELECT  CAST ('10.2' AS DOUBLE);  [Err] ERROR: type "double" does not exist  LINE 2: CAST ('10.2' AS DOUBLE) |

The correct syntax would be to cast ’10.2’ as a Double Precision data type.

|  |
| --- |
| SELECT  CAST ('10.2' AS DOUBLE PRECISION); |

Additionally, you can use the notational shorthand ‘::’ to automatically convert data types. For example:

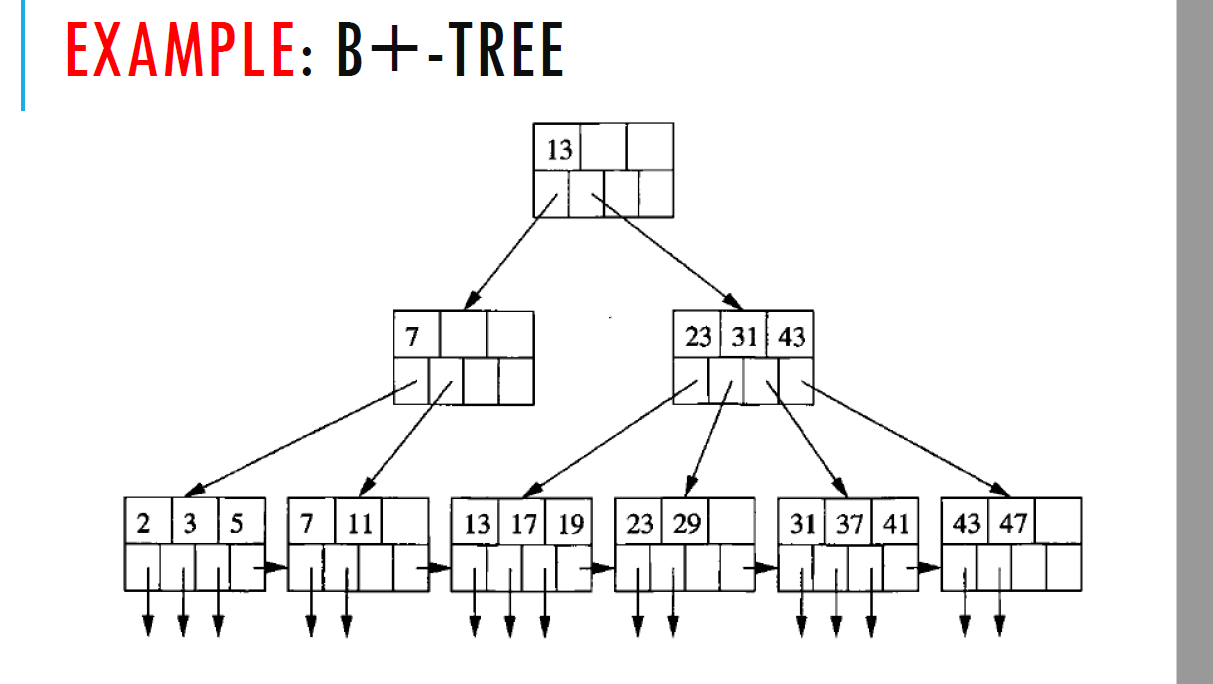
|  |
| --- |
| SELECT  '100'::INTEGER;    SELECT  '01-OCT-2015'::DATE; |

# Indexes

What are indexes?

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book. There are a number of different types of index implementations available, however, the one we’ll examine in detail is the B+-Tree. This is the default data structure used by PostgresSQL for implementing indexes.

Here is a graphical representation of a B+ Tree.



The advantages of a B+ tree are that they significantly speed up SELECT queries on tables, however, they suffer a performance penalty when users request INSERT, UPDATE or DELETE operations on the table.

Let’s take a look at the index types available in PostgresSQL.

* Single Index
* Multicolumn Index
* Unique Index
* Partial Index
* Implicit Index

Indexes are designed to sort tables into an order that makes it easier and faster to query and retrieve data. A single index sorts the table on a single column. Here is the syntax for creating a single index.

|  |
| --- |
| CREATE INDEX index\_name  ON table\_name (column\_name); |

A multicolumn index I defined on more than one column of a table. The basic syntax for creating a multicolumn index is a follows:

|  |
| --- |
| CREATE INDEX index\_name  ON table\_name (column1\_name, column2\_name); |

Whether to create a single-column index or a multicolumn index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the multicolumn index would be the best choice.

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

|  |
| --- |
| CREATE UNIQUE INDEX index\_name  on table\_name (column\_name); |

A partial index is an index built over a subset of a table; the subset is defined by a conditional expression (called the predicate of the partial index). The index contains entries only for those table rows that satisfy the predicate. The basic syntax is as follows

|  |
| --- |
| CREATE INDEX index\_name  on table\_name (conditional\_expression); |

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

Let us consider the following example. We have the company table in our database that we want to create an index for.

|  |
| --- |
| # CREATE INDEX salary\_index ON COMPANY (salary);  # \d company  Table "public.company"  Column | Type | Modifiers  ---------+---------------+-----------  id | integer | not null  name | text | not null  age | integer | not null  address | character(50) |  salary | real |  Indexes:  "company\_pkey" PRIMARY KEY, btree (id)  "salary\_index" btree (salary) |

Note that we have two indexes, an implicit index for the company primary key field and one for the salary index that we have created.

We can also delete indexes using the DROP INDEX command. For example:

|  |
| --- |
| DROP INDEX index\_name; |

So, if we want to drop the salary index, we can do the following:

|  |
| --- |
| # DROP INDEX salary\_index; |

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered −

* Indexes should not be used on small tables
* Tables that have frequent, large batch update or insert operations.
* Indexes should not be used on columns that contain a high number of NULL values.
* Columns that are frequently manipulated should not be indexed.

# Table Views

Views are pseudo-tables. That is, they are not real tables; nevertheless appear as ordinary tables to SELECT. A view can represent a subset of a real table, selecting certain columns or certain rows from an ordinary table. A view can even represent joined tables. Because views are assigned separate permissions, you can use them to restrict table access so that the users see only specific rows or columns of a table.

A view can contain all rows of a table or selected rows from one or more tables. A view can be created from one or many tables, which depends on the written PostgreSQL query to create a view.

Views, which are kind of virtual tables, allow users to do the following −

Structure data in a way that users or classes of users find natural or intuitive.

Restrict access to the data such that a user can only see limited data instead of complete table.

Summarize data from various tables, which can be used to generate reports.

Since views are not ordinary tables, you may not be able to execute a DELETE, INSERT, or UPDATE statement on a view. However, you can create a RULE to correct this problem of using DELETE, INSERT or UPDATE on a view.

The basic syntax for creating a view is as follows:

|  |
| --- |
| CREATE [TEMP | TEMPORARY] VIEW view\_name AS  SELECT column1, column2.....  FROM table\_name  WHERE [condition]; |

Let’s consider the company table and see how we can create views from it.

|  |
| --- |
| testdb=# CREATE VIEW COMPANY\_VIEW AS  SELECT ID, NAME, AGE  FROM COMPANY; |

Now, you can query COMPANY\_VIEW in a similar way as you query an actual table. Following is the example

|  |
| --- |
| testdb=# SELECT \* FROM COMPANY\_VIEW;  id | name | age  ----+-------+-----  1 | Paul | 32  2 | Allen | 25  3 | Teddy | 23  4 | Mark | 25  5 | David | 27  6 | Kim | 22  7 | James | 24  (7 rows) |

As with indexes, we can delete views using the drop view syntax.

|  |
| --- |
| testdb=# DROP VIEW view\_name; |

To drop our company view we do the following:

|  |
| --- |
| testdb=# DROP VIEW COMPANY\_VIEW; |

# Constraints

Constraints are the rules enforced on data columns on table. These are used to prevent invalid data from being entered into the database. This ensures the accuracy and reliability of the data in the database.

Constraints could be column level or table level. Column level constraints are applied only to one column whereas table level constraints are applied to the whole table. Defining a data type for a column is a constraint in itself. For example, a column of type DATE constrains the column to valid dates.

The following are commonly used constraints available in PostgresSQL.

* NOT NULL Constraint − Ensures that a column cannot have NULL value.
* UNIQUE Constraint − Ensures that all values in a column are different.
* PRIMARY Key − Uniquely identifies each row/record in a database table.
* FOREIGN Key − Constrains data based on columns in other tables.
* CHECK Constraint − The CHECK constraint ensures that all values in a column satisfy certain conditions.
* EXCLUSION Constraint − The EXCLUDE constraint ensures that if any two rows are compared on the specified column(s) or expression(s) using the specified operator(s), not all of these comparisons will return TRUE.

## NOT NULL Constraint

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column. A NOT NULL constraint is always written as a column constraint.

A NULL is not the same as no data; rather, it represents unknown data.

Example

For example, the following PostgreSQL statement creates a new table called COMPANY1 and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULL values −

|  |
| --- |
| CREATE TABLE COMPANY1(  ID INT PRIMARY KEY NOT NULL,  NAME TEXT NOT NULL,  AGE INT NOT NULL,  ADDRESS CHAR(50),  SALARY REAL  ); |

## UNIQUE Constraint

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the COMPANY table, for example, you might want to prevent two or more people from having identical age.

Example

For example, the following PostgresSQL statement creates a new table called COMPANY3 and adds five columns. Here, AGE column is set to UNIQUE, so that you cannot have two records with same age

|  |
| --- |
| CREATE TABLE COMPANY3(  ID INT PRIMARY KEY NOT NULL,  NAME TEXT NOT NULL,  AGE INT NOT NULL UNIQUE,  ADDRESS CHAR(50),  SALARY REAL DEFAULT 50000.00  ); |

## PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table. There can be more UNIQUE columns, but only one primary key in a table. Primary keys are important when designing the database tables. Primary keys are unique ids.

We use them to refer to table rows. Primary keys become foreign keys in other tables, when creating relations among tables.

A primary key is a field in a table, which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key.

If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

Example

You already have seen various examples above where we have created COMAPNY4 table with ID as primary key.

|  |
| --- |
| CREATE TABLE COMPANY4(  ID INT PRIMARY KEY NOT NULL,  NAME TEXT NOT NULL,  AGE INT NOT NULL,  ADDRESS CHAR(50),  SALARY REAL  ); |

## FOREIGN KEY Constraint

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the referential integrity between two related tables. They are called foreign keys because the constraints are foreign; that is, outside the table. Foreign keys are sometimes called a referencing key.

Example

For example, the following PostgreSQL statement creates a new table called COMPANY5 and adds five columns.

|  |
| --- |
| CREATE TABLE COMPANY6(  ID INT PRIMARY KEY NOT NULL,  NAME TEXT NOT NULL,  AGE INT NOT NULL,  ADDRESS CHAR(50),  SALARY REAL  ); |

For example, the following PostgresSQL statement creates a new table called DEPARTMENT1, which adds three columns. The column EMP\_ID is the foreign key and references the ID field of the table COMPANY6.

|  |
| --- |
| CREATE TABLE DEPARTMENT1(  ID INT PRIMARY KEY NOT NULL,  DEPT CHAR(50) NOT NULL,  EMP\_ID INT references COMPANY6(ID)  ); |

## CHECK Constraint

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and is not entered into the table.

Example

For example, the following PostgreSQL statement creates a new table called COMPANY5 and adds five columns. Here, we add a CHECK with SALARY column, so that you cannot have any SALARY as Zero.

|  |
| --- |
| CREATE TABLE COMPANY5(  ID INT PRIMARY KEY NOT NULL,  NAME TEXT NOT NULL,  AGE INT NOT NULL,  ADDRESS CHAR(50),  SALARY REAL CHECK(SALARY > 0)  ); |

## EXCLUSION Constraint

Exclusion constraints ensure that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of these operator comparisons will return false or null.

Example

For example, the following PostgreSQL statement creates a new table called COMPANY7 and adds five columns. Here, we add an EXCLUDE constraint −

|  |
| --- |
| CREATE TABLE COMPANY7(  ID INT PRIMARY KEY NOT NULL,  NAME TEXT,  AGE INT ,  ADDRESS CHAR(50),  SALARY REAL,  EXCLUDE USING gist  (NAME WITH =,  AGE WITH <>)  ); |

Here, USING gist is the type of index to build and use for enforcement.

You need to execute the command CREATE EXTENSION btree\_gist, once per database. This will install the btree\_gist extension, which defines the exclusion constraints on plain scalar data types.

As we have enforced the age has to be same, let us see this by inserting records to the table −

|  |
| --- |
| INSERT INTO COMPANY7 VALUES(1, 'Paul', 32, 'California', 20000.00 );  INSERT INTO COMPANY7 VALUES(2, 'Paul', 32, 'Texas', 20000.00 );  INSERT INTO COMPANY7 VALUES(3, 'Allen', 42, 'California', 20000.00 );  For the first two INSERT statements, the records are added to the COMPANY7 table. For the third INSERT statement, the following error is displayed −  ERROR: duplicate key value violates unique constraint "company7\_pkey"  DETAIL: Key (id)=(3) already exists. |

## Dropping Constraints

To remove a constraint you need to know its name. If the name is known, it is easy to drop. Else, you need to find out the system-generated name. The psql command \d table name can be helpful here. The general syntax is

|  |
| --- |
| ALTER TABLE table\_name DROP CONSTRAINT some\_name; |

# Triggers

PostgreSQL Triggers are database callback functions, which are automatically performed/invoked when a specified database event occurs.

The following are important points about PostgreSQL triggers –

* PostgreSQL trigger can be specified to fire
  + Before the operation is attempted on a row (before constraints are checked and the INSERT, UPDATE or DELETE is attempted)
  + After the operation has completed (after constraints are checked and the INSERT, UPDATE, or DELETE has completed)
  + Instead of the operation (in the case of inserts, updates or deletes on a view)
* A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies.
* Both, the WHEN clause and the trigger actions, may access elements of the row being inserted, deleted or updated using references of the form NEW.column-name and OLD.column-name, where column-name is the name of a column from the table that the trigger is associated with.
* If a WHEN clause is supplied, the PostgreSQL statements specified are only executed for rows for which the WHEN clause is true. If no WHEN clause is supplied, the PostgreSQL statements are executed for all rows.
* If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.
* The BEFORE, AFTER or INSTEAD OF keyword determines when the trigger actions will be executed relative to the insertion, modification or removal of the associated row.
* Triggers are automatically dropped when the table that they are associated with is dropped.
* The table to be modified must exist in the same database as the table or view to which the trigger is attached and one must use just tablename, not database.tablename.
* A CONSTRAINT option when specified creates a constraint trigger. This is the same as a regular trigger except that the timing of the trigger firing can be adjusted using SET CONSTRAINTS. Constraint triggers are expected to raise an exception when the constraints they implement are violated.

The basic syntax of creating a trigger is as follows −

|  |
| --- |
| CREATE TRIGGER trigger\_name [BEFORE|AFTER|INSTEAD OF] event\_name  ON table\_name  [  -- Trigger logic goes here....  ]; |

Here, event\_name could be INSERT, DELETE, UPDATE, and TRUNCATE database operation on the mentioned table table\_name. You can optionally specify FOR EACH ROW after table name.

The following is the syntax of creating a trigger on an UPDATE operation on one or more specified columns of a table as follows −

|  |
| --- |
| CREATE TRIGGER trigger\_name [BEFORE|AFTER] UPDATE OF column\_name  ON table\_name  [  -- Trigger logic goes here....  ]; |

Example

Let us consider a case where we want to keep audit trial for every record being inserted in COMPANY table, which we will create newly as follows.

|  |
| --- |
| testdb=# CREATE TABLE COMPANY(  ID INT PRIMARY KEY NOT NULL,  NAME TEXT NOT NULL,  AGE INT NOT NULL,  ADDRESS CHAR(50),  SALARY REAL  ); |

To keep audit trial, we will create a new table called AUDIT where log messages will be inserted whenever there is an entry in COMPANY table for a new record −

|  |
| --- |
| testdb=# CREATE TABLE AUDIT(  EMP\_ID INT NOT NULL,  ENTRY\_DATE TEXT NOT NULL  ); |

Here, ID is the AUDIT record ID, and EMP\_ID is the ID, which will come from COMPANY table, and DATE will keep timestamp when the record will be created in COMPANY table. So now, let us create a trigger on COMPANY table as follows −

|  |
| --- |
| testdb=# CREATE TRIGGER example\_trigger AFTER INSERT ON COMPANY  FOR EACH ROW EXECUTE PROCEDURE auditlogfunc(); |

Where auditlogfunc() is a PostgresSQL procedure and has the following definition −

|  |
| --- |
| CREATE OR REPLACE FUNCTION auditlogfunc() RETURNS TRIGGER AS $example\_table$  BEGIN  INSERT INTO AUDIT(EMP\_ID, ENTRY\_DATE) VALUES (new.ID, current\_timestamp);  RETURN NEW;  END;  $example\_table$ LANGUAGE plpgsql; |

Now, we will start the actual work. Let us start inserting record in COMPANY table which should result in creating an audit log record in AUDIT table. So let us create one record in COMPANY table as follows −

|  |
| --- |
| testdb=# INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)  VALUES (1, 'Paul', 32, 'California', 20000.00 );  id | name | age | address | salary  ----+------+-----+--------------+--------  1 | Paul | 32 | California | 20000  emp\_id | entry\_date  --------+-------------------------------  1 | 2013-05-05 15:49:59.968+05:30  (1 row) |

At the same time, one record will be created in AUDIT table. This record is the result of a trigger, which we have created on INSERT operation on COMPANY table. Similarly, you can create your triggers on UPDATE and DELETE operations based on your requirements.

You can list down all the triggers in the current database from pg\_trigger table as follows:

|  |
| --- |
| testdb=# SELECT \* FROM pg\_trigger; |

The above given PostgreSQL statement will list down all triggers.

If you want to list the triggers on a particular table, then use AND clause with table name as follows −

|  |
| --- |
| testdb=# SELECT tgname FROM pg\_trigger, pg\_class WHERE tgrelid=pg\_class.oid AND relname='company'; |

The above given PostgreSQL statement will also list down only one entry as follows −

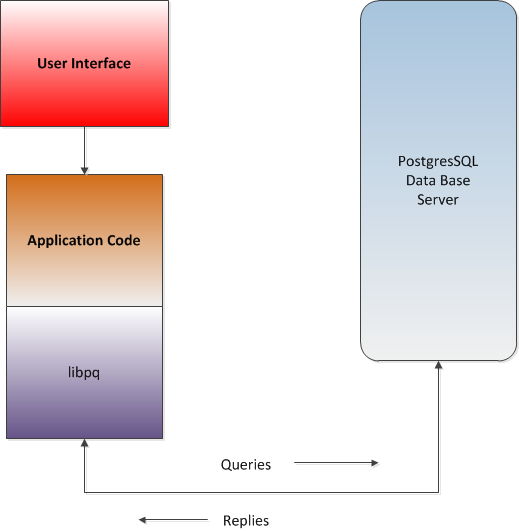
|  |
| --- |
| tgname  -----------------  example\_trigger  (1 row) |

The following is the DROP command, which can be used to drop an existing trigger:

|  |
| --- |
| testdb=# DROP TRIGGER trigger\_name; |

# PostgresSQL Query Optmization

The PostgresSQL query optimizer is the ‘brain’ of the database. The optimizer interprets SQL queries and determines the fastest method of execution. We can use the SQL EXPLAIN command to show how the optimizer interprets queries and determines optimal execution.



When PostgresSQL receives a query from the user, it parses the query and then may rewrite it. From this query it then generates an *execution plan*.

This plan can be shown with the EXPLAIN command which will print out host the database server plans to execute the query.

When postgres receives the query the first thing it does is to parse it and probably rewrites it, usually this is not a problem when thinking about performance. The next thing it does is to generate an execution plan, actually it does more than this but lets keep thing simple.

To optimize queries, first we want to log queries that take longer than a certain time.

We can set the parameter log\_min\_duration\_statement = 300 (or whatever value you desire) in the postgresql.conf file located by default in /etc/postgresql/<version number>/main

This will log slow queries to the postgressql log file. For example:

|  |
| --- |
| 2016-11-15 15:06:10 ART LOG: duration: 300898.426 ms statement: SELECT example\_table.id AS example\_table\_id, example\_table.identification AS example\_table\_identification, example\_table.column1 AS example\_table\_column1, example\_table.column2 AS example\_table\_column2, example\_table.column4 AS example\_table\_column4, example\_table.column3 AS example\_table\_column3, example\_table.column5 AS example\_table\_column5, example\_table."column6" AS "example\_table\_column6", example\_table.psa AS example\_table\_psa, example\_table.fingerprint AS example\_table\_fingerprint, example\_table.duplicate AS example\_table\_duplicate  FROM example\_table  WHERE example\_table.column1 <= 444.53736 AND example\_table.column1 >= 414.53736 AND example\_table.column2 <= 4.0 AND example\_table.column2 >= 4.0 AND example\_table.column4 <= 1 AND example\_table.column4 >= 1 AND example\_table.column3 <= 6 AND example\_table.column3 >= 6 AND example\_table."column6" <= 4.7748 AND example\_table."column6" >= 2.7748 AND example\_table.column5 = 1 |

The above shows a SELECT query that takes over 300 milliseconds.

### Vacuuming.

Before attempting to analyze SELECT statements, it is a good idea to perform proper hygiene on your tables using the VACUUM command. The VACUUM statement is used to reclaim storage by removing obsolete data or tuples from the database.

The syntax for the vacuum command is:

|  |
| --- |
| VACUUM [FULL] [FREEZE] [VERBOSE] [table\_name ];  OR  VACUUM [FULL] [FREEZE] [VERBOSE]  ANALYZE table\_name [ (col1, col2, ... col\_n) ]; |

Reclaim Space to be Reused by Same Table

This first example shows how to reclaim space so that the unused space can be used by the same table. It does not reduce the size of the PostgreSQL database file as the space is not reclaimed by the operating system, only by the table from which the space was allocated.

For example:

|  |
| --- |
| VACUUM; |

This example would vacuum all tables within the database. It would free up the space within each table and leave the space available to be reused by the same table. It does not return the space to the operating system, therefore, the size of the database file would not be reduced.

Reclaim Space and Minimize Database File

If you wanted to vacuum all tables and minimize the database file by returning the unused space to the operating system, you would run the following vacuum statement:

|  |
| --- |
| VACUUM FULL; |

This example would rewrite all tables into a new file, thus requiring an exclusive lock on each table. The database file would be minimized as all of the unused space is reclaimed back to the operating system.

Reclaim Space on a Table

Next, let's look at how to vacuum a specific table, instead of the entire database.

For example:

VACUUM products;

This example would vacuum only the products table. It would free up the space within the products table and leave the space available to be used by only the products table. The size of the database file would not be reduced.

If you wanted to allocate the unused space back to the operating system, you would have to add the FULL option to the VACUUM statement as follows:

|  |
| --- |
| VACUUM FULL products; |

This would not only free up the unused space in the products table, but it would also allow the operating system to reclaim the space and reduce the database size.

Vacuum Activity Report

Finally, you can add the VERBOSE option to the VACUUM command to display an activity report of the vacuum process.

For example:

|  |
| --- |
| VACUUM FULL VERBOSE products; |

This would perform a full vacuum of the products table. Let's show you what you can expect to see as output for a vacuum activity report:

|  |
| --- |
| testdb=# VACUUM FULL VERBOSE products;  INFO: vacuuming "public.products"  INFO: "products": found 4 removable, 5 nonremovable row versions in 1 pages  DETAIL: 0 dead row versions cannot be removed yet.  CPU 0.00s/0.00u sec elapsed 0.04 sec.  VACUUM |

This activity report will display the tables that are vacuumed as well as the details and time taken to perform the vacuum operation.

## Autovacuuming

Introduced in PostgreSQL 8.1, the AUTOVACUUM daemon is an optional feature that automatically vacuums the database so that you don't have to manually run the VACUUM statement. The AUTOVACUUM daemon is enabled in the default configuration.

The AUTOVACUUM daemon is made up of multiple processes that reclaim storage by removing obsolete data or tuples from the database. It checks for tables that have a significant number of inserted, updated, or deleted records and vacuums these tables based on the configuration settings below.

The AUTOVACUUM settings for PostgreSQL can be found in the postgresql.conf file and control when/how the daemon runs. The following is an example of these settings:

|  |
| --- |
| #------------------------------------------------------------------------------  # AUTOVACUUM PARAMETERS  #------------------------------------------------------------------------------  autovacuum = on # Enable autovacuum subprocess? 'on'  # requires track\_counts to also be on.  log\_autovacuum\_min\_duration = -1 # -1 disables, 0 logs all actions and  # their durations, > 0 logs only  # actions running at least this number  # of milliseconds.  autovacuum\_max\_workers = 3 # max number of autovacuum subprocesses  # (change requires restart)  autovacuum\_naptime = 1min # time between autovacuum runs  autovacuum\_vacuum\_threshold = 50 # min number of row updates before vacuum  autovacuum\_analyze\_threshold = 50 # min number of row updates before analyze  autovacuum\_vacuum\_scale\_factor = 0.2 # fraction of table size before vacuum  autovacuum\_analyze\_scale\_factor = 0.1 # fraction of table size before analyze  autovacuum\_freeze\_max\_age = 200000000 # maximum XID age before forced vacuum  # (change requires restart)  autovacuum\_multixact\_freeze\_max\_age = 400000000  # maximum multixact age  # before forced vacuum  # (change requires restart)  autovacuum\_vacuum\_cost\_delay = 20ms # default vacuum cost delay for  # autovacuum, in milliseconds;  # -1 means use vacuum\_cost\_delay  autovacuum\_vacuum\_cost\_limit = -1 # default vacuum cost limit for  # autovacuum, -1 means use  # vacuum\_cost\_limit |

These settings are located in the postgresql.conf file. Note that when the auto vacuum is running, it will slow down the databases, so care has to be taken as to when the auto vacuum is run with production databases.

Consider the following table schema:

|  |
| --- |
| Table "public.example\_table"  Column | Type | Modifiers  ------------------------+-------------------+-------------------------------------------------------  id | integer | not null default nextval('example\_table\_id\_seq'::regclass)  identification | character varying |  column1 | double precision |  column2 | integer |  column4 | integer |  column3 | integer |  column5 | integer |  column6 | double precision |  fingerprint | character varying |  duplicate | boolean |  Indexes:  "example\_table\_pkey" PRIMARY KEY, btree (id)  "example\_table\_identification\_key" UNIQUE CONSTRAINT, btree (identification)  "example\_table\_identification\_idx" btree (identification)  "multi\_index" btree (column1, column2, column4, column3, "column6", column5)  Referenced by:  TABLE "history" CONSTRAINT "history\_id\_fkey" FOREIGN KEY (decoy\_id) REFERENCES example\_table(id)  TABLE "result" CONSTRAINT "result\_id\_fkey" FOREIGN KEY (decoy\_id) REFERENCES example\_table(id) |

Using the slow queries in our log, we are going to use the explain analyze to diagnose the possible problems. Assume that the table is populated with 100 Millon of rows using the same data distribution of the production environment.

Here

|  |
| --- |
| EXPLAIN ANALYZE SELECT example\_table.id AS example\_table\_id, example\_table.identification AS example\_table\_identification, example\_table.column1 AS example\_table\_column1, example\_table.column2 AS example\_table\_column2, example\_table.column4 AS example\_table\_column4, example\_table.column3 AS example\_table\_column3, example\_table.column5 AS example\_table\_column5, example\_table."column6" AS "example\_table\_column6", example\_table.psa AS example\_table\_psa, example\_table.fingerprint AS example\_table\_fingerprint, example\_table.duplicate AS example\_table\_duplicate |

|  |
| --- |
| Index Scan using multi\_index on example\_table (cost=0.57..664243.33 rows=98 width=212) (actual time=0.943..387378.003 rows=5685 loops=1)  Index Cond: ((column1 <= 284.3018::double precision) AND (column1 >= 254.3018::double precision) AND (column4 <= 3) AND (column4 >= 3) AND (column3 <= 5) AND (column3 >= 5) AND ("column6" <= 2.4414::double precision) AND ("column6" >= 0.4414::double precision) AND (column5 = 0))  Filter: (((column2)::numeric <= 5.0) AND ((column2)::numeric >= 5.0))  Rows Removed by Filter: 27429  Total runtime: 387403.832 ms  (5 rows) |

As we can see postgresSQL is using the multi\_index as we expect. However the execution time was totally unacceptable. Since we have an index scan we also execute a VACUUM FULL. Since we use equality on column5 and column3, lets try to add a partial index using in those columns. Recall that a partial index is an index over a subset of the table based on a predicate condition.

In particular column5 partitions our table in two, but column3 was not the case. we usually execute queries with other values, like 2 and 1.

CREATE INDEX column3\_and\_column5\_idx ON example\_table ON example\_table where column5=0 and column3=1 or column3=2 or column3=5; CREATE INDEX column3\_and\_column5\_idx ON example\_table ON example\_table where column5=1 and column3=1 or column3=2 or column3=5;