Python for PostgresSQL developers

A brief history of PostgresSQL

A Brief History of PostgreSQL

PostgreSQL, originally called Postgres, was created at UCB by a computer science professor named Michael Stonebraker. Stonebraker started Postgres in 1986 as a follow-up project to its predecessor, Ingres, now owned by Computer Associates.

1977-1985 – A project called INGRES was developed.

Proof-of-concept for relational databases

Established the company Ingres in 1980

Bought by Computer Associates in 1994

1986-1994 – POSTGRES

Development of the concepts in INGRES with a focus on object orientation and the query language - Quel

The code base of INGRES was not used as a basis for POSTGRES

Commercialized as Illustra (bought by Informix, bought by IBM)

1994-1995 – Postgres95

Support for SQL was added in 1994

Released as Postgres95 in 1995

Re-released as PostgreSQL 6.0 in 1996

Establishment of the PostgreSQL Global Development Team

Key Features of PostgresSQL

PostgreSQL runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It supports text, images, sounds, and video, and includes programming interfaces for C / C++, Java, Perl, Python, Ruby, Tcl and Open Database Connectivity (ODBC).

PostgreSQL supports a large part of the SQL standard and offers many modern features including the following –

Complex SQL queries

SQL Sub-selects

Foreign keys

Trigger

Views

Transactions

Multiversion concurrency control (MVCC)

Streaming Replication (as of 9.0)

Hot Standby (as of 9.0)

You can check official documentation of PostgreSQL to understand the above-mentioned features. PostgreSQL can be extended by the user in many ways. For example by adding new –

Data types

Functions

Operators

Aggregate functions

Index methods

Language support

PostgreSQL supports four standard procedural languages, which allows the users to write their own code in any of the languages and it can be executed by PostgreSQL database server. These procedural languages are - PL/pgSQL, PL/Tcl, PL/Perl and PL/Python. Besides, other non-standard procedural languages like PL/PHP, PL/V8, PL/Ruby, PL/Java, etc., are also supported.

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,	6 decimal digits

		inexact	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	(a <> b) is true.

	values are not equal then condition becomes 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

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:

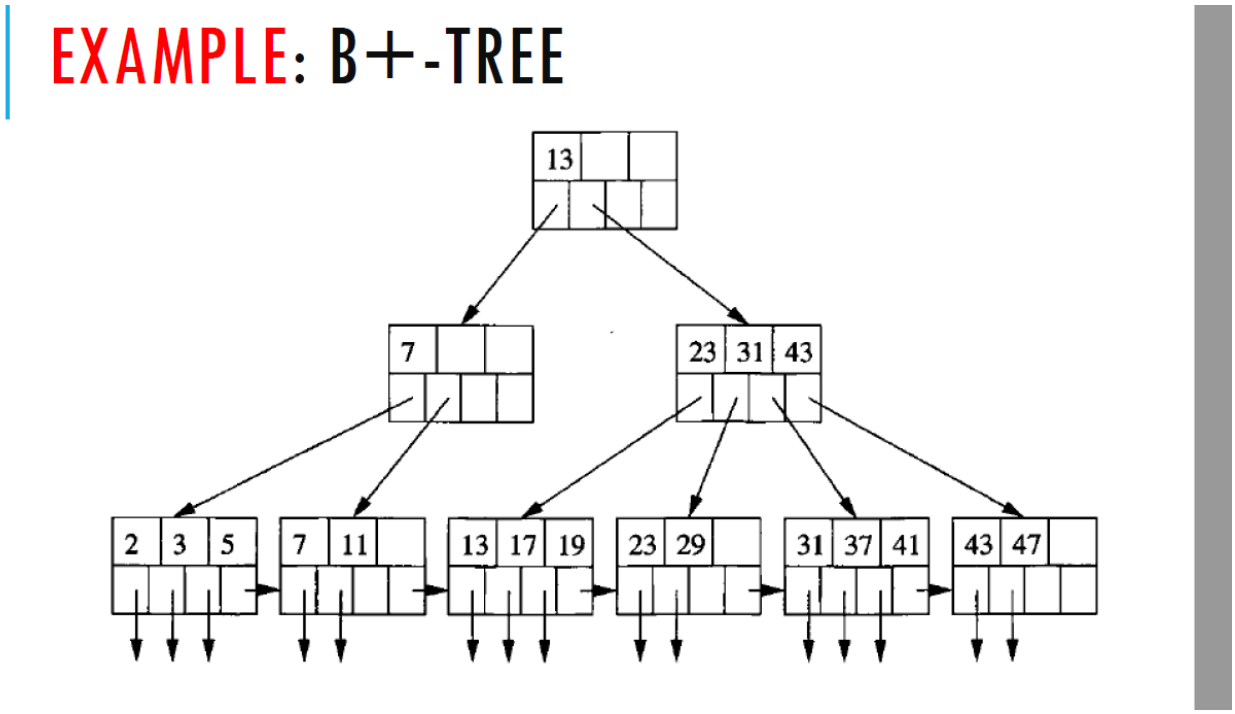
```
SELECT
 '100'::INTEGER;

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 is 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 PostgreSQL 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)
);
```

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)
```

Here is an example of a Postgres stored procedure in Python.

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

The above function returns the greater of the two parameters by value.

Note that because of how these functions are actually defined in Python, you cannot reassign the value of a parameter inside the function without using the global keyword. For example, you cannot do this:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # error
  return x
$$ LANGUAGE plpythonu;
```

Instead, you must use the global statement like so:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # ok now
  return x
$$ LANGUAGE plpythonu;
```

However, it is better practice not to rely on this implementation detail. The best way forward is to treat function parameters as read-only variables.

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}
```

Postgres 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.

Syntax

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....
];
```

Here's an example of a trigger

```
testdb=# CREATE TRIGGER example_trigger AFTER INSERT ON COMPANY
FOR EACH ROW EXECUTE PROCEDURE auditlogfunc();
```

And here's the audit log function referred to by the trigger.

```
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;
```

So, assuming that we're doing an INSERT on the company table with the following data:

```
testdb=# INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

This will insert a record in to the company table:

```
id | name | age | address      | salary
----+------+-----+-------------+--------
  1 | Paul |  32 | California   |  20000
```

But, the trigger will also call the auditlog function which will insert a record into the audit table like so:

```
emp_id |          entry_date
--------+-------------------------------
      1 | 2013-05-05 15:49:59.968+05:30
```

We can also delete triggers by DROPping them. Like so:

```
testdb=# DROP TRIGGER trigger_name;
```

Using the psycopg2 library to access Postgres from Python.

Psycopg2 is a DB API 2.0 compliant PostgreSQL driver that is actively developed. It is designed for multi-threaded applications and manages its own connection pool. Other interesting features of the adapter are that if you are using the PostgreSQL array data type, Psycopg will automatically convert a result using that data type to a Python list.

Here is an example of connecting to Postgres from Python.

```
#!/usr/bin/python3
#
# Small script to show PostgreSQL and Pyscopg together
#

import psycopg2

try:
    conn = psycopg2.connect("dbname='template1' user='dbuser'
host='localhost' password='dbpass'")
except:
    print ("I am unable to connect to the database")
```

The above will import the adapter and try to connect to the database. If the connection fails a print statement will occur to STDOUT. You could also use the exception to try the connection again with different parameters if you like.

The next step is to define a cursor to work with. It is important to note that Python/Psycopg cursors are not cursors as defined by PostgreSQL. They are completely different beasts.

```
#/usr/bin/python3
#
#

import psycopg2

# Try to connect

try:
    conn=psycopg2.connect("dbname='template1' user='dbuser'
password='mypass'")
except:
    print "I am unable to connect to the database."

cur = conn.cursor()
try:
    cur.execute("""SELECT datname from pg_database""")
    rows = cur.fetchall()
    print ("\nShow me the databases:\n")
    for row in rows:
        print ("   ", row[0])
except:
    print ("I can't drop our test database!")
```

Note that we have a number of functions available to the cursor module.

1	psycopg2.connect(database="testdb", user="postgres", password="cohondob", host="127.0.0.1", port="5432") This API opens a connection to the PostgreSQL database. If database is opened successfully, it returns a connection object.
2	connection.cursor() This routine creates a cursor which will be used throughout of your database programming with Python.
3	cursor.execute(sql [, optional parameters]) This routine executes an SQL statement. The SQL statement may be parameterized (i.e., placeholders instead of SQL literals). The psycopg2 module supports placeholder using %s sign For example:cursor.execute("insert into people values (%s, %s)", (who, age))
4	cursor.executemany(sql, seq_of_parameters) This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.
5	cursor.callproc(procname[, parameters]) This routine executes a stored database procedure with the given

	name. The sequence of parameters must contain one entry for each argument that the procedure expects.
6	cursor.rowcount This read-only attribute which returns the total number of database rows that have been modified, inserted, or deleted by the last last execute*().
7	connection.commit() This method commits the current transaction. If you do not call this method, anything you did since the last call to commit() is not visible from other database connections.
8	connection.rollback() This method rolls back any changes to the database since the last call to commit().
9	connection.close() This method closes the database connection. Note that this does not automatically call commit(). If you just close your database connection without calling commit() first, your changes will be lost!``
10	cursor.fetchone() This method fetches the next row of a query result set, returning a single sequence, or None when no more data is available.
11	cursor.fetchmany([size=cursor.arraysize]) This routine fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available. The method tries to fetch as many rows as indicated by the size parameter.
12	cursor.fetchall() This routine fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available.

Here is an example of creating a table.

```
#!/usr/bin/python3

import psycopg2

conn = psycopg2.connect(database = "testdb", user = "postgres",
password = "pass123", host = "127.0.0.1", port = "5432")
print "Opened database successfully"

cur = conn.cursor()
cur.execute('''CREATE TABLE COMPANY
      (ID INT PRIMARY KEY     NOT NULL,
      NAME           TEXT    NOT NULL,
      AGE            INT     NOT NULL,
      ADDRESS        CHAR(50),
      SALARY         REAL);''')
print ("Table created successfully")

conn.commit()
conn.close()
```

Here's an example of an INSERT.

```
#!/usr/bin/python3

import psycopg2

conn = psycopg2.connect(database = "testdb", user = "postgres",
password = "pass123", host = "127.0.0.1", port = "5432")
print ("Opened database successfully")

cur = conn.cursor()

cur.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'Paul', 32, 'California', 20000.00 )");

cur.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

cur.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

cur.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print ("Records created successfully";)
conn.close()
```

Here is an example of a SELECT statement.

```
#!/usr/bin/python3

import psycopg2

conn = psycopg2.connect(database = "testdb", user = "postgres",
password = "pass123", host = "127.0.0.1", port = "5432")
print ("Opened database successfully")

cur = conn.cursor()

cur.execute("SELECT id, name, address, salary  from COMPANY")
rows = cur.fetchall()
for row in rows:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print ("Operation done successfully";)
conn.close()
```

Using other types of cursors.

The dict cursors allow to access to the retrieved records using an interface similar to the Python dictionaries instead of the tuples.

Here is an example. Note that we have to import the psycopg2.extras module as well.

```
dict_cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
dict_cur.execute("INSERT INTO test (num, data) VALUES(%s, %s)",(100,
"abc'def"))
dict_cur.execute("SELECT * FROM test")
rec = dict_cur.fetchone()
print (rec['id'])
print(rec['num'])
print (rec['data'])
```

Using the NamedTuple cursor

These objects require collections.namedtuple() to be found, so it is available out-of-the-box only from Python 2.6.

```
import collections
collections.namedtuple = namedtuple
from psycopg.extras import NamedTupleConnection
nt_cur = conn.cursor(cursor_factory=psycopg2.extras.NamedTupleCursor)
```

```
rec = nt_cur.fetchone()
print (rec)
```

Using bound parameters in a SQL statement

It is highly recommended that you use parameterized SQL queries, which means that instead of supplying the values directly inside an SQL statement, you supply variable names to the SQL query and let Python replace them with their stored values, like so:

```
cursor.execute('SELECT * from table where id = %(some_id)d',
{'some_id': 1234})
```

Here's another example using a dictionary to store the parameters, and then iterating through the dictionary key to supply the values to the SQL query .

```
fields = ', '.join(my_dict.keys())
values = ', '.join(['%%(%s)s' % x for x in my_dict])
query = 'INSERT INTO some_table (%s) VALUES (%s)' % (fields, values)
cursor.execute(query, my_dict)
```

Note that values should be specified in the code, not via user input, in order to prevent SQL injection attacks.

Warning Never, never, NEVER use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string.

The correct way to pass variables in a SQL command is using the second argument of the execute() method:

```
SQL = "INSERT INTO authors (name) VALUES (%s);" # Note: no quotes
data = ("O'Reilly", )
cur.execute(SQL, data) # Note: no % operator
```

Remember that a major exploit is to supply a string with only one apostrophe, which can cause an error and allow malicious SQL code to be inserted afterwards.

Doing this the correct way ensure that all data passed to the cursor is a string, and not malicious SQL code. Effectively, single quotes will be preceeded with a backslash (\) which escapes it.

Introduction to SQL Alchemy

A common task when programming any web service is the construction of a solid database backend. In the past, programmers would write raw SQL statements, pass them to the database engine and parse the returned results as a normal array of records. Nowadays, programmers can write Object-relational mapping (ORM) programs to remove the necessity of writing tedious and error-prone raw SQL statements that are inflexible and hard-to-maintain.

ORM is a programming technique for converting data between incompatible type systems in object-oriented programming languages. Usually, the type system used in an OO language such as Python contains types that are non-scalar, namely that those types cannot be expressed as primitive types such as integers and strings. For example, a Person object may have a list of Address objects and a list of PhoneNumber objects associated with it. In turn, an Address object may have a PostCode object, a StreetName object and a StreetNumber object associated with it. Although simple objects such as PostCodes and StreetNames can be expressed as strings, a complex object such as a Address and a Person cannot be expressed using only strings or integers. In addition, these complex objects may also include instance or class methods that cannot be expressed using a type at all.

In order to deal with the complexity of managing objects, people developed a new class of systems called ORM. Our previous example can be expressed as an ORM system with a Person class, a Address class and a PhoneNumber class, where each class maps to a table in the underlying database. Instead of writing tedious database interfacing code yourself, an ORM takes care of these issues for you while you can focus on programming the logics of the system.

In this design, we have two tables person and address and address.person_id is a foreign key to the person table. Now we write the corresponding database initialization code in a file sqlInit_ex.py.

This is the old way to write the code.

```
import psycopg2
conn = psycopg2.connect('example.db')

c = conn.cursor()
c.execute('''
          CREATE TABLE person
          (id INTEGER PRIMARY KEY ASC, name varchar(250) NOT NULL)
          ''')
c.execute('''
          CREATE TABLE address
          (id INTEGER PRIMARY KEY ASC, street_name varchar(250),
street_number varchar(250),
           post_code varchar(250) NOT NULL, person_id INTEGER NOT
```

```
NULL,
            FOREIGN KEY(person_id) REFERENCES person(id))
          ''')

c.execute('''
          INSERT INTO person VALUES(1, 'pythoncentral')
          ''')
c.execute('''
          INSERT INTO address VALUES(1, 'python road', '1', '00000',
1)
          ''')

conn.commit()
conn.close()
```

And here's how we do it with SQLAlchemy

```
import os
import sys
from sqlalchemy import Column, ForeignKey, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship
from sqlalchemy import create_engine

Base = declarative_base()

class Person(Base):
    __tablename__ = 'person'
    # Here we define columns for the table person
    # Notice that each column is also a normal Python instance
attribute.
    id = Column(Integer, primary_key=True)
    name = Column(String(250), nullable=False)

class Address(Base):
    __tablename__ = 'address'
    # Here we define columns for the table address.
    # Notice that each column is also a normal Python instance
attribute.
    id = Column(Integer, primary_key=True)
    street_name = Column(String(250))
    street_number = Column(String(250))
    post_code = Column(String(250), nullable=False)
    person_id = Column(Integer, ForeignKey('person.id'))
    person = relationship(Person)

# Create an engine that stores data in the local directory's
```

```
# sqlalchemy_example.db file.
engine = create_engine('postgres:///sqlalchemy_example.db')

# Create all tables in the engine. This is equivalent to "Create
Table"
# statements in raw SQL.
Base.metadata.create_all(engine)
```