

Python for PostgreSQL developers

A brief history of PostgreSQL

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 PostgreSQL

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.

PostgreSQL Data Types.

PostgreSQL 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.
 - o 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.
 - o 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.
 - o Be mindful of rounding and truncation in the decimal fraction and inadvertent NaN's.

- Avoid floating point data types for currency applications. 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 Number.

`numeric(precision)`

- This is effectively an integer.

An example of the numeric datatype:

```
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 the 'lc_monetary' setting.
- Output based on lc_monetary. E.g.
 - o '\$1000.00'
- The reality.
 - o Don't use the monetary type.
 - o Store currency as Integer or Numeric types.
 - o Money is based on a database wide environment setting.
 - o This setting can change widely between instances.
 - o You cannot control the environment setting for specific database columns.
 - o 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:
 - o Command: SET <datestyle>
 - o Modify postgresql.conf – ‘DateStyle’ parameter
 - o 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 PostgreSQL statement's WHERE clause.
OR	The OR operator is used to combine multiple conditions in a PostgreSQL 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 bitstring 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:

$$A \& B = 0000\ 1100$$

$$A | B = 0011\ 1101$$

$$\sim A = 1100\ 0011$$

Note that the (\sim) symbol is the NOT operator.

The bitwise operators are as follows:

Operator	Description	Example
$\&$ \sim	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 PostgreSQL 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
-----
```

```
(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 PostgreSQL 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 PostgreSQL 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 PostgreSQL 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 PostgreSQL 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;
```

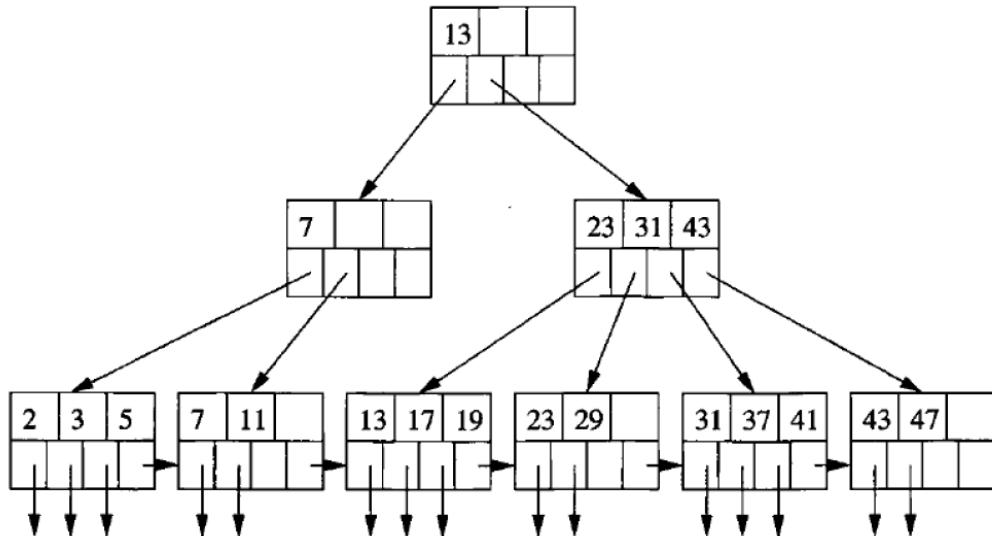
Indexes

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 PostgreSQL for implementing indexes.

Here is a graphical representation of a B+ Tree.

EXAMPLE: 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 PostgreSQL.

- 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 as 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 PostgreSQL.

- 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 PostgreSQL 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 COMPANY4 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 Stored Procedures

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.,

Syntax

The basic syntax to create a function is as follows—

```
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.

```

CREATE OR REPLACE FUNCTION accdisvalues(thisdate date)
RETURNS void AS
$BODY$
    Update my_table
    Set
        mycolumn = true
        where mydatecol = thisdate;
$BODY$
LANGUAGE sql VOLATILE SECURITY DEFINER
COST 100;
ALTER FUNCTION accdisvalues(date)
OWNER TO myconnect;

```

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.

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 Psycopg 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	<code>psycopg2.connect(database="testdb", user="postgres", password="cohondob", host="127.0.0.1", port="5432")</code>
	This API opens a connection to the PostgreSQL database. If database is opened successfully, it returns a connection object.
2	<code>connection.cursor()</code>
	This routine creates a cursor which will be used throughout of your database programming with Python.
3	<code>cursor.execute(sql [, optional parameters])</code>
	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: <code>cursor.execute("insert into people values (%s, %s)", (who, age))</code>
4	<code>cursor.executemany(sql, seq_of_parameters)</code>
	This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.
5	<code>cursor.callproc(procname [, parameters])</code>
	This routine executes a stored database procedure with the given

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

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 psycopg2.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)
    ''')
```

```

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)
```

Postgresql Optimization

Optimization is required because the problem is that every database is not only different in its design, but also its requirements. Some systems are used to log mountains of data that is almost never queried. Others have essentially static data that is queried constantly, sometimes feverishly. Most systems however have some, usually unequal, level of reads and writes to the database. Add this little complexity on top of your totally unique table structure, data, and hardware configuration and hopefully you begin to see why tuning can be difficult.

The default configuration PostgreSQL ships with is a very solid configuration aimed at everyone's best guess as to how an "average" database on "average" hardware should be setup.

Understanding the process

The first step to learning how to tune your PostgreSQL database is to understand the life cycle of a query. Here are the steps of a query:

Transmission of query string to database backend

Parsing of query string

Planning of query to optimize retrieval of data

Retrieval of data from hardware

Transmission of results to client

The first step is the sending of the query string (the actual SQL command you type in or your application uses) to the database backend. There isn't much you can tune about this step, however if you have a very large queries that cannot be prepared in advance it may help to put them into the database as a stored procedure and cut the data transfer down to a minimum.

Once the SQL query is inside the database server it is parsed into tokens. This step can also be minimized by using stored procedures.

The planning of the query is where PostgreSQL really starts to do some work. This stage checks to see if the query is already prepared if your version of PostgreSQL and client library support this feature. It also analyzes your SQL to determine what the most efficient way of retrieving your data is. Should we use an index and if so which one? Maybe a hash join on those two tables is appropriate? These are some of the decisions the database makes at this point of the process. This step can be eliminated if the query is previously prepared.

Now that PostgreSQL has a plan of what it believes to be the best way to retrieve the data, it is time to actually get it. While there are some tuning options that help here, this step is mostly effected by your hardware configuration.

And finally the last step is to transmit the results to the client. While there aren't any real tuning options for this step, you should be aware that all of the data that you are returning is pulled from the disk and sent over the wire to your client. Minimizing the number of rows and columns to only those that are necessary can often increase your performance.

General Tuning

There are several postmaster options that can be set that drastically affect performance, below is a list of the most commonly used and how they effect performance:

`max_connections=<num>` — This option sets the maximum number of database backend to have at any one time. Use this feature to ensure that you do not launch so many backends that you begin swapping to disk and kill the performance of all the children. Depending on your application it may be better to deny the connection entirely rather than degrade the performance of all of the other children.

`shared_buffers=<num>` — Editing this option is the simplest way to improve the performance of your database server. The default is pretty low for most modern hardware. General wisdom says that this should be set to roughly 25% of available RAM on the system. Like most of the options I will outline here you will simply need to try them at different levels (both up and down) and see how well it works on your particular system. Most people find that setting it larger than a third starts to degrade performance.

`effective_cache_size=<num>` — This value tells PostgreSQL's optimizer how much memory PostgreSQL has available for caching data and helps in determining whether or not it uses an index or not. The larger the value increases the likely hood of using an index. This should be set to the amount of memory allocated to shared_buffers plus the amount of OS cache available. Often this is more than 50% of the total system memory.

`work_mem=<num>` — This option is used to control the amount of memory used in sort operations and hash tables. While you may need to increase the amount of memory if you do a ton of sorting in your application, care needs to be taken. This isn't a system wide parameter, but a per operation one. So if a complex query has several sort operations in it it will use multiple work_mem units of memory. Not to mention that multiple backends could be doing this at once. This query can often lead your database server to swap if the value is too large. This option was previously called sort_mem in older versions of PostgreSQL.

`max_fsm_pages=<num>` — This option helps to control the free space map. When something is deleted from a table it isn't removed from the disk immediately, it is simply marked as "free" in the free space map. The space can then be reused for any new INSERTs that you do on the table. If your setup has a high rate of DELETEs and INSERTs it may be necessary increase this value to avoid table bloat.

`fsync= <boolean>` — This option determines if all your WAL pages are fsync()'ed to disk before a transaction is committed. Having this on is safer, but can reduce write performance. If fsync is not enabled there is the chance of unrecoverable data corruption. Turn this off at your own risk.

`commit_delay=<num>` and `commit_siblings=<num>` — These options are used in concert to help improve performance by writing out multiple transactions that are committing at once. If there are `commit_siblings` number of backends active at the instant your transaction is committing then the server waiting `commit_delay` microseconds to try and commit multiple transactions at once.

`random_page_cost=<num>` — `random_page_cost` controls the way PostgreSQL views non-sequential disk reads. A higher value makes it more likely that a sequential scan will be used over an index scan indicating that your server has very fast disks.

Note that many of these options consume shared memory and it will probably be necessary to increase the amount of shared memory allowed on your system to get the most out of these options.

Hardware Issues

Obviously the type and quality of the hardware you use for your database server drastically impacts the performance of your database. Here are a few tips to use when purchasing hardware for your database server (in order of importance):

RAM — The more RAM you have the more disk cache you will have. This greatly impacts performance considering memory I/O is thousands of times faster than disk I/O.

Disk types — Obviously fast Ultra-320 SCSI disks are your best option, however high end SATA drives are also very good. With SATA each disk is substantially cheaper and with that you can afford more spindles than with SCSI on the same budget.

Disk configuration — The optimum configuration is RAID 1+0 with as many disks as possible and with your transaction log (`pg_xlog`) on a separate disk (or stripe) all by itself. RAID 5 is not a very good option for databases unless you have more than 6 disks in your volume. With newer versions of PostgreSQL you can also use the tablespaces option to put different tables, databases, and indexes on different disks to help optimize performance. Such as putting your often used tables on a fast SCSI disk and the less used ones slower IDE or SATA drives.

CPUs — The more CPUs the better, however if your database does not use many complex functions your money is best spent on more RAM or a better disk subsystem.

In general the more RAM and disk spindles you have in your system the better it will perform. This is because with the extra RAM you will access your disks less. And the extra spindles help spread the reads and writes over multiple disks to increase throughput and to reduce drive head congestion.

Another good idea is to separate your application code and your database server onto different hardware. Not only does this provide more hardware dedicated to the database server, but the operating system's disk cache will contain more PostgreSQL data and not other various application or system data this way.

For example, if you have one web server and one database server you can use a cross-over cable on a separate ethernet interface to handle just the web server to database network traffic to ensure you reduce any possible bottlenecks there. You can also obviously create an entirely different physical network for database traffic if you have multiple servers that access the same database server.

Useful Tuning Tools

The most useful tool in tuning your database is the SQL command EXPLAIN ANALYZE. This allows you to profile each SQL query your application performs and see exactly how the PostgreSQL planner will process the query. Let's look at a short example, below is a simple table structure and query.

```
CREATE TABLE authors (
    id      int4 PRIMARY KEY,
    name    varchar
);

CREATE TABLE books (
    id          int4 PRIMARY KEY,
    author_id   int4,
    title       varchar
);
```

If we use the query:

```
EXPLAIN ANALYZE SELECT authors.name, books.title
    FROM books, authors
    WHERE books.author_id=16 and authors.id = books.author_id
    ORDER BY books.title;
```

You will get output similar to the following:

```
QUERY PLAN
```

```
-----  
-----  
Sort  (cost=29.71..29.73 rows=6 width=64) (actual  
time=0.189..16.233 rows=7 loops=1)  
  Sort Key: books.title  
    -> Nested Loop  (cost=0.00..29.63 rows=6 width=64) (actual  
time=0.068..0.129 rows=7 loops=1)  
      -> Index Scan using authors_pkey on authors  
(cost=0.00..5.82 rows=1 width=36) (actual time=0.029..0.033 rows=1  
loops=1)  
        Index Cond: (id = 16)  
      -> Seq Scan on books  (cost=0.00..23.75 rows=6  
width=36) (actual time=0.026..0.052 rows=7 loops=1)  
        Filter: (author_id = 16)  
Total runtime: 16.386 ms
```

You need to read this output from bottom to top when analyzing it. The first thing PostgreSQL does is do a sequence scan on the books table looking at each author_id column for values that equal 16. Then it does an index scan of the authors table, because of the implicit index created by the PRIMARY KEY options. Then finally the results are sorted by books.title.

The values we see in parenthesis are the estimated and actual cost of that portion of the query. The closer together the estimate and the actual costs are the better performance we will typically see.

Now, let's change the structure a little bit by adding an index on books.author_id to avoid the sequence scan with this command:

```
CREATE INDEX books_idx1 on books(author_id);
```

If you rerun the query again, we won't see any noticeable change in the output. This is because PostgreSQL has not yet re-analyzed the data and determined that the new index may help for this query. This can be solved by running:

```
ANALYZE books;
```

However, in this small test case we are working with the planner still favors the sequence scan because there aren't very many rows in the books table. If a query is going to return a large portion of a table then the planner chooses a sequence scan over an index because it is actually faster. You can also force PostgreSQL to favor index scans over sequential scans by setting the configuration parameter enable_seqscan to off. This doesn't remove all sequence scans, since some tables may not have an index, but it does force the planner's hand into always using an index scan when it is available. This is probably best done by sending the command SET enable_seqscan = off at the start of every connection rather than setting this option database wide. This way we can control via your application code when this is in effect. However, in general disabling sequence scans should only be used in tuning our application and is not really intended for every day use.

Typically the best way to optimize our queries is to use indexes on specific columns and combinations of columns to correspond to often used queries. Unfortunately this is done by trial and error. We should also note that increasing the number of indexes on a table increases the number of write operations that need to be performed for each INSERT and UPDATE. So don't do anything silly and just add indexes for each column in each table.

We can help PostgreSQL do what you want by playing with the level of statistics that are gathered on a table or column with the command:

```
ALTER TABLE <table> ALTER COLUMN <column> SET STATISTICS <number>;
```

This value can be a number between 0 and 1000 and helps PostgreSQL determine what level of statistics gathering should be performed on that column. This helps us to control the generated query plans without having slow vacuum and analyze operations because of generating large amounts of stats for all tables and columns.

Another useful tool to help determine how to tune your database is to turn on query logging. We can tell PostgreSQL which queries we are interested in logging via the log_statement configuration option. This is very useful in situations where we have many users executing ad hoc queries to your system via something like Crystal Reports or via psql directly.

Database Design and Layout

Sometimes the design and layout of your database affects performance. For example, if we have an employee database that looks like this:

```
CREATE TABLE employees (
    id                  int4 PRIMARY KEY,
    active              boolean,
    first_name          varchar,
    middle_name         varchar,
    last_name           varchar,
    ssn                varchar,
    address1            varchar,
    address2            varchar,
    city               varchar,
    state              varchar(2),
    zip                varchar,
    home_phone          varchar,
    work_phone          varchar,
    cell_phone          varchar,
    fax_phone           varchar,
    pager_number        varchar,
    business_email      varchar,
    personal_email       varchar,
    salary              int4,
```

```

vacation_days      int2,
sick_days          int2,
employee_number    int4,
office_addr_1      varchar,
office_addr_2      varchar,
office_city        varchar,
office_state       varchar(2),
office_zip         varchar,
department        varchar,
title              varchar,
supervisor_id     int4
);

```

This design is easy to understand, but isn't very good on several levels. While it will depend on our particular application, in most cases we won't need to access all of this data at one time. In portions of our application that deal with HR functions you are probably only interested in their name, salary, vacation time, and sick days. However, if the application displays an organization chart it would only be concerned with the department and supervisor_id portions of the table.

By breaking up this table into smaller tables we can get more efficient queries since PostgreSQL has less to read through, not to mention better functionality. Below is one way to make this structure better:

```

CREATE TABLE employees (
    id                  int4 PRIMARY KEY,
    active             boolean,
    employee_number    int4,
    first_name         varchar,
    middle_name        varchar,
    last_name          varchar,
    department         varchar,
    title              varchar,
    email              varchar
);

CREATE TABLE employee_address (
    id                int4 PRIMARY KEY,
    employee_id       int4,
    personal          boolean,
    address_1         varchar,
    address_2         varchar,
    city              varchar,
    state             varchar(2),
    zip               varchar
);

CREATE TABLE employee_number_type (
    id                int4 PRIMARY KEY,
    type              varchar
);

CREATE TABLE employee_number (

```

```

        id          int4 PRIMARY KEY,
employee_id      int4,
type_id         int4,
number          varchar
) ;

CREATE TABLE employee_hr_info (
        id          int4 PRIMARY KEY,
employee_id      int4,
ssn            varchar,
salary          int4,
vacation_days  int2,
sick_days       int2
) ;

```

With this table structure the data associated with an employee is broken out into logical groupings. The main table contains the most frequently used information and the other tables store all of the rest of the information. The added benefit of this layout is that we can have any number of phone numbers and addresses associated with a particular employee now.

Another useful tip is to use partial indexes on columns where we typically query a certain value more often than another. Take for example the employee table above. We're probably only displaying active employees throughout the majority of the application, but creating a partial index on that column where the value is true can help speed up the query and may help the planner to choose to use the index in cases where it otherwise would not. We can create a partial index like this:

```
CREATE INDEX employee_idx2 ON employee(active) WHERE active='t';
```

Or we may have a situation where a row has a column named 'employee_id' that is null until the row is associated with an employee, maybe in some trouble ticket like system. In that type of application we would probably have a 'View Unassigned Tickets' portion of the application which would benefit from a partial index such as this:

```
CREATE INDEX tickets_idx1 ON tickets(employee_id) WHERE employee_id IS NULL;
```

Application Development

There are many different ways to build applications which use a SQL database, but there are two very common themes that we will call stateless and stateful. In the area of performance there are different issues that impact each.

Stateless is typically the access type used by web based applications. Our software connects to the database, issues a couple of queries, returns to results to the user, and disconnects. The next action the users takes restarts this process with a new connect, new set of queries, etc.

Stateful applications are typically non-web based user interfaces where an application initiates a database connection and holds it open for the duration the application is in use.

Stateless Applications

In web based applications each time something is requested by the user, the application initiates a new database connection. While PostgreSQL has a very short connection creation time and in general it is not a very expensive operation, it is best to use some sort of database connection pooling method to get maximum performance.

There are several ways to accomplish database connection pooling, here is a short list of common ones:

Pgpool is a small server that you run on the same server as your clients that will pool database connections to some local or remote server. The application simply points at the pgpool instance instead of the normal postmaster. From the application's perspective nothing has changed as the connection pooling is hidden from it.

In a mod_perl environment you can use Apache::DBI to handle database connection pooling inside of Apache itself.

SQLRelay is another db connection manager that is somewhat database agnostic. It works with with several databases other than PostgreSQL.

You can always write a small bit of code to do this for you yourself, but I would highly recommend using an already developed solution to reduce the amount of debugging you have to do.

It should be noted that in a few bizarre instances it is possible that database connection pooling can reduce the performance of web based applications. At a certain point the cost of handling the pooling is more expensive than simply creating a new connection. It is suggested to test it both ways to see which is best for the application environment.

Stateful Applications

When building stateful applications we should look into using database cursors via the DECLARE command. A cursor allows us to plan and execute a query, but only pull back the data as you need it, for example one row at a time. This can greatly increase the snappiness of the UI.

General Application Issues

These issues typically effect both stateful and stateless applications in the same fashion. One good technique is to use server side prepared queries for any queries we execute often. This reduces the overall query time by caching the query plan for later use.

It should be noted however if you prepare a query in advance using placeholder values (such as 'column_name = ?') then the planner will not always be able to choose the best plan. For example, our query has a placeholder for the boolean column 'active' and you have a partial index on false values the planner won't use it because it cannot be sure the value passed in on execution will be true or false.

We can also obviously utilize stored procedures here to reduce the transmit, parse, and plan portions of the typical query life cycle. It is best to profile the application and find commonly used queries and data manipulations and put them into a stored procedure.