AgensGraph Developer Manual

SKAI worldwide Co., Ltd.

2018 년 12 월 12 일

Introduction

AgensGraph Highlights

AgensGraph, a database built on a graph data model that ensures ACID transactions, was implemented by utilizing main features of PostgreSQL, an open source database.

Key features include:

- Multi-model database
  - Supports Graph, Relational, and Document models
  - Intuitive and flexible data modeling using Graph and JSON documents
- High-level query features
  - Supports ANSI SQL and Cypher query
  - Supports ACID transactions
  - Possible to create a hybrid query statement that combines SQL and Cypher syntax when creating a query
  - Able to create hierarchical graph labels
- High performance query statement processing
  - Supports graph indexing for fast graph traversal
  - Possible to generate vertex and edge index
  - Supports a full-text search for JSON document processing
- Constraints
  - Supports Unique, Mandatory, Check constraints
- High Availability
  - Possible active-standby configuration
- Visualization tools
  - Visualizes the result data of graph queries
- Advanced security features
  - Implements an authentication system using Kerberos and LDAP
  - Encrypts via SSL/TLS protocols
- Connectivity
  - Provides JDBC and Hadoop drivers

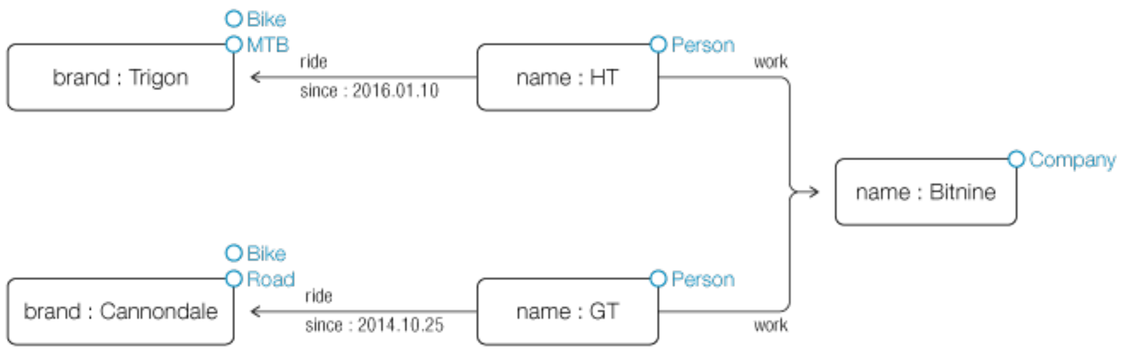
Graph Database Concepts

This section introduces the graph data model.

AgensGraph Database

The graph database stores and manages objects of a real model in graph form.
A relation (edge) exists between objects (vertices), and a group of similar vertices can be expressed as a group (label).
As vertices and edges have data (properties), they can be called Property Graph Models as well.

Let's take a closer look at the components of this graph data model.
The following example shows components of a graph:



Vertices

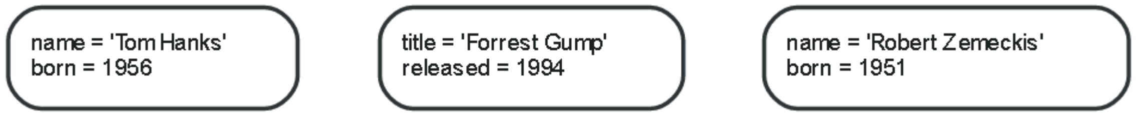
Vertices are the most basic elements in the graph data model. They represent entities in the real world and have properties.

A graph has vertices and edges as the base units. In AgensGraph, both vertices and edges may contain Properties. While entities are usually represented by vertices, they may be indicated using edges in some cases. Unlike edges and properties, vertices may have zero or multiple label values.

The simplest form of a graph consists of a single vertex. A vertex can have zero or more properties.

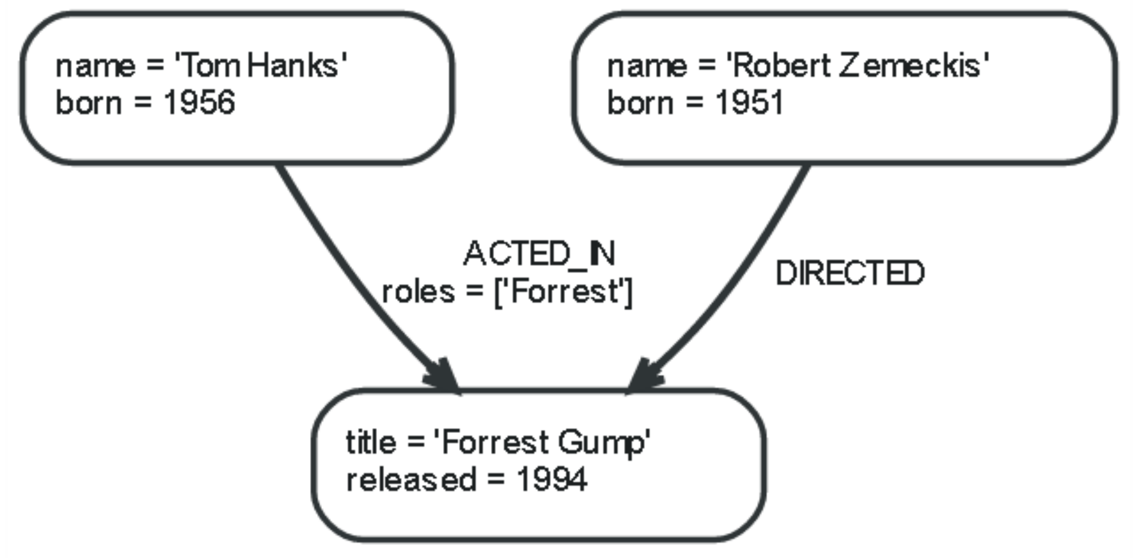
title = 'Forrest Gump'

The next step is to construct a graph with multiple vertices. Add two or more vertices to the graph of the previous step and add one or more properties to the existing vertex.



Edges

Edges connect vertices. When two vertices are connected via en edge and each vertex plays as start vertex or end vertex depending on the direction of the edge. Like vertices, edges have properties.
The edges between vertices play an important role in the graph database, especially when you search for linked data.
With edges, you may make vertices into a variety of data structures, such as lists, trees, maps, and composite entities. By adding edges to the example we are building, we can represent more meaningful data.

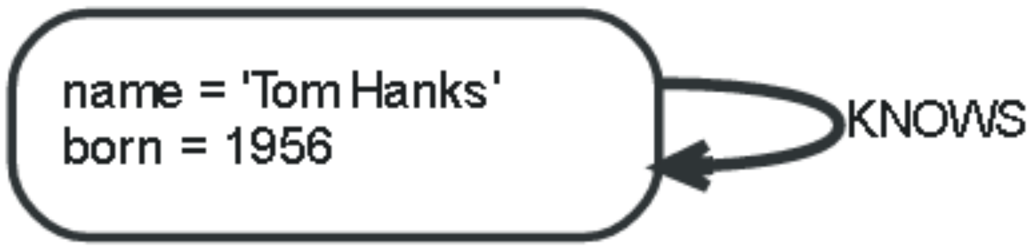


In the example, *ACTED_IN* and *DIRECTED*are used as edge types. The *ACTED_IN* property, *Roles*, stores the value of array type.

The *ACTED_IN* edge has the *Tom Hanks* vertex as start vertex and the *Forrest Gump* vertex as end vertex. In other words, we can say that the *Tom Hanks* vertex has an outgoing edge and the *Forrest Gump* vertex has an incoming edge.

If there is an edge in a single direction, you do not have to duplicate the edge and add it in the opposite direction; this is related to the graph traversal or performance.

Edges are always directional, but they may ignore directionality if it is not needed in your application. The diagram below shows a vertex having an edge pointing to itself.



All edges are recommended to have an edge type to perform the graph traversal in a more efficient manner.

Properties

Both vertices and edges may have properties. Properties are attribute values, and each attribute name should be defined only as a string type.

The available data types for property values are:

- Numeric type
- String type
- Boolean type
- List type (a collection of various data types)

NULL values cannot be used as property values. If NULL is entered, the property itself is assumed to be absent. *NULL* values, however, can be used in List.

Type	**Description**	**Value range**
boolean		true/false
byte	8-bit integer	-128 to 127, inclusive
short	16-bit integer	-32768 to 32767, inclusive
int	32-bit integer	-2147483648 to 2147483647, inclusive
long	64-bit integer	-9223372036854775808 to

		9223372036854775807, inclusive
float	variable-precision, inexact	15 decimal digits precision
char	16-bit unsigned integers representing Unicode characters	u0000 to uffff (0 to 65535)
String	sequence of Unicod characters	infinite

Labels

You may define the roles or types of vertices or edges using labels. Vertices or edges with similar characteristics can be grouped and the name of such a group can be defined, which is called a "label." That is, all vertices or edges with similar labels belong to the same group.

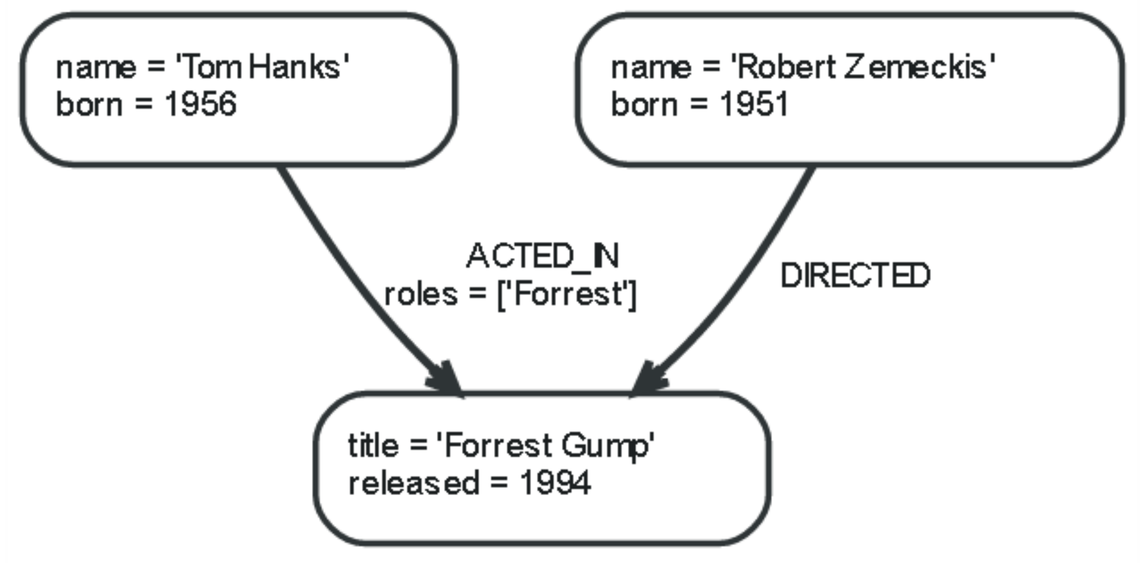
Database query statements can be performed only on the group (not the entire graph) using labels, which is helpful for a more efficient querying.

Using labels for vertices is optional, and each vertex may have zero or only one label.

Labels can also be used to define constraints on properties or to add indexes.

You may also assign a label similar to a vertex to an edge. Unlike vertices, there is no edge without a label; all edges should have at least one label.

Let us add *Person* and *Movie* labels to the existing example graph.



Label names

Label names can be expressed using letters and numbers, all converted to lowercase letters.

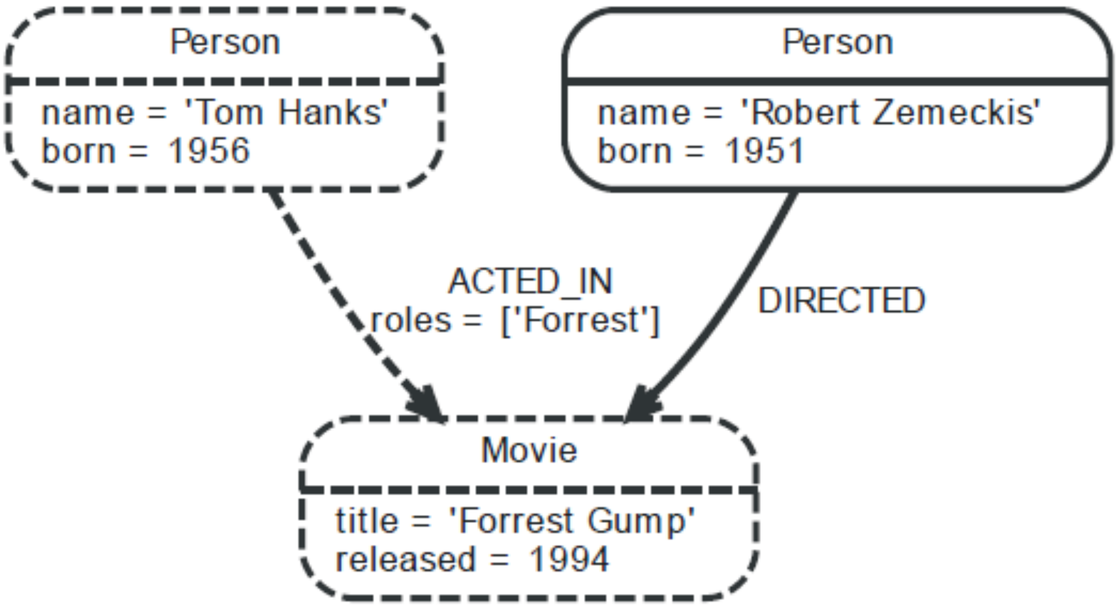
Labels stores a unique id of int type, which means that the database may contain up to 2^16-1(65535) labels.

Traversal

Traversal is to traverse paths while exploring a graph to answer the requested query. Traversal is a process of searching for the relevant vertices from start vertex to find the answer to the requested query. In other words, it refers to following the vertices that are traversing the graph and the derived edges according to a specific rule.

In the examples illustrated so far, we try to find a movie featured by *Tom Hanks*. Starting with the *Tom Hanks* vertex, you can traverse all the processes that end at the vertex of

Forrest Gump along the *ACTED_IN* edge associated with it.

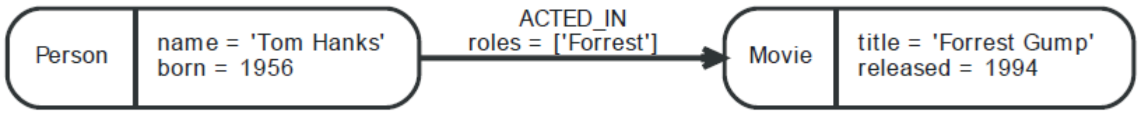


By using the traversal of cypher query statements and additional techniques in the graph database, you may derive better result data. For more information, see Cypher Query Language.

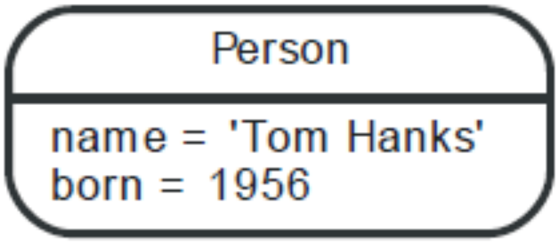
Paths

Paths are the result data of a query statement or traversal, which shows one or more vertices and the edges connected to them.

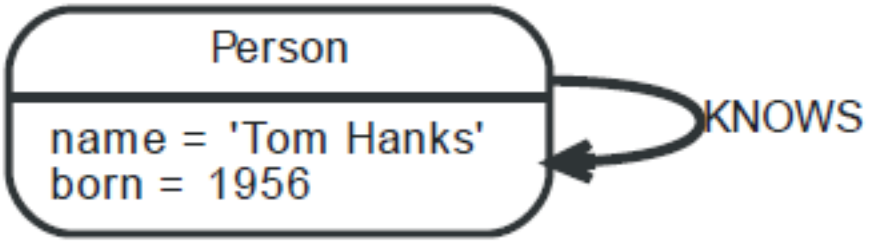
The path (traversal result data) from the previous example is as follows:



The length of the above path is 1. The shortest path length is 0, which is the case when a single vertex does not have edges.



If the vertex has an edge pointing to itself, the length of path is 1.



Get Started

Install AgensGraph

Pre-Installation on Linux

Get the pre-compiled binary

AgensGraph works on Linux/Windows and can be installed in two ways. One is to download its binary package, and the other is to download its source code and compile the package. The binary package can be downloaded from the SKAI worldwides's website, and the source code can be downloaded from github. If you want to know your system environment, enter the following command in the command window.

```
uname -sm
```

Extract the package

Unzip the downloaded file in the desired location. (e.g.: /usr/local/AgensGraph/)

```
tar xvf /path/to/your/use
```

Post-Installation Setup and Configuration

Setting environment variables (Optional)

Add the following three lines to your shell startup file (e.g. .bash_profile).

```
export LD_LIBRARY_PATH=/usr/local/AgensGraph/lib:$LD_LIBRARY_PATH
export PATH=/usr/local/AgensGraph/bin:$PATH
export AGDATA=/path/to/make/db_cluster
```

Creating a database cluster

Create a database cluster using the following command: If you do not specify the `-D` option, use the `AGDATA` specified in `.bash_profile`.

```
initdb [-D /path/to/make/db_cluster]
```

Starting the server

You can start AgensGraph with the following command.

```
ag_ctl start [-D /path/created/by/initdb]
```

Creating a database

```
createdb [dbname]
```

If you do not specify the database name and user name, the default value will apply. The default is the name of the current user.

Running a terminal

```
agens [dbname]
```

When you run a terminal like above, you can see the following screen:

```
username=#
```

If it is a super user, "= #" will be displayed at the prompt; "=>" will be displayed for other users.

```
username=# CREATE
username-# (
username(# (
username(# )
username(# )
username-#
```

In case you are in the middle of entering a query, `-` is displayed. If you are typing content to be included in parentheses, `(` is displayed. Even multi-parentheses can be expressed as needed. You may customize the input prompt, and the detailed options can be found in the following link.

Setting server parameters

AgensGraph enables you to configure the server to improve performance. Setting the server parameters in consideration of the size of data and resources of the server (memory, CPU, disk size, and speed) is critical for improvement of performance. The following server variables have a significant effect on AgensGraph's graph query performance. You may change server parameters by changing `$AGDATA/postgresql.conf`. If you modify $AGDATA/postgresql.conf, you need to restart the server.

- `shared_buffers`: Memory size for data object caching. This variable must conform to the product environment. It is optimal when this variable is as large as the size of data. shared_buffers should be set carefully considering the amount of memory allocated for concurrent sessions and each query. The recommended value is half the physical memory.
- `work_mem`: Increases in size depending on the physical memory and the attributes of the query being executed.
- random_page_cost: A parameter for query optimization. This parameter should be lowered to 1 or 0.005 for graph queries (when the graph data is completely cached in memory)

Get started with Cypher

This section introduces Cypher and the followings:

- Basic understanding on graphs and patterns
- Simple troubleshooting
- Cypher syntax writing

About Cypher

AgensGraph supports Cypher, a query language, for retrieving and processing graph data. Cypher is a declarative language similar to SQL.

Pattern

AgensGraph's graph consists of vertices and edges. Vertices and edges can have many properties. The actual data consists of simple graphs in patterns. AgensGraph searches and processes the patterns of graphs through cypher.

Creating Graphs

AgensGraph may store multiple graphs in a single database. Cypher cannot figure out multiple graphs. AgensGraph supports variables for generating and managing graphs using DDL and Cypher. The following syntax generates a graph called network and sets the current graph.

```
CREATE GRAPH network;
SET graph_path = network;
```

In this example, the `graph_path` variable is set to network. However, if `graph_path` is not set before creating a graph, it will be set automatically after a graph is generated.

Creating Users

```
CREATE ROLE user1 LOGIN IN ROLE graph_owner;
DROP ROLE user1;
```

You can create new users to manage ownership and other privileges on database objects. If a new user wants to generate a label in the graph, the new user must be in the same group as the user who created the graph. See this link for more options regarding creating users.

Creating Labels

You should generate a label before generating graph data in principle. However, for your convenience, a label will be automatically created if it is specified when Cypher's CREATE is executed. In AgensGraph, you should have one label for a vertex and an edge. The following example creates a vertex label named `person` and an edge label `knows`.

```
CREATE VLABEL person;
CREATE ELABEL knows;
CREATE (n:movie {title:'Matrix'});
```

Creating Vertices and Edges

This section uses `CREATE` of Cypher to create the *person* vertex and *knows* edge. The `CREATE` clause creates a pattern consisting of vertices and edges. `(variable:label {property: value, ...})` is a vertex type, and `-[variable:label {property: value, ...}]-` is an edge type. The direction of the edge can be represented by < or >. Variables may or may not exist in the forms of vertices and edges.

Note : AgensGraph does not support `--` in edge patterns. `--` means a comment at the end of a sentence.

In the following example, patterns such as "Tom knows Summer," "Pat knows Nikki" and "Olive knows Todd" are created.

```
CREATE (:person {name: 'Tom'})-[:knows]->(:person {name: 'Summer'});
CREATE (:person {name: 'Pat'})-[:knows]->(:person {name: 'Nikki'});
CREATE (:person {name: 'Olive'})-[:knows]->(:person {name: 'Todd'});
```

AgensGraph uses the *jsonb* type for vertex/edge attributes. Attributes are expressed using JSON objects.

Cypher Query Language

Introduction

Cypher is

A graph query language for querying graph data. The main features are as follows:

- **Declarative**
  Cypher is a declarative language that describes **what** it is, rather than **how** it should be done. In contrast to imperative languages that specify algorithms to be executed, such as C and Java, Cypher specifies goals. This type of processing relieves the user of the detailed implementation of the query. This type of processing relieves the burden of detailed implementation in user queries.

- **Pattern Matching**
  Cypher is a language that illustrates the graph data that you are looking for. The graph pattern to be searched is expressed by using parentheses and dashes as ASCII Art, and the graph data matching the pattern is found. You can create queries in an intuitive manner because it allows you to draw the form you want to search.

- **Expressive**
  Cypher borrowed various processing methods for expressive queries. Most keywords such as WHERE and ORDER BY are borrowed from SQL, pattern matching from SPARQL, and the collection concept from languages such as Haskell and Python. This makes it possible for you to express queries in an easy and simple manner.

Elements of Cypher

The basic elements of Cypher include vertex, edge, vlabel, elabel, property, and variable.

- **Vertex**
  The most basic element that constructs a graph, representing an entity. Even if vertices are mostly used to represent entities, their uses can vary depending on the purposes.

- **Edge**
  Represents the relationship between vertices and cannot exist as an edge alone.

- **Vlabel**
  A specific name given by the user to be a criterion for classifying vertices.
- **Elabel**
  The name of the edge; it represents the relationship between vertices.
- **Property**
  An attribute that can be assigned individually to a vertex or edge.
- **Variable**
  An identifier that is assigned arbitrarily to a vertex or an edge.

Handling Graph

Before describing Cypher in detail, let's look at how to use it. We will show you how to create a graph, query the graph, and modify the graph.

Creating Graph

AgensGraph is capable of storing multiple graphs within a single database. Thus, you should first create/select graphs to use to enable graph query using Cypher.

- Create Graph

```
CREATE GRAPH graphName;
SET graph_path = graphName;
```

  `CREATE GRAPH` is a command to create a graph. The command is used along with the name of the graph to be generated (Spaces are allowed in graphName from v1.3). `graph_path` is a variable to handle the current graph. Set the name of the graph you want to handle using `SET`.

- Drop Graph

```
DROP GRAPH graphname CASCADE;
```

  `DROP GRAPH` is a command to delete a graph from graph database. To be noticed, a graph automatically contains vertex and edge labels when it is created, so to delete a graph you first must delete labels which are bound to that graph. To delete a graph and its' labels altogether, use `DROP GRAPH` graphname CASCADE.

- Create Labels

```
CREATE VLABEL vlabelName;
CREATE ELABEL elabelName;

CREATE VLABEL childVlabelName inherits (parentVlabelName);
CREATE ELABEL childElabelName inherits (parentElabelName1, parentElabelNa
me2);
```

`CREATE VLABEL` creates vlabel, and `CREATE ELABEL` creates elabel. Each command is used along with the name of vlabel or elabel to generate.
`inherits()` is a command that inherits another label. When you create a label, you can inherit other labels by specifying the names of the parent labels along with the corresponding keyword after the child label name. There can be multiple parent labels for inheritance.

- Drop Labels

```
DROP VLABEL vlabelName;
DROP ELABEL elabelName;

DROP ELABEL elabelName CASCADE;
```

`DROP VLABEL` deletes vlabel and `DROP ELABEL` deletes elabel. Each command should be used with the names of vlabel and elabel you wish to erase. A vlabel that is inherited by other vlabels cannot be deleted directly, so you must use CASCADE command to delete all dependant data.

- Create vertices and edges

```
CREATE (:person {name: 'Jack'});
CREATE (:person {name: 'Emily'})-[:knows]->(:person {name: 'Tom'});
```

The `CREATE` clause is used to create vertices and edges. When using this command, make sure to write vertices or edges to generate correctly using pattern. (The pattern that represent various clauses and vertices/edges along with the CREATE clause will be described in detail later on).

Querying Graph

Querying graph is the process of getting information you need from the graph(s) by describing some specific graph pattern and finding it in that graph.

```
MATCH (:person {name: 'Jack'})-[:likes]->(v:person)
RETURN v.name;

name
-------
Emily
(1 row)
```

The MATCH clause finds graph data that matches the pattern in the clause. In the RETURN clause, specify only the elements that you want to return from the graph you find.

Manipulating Graph

In order to modify the existing graph data, the pattern of the MATCH clause should be displayed; after finding the matched graph, modifications can be made to.

```
MATCH (v:person {name: 'Jack'})
SET v.age = '24';
```

You can use the SET clause to set property values of vertices and edges.

Data Type

This section describes the graph data types such as graphid, graphpath, vertex, and edge. Each data type is specified below.

Name	Size	Description
graphid	8	unique ID of vertex/edge
graphpath	tuple	Array of consecutive vertices and edges
vertex	tuple	A tuple that combines id and properties
edge	tuple	A tuple that combines start vertex id, end vertex id, edge id and properties

graphid

An unique ID that is assigned to the newly-created label of vertex/edge. Graphid is generated based on the labid of ag_label and the sequence values of vertex and edge.

graphpath

An array type of continuous vertices and edges.

Column	Type	Description
vertices	vertex[]	
edges	edge[]	

vertex

vA vertex has a vertex id and properties

Column	Type	Description
id	graphid	vertex id
properties	jsonb	vertex property

edge

An edge connects vertices, and has edge id, start vertex id, end vertex id, and properties.

Column	Type	Description
id	graphid	edge id
start	graphid	start vertex id
end	graphid	end vertex id
properties	jsonb	edge property

Syntax

Pattern

A pattern is an expression that represents a graph. As a pattern is represented by a combination of one or more vertices or edges, how you create vertices and edges as a pattern is critical.

Vertex

Vertex is expressed using parentheses, and can be specified with vlabel, property, and variable to further refine the vertex to be searched.

```
( )
```

Vertex is represented by **`( )`**. A pattern that does not have vlabel or property in parentheses, as in the example above, means all vertices.

```
(:person)
```

To represent vlabel in a vertex, use **`(:vlabelName)`**. The name of vlabel is indicated with a colon in parentheses indicating the vertex. The above example represents a vertex with vlabel named "person."

```
(v)
(var)
(var_1)
(v:person)
```

If you want to assign a variable to the vertex, use **`(variableName)`**. A variable can be named a combination of alphanumeric (a~z, 0~9) and underbars (note that it should not start with a number). When you want to display the variable and vlabel at the same time in vertex, use **`(variableName: vlabelName)`**.

```
({name: 'Jack'})
(v:person {name: 'Jack'})
(v:person {name: 'Jack', age: 24})
```

You can further refine the properties by listing them in the vertex. If you want to represent a property, use
`({propertyName:propertyValue})`. If the value is a string, the value must be wrapped with **' '**. If you want to represent multiple properties, **`,`** should be used.

Edge

An edge is expressed with two dashes, and can be marked along with elabel, property, and variable.

```
-[]-
-[]->
<-[]-
```

An edge is expressed using **`-[]-`**. You can express direction with **`< >`**. In the above example, **-[ ]-** expressed as dashes only means "all edges" since no constraint is indicated. **-[ ]-> <-[ ]-** with additional angle brackets means "all edges with one directionality."

```
-[:knows]->
```

If you want to express other elements in the edge, specify them in **`[ ]`**. If you want to represent elabel on the edge, use **`-[:elabelName]->`**. The above example means an edge with elabel named "knows."

```
-[e]->
-[e:likes]->
```

If you want to assign a variable to an edge, use **`-[variableName]->`**. When you want to display variable and elabel simultaneously on the edge, use **`-[variableName:elabelName]->`**.

```
-[{why: 'She is lovely'}]->
-[:likes {why: 'She is lovely'}]->
-[e:likes {why: 'She is lovely'}]->
```

If you want to represent a property on the edge use **`-[{propertyName:propertyValue}]->`**. If the value is a string, use **`''`**. If you want to represent multiple properties, **`,`** should be used.

Vertices and Edges

Patterns can be used to express vertices, edges, or a combination of them.

```
()-[]->()
(jack:person {name: 'Jack'})-[k:knows]->(emily:person {name: 'Emily'})
p = (:person)-[:knows]->(:person)
```

The basic frame of a pattern consisting of a combination of vertices and edges is **`( )-[]->( )`**. It is good to give meaning to the pattern by specifying properties and labels on vertices and edges. Variables can be assigned to individual vertices and edges, or the entire pattern. If you want to assign a variable to the entire pattern, use **`=`**. In the above example, **p** is a variable and is applied to the entire pattern by using **`=`**.

Path

The number of vertices and edges in a pattern may continue to increase. The unit for a series of path in a pattern is called a path.

```
(a)-[]->( )-[]->(c)
(a)-[*2]->(c)

(a)-[]->( )-[]->( )-[]->(d)
(a)-[*3]->(d)

(a)-[]->( )-[]->( )-[]->( )-[]->(e)
(a)-[*4]->(e)
```

In the case of a long path, it can be written in abbreviated form for better readability and ease of writing. If n vertices go through an edge n-1 times, you can put an asterisk (*) and n-1 in brackets (see example above).

Flexible Length

Depending on the situation, you may need to dynamically change the number of edges that must be gone through in a query.

```
(a)-[*2..]->(b)
(a)-[*..7]->(b)
(a)-[*3..5]->(b)
(a)-[*]->(b)
```

If you want to give a dynamic change to the length of the path, `..` can be used. The example above shows a path with two or more edges, a path with seven or fewer paths, a path with three or more and five or fewer paths, and an infinite path (in sequence from top to bottom).

Clauses

Read Clauses

MATCH

The MATCH clause is a clause that describes the graph pattern you want to find in the database. This is the most basic way to import data. For more information on the Pattern specification, see the Pattern section above.

- Mathing

```
MATCH (j {name: 'Jack'})
RETURN j;

MATCH (j {name: 'Jack'})-[l:knows]->(e)
RETURN l;

MATCH (j {name: 'Jack'})
RETURN j.age;

MATCH p=(j {name: 'Jack'})-[l:knows]->(e)
RETURN p;
```

  The graph pattern to be searched is described in the MATCH clause and then returned using the RETURN clause.

- Various Notation

```
MATCH (j:person {name: 'Jack'})-[:knows]->(v:person)
MATCH (e:person {name: 'Emily'})-[:knows]->(v)
```

```
RETURN v.name;

MATCH (j:person {name: 'Jack'})-[:knows]->(v:person),
      (e:person {name: 'Emily'})-[:knows]->(v)
RETURN v.name;

MATCH (j:person {name: 'Jack'})-[:knows]->(v:person)<-[:knows]-(e:person
{name: 'Emily'})
RETURN v.name;
```

The above three queries output the names of common acquaintances of Jack and Emily. They all output the same result, but the number of MATCH clauses and the pattern notations in the MATCH clause are different. The first query used the MATCH clause twice, while the second query expressed two patterns using a comma in one MATCH clause; the third query used one pattern in one MATCH clause. Likewise, you can use the MATCH clause in a variety of ways to derive the same result.

When you generate, query, delete, add, or modify a graph data, you will need to find a graph that matches a certain pattern first in most cases. Thus, the MATCH clause is a very basic and important clause.

[Reference]

When a large volume of data across multiple pages are printed in Agens, you may see help on scrolling by using the **?** keyword.

```
Most commands optionally preceded by integer argument k.  Defaults in bra
ckets.
Star (*) indicates argument becomes new default.
-------------------------------------------------------------------------
------
<space>                 Display next k lines of text [current screen size]
z                       Display next k lines of text [current screen size]
*
<return>                Display next k lines of text [1]*
d or ctrl-D             Scroll k lines [current scroll size, initially 11]
*
q or Q or <interrupt>   Exit from more
s                       Skip forward k lines of text [1]
f                       Skip forward k screenfuls of text [1]
b or ctrl-B             Skip backwards k screenfuls of text [1]
'                       Go to place where previous search started
=                       Display current line number
/<regular expression>   Search for 'k'th occurrence of regular expression
 [1]
```

```
    n                         Search for 'k'th occurrence of last r.e [1]
    !<cmd> or :!<cmd>         Execute <cmd> in a subshell
    v                         Start up /usr/bin/vi at current line
    ctrl-L                    Redraw screen
    :n                        Go to kth next file [1]
    :p                        Go to kth previous file [1]
    :f                        Display current file name and line number
    .                         Repeat previous command
    -------------------------------------------------------------------------
    --
```

Shortest Path

- Single Shortest Path
  Finding a single shortest path between two vertices using `shortestpath` function.

```
MATCH p = shortestpath( (j:person {name: 'Jack'})-[l:knows*..15]-(a:perso
n {name: 'Alice'}) )
RETURN p;
```

  Find single shortest path between two vertices, `Alice` and `Jack`, between which 15 edges exist. Feed a graph path consists of the start vertex, the end vertex, and the edges between them (single link path) into the function as its argument. Edge type, max hops and directions are all used in finding the shortest path.

- All Shortest Paths
  Find all the shortest paths between two vertices.

```
MATCH p = allshortestpaths( (j:person {name: 'Jack'})-[l:knows*]-(a:perso
n {name: 'Alice'}) )
RETURN p;
```

  Find all the shortest paths between two vertices. `Alice` and `Jack`.

OPTIONAL MATCH

The OPTIONAL MATCH clause, like the MATCH clause, is a clause describing the graph pattern to be searched. The OPTIONAL MATCH clause differs from the MATCH clause in that it returns NULL if there are no results to return.

- Optional Matching

```
OPTIONAL MATCH (e:person {name:'Emily'})
RETURN e.hobby;
```

The use of the OPTIONAL MATCH clause is the same as the MATCH clause. Put the pattern you want to find in the OPTIONAL MATCH clause and return the result through the RETURN clause. The returned result may contain NULL.

MATCH ONLY

The Only keyword allows you to use the MATCH query to return results that exclude child labels.

- Match only

```
MATCH (n:person ONLY) RETURN n;
MATCH ()-[r:knows ONLY]->() RETURN r;
```

Projecting Clauses

RETURN

The RETURN clause is a clause that specifies the result of a cypher query. It returns data that matches the graph pattern you are looking for, and can return vertices, edges, properties, and so on.

- Vertex Return

```
MATCH (j {name: 'Jack'})
RETURN j;
```

If you want to return a vertex, specify the vertex you want to find in the MATCH clause and assign a variable to it. You can then return a vertex by specifying the corresponding variable in the RETURN clause.

- Edge Return

```
MATCH (j {name: 'Jack'})-[k:knows]->(e)
RETURN k;
```

If you want to return an edge, specify the edge you want to find in the MATCH clause and assign a variable to it. You can then return an edge by specifying the corresponding variable in the RETURN clause.

- Property Return

```
MATCH (j {name: 'Jack'})
RETURN j.age;
```

If you want to return a property of a vertex or edge, specify the vertex or edge you want to find in the MATCH clause and assign a variable to it. In the RETURN clause, you can return the property by specifying **.** with the variable and property.

- All Return

```
MATCH (j {name: 'Jack'})-[k:knows]->(e)
RETURN *;
```

If you want to return all the elements in the MATCH clause, specify ***** in the RETURN clause. The elements returned are the elements to which variables are assigned. Elements with no variable assigned will not be returned.

- Path Return

```
MATCH p=(j {name: 'Jack'})-[k:knows]->(e)
RETURN p;
```

If you want to return a path that matches the pattern shown in the MATCH clause, you should apply a variable to the entire pattern. If the pattern is preceded by a variable and **=**, the variable can be applied to the entire pattern. You can then return the path by specifying the variable in the RETURN clause.

- Alias Return

```
MATCH (j {name: 'Jack'})
RETURN j.age AS HisAge;

MATCH (j {name: 'Jack'})
RETURN j.age AS "HisAge";
```

You can output an alias to the column of the returned result. The returned element is followed by an alias with the **AS** keyword. If you list the alias without double quotation marks, the output will be in lower case; with double quotation marks, the output will be printed as it is.

- Function Return

```
MATCH (j {name: 'Jack'})-[k:knows]->(e)
RETURN id(k), properties(k);

MATCH p=(j {name: 'Jack'})-[k:knows]->(e)
RETURN length(p), nodes(p), edges(p);

MATCH (a)
RETURN count(a);
```

It provides a function to obtain only id and properties for vertices and edges, and a function to get length and each element separately of the path. You may also use the aggregation function supported by PostgreSQL.

- Constant Return

```
RETURN 3 + 4, 'Hello ' + 'AgensGraph';
RETURN 3 + 4 AS lucky, 'Hello ' + 'AgensGraph' AS greeting
```

Constant values and their operators can be expressed. All of the expressions useable in the SQL SELECT statement can be used with the RETURN clause, except for table and column expressions. For more information, see PostgreSQL Expression.

- Unique Return

```
MATCH (j { name: 'Jack' })-[k:knows]->(e)
RETURN DISTINCT e;
```

Removes the duplicate records of the returned result and outputs only the unique result. You can add DISTINCT before the returned element.

WITH

The WITH clause is a clause that connects multiple cypher queries. It passes the result of the preceding WITH clause query to the following WITH clause query. That is, the WITH clause can be regarded as a RETURN clause that passes the value of the preceding query to the input of the following query.

- WITH

```
MATCH (j:person {name:'Jack'})-[:knows]->(common:person)<-[:knows]-(other:
person)
RETURN other, count(common);

WHERE count(common) > 1
RETURN other;

MATCH (j:person {name:'Jack'})-[:knows]->(common:person)<-[:knows]-(other:
person)
WITH other, count(common) AS cnt
WHERE to_jsonb(cnt) > 1
RETURN other;
```

The WITH clause is described as an example of the above three queries. The first query returns the number of acquaintances that Jack and 'someone' know in common with the 'someone'. If we think the second query in line with the first query, we can figure out that it is a query that returns 'someone' when the number of common acquaintances is greater than one (1). The second query cannot be executed alone; it is only meaningful when combined with the first query. The WITH clause plays a role of connecting the two queries.

The third query is a query that includes a WITH clause that links the first and second queries. It holds the returned results of the first query in variable and alias forms, and then passes the results to the second query. In the case where you want to connect multiple queries, as in this case, and the output of the preceding query is used as the input of the trailing query, you can concatenate them with the WITH clause. The WITH clause must be held in variable or alias form.

- Partitioning

```
MATCH (j:person {name:'Jack'})-[:knows]->(common:person)<-[:knows]-(other:
person)
RETURN other, count(common)
ORDER BY other.name
LIMIT 10;

WHERE count(common) > 1
RETURN other;

MATCH (j:person {name:'Jack'})-[:knows]->(common:person)<-[:knows]-(other:
person)
WITH other, count(common) AS cnt
ORDER BY other.name
LIMIT 10
WHERE to_jsonb(cnt) > 1
RETURN other;
```

It is important to partition the entire query when using the WITH clause. Think of the WITH clause as a RETURN clause, and consider ORDER BY, LIMIT as a partition after RETURN. Therefore, if there is an ORDER BY, LIMIT clause in the preceding query, the clause must be written after the WITH clause, and then the following query is created.

Reading sub-Clauses

WHERE

The WHERE clause is a clause that adds constraints to the MATCH or OPTIONAL MATCH clause or filters the results of the WITH clause. The WHERE clause cannot be used alone; it is dependent on the MATCH, OPTIONAL MATCH, START, and WITH clauses when used.

Basic Usage

- Boolean operations

```
MATCH (v)
WHERE v.name = 'Jack' AND v.age < '30' OR NOT (v.name= 'Emily' OR v.name
='Tom')
RETURN v.name, v.age;
```

  You can use the boolean operators `AND`, `OR`.

- Filter on vertex label

```
MATCH (v)
WHERE label(v) = 'person'
RETURN v.name;
```

  To filter nodes by label, write a label predicate after the WHERE keyword using `WHERE label(n) = 'foo'`.

- Filter on vertex property

```
MATCH (v)
WHERE v.age < '30'
RETURN v.name, v.age;
```

  To filter on a vertex property.

- Filter on edge property

```
MATCH (v)-[l:likes]->(p)
WHERE l.why = 'She is lovely'
RETURN p;
```

  To filter on aedge property.

String matching

STARTS WITH and ENDS WITH can be used to search for a character having certain prefix and suffix in a string. You can use CONTAINS when retrieving a matching string regardless of its position. It is case-sensitive, and returns null if a non-string value is queried.

- Prefix string search using `STARTS WITH`

```
MATCH (v)
WHERE v.name STARTS WITH 'Em'
RETURN v.name, v.age;
```

STARTS WITH is used to perform a case-sensitive matching at the beginning of a string.

- Suffix string search using `ENDS WITH`

```
MATCH (v)
WHERE v.name ENDS WITH 'ly'
RETURN v.name, v.age;
```

ENDS WITH is used to perform a case-sensitive matching at the end of a string.

- Substring search using `CONTAINS`

```
MATCH (n)
WHERE n.name CONTAINS 'il'
RETURN n.name, n.age;
```

CONTAINS is used for a case-sensitive matching regardless of the position in the string.

ORDER BY

The ORDER BY clause is a clause that sorts result values. It is used with the RETURN clause or WITH clause and determines whether to sort in ascending or descending order by column.

- Order by Property

```
MATCH (a:person)
RETURN a.name AS NAME
ORDER BY NAME;

MATCH (a:person)
RETURN a.name AS NAME, a.age AS AGE
ORDER BY NAME, AGE;

MATCH (a:person)
RETURN a
ORDER BY a.name;
```

The ORDER BY clause basically sorts by properties of vertices and edges. You may specify the aliases of the properties that will be the basis of sorting in the ORDER BY clause. That is, in the RETURN clause, an alias is given to a property to be sorted, and then the alias is specified in the ORDER BY clause. If multiple sorting criteria are specified in the ORDER BY clause, sorting is done first based on the first criterion and then the second sorting is performed only for duplicated values after the first sorting. However, if a property is not specified in the RETURN clause, you can directly specify the property other than the corresponding alias in the ORDER BY clause.

- Descending Order

```
MATCH (a:person)
RETURN a.name AS NAME
ORDER BY NAME DESC;

MATCH (a:person)
RETURN a.name AS NAME, a.age AS AGE
ORDER BY NAME DESC, AGE;
```

The ORDER BY clause basically sorts in ascending order. If you want to sort in descending order, you can write the **DESC** keyword. If multiple criteria are specified in the ORDER BY clause, **DESC** must be followed by the elements to be sorted in descending order.

SKIP

The SKIP clause is a clause that changes the search start location.

- Skip

```
MATCH (a)
RETURN a.name
SKIP 3;
```

When searching for a pattern in the MATCH clause, skip the number of patterns specified in the SKIP clause and start the search.

LIMIT

The LIMIT clause is a clause that limits the number of results in a result set.

- Limit Result Set

```
MATCH (a)
RETURN a.name
LIMIT 10;
```

If you want to limit the number of results you want to output, you can specify the number in the LIMIT clause. If the number of results in the result set is smaller than or equal to the number specified in the LIMIT clause, all results will be output; if bigger, only the number of results specified in the LIMIT clause will be output.

Writing Clauses

CREATE

The CREATE clause is a clause that creates graph elements such as vertices and edges.

- Create Vertex

```
CREATE ( );
CREATE (:person);
CREATE (:person {name: 'Edward'});
CREATE (:person {name: 'Alice', age: 20});
CREATE (a {name:'Alice'}), (b {name:a.name});
```

When you want to create a vertex, you need to write a vertex-related pattern in the CREATE clause. When creating a vertex, you can create it by referring to the vertex specified earlier as in the last example.

- Create Edge

```
MATCH (E:person {name: 'Edward'}), (A:person {name: 'Alice'})
CREATE (E)-[:likes]->(A);

MATCH (E:person {name: 'Edward'}), (A:person {name: 'Alice'})
CREATE (E)-[:likes {why: 'She is lovely'}]->(A);

MATCH (E:person {name: 'Edward'})
CREATE (E)-[:IS_PROUD_OF]->(E);
```

An edge is a link between two vertices. Therefore, we need to create an edge after finding two vertices through the MATCH clause. Note that two vertices do not necessarily mean different vertices. Like the third query above, self-edge is also possible.

- Create Path

```
CREATE (E:person {name: 'Edward'})-[:likes]->(A:person {name: 'Alice'});

MATCH (E:person {name: 'Edward'})
CREATE (E)-[:likes]->(A:person {name: 'Alice'});

MATCH (E:person {name: 'Edward'}), (A:person {name: 'Alice'})
CREATE (E)-[:likes]->(A);
```

You may create each element separately, but you may also create a pattern you want at a time by listing the path in the CREATE clause. Be careful when creating a path at once. If it is not used with the MATCH clause like the first query, it creates a new data even if there is the same data as the corresponding pattern (i.e. duplicate data). If you use it with a MATCH clause like the second query, it finds the Edward vertex and then creates the likes edge on the vertex and Alice vertex. If you want to create only the likes edges in existing Edward and Alice vertices, you can use it like the third query.

DELETE

The DELETE clause is a clause that removes vertices or edges.

- Delete vertex

```
MATCH (m:person {name: 'Michael'})
DELETE m;
```

Find the vertices you want to remove through the MATCH clause and remove them by marking corresponding variables in the DELETE clause. However, if the vertices you want to remove are connected to other vertices and edges, remove the edges first before removing the vertices.

- Delete edge

```
MATCH (m:person {name: 'Michael'})-[l:likes]->(b:person {name: 'Bella'})
DELETE l;
```

Find the edges you want to remove with the MATCH clause and remove them by marking corresponding variables in the DELETE clause.

DETACH DELETE

- Delete a vertex with all its relationships

```
MATCH (m:person {name: 'Michael'})
DETACH DELETE m;
```

If the DETACH keyword is used together, the edges associated with the vertices are also removed. If the vertices you want to remove are linked to other vertices and edges, you may skip the edge-removing process.

SET

The SET clause is a clause that adds, sets, or removes properties, or adds vlabels.

- Add Property

```
MATCH (E:person {name: 'Edward'})
SET E.habit = 'play the guitar';
```

Find the vertices or edges to which you want to add properties through the MATCH clause, and specify the names and values of the properties to add to the SET clause. When describing a property, mark it with single or double quotation marks.

- Modify Property

```
MATCH (E:person {name: 'Edward'})
SET E.name = 'Edward Williams';
```

Find the vertices or edges whose properties you want to change through the MATCH clause, and specify the names of the properties to be changed in the SET clause. When describing a property, mark it with single or double quotation marks.

- Remove Property

```
MATCH (E:person {name: 'Edward'})
SET E.hobby = NULL;
```

Find the vertices or edges from which you want to remove the properties through the MATCH clause, and mark the names of the properties to be removed and NULL in the SET clause.

- Copy properties between nodes and relationships

```
MATCH (at {name:'Edward'}), (pn {name: 'Peter'})
SET at = pn
RETURN at.name, at.age, pn.name, pn.age;
```

SET can be used to copy all properties from one vertex/edge to another vertex/edge. All properties of "pn" are copied to the properties of "at", and all existing properties of "at" are removed.

- Replace all properties using a map and =

```
MATCH (p { name: 'Edward' })
SET p = { name: 'Edward William', position: 'Enterpreneur' }
RETURN p.name, p.age, p.position;
```

Use the = operator to replace the existing properties of the vertex/edge with the properties provided by the map.

- Remove all properties using an empty map and =

```
MATCH (p { name: 'Edward' })
SET p = { }
RETURN p.name, p.age, p.position;
```

Use the = operator to remove all existing properties of the vertex/edge.

- Mutate specific properties using a map and +=

```
MATCH (p { name: 'Edward' })
SET p += { age: 38, hungry: TRUE, position: 'Enterpreneur' }
RETURN p.name, p.age, p.hungry, p.position;;
```

Use the += operator to change existing properties of the vertex/edge or add new ones.
The properties of the vertex and edge is not removed if the map is empty as shown below.

```
MATCH (p { name: 'Edward' })
SET p += { }
RETURN p.name, p.age, p.hungry, p.position;
```

- Set multiple properties using one SET clause

```
MATCH (n { name: 'Peter' })
SET n.position = 'Developer', n.surname = 'Taylor';
```

Sets multiple properties at once by separating them with commas (,).

REMOVE

The REMOVE clause is a clause that removes properties.

- Remove property

```
MATCH (E:person {name: 'Edward'})
SET E.habit = NULL;

MATCH (E:person {name: 'Edward'})
REMOVE E.habit;
```

How to remove a property using the SET clause has already been described in SET.
You can also remove a property using the REMOVE clause.
Locate the element from which you want to remove the property through the MATCH clause, and name the property to be removed in the REMOVE clause.

Reading/Writing Clauses

MERGE

The MERGE clause is a clause that i) adds a corresponding pattern (like the CREATE clause) if the specified pattern does not exist in the graph, and ii) verifies existence of the pattern (like the Match clause) if it already exists in the graph. The MERGE clause recognizes the entire pattern specified in the clause.

- Merge

```
MERGE (:person {name: 'Edward'});

MATCH (p:person {name: 'Edward'}) RETURN p;

CREATE (:person {name: 'Edward'});
```

  If you specify a pattern in the MERGE clause and execute it, you can see whether the pattern exists or not in the graph. If it is already in the graph, it functions like the MATCH clause; if it is not in the graph, it newly creates the pattern like the CREATE clause.

- Merge Path

```
MERGE (E:person {name: 'Edward'})-[L:likes]->(A:person {name: 'Alice'})
RETURN E, L, A;

MERGE (E:person {name: 'Edward'})
MERGE (A:person {name: 'Alice'})
MERGE (E)-[L:likes]->(A)
RETURN E, L, A;
```

  The MERGE clause recognizes the entire pattern. Even if the specified pattern partially exists in the graph, it does not create only the rest of it that does not exist. We will explain this with the above query as an example. The first query does not create an edge if the Edward and Alice vertices already exist in the graph and are not linked to the likes edge. New Edward and Alice vertices are created and an edge is created between the two vertices.
  If you are not sure whether some elements of the pattern already exist in the graph, you had better divide each element as in the second query and use the MERGE clause.

- ON CREATE SET and ON MATCH SET

```
MERGE (E:person {name: 'Edward'})
ON CREATE SET E.lastMERGEOP = 'CREATE'
ON MATCH SET E.lastMERGEOP = 'MATCH'
RETURN E.lastMERGEOP;
```

  ON CREATE SET and ON MATCH SET clauses can be used to set properties depending on how the MERGE clause behaves. If the MERGE clause behaves like a CREATE clause, the ON CREATE SET clause will be performed; if the MERGE clause behaves like a MATCH clause, the ON MATCH SET clause will be performed.

Set operations

UNION and UNION ALL

The UNION clause combines the results of several queries into one.

- UNION and UNION ALL

```
MATCH (a:person)
WHERE 20 < a.age
RETURN a.name AS name
UNION
MATCH (b:person)
WHERE b.age < 50
RETURN b.name AS name;

MATCH (a:person)
WHERE 20 < a.age
RETURN a.name AS name
UNION ALL
MATCH (b:person)
WHERE b.age < 50
RETURN b.name AS name;
```

If you put the **`UNION`** keyword between the two queries, you can combine the two results together. In the case of duplicate values, **`UNION`** outputs it only once. When you use both the **`UNION`** and **`ALL`** keywords, the duplicate values (when the two results are combined) will be output as they are.

LOAD Clauses

LOAD FROM

You can load data by table using the LOAD FROM clause.

- create vertex

```
LOAD FROM person AS v
CREATE (:person {id: v.id, name: v.name});
```

- create edge

```
LOAD FROM friend AS e
MATCH (a:person),(b:person)
```

```
WHERE id(a) = to_jsonb(e.start_id)
  AND id(b) = to_jsonb(e.end_id)
CREATE (a)-[:friend {date: '2018'}]->(b);
```

Schema Clauses

CREATE and DROP CONSTRAINT

Provides the ability to control data by setting constraints on properties.

- CREATE and DROP CONSTRAINT

```
CREATE CONSTRAINT [constraint_name] ON label_name ASSERT field_expr IS UN
IQUE
CREATE CONSTRAINT [constraint_name] ON label_name ASSERT check_expr
DROP CONSTRAINT constraint_name ON label_name

CREATE CONSTRAINT ON person ASSERT id IS UNIQUE;
CREATE CONSTRAINT ON person ASSERT name IS NOT NULL;
CREATE CONSTRAINT ON person ASSERT age > 0 AND age < 128;
```

If the constraint name is omitted, it is generated automatically.
UNIQUE restricts the value of a field to be unique in that label.
check_expr returns a true or false value of the properties that are newly-entered or modified. If the result is false, the corresponding input/change will fail.
You may use \dGv, \dGe commands to check the constraints when querying information on the vertices and edges.

```
CREATE VLABEL people;
CREATE CONSTRAINT ON people ASSERT age > 0 AND age < 99;

MERGE (s:people {name: 'David', age: 45});

MATCH (s:people) return s;
                    s
------------------------------------------
 people[24.1]{"age": 45, "name": "David"}
(1 row)


MERGE (s:people {name: 'Daniel', age: 100});
ERROR:  new row for relation "people" violates check constraint "people_p
roperties_check"
DETAIL:  Failing row contains (24.2, {"age": 100, "name": "Daniel"}).
```

```
    MERGE (s:people {name: 'Emma', age: -1});
    ERROR:  new row for relation "people" violates check constraint "people_p
    roperties_check"
    DETAIL:  Failing row contains (24.3, {"age": -1, "name": "Emma"}).

    MATCH (s:people) return s;
                        s
    ------------------------------------------
     people[24.1]{"age": 45, "name": "David"}
    (1 row)
```

functions

Aggregation functions

Create the data to be used in the example.

```
CREATE (:person {name: 'Elsa', age: 20});
CREATE (:person {name: 'Jason', age: 30});
CREATE (:person {name: 'James', age: 40});
CREATE (:person {name: 'Daniel', age: 50});
```

- avg()
  Returns the average of numeric values.

```
    MATCH (v:person)
    RETURN avg(v.age);
```

- collect()
  Returns a list containing the values returned by the expression; aggregates data by merging multiple records or values into a single list.

```
    MATCH (v:Person)
    RETURN collect(v.age);
```

- count()
  Prints the number of result rows; able to print the number or properties of vertices and edges and can be given an alias.

```
    MATCH (v:person)
    RETURN count(v);

    MATCH (v:person)-[k:knows]->(p)
    RETURN count(*);
```

```
MATCH (v:person)
RETURN count(v.name) AS CNT;
```

- min()/max()
  Takes the numeric attribute as input, returns the minimum/maximum values to the corresponding column.

```
MATCH (v:person)
RETURN max(v.age);

MATCH (v:person)
RETURN min(v.age);
```

- stDev()
  Returns the standard deviation. The stDev function returns the standard deviation of the sample population and must cast the property.

```
MATCH (v:person)
RETURN stDev(v.age);
```

- stDevP()
  Returns the standard deviation. The stDevP function returns the standard deviation of the sample population and must cast the property.

```
MATCH (v:person)
RETURN stDevP(v.age);
```

- sum()
  Returns the sum of the numeric values. As it is the sum of the numeric values. The property must be cast.

```
MATCH (v:person)
RETURN sum(v.age);
```

Predicates functions

- all()
  Returns true if all elements in the list satisfy the condition.

```
RETURN ALL(x in [] WHERE x = 0);
```

- any()
  Returns true if at least one element in the list satisfies the condition.

```
RETURN ANY(x in [0] WHERE x = 0);
```

- none()
  Returns true if no elements in the list satisfy the condition.

```
RETURN NONE(x in [] WHERE x = 0);
```

- single()
  Returns true if a single function satisfies only a condition in the list.

```
RETURN SINGLE(x in [] WHERE x = 0);
```

Scalar functions

- coalesce()
  Returns the first non-null value in the list.

```
CREATE (:person {name: 'Jack', job: 'Teacher'});

MATCH (a)
WHERE a.name = 'Jack'
RETURN coalesce(a.age, a.job);
```

- endNode()
  Returns the last node in the relationship.

```
CREATE vlabel Developer;
CREATE vlabel language;
CREATE elabel be_good_at;
CREATE (:Developer {name: 'Jason'})-[:be_good_at]->(:language {name: 'C
'});
CREATE (:Developer {name: 'David'})-[:be_good_at]->(:language {name: 'JAV
A'});

MATCH (x:Developer)-[r]-()
RETURN endNode(r);
```

- head()
  Returns the first element in the list.

```
CREATE (:person {name: 'Richard', array: [ 1, 2, 3 ]});

MATCH(a)
where a.name = 'Richard'
RETURN a.array, head(a.array);
```

- id()
  Returns the relationship or id of the node; returns the node id for all nodes specified in the argument.

```
MATCH (a)
RETURN id(a);
```

- last()
  Returns the last element in the list.

```
MATCH (a)
WHERE a.name = 'Richard'
RETURN a.array, last(a.array);
```

- length()
  Returns the length of a string or path. If you specify the property of a string or a string type as an argument, the number of characters in the string is returned.

```
RETURN length('string');

MATCH (a:person)
WHERE length(a.name) > 4
RETURN a.name;
```

- properties()
  Converts the arguments to a list of key/value mappings. If the argument is already a key/value mapped list, it is returned unchanged.

```
CREATE (p:Person { name: 'Stefan', city: 'Berlin' })
RETURN properties(p);
```

- startNode()
  Returns the starting node of the relationship.

```
MATCH (x:Developer)-[r]-()
RETURN startNode(r);
```

- toBoolean()
  Converts a string to a boolean.

```
RETURN toBoolean('TRUE'), toBoolean('FALSE');
```

- type()
  Returns the elabel of the edge passed as an argument. If the elabel of the edge also inherits another elabel, it returns a parent elabel as well. You should be careful when passing arguments to the type function; when you find an edge that matches the

pattern using MATCH clause, assign a variable, and then pass the variable as an argument, the edge itself cannot be passed as an argument to the type function, but must always be passed as a variable.

```
CREATE elabel loves;
CREATE (:person {name: 'Adam'})-[:loves]->(:person {name: 'Eve'});

MATCH ({name: 'Adam'})-[r]->({name: 'Eve'})
RETURN type(r);
```

List functions

- keys()
  Returns a list containing strings for all attribute names of nodes, relationships, and maps.

```
MATCH (a)
WHERE a.name = 'Jack'
RETURN keys(a);
```

- labels()
  Returns vlabel of the vertex passed as an argument. You should be careful when passing arguments to the label function; when you find a vertex that matches the pattern using MATCH clause, assign a variable, and pass that variable as an argument, the vertex itself cannot be passed as an argument to the label function, but must always be passed as a variable.

```
MATCH (a)
WHERE a.name='Jack'
RETURN labels(a);
```

- nodes()
  Returns a vertex that exists in the path passed as an argument. You should be careful when passing arguments to the nodes function; when you find a path that matches the pattern using MATCH clause, assign a variable, and pass that variable as an argument, the path itself cannot be passed as an argument to the nodes function, but must always be passed as variable. When used with the length function, the number of vertices in the path can be found.

```
MATCH p = (a)-[r]->(b)
WHERE a.name = 'Adam' and b.name = 'Eve'
RETURN nodes(p);

MATCH p = (a)-[r]->(b)
```

```
WHERE a.name = 'Adam' and b.name = 'Eve'
RETURN length(nodes(p));
```

- relationships()
  Returns the edges present in the path passed as an argument. You should be careful when passing arguments to the relationships function; when you find a path that matches the pattern using MATCH clause, assign a variable, and pass that variable as an argument, the path itself cannot be passed as an argument to the relationships function, but must always be passed as variable. When used with the count function, the number of edges in the path can be found.

```
MATCH p =  (a)-[r]->(b)
WHERE a.name = 'Adam' and b.name = 'Eve'
RETURN relationships(p);

MATCH p =  (a)-[r]->(b)
WHERE a.name = 'Adam' and b.name = 'Eve'
RETURN count(relationships(p));
```

- tail()
  Returns a list result that contains all elements except the first element in the list.

```
MATCH (a)
WHERE a.name = 'Richard'
RETURN a.array, tail(a.array);
```

Mathematics functions

Number

- abs()
  Returns a numeric value passed as an argument. It may be passed as a decimal number or as a subtraction. The MATCH clause can be used to find a specific element and pass a subtraction of the properties, which are numeric values, among the properties of the elements.

```
RETURN abs(-3.14);

RETURN abs(20-45);

MATCH (a {name:'Jack'}), (b {name:'Emily'})
RETURN abs(a.age-b.age);
```

- ceil(), floor(), round()
  The ceil function rounds the numeric value passed as an argument to the first decimal

place. The floor function returns the numeric value passed as an argument to the first decimal place. The round function rounds the numeric value passed as the argument to the first decimal place.

```
RETURN ceil(3.1);
RETURN ceil(1);
RETURN ceil(-12.19);

RETURN floor(3.1);
RETURN floor(1);
RETURN floor(-12.19);

RETURN round(3.1);
RETURN round(3.6);
RETURN round(-12.19);
RETURN round(-12.79);
```

- rand()
  Returns an arbitrary floating-point number between 0 and 1.

```
RETURN rand();
```

- sign()
  Returns the sign of the numeric value passed as an argument; returns '1' if the argument passed is positive, '-1' if negative, and '0' if zero.

```
RETURN sign(25);
RETURN sign(-19.93);
RETURN sign(0);
```

Logarithmic

- log()
  Returns the natural logarithm of a number.

```
RETURN log(27);
```

The natural logarithm of 27 is returned.

- log10()
  Returns the common logarithm (base 10) of a number.

```
RETURN log10(27);
```

The common logarithm of 27 is returned.

- exp()
  The exp function returns a power value to base (e) exponentiated by the numeric value passed as an argument. That is, exp(1) returns e^1≒2.71828182845905, exp(2) returns e^2≒7.38905609893065, and exp(-1), e^-1≒0.367879441171442.

```
RETURN exp(1);
RETURN exp(2);
RETURN exp(-1);
```

- sqrt()
  Returns the square root of the numeric value passed as an argument. The sqrt function cannot pass a negative number as an argument.

```
RETURN sqrt(25);
```

Trigonometric

- sin()/cos()/tan()
  The sin, cos, and tan functions return the sine, cosine, and tangent values of the numeric values passed as arguments, respectively. sin(), cos(), and tan() print the values in radians; sind(), cosd(), and tand() are used to print the values in degrees.

```
RETURN sin(0.5);
RETURN sin(-1.5);

RETURN cos(0);
RETURN cos(-1);

RETURN tan(0);
RETURN tan(15.2);
```

- cot()/asin()/acos()/atan()/atan2()
  The cot function returns a cotangent value (inverse of tangent) of the numeric value passed as an argument, the asin function returns an arcsine value (inverse of sine) of the numeric value passed as the argument, and the acos function returns an arccosine value of the numeric value (inverse of cosine). The atan, atan2 functions return the arctangent value (inverse of tangent) of the numeric value passed as an argument. The argument range of the acos function is a numeric value between -1 and 1. atan2 has two arguments in order to make the atan function more granular. Conceptually, atan2(a, b) is equivalent to atan (a/b). However, it is not clear whether atan(-5) is atan(-15/3) or atan (15/(-3)). The trigonometric function requires a definite distinction because the argument is a radian value. Therefore, using atan2 rather than atan makes it more accurate.

```
RETURN cot(1.5);

RETURN asin(1);

RETURN acos(0.5);

RETURN atan(-1);

RETURN atan2(-1.5, 1.3);
```

- pi()/degrees()/radians()
  The pi function returns pi as a number. The degrees function takes the arguments passed as radians and returns them to degrees. The radians function converts the arguments passed to degrees into radians.

```
RETURN pi();

RETURN degrees(12.3);
RETURN degrees(pi());

RETURN radians(180);
```

String functions

- replace()
  If the second argument is contained in the first argument, replace the second argument with the third argument.

```
RETURN replace('ABCDEFG', 'C', 'Z');
RETURN replace('ABCDEFG', 'CD', 'Z');
RETURN replace('ABCDEFG', 'C', 'ZX');
RETURN replace('ABCDEFG', 'CD', 'ZXY');
```

- substring()
  Prints the first argument from the nth digit (n is the number indicated by the second argument). The third argument indicates how many characters to be printed. If there is no third argument and it is greater than the number of characters in the first argument, it prints to the end.

```
RETURN substring('ABCDEFG', 2);
RETURN substring('ABCDEFG', 2, 3);
RETURN substring('ABCDEFG', 4, 10);
```

- left()/right()
  The left function prints the first argument from the left, and the right function prints

characters up to length of n (n is the number indicated by in the second argument) from the right. If the value of the second argument is larger than the number of characters remaining, the number of characters remaining will be printed out.

```
RETURN left('AAABBB', 3);
RETURN right('AAABBB', 3);
```

- lTrim()/rTrim()
  The lTrim function removes all left whitespace from the passed argument, and the rTrim function removes all right whitespace before printing.

```
RETURN lTrim('    ABCD    ');
RETURN rTrim('    ABCD    ');
```

- toLower()/toUpper()
  The toLower function converts all passed arguments to lower case and the toUpper function converts them to upper case.

```
RETURN toLower('AbCdeFG');
RETURN toUpper('AbCdeFG');
```

- reverse()
  Prints the arguments in reverse order.

```
RETURN reverse('ABCDEFG');
```

- toString()
  Converts an integer, floating, or boolean value to a string.

```
RETURN toString(11.5), toString('already a string'), toString(TRUE);
```

- trim()
  Returns the original string with the leading and trailing spaces removed.

```
RETURN trim('   hello   ');
```

SQL Language

Introduction

AgensGraph supports SQL for relational data queries. It supports DDL (create, alter, drop, etc.) for creating, modifying, and deleting objects such as table, column, constraints, and schema and DML (insert, update, delete, etc.) for inserting, modifying and deleting data.

SQL Syntax is identical with PostgreSQL's SQL Syntax. See PostgreSQL-The SQL Language for more information.

Data Type

AgensGraph provides diverse data types. You can add new types as well using CREATE TYPE command. Refer to the User-defined Type clause for more information on CREATE TYPE.

The following table lists the generic data types provided by default, and some of the types that are used internally or not used may not be included. ("Alias" is an internally-used name.)

Name	Alias	Description
bigint	int8	Signed 8-byte integer
bigserial		Auto-incrementing 8-byte integer
bit [(n)]	16-bit integer	Fixed length bit string
bit varying [(n)]	varbit	Variable-length bit string
boolean	bool	Logical Boolean (true / false)
box		Square box on a plane
bytea		Binary data ("byte array")
character [(n)]	char [(n)]	Fixed-length character string
character varying [(n)]	varchar [(n)]	Variable-length character string
cidr		IPv4 or IPv6 network address
circle		Circle on a plane
date		Calendar date (year, month, day)

double precision	float8	Double-precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	Signed 4-byte integer
interval [fields] [(p)]		Time range
json		Text JSON data
jsonb		Binary JSON data, disjointed
line		Infinite straight line on a plane
lseg		Segment on a plane
macaddr		Media Access Control (MAC) address
money		Traffic volume
numeric [(p, s)]	decimal [(p, s)]	The exact number of selectable digits
path		Geometric path in the plane
pg_lsn		AgensGraph log sequence number
point		Geometric points on a plane
polygon		Geometrically-closed path in plane
real	float4	Single-precision floating-point number (4 bytes)
smallint	int2	Signed two-byte integer

smallserial	serial2	Auto-incrementing 2-byte integer
serial	serial4	Auto-incrementing 4-byte integer
text		Variable-length character string
time [(p)] [without time zone]		Time (no time zone)
time [(p)] with time zone	timetz	Includes time and time zone
timestamp [(p)] [without time zone]		Date and time (no timezone)
timestamp [(p)] with time zone	timestamptz	Date and time, including time zone
tsquery		Text search query
tsvector		Text Search Document
txid_snapshot		User-level transaction ID snapshot
uuid		Universal unique identifier
xml		XML data

Numeric Types

A numeric type consists of a 2/4/8 byte integer, a 4/8 byte floating point numbers, and a selectable total number of digits.

The following numeric types are available:

Name	Storage Size	Description	Range
smallint	2 bytes	A small range of	-32768 to +32767

		integers	
integer	4 bytes	Common integer	-2147483648 to +2147483647
bigint	8 bytes	A large range of integers	-9223372036854775808 to +9223372036854775807
decimal	variable	Custom precision, correct	Up to 131072 digits before the decimal point, up to 16383 digits after the decimal point
numeric	variable	Custom precision, correct	Up to 131072 digits before the decimal point, up to 16383 digits after the decimal point
real	4 bytes	Variable precision, incorrect	6 digits precision
double precision	8 bytes	Variable precision, incorrect	15 digits precision
smallserial	2 bytes	Automatic incremental constant (small)	1 to 32767
serial	4 bytes	Automatic incremental constant	1 to 2147483647
bigserial	8 bytes	Automatic incremental integer (large)	1 to 9223372036854775807

Integer Types

smallint, integer, and bigint types store a wide range of integers without decimal fractions. If you try to store a value beyond the allowable range, an error will occur.

- integer type: This type is generally chosen as it provides the best balance point of range, storage size, and performance.

- smallint type: Typically used only when there is insufficient disk space.
- bigint type: To be used when the integer type range is insufficient.

SQL specifies only integer (or int), smallint, and bigint. (available as int2, int4, and int8 as well).

Arbitrary Precision Numbers

Numeric types may store a very large number of digits and perform calculations correctly. It is especially recommended when storing amounts and quantities that should be accurate. Arithmetic of numeric values, however, is much slower than the integer types or floating-point types described in the next section.

Scale in the numeric type means the number of digits to the right of the decimal point. Precision means the total number of significant digits of the total number. That is, the total number of digits on both sides of the decimal point. Thus, precision and scale of the number 23.5141 is 6 and 4, respectively. The scale of whole number can be considered 0 (Scale 0).

You may configure both the maximum precision and the maximum scale of a numeric column.

To declare a column of numeric type, use the following syntax:

```
NUMERIC(precision, scale)
```

Precision must be positive and scale must be zero or positive.

```
NUMERIC(precision)
```

If you specify numeric without precision or scale as follows, it selects Scale 0.

```
NUMERIC
```

If you create a numeric column that can store precision and scale values without specifying them, you may store the precision value. This kind of column can be used without restriction if it does not specify a specific scale. However, a numeric column with a scale value specified will be limited by the scale value. (When transferring data, make sure to specify the precision and scale always).

Note: The maximum allowable precision is 1000 if explicitly specified in the type declaration. NUMERICs that do not specify precision are limited to the ranges described in the table.

In the case where the scale in the value to be stored is greater than the scale declared in the column, the system rounds off the value to the specified scale; after rounding-off, if the number of digits to the right of the decimal point exceeds "the declared scale subtracted from the declared precision," an error will occur. Numeric values are physically stored without extra `0` values. Therefore, the precision and scale of the declared column is the maximum (not a fixed allocation). In this sense, numeric types are closer to varchar (n) than to char (n). The actual storage requirement is 2 bytes for each 4-digit group plus 3 for 8-byte overhead.

The decimal and numeric types are the same and both are SQL standards.

Floating-Point Types

The real and double precision data types are inaccurate variable-precision numeric types. Inaccuracy means that some values cannot be accurately converted to their internal form and are stored as approximate values, making the storage and retrieval of values somewhat inconsistent.

- If you need accurate storage and computation (e.g. amount), use the numeric type instead.

- If you need to perform complex calculations using this type for some unavoidable reason (e.g. when you need a specific behavior in boundary cases (infinity, underflow)), you should be careful with the implementation.

- Comparing two floating-point values to see if they are equal may not always work as expected.

On most platforms, the real type has a minimum range of 1E-37 to 1E+37, with precision of at least 6. The double precision type generally has precision of at least 15, ranging from 1E-307 to 1E+308. Too large or too small values will generate an error. If precision of the entered number is too large, it may be rounded up. An underflow error occurs if the number is so close to zero that it cannot be marked as non-zero.

Serial Types

The smallserial, serial, and bigserial data types are not actual types, but are international notations for creating unique identifier columns (similar to the AUTO_INCREMENT attribute supported by some other databases).

The following two statements work in the same manner:

```
-- SERIAL
CREATE TABLE tablename (
colname SERIAL
);

-- SEQUENCE
CREATE SEQUENCE tablename_colname_seq;

CREATE TABLE tablename (
colname integer NOT NULL DEFAULT nextval ('tablename_colname_seq')
);

ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

Create an integer column and sort it by default assigned in the sequence. A NOT NULL constraint is applied to prevent null values from being inserted. (In most cases, it is possible to prevent duplicate values from being accidentally entered with a UNIQUE or PRIMARY KEY constraint; such a constraint is not automatically generated.)

Finally, the sequence is marked as column "owned by," so that it is not deleted unless the column or table is deleted.

The serial column default is specified in order to insert the next value of sequence into the serial column. This can be done by excluding columns from the column list of the INSERT statement or by using the DEFAULT keyword.

- The serial type is the same as serial4; both generate an integer column.
- The bigserial and serial8 types work the same except when creating a bigint column. bigserial should be used if you expect to use more than 231 identifiers throughout the lifetime of the table.
- The smallserial and serial2 types operate the same except when creating a smallint column.

The sequence created for the serial column is automatically deleted when the owning column is deleted. You can delete the sequence without deleting the column, but the column base expression is forcibly deleted.

Character Types

The following character types are available:

Name	Description
character varying(n), varchar(n)	Variable with limit

character(n), char(n)	Fixed length, fill in blank
text	Unlimited variable length

The two basic character types define character variation(n) and character(n). The value of n is a positive integer, and both can store up to n length characters (not bytes).

If you store a string that is longer than n, an error occurs if the excess string is not empty; if it is blank, it is truncated to the length of the specified n value. If you store a string that is shorter than n characters, it is filled with blanks for the character type. For character varying, the string except blanks is stored.

Trailing blanks in character types are treated as not syntactically significant and are ignored when comparing two values of the character type. Trailing blanks are syntactically significant when using pattern-matching regular expressions such as character varying, text, and LIKE.

char(n) and varchar(n) are aliases of character variation(n) and character(n). A character without a specifier is equivalent to a character(1), and a character varying without a specifier stores strings regardless of its size. It also provides a text type for storing strings, regardless of length.

An example of using the character type is shown below:

```
create table test1 (a character (4));
CREATE TABLE
insert into test1 values ('ok');
INSERT 0 1
select a, char_length(a) from test1;
  a   | char_length
------+-------------
 ok   |           2

create table test2 (b varchar(5));
CREATE TABLE
insert into test2 values ('ok');
INSERT 0 1
insert into test2 values ('good ');
INSERT 0 1
insert into test2 values ('too long');
Error: character varying(5) tries to store data that is too long for the data
 type.
insert into test2 values ('too long'::varchar(5));
INSERT 0 1
```

```
select b, char_length(b) from test2;
   b    | char_length
-------+-------------
 ok     |           2
 good   |           5
 too l  |           5
```

Date/Time Types

The following date/time types are available:

Name	Storage Size	Description	Low Value	High Value	Resolution
timestamp [(p)] [without time zone]	8 bytes	Both date and time (no timezone)	4713 BC	294276 AD	1 microsecond / 14 digits
timestamp [(p)] with time zone	8 bytes	Both date and time, including timezone	4713 BC	294276 AD	1 microsecond / 14 digits
date	4 bytes	Date (no time)	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	8 bytes	Time (no date)	00:00:00	24:00:00	1 microsecond / 14 digits
time [(p)] with time zone	12 bytes	Include time of day, time zone	00:00:00 +1459	24:00:00-1459	1 microsecond / 14 digits
interval [fields] [(p)]	16 bytes	Time interval	-178000000 years	178000000 years	1 microsecond / 14 digits

Note: The SQL standard uses timestamp and timestamp without time zone equally, and timestamptz is the abbreviation for timestamp with time zone.

time, timestamp, and interval may set the scale (the number of digits of the decimal fraction) on the second field as the p value, and there is no explicit restriction on precision (total number of digits) by default. The allowable range of p is 0 to 6 for timestamp and interval type.

For the time type, the allowable range of p is 0 to 6 when using 8-byte integer storage, and 0 to 10 when using floating-point storage.

The interval type has an additional option of limiting the set of stored fields by writing one of the following statements:

```
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
```

Date/Time Input

The date and time input can be specified in the order of day, month, and year as follows:

```sql
set datestyle to sql, mdy;
set datestyle to sql, dmy;
set datestyle to sql, ymd;
```

Set the DateStyle parameter to MDY to select the month-day-year interpretation, DMY to select the day-month-year interpretation, or YMD to select the year-month-day interpretation.

The date or time input must be preceded and followed by single quotation marks, like a text string.

- Date
  Available date types:

Example	Description
1999-01-08	ISO 8601. January 8 in random mode (recommended format)
January 8, 1999	Ambiguous in datestyle input mode

1/8/1999	January 8 in MDY mode. August 1 in DMY mode
1/18/1999	January 18 in MDY mode. Rejected in other modes
01/02/03	January 2, 2003 in MDY mode February 1, 2003 in DMY mode February 3, 2001 in YMD mode
1999-Jan-08	January 8 in random mode
Jan-08-1999	January 8 in random mode
08-Jan-1999	January 8 in random mode
99-Jan-08	January 8 in YMD mode; others are errors
08-Jan-99	January 8, except error in YMD mode
Jan-08-99	January 8, except error in YMD mode
19990108	ISO 8601. January 8, 1999 in random mode
990108	ISO 8601. January 8, 1999 in random mode
1999.008	Year and day of the year
J2451187	Julian date
January 8, 99 BC	Year 99 BC

- Time
  The visual type is time [(p)] without time zone and time [(p)] with time zone. time alone is the same as time without time zone. A valid entry of this type consists of time, followed by an optional time zone. (See the table below.) If the time zone is specified as an input for time without time zone, it is ignored. You can specify the date, but ignore it except when using the time zone name associated with daylight saving time, such as America/New_York. In this case, a date specification is required to determine

whether the standard time or daylight saving time period is applied. The appropriate time zone offset is recorded in the time with time zone value.

Table - [Enter Time]

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	Same as 04:05. AM does not affect the value.
04:05 PM	Same as 16:05. Input must be <= 12.
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	Time zone specified by abbreviation
2003-04-12 04:05:06 America/New_York	Time zone specified by full name

Table - [En ter Time Slot]

Example	Description
PST	Abbreviation (for Pacific Time)

America/New_York	All time zone names
PST8PDT	POSIX style time zone specification
-8:00	ISO-8601 Offset for PST
-800	ISO-8601 Offset for PST
-8	ISO-8601 Offset for PST
zulu	Military acronym for UTC

See the Time Zones section below for how to specify the time zone.

- Time stamp
  The valid input of timestamp type consists of a connection of date and time followed by an optional time zone and optional AD or BC. (AD/BC may appear before the time zone, which is not the preferred order.)

```
1999-01-08 04:05:06
1999-01-08 04:05:06 -8:00
```

These are valid values in accordance with the ISO 8601 standard. The following command format is also supported:

```
January 8 04:05:06 1999 PST
```

The SQL standard distinguishes timestamp without time zone and timestamp with time zone literals by the presence of a "+" or "-" sign and the time zone offset after that time.

```
 -- timestamp without time zone
 TIMESTAMP '2004-10-19 10:23:54'

 -- timestamp with time zone
 TIMESTAMP '2004-10-19 10:23:54+02'
```

If you do not specify timestamp with time zone, it is treated as timestamp without time zone; thus, if you use timestamp with time zone, you must specify an explicit type.

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

- Special values
  Some special date/time input values are supported for convenience as shown in the table. The values infinity and -infinity are simple abbreviations that are specially represented within the system, without modification, but otherwise converted to the default date/time value when read. (Note the now and related strings are converted to specific time values at the moment they are read.) If you use all of these values as constants in your SQL command, you must use single quotation marks around the constants.

Input String	Valid Type	Description
epoch	date, timestamp	1970-01-01 00: 00: 00 + 00 (Unix system time 0)
infinity	date, timestamp	After all other timestamps
-infinity	date, timestamp	Before all other timestamps
now	date, time, timestamp	Start time of current transaction
today	date, timestamp	Midnight today
tomorrow	date, timestamp	Tomorrow midnight
yesterday	date, timestamp	Yesterday midnight
allballs	time	00:00:00.00 UTC

  You can also use the following SQL-compatible functions to get the current time values of the data types such as CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, and LOCALTIMESTAMP.

Date/Time Output

The output format of the date/time type can be set to one of four styles: ISO 8601, SQL (Ingres), typical POSTGRES (Unix date format) or German. The default is ISO format.

The following table shows an example of each output style. The output of date/time type has usually only the date or time depending on the example given.

Style Specification	Description	Example
ISO	ISO 8601, SQL standard	1997-12-17 07:37:16-08
SQL	Traditional style	12/17/1997 07:37:16.00 PST
Postgres	Original style	Wed Dec 17 07:37:16 1997 PST
German	Local style	| 17.12.1997 07:37:16.00 PST

If the DMY field order is specified, it is output in order of month, day. In other cases, it is output in order of day and month. The following table is an example:

datestyle Setting	Input Ordering	Example Output
SQL, DMY	day/month/year	17/12/1997 15:37:16.00 CET
SQL, MDY	month/day/year	12/17/1997 07:37:16.00 PST
Postgres, DMY	day/month/year	Wed 17 Dec 07:37:16 1997 PST

The date/time style can be selected by the user using SET datestyle command, DateStyle parameter in the postgresql.conf configuration file, or PGDATESTYLE environment variables of the server or client. The formatting function to_char allows you to specify date/time output formats in a more flexible manner.

Time Zones

Time zones and notations are affected by political decisions in addition to topographical features. Worldwide time zones have been standardized in the 1900s but are constantly changing due to daylight time regulations. AgensGraph uses the IANA (Olson) timezone database. For future times, it considers that the latest known rules for a specified time zone will be complied with indefinitely in the distant future. However, it has a peculiar mix of dates, time types and features.

Two obvious problems are:

- Date type does not have an associated time zone (only time type has). In the real world, the time zone has no meaning unless it is associated with date and time, since the offset can vary over the year containing the daylight saving time boundary.

- The default time zone is specified as a constant numeric offset from UTC. Thus, it may not be possible to adapt to daylight saving time when performing date/time calculations across DST boundaries.

When using time zone to solve this problem, we recommend using date/time type that includes both date and time (this is supported for compatibility but we do not recommend using the time with time zone type). Assume a local time zone for types that only contain dates or times. The date and time in the time zone are internally stored in UTC. Then, they are converted to a local time in the Time Zone specified by the TimeZone configuration parameters before being displayed to the client. Allow the time zone to be specified in three different formats.

- The full names of time zone (e.g. America/New_York). The recognized time zone names are listed in the
  pg_timezone_names view. (Identical time zone names are recognized by many other software applications as well.)

- Time zone abbreviations (e.g. PST). In contrast to the full names of time zone that can imply the daylight time conversion date rule set, the corresponding specification simply defines a specific offset from UTC. The recognized abbreviations are listed in the pg_timezone_abbrevs view. You cannot set the configuration parameters TimeZone or log_timezone as a time zone abbreviation, but may use an abbreviation and AT TIME ZONE operator as date/time input values.
- In addition to time zone names and abbreviations, it accepts POSIX style time zone specifications of STDoffset or STDoffsetDST; STD is a regional abbreviation, offsetUTC is the numerical offset for time of the west, and DST is the optional summertime local abbreviation that is assumed to be one hour earlier than the specified offset. For instance, if EST5EDT is not yet a recognized local name, it is accepted and becomes functionally identical to the US East Coast time. In this syntax, local abbreviations can be any string of characters or any string using angle brackets (<>). If a daylight saving area abbreviation is present, it is assumed to be used in accordance with the same daylight saving time conversion rules as those used in the posixrules entry in the IANA time zone database. As posixrules are identical with US/Eastern in a standard AgensGraph installation, the POSIX style time zone specification complies with the USA Daylight Saving Time rules. You can adjust this behavior by replacing the posixrules file, if necessary.

Simply put, this is the difference between abbreviations and full names. Time zone abbreviations represent a specific offset from UTC, while many of time zone full names imply a local daylight time rule and have two possible UTC offsets. For instance, "2014-06-04 12:00 America/New_York" representing the noon in the New York area was Eastern

Daylight Time (UTC-4) for the day. Accordingly, "2014-06-04 12:00 EDT" specifies the same time instance. However, "2014-06-04 12:00 EST" specifies noon Eastern Standard Time (UTC-5) regardless of whether daylight saving time is nominally in effect on that date.

To complicate the problem, some areas use the same time zone abbreviations meaning different UTC offsets at different times. For example, MSK in Moscow meant UTC+3 for several years, and UTC+4 for other cases. Even if these abbreviations are interpreted according to their meanings for a given date (or most recent meaning), as in the above EST example, this does not necessarily match the local time of the date. Note that the POSIX style time zone function does not check for the correctness of local abbreviations; this means you may silently accept false input. For example, SET TIMEZONE TO FOOBAR0 works by letting the system use specific abbreviations for UTC. Another problem to keep in mind is that positive offset is used for the Greenwich's position west in the POSIX time zone region name. Elsewhere, AgensGraph conforms to the ISO-8601 notation, where the positive time-zone offset is east of Greenwich. Time zone names and abbreviations are case sensitive and are not embedded in the server; they are to be searched from the configuration files stored in the installation directory (.../share/timezone/and .../share/timezonesets/). TimeZone configuration parameters can be set in other standard ways, which is different from postgresql.conf as follows:

- The SQL command SET TIME ZONE sets a time zone for the session. You can use SET TIMEZONE TO, which is more compatible with the SQL specification.

- The PGTZ environment variable is used by the libpq client to send SET TIME ZONE command to the server when connected.

Interval Input

The interval value can be written using the following detailed syntax:

```
[@] quantity unit [quantity unit...] [direction]
```

Where quantity is a number and unit can be microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium, or their abbreviations, singular or plural; direction can be ago or empty. The at (@) sign is an optional noise. Quantities in different units are implicitly added using an appropriate sign accounting. This syntax is also used for interval output when IntervalStyle is set to postgres_verbose.

Days, hours, minutes and seconds can be specified without explicitly marking units. For example, "1 12:59:10" is read the same as "1 day 12 hours 59 min 10 sec."

Combinations of years and months can be specified using dashes as well. For example, "200-10" is read the same as "200 years 10 months." (This short format is actually the only one allowed by the SQL standards, and is used for output if IntervalStyle is set to sql_standard).

The format using specifiers:

```
P quantity unit [ quantity unit ...] [ T [ quantity unit  ...]]
```

The string must contain P, and may contain T to include the unit of time. Available unit abbreviations are listed in the table below. Units can be omitted and can be specified in any order, but units of less than one day must appear after T. In particular, the meaning of M varies depending on whether it is before or after T.

Table [ISO 8601 Interval Unit Abbreviations]

Abbreviation	**Meaning**
Y	Year
M	Month(in the date part)
W	Week
D	Day
H	Hour
M	Minute(in the time part)
S	Second

Alternative format:

```
P [ years-months-days ] [ T hours:minutes:seconds ]
```

The string must start with P, and T separates the date and time of the interval. The value is specified as a number similar to the ISO 8601 date. If you create an interval constant with the fields specification, or if you assign a string to an interval column defined by the fields specification, the interpretation of the unmarked quantity varies depending on the fields. For example, INTERVAL '1' YEAR is interpreted as a year, whereas INTERVAL '1' means 1 second. The lowest right field value allowed by the fields specification is ignored.

For example, INTERVAL '1 day 2:03:04' HOUR TO MINUTE will eventually delete the "seconds" field (not the date field). As all fields of interval values must have the same signs according to the SQL standards, the leading negative signs apply to all fields. For example, the negative sign in the interval literal '-1 2:03:04' applies to both date and hour/minute/second parts. As the fields are allowed to have different signs and the signs of each field are independently processed in text representation, the hour/minute/second part is regarded as a positive value in this example. If IntervalStyle is set to sql_standard, the leading sign is assumed to be applied to all fields (if there is no additional sign).

If the field is negative, it is better to explicitly append the (negative) sign to avoid ambiguity. Internally, interval values are stored as months, days, and seconds. This is because the number of days in the month is different, and a day can be 23 or 25 hours depending on implementation of daylight saving time. The month and day fields are integers, and the seconds field can be stored as decimals. Since the interval is usually generated by constant strings or timestamp subtraction, this way of storage is not problematic in most cases. The functions justify_days and justify_hours are useful when you want to control overflowing days and times in the normal range. Field values in some fields of the detailed input format and the simpler input field may have a decimal fraction (e.g. "1.5 week" or "01:02:03.45"). These inputs are converted to an appropriate number of months, days, and seconds for storage. This results in the decimal(s) of months or days being added to a subfield using the conversion factor Jan=30 days and 1 day=24 hours. For example, "1.5 month" is one month and 15 days. Only the seconds are displayed with a decimal fraction. The table below shows some examples of valid interval inputs.
Table [Interval Input]

Example	**Description**
1-2	SQL standard format: 1 year 2 months
3 4:05:06	SQL standard format: 3 days 4 hours 5 minutes 6 seconds
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	Typical Postgres format: 1 year 2 months 3 days 4 hours 5 minutes 6 seconds
P1Y2M3DT4H5M6S	ISO 8601 "format with designators": same as above
P0001-02-03T04:05:06	ISO 8601 "alternative format": same as above

Interval Output

The output format of interval type can be set to one of the four styles sql_standard, postgres, postgres_verbose, or iso_8601 using the command SET intervalstyle. The default is postgres. The table below shows an example of each output style. The sql_standard style generates an output that conforms to the SQL standard specification for the interval literal string if the interval value satisfies the standard limit (year-month or day-minute only if positive and negative values are not mixed). Otherwise, a sign is explicitly added to eliminate the confusion of the sign mixing interval; the output appears as if the standard year-month literal string is followed by a day-time literal string.
Table [Example of interval output style]

Style Specification	Year-Month Interval	Day-Time Interval	Mixed Interval
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06
postgres	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
postgres_verbose	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

Boolean Type

It provides a standard SQL type boolean; on top of "true" and "false" states, it has "unknown," a third state that is expressed as SQL null.

Name	Storage Size	Description
boolean	1 byte	True or False state

- Valid literal values in the true state are:

```
TRUE
't'
'true'
'y'
'yes'
```

```
    'on'
    '1'
```

- In the false state, you may use the following values:

```
    FALSE
    'f'
    'false'
    'n'
    'no'
    'off'
    '0'
```

Leading or trailing blanks are ignored and it is not case sensitive. Using the keywords TRUE and FALSE is preferred. The following example shows that a boolean value is displayed (output) using the characters t and f.

An example of using the boolean type:

```
CREATE TABLE test1 (a boolean, b text);

INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');

SELECT * FROM test1;
a | b
---+---------
t | sic est
f | non est

SELECT * FROM test1 WHERE a;
a | b
---+---------
t | sic est
```

Geometric Types

Geometric types display two-dimensional spatial objects and the available geometric types are as follows:

Name	Storage Size	Description	Representation
point	16bytes	Point on the plane	(x,y)
line	32bytes	Infinite straight line	{A,B,C}

lseg	32bytes	Infinite segment	((x1,y1),(x2,y2))
box	32bytes	Square box	((x1,y1),(x2,y2))
path	16+16n bytes	Closed path (similar to a polygon)	((x1,y1),...)
path	16+16n bytes	| Open path	| [(x1,y1),...]
polygon	40+16n bytes	Polygon (similar to a closed path)	((x1,y1),...)
circle	24bytes	Circle	<(x,y),r> (Center point and radius)

Points

Points are the basic two-dimensional building blocks for geometric types. The value of point type is specified using one of the following syntaxes:

```
( x , y )
  x , y
```

Where x and y are the floating-point coordinates, respectively. Points are output using the first syntax.

Lines

Lines are expressed by a linear equation Ax+By+C=0; where A and B are both nonzero. The value of line type is the input and output in the following format.

```
{ A, B, C }
```

Alternatively, you can use one of the following formats for input:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

Where (x1, y1) and (x2, y2) are two different points on a straight line.

Line Segments

A line segment is represented by a pair of points that form the two end points of a line segment. The value of lseg type is specified using one of the following syntaxes.

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

Where (x1, y1) and (x2, y2) are the two end points of a line segment. The Line Segment is output using the first syntax.

Boxes

A box is represented by a pair of points that form the opposite corners of the box. The value of box type is specified using one of the following syntaxes:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

Where (x1, y1) and (x2, y2) are the two opposite corners of the box. A box is output using the second syntax. Two opposite corners are provided as inputs, but values are reordered as needed to be saved as upper right corner and lower left corner.

Paths

A path is represented by a list of connected points. If the first and last points of the list are considered unconnected, it is an open path. If the first and last points are considered to be connected, it is a closed path. The value of path type is specified using one of the following syntaxes:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

Where the points are the two end points of a line segment constituting the path. Square brackets ([]) indicate open paths and parentheses (()) indicate closed paths. If the outermost parentheses are omitted, as in the third through fifth statements, they are considered closed paths. The path is output using the second syntax properly.

Polygons

Polygons are represented by a list of points (polygon vertices). Polygons are very similar to closed paths, but are stored differently and have their own set of routines supported. The value of polygon type is specified using one of the following syntaxes:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

Where the points are the two end points of a line segment constituting the boundary of the polygon. The polygon is output using the first syntax.

Circles

A circle is represented by the center point and radius. The value of circle type is specified using one of the following syntaxes:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

Where (x, y) is the center of the circle and r is the radius. Circle is output using the first syntax.

XML Type

The XML type is used to store XML data. It allows you to inspect input values in a well-formed format rather than storing XML data in a text field, and has functions that supports safe operations. This data type requires an installation built with configure --with-libxml.

Creating XML Values

To generate an xml type value from character data, use the function below:

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Examples:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter
>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

The XML type does not check the input value for the document type definition (DTD) even if the input value is specified as DTD. It does not support validation of other XML schema languages (e.g. XML schema).

The inverse operation to generate a string value in xml uses the following function:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

type can be character, character varying, text (or an alias in it). According to the SQL standard, this is the only way to convert XML and character types, but may simply cast the values.

Choosing DOCUMENT and CONTENT is determined by the "XML Option" session configuration parameters, which can be set using standard commands when a string value is cast to xml type or passes through XMLSARIALIZE without going through XMLPARSE or XMLSERIALIZE.

```
SET xmloption TO { DOCUMENT | CONTENT };
```

As the default is CONTENT, XML data in any format are allowed.

Note: You cannot directly convert a string containing DTD to an XML format, since the definition of an XML content fragment does not allow XML if the default XML option settings are used. You should use XMLPARSE or change the XML option.

Encoding Handling

Care should be taken when you process multiple character encodings on the client and when XML data is passed through it. If you use text mode to pass a query to the server and pass a query result to the client (normal mode), convert all character data passed between the client and server, and convert the character encoding backwards at each end. It contains a text representation of the XML value as shown in the example above. Usually this implies that the encoding declaration contained in the XML data can be invalidated, as the embedded encoding declaration does not change, if the character data is converted to another encoding during transmission between the client and server. To cope with this behavior, the encoding declaration contained in the character string that exists for the XML type input is ignored and CONTENT is regarded as the current server encoding. As a result, strings of XML data must be transmitted from the client through the current client encoding for proper processing. It is the client's responsibility to convert the document to the current client encoding or to properly adjust the client encoding before sending it to the server. The value of type XML in the output does not have an encoding declaration, and the client considers all data to be the current client encoding.

If you use binary mode to pass query parameters to the server and pass the query results back to the client, the character set conversion is not performed, making things more difficult. In this case, the encoding declaration of the XML data is complied with; if absent, the data is assumed to be UTF-8 (note that this is required by the XML standard and does not support UTF-16). At the output, the data has an encoding declaration specifying the client encoding, if it is not the client encoding is UTF-8; if it is, the encoding declaration will be omitted.

Processing XML data is less likely to cause errors, and is more efficient when the XML data encoding, client encoding, and server encoding are all identical. Since XML data is internally processed as UTF-8, the calculation is most efficient when the server is UTF-8 as well.

Note: Some XML-related functions may not work with non-ASCII data if the server encoding is not UTF-8. This is especially known as a problem in xpath ().

Accessing XML Values

The XML data type is unique in that it does not provide a comparison operator. This is because there is no well-defined and universally useful comparison algorithm for XML data. As a result, you cannot retrieve a row by comparing the xml column with the search value. You must use XML with a separate key field (e.g. ID). An alternative solution to compare XML values is to convert it to a string first. Note, however, that string comparisons have nothing to do with useful XML comparison methods.

Since there is no comparison operator for XML data types, it is not possible to create an index directly on this type of column. If you need to retrieve XML data fast, the possible solutions include casting and indexing expressions to character string types or indexing XPath expressions. Of course, the search should be adjusted by the indexed expressions in actual querying.

The text search feature can also be used to speed up retrieval of entire documents in XML data. However, the necessary preprocessing support is not yet available.

JSON Types

The JSON data type is for storing JavaScript Object Notation (JSON) data as specified in RFC 71591. The applicable data can also be saved as text, but the JSON data type has an advantage of enforcing each stored value to be valid according to the JSON rules. There are also a number of JSON-specific functions and operators available for these types of stored data.

There are two JSON data types: JSON and JSONB. These two types accept almost the same set of values as input. A substantial difference between the two is efficiency. The JSON data type stores an exact copy of the input text, and its processing function must be re-parsed for each execution. On the other hand, the JSONB data is stored in decomposed binary format, which is a bit slower on input due to the added conversion overhead, but much faster during processing because no re-parsing is required. JSONB also supports indexing, which can be a significant advantage.

The JSON type stores exact copies of the input text, preserving syntactically-insignificant whitespace between tokens and key sequences in the JSON object. In addition, if a JSON object inside a value contains the same key more than once, all key/value pairs are retained. (Processing functions assume the last value is valid.) Conversely, JSONB does not retain whitespace, the order of the object keys, and duplicate object keys. If a duplicate key is specified in the input, only the last value is retained. In general, most applications should store JSON data as JSONB unless there is a specific reason such as existing assumptions about object key ordering. Only one character set encoding per database is allowed. Therefore, when the database encoding is not UTF8, it is impossible for the JSON type to strictly comply with the JSON specification. Attempts to directly include characters that cannot be represented in the database encoding would fail. In contrast, characters that cannot be represented by UTF8 but can be represented by database encoding are allowed. RFC 7159 allows a JSON string to contain a Unicode escape sequence denoted by `\uXXXX`. In a JSON type input function, Unicode escapes are allowed regardless of the database encoding, and are checked for syntactical correctness (i.e. \u followed by four hexadecimal digits). However, the input function for JSONB is stricter. When the database encoding is not UTF8, it does not allow Unicode escapes for non-ASCII characters (U+007F and above). In addition, in JSONB types, use of a Unicode surrogate pair that denies `\u0000` (as it cannot be represented as a text type) and specifies characters out of the Unicode Basic Multilingual Plane (BMT) must be correct; it is converted to ASCII or UTF8 characters corresponding to a valid Unicode escape and stored (overlapping surrogate pairs are included as a single character).

Note: Many JSON processing functions convert Unicode escapes to regular characters, and thus return an error even if the input is of type json rather than jsonb. In general, it is best not to mix Unicode escapes with UTF8 database encodings in JSON whenever possible.

If you convert the text JSON input to JSONB, the primitive types described in RFC 7159 are effectively mapped to the native AgensGraph types, as shown in Table 8-23. Accordingly, several local constraints that do not apply to the JSON type and JSON (even abstractly) and that constitute valid JSONB data are added; this corresponds to a restriction on whether it can be represented as a primitive data type or not. In particular, JSONB rejects numbers that are outside the range of the AgensGraph numeric data type and that do not leave JSON. Constraints on which the corresponding implementations are defined are allowed by RFC 7159. In practice, however, this problem is much more common in other implementations, as JSON's number base type is commonly used to represent IEEE 754 double floating point (explicitly predicted and allowed in RFC 7159). If you use JSON in interchange format with the system, you should consider the risk of loss of numeric precision when compared to the original data stored in AgensGraph. Conversely, there are some local constraints on the input type of the JSON base type that do not apply to the corresponding AgensGraph type, as shown in the table.

[JSON basic type and corresponding AgensGraph type]

JSON primitive type	Type	Notes
string	text	If the database encoding is not UTF8, 000 is not allowed as it is a non-ASCII Unicode escape
number	numeric	NaN and infinity values are not allowed
boolean	boolean	Only lowercase spelling true and false are accepted
null	(없음)	SQL NULL is conceptually different

JSON Input and Output Syntax

The input/output syntaxes in the JSON data type can be specified as in RFC 7159. The following example shows all valid JSON (or JSONB) expressions.

```
-- Simple scalar/primitive value
-- Primitive values can be numbers, quoted strings, true, false, or null
SELECT '5'::json;

-- Array of zero or more elements (elements need not be of same type)
SELECT '[1, 2, "foo", null]'::json;

-- Object containing pairs of keys and values
-- Note that object keys must always be quoted strings
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Arrays and objects can be nested arbitrarily
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

As mentioned earlier, when a JSON value is input and printed without further processing, JSON outputs the same text as the input, and JSONB does not retain syntactically-insignificant content (e.g. spaces). Refer to the differences presented below.

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                      json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
```

```
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                      jsonb
--------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

One thing to note, though not syntactically-significant, is that it is in JSONB, where numbers are printed according to the default numeric type. In practice, this means that the number marked as "e" will be omitted when printed. Here is an example:

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+-------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

As shown in this example, however, JSONB preserves the trailing zero (0) for checkup even if it is syntactically-insignificant.

Designing JSON documents effectively

Expressing data in JSON can be much more flexible than that in traditional relational data models where requirements are enforced in a variable environment. Both methods may coexist and complement each other in the same application. However, in applications that require maximum flexibility, it is recommended that the JSON documents have a slightly fixed structure. Structures are not generally applicable (you can also apply some business rules declaratively), but using a predictable structure makes it easier to write queries that usefully summarize a "document" (datum) set of tables. JSON data is affected by the same concurrency control considerations as other data types when stored in tables. It should be noted that, even though it is feasible to store a large document, updates obtain row-level locking for the entire row. You should consider limiting the JSON document to a manageable size in order to reduce lock contention between update transactions. In principle, JSON documents indicate that the atomic data pointed by business rules cannot be segmented into smaller datums, each of which can be modified independently.

JSONB Containment and Existence

Testing containment is an important feature of JSONB. There is no feature set similar to the JSON type. Containment tests whether a single JSONB document is contained in another document. This example returns true except where noted.

```
-- Simple scalar/primitive values contain only the identical value:
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;
```

```
-- The array on the right side is contained within the one on the left:
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- Order of array elements is not significant, so this is also true:
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- Duplicate array elements don't matter either:
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- The object with a single pair on the right side is contained
-- within the object on the left side:
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb
       @> '{"version": 9.4}'::jsonb;

-- The array on the right side is not considered contained within the
-- array on the left, even though a similar array is nested within it:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- yields false

-- But with a layer of nesting, it is contained:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- Similarly, containment is not reported here:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;  -- yield
s false

-- A top-level key and an empty object is contained:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

The general principle is that the structure and data content of the contained objects are consistent with the containing objects. It may be possible after discarding unmatched (inconsistent) array elements or object key/value pairs from the containing object. However, when comparing containments, the order of array elements is not important, and duplicate array elements are actually considered only once.

As a special exception to the general principle that structures must be matched, arrays may contain primitive values.

```
-- This array contains the primitive string value:
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- This exception is not reciprocal -- non-containment is reported here:
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- yields false
```

JSONB has the existence operator that is a variation of the containing theme, which tests whether a string (specified as a text value) appears at the top level of the JSONB value as an object key or array element. This example returns true, except where noted.

```
-- String exists as array element:
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- String exists as object key:
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- Object values are not considered:
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- yields false

-- As with containment, existence must match at the top level:
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- yields false

-- A string is considered to exist if it matches a primitive JSON string:
SELECT '"foo"'::jsonb ? 'foo';
```

Unlike arrays, JSON objects are optimized internally for search and do not perform linear searches. This means that they are more suitable than arrays that test for containment or existence when there are many related keys and elements.

JSONB Indexing

The GIN index can be used to efficiently search for a key or a key/value pair in a number of JSONB documents (datums). Two GIN "operator classes" with different performance and flexibility tradeoffs are provided. The default GIN operator classes for JSONB support queries using operators such as @>, ?, ?&, and ?|. Here is an example of creating an index within this operator class:

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

jsonb_path_ops, which is not a default GIN operator class, supports indexing of @> operator only. Here is an example of creating an index within this operator class:

```
CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

Consider a table example that stores a JSON document retrieved from a third-party Web service using a documented schema definition. The general document will be as follows:

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

Save this document as a JSONB column called jdoc in a table called api. When a GIN index is created in this column, the following query uses the index.

```
-- Find documents in which the key "company" has value "Magnafone"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnaf
one"}';
```

However, even though the operator "?" can be indexed, it is not directly applied to the indexed column jdoc. Thus, the index cannot be used in the following query.

```
-- Find documents in which the key "tags" contains key or array element "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

Still, the above query can use the index, if an expression index is properly used. If a query for a particular item is common within the "tags" key, an index definition like the one below is useful.

```
CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));
```

WHERE clause jdoc->'tags' ? 'qui' is an application of the indexable operator "?" and the indexed expression jdoc -> 'tags' is recognized.

Another way to query is to make the best use of containment. For example, a simple GIN index on a jdoc column can support this query.

```
-- Find documents in which the key "tags" contains array element "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

However, while these indexes store copies of all the keys and values of the jdoc column, the expression index in the previous example stores only the data found under the tags key. It is true that the simple index approach is much more flexible (as it supports queries on arbitrary keys), but the targeted expression index is much smaller and more fast-searched than the simple index.

The jsonb_path_ops operator class only supports queries using @> operator, but it has a good performance advantage over the default operator class, jsonb_ops. With the same data, the size and search specificity of the jsonb_path_ops index are generally much smaller and better than those of the jsonb_ops index, respectively. This is especially true for queries that contain keys that appear frequently in the data. Searches with this class are therefore generally much better than using the default operator class.

A technical difference between jsonb_ops and jsonb_path_ops GIN indexes is that the former creates index entries that are independent for each key and value of the data, while the latter only creates index entries for the data values. By default, each jsonb_path_ops index entry is a hash of values and keys leading to it. Let's take a look at an index {"foo": {"bar": "baz"} as an example; a single index entry is generated by combining all three of foo, bar and baz into a hash value. Thus, a constraint query that looks for such a structure results in an extremely specific index search. However, there is no way to know at all whether foo will appear as a key. Conversely, the jsonb_ops index creates three index entries that represent foo, bar, and baz, respectively. Then, to execute a constraint query, it looks up a row that contains all three of these items. A GIN index can perform its AND search very efficiently, but is less specific and slower than an equivalent jsonb_path_ops search, especially when there are a great deal of rows containing one of the three index entries. A disadvantage of the jsonb_path_ops approach is that it does not create index entries for JSON structures that do not contain values such as {"a": {}}. A full index scan is required if a document search containing the structure is requested, which is very slow. Thus, jsonb_path_ops is not suitable for applications that perform frequent searches. JSONB also supports btree and hash indexes, which is useful only when checking the equivalence of the entire JSON document. The B-tree ordering of JSONB data is not that important, but is necessary for completeness.

```
Object > Array > Boolean > Number > String > Null

Object with n pairs > object with n - 1 pairs

Array with n elements > array with n - 1 elements
```

Objects that have the same number of pairs are compared in the following order:

```
key-1, value-1, key-2 ...
```

Object keys are compared in the order of storage; in particular, storing short keys in front of long keys leads to non-intuitive results:

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

Similarly, arrays with the same number of elements are compared in the following order:

```
element-1, element-2 ...
```

The native JSON values are compared using the comparison rules, which also apply to the primitive data types. Strings are compared using the default database collation.

Arrays

Arrays allow the column type of the table to be defined as a variable length multidimensional array. You can create arrays of built-in or user-defined base types, enum types, or composite types. Arrays of domains are not yet supported.

Declaration of Array Types

In order to explain use of array types, let's create a table as follows:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

As indicated, the array data type is named by adding square brackets ([]) to the array element's data type name. The above command creates a table called sal_emp with a column of type text (name), where a one-dimensional array of type integer (pay_by_quarter) represents the employee's quarterly salary and a two-dimensional array of text (schedule) indicates the employee's weekly schedule. CREATE TABLE allows the exact size of the array to be specified. For example:

```
CREATE TABLE tictactoe (
squares integer[3][3]
);
```

However, the current implementation ignores the array size limits provided. That is, the operation is the same as an array of unspecified length. The current implementation does not enforce the declared number of dimensions. Arrays of particular element types are considered to be of the same type, regardless of their size or the number of dimensions. Therefore, declaring an array size or the number of dimensions in CREATE TABLE is simply documentation and does not affect runtime behavior.

You can utilize the keyword ARRAY to use an alternative syntax that conforms to the SQL standard for one-dimensional arrays. pay_by_quarter may have been defined as:

```
pay_by_quarter  integer ARRAY[4],
```

If the array size is not specified:

```
pay_by_quarter  integer ARRAY,
```

In any case, however, size constraints are not enforced.

Array Value Input

To create an array value as a literal constant, place the value in braces and separate it with a comma. You can use double quotes around the value; you should do this if it contains a comma or brace. The general form of an array constant:

```
'{ val1 delim val2 delim ... }'
```

In the example above, delim is the delimiter for the type recorded in the pg_type entry. All of the standard data types provided by the AgensGraph distribution use commas, except for the type box that uses semicolons (;). Each val is a constant of the array element type or subarray. For example:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

This constant is a two-dimensional 3x3 array consisting of three integer sub-arrays. To set an element of an array constant to NULL, create NULL for the element value. (NULL case can be changed.) If you want the actual string value "NULL," use double quotation marks before and after NULL. (Constants are initially processed as strings and passed to the array input conversion routine, which may require explicit type specifications.)

The following is an example of INSERT statement and its execution result:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');

SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                  schedule
-------+---------------------------+-----------------------------------------
--
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

A multidimensional array must have a matching range for each dimension, and an inconsistency will cause the following error:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  multidimensional arrays must have array expressions with matching dim
ensions
```

The constructor syntax of ARRAY can also be used.

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

An array element is a regular SQL constant or expression. A string literal is enclosed in single quotes instead of double quotes as in an array literal.

Accessing Arrays

You can run some queries from the table created above. Access a single element of an array. The following query retrieves the name of the employee whose salary has been changed in the second quarter.

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

Array numbers are enclosed in square brackets; use array notation starting from 1. That is, the array of n elements starts at array[1] and ends at array[n].

This query retrieves the salary of all employees for the third quarter.

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

You can also access any rectangular slice in an array or subarray. An array slice is represented using lower-bound:upper-bound for one or more array dimensions.

For example, the following query retrieves the first item in Bill's schedule on the first two days of the week.

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

If you create a dimension as a slice (for example, including a colon), all dimensions are processed as slices. A dimension with only a single number (no colon) is processed as being from 1 to the specified number. For example, [2] is processed as [1: 2], as follows:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

                 schedule
-------------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

To avoid confusion in non-slice cases, it is best to use the slice syntax for all dimensions such as [1:2] [1:1] rather than [2] [1:1].

You can omit the lower-bound or upper-bound of the slice specifier. The missing bounds are replaced by lower or upper of the array, as shown in the following example:

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{lunch},{presentation}}
(1 row)

SELECT schedule[:][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

The array subscript expression returns null if the array itself or subscript expression is null. In addition, null is returned if the subscript is out of the bounds of the array (in which case no errors are produced). For example, schedule [3][3], which is referenced when schedule is the current dimension [1:3] [1:2], prints NULL. Similarly, an array referencing an incorrect number of subscripts prints null, not an error.

Array slice expressions similarly return null if the array itself or the subscript expression is null. However, in other cases (e.g. selecting an array slice that is completely out of the bounds of the current array), the slice expression outputs an empty (zero-dimensional) array instead of null. If the requested slice partially overlaps the array bounds, it is reduced to an overlap region instead of returning null.

The current dimensions of array values can be retrieved using array_dims function.

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

array_dims produces a text result, which is more readable to humans, but inconvenient to the program. Dimensions can also be retrieved using array_upper and array_lower, which return the upper and lower bounds of the specified array dimension, respectively.

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

array_length returns the length of the specified array dimension.

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

cardinality returns the total number of elements in an array of all dimensions; it is actually the number of rows generated by the call on unnest.

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';

 cardinality
-------------
```

```
           4
(1 row)
```

Modifying Arrays

You may update the array value as a whole;

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

or use ARRAY expression syntax.

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

An array can update a single element;

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

or update it only partially.

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

The omitted lower-bound or upper-bound slice syntax can be used only when updating NULL or nonzero array values.

The stored array value can be expanded by assigning it to an element that does not yet exist. The position between the previously existing element and the newly allocated element is filled with null. For example, if the array myarray currently has 4 elements, it will have 6 elements after an update of assigning to myarray [6]. myarray [5] contains null. Currently, the expansion in this manner is only allowed for one-dimensional arrays (not multidimensional arrays). Subscripted assignments allow you to create arrays whose subscripts do not start at one (1). For example, to create an array with subscript values -2 through 7, you may assign it to myarray [-2:7].

The new array value can also be created using the concatenation operator ||.

```
SELECT ARRAY[1,2] || ARRAY[3,4] as array;
   array
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]] as array;
        array
---------------------
```

```
 {{5,6},{1,2},{3,4}}
(1 row)
```

The concatenation operator lets you push a single element into the beginning or end of a one-dimensional array. It accommodates two N-dimensional arrays (or N-dimensional and N+1-dimensional arrays).

If you push a single element to the beginning or end of a one-dimensional array as shown in the following example, the result is an array filled with the same lower bound subscript as the array operand.

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

When concatenating two arrays having the same number of dimensions, the result holds the lower bound of the outer dimension of the left operand. The result is that all the elements of the right operand are followed by an array of all the elements of the left operand, as the following example shows:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

If you push an N-dimensional array from the beginning to the end of an N+1-dimensional array, the result is similar to the element array example above. Each N-dimensional subarray is essentially an element of the outer dimension of the N+1-dimensional array, as shown in the following example:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

You can also create functions using the functions array_prepend, array_append, or array_cat; while the first two only support one-dimensional arrays, array_cat supports multidimensional arrays. The concatenation operator discussed above prefers the direct use of these functions. In effect, these functions exist primarily for use when implementing a concatenation operator. However, they can also be useful when creating user-defined aggregates. For example:

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

In a simple case, the concatenation operator described above is preferred over using these functions directly. However, using one of these functions may help avoid ambiguity, since the concatenation operator is overloaded to process all three cases.

```
SELECT ARRAY[1, 2] || '{3, 4}' as array;  -- the untyped literal is taken as
an array
   array
-----------
 {1,2,3,4}

SELECT ARRAY[1, 2] || '7';                  -- so is this one
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL as array;         -- so is an undecorated NULL
 array
-------
 {1,2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL);    -- this might have been meant
 array_append
--------------
 {1,2,NULL}
```

In the above example, parser sees an integer array on one side of the concatenation operator, and a constant of indeterminate type on the other. An empirical method used to analyze the constant type is to assume it is the same type as the other inputs of the operator (in this case, an integer array). So the concatenation operator is considered array_cat, not array_append. If that is a wrong choice, you can modify the constant by casting it to an element type of the array. Using array_append can be a desirable solution.

Searching in Arrays

To retrieve the value of an array, you should check each value; you may do this if you know the size of the array. For example:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

However, in large arrays, it quickly gets bored and is not helpful when the array size is unknown. The above query can be replaced by:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

You can also find all the rows with an array value equal to 10000:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

Alternatively, you can use the generate_subscripts function.

```
SELECT * FROM
   (SELECT pay_by_quarter,
           generate_subscripts(pay_by_quarter, 1) AS s
      FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

It is also possible to search for an array using && operator, which checks whether the left operand overlaps with the right operand.

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

You can also use array_position and array_positions functions to retrieve a specific value in an array. The former returns the subscript of the first value in the array, and the latter returns an array with all the subscripts of the values in the array.

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon
');
 array_positions
-----------------
 2

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
```

Array Input and Output Syntax

The external text expression of an array value consists of the I/O conversion rules for the array element type and the items that are interpreted according to the decoration denoting the array structure. Decoration consists of braces at both ends of the value of an array and delimiters between adjacent items. The delimiters are usually commas (,), but can be something else. It is determined by the typedelim setting for the element type of the array. All of the standard data types use commas, except for type box that uses semicolons (;). In a multidimensional array, each dimension (row, plane, cube, etc.) has its own level of braces and delimiters must be used between adjacent levels of brace entities.

Array output routines use double quotes around element values if they are empty strings, contain braces, delimiters, double quotes, backslashes, or whitespace, or match the word NULL. Double quotes and backslashes embedded in element values are escaped by a backslash. In the case of numeric data types, it is safe to assume that double quotes never appear. However, for text data types you must cope with presence or absence of quotation marks. By default, the lower bound index value is set to 1 in the dimension of the array. If you specify an array using another lower bound, the range of array subscripts can be explicitly specified before creating the array content. This type of decoration consists of brackets ([]) before and after the lower/upper limits of each array dimension and colons (:) as space separator s. An array dimension decoration is followed by an equal sign (=).

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

The array output routine includes an explicit dimension in the result only if there is more than one lower bound.

If the value created for an element is NULL (in the case of variation), the element is considered to be NULL. Presence of quotes or backslashes disables this and allows input of the literal string value "NULL." You may also set the array_nulls configuration parameter to off to prevent NULL from being recognized as NULL. As indicated earlier, you can use double quotes around individual array elements when creating array values. This should be done in the case where the element values can be confused by the array value parser. For example, elements that contain braces, commas (or data type separators), double quotes, backslashes, and leading or trailing whitespace must use double quotes. Empty strings and strings that match NULL must also be quoted. To add double quotes or backslashes into the quoted array element values, use an escape string syntax and precede it with a backslash. You may also use backslash escaping to avoid quotation marks and protect all data characters that might be processed incorrectly as array syntax.

You may add a space before the left brace or after the right brace. You can also add a space before or after the individual item string. Spaces are ignored in all these cases. Whitespaces within double-quote elements and spaces at both ends of non-whitespace characters in elements are ignored.

Range Types

The range type is a data type that represents the range of values of some element type. For example, the range of timestamps can be used to indicate the reserved time range of a meeting room. In this case, the data type is tsrange (short for "timestamp range") and timestamp is subtype. The subtype must have a total order so that whether the element value is within, before, or after the value range can be well-defined.

The range types are useful in that they can represent multiple element values as a single range value and clearly express concepts like the overlap range. One of the most obvious examples is to use time and date ranges for scheduling. These types can also be useful for price ranges, measuring ranges of instruments, etc.

Built-in Range Types

The following built-in range types are provided:

int4range - Range of integer

int8range - Range of bigint

numrange - Range of numeric

tsrange - Range of timestamp without time zone

tstzrange - Range of timestamp with time zone

daterange - Range of date

You can also define your own range types. See User-defined Type for more information.

Examples

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- Containment
SELECT int4range(10, 20) @> 3;

-- Overlaps
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extract the upper bound
SELECT upper(int8range(15, 25));

-- Compute the intersection
SELECT int4range(10, 20) * int4range(15, 25);

-- Is the range empty?
SELECT isempty(numrange(1, 5));
```

Inclusive and Exclusive Bounds

All non-empty ranges have two bounds, a lower bound and an upper bound. All points between these values are included in the range. Inclusive bounds mean that the boundary points themselves are included in the range, and exclusive bounds mean that the boundary points are not included in the range. In the text form of the range, the inclusive lower bound is expressed as "[" and the exclusive lower bound is expressed as "(." Similarly, the inclusive upper bound is expressed as "]", and the exclusive upper bound is expressed as ")."

The functions lower_inc and upper_inc test the lower and upper bounds of the range value, respectively.

Infinite (Unbounded) Ranges

The lower bound of the range can be omitted, meaning that all points below the upper bound are included in the range. Likewise, if the upper bound of the range is omitted, all points above the lower bound are included in the range. If both the lower and upper bounds are omitted, all values of the element type are considered to be included in the range.

This corresponds to considering that the lower bound is "negative infinity" or the upper bound is "positive infinity." Note, however, that this infinite value is never a value of the element type of the range, and cannot be part of the range. (Therefore, there is no such thing as inclusive infinite bounds; when you try to create one, it is automatically converted to an exclusive bound).

It is true that some element types have an "infinite" notation, but it is another different value in relation to the range type mechanism. For example, in the timestamp range, [today,] is the same as [today,). However, [today, infinity] can be sometimes different from [today, infinity). The latter excludes the special timestamp value infinity.

The functions lower_inf and upper_inf test infinite lower/upper bounds of each range, respectively.

Range Input/Output

The input for the range value should follow one of the following patterns:

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

The parentheses or square brackets indicate whether the lower and upper bounds are excluded or included as described above. The last pattern is empty, indicating an empty range (a range with no points). The lower-bound can be a string that is a valid input to a subtype, or can be left empty if there is no lower bound. Similarly, upper-bound can be a string that is a valid input to a subtype, or can be left empty if there is no upper bound. Each boundary value can be quoted using "(double quote) characters; this is a must since, if the boundary value contains parentheses, square brackets, commas, double quotes, or backslashes, these characters may be mistaken for part of the range syntax. To insert double quotes or backslashes to quoted boundary values, you must precede them with a backslash. (Double quotation pairs within boundary values in double quotes are also processed as double quotation marks, similar to the rules for single quotation marks in SQL literal strings.) To protect all data characters that might be processed incorrectly with the range syntax, you may avoid quoting and use backslash escaping. In addition, when creating a boundary value that is an empty string, you should write ""; if you do not enter anything, it will mean infinite boundary. Whitespaces are allowed before and after the range values, but spaces between parentheses or square brackets are considered to be part of the lower or upper bound value. (It may or may not be important depending on the element type).

Examples:

```
-- includes 3, does not include 7, and does include all points in between
SELECT '[3,7)'::int4range;

-- does not include either 3 or 7, but includes all points in between
SELECT '(3,7)'::int4range;

-- includes only the single point 4
SELECT '[4,4]'::int4range;

-- includes no points (and will be normalized to 'empty')
SELECT '[4,4)'::int4range;
```

Constructing Ranges

Each range type has a constructor function with the same name as the range type. Using a constructor function is more convenient than writing a range literal constant, because you do not need to quote additional boundary values. A constructor function accepts two or three arguments. While the three-argument form creates a range from the third argument to the boundary of the form specified, the two-argument form creates a range of standard forms (lower bound, excluding upper bound). The third argument must be one of the followings strings: "()", "(]", "[]" or "[]"

Examples:

```
-- The full form is: lower bound, upper bound, and text argument indicating
-- inclusivity/exclusivity of bounds.
SELECT numrange(1.0, 14.0, '(]');
-- If the third argument is omitted, '[)' is assumed.
SELECT numrange(1.0, 14.0);

-- Although '(]' is specified here, on display the value will be converted to
-- canonical form, since int8range is a discrete range type (see below).
SELECT int8range(1, 14, '(]');

-- Using NULL for either bound causes the range to be unbounded on that side.
SELECT numrange(NULL, 2.2);
```

Discrete Range Types

The discrete range is a well-defined "step-by-step" type of which element type is integer or date. In this type, if there is no valid value between two elements, they can be said to be adjacent. In contrast to the continuous range, this is always (or almost always) possible to recognize different element values between the two given values. For example, a range beyond the numeric type, like the range beyond timestamp, is continuous. (timestamp can be processed discretely in theory because of its precision limitations, but it is better to regard it as a sequence when the size of the step is not of interest.) Another way to think about the discrete range type is to have a clear idea of the "next" or "previous" value for each element value. It should be noted that, by selecting the next or previous element value rather than the given original, it is possible to convert between expressions including the boundaries of the ranges and expressions excluding the boundaries of the ranges. For example, integer range types [4,8] and (3,9) denote the same set of values. This is not the case, however, for numerical ranges. The discrete range type must have a canonicalization function that recognizes the desired step size for the element type. The canonicalization function is especially responsible for converting the values equally to the range types that have an identity representation of a consistent inclusion or exclusion range. Unless a canonicalization function is specified, the ranges of different types are always processed as non-equivalence, even though they actually denote the same set of values. The built-in range types, int4range, int8range, and daterange, use the canonical form, which includes the lower bound and excludes the upper bound (i.e."[)"). User defined range types may use other notations.

Defining New Range Types

You may define your own range type. The most common reason for doing this is to use a subtype range that is not provided as a built-in range type. This is an example of defining a new range type for the subtype float8.

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);
```

```
SELECT '[1.234, 5.678]'::floatrange;
```

Since float8 is not a meaningful "step", we do not define a canonicalization function in this example. If the subtype has a discrete value rather than a continuous value, CREATE TYPE command must specify a canonical function. The canonicalization function must take an input range value and return an equivalent range value that may be different from the boundary type. The canonical outputs of the two ranges representing the same set of values (e.g. integer range [1, 7] and [1, 8)) should be the same. It does not matter which expression is chosen to be canonical, as long as two equivalent values of the same type are always mapped to the same value of the same type. Besides controlling inclusive/exclusive bounds format, the canonicalization function can process boundary values well if the desired step size is greater than the subtype's storable size. For example, the step size of a range type beyond timestamp can be defined as time. On this occasion, the canonicalization function may require rounding-off if it is not a multiple of the time, or an error may occur instead. If you define your own range type, you may specify different subtype B-tree operator classes or collations to use in order to change the sort order that determines which values belong to a given range. In addition, a range type to be used in a GiST or SP-GiST index should define a subtype difference or subtype_diff, a function(An index works without subtype_diff, but is much less efficient when a difference function is provided). The subtype difference function takes two input values of a subtype and returns the difference expressed as a float8 value (e.g. X minus Y). In the above example, you may use a function that underlies the normal float8 subtraction operator, but for other subtypes, type conversions may be required. Several creative ideas about how to represent differences in numbers may be needed. To the greatest range possible, the subtype_diff function must agree on the sort order implied by the selected operator class es and collations. That is, the result of this should always be a positive value when the first argument is greater than the second argument according to the sort order.

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

For more information on creating range types, see User-defined Type.

Indexing

GiST and SP-GiST indexes are able to generate table columns of range type. The following is an example of creating a GiST index.

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

GiST or SP-GiST indexes can speed up queries involving range operators such as =, &&, <@, @>, <<, >>, -|-, &<, and &>.

B-tree and hash indexes may create table columns of range type as well. For these index types, the only useful range operation is basically "=". Even if there is a B-tree sort order that uses "<" and ">" operators and is defined for range values, the order itself is arbitrary and not very useful in the real world. The B-tree and hash support for range types is intended primarily to allow sorting and hashing inside queries rather than creating actual indexes.

Constraints on Ranges

UNIQUE is a natural constraint on scalar values. However, it is appropriate not for range types but for exclusion constraints, mainly. An exclusion constraint allows the specification of constraints such as "nonoverlap" in range types.

Examples:

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

The constraint prevents overlapping values from being simultaneously present in the table.

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
ERROR:  conflicting key value violates exclusion constraint "reservation_duri
ng_excl"
DETAIL:  Key (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00")) conflic
ts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01 15:00:00")).
```

You can use the btree_gist extension to define an exclusion constraint on a regular scalar data type, which makes it possible to exclude/combine the range with the maximum flexibility. For instance, after btree_gist is installed, the following constraint rejects overlapping ranges only when the number of meeting rooms is equal.

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint "room_reservation
_room_during_excl"
DETAIL:  Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01 15:30:0
0")) conflicts
with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01 15:
00:00")).

INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

User-defined Type

You can add a new type using CREATE TYPE command. There are five types of CREATE TYPE: Composite Type, Enum Type, Range Type, Base Type, and Shell Type.

Syntex :

```
CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name
```

Examples :

This is an example of creating a composite type and using it for function definition.

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

This is an example of creating an enum type and using it for table definition.

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

This is an example of creating a range type.

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

This is an example of creating a base type and then using it for table definition.

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

functions

Comparison functions

- num_nonnulls(VARIADIC "any")
  Returns the number of non-null arguments.

```
    SELECT num_nonnulls(1, NULL, 2);

    Result:
    num_nonnulls
    ------------
               2
         (1 row)
```

- num_nulls(VARIADIC "any")
  Returns the number of null arguments.

```
    SELECT num_nulls(1, NULL, 2);

    Result:
    num_nonnulls
    ------------
```

```
             1
        (1 row)
```

Mathematics functions

It provides various functions related to numbers, and the argument specified as dp indicates double precision.

- abs(x)
  Returns absolute value of the argument.

```
SELECT abs(-17.4);

Result:
 abs
-----
17.4
(1 row)
```

- cbrt(dp)
  Returns cube root of the argument.

```
SELECT cbrt(27.0);

Result:
 cbrt
------
    3
(1 row)
```

- ceil(dp or numeric) or ceiling(dp or numeric)
  Returns nearest integer greater than or equal to argument.

```
SELECT ceil(-42.8);

Result:
 ceil
------
  -42
(1 row)
```

- degrees(dp)
  Converts radians to degrees.

```
SELECT degrees(0.5);
```

```
Result:
     degrees
------------------
 28.6478897565412
           (1 row)
```

- div(y numeric, x numeric)
  Returns integer quotient of y/x.

```
SELECT div(9,4);

Result:
 div
-----
   2
(1 row)
```

- exp(dp or numeric)
  Returns exponential.

```
SELECT exp(1.0);

Result:
        exp
--------------------
 2.7182818284590452
             (1 row)
```

- floor(dp or numeric)
  Returns nearest integer less than or equal to the argument.

```
SELECT floor(-42.8);

Result:
 floor
-------
   -43
(1 row)
```

- ln(dp or numeric)
  Returns natural logarithm of the argument.

```
SELECT ln(2.0);

Result:
        ln
```

```
-------------------
 0.693147180559945
            (1 row)
```

- log(dp or numeric)
  Returns base 10 logarithm.

```
SELECT log(100.0);

Result:
        log
--------------------
 2.0000000000000000
             (1 row)
```

- log(b numeric, x numeric)
  Returns logarithm to base b.

```
SELECT log(2.0, 64.0);

Result:
        log
--------------------
 6.0000000000000000
             (1 row)
```

- mod(y, x)
  Returns remainder of y/x.

```
SELECT mod(9,4);

Result:
 mod
-----
   1
(1 row)
```

- pi()
  Returns "π" constant.

```
SELECT pi();

Result:
        pi
------------------
```

```
 3.14159265358979
          (1 row)
```

- power(a dp, b dp) or power(a numeric, b numeric)
  Returns a raised to the power of b.

```
SELECT power(9.0, 3.0);

Result:
        power
--------------------
 729.00000000000000
            (1 row)
```

- radians(dp)
  Converts degrees to radians.

```
SELECT radians(45.0);

Result:
      radians
-------------------
 0.785398163397448
           (1 row)
```

- round(dp or numeric)
  Rounds to nearest integer.

```
SELECT round(42.4);

Result:
 round
-------
    42
(1 row)
```

- round(v numeric, s int)
  Rounds to s decimal places of v argument.

```
SELECT round(42.4382, 2);

Result:
 round
-------
 42.44
(1 row)
```

- scale(numeric)
  Returns scale of the argument.

```
SELECT scale(8.41);

Result:
 scale
-------
     2
(1 row)
```

- sign(dp or numeric)
  Returns the sign (-1, 0, 1) of the argument.

```
SELECT sign(-8.4);

Result:
 sign
------
   -1
(1 row)
```

- sqrt(dp or numeric)
  Returns square root of the argument.

```
SELECT sqrt(2.0);

Result:
        sqrt
-------------------
 1.414213562373095
            (1 row)
```

- trunc(dp or numeric)
  Truncates toward zero.

```
SELECT trunc(42.8);

Result:
 trunc
-------
    42
(1 row)
```

- trunc(v numeric, s int)
  Truncates to s decimal places.

```
SELECT trunc(42.4382, 2);

Result:
 trunc
-------
 42.43
(1 row)
```

- width_bucket(operand dp, b1 dp, b2 dp, count int) or width_bucket(operand numeric, b1 numeric, b2 numeric, count int)
  Returns the bucket number to which operand would be assigned in a histogram having count equal-width buckets spanning the range b1 to b2; returns 0 or count+1 for an input outside the range.

```
SELECT width_bucket(5.35, 0.024, 10.06, 5);

Result:
 width_bucket
--------------
            3
       (1 row)
```

- width_bucket(operand anyelement, thresholds anyarray)
  Returns the bucket number to which operand would be assigned given an array listing the lower bounds of the buckets; returns 0 for an input less than the first lower bound; the thresholds array must be sorted, smallest first.

```
SELECT width_bucket(now(), array['yesterday', 'today', 'tomorrow']::times
tamptz[]);

Result:
 width_bucket
--------------
            2
       (1 row)
```

String functions

- ascii(*string*)
  ASCII code of the first character of the argument. For UTF8 returns the Unicode code point of the character. For other multibyte encodings, the argument must be an ASCII character.

```
SELECT ascii('x');
```

```
Result:
 ascii
-------
   120
(1 row)
```

- btrim(*string* text [, *characters* text])
  Removes the longest string consisting only of characters in *characters* (a space by default) from the start and end of *string*.

```
SELECT btrim('xyxtrimyyx', 'xyz');

Result:
 btrim
-------
  trim
(1 row)
```

- chr(int)
  Character with the given code. For UTF8 the argument is treated as a Unicode code point. For other multibyte encodings the argument must designate an ASCII character. The NULL (0) character is not allowed because text data types cannot store such bytes.

```
SELECT chr(65);

Result:
 chr
-----
   A
(1 row)
```

- concat(*str* "any" [, *str* "any" [, ...]])
  Concatenates the text representations of all the arguments. NULL arguments are ignored.

```
SELECT concat('abcde', 2, NULL, 22);

Result:
  concat
----------
 abcde222
   (1 row)
```

- concat_ws(*sep* text, *str* "any" [, *str* "any" [, ...]])
  Concatenates all but the first argument with separators. The first argument is used as the separator string. NULL arguments are ignored.

```
SELECT concat_ws(',', 'abcde', 2, NULL, 22);

Result:
  concat_ws
-------------
 abcde,2,22
     (1 row)
```

- convert(*string* bytea, *src_encoding* name, *dest_encoding* name)
  Converts string to *dest_encoding*. The original encoding is specified by *src_encoding*. The *string* must be valid in this encoding. Conversions can be defined by CREATE CONVERSION. Also there are some predefined conversions. See this link for available conversions.

```
SELECT convert('text_in_utf8', 'UTF8', 'LATIN1');

Result:
          convert
----------------------------
  x746578745f696e5f75746638
                    (1 row)
```

- convert_from(*string* bytea, *src_encoding* name)
  Converts string to the database encoding. The original encoding is specified by *src_encoding*. The *string* must be valid in this encoding.

```
SELECT convert_from('text_in_utf8', 'UTF8');

Result:
 convert_from
--------------
 text_in_utf8 (A string represented by the current database encoding)
       (1 row)
```

- convert_to(*string* text, *dest_encoding* name)
  Converts string to *dest_encoding*.

```
SELECT convert_to('some text', 'UTF8');

Result:
      convert_to
----------------------
  x736f6d652074657874
              (1 row)
```

- decode(*string* text, *format* text)
  Decodes binary data from textual representation in *string*. Options for *format* are same as in encode.

```
SELECT decode('MTIzAAE=', 'base64');

Result:
    decode
--------------
  x3132330001
       (1 row)
```

- encode(*data* bytea, *format* text)
  Encode binary data into a textual representation. Supported formats are: `base64`, `hex`, `escape`. `escape` converts zero bytes and high-bit-set bytes to octal sequences (`\`*nnn*) and doubles backslashes.

```
SELECT encode(E'123\\000\\001', 'base64');

Result:
  encode
----------
 MTIzAAE=
   (1 row)
```

- format(*formatstr* text [, *formatarg* "any" [, ...]])
  Formats arguments according to a format string. This function is similar to the C function `sprintf`.

```
SELECT format('Hello %s, %1$s', 'World');

Result:
       format
--------------------
 Hello World, World
             (1 row)
```

- initcap(*string*)
  Converts the first letter of each word to upper case and the rest to lower case. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.

```
SELECT initcap('hi THOMAS');

Result:
  initcap
```

```
-----------
 Hi Thomas
    (1 row)
```

- length(*string*)
  Returns the number of characters in *string*.

```
SELECT length('jose');

Result:
 length
--------
      4
 (1 row)
```

- length(*string* bytea, *encoding* name)
  Returns the number of characters in *string* in the given *encoding*. The *string* must be valid in this encoding.

```
SELECT length('jose', 'UTF8');

Result:
 length
--------
      4
 (1 row)
```

- lpad(*string* text, *length* int [, *fill* text])
  Fills up the *string* to length *length* by prepending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated (on the right).

```
SELECT lpad('hi', 5, 'xy');

Result:
 lpad
-------
 xyxhi
(1 row)
```

- ltrim(*string* text [, *characters* text])
  Removes the longest string containing only *characters* from characters (a space by default) from the start of *string*.

```
SELECT ltrim('zzzytest', 'xyz');

Result:
 ltrim
-------
  test
(1 row)
```

- md5(*string*)
  Calculates the MD5 hash of *string*, returning the result in hexadecimal.

```
SELECT md5('abc');

Result:
                md5
----------------------------------
 900150983cd24fb0d6963f7d28e17f72
                           (1 row)
```

- parse_ident(*qualified_identifier* text [, *strictmode* boolean DEFAULT true])
  Splits *qualified_identifier* into an array of identifiers, removing any quoting of individual identifiers. By default, extra characters after the last identifier are considered an error; but if the second parameter is `false`, then such extra characters are ignored. (This behavior is useful for parsing names for objects like functions). Note that this function does not truncate over-length identifiers. If you want truncation you can cast the result to `name[]`.

```
SELECT parse_ident('"SomeSchema".someTable');

Result:
       parse_ident
------------------------
 {SomeSchema,sometable}
                 (1 row)
```

- pg_client_encoding()
  Returns current client encoding name.

```
SELECT pg_client_encoding();

Result:
 pg_client_encoding
--------------------
     SQL_ASCII
             (1 row)
```

- quote_ident(*string* text)
  Returns the given string suitably quoted to be used as an identifier in an SQL

statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled.

```
SELECT quote_ident('Foo bar');

Result:
 quote_ident
-------------
   "Foo bar"
      (1 row)
```

- quote_literal(*string* text)
  Returns the given string suitably quoted to be used as a string literal in an SQL statement string. Embedded single-quotes and backslashes are properly doubled. Note that `quote_literal` returns null on null input; if the argument might be null, `quote_nullable` is often more suitable.

```
SELECT quote_literal(E'O\'Reilly');

Result:
 quote_literal
---------------
   'O''Reilly'
        (1 row)
```

- quote_literal(*value* anyelement)
  Coerces the given value to text and then quote it as a literal. Embedded single-quotes and backslashes are properly doubled.

```
SELECT quote_literal(42.5);

Result:
 quote_literal
---------------
        '42.5'
        (1 row)
```

- quote_nullable(*string* text)
  Returns the given string suitably quoted to be used as a string literal in an statement string; or, if the argument is null, return `NULL`. Embedded single-quotes and backslashes are properly doubled.

```
SELECT quote_nullable(NULL);

Result:
 quote_nullable
---------------
          NULL
        (1 row)
```

- quote_nullable(*value* anyelement)
  Coerces the given value to text and then quote it as a literal; or, if the argument is null, return `NULL`. Embedded single-quotes and backslashes are properly doubled.

```
SELECT quote_nullable(42.5);

Result:
 quote_nullable
----------------
         '42.5'
         (1 row)
```

- regexp_matches(*string* text, *pattern* text [, *flags* text])
  Returns captured substring(s) resulting from the first match of a POSIX regular expression to the *string*.

```
SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');

Result:
 regexp_matches
----------------
    {bar,beque}
         (1 row)
```

- regexp_replace(*string* text, *pattern* text, *replacement* text [, *flags* text])
  Replaces substring(s) matching a POSIX regular expression.

```
SELECT regexp_replace('Thomas', '.[mN]a.', 'M');

Result:
 regexp_replace
----------------
            ThM
         (1 row)
```

- regexp_split_to_array(string text, pattern text [, flags text])
  Splits *string* using a POSIX regular expression as the delimiter.

```
SELECT regexp_split_to_array('hello world', E'\\s+');

Result:
 regexp_split_to_array
-----------------------
        {hello,world}
               (1 row)
```

- regexp_split_to_table(*string* text, *pattern* text [, *flags* text])
  Splits *string* using a POSIX regular expression as the delimiter.

```
SELECT regexp_split_to_table('hello world', E'\\s+');

Result:
 regexp_split_to_array
-----------------------
                 hello
                 world
              (2 rows)
```

- repeat(*string* text, *number* int)
  Repeats *string* the specified *number* of times.

```
SELECT repeat('Pg', 4);

Result:
  repeat
----------
 PgPgPgPg
   (1 row)
```

- replace(*string* text, *from* text, *to* text)
  Replaces all occurrences in *string* of substring *from* with substring *to*.

```
SELECT replace('abcdefabcdef', 'cd', 'XX');

Result:
    replace
--------------
 abXXefabXXef
       (1 row)
```

- reverse(*str*)
  Return reversed string.

```
SELECT reverse('abcde');

Result:
 reverse
---------
   edcba
  (1 row)
```

- right(*str* text, *n* int)
  Returns last *n* characters in the string. When *n* is negative, it returns all but first |*n*| characters.

```
SELECT right('abcde', 2);

Result:
 right
-------
    de
(1 row)
```

- rpad(*string* text, *length* int [, *fill* text])
  Fills up the *string* to length *length* by appending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated.

```
SELECT rpad('hi', 5, 'xy');

Result:
 rpad
-------
 hixyx
(1 row)
```

- rtrim(*string* text [, *characters* text])
  Removes the longest string containing only characters from *characters* (a space by default) from the end of *string*.

```
SELECT rtrim('testxxzx', 'xyz');

Result:
 rtrim
-------
  test
(1 row)
```

- split_part(*string* text, *delimiter* text, *field* int)
  Splits *string* on *delimiter* and return the given field (counting from one).

```
SELECT split_part('abc~@~def~@~ghi', '~@~', 2);

Result:
 split_part
------------
       def
     (1 row)
```

- strpos(*string*, *substring*)
  Location of specified *substring* (same as *position* (*substring* in *string*), but note the reversed argument order).

```
SELECT strpos('high', 'ig');

Result:
 strpos
--------
      2
 (1 row)
```

- substr(*string*, *from* [, *count*])
  Extracts substring (same as `substring`(*string* from *from* for *count*)).

```
SELECT substr('alphabet', 3, 2);

Result:
 substr
--------
     ph
 (1 row)
```

- to_ascii(*string* text [, *encoding* text])
  Converts *string* to ASCII from another encoding (only supports conversion from `LATIN1`, `LATIN2`, `LATIN9`, and `WIN1250` encodings).

```
SELECT to_ascii('Karel');

Result:
 to_ascii
----------
    Karel
   (1 row)
```

- to_hex(*number* int or bigint)
  Converts *number* to its equivalent hexadecimal representation.

```
SELECT to_hex(2147483647);

Result:
  to_hex
----------
 7fffffff
   (1 row)
```

- translate(*string* text, *from* text, *to* text)
  Any character in *string* that matches a character in the *from* set is replaced by the corresponding character in the *to* set. If *from* is longer than *to*, occurrences of the extra characters in *from* are removed.

```
SELECT translate('12345', '143', 'ax');

Result:
 translate
-----------
      a2x5
    (1 row)
```

Binary String functions

Defines some string functions that use key words, rather than commas, to separate arguments.

- octet_length(string)
  Returns the number of bytes in binary string.

```
SELECT octet_length(E'jo\\000se'::bytea);

Result:
 octet_length
--------------
            5
       (1 row)
```

- overlay(string placing string from int [for int])
  Replaces substring.

```
SELECT overlay(E'Th\\000omas'::bytea placing E'\\002\\003'::bytea FROM 2
for 3);

Result:
     overlay
----------------
  x5402036d6173
         (1 row)
```

- position(substring in string)
  Returns the location of specified substring.

```
SELECT position(E'\\000om'::bytea in E'Th\\000omas'::bytea);

Result:
 position
----------
        3
   (1 row)
```

- substring(string [from int] [for int])
  Extracts substring.

```
SELECT substring(E'Th\\000omas'::bytea FROM 2 for 3);

Result:
 substring
-----------
  x68006f
    (1 row)
```

- trim([both] bytes from string)
  Removes the longest string containing only bytes appearing in bytes from the start and end of string.

```
SELECT trim(E'\\000\\001'::bytea FROM E'\\000Tom\\001'::bytea);

Result:
   trim
----------
  x546f6d
    (1 row)
```

- btrim(*string* bytea, *bytes* bytea)
  Removes the longest string containing only bytes appearing in *bytes* from the start and end of *string*.

```
SELECT btrim(E'\\000trim\\001'::bytea, E'\\000\\001'::bytea);

Result:
    btrim
------------
 x7472696d
     (1 row)
```

- decode(*string* text, *format* text)
  Decodes binary data from textual representation in *string*. Options for *format* are same as in encode.

```
SELECT decode(E'123\\000456', 'escape');

Result:
      decode
------------------
 x31323300343536
           (1 row)
```

- encode(*data* bytea, *format* text)
  Encodes binary data into a textual representation. Supported formats are: `base64`, `hex`, `escape`. `escape` converts zero bytes and high-bit-set bytes to octal sequences (`\`*nnn*) and doubles backslashes.

```
SELECT encode(E'123\\000456'::bytea, 'escape');

Result:
   encode
------------
 123\000456
     (1 row)
```

- get_bit(*string*, *offset*)
  Extracts bit from string.

```
SELECT get_bit(E'Th\\000omas'::bytea, 45);

Result:
 get_bit
---------
       1
  (1 row)
```

- get_byte(*string*, *offset*)
  Extract byte from string.

```
SELECT get_byte(E'Th\\000omas'::bytea, 4);

Result:
 get_byte
----------
      109
   (1 row)
```

- length(*string*)
  Returns the length of binary string.

```
SELECT length(E'jo\\000se'::bytea);

Result:
 length
--------
      5
 (1 row)
```

- md5(*string*)
  Calculates the MD5 hash of *string*, returning the result in hexadecimal.

```
SELECT md5(E'Th\\000omas'::bytea);

Result:
                md5
----------------------------------
 8ab2d3c9689aaf18b4958c334c82d8b1
                          (1 row)
```

- set_bit(*string*, *offset*, *newvalue*)
  Returns the set bit in string.

```
SELECT set_bit(E'Th\\000omas'::bytea, 45, 0);

Result:
      set_bit
------------------
  x5468006f6d4173
         (1 row)
```

- set_byte(*string*, *offset*, *newvalue*)
  Returns the set byte in string.

```
SELECT set_byte(E'Th\\000omas'::bytea, 4, 64);

Result:
     set_byte
------------------
  x5468006f406173
         (1 row)
```

Date Type Formatting functions

The formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, and numeric to formatted strings and for converting from formatted strings to specific data types. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a template that defines the output or input format.

- to_char(*timestamp*, *text*)
  Converts timestamp to string.

```
SELECT to_char(current_timestamp, 'HH12:MI:SS');

Result:
 to_char
----------
 04:56:02
   (1 row)
```

- to_char(*interval*, *text*)
  Converts interval to string.

```
SELECT to_char(interval '15h 2m 12s', 'HH24:MI:SS');

Result:
 to_char
----------
 15:02:12
   (1 row)
```

- to_char(*int*, *text*)
  Converts integer to string.

```
SELECT to_char(125, '999');

Result:
 to_char
----------
      125
   (1 row)
```

- to_char(*double* precision, *text*)
  Converts real/double precision to string.

```
SELECT to_char(125.8::real, '999D9');
```

```
Result:
 to_char
----------
    125.8
   (1 row)
```

- to_char(*numeric*, *text*)
  Converts numeric to string.

```
SELECT to_char(-125.8, '999D99S');

Result:
 to_char
----------
  125.80-
   (1 row)
```

- to_date(*text*, *text*)
  Converts string to date.

```
SELECT to_date('05 Dec 2000', 'DD Mon YYYY');

Result:
   to_date
------------
 2000-12-05
     (1 row)
```

- to_number(*text*, *text*)
  Converts string to numeric.

```
SELECT to_number('12,454.8-', '99G999D9S');

Result:
 to_number
-----------
  -12454.8
    (1 row)
```

- to_timestamp(*text*, *text*)
  Converts string to timestamp.

```
SELECT to_timestamp('05 Dec 2000', 'DD Mon YYYY');

Result:
      to_timestamp
```

```
    ------------------------
     2000-12-05 00:00:00+09
                    (1 row)
```

Date/Time functions

All the functions and operators described below that take time or timestamp inputs actually come in two variants: one that takes time with time zone or timestamp with time zone, and one that takes time without time zone or timestamp without time zone. For brevity, these variants are not shown separately. Also, + and * operators come in commutative pairs (for example both date + integer and integer + date); we show only one of each such pair.

- age(*timestamp*, *timestamp*)
  Subtracts arguments, producing a result that uses years and months, rather than just days.

```
    SELECT age(timestamp '2001-04-10', timestamp '1957-06-13');

    Result:
              age
    -------------------------
     43 years 9 mons 27 days
                     (1 row)
```

- age(*timestamp*)
  Subtracts from current_date (at midnight).

```
    SELECT age(timestamp '1957-06-13');

    Result:
              age
    -------------------------
     60 years 3 mons 15 days
                     (1 row)
```

- clock_timestamp()
  Returns current date and time (changes during statement execution).

```
    SELECT clock_timestamp();

    Result:
            clock_timestamp
    -------------------------------
     2017-09-28 17:47:31.208076+09
                           (1 row)
```

- current_date
  Returns current date.

```
SELECT current_date;

Result:
    date
------------
 2017-09-28
     (1 row)
```

- current_time
  Returns current time of day.

```
SELECT current_time;

Result:
       timetz
--------------------
 17:53:23.972231+09
             (1 row)
```

- current_timestamp
  Returns current date and time.

```
SELECT current_timestamp;

Result:
              now
-------------------------------
 2017-09-28 18:01:43.890977+09
                        (1 row)
```

- date_part(*text*, *timestamp*)
  Returns subfield specified in *text*.

```
SELECT date_part('hour', timestamp '2001-02-16 20:38:40');

Result:
 date_part
-----------
        20
    (1 row)
```

- date_part(*text*, *interval*)
  Returns subfield specified in *text*.

```
SELECT date_part('month', interval '2 years 3 months');

Result:
 date_part
-----------
         3
    (1 row)
```

- date_trunc(*text*, *timestamp*)
  Truncates to specified precision.

```
SELECT date_trunc('hour', timestamp '2001-02-16 20:38:40');

Result:
      date_trunc
---------------------
 2001-02-16 20:00:00
              (1 row)
```

- date_trunc(*text*, *interval*)
  Truncates to specified precision.

```
SELECT date_trunc('hour', interval '2 days 3 hours 40 minutes');

Result:
    date_trunc
-----------------
 2 days 03:00:00
          (1 row)
```

- extract(field from timestamp)
  Extracts the specified field.

```
SELECT extract(hour FROM timestamp '2001-02-16 20:38:40');

Result:
 date_part
-----------
        20
    (1 row)
```

- extract(field from interval)
  Extracts the specified field.

```
SELECT extract(month FROM interval '2 years 3 months');
```

```
Result:
 date_part
-----------
         3
    (1 row)
```

- isfinite(*date*)
  Returns the result of testing whether the input argument is finite (no +/- infinite).

```
SELECT isfinite(date '2001-02-16');

Result:
 isfinite
----------
        t
   (1 row)
```

- isfinite(*timestamp*)
  Returns the result of testing whether the input argument is finite (no +/- infinite).

```
SELECT isfinite(timestamp '2001-02-16 21:28:30');

Result:
 isfinite
----------
        t
   (1 row)
```

- isfinite(*interval*)
  Returns the result of testing whether the input argument is finite.

```
SELECT isfinite(interval '4 hours');

Result:
 isfinite
----------
        t
   (1 row)
```

- justify_days(*interval*)
  Adjusts interval so 30-day time periods are represented as months.

```
SELECT justify_days(interval '35 days');

Result:
 justify_days
```

```
--------------
 1 mon 5 days
       (1 row)
```

- justify_hours(*interval*)
  Adjusts interval so 24-hour time periods are represented as days.

```
SELECT justify_hours(interval '27 hours');

Result:
 justify_hours
----------------
 1 day 03:00:00
         (1 row)
```

- justify_interval(*interval*)
  Adjusts interval using justify_days and justify_hours, with additional sign adjustments.

```
SELECT justify_interval(interval '1 mon -1 hour');

Result:
 justify_interval
------------------
 29 days 23:00:00
           (1 row)
```

- localtime
  Returns current time of day.

```
SELECT localtime;

Result:
      time
----------------
 11:27:04.72722
         (1 row)
```

- localtimestamp
  Returns current date and time (start of current transaction).

```
SELECT localtimestamp;

Result:
          timestamp
----------------------------
```

```
 2017-09-29 11:29:52.230028
                    (1 row)
```

- make_date(*year* int, *month* int, *day* int)
  Creates date from year, month and day fields.

```
SELECT make_date(2013, 7, 15);

Result:
 make_date
------------
 2013-07-15
     (1 row)
```

- make_interval(*years* int DEFAULT 0, *months* int DEFAULT 0, *weeks* int DEFAULT 0, *days* int DEFAULT 0, *hours* int DEFAULT 0, *mins* int DEFAULT 0, *secs* double precision DEFAULT 0.0)
  Creates interval from years, months, weeks, days, hours, minutes and seconds fields.

```
SELECT make_interval(days => 10);

Result:
 make_interval
---------------
       10 days
        (1 row)
```

- make_time(*hour* int, *min* int, *sec* double precision)
  Creates time from hour, minute and seconds fields.

```
SELECT make_time(8, 15, 23.5);

Result:
 make_time
------------
 08:15:23.5
     (1 row)
```

- make_timestamp(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec* double precision)
  Creates timestamp from year, month, day, hour, minute and seconds fields.

```
SELECT make_timestamp(2013, 7, 15, 8, 15, 23.5);

Result:
    make_timestamp
-----------------------
```

```
 2013-07-15 08:15:23.5
               (1 row)
```

- make_timestamptz(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec* double precision, [*timezone* text])
  Creates timestamp with time zone from year, month, day, hour, minute and seconds fields; if timezone is not specified, the current time zone is used.

```
SELECT make_timestamptz(2013, 7, 15, 8, 15, 23.5);

Result:
      make_timestampz
--------------------------
 2013-07-15 08:15:23.5+01
                  (1 row)
```

- now()
  Returns current date and time (start of current transaction).

```
SELECT now();

Result:
              now
-------------------------------
 2017-10-11 16:09:51.154262+09
                       (1 row)
```

- statement_timestamp()
  Returns current date and time.

```
SELECT statement_timestamp();

Result:
      statement_timestamp
-------------------------------
 2017-10-11 16:08:59.641426+09
                       (1 row)
```

- timeofday()
  Returns current date and time.

```
SELECT timeofday();

Result:
              timeofday
-------------------------------------
```

```
 Wed Oct 11 16:09:26.934061 2017 KST
                             (1 row)
```

- transaction_timestamp()
  Returns current date and time.

```
SELECT transaction_timestamp();

Result:
     transaction_timestamp
-------------------------------
 2017-10-11 16:10:21.530521+09
                       (1 row)
```

- to_timestamp(*double* precision)
  Converts Unix epoch (seconds since 1970-01-01 00:00:00+00) to timestamp.

```
SELECT to_timestamp(1284352323);

Result:
      to_timestamp
------------------------
 2010-09-13 13:32:03+09
                (1 row)
```

Enum Support functions

For enum types, there are several functions that allow cleaner programming without hard-coding particular values of an enum type.

To execute the example in the function description, create an enum type as shown below first.

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'pur
ple');
```

- enum_first(*anyenum*)
  Returns the first value of the input enum type.

```
SELECT enum_first(null::rainbow);

Result:
 enum_first
------------
     red
     (1 row)
```

- enum_last(*anyenum*)
  Returns the last value of the input enum type.

```
SELECT enum_last(null::rainbow);

Result:
 enum_last
-----------
  purple
    (1 row)
```

- enum_range(*anyenum*)
  Returns all values of the input enum type in an ordered array.

```
SELECT enum_range(null::rainbow);

Result:
              enum_range
---------------------------------------
 {red,orange,yellow,green,blue,purple}
                                (1 row)
```

- enum_range(*anyenum*, *anyenum*)
  Returns the range between the two given enum values, as an ordered array. The values must be from the same enum type. If the first parameter is null, the result will start with the first value of the enum type. If the second parameter is null, the result will end with the last value of the enum type.

```
SELECT enum_range('orange'::rainbow, 'green'::rainbow);

Result:
      enum_range
-----------------------
 {orange,yellow,green}
                (1 row)

SELECT enum_range(NULL, 'green'::rainbow);
Result:
        enum_range
---------------------------
 {red,orange,yellow,green}
                    (1 row)

SELECT enum_range('orange'::rainbow, NULL);
Result:
```

```
            enum_range
-----------------------------------
 {orange,yellow,green,blue,purple}
                             (1 row)
```

Geometric Functions

- area(*object*)
  Returns area.

```
SELECT area(box '((0,0),(1,1))');

Result:
 area
------
    1
(1 row)
```

- center(*object*)
  Returns the center coordinates of object.

```
SELECT center(box '((0,0),(1,2))');

Result:
 center
--------
 (0.5,1)
 (1 row)
```

- diameter(*circle*)
  Returns the diameter of circle.

```
SELECT diameter(circle '((0,0),2.0)');

Result:
 diameter
----------
        4
   (1 row)
```

- height(*box*)
  Returns the vertical size of box.

```
SELECT height(box '((0,0),(1,1))');

Result:
 height
--------
      1
 (1 row)
```

- isclosed(*path*)
  Returns a logical value indicating whether the input path is a closed path.

```
SELECT isclosed(path '((0,0),(1,1),(2,0))');

Result:
 isclosed
----------
        t
  (1 row)
```

- isopen(*path*)
  Returns a logical value indicating whether the input path is an open path.

```
SELECT isopen(path '[(0,0),(1,1),(2,0)]');

Result:
 isopen
--------
      t
 (1 row)
```

- length(*object*)
  Returns length of the path.

```
SELECT length(path '((-1,0),(1,0))');

Result:
 length
--------
      4
 (1 row)
```

- npoints(*path*)
  Returns the number of points of the input path.

```
SELECT npoints(path '[(0,0),(1,1),(2,0)]');

Result:
 npoints
---------
```

```
       3
  (1 row)
```

- npoints(*polygon*)
  Returns the number of polygon points.

```
SELECT npoints(polygon '((1,1),(0,0))');

Result:
 npoints
---------
       2
  (1 row)
```

- pclose(*path*)
  Converts the input path to closed.

```
SELECT pclose(path '[(0,0),(1,1),(2,0)]');

Result:
       pclose
---------------------
 ((0,0),(1,1),(2,0))
              (1 row)
```

- popen(*path*)
  Converts the input path to open.

```
SELECT popen(path '((0,0),(1,1),(2,0))');

Result:
        popen
---------------------
 [(0,0),(1,1),(2,0)]
              (1 row)
```

- radius(*circle*)
  Returns the radius of circle.

```
SELECT radius(circle '((0,0),2.0)');

Result:
 radius
--------
      2
 (1 row)
```

- width(*box*)
  Returns the radius of circle.

```
SELECT width(box '((0,0),(1,1))');

Result:
 width
-------
     1
(1 row)
```

- box(*circle*)
  Returns a box circumscribed about circle.

```
SELECT box(circle '((0,0),2.0)');

Result:
                                    box


---------------------------------------------------------------------------
--
 (1.41421356237309,1.41421356237309),(-1.41421356237309,-1.41421356237309)
                                                                      (1 ro
w)
```

- box(*point*)
  Returns a box whose width is zero (empty box) centered on the input point.

```
SELECT box(point '(0,0)');

Result:
     box
-------------
 (0,0),(0,0)
      (1 row)
```

- box(*point, point*)
  Returns a box whose vertices are the two input points.

```
SELECT box(point '(0,0)', point '(1,1)');

    Result:
     box
-------------
 (1,1),(0,0)
      (1 row)
```

- box(*polygon*)
  Returns a box circumscribed about polygon.

```
SELECT box(polygon '((0,0),(1,1),(2,0))');

Result:
     box
-------------
 (2,1),(0,0)
      (1 row)
```

- bound_box(*box*, *box*)
  Returns the smallest box that contains the two boxes entered.

```
SELECT bound_box(box '((0,0),(1,1))', box '((3,3),(4,4))');

Result:
  bound_box
-------------
 (4,4),(0,0)
      (1 row)
```

- circle(*box*)
  Returns a circle circumscribed about box.

```
SELECT circle(box '((0,0),(1,1))');

Result:
            circle
-------------------------------
 <(0.5,0.5),0.707106781186548>
                        (1 row)
```

- circle(*point*, *double precision*)
  Returns the circle created using the center coordinates and radius of the input circle.

```
SELECT circle(point '(0,0)', 2.0);

Result:
  circle
-----------
 <(0,0),2>
    (1 row)
```

- circle(*polygon*)
  Returns a circle with the average of the input coordinate pairs as the center of the

circle and the average distance from the point to the input coordinate pair as the radius.

```
SELECT circle(polygon '((0,0),(1,1),(2,0))');

    Result:
                  circle
-------------------------------------------
 <(1,0.333333333333333),0.924950591148529>
                                   (1 row)
```

- line(*point*, *point*)
  Returns the value of the line.

```
SELECT line(point '(-1,0)', point '(1,0)');

Result:
   line
----------
 {0,-1,0}
   (1 row)
```

- lseg(*box*)
  Returns box diagonal to line segment.

```
SELECT lseg(box '((-1,0),(1,0))');

Result:
      lseg
----------------
 [(1,0),(-1,0)]
         (1 row)
```

- lseg(*point*, *point*)
  Returns a line segment with two input points taken as start and end points.

```
SELECT lseg(point '(-1,0)', point '(1,0)');

Result:
      lseg
----------------
 [(1,0),(-1,0)]
         (1 row)
```

- path(*polygon*)
  Returns a polygon path.

```
SELECT path(polygon '((0,0),(1,1),(2,0))');

Result:
        path
---------------------
 ((0,0),(1,1),(2,0))
             (1 row)
```

- point(*double precision, double precision*)
  Returns the value that construct the point.

```
SELECT point(23.4, -44.5);

Result:
    point
--------------
 (23.4,-44.5)
      (1 row)
```

- point(*box*)
  Returns center of box.

```
SELECT point(box '((-1,0),(1,0))');

Result:
 point
-------
 (0,0)
(1 row)
```

- point(*circle*)
  Returns center of circle.

```
SELECT point(circle '((0,0),2.0)');

Result:
 point
-------
 (0,0)
(1 row)
```

- point(*lseg*)
  Returns center of line segment.

```
SELECT point(lseg '((-1,0),(1,0))');
```

```
Result:
 point
-------
 (0,0)
(1 row)
```

- point(*polygon*)
  Returns center of polygon.

```
SELECT point(polygon '((0,0),(1,1),(2,0))');

Result:
         point
-----------------------
 (1,0.333333333333333)
                (1 row)
```

- polygon(*box*)
  Returns box to 4-point polygon.

```
SELECT polygon(box '((0,0),(1,1))');

Result:
          polygon
---------------------------
 ((0,0),(0,1),(1,1),(1,0))
                    (1 row)
```

- polygon(*circle*)
  Returns circle to 12-point polygon.

```
SELECT polygon(circle '((0,0),2.0)');

Result:
                                        polygon

-------------------------------------------------------------------------
----------------
 ((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-
16,2),
 (1,1.73205080756888),(1.73205080756888,1),(2,2.44929359829471e-16),
 (1.73205080756888,-0.999999999999999),(1,-1.73205080756888),(3.673940397
44206e-16,-2),
 (-0.999999999999999,-1.73205080756888),(-1.73205080756888,-1))

         (1 row)
```

- polygon(*npts, circle*)
  Returns circle to npts-point polygon.

```
SELECT polygon(12, circle '((0,0),2.0)');

                                        polygon


-------------------------------------------------------------------------
----------------
 ((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-
16,2),
 (1,1.73205080756888),(1.73205080756888,1),(2,2.44929359829471e-16),
 (1.73205080756888,-0.999999999999999),(1,-1.73205080756888),(3.673940397
44206e-16,-2),
 (-0.999999999999999,-1.73205080756888),(-1.73205080756888,-1))

         (1 row)
```

- polygon(*path*)
  Converts path into a polygon.

```
SELECT polygon(path '((0,0),(1,1),(2,0))');

Result:
       polygon
---------------------
 ((0,0),(1,1),(2,0))
             (1 row)
```

Network Address Functions

- abbrev(*inet*)
  Returns abbreviated display format as text.

```
SELECT abbrev(inet '10.1.0.0/16');

Result:
   abbrev
-------------
 10.1.0.0/16
  (1 row)
```

- abbrev(*cidr*)
  Returns abbreviated display format as text.

```
SELECT abbrev(cidr '10.1.0.0/16');

Result:
 abbrev
---------
 10.1/16
   (1 row)
```

- broadcast(*inet*)
  Returns broadcast address for network.

```
SELECT broadcast('192.168.1.5/24');

Result:
broadcast
------------------
 192.168.1.255/24
       (1 row)
```

- family(*inet*)
  Extracts family of address; 4 for IPv4, 6 for IPv6.

```
SELECT  family('::1');

Result:
 family
--------
      6
 (1 row)
```

- host(*inet*)
  Extracts IP address as text.

```
SELECT  host('192.168.1.5/24');

Result:
    host
-------------
 192.168.1.5
      (1 row)
```

- hostmask(*inet*)
  Constructs host mask for network.

```
SELECT hostmask('192.168.23.20/30');
```

```
Result:
 hostmask
----------
  0.0.0.3
   (1 row)
```

- masklen(*inet*)
  Extracts netmask length.

```
SELECT masklen('192.168.1.5/24');

Result:
 masklen
---------
      24
  (1 row)
```

- netmask(*inet*)
  Constructs netmask for network.

```
SELECT netmask('192.168.1.5/24');

Result:
    netmask
---------------
 255.255.255.0
       (1 row)
```

- network(*inet*)
  Extracts network part of address.

```
SELECT network('192.168.1.5/24');

Result:
    network
----------------
 192.168.1.0/24
        (1 row)
```

- set_masklen(*inet*, *int*)
  Returns the set netmask length for inet value.

```
SELECT set_masklen('192.168.1.5/24', 16);

Result:
  set_masklen
```

```
----------------
 192.168.1.5/16
        (1 row)
```

- set_masklen(*cidr*, *int*)
  Returns the set netmask length for cidr value.

```
SELECT set_masklen('192.168.1.0/24'::cidr, 16);

Result:
  set_masklen
----------------
 192.168.0.0/16
        (1 row)
```

- text(*inet*)
  Returns IP address and netmask length as text.

```
SELECT text(inet '192.168.1.5');

Result:
      text
----------------
 192.168.1.5/32
        (1 row)
```

- inet_same_family(*inet*, *inet*)
  Returns a logical value indicating whether the address is a family value.

```
SELECT inet_same_family('192.168.1.5/24', '::1');

Result:
 inet_same_family
------------------
        f
          (1 row)
```

- inet_merge(*inet*, *inet*)
  Returns the smallest network which includes all of the entered networks.

```
SELECT inet_merge('192.168.1.5/24', '192.168.2.5/24');

Result:
   inet_merge
----------------
```

```
 192.168.0.0/22
         (1 row)
```

- trunc(*macaddr*)
  Sets the last 3 bytes to zero.

```
SELECT trunc(macaddr '12:34:56:78:90:ab');

Result:
       trunc
-------------------
 12:34:56:00:00:00
           (1 row)
```

Text Search Functions

- array_to_tsvector(text[])
  Converts array of lexemes to *tsvector*.

```
SELECT array_to_tsvector('{fat,cat,rat}'::text[]);

Result:
 array_to_tsvector
-------------------
 'cat' 'fat' 'rat'
           (1 row)
```

- get_current_ts_config()
  Returns the default text search configuration.

```
SELECT get_current_ts_config();

Result:
 get_current_ts_config
-----------------------
               english
               (1 row)
```

- length(tsvector)
  Returns number of lexemes in *tsvector*.

```
SELECT length('fat:2,4 cat:3 rat:5A'::tsvector);

Result:
 length
--------
```

```
        3
 (1 row)
```

- numnode(tsquery)
  Returns the number of lexemes plus operators in *tsquery*.

```
SELECT numnode('(fat & rat) | cat'::tsquery);

Result:
 numnode
---------
       5
 (1 row)
```

- plainto_tsquery([*config* regconfig ,] *query* text)
  Produces *tsquery* ignoring punctuation.

```
SELECT plainto_tsquery('english', 'The Fat Rats');

Result:
 plainto_tsquery
-----------------
  'fat' & 'rat'
          (1 row)
```

- phraseto_tsquery([*config* regconfig ,] *query* text)
  Produces *tsquery* that searches for a phrase, ignoring punctuation.

```
SELECT phraseto_tsquery('english', 'The Fat Rats');

Result:
 phraseto_tsquery
------------------
 'fat' <-> 'rat'
           (1 row)
```

- querytree(*query* tsquery)
  Returns indexable part of a tsquery.

```
SELECT querytree('foo & ! bar'::tsquery);

Result:
 querytree
-----------
   'foo'
    (1 row)
```

- setweight(*vector* tsvector, *weight* "char")
  Assigns *weight* to each element of *vector*.

```
SELECT setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A');

Result:
          setweight
-------------------------------
 'cat':3A 'fat':2A,4A 'rat':5A
                        (1 row)
```

- setweight(*vector* tsvector, *weight* "char", *lexemes* text[])
  Assigns *weight* to elements of *vector* that are listed in *lexemes*.

```
SELECT setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A', '{cat,rat}');

Result:
          setweight
-----------------------------
 'cat':3A 'fat':2,4 'rat':5A
                      (1 row)
```

- strip(tsvector)
  Removes positions and weights from `tsvector`.

```
SELECT strip('fat:2,4 cat:3 rat:5A'::tsvector);

Result:
       strip
-------------------
 'cat' 'fat' 'rat'
            (1 row)
```

- to_tsquery([*config* regconfig ,] *query* text)
  Normalizes words and converts to `tsquery`.

```
SELECT to_tsquery('english', 'The & Fat & Rats');

Result:
  to_tsquery
---------------
 'fat' & 'rat'
        (1 row)
```

- to_tsvector([*config* regconfig ,] *document* text)
  Reduce document text to `tsvector`.

```
SELECT to_tsvector('english', 'The Fat Rats');

Result:
   to_tsvector
-----------------
 'fat':2 'rat':3
          (1 row)
```

- ts_delete(*vector* tsvector, *lexeme* text)
  Removes given *lexeme* from *vector*.

```
SELECT ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat');

Result:
    ts_delete
------------------
 'cat':3 'rat':5A
           (1 row)
```

- ts_delete(*vector* tsvector, *lexemes* text[])
  Removes any occurrence of lexemes in *lexemes* from *vector*.

```
SELECT ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat']);

Result:
 ts_delete
-----------
 'cat':3
    (1 row)
```

- ts_filter(*vector* tsvector, *weights* "char"[])
  Selects only elements with given *weights* from *vector*.

```
SELECT ts_filter('fat:2,4 cat:3b rat:5A'::tsvector, '{a,b}');

Result:
     ts_filter
-------------------
 'cat':3B 'rat':5A
            (1 row)
```

- ts_headline([*config* regconfig,] *document* text, *query* tsquery [, *options* text])
  Displays a query match.

```
SELECT ts_headline('x y z', 'z'::tsquery);
```

```
Result:
 ts_headline
--------------
 x y <b>z</b>
       (1 row)
```

- ts_rank([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])
  Ranks documents for query.

```
SELECT ts_rank(to_tsvector('This is an example of document'), to_tsquery
('example'));

Result:
  ts_rank
-----------
 0.0607927
    (1 row)
```

- ts_rank_cd([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])
  Ranks documents for query using cover density.

```
SELECT ts_rank_cd(to_tsvector('This is an example of document'), to_tsque
ry('example'));

Result:
 ts_rank_cd
------------
        0.1
     (1 row)
```

- ts_rewrite(*query* tsquery, *target* tsquery, *substitute* tsquery)
  Replaces *target* with *substitute* within query.

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery);

Result:
       ts_rewrite
-------------------------
 'b' & ( 'foo' | 'bar' )
                  (1 row)
```

- ts_rewrite(*query* tsquery, *select* text)
  Replaces the first column value of the SELECT result with the second column value of the SELECT result.

```
create table aliases (t tsquery primary key, s tsquery);
insert into aliases values ('a', 'foo|bar');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');

Result:
       ts_rewrite
-------------------------
 'b' & ( 'bar' | 'foo' )
                  (1 row)
```

- tsquery_phrase(*query1* tsquery, *query2* tsquery)
  Makes query that searches for *query1* followed by *query2* (same as <-> operator).

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'));

Result:
 tsquery_phrase
-----------------
 'fat' <-> 'cat'
          (1 row)
```

- tsquery_phrase(*query1* tsquery, *query2* tsquery, *distance* integer)
  Makes query that searches for *query1* followed by *query2* at distance distance.

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);

Result:
  tsquery_phrase
------------------
 'fat' <10> 'cat'
           (1 row)
```

- tsvector_to_array(tsvector)
  Converts `tsvector` to array of lexemes.

```
SELECT tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector);

Result:
 tsvector_to_array
-------------------
   {cat,fat,rat}
            (1 row)
```

- tsvector_update_trigger()
  Triggers the function for automatic `tsvector` column update.

```
CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE ON messages
FOR EACH ROW EXECUTE PROCEDURE
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES ('title here', 'the body text is here');

SELECT * FROM messages;

Result:
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1
                                                          (1 row)
```

- tsvector_update_trigger_column()
  Triggers the function for automatic `tsvector` column update.

```
CREATE TRIGGER ... tsvector_update_trigger_column(tsv, configcol, title,
body);
```

- unnest(tsvector, OUT *lexeme* text, OUT *positions* smallint[], OUT *weights* text)
  Expands a tsvector to a set of rows.

```
SELECT unnest('fat:2,4 cat:3 rat:5A'::tsvector);

Result:
        unnest
-----------------------
 (cat,{3},{D})
 (fat,"{2,4}","{D,D}")
 (rat,{5},{A})
                (3 row)
```

- ts_debug([*config* regconfig,] *document* text, OUT *alias* text, OUT *description* text, OUT *token* text, OUT *dictionaries* regdictionary[], OUT *dictionary* regdictionary, OUT *lexemes* text[])
  Tests a configuration.

```
SELECT ts_debug('english', 'The Brightest supernovaes');

Result:
                                     ts_debug


-------------------------------------------------------------------------
----------
```

```
 (asciiword,"Word, all ASCII",The,{english_stem},english_stem,{})
 (blank,"Space symbols"," ",{},,)
 (asciiword,"Word, all ASCII",Brightest,{english_stem},english_stem,{brig
htest})
 (blank,"Space symbols"," ",{},,)
 (asciiword,"Word, all ASCII",supernovaes,{english_stem},english_stem,{su
pernova})

   (5 row)
```

- ts_lexize(*dict* regdictionary, *token* text)
  Tests a dictionary.

```
SELECT ts_lexize('english_stem', 'stars');

Result:
 ts_lexize
-----------
  {star}
   (1 row)
```

- ts_parse(*parser_name* text, *document* text, OUT *tokid* integer, OUT *token* text)
  Tests a parser.

```
SELECT ts_parse('default', 'foo - bar');

Result:
 ts_parse
-----------
 (1,foo)
 (12," ")
 (12,"- ")
 (1,bar)
   (4 row)
```

- ts_parse(*parser_oid* oid, *document* text, OUT *tokid* integer, OUT *token* text)
  Tests a parser with oid.

```
SELECT ts_parse(3722, 'foo - bar');

Result:
 ts_parse
-----------
 (1,foo)
 (12," ")
 (12,"- ")
 (1,bar)
   (4 row)
```

- ts_token_type(*parser_name* text, OUT *tokid* integer, OUT *alias* text, OUT *description* text)
  Gets token types defined by parser.

```
SELECT ts_token_type('default');

Result:
         ts_token_type
---------------------------------
 (1,asciiword,"Word, all ASCII")
 ...
                       (23 row)
```

- ts_token_type(*parser_oid* oid, OUT *tokid* integer, OUT *alias* text, OUT *description* text)
  Gets token types defined by parser.

```
SELECT ts_token_type(3722);

Result:
         ts_token_type
---------------------------------
 (1,asciiword,"Word, all ASCII")
 ...
                       (23 row)
```

- ts_stat(*sqlquery* text, [*weights* text,] OUT *word* text, OUT *ndoc* integer, OUT *nentry* integer)
  Returns statistics of a `tsvector` column.

```
SELECT ts_stat('SELECT vector FROM apod');

Result:
  ts_stat
------------
(foo,10,15)
...
     (4 row)
```

JSON Functions

- to_json(anyelement), to_jsonb(anyelement)
  Returns the value as `json` or `jsonb`. Arrays and composites are converted to arrays and objects; otherwise, if there is a cast from the type to json, the cast function will be used to perform the conversion or a scalar value is produced. For any scalar type other

than a number, a Boolean, or a null value, the text representation will be used, in such a fashion that it is a valid `json` or `jsonb` value.

```
SELECT to_json('Fred said "Hi."'::text);

Result:
       to_json
---------------------
 "Fred said "Hi.\""
              (1 row)
```

- array_to_json(anyarray [, pretty_bool])
  Returns the array as a JSON array. Line feeds will be added between dimension-1 elements if returns `pretty_bool` is true.

```
SELECT array_to_json('{{1,5},{99,100}}'::int[]);

Result:
  array_to_json
------------------
 [[1,5],[99,100]]
           (1 row)

SELECT array_to_json('{{1,5},{99,100}}'::int[], true);

Result:
 array_to_json
---------------
 [[1,5],      +
  [99,100]]
         (1 row)
```

- row_to_json(record [, pretty_bool])
  Returns the row as a JSON object. Line feeds will be added between level-1 elements if `pretty_bool` is true.

```
SELECT row_to_json(row(1,'foo'));

Result:
     row_to_json
---------------------
 {"f1":1,"f2":"foo"}
              (1 row)
```

- json_build_array(VARIADIC "any"), jsonb_build_array(VARIADIC "any")
  Builds a possibly-heterogeneously-typed JSON array out of a variable argument list.

```
SELECT json_build_array(1,2,'3',4,5);

Result:
 json_build_array
-------------------
 [1, 2, "3", 4, 5]
            (1 row)
```

- json_build_object(VARIADIC "any"), jsonb_build_object(VARIADIC "any")
  Builds a JSON object out of a variable argument list. By convention, the argument list consists of alternating keys and values.

```
SELECT json_build_object('foo',1,'bar',2);

Result:
   json_build_object
------------------------
 {"foo" : 1, "bar" : 2}
                 (1 row)
```

- json_object(text[]), jsonb_object(text[])
  Builds a JSON object out of a text array. The array must have either exactly one dimension with an even number of members, in which case they are taken as alternating key/value pairs, or two dimensions such that each inner array has exactly two elements, which are taken as a key/value pair.

```
SELECT json_object('{a, 1, b, "def", c, 3.5}');
SELECT json_object('{{a, 1},{b, "def"},{c, 3.5}}');

Result:
              json_object
---------------------------------------
 {"a" : "1", "b" : "def", "c" : "3.5"}
                                (1 row)
```

- json_object(keys text[], values text[]), jsonb_object(keys text[], values text[])
  This form of `json_object` takes keys and values pairwise from two separate arrays. In all other respects it is identical to the one-argument form.

```
SELECT json_object('{a, b}', '{1,2}');

Result:
      json_object
------------------------
 {"a" : "1", "b" : "2"}
                 (1 row)
```

- json_array_length(json), jsonb_array_length(jsonb)
  Returns the number of elements in the outermost JSON array.

```
SELECT json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]');

Result:
 json_array_length
-------------------
                 5
            (1 row)
```

- json_each(json), jsonb_each(jsonb)
  Expands the outermost JSON object into a set of key/value pairs.

```
SELECT * FROM json_each('{"a":"foo", "b":"bar"}');

Result:
 key | value
-----+-------
 a   | "foo"
 b   | "bar"
      (2 row)
```

- json_each_text(json), jsonb_each_text(jsonb)
  Expands the outermost JSON object into a set of key/value pairs. The returned values will be of type `text`.

```
SELECT * FROM json_each_text('{"a":"foo", "b":"bar"}');

Result:
 key | value
-----+-------
 a   | foo
 b   | bar
      (2 row)
```

- json_extract_path(from_json json, VARIADIC path_elems text[]), jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])
  Returns JSON value pointed to by `path_elems` (equivalent to #> operator).

```
SELECT * FROM json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}
','f4');

Result:
  json_extract_path
----------------------
 {"f5":99,"f6":"foo"}
              (1 row)
```

- json_extract_path_text(from_json json, VARIADIC path_elems text[]), jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])
  Returns JSON value pointed to by `path_elems` as `text` (equivalent to #>> operator).

```
SELECT json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}',
'f4', 'f6');

Result:
 json_extract_path_text
------------------------
        foo
               (1 row)
```

- json_object_keys(json), jsonb_object_keys(jsonb)
  Returns set of keys in the outermost JSON object.

```
SELECT json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}');

Result:
 json_object_keys
------------------
       f1
       f2
           (2 row)
```

- json_populate_record(base anyelement, from_json json), jsonb_populate_record(base anyelement, from_json jsonb)
  Expands the object in `from_json` to a row whose columns match the record type defined by `base`.

```
CREATE TABLE myrowtype (a int, b int);

SELECT * FROM json_populate_record(null::myrowtype, '{"a":1,"b":2}');

Result:
 a | b
```

```
---+---
 1 | 2
(1 row)
```

- json_populate_recordset(base anyelement, from_json json), jsonb_populate_recordset(base anyelement, from_json jsonb)
  Expands the outermost array of objects in `from_json` to a set of rows whose columns match the record type defined by `base`.

```
SELECT * FROM json_populate_recordset(null::myrowtype, '[{"a":1,"b":2},{"
a":3,"b":4}]');

Result:
 a | b
---+---
 1 | 2
 3 | 4
(2 row)
```

- json_array_elements(json), jsonb_array_elements(jsonb)
  Expands a JSON array to a set of JSON values.

```
SELECT * FROM json_array_elements('[1,true, [2,false]]');

Result:
   value
-----------
     1
   true
[2,false]]
  (3 row)
```

- json_array_elements_text(json), jsonb_array_elements_text(jsonb)
  Expands a JSON array to a set of `text` values.

```
SELECT * FROM json_array_elements_text('["foo", "bar"]');

Result:
 value
-------
  foo
  bar
(2 row)
```

- json_typeof(json), jsonb_typeof(jsonb)
  Returns the type of the outermost JSON value as a text string. Possible types are `object`, `array`, `string`, `number`, `boolean`, and `null`.

```
SELECT json_typeof('-123.4');

Result:
 json_typeof
-------------
    number
      (1 row)
```

- json_to_record(json), jsonb_to_record(jsonb)
  Builds an arbitrary record from a JSON object. As with all functions returning `record`, the caller must explicitly define the structure of the record with an `AS` clause.

```
SELECT * FROM json_to_record('{"a":1,"b":[1,2,3],"c":"bar"}') as x(a int,
 b text, d text);

Result:
 a |    b    | d
---+---------+---
 1 | [1,2,3] |
          (1 row)
```

- json_to_recordset(json), jsonb_to_recordset(jsonb)
  Builds an arbitrary set of records from a JSON array of objects. As with all functions returning `record`, the caller must explicitly define the structure of the record with an `AS` clause.

```
SELECT * FROM json_to_recordset('[{"a":1,"b":"foo"},{"a":"2","c":"bar"}]')

as x(a int, b text);

Result:
 a |  b
---+-----
 1 | foo
 2 |
 (2 row)
```

- json_strip_nulls(from_json json), jsonb_strip_nulls(from_json jsonb)
  Returns `from_json` with all object fields that have null values omitted. Other null values are unchanged.

```
SELECT json_strip_nulls('[{"f1":1,"f2":null},2,null,3]');

Result:
  json_strip_nulls
---------------------
 [{"f1":1},2,null,3]
              (1 row)
```

- jsonb_set(target jsonb, path text[], new_value jsonb[, create_missing boolean])
  Returns `target` with the section designated by `path` replaced by `new_value`, or with `new_value` added
  if `create_missing` is true (default is `true`) and the item designated by `path` does not exist. As with the `path` orientated operators, negative integers that appear in path count from the end of JSON arrays.

```
SELECT jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', fal
se);

Result:
                  jsonb_set
---------------------------------------------
 [{"f1": [2, 3, 4], "f2": null}, 2, null, 3]
                                       (1 row)

SELECT jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}','[2,3,4]');

Result:
                  jsonb_set
---------------------------------------------
 [{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]
                                       (1 row)
```

- jsonb_insert(target jsonb, path text[], new_value jsonb, [insert_after boolean])
  Returns target with `new_value` inserted. If target section designated by `path` is in a JSONB array, `new_value` will be inserted before `target` or after if `insert_after` is true (default is `false`). If target section designated by `path` is in JSONB object, `new_value` will be inserted only if `target` does not exist. As with the `path` orientated operators, negative integers that appear in path count from the end of JSON arrays.

```
SELECT jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"');

Result:
         jsonb_insert
```

```
-------------------------------
 {"a": [0, "new_value", 1, 2]}
                        (1 row)

SELECT jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true);

Result:
         jsonb_insert
-------------------------------
 {"a": [0, 1, "new_value", 2]}
                        (1 row)
```

- jsonb_pretty(from_json jsonb)
  Returns `from_json` as indented JSON text.

```
SELECT jsonb_pretty('[{"f1":1,"f2":null},2,null,3]');

Result:
    jsonb_pretty
--------------------
 [                 +
     {             +
         "f1": 1,  +
         "f2": null+
     },            +
     2,            +
     null,         +
     3             +
 ]
            (1 row)
```

Sequence Manipulation Functions

This section describes functions for operating on sequences. Sequences can be created with CREATE SEQUENCE.

```
CREATE SEQUENCE serial increment by 1 start 101;
```

- nextval(regclass)
  Advances sequence and returns new value.

```
SELECT nextval('serial');

Result:
 nextval
---------
```

```
    101
  (1 row)
```

- currval(regclass)
  Returns value most recently obtained with `nextval` for specified sequence.

```
SELECT currval('serial');

Result:
 currval
---------
     101
  (1 row)
```

- lastval()
  Returns value most recently obtained with `nextval` for any sequence.

```
SELECT lastval();

Result:
 lastval
---------
     101
  (1 row)
```

- setval(regclass, bigint)
  Sets sequence's current value.

```
SELECT setval('serial', 101);

Result:
 setval
--------
    101
 (1 row)
```

- setval(regclass, bigint, boolean)
  Sets sequence's current value and `is_called` flag.

```
-- true
SELECT setval('serial', 101, true);

Result:
 setval
--------
    101
 (1 row)

SELECT nextval('serial');

Result:
 nextval
---------
     102
 (1 row)

-- false
SELECT setval('serial', 101, false);

Result:
 setval
--------
    101
 (1 row)

SELECT nextval('serial');

Result:
 nextval
---------
     101
 (1 row)
```

Array Functions

- array_append(anyarray, anyelement)
  Appends `anyelement` to the end of an array.

```
SELECT array_append(ARRAY[1,2], 3);

Result:
 array_append
--------------
   {1,2,3}
       (1 row)
```

- array_cat(anyarray, anyarray)
  Concatenates two arrays.

```
SELECT array_cat(ARRAY[1,2,3], ARRAY[4,5]);
```

```
Result:
  array_cat
-------------
 {1,2,3,4,5}
     (1 row)
```

- array_ndims(anyarray)
  Returns the number of dimensions of the array.

```
SELECT array_ndims(ARRAY[[1,2,3], [4,5,6]]);

Result:
 array_ndims
-------------
           2
     (1 row)
```

- array_dims(anyarray)
  Returns a text representation of array's dimensions.

```
SELECT array_dims(ARRAY[[1,2,3], [4,5,6]]);

Result:
 array_dims
------------
 [1:2][1:3]
    (1 row)
```

- array_fill(anyelement, int[], [, int[]])
  Returns an array initialized with optionally-supplied value and dimensions.

```
SELECT array_fill(7, ARRAY[3], ARRAY[2]);

Result:
  array_fill
---------------
 [2:4]={7,7,7}
       (1 row)
```

- array_length(anyarray, int)
  Returns the length of the requested array dimension.

```
SELECT array_length(array[1,2,3], 1);

Result:
 array_length
```

```
--------------
            3
      (1 row)
```

- array_lower(anyarray, int)
  Returns the lower bound of the requested array dimension.

```
SELECT array_lower('[0:2]={1,2,3}'::int[], 1);

Result:
 array_lower
-------------
          0
     (1 row)
```

- array_position(anyarray, anyelement [, int])
  Returns the index of the first occurrence of the second argument in the array, starting at the element indicated by the third argument or at the first element (array must be one-dimensional).

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], '
mon');

Result:
 array_position
----------------
              2
         (1 row)
```

- array_positions(anyarray, anyelement)
  Returns an array of indexes of all occurrences of the second argument in the array given as first argument (array must be one-dimensional).

```
SELECT array_positions(ARRAY['A','A','B','A'], 'A');

Result:
 array_positions
-----------------
     {1,2,4}
          (1 row)
```

- array_prepend(anyelement, anyarray)
  Appends `anyelement` to the beginning of an array.

```
SELECT array_prepend(1, ARRAY[2,3]);
```

```
Result:
 array_prepend
---------------
    {1,2,3}
        (1 row)
```

- array_remove(anyarray, anyelement)
  Removes all elements equal to the given value from the array.

```
SELECT array_remove(ARRAY[1,2,3,2], 2);

Result:
 array_remove
--------------
     {1,3}
       (1 row)
```

- array_replace(anyarray, anyelement, anyelement)
  Replaces each array element equal to the given value with a new value.

```
SELECT array_replace(ARRAY[1,2,5,4], 5, 3);

Result:
 array_replace
---------------
   {1,2,3,4}
        (1 row)
```

- array_to_string(anyarray, text [, text])
  Concatenates array elements using specified delimiter and optional null string.

```
SELECT array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*');

Result:
 array_to_string
-----------------
    1,2,3,*,5
          (1 row)
```

- array_upper(anyarray, int)
  Returns upper bound of the requested array dimension.

```
SELECT array_upper(ARRAY[1,8,3,7], 1);

Result:
 array_upper
```

```
-------------
           4
      (1 row)
```

- cardinality(anyarray)
  Returns the total number of elements in the array, or 0 if the array is empty.

```
SELECT cardinality(ARRAY[[1,2],[3,4]]);

Result:
 cardinality
-------------
           4
      (1 row)
```

- string_to_array(text, text [, text])
  Splits string into array elements using supplied delimiter and optional null string.

```
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'yy');

Result:
 string_to_array
-----------------
   {xx,NULL,zz}
          (1 row)
```

- unnest(anyarray)
  Expands an array to a set of rows.

```
SELECT unnest(ARRAY[1,2]);

Result:
 unnest
--------
      1
      2
  (2 row)
```

- unnest(anyarray, anyarray [, ...])
  Expands multiple arrays (possibly of different types) to a set of rows. This is only allowed in the FROM clause.

```
SELECT * FROM unnest(ARRAY[1,2],ARRAY['foo','bar','baz']);

Result:
 unnest | unnest
```

```
--------+--------
      1 | foo
      2 | bar
        | baz

          (1 row)
```

Range Functions and Operators

- lower(anyrange)
  Returns the lower bound of the input numeric range.

```
SELECT * FROM lower(numrange(1.1,2.2));

Result:

 lower
-------
   1.1
 (1 row)
```

- upper(anyrange)
  Returns upper of the input numeric range.

```
SELECT * FROM upper(numrange(1.1,2.2));

Result:

 upper
-------
   2.2
 (1 row)
```

- isempty(anyrange)
  Returns a boolean value indicating whether the entered number range is empty.

```
SELECT * FROM isempty(numrange(1.1,2.2));

Result:

 isempty
---------
 f
 (1 row)
```

- lower_inc(anyrange)
  Returns whether the lower bound of the entered number range exists.

```
SELECT * FROM lower_inc(numrange(1.1,2.2));

Result:

 lower_inc
-----------
 t
 (1 row)
```

- upper_inc(anyrange)
  Returns a logical value indicating whether the upper bound of the input numeral range exists.

```
SELECT * FROM upper_inc(numrange(1.1,2.2));

Result:

 lower_inc
-----------
 f
 (1 row)
```

- lower_inf(anyrange)
  Returns a logical value indicating whether the lower bound of the entered number range is infinite.

```
SELECT * FROM lower_inf('(,)'::daterange);

Result:

 lower_inf
-----------
 t
 (1 row)
```

- upper_inf(anyrange)
  Returns a logical value indicating whether the upper bound of the entered number range is infinite.

```
SELECT * FROM upper_inf('(,)'::daterange);

Result:
 upper_inf
-----------
 t
 (1 row)
```

- range_merge(anyrange, anyrange)
  Returns the smallest range which includes both of the given ranges.

```
SELECT * FROM range_merge('[1,2)'::int4range, '[3,4)'::int4range);

Result:

 range_merge
-------------
 [1,4)
 (1 row)
```

Aggregate Functions

- array_agg(expression)
  Returns input values, including nulls, concatenated into an array.

  Argument Type(s): `any non-array type`
  Return Type: `array of the argument type`

- array_agg(expression)
  Returns input arrays concatenated into array of one higher dimension (note: inputs must all have same dimensionality, and cannot be empty or NULL).

  Argument Type(s): `any array type`
  Return Type: `same as argument data type`

- avg(expression)
  Returns the average (arithmetic mean) of all input values.

  Argument Type(s): `smallint, int, bigint, real, double precision, numeric, or interval`
  Return Type: `numeric for any integer-type argument, double precision for a floating-point argument, otherwise the same as the argument data type`

- bit_and(expression)
  Returns the bitwise AND of all non-null input values, or null if none.

  Argument Type(s): `smallint, int, bigint, or bit`
  Return Type: `same as argument data type`

- bit_or(expression)
  Returns the bitwise OR of all non-null input values, or null if none.

  Argument Type(s): `smallint, int, bigint, or bit`
  Return Type: `same as argument data type`

- bool_and(expression)
  Returns true if all input values are true, otherwise false.

  Argument Type(s): `bool`
  Return Type: `bool`

- bool_or(expression)
  Returns true if at least one input value is true, otherwise false.

  Argument Type(s): `bool`
  Return Type: `bool`

- count(anything)
  Returns the number of input rows.

  Argument Type(s): `any`
  Return Type: `bigint`

- count(expression)
  Returns the number of input rows for which the value of expression is not null.

  Argument Type(s): `any`
  Return Type: `bigint`

- every(expression)
  Equivalent to bool_and.

  Argument Type(s): `bool`
  Return Type: `bool`

- json_agg(expression)
  Aggregates values as a JSON array.

  Argument Type(s): `any`
  Return Type: `json`

- jsonb_agg(expression)
  Aggregates values as a JSON array.

  Argument Type(s): `any`
  Return Type: `jsonb`

- json_object_agg(name, value)
  Aggregates name/value pairs as a JSON object.

  Argument Type(s): `(any, any)`
  Return Type: `json`

- jsonb_object_agg(name, value)
  Aggregates name/value pairs as a JSON object.

  Argument Type(s): `(any, any)`
  Return Type: `jsonb`

- max(expression)
  Returns the maximum value of expression across all input values.

  Argument Type(s): `any numeric, string, date/time, network, or enum type, or arrays of these types`
  Return Type: `same as argument type`

- min(expression)
  Returns the minimum value of expression across all input values.

  Argument Type(s): `any numeric, string, date/time, network, or enum type, or arrays of these types`
  Return Type: `same as argument type`

- string_agg(expression, delimiter)
  Returns the input values concatenated into a string, separated by delimiter.

  Argument Type(s): `(text, text) or (bytea, bytea)`
  Return Type: `same as argument types`

- sum(expression)
  Returns the sum of expression across all input values.

  Argument Type(s): `smallint, int, bigint, real, double precision, numeric, interval, or money`
  Return Type: `bigint for smallint or int arguments, numeric for bigint arguments, otherwise the same as the argument data type`

- xmlagg(expression)
  Returns the concatenation of XML values.

  Argument Type(s): `xml`
  Return Type: `xml`

- corr(Y, X)
  Returns correlation coefficient of the two entered numbers.

  Argument Type(s): `double precision`
  Return Type: `double precision`

- covar_pop(Y, X)
  Returns population covariance of the two entered numbers.

  Argument Type(s): `double precision`
  Return Type: `double precision`

- covar_samp(Y, X)
  Returns sample covariance of the two entered numbers.

  Argument Type(s): `double precision`
  Return Type: `double precision`

- regr_avgx(Y, X)
  Returns average of the independent variable X (sum(X)/N).

  Argument Type(s): `double precision`
  Return Type: `double precision`

- regr_avgy(Y, X)
  Returns average of the dependent variable Y (sum(Y)/N).

  Argument Type(s): `double precision`
  Return Type: `double precision`

- regr_count(Y, X)
  Returns the number of input rows in which both expressions are nonnull.

  Argument Type(s): `double precision`
  Return Type: `bigint`

- regr_intercept(Y, X)
  Returns y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs.

  Argument Type(s): `double precision`
  Return Type: `double precision`

- regr_r2(Y, X)
  Returns square of the correlation coefficient of the two entered numbers.

Argument Type(s): `double precision`
Return Type: `double precision`

- regr_slope(Y, X)
  Returns slope of the least-squares-fit linear equation determined by the (X, Y) pairs.

  Argument Type(s): `double precision`
  Return Type: `double precision`

- regr_sxx(Y, X)
  Returns sum(X^2) - sum(X)^2/N ("sum of squares" of the independent variable).

  Argument Type(s): `double precision`
  Return Type: `double precision`

- regr_sxy(Y, X)
  Returns sum(X*Y) - sum(X)* sum(Y)/N ("sum of products" of independent times dependent variable).

  Argument Type(s): `double precision`
  Return Type: `double precision`

- regr_syy(Y, X)
  Returns sum(Y^2) - sum(Y)^2/N ("sum of squares" of the dependent variable).

  Argument Type(s): `double precision`
  Return Type: `double precision`

- stddev(expression)
  Returns historical alias for stddev_samp.

  Argument Type(s): `smallint, int, bigint, real, double precision, or numeric`
  Return Type: `double precision for floating-point arguments, otherwise numeric`

- stddev_pop(expression)
  Returns population standard deviation of the input values.

  Argument Type(s): `smallint, int, bigint, real, double precision, or numeric`
  Return Type: `double precision for floating-point arguments, otherwise numeric`

- stddev_samp(expression)
  Returns sample standard deviation of the input values.

Argument Type(s): `smallint, int, bigint, real, double precision, or numeric`
Return Type: `double precision for floating-point arguments, otherwise numeric`

- variance(expression)
  Returns historical alias for var_samp.

  Argument Type(s): `smallint, int, bigint, real, double precision, or numeric`
  Return Type: `double precision for floating-point arguments, otherwise numeric`

- var_pop(expression)
  Returns population variance of the input values (square of the population standard deviation).

  Argument Type(s): `smallint, int, bigint, real, double precision, or numeric`
  Return Type: `double precision for floating-point arguments, otherwise numeric`

- var_samp(expression)
  Returns sample variance of the input values (square of the sample standard deviation).

  Argument Type(s): `smallint, int, bigint, real, double precision, or numeric`
  Return Type: `double precision for floating-point arguments, otherwise numeric`

- mode() WITHIN GROUP (ORDER BY sort_expression)
  Returns the most frequent input value (arbitrarily choosing one if there are multiple equally-frequent results).

  Argument Type(s): `any sortable type`
  Return Type: `same as sort expression`

- percentile_cont(fraction) WITHIN GROUP (ORDER BY sort_expression)
  Returns a value corresponding to the specified fraction in the ordering, interpolating between adjacent input items if needed.

  Argument Type(s): `double precision or interval`
  Return Type: `same as sort expression`

- percentile_cont(fractions) WITHIN GROUP (ORDER BY sort_expression)
  Returns an array of results matching the shape of the fractions parameter, with each non-null element replaced by the value corresponding to that percentile

  Argument Type(s): `double precision or interval`
  Return Type: `array of sort expression's type`

- percentile_disc(fraction) WITHIN GROUP (ORDER BY sort_expression)
  Returns the first input value whose position in the ordering equals or exceeds the specified fraction.

  Argument Type(s): `any sortable type`
  Return Type: `same as sort expression`

- percentile_disc(fractions) WITHIN GROUP (ORDER BY sort_expression)
  Returns an array of results matching the shape of the fractions parameter, with each non-null element replaced by the input value corresponding to that percentile.

  Argument Type(s): `any sortable type`
  Return Type: `array of sort expression's type`

- rank(args) WITHIN GROUP (ORDER BY sorted_args)
  Returns rank of the argument, with gaps for duplicate rows.

  Argument Type(s): `VARIADIC "any"`
  Return Type: `bigint`

- dense_rank(args) WITHIN GROUP (ORDER BY sorted_args)
  Returns rank of the argument, without gaps for duplicate rows.

  Argument Type(s): `VARIADIC "any"`
  Return Type: `bigint`

- percent_rank(args) WITHIN GROUP (ORDER BY sorted_args)
  Returns relative rank of the argument, ranging from 0 to 1.

  Argument Type(s): `VARIADIC "any"`
  Return Type: `double precision`

- cume_dist(args) WITHIN GROUP (ORDER BY sorted_args)
  Returns relative rank of the argument, ranging from 1/N to 1.

  Argument Type(s): `VARIADIC "any"`
  Return Type: `double precision`

- GROUPING(args...)
  Returns integer bit mask indicating which arguments are not being included in the current grouping set.

  Return Type: `integer`

Window Functions

- row_number()
  Returns the number of the current row within its partition, counting from 1.

  Return Type: `bigint`

- rank()
  Returns rank of the current row with gaps; same as row_number of its first peer.

  Return Type: `bigint`

- dense_rank()
  Returns rank of the current row without gaps; this function counts peer groups.

  Return Type: `bigint`

- percent_rank()
  Returns relative rank of the current row: (rank - 1)/(total partition rows - 1).

  Return Type: `double precision`

- cume_dist()
  Returns cumulative distribution: (number of partition rows preceding or peer with current row)/total partition rows.

  Return Type: `double precision`

- ntile(num_buckets integer)
  Returns integer ranging from 1 to the argument value, dividing the partition as equally as possible.

  Return Type: `integer`

- lag(value anyelement [, offset integer [, default anyelement]])
  Returns value evaluated at the row that is offset rows before the current row within the partition; if there is no such row, instead return default (which must be of the same type as value). Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to null.

  Return Type: `same type as value`

- lead(value anyelement [, offset integer [, default anyelement]])
  Returns value evaluated at the row that is offset rows after the current row within the partition; if there is no such row, instead return default (which must be of the same type as value). Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to null.

  Return Type: `same type as value`

- first_value(value any)
  Returns the value evaluated at the row that is the first row of the window frame.

  Return Type: `same type as value`

- last_value(value any)
  Returns the value evaluated at the row that is the last row of the window frame.

  Return Type: `same type as value`

- nth_value(value any, nth integer)
  Returns the value evaluated at the row that is the nth row of the window frame (counting from 1); null if there is no such row.

  Return Type: `same type as value`

System Information Functions

- current_catalog
  Name of current database (called “catalog” in the SQL standard).

```
SELECT current_catalog;

Result:
 current_database
------------------
       test
           (1 row)
```

- current_database()
  Returns name of current database.

```
SELECT current_database();

Result:
 current_database
------------------
```

```
           test
               (1 row)
```

- current_query()
  Returns text of the currently executing query, as submitted by the client (might contain more than one statement).

```
SELECT current_query();

Result:
      current_query
-------------------------
 SELECT current_query();
                  (1 row)
```

- current_role
  Returns equivalent to `current_user`.

```
SELECT current_role;

Result:
 current_user
--------------
    agens
       (1 row)
```

- current_schema[()]
  Returns name of current schema.

```
SELECT current_schema();

Result:
 current_schema
----------------
     public
         (1 row)
```

- current_schemas(boolean)
  Returns names of schemas in search path, optionally including implicit schemas.

```
SELECT current_schemas(true);

Result:
   current_schemas
---------------------
```

```
 {pg_catalog,public}
              (1 row)
```

- current_user
  Returns user name of current execution context.

```
SELECT current_user;

Result:
 current_user
--------------
    agens
      (1 row)
```

- inet_client_addr()
  Returns address of the remote connection.

```
SELECT inet_client_addr();

Result:
 inet_client_addr
------------------
       ::1
           (1 row)
```

- inet_client_port()
  Returns port of the remote connection.

```
SELECT inet_client_port();

Result:
 inet_client_port
------------------
      64427
           (1 row)
```

- inet_server_addr()
  Returns address of the local connection.

```
SELECT inet_server_addr();

Result:
 inet_server_addr
------------------
       ::1
           (1 row)
```

- inet_server_port()
  Returns port of the local connection.

```
SELECT inet_server_port();

Result:
 inet_server_port
------------------
       5432
           (1 row)
```

- pg_backend_pid()
  Returns the process ID of the server process attached to the current session.

```
SELECT pg_backend_pid();

Result:
 pg_backend_pid
----------------
          61675
         (1 row)
```

- pg_blocking_pids(int)
  Returns the process ID(s) that are blocking specified server process ID.

```
SELECT pg_blocking_pids(61675);

Result:
 pg_blocking_pids
------------------
        {}
           (1 row)
```

- pg_conf_load_time()
  Returns configuration load time.

```
SELECT pg_conf_load_time();

Result:
      pg_conf_load_time
------------------------------
 2017-10-18 13:36:51.99984+09
                       (1 row)
```

- pg_my_temp_schema()
  Returns OID of session's temporary schema, or 0 if none.

```
SELECT pg_my_temp_schema();

Result:
 pg_my_temp_schema
-------------------
                 0
           (1 row)
```

- pg_is_other_temp_schema(oid)
  Returns whether schema is another session's temporary schema.

```
SELECT pg_is_other_temp_schema(61675);

Result:
 pg_is_other_temp_schema
-------------------------
           f
                 (1 row)
```

- pg_listening_channels()
  Returns channel names that the session is currently listening on.

```
SELECT pg_listening_channels();

Result:
 pg_listening_channels
-----------------------
               (0 row)
```

- pg_notification_queue_usage()
  Returns fraction of the asynchronous notification queue currently occupied (0-1).

```
SELECT pg_notification_queue_usage();

Result:
 pg_notification_queue_usage
-----------------------------
                           0
                     (1 row)
```

- pg_postmaster_start_time()
  Returns server start time.

```
SELECT pg_postmaster_start_time();

Result:
   pg_postmaster_start_time
-------------------------------
 2017-10-18 13:36:52.019037+09
                        (1 row)
```

- pg_trigger_depth()
  Returns current nesting level of PostgreSQL triggers (0 if not called, directly or indirectly, from inside a trigger).

```
SELECT pg_trigger_depth();

Result:
 pg_trigger_depth
------------------
                0
           (1 row)
```

- session_user
  Returns session user name.

```
SELECT session_user;

Result:
 session_user
--------------
     agens
       (1 row)
```

- user
  Returns the equivalent to current_user.

```
SELECT user;

Result:
 current_user
--------------
     agens
       (1 row)
```

- version()
  Returns AgensGraph's version info.

```
SELECT version();

Result:
                                version
```

```
----------------------------------------------------------------------------
------
 PostgreSQL 9.6.2 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.4.7 201
20313
 (Red Hat 4.4.7-17), 64-bit

                                                                       (1 r
ow)
```

- has_any_column_privilege(user, table, privilege)
  Returns a boolean value indicating whether user has the same privileges on all columns of table as others.

```
SELECT has_any_column_privilege('agens', 'myschema.mytable', 'SELECT');


Result:
 has_any_column_privilege
--------------------------
            t
                   (1 row)
```

- has_any_column_privilege(table, privilege)
  Returns whether current user has privilege for all columns of table.

```
SELECT has_any_column_privilege('myschema.mytable', 'SELECT');

Result:
 has_any_column_privilege
--------------------------
            t
                   (1 row)
```

- has_column_privilege(user, table, column, privilege)
  Returns whether user has privilege for table's column.

```
SELECT has_column_privilege('agens', 'myschema.mytable', 'col1', 'SELECT
');

Result:
 has_column_privilege
----------------------
          t
               (1 row)
```

- has_column_privilege(table, column, privilege)
  Returns whether current user has privilege for table's column.

```
SELECT has_column_privilege('myschema.mytable', 'col1', 'SELECT');

Result:
 has_column_privilege
----------------------
          t
               (1 row)
```

- has_database_privilege(user, database, privilege)
  Returns whether user has privilege for database.

```
SELECT has_database_privilege('agens', 'test', 'connect');

Result:
 has_database_privilege
------------------------
           t
                 (1 row)
```

- has_database_privilege(database, privilege)
  Returns whether current user has privilege for database.

```
SELECT has_database_privilege('test', 'connect');

Result:
 has_database_privilege
------------------------
           t
               (1 row)
```

- has_foreign_data_wrapper_privilege(user, fdw, privilege)
  Returns whether user has privilege for foreign-data wrapper.

```
CREATE EXTENSION postgres_fdw;

SELECT has_foreign_data_wrapper_privilege('agens', 'postgres_fdw', 'usage
');

Result:
 has_foreign_data_wrapper_privilege
------------------------------------
                 t
                             (1 row)
```

- has_foreign_data_wrapper_privilege(fdw, privilege)
  Returns whether current user has privilege for foreign-data wrapper.

```
SELECT has_foreign_data_wrapper_privilege('postgres_fdw', 'usage');

Result:
 has_foreign_data_wrapper_privilege
------------------------------------
                 t
                             (1 row)
```

- has_function_privilege(user, function, privilege))
  Returns whether user has privilege for function.

```
SELECT has_function_privilege('agens', 'getfoo()', 'execute');

Result:
 has_function_privilege
------------------------
           t
                 (1 row)
```

- has_function_privilege(function, privilege)
  Returns whether current user has privilege for function.

```
SELECT has_function_privilege('getfoo()', 'execute');

Result:
 has_function_privilege
------------------------
           t
                 (1 row)
```

- has_language_privilege(user, language, privilege)
  Returns whether the user has privilege for language.

```
SELECT has_language_privilege('agens', 'c', 'usage');

Result:
 has_language_privilege
------------------------
           t
                 (1 row)
```

- has_language_privilege(language, privilege)
  Returns whether current user has privilege for language.

```
SELECT has_language_privilege('c', 'usage');

Result:
 has_language_privilege
------------------------
           t
                 (1 row)
```

- has_schema_privilege(user, schema, privilege)
  Returns whether user has privilege for schema.

```
SELECT has_schema_privilege('agens', 'myschema', 'usage');

Result:
 has_schema_privilege
----------------------
          t
               (1 row)
```

- has_schema_privilege(schema, privilege)
  Returns whether current user has privilege for schema.

```
SELECT has_schema_privilege('myschema', 'usage');

Result:
 has_schema_privilege
----------------------
          t
               (1 row)
```

- has_sequence_privilege(user, sequence, privilege)
  Returns whether the user has privilege for sequence.

```
 SELECT has_sequence_privilege('agens', 'serial', 'usage');

 Result:
  has_sequence_privilege
 ------------------------
            t
                  (1 row)
```

- has_sequence_privilege(sequence, privilege)
  Returns whether user has privilege for sequence.

```
SELECT has_sequence_privilege('serial', 'usage');
```

```
     Result:
      has_sequence_privilege
     ------------------------
                t
                      (1 row)
```

- has_server_privilege(user, server, privilege)
  Returns whether user has privilege for server.

```
    CREATE SERVER app_database_server
    FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (host '127.0.0.1', dbname 'agens');

    SELECT has_server_privilege('agens', 'app_database_server', 'usage');

    Result:
     has_server_privilege
    ----------------------
              t
                   (1 row)
```

- has_server_privilege(server, privilege)
  Returns whether current user has privilege for server.

```
    SELECT has_server_privilege('app_database_server', 'usage');

    Result:
     has_server_privilege
    ----------------------
              t
                   (1 row)
```

- has_table_privilege(user, table, privilege)
  Returns whether user has privilege for table.

```
    SELECT has_table_privilege('agens', 'myschema.mytable', 'SELECT');

    Result:
     has_table_privilege
    ---------------------
              t
                   (1 row)
```

- has_table_privilege(table, privilege)
  Returns whether current user has privilege for table.

```
SELECT has_table_privilege('myschema.mytable', 'SELECT');

Result:
 has_table_privilege
---------------------
          t
               (1 row)
```

- has_tablespace_privilege(user, tablespace, privilege)
  Returns whether user has privilege for tablespace.

```
SELECT has_tablespace_privilege('agens', 'pg_default', 'create');

Result:
 has_tablespace_privilege
--------------------------
             t
                  (1 row)
```

- has_tablespace_privilege(tablespace, privilege)
  Returns whether current user has privilege for tablespace.

```
SELECT has_tablespace_privilege('pg_default', 'create');

Result:
 has_tablespace_privilege
--------------------------
             t
                  (1 row)
```

- has_type_privilege(user, type, privilege)
  Returns whether user has privilege for type.

```
SELECT has_type_privilege('agens', 'rainbow', 'usage');

Result:
 has_type_privilege
--------------------
         t
             (1 row)
```

- has_type_privilege(type, privilege)
  Returns whether current user has privilege for type.

```
SELECT has_type_privilege('rainbow', 'usage');
```

```
Result:
 has_type_privilege
--------------------
         t
             (1 row)
```

- pg_has_role(user, role, privilege)
  Returns whether user has privilege for role.

```
SELECT pg_has_role('agens', 'agens', 'usage');

Result:
 pg_has_role
-------------
      t
      (1 row)
```

- pg_has_role(role, privilege)
  Returns whether current user has privilege for role.

```
SELECT pg_has_role('agens', 'usage');

Result:
 pg_has_role
-------------
      t
      (1 row)
```

- row_security_active(table)
  Returns whether current user has row level security active for table.

```
SELECT row_security_active('myschema.mytable');

Result:
 row_security_active
---------------------
         f
              (1 row)
```

- pg_collation_is_visible(collation_oid)
  Returns whether collation is visible in search path.

```
SELECT pg_collation_is_visible(100);

Result:
 pg_collation_is_visible
```

```
-------------------------
            t
                  (1 row)
```

- pg_conversion_is_visible(conversion_oid)
  Returns whether conversion is visible in search path.

```
SELECT pg_conversion_is_visible(12830);

Result:
 pg_conversion_is_visible
--------------------------
            t
                   (1 row)
```

- pg_function_is_visible(function_oid)
  Returns whether function is visible in search path.

```
SELECT pg_function_is_visible(16716);

Result:
 pg_function_is_visible
------------------------
            t
                 (1 row)
```

- pg_opclass_is_visible(opclass_oid)
  Returns whether opclass is visible in search path.

```
SELECT pg_opclass_is_visible(10007);

Result:
 pg_opclass_is_visible
-----------------------
            t
                 (1 row)
```

- pg_operator_is_visible(operator_oid)
  Returns whether operator is visible in search path.

```
SELECT pg_operator_is_visible(15);

Result:
 pg_operator_is_visible
------------------------
```

```
            t
                 (1 row)
```

- pg_opfamily_is_visible(opclass_oid)
  Returns whether opfamily is visible in search path.

```
SELECT pg_opfamily_is_visible(421);

Result:
 pg_opfamily_is_visible
------------------------
            t
                 (1 row)
```

- pg_table_is_visible(table_oid)
  Returns whether table is visible in search path.

```
SELECT pg_table_is_visible(16553);

Result:
 pg_table_is_visible
---------------------
          t
              (1 row)
```

- pg_ts_config_is_visible(config_oid)
  Returns whether text search configuration is visible in search path.

```
SELECT pg_ts_config_is_visible(3748);

Result:
 pg_ts_config_is_visible
-------------------------
            t
                  (1 row)
```

- pg_ts_dict_is_visible(dict_oid)
  Returns whether text search dictionary is visible in search path.

```
SELECT pg_ts_dict_is_visible(3765);

Result:
 pg_ts_dict_is_visible
-----------------------
           t
                (1 row)
```

- pg_ts_parser_is_visible(parser_oid)
  Returns whether text search parser is visible in search path.

```
SELECT pg_ts_parser_is_visible(3722);

Result:
 pg_ts_parser_is_visible
-------------------------
             t
                  (1 row)
```

- pg_ts_template_is_visible(template_oid)
  Returns whether text search template is visible in search path.

```
SELECT pg_ts_template_is_visible(3727);

Result:
 pg_ts_template_is_visible
---------------------------
             t
                    (1 row)
```

- pg_type_is_visible(type_oid)
  Returns whether type or domain is visible in search path.

```
SELECT pg_type_is_visible(16);

Result:
 pg_type_is_visible
--------------------
          t
             (1 row)
```

- format_type(type_oid, typemod)
  Gets the name of a data type.

```
SELECT format_type(16, 1);

Result:
 format_type
-------------
   boolean
      (1 row)
```

- pg_get_constraintdef(constraint_oid)
  Gets definition of a constraint.

```
SELECT pg_get_constraintdef(13096);

Result:
 pg_get_constraintdef
----------------------
 CHECK ((VALUE >= 0))
               (1 row)
```

- pg_get_constraintdef(constraint_oid, pretty_bool)
  Gets definition of a constraint.

```
SELECT pg_get_constraintdef(13096, true);

Result:
 pg_get_constraintdef
----------------------
 CHECK ((VALUE >= 0))
               (1 row)
```

- pg_get_functiondef(func_oid)
  Gets definition of a function.

```
SELECT pg_get_functiondef(16716);

Result:
             pg_get_functiondef
--------------------------------------------
 CREATE OR REPLACE FUNCTION public.getfoo()+
 ...
                                     (1 row)
```

- pg_get_function_arguments(func_oid)
  Gets argument list of function's definition (with default values).

```
SELECT pg_get_function_arguments(16739);

Result:
     pg_get_function_arguments
------------------------------------
 double precision, double precision
                             (1 row)
```

- pg_get_function_identity_arguments(func_oid)
  Gets argument list to identify a function (without default values).

```
SELECT pg_get_function_identity_arguments(16739);

Result:
 pg_get_function_identity_arguments
------------------------------------
 double precision, double precision
                             (1 row)
```

- pg_get_function_result(func_oid)
  Gets RETURNS clause for function.

```
SELECT pg_get_function_result(16739);

Result:
 pg_get_function_result
------------------------
       float8_range
              (1 row)
```

- pg_get_indexdef(index_oid)
  Gets `CREATE INDEX` command for index.

```
SELECT pg_get_indexdef(828);

    Result:
                                 pg_get_indexdef

-------------------------------------------------------------------------
---------
 CREATE UNIQUE INDEX pg_default_acl_oid_index ON pg_default_acl USING btr
ee (oid)

  (1 row)
```

- pg_get_indexdef(index_oid, column_no, pretty_bool)
  Gets CREATE INDEX command for index, or definition of just one index column when column_no is not zero.

```
SELECT pg_get_indexdef(828, 1, true);

Result:
 pg_get_indexdef
-----------------
       oid
          (1 row)
```

- pg_get_keywords()
  Gets list of SQL keywords and their categories.

```
SELECT pg_get_keywords();

Result:
      pg_get_keywords
---------------------------
   (abort,U,unreserved)
  (absolute,U,unreserved)
 ...
                  (434 row)
```

- pg_get_ruledef(rule_oid)
  Gets `CREATE RULE` command for rule.

```
SELECT pg_get_ruledef(11732);

Result:
            pg_get_ruledef
---------------------------------------
 CREATE RULE "_RETURN" AS
 ...
                                (1 row)
```

- pg_get_ruledef(rule_oid, pretty_bool)
  Gets `CREATE RULE` command for rule.

```
SELECT pg_get_ruledef(11732, true);

Result:
            pg_get_ruledef
---------------------------------------
 CREATE RULE "_RETURN" AS
 ...
                                (1 row)
```

- pg_get_serial_sequence(table_name, column_name)
  Returns the name of the sequence using `serial`, `smallserial`, and `bigserial` columns.

```
SELECT pg_get_serial_sequence('serial_t', 'col1');

Result:
  pg_get_serial_sequence
--------------------------
```

```
 public.serial_t_col1_seq
                     (1 row)
```

- pg_get_triggerdef(trigger_oid)
  Gets `CREATE [ CONSTRAINT ] TRIGGER` command for trigger.

```
SELECT pg_get_triggerdef(16887);

Result:
                          pg_get_triggerdef
-------------------------------------------------------------------------
 CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE ON messages
 FOR EACH ROW EXECUTE PROCEDURE
 tsvector_update_trigger_column('tsv', 'configcol', 'title', 'body')
                                                                  (1 row)
```

- pg_get_triggerdef(trigger_oid, pretty_bool)
  Gets `CREATE [ CONSTRAINT ] TRIGGER` command for trigger.

```
SELECT pg_get_triggerdef(16887, true);

Result:
                          pg_get_triggerdef
-------------------------------------------------------------------------
 CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE ON messages
 FOR EACH ROW EXECUTE PROCEDURE
 tsvector_update_trigger_column('tsv', 'configcol', 'title', 'body')
                                                                  (1 row)
```

- pg_get_userbyid(role_oid)
  Gets role name with given OID.

```
SELECT pg_get_userbyid(13096);

Result:
 pg_get_userbyid
-----------------
      agens
          (1 row)
```

- pg_get_viewdef(view_oid)
  Gets underlying `SELECT` command for view or mview.

```
SELECT pg_get_viewdef(17046);

Result:
          pg_get_viewdef
-----------------------------------
  SELECT pg_class.relname,        +
     pg_class.relnamespace,       +
     ...
    FROM pg_class;
                            (1 row)
```

- pg_get_viewdef(view_oid, pretty_bool)
  Gets underlying `SELECT` command for view or mview.

```
SELECT pg_get_viewdef(17046, true);

Result:
          pg_get_viewdef
-----------------------------------
  SELECT pg_class.relname,        +
     pg_class.relnamespace,       +
     ...
    FROM pg_class;
                           (1 row)
```

- pg_get_viewdef(view_oid, wrap_column_int)
  Gets underlying `SELECT` command for view or mview; lines with fields are wrapped to specified number of columns.

```
SELECT pg_get_viewdef(17046,50);

Result:
                   pg_get_viewdef
-----------------------------------------------------
  SELECT pg_class.relname, pg_class.relnamespace,   +
     pg_class.reltype, pg_class.reloftype,          +
     pg_class.relowner, pg_class.relam,             +
     pg_class.relfilenode, pg_class.reltablespace,  +
     pg_class.relpages, pg_class.reltuples,         +
     pg_class.relallvisible, pg_class.reltoastrelid,+
     pg_class.relhasindex, pg_class.relisshared,    +
     pg_class.relpersistence, pg_class.relkind,     +
     pg_class.relnatts, pg_class.relchecks,         +
     pg_class.relhasoids, pg_class.relhaspkey,      +
     pg_class.relhasrules, pg_class.relhastriggers, +
     pg_class.relhassubclass,                       +
     pg_class.relrowsecurity,                       +
     pg_class.relforcerowsecurity,                  +
     pg_class.relispopulated, pg_class.relreplident,+
     pg_class.relfrozenxid, pg_class.relminmxid,    +
     pg_class.relacl, pg_class.reloptions           +
    FROM pg_class;
                                              (1 row)
```

- pg_index_column_has_property(index_oid, column_no, prop_name)
  Tests whether an index column has a specified property.

```
SELECT pg_index_column_has_property(17134, 1, 'orderable');

Result:
 pg_index_column_has_property
------------------------------
             t
                      (1 row)
```

- pg_index_has_property(index_oid, prop_name)
  Tests whether an index has a specified property.

```
SELECT pg_index_has_property(17134, 'clusterable');

Result:
 pg_index_has_property
-----------------------
          t
               (1 row)
```

- pg_indexam_has_property(am_oid, prop_name)
  Tests whether an index access method has a specified property.

```
SELECT pg_indexam_has_property(403, 'can_order');

Result:
 pg_indexam_has_property
-------------------------
          t
                  (1 row)
```

- pg_options_to_table(reloptions)
  Gets name/value pairs of the set of storage option.

```
SELECT pg_options_to_table(reloptions) FROM pg_class;

Result:
   pg_options_to_table
-------------------------
 (security_barrier,true)
                 (1 row)
```

- pg_tablespace_databases(tablespace_oid)
  Gets the set of database OIDs that have objects in the tablespace.

```
SELECT pg_tablespace_databases(1663);

Result:
 pg_tablespace_databases
-------------------------
                       1
                   13372
                   13373
                   16384
                   16482
                 (5 row)
```

- pg_tablespace_location(tablespace_oid)
  Gets the path in the file system that this tablespace is located in.

```
SELECT pg_tablespace_location(1663);

Result:
      pg_tablespace_location
----------------------------------
 /home/agens/AgensGraph/db_cluster
                           (1 row)
```

- pg_typeof(any)
  Gets the data type of any value.

```
SELECT pg_typeof(1);

Result:
 pg_typeof
-----------
  integer
    (1 row)
```

- collation for (any)
  Gets the collation of the argument.

```
SELECT collation for ('foo' COLLATE "de_DE");

Result:
 pg_collation_for
------------------
     "de_DE"
           (1 row)
```

- pg_describe_object(catalog_id, object_id, object_sub_id)
  Gets description of a database object.

```
SELECT pg_describe_object(1255, 16716, 0);

Result:
 pg_describe_object
--------------------
  function 16716
             (1 row)
```

- pg_identify_object(catalog_id oid, object_id oid, object_sub_id integer)
  Gets identity info of a database object.

```
SELECT pg_identify_object(1255, 16716, 0);

Result:
          pg_identify_object
--------------------------------------
 (function,public,,"public.getfoo()")
                               (1 row)
```

- pg_identify_object_as_address(catalog_id oid, object_id oid, object_sub_id integer)
  Gets external representation of a database object's address.

```
SELECT pg_identify_object_as_address(1255, 16716, 0);

Result:
  pg_identify_object_as_address
---------------------------------
 (function,"{public,getfoo}",{})
                          (1 row)
```

- pg_get_object_address(type text, name text[], args text[])
  Gets address of a database object, from its external representation.

```
SELECT pg_get_object_address('type', '{public.comp}', '{}');

Result:
 pg_get_object_address
-----------------------
     (1247,17063,0)
                (1 row)
```

- col_description(table_oid, column_number)
  Gets comment for a table column.

```
SELECT col_description(17064, 1);

Result:
 col_description
-----------------
   code_number
          (1 row)
```

- obj_description(object_oid, catalog_name)
  Gets comment for a database object.

```
SELECT obj_description(16887, 'pg_trigger');

Result:
  obj_description
--------------------
 comment on trigger
             (1 row)
```

- obj_description(object_oid)
  Gets comment for a database object (no longer used).

```
SELECT obj_description(16887);

Result:
  obj_description
--------------------
 comment on trigger
             (1 row)
```

- shobj_description(object_oid, catalog_name)
  Gets comment for a shared database object.

```
SELECT shobj_description(1262,'pg_database');
```

```
Result:
 shobj_description
-------------------


            (1 row)
```

- txid_current()
  Gets current transaction ID, assigning a new one if the current transaction does not have one.

```
SELECT txid_current();

Result:
 txid_current
--------------
         2061
       (1 row)
```

- txid_current_snapshot()
  Gets current snapshot.

```
SELECT txid_current_snapshot();

Result:
 txid_current_snapshot
-----------------------
      2062:2062:
                (1 row)
```

- txid_snapshot_xip(txid_snapshot)
  Gets in-progress transaction IDs in snapshot.

```
SELECT txid_snapshot_xip('2095:2095:');

Result:
 txid_snapshot_xip
-------------------


            (1 row)
```

- txid_snapshot_xmax(txid_snapshot)
  Gets xmax of snapshot.

```
SELECT txid_snapshot_xmax('2094:2095:');

Result:
 txid_snapshot_xmax
--------------------
               2095
             (1 row)
```

- txid_snapshot_xmin(txid_snapshot)
  Gets xmin of snapshot.

```
SELECT txid_snapshot_xmin('2094:2095:');

Result:
 txid_snapshot_xmin
--------------------
               2094
             (1 row)
```

- txid_visible_in_snapshot(bigint, txid_snapshot)
  Returns whether transaction ID is visible in snapshot (do not use with subtransaction ids).

```
SELECT txid_visible_in_snapshot(2099, '2100:2100:');

Result:
 txid_visible_in_snapshot
--------------------------
            t
                   (1 row)
```

- pg_xact_commit_timestamp(xid)
  Gets commit timestamp of a transaction (`track_commit_timestamp` parameter should be set to on).

```
SELECT pg_xact_commit_timestamp('2097'::xid);

Result:
   pg_xact_commit_timestamp
-------------------------------
 2017-10-18 13:38:09.738211+09
                        (1 row)
```

- pg_last_committed_xact()
  Gets transaction ID and commit timestamp of latest committed transaction (`track_commit_timestamp` parameter should be set to on).

```
SELECT pg_last_committed_xact();
```

```
Result:
         pg_last_committed_xact
----------------------------------------
 (2097,"2017-10-18 13:38:09.738211+09")
                               (1 row)
```

- pg_control_checkpoint()
  Returns information about current checkpoint state.

```
SELECT pg_control_checkpoint();

Result:
                     pg_control_checkpoint

-------------------------------------------------------------------
 (0/1D4B0D0,0/1D4B038,0/1D4B0D0,000000010000000000000001,
 1,1,t,0:2063,24576,1,0,1751,1,0,1,1,0,0,"2017-10-16 16:26:21+09")
                                                            (1 row)
```

- pg_control_system()
  Returns information about current control file state.

```
SELECT pg_control_system();

Result:
                      pg_control_system
--------------------------------------------------------------
 (960,201608131,6469891178207434037,"2017-10-16 16:26:21+09")
                                                       (1 row)
```

- pg_control_init()
  Returns information about cluster initialization state.

```
SELECT pg_control_init();

Result:
                    pg_control_init
-------------------------------------------------------
 (8,8192,131072,8192,16777216,64,32,1996,2048,t,t,t,0)
                                                (1 row)
```

- pg_control_recovery()
  Returns information about recovery state.

```
SELECT pg_control_recovery();
```

```
Result:
 pg_control_recovery
---------------------
  (0/0,0,0/0,0/0,f)
              (1 row)
```

System Administration Functions

- current_setting(setting_name [, missing_ok])
  Gets current value of setting.

```
SELECT current_setting('datestyle');

Result:
 current_setting
-----------------
 ISO, YMD
        (1 row)
```

- set_config(setting_name, new_value, is_local)
  Sets parameter and returns new value.

```
SELECT set_config('log_statement_stats', 'off', false);

Result:
 set_config
------------
    off
     (1 row)
```

- pg_cancel_backend(pid int)
  Cancels a backend's current query. This is also allowed if the calling role is a member of the role whose backend is being canceled or the calling role has been granted pg_signal_backend. However, only superusers can cancel superuser backends.

```
SELECT pg_cancel_backend(30819);
Error: Cancel operation by user request.
```

- pg_reload_conf()
  Causes server processes to reload their configuration files.

```
SELECT pg_reload_conf();

Result:
 pg_reload_conf
----------------
```

```
           t
             (1 row)
```

- pg_rotate_logfile()
  Signals the log-file manager to switch to a new log file immediately (This works only when the built-in log collector is running).

```
SELECT pg_rotate_logfile();

 pg_rotate_logfile
-------------------
         f
            (1 row)
```

- pg_terminate_backend(pid int)
  Terminates a backend. This is also allowed if the calling role is a member of the role whose backend is being terminated or the calling role has been granted `pg_signal_backend`. However, only superusers can terminate superuser backends.

```
SELECT pg_terminate_backend(30819);

Result:
Fatal error: Connection is terminated by an administrator request.
             The server suddenly closed the connection.
             This type of processing means the server was abruptly termin
ated
             while or before processing the client's request.
             The server connection has been lost. Attempt to reconnect: S
uccess.
```

- pg_create_restore_point(name text)
  Creates a named point for performing restore (restricted to superusers by default, but other users can be granted EXECUTE to run the function).

```
SELECT pg_create_restore_point( 'important_moment' );

Result:
 pg_create_restore_point
-------------------------
       0/1D72DC0
                  (1 row)
```

- pg_current_xlog_flush_location()
  Returns the transaction log flush location.

```
SELECT pg_current_xlog_flush_location();

Result:
 pg_current_xlog_flush_location
---------------------------------
            0/1D72ED8
                          (1 row)
```

- pg_current_xlog_insert_location()
  Returns the location of the current transaction log insert.

```
SELECT pg_current_xlog_insert_location();

Result:
 pg_current_xlog_insert_location
---------------------------------
            0/1D72ED8
                          (1 row)
```

- pg_current_xlog_location()
  Returns the location of the current transaction log write.

```
SELECT pg_current_xlog_location();

Result:
 pg_current_xlog_location
--------------------------
         0/1D72ED8
                   (1 row)
```

- pg_start_backup(label text [, fast boolean [, exclusive boolean]])
  Prepares for performing on-line backup (restricted to superusers by default, but other users can be granted EXECUTE to run the function).

```
SELECT pg_start_backup('my_backup', true, false);

Result:
 pg_start_backup
-----------------
     0/2000028
          (1 row)
```

- pg_stop_backup()
  Finishes performing exclusive on-line backup (restricted to superusers by default, but other users can be granted EXECUTE to run the function).

```
SELECT pg_stop_backup();

Result:
NOTICE: The pg_stop_backup operation is finished.
        All necessary WAL pieces have been archived.
 pg_stop_backup
----------------
 (0/50000F8,,)
         (1 row)
```

- pg_stop_backup(exclusive boolean)
  Finishes performing exclusive or non-exclusive on-line backup (restricted to superusers by default, but other users can be granted EXECUTE to run the function).

```
SELECT pg_stop_backup(false);

Result:
NOTICE:  WAL archiving is not enabled; you must ensure that all required
WAL
         segments are copied through other means to complete the backup

                                             pg_stop_backup

-------------------------------------------------------------------------
--
 (0/3000088,"START WAL LOCATION: 0/2000028 (file 000000010000000000000002)
+
 CHECKPOINT LOCATION: 0/2000060
 +
 BACKUP METHOD: streamed
 +
 BACKUP FROM: master
 +
 START TIME: 2017-10-17 10:00:18 KST
 +
 LABEL: my_backup
 +
 ","17060 /home/agens/AgensGraph/db_cluster/data
 +
 ")
                                                                   (1 ro
w)
```

- pg_is_in_backup()
  Returns true if an on-line exclusive backup is still in progress.

```
SELECT pg_is_in_backup();

Result:
 pg_is_in_backup
-----------------
        t
          (1 row)
```

- pg_backup_start_time()
  Gets start time of an on-line exclusive backup in progress.

```
SELECT pg_backup_start_time();

Result:
  pg_backup_start_time
------------------------
 2017-10-17 10:29:26+09
                 (1 row)
```

- pg_switch_xlog()
  Forces switch to a new transaction log file (restricted to superusers by default, but other users can be granted EXECUTE to run the function).

```
SELECT pg_switch_xlog();

Result:
 pg_switch_xlog
----------------
    0/9000120
         (1 row)
```

- pg_xlogfile_name(location pg_lsn)
  Converts the transaction log location string to file name.

```
SELECT pg_xlogfile_name('0/9000028');

Result:
     pg_xlogfile_name
--------------------------
 000000010000000000000009
                   (1 row)
```

- pg_xlogfile_name_offset(location pg_lsn)
  Converts the transaction log location string to file name and decimal byte offset within file.

```
SELECT pg_xlogfile_name_offset('0/9000028');

Result:
    pg_xlogfile_name_offset
-------------------------------
 (000000010000000000000009,40)
                        (1 row)
```

- pg_xlog_location_diff(location pg_lsn, location pg_lsn)
  Calculates the difference between two transaction log locations.

```
SELECT pg_xlog_location_diff('0/9000120', '0/9000028');

Result:
 pg_xlog_location_diff
-----------------------
                (1 row)
```

- pg_is_in_recovery()
  Returns true if recovery is still in progress.

```
SELECT pg_is_in_recovery();

Result:
 pg_is_in_recovery
-------------------
        t
            (1 row)
```

- pg_last_xlog_receive_location()
  Gets the last transaction log location received and synced to disk by streaming replication. While streaming replication is in progress this will increase monotonically. If recovery has completed this will remain static at the value of the last WAL record received and synced to disk during recovery. If streaming replication is disabled, or if it has not yet started, the function returns NULL.

```
SELECT pg_last_xlog_receive_location();

Result:
 pg_last_xlog_receive_location
-------------------------------

                        (1 row)
```

- pg_last_xlog_replay_location()
  Gets the last transaction log location replayed during recovery. If recovery is still in progress this will increase monotonically. If recovery has completed then this value will remain static at the value of the last WAL record applied during that recovery. When the server has been started normally without recovery the function returns NULL.

```
SELECT pg_last_xlog_replay_location();

Result:
 pg_last_xlog_replay_location
------------------------------

                       (1 row)
```

- pg_last_xact_replay_timestamp()
  Gets timestamp of last transaction replayed during recovery. This is the time at which the commit or abort WAL record for that transaction was generated on the primary. If no transactions have been replayed during recovery, this function returns NULL. Otherwise, if recovery is still in progress this will increase monotonically. If recovery has completed then this value will remain static at the value of the last transaction applied during that recovery. When the server has been started normally without recovery the function returns NULL.

```
SELECT pg_last_xact_replay_timestamp();

Result:
 pg_last_xact_replay_timestamp
-------------------------------

                        (1 row)
```

- pg_export_snapshot()
  Saves the current snapshot and returns its identifier.

```
SELECT pg_export_snapshot();

Result:
 pg_export_snapshot
--------------------
     00000816-1
             (1 row)
```

- pg_create_physical_replication_slot(slot_name name [, immediately_reserve boolean])
  Creates a new physical replication slot named `slot_name`. The optional second

parameter, when `true`, specifies that the LSN for this replication slot be reserved immediately; otherwise the LSN is reserved on first connection from a streaming replication client. Streaming changes from a physical slot is only possible with the streaming-replication protocol (see this link for more technical info). This function corresponds to the replication protocol command `CREATE_REPLICATION_SLOT ... PHYSICAL`.

```
SELECT pg_create_physical_replication_slot('test_slot', 'true');

Result:
 pg_create_physical_replication_slot
-------------------------------------
        (test_slot,0/D000220)
                                  (1 row)
```

- pg_drop_replication_slot(slot_name name)
  Drops the physical or logical replication slot named `slot_name`. Same as replication protocol command
  `DROP_REPLICATION_SLOT`.

```
SELECT pg_drop_replication_slot('test_slot');

Result:
 pg_drop_replication_slot
--------------------------


                   (1 row)
```

- pg_create_logical_replication_slot(slot_name name, plugin name)
  Creates a new logical (decoding) replication slot named `slot_name` using the output `plugin` plugin. A call to this function has the same effect as the replication protocol command `CREATE_REPLICATION_SLOT ... LOGICAL`.

```
SELECT pg_create_logical_replication_slot('test_slot', 'test_decoding');

Result:
 pg_create_logical_replication_slot
------------------------------------
       (test_slot,0/D000338)
                                 (1 row)
```

- pg_logical_slot_get_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])
  Returns changes in the slot `slot_name`, starting from the point at which since changes have been consumed last. If `upto_lsn` and `upto_nchanges` are NULL, logical decoding

will continue until end of WAL. If `upto_lsn` is non-NULL, decoding will include only those transactions which commit prior to the specified LSN.
If `upto_nchanges` is non-NULL, decoding will stop when the number of rows produced by decoding exceeds the specified value. Note, however, that the actual number of rows returned may be larger, since this limit is only checked after adding the rows produced when decoding each new transaction commit.

```
SELECT pg_logical_slot_get_changes('regression_slot',null, null);

Result:
  pg_logical_slot_get_changes
--------------------------------
 (0/F000190,2079,"BEGIN 2079")
 (0/F028360,2079,"COMMIT 2079")
                        (2 row)
```

- pg_logical_slot_peek_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])
  Behaves just like the `pg_logical_slot_get_changes()` function, except that changes are not consumed.

```
SELECT pg_logical_slot_peek_changes('regression_slot',null, null);

Result:
                        pg_logical_slot_peek_changes

-------------------------------------------------------------------------
---
 (0/F028398,2080,"BEGIN 2080")
 (0/F028468,2080,"table public.data: INSERT: id[integer]:1 data[text]:'3'
")
 (0/F028568,2080,"COMMIT 2080")
                                                                    (3 r
ow)
```

- pg_logical_slot_get_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])
  Behaves just like the `pg_logical_slot_get_changes()` function, except that changes are returned as bytea.

```
SELECT pg_logical_slot_get_binary_changes('regression_slot',null, null);

Result:
               pg_logical_slot_get_binary_changes
-------------------------------------------------------------------
```

```
 (0/F028398,2080,"\\x424547494e2032303830")
 (0/F028468,2080,"\\x7461626c65207075626c69632e646174613a20494e5345
52543a2069645b696e74656765725d3a3120646174615b746578745d3a273327")
 (0/F028568,2080,"\\x434f4d4d49542032303830")
                                                             (3 row)
```

- pg_logical_slot_peek_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])
  Behaves just like the `pg_logical_slot_get_changes()` function, except that changes are returned as bytea and that changes are not consumed.

```
SELECT pg_logical_slot_peek_binary_changes('regression_slot',null, null);

Result:
               pg_logical_slot_peek_binary_changes
--------------------------------------------------------------------
 (0/F028398,2080,"\\x424547494e2032303830")
 (0/F028468,2080,"\\x7461626c65207075626c69632e646174613a20494e5345
52543a2069645b696e74656765725d3a3120646174615b746578745d3a273327")
 (0/F028568,2080,"\\x434f4d4d49542032303830")
                                                             (3 row)
```

- pg_replication_origin_create(node_name text)
  Creates a replication origin with the given external name, and returns the internal id assigned to it.

```
SELECT pg_replication_origin_create('test_decoding: regression_slot');

Result:
 pg_replication_origin_create
------------------------------
                            1
                       (1 row)
```

- pg_replication_origin_drop(node_name text)
  Deletes a previously created replication origin, including any associated replay progress.

```
SELECT pg_replication_origin_drop('test_decoding: temp');

Result:
 pg_replication_origin_drop
----------------------------

                     (1 row)
```

- pg_replication_origin_oid(node_name text)
  Look ups a replication origin by name and returns the internal id. If no corresponding replication origin is found an error is thrown.

```
SELECT pg_replication_origin_oid('test_decoding: temp');

Result:
 pg_replication_origin_oid
---------------------------
                         2
                    (1 row)
```

- pg_replication_origin_session_setup(node_name text)
  Marks the current session as replaying from the given origin, allowing replay progress to be tracked.
  Use `pg_replication_origin_session_reset` to revert. Can only be used if no previous origin is configured.

```
SELECT pg_replication_origin_session_setup('test_decoding: regression_slo
t');

Result:
 pg_replication_origin_session_setup
-------------------------------------

                              (1 row)
```

- pg_replication_origin_session_reset()
  Cancels the configuration of pg_replication_origin_session_setup().

```
SELECT pg_replication_origin_session_reset();

Result:
 pg_replication_origin_session_reset
-------------------------------------

                              (1 row)
```

- pg_replication_origin_session_is_setup()
  Returns whether a replication origin has been configured in the current session.

```
SELECT pg_replication_origin_session_is_setup();

Result:
 pg_replication_origin_session_is_setup
```

```
----------------------------------------
                  t
                                   (1 row)
```

- pg_replication_origin_session_progress(flush bool)
  Returns the replay location for the replication origin configured in the current session. The parameter `flush` determines whether the corresponding local transaction will be guaranteed to have been flushed to disk or not.

```
SELECT pg_replication_origin_session_progress(false);

Result:
 pg_replication_origin_session_progress
----------------------------------------
              0/AABBCCDD
                                   (1 row)
```

- pg_replication_origin_xact_setup(origin_lsn pg_lsn, origin_timestamp timestamptz)
  Current transaction as replaying a transaction that has committed at the given LSN and timestamp. Can only be called when a replication origin has previously been configured using `pg_replication_origin_session_setup()`.

```
SELECT pg_replication_origin_xact_setup('0/AABBCCDD', '2017-01-01 00:00');

Result:
 pg_replication_origin_xact_setup
----------------------------------


                             (1 row)
```

- pg_replication_origin_xact_reset()
  Cancels the configuration of `pg_replication_origin_xact_setup()`.

```
SELECT pg_replication_origin_xact_reset();

Result:
 pg_replication_origin_xact_reset
----------------------------------


                             (1 row)
```

- pg_replication_origin_advance(node_name text, pos pg_lsn)
  Sets replication progress for the given node to the given location. This primarily is useful for setting up the initial location or a new location after configuration changes

and similar. Be aware that careless use of this function can lead to inconsistently replicated data.

```
SELECT pg_replication_origin_advance('test_decoding: regression_slot', '0
/1');

Result:
 pg_replication_origin_advance
-------------------------------

                        (1 row)
```

- pg_replication_origin_progress(node_name text, flush bool)
  Returns the replay location for the given replication origin. The parameter `flush` determines whether the corresponding local transaction will be guaranteed to have been flushed to disk or not.

```
SELECT pg_replication_origin_progress('test_decoding: temp', true);

Result:
 pg_replication_origin_progress
--------------------------------
           0/AABBCCDD
                         (1 row)
```

- pg_logical_emit_message(transactional bool, prefix text, content text)
  Emits a text logical decoding message. This can be used to pass generic messages to logical decoding plugins through WAL. The parameter `transactional` specifies if the message should be part of current transaction or if it should be written immediately and decoded as soon as the logical decoding reads the record. The prefix is textual `prefix` used by the logical decoding plugins to easily recognize interesting messages for them. The `content` is the text of the message.

```
SELECT pg_logical_emit_message(false, 'test', 'this message will not be d
ecoded');

Result:
 pg_logical_emit_message
-------------------------
        0/F05E1D0
                  (1 row)
```

- pg_logical_emit_message(transactional bool, prefix text, content bytea)
  Emits binary logical decoding message. This can be used to pass generic messages to logical decoding plugins through WAL. The parameter `transactional` specifies if the

message should be part of current transaction or if it should be written immediately and decoded as soon as the logical decoding reads the record. The prefix is textual `prefix` used by the logical decoding plugins to easily recognize interesting messages for them. The `content` is the binary content of the message.

```
SELECT pg_logical_emit_message(false, 'test', '0/F05E1D0');

Result:
 pg_logical_emit_message
-------------------------
        0/F05E2C8
                  (1 row)
```

- pg_column_size(any)
  Returns the number of bytes used to store a particular value.

```
SELECT pg_column_size('SELECT fooid FROM foo');

Result:
 pg_column_size
----------------
             22
         (1 row)
```

- pg_database_size(oid)
  Returns disk space used by the database with the specified OID.

```
SELECT pg_database_size(16482);

Result:
 pg_database_size
------------------
          9721508
           (1 row)
```

- pg_database_size(name)
  Returns disk space used by the database with the specified name.

```
SELECT pg_database_size('test');

Result:
 pg_database_size
------------------
          9721508
           (1 row)
```

- pg_indexes_size(regclass)
  Returns total disk space used by indexes attached to the specified table.

```
SELECT pg_indexes_size(2830);

Result:
 pg_indexes_size
-----------------
            8192
          (1 row)
```

- pg_relation_size(relation regclass, fork text)
  Returns disk space used by the specified fork ('main', 'fsm', 'vm', or 'init') of the specified table or index.

```
SELECT  pg_relation_size(16881, 'main');

Result:
 pg_relation_size
------------------
                0
           (1 row)
```

- pg_relation_size(relation regclass)
  Shorthand for `pg_relation_size(..., 'main')`.

```
SELECT  pg_relation_size(16881);

Result:
 pg_relation_size
------------------
                0
           (1 row)
```

- pg_size_bytes(text)
  Converts a size in human-readable format with size units into bytes.

```
SELECT pg_size_bytes('100');

Result:
 pg_size_bytes
---------------
           100
        (1 row)
```

- pg_size_pretty(bigint)
  Converts a size in bytes expressed as a 64-bit integer into a human-readable format with size units.

```
SELECT pg_size_pretty(10::bigint);

Result:
 pg_size_pretty
----------------
       10 bytes
         (1 row)
```

- pg_size_pretty(numeric)
  Converts a size in bytes expressed as a numeric value into a human-readable format with size units.

```
SELECT pg_size_pretty(10::numeric);

Result:
 pg_size_pretty
----------------
       10 bytes
         (1 row)
```

- pg_table_size(regclass)
  Returns disk space used by the specified table, excluding indexes (but including TOAST, free space map, and visibility map).

```
SELECT pg_table_size('myschema.mytable');

Result:
 pg_table_size
---------------
          8192
        (1 row)
```

- pg_tablespace_size(oid)
  Returns disk space used by the tablespace with the specified OID.

```
SELECT pg_tablespace_size(1663);

Result:
 pg_tablespace_size
--------------------
           40859636
             (1 row)
```

- pg_tablespace_size(name)
  Returns disk space used by the tablespace with the specified name.

```
SELECT pg_tablespace_size('pg_default');

Result:
 pg_tablespace_size
--------------------
           40859636
             (1 row)
```

- pg_total_relation_size(regclass)
  Returns total disk space used by the specified table, including all indexes and TOAST data.

```
SELECT  pg_total_relation_size(16881);

Result:
 pg_total_relation_size
------------------------
                   8192
                 (1 row)
```

- pg_relation_filenode(relation regclass)
  Returns the filenode number of the specified relation.

```
SELECT pg_relation_filenode('pg_database');

Result:
 pg_relation_filenode
----------------------
                 1262
               (1 row)
```

- pg_relation_filepath(relation regclass)
  Returns file path name of the specified relation.

```
SELECT pg_relation_filepath('pg_database');

Result:
 pg_relation_filepath
----------------------
          global/1262
               (1 row)
```

- pg_filenode_relation(tablespace oid, filenode oid)
  Finds the relation associated with a given tablespace and filenode.

```
SELECT pg_filenode_relation(1663, 16485);

Result:
 pg_filenode_relation
----------------------
  test.ag_label_seq
               (1 row)
```

- brin_summarize_new_values(index regclass)
  Summarizes page ranges not already summarized.

```
SELECT brin_summarize_new_values('brinidx');

Result:
 brin_summarize_new_values
---------------------------
                         0
                    (1 row)
```

- gin_clean_pending_list(index regclass)
  Moves GIN pending list entries into main index structure.

```
SELECT gin_clean_pending_list('gin_test_idx');

Result:
 gin_clean_pending_list
------------------------
                      0
                 (1 row)
```

- pg_ls_dir(dirname text [, missing_ok boolean, include_dot_dirs boolean])
  Lists the content of a directory.

```
SELECT pg_ls_dir('.');

Result:
      pg_ls_dir
----------------------
 pg_xlog
 global
 ...
              (28 row)
```

- pg_read_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])
  Returns the content of a text file.

```
SELECT pg_read_file('test.sql');

Result:
 pg_read_file
--------------
 test        +
      (1 row)
```

- pg_read_binary_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])
  Returns the content of a file.

```
SELECT pg_read_binary_file('test');

Result:
 pg_read_binary_file
---------------------
     x6161610a
              (1 row)
```

- pg_stat_file(filename text[, missing_ok boolean])
  Returns information about a file.

```
SELECT pg_stat_file('test');

Result:
                                   pg_stat_file

--------------------------------------------------------------------------
----------
 (4,"2017-10-18 11:05:09+09","2017-10-18 11:04:55+09","2017-10-18 11:04:5
5+09",,f)

   (1 row)
```

- pg_advisory_lock(key bigint)
  Obtains exclusive session level advisory lock.

```
SELECT pg_advisory_lock(1);
SELECT locktype, classid, objid, mode FROM pg_locks where objid=1;

Result:
 locktype | classid | objid |     mode
----------+---------+-------+---------------
```

```
 advisory |       0 |     1 | ExclusiveLock
                               (1 row)
```

- pg_advisory_lock(key1 int, key2 int)
  Obtains exclusive session level advisory lock.

```
SELECT pg_advisory_lock(1,2);
SELECT locktype, classid, objid, mode FROM pg_locks where objid=2;

Result:
 locktype | classid | objid |     mode
----------+---------+-------+---------------
 advisory |       1 |     2 | ExclusiveLock
                               (1 row)
```

- pg_advisory_lock_shared(key bigint)
  Obtains shared session level advisory lock.

```
SELECT pg_advisory_lock_shared(10);
SELECT locktype, classid, objid, mode FROM pg_locks where objid=10;

Result:
 locktype | classid | objid |   mode
----------+---------+-------+----------
 advisory |       0 |    10 | ShareLock
                           (1 row)
```

- pg_advisory_lock_shared(key1 int, key2 int)
  Obtains shared session level advisory lock.

```
SELECT pg_advisory_lock_shared(10,20);
SELECT locktype, classid, objid, mode FROM pg_locks where objid=20;

Result:
 locktype | classid | objid |   mode
----------+---------+-------+----------
 advisory |      10 |    20 | ShareLock
                           (1 row)
```

- pg_advisory_unlock(key bigint)
  Releases an exclusive session level advisory lock.

```
SELECT pg_advisory_unlock(1);

Result:
 pg_advisory_unlock
```

```
--------------------
         t
             (1 row)
```

- pg_advisory_unlock(key1 int, key2 int)
  Releases an exclusive session level advisory lock.

```
SELECT pg_advisory_unlock(1,2);

Result:
 pg_advisory_unlock
--------------------
         t
             (1 row)
```

- pg_advisory_unlock_all()
  Releases all session level advisory locks held by the current session.

```
SELECT pg_advisory_unlock_all();

Result:
 pg_advisory_unlock_all
------------------------

                 (1 row)
```

- pg_advisory_unlock_shared(key bigint)
  Releases a shared session level advisory lock.

```
SELECT pg_advisory_unlock_shared(10);

Result:
 pg_advisory_unlock_shared
---------------------------
             t
                    (1 row)
```

- pg_advisory_unlock_shared(key1 int, key2 int)
  Releases a shared session level advisory lock.

```
SELECT pg_advisory_unlock_shared(10,20);

Result:
 pg_advisory_unlock_shared
---------------------------
```

```
                 t
                        (1 row)
```

- pg_advisory_xact_lock(key bigint)
  Obtains exclusive transaction level advisory lock.

```
SELECT pg_advisory_xact_lock(1);

Result:
 pg_advisory_xact_lock
------------------------


                 (1 row)
```

- pg_advisory_xact_lock(key1 int, key2 int)
  Obtains exclusive transaction level advisory lock.

```
SELECT pg_advisory_xact_lock(1,2);

Result:
 pg_advisory_xact_lock
------------------------


                 (1 row)
```

- pg_advisory_xact_lock_shared(key bigint)
  Obtains shared transaction level advisory lock.

```
SELECT pg_advisory_xact_lock_shared(10);

Result:
 pg_advisory_xact_lock_shared
------------------------------


                       (1 row)
```

- pg_advisory_xact_lock_shared(key1 int, key2 int)
  Obtains shared transaction level advisory lock.

```
SELECT pg_advisory_xact_lock_shared(10,20);

Result:
 pg_advisory_xact_lock_shared
------------------------------


                       (1 row)
```

- pg_try_advisory_lock(key bigint)
  Obtains exclusive session level advisory lock if available.

```
SELECT pg_try_advisory_lock(100);

Result:
 pg_try_advisory_lock
----------------------
          t
               (1 row)
```

- pg_try_advisory_lock(key1 int, key2 int)
  Obtains exclusive session level advisory lock if available.

```
SELECT pg_try_advisory_lock(100,200);

Result:
 pg_try_advisory_lock
----------------------
          t
               (1 row)
```

- pg_try_advisory_lock_shared(key bigint)
  Obtains shared session level advisory lock if available.

```
SELECT pg_try_advisory_lock_shared(1000);

Result:
 pg_try_advisory_lock_shared
-----------------------------
              t
                      (1 row)
```

- pg_try_advisory_lock_shared(key1 int, key2 int)
  Obtains shared session level advisory lock if available.

```
SELECT pg_try_advisory_lock_shared(1000,2000);

Result:
 pg_try_advisory_lock_shared
-----------------------------
              t
                      (1 row)
```

- pg_try_advisory_xact_lock(key bigint)
  Obtains exclusive transaction level advisory lock if available.

```
SELECT pg_try_advisory_xact_lock(1000);

Result:
 pg_try_advisory_xact_lock
---------------------------
             t
                    (1 row)
```

- pg_try_advisory_xact_lock(key1 int, key2 int)
  Obtains exclusive transaction level advisory lock if available.

```
SELECT pg_try_advisory_xact_lock(1000,2000);

Result:
 pg_try_advisory_xact_lock
---------------------------
             t
                    (1 row)
```

- pg_try_advisory_xact_lock_shared(key bigint)
  Obtains shared transaction level advisory lock if available.

```
SELECT pg_try_advisory_xact_lock_shared(10000);

Result:
 pg_try_advisory_xact_lock_shared
----------------------------------
                t
                           (1 row)
```

- pg_try_advisory_xact_lock_shared(key1 int, key2 int)
  Obtains shared transaction level advisory lock if available.

```
SELECT pg_try_advisory_xact_lock_shared(10000,20000);

Result:
 pg_try_advisory_xact_lock_shared
----------------------------------
                t
                           (1 row)
```

User-defined function

AgensGraph enables you to create and use functions you need.

As AgensGraph can use SQL and Cypher query statements at the same time, it is easy to create functions using them, and the created functions can be confirmed with `\df` command. The generated functions can also be called using SELECT or RETURN syntax.

You may refer to PostgreSQL documentation for more information on creation and grammars of user-defined functions.

- User-defined function

```
CREATE FUNCTION func_name (integer, integer) RETURNS integer
    AS 'SELECT $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;

SELECT func_name (1, 1);

Result:
 add
-----
   2
(1 row)

RETURN func_name (1, 1);

Result:
 add
-----
   2
(1 row)

DROP FUNCTION func_name (integer, integer);
```

Hybrid Query Language

Introduction

This section explains how to use a SQL query and a Cypher query statement together as shown in the example below.

Through the SQL statement used in the RDB, table and column aggregation, statistical processing, and GDB's Cypher syntax replace the RDB's join operation to support better data queries.

```
CREATE GRAPH SKAI worldwide;
CREATE VLABEL dev;
CREATE (:dev {name: 'someone', year: 2015});
CREATE (:dev {name: 'somebody', year: 2016});

CREATE TABLE history (year, event)
AS VALUES (1996, 'PostgreSQL'), (2016, 'AgensGraph');
```

Syntax

Cypher in SQL

It is possible to use a Cypher syntax inside the FROM clause to utilize the dataset of the vertices or edges stored in the graph DB as a data set in the SQL statement.

Syntax :

```
SELECT [column_name]
FROM ({table_name|SQLquery|CYPHERquery})
WHERE [column_name operator value];
```

It can be used as the following example:

```
SELECT n.name
FROM history, (MATCH (n:dev) RETURN n) AS dev
WHERE history.year > n.year::int;
  name
---------
 someone
(1 row)
```

SQL in Cypher

When querying the content of graph DB through Cypher syntax, it is possible to use Match and Where syntaxes for search by specific data of RDB. However, the resulting dataset in the SQL statement should be configured to return a single row of results.

Syntax :

```
MATCH [table_name]
WHERE (column_name operator {value|SQLquery|CYPHERquery})
RETURN [column_name];
```

It can be used as the following example:

```
MATCH (n:dev)
WHERE n.year < to_jsonb((SELECT year FROM history WHERE event = 'AgensGraph'))
RETURN properties(n) AS n;

                 n
-----------------------------------
 {"name": "someone", "year": 2015}
(1 row)
```

Drivers

Introduction

AgensGraph supports DB connection using a client driver. The SKAI worldwide's official drivers and existing PostgreSQL drivers support various languages.
 AgensGraph officially provides the JDBC and Python drivers.

Usage of the Java Driver

This section describes how the graph data of AgensGraph is processed and handled in Java applications. AgensGraph's JDBC driver is based on the PostgreSQL JDBC driver and allows Java applications to access the AgensGraph database. APIs of the AgensGraph Java driver are very similar with those of the Postgres JDBC driver. The only difference is that the AgensGraph JDBC driver supports the Cypher query language as well as SQL, utilizing graph data (vertices, edges and paths) as data types (Java classes and instances).

How to use the AgensGraph JDBC driver is described throughout this section with examples.

Get the Driver

Download the driver (jar) from AgensGraph JDBC Download or use maven as follows:

```
<dependency>
  <groupId>net.SKAI worldwide</groupId>
  <artifactId>agensgraph-jdbc</artifactId>
  <version>1.4.2</version>
</dependency>
```

Connection

There are two things to consider when connecting to AgensGraph using the Java driver: Class name and connection string, which are to be loaded into the Java driver.

- class name : `net.SKAI worldwide.agensgraph.Driver`.
- Connection string consisting of sub-protocol, server, port, and database.
  - sub-protocol : `jdbc:agensgraph://`.
  - connection string(including sub-protocol) : `jdbc:agensgraph://server:port/database`.

The following code is an example of connection to AgensGraph. Connect to AgensGraph through the Connection object.

```
import java.sql.DriverManager;
import java.sql.Connection;

public class AgensGraphTest {
  public static void main(String[] args) {
  try{
        Class.forName("net.SKAI worldwide.agensgraph.Driver");
        Connection conn = DriverManager.getConnection
        ("jdbc:agensgraph://127.0.0.1:5432/agens","agens","agens");
        System.out.println("connection success");
     } catch (Exception e) {
          e.printStackTrace();
          }
  }
}
```

Retrieving Data

This section describes how to query graph data in AgensGraph using the `MATCH` clause.

The result of the query is a `vertex` of AgensGraph. You may get the attributes by importing the result as a `vertex` object.

```
import net.SKAI worldwide.agensgraph.util.*;
import java.sql.*;
import net.SKAI worldwide.agensgraph.graph.Vertex;

public class AgensGraphSelect {
      public static void main(String[] args) throws SQLException, ClassNotFou
ndException {
        Class.forName("net.SKAI worldwide.agensgraph.Driver");
        Connection conn = DriverManager.getConnection
        ("jdbc:agensgraph://127.0.0.1:5432/agens","agens","agens");

        try{
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery
            ("MATCH (:person {name: 'John'})-[:knows]-(friend:person) RETURN 
friend");
            while (rs.next()) {
              Vertex person = (Vertex) rs.getObject(1);
              System.out.println(person.getString("name"));
                      }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Creating Data

This section describes how to insert graph data into AgensGraph.

The following example creates a vertex with a vlabel named Person. We used `JsonObject` to add the property of the corresponding vertex.

```
import net.SKAI worldwide.agensgraph.util.*;
import java.sql.*;

public class AgensGraphCreate {
      public static void main(String[] args) throws SQLException, ClassNotFou
ndException {

        Class.forName("net.SKAI worldwide.agensgraph.Driver");
        Connection conn = DriverManager.getConnection
        ("jdbc:agensgraph://127.0.0.1:5432/agens","agens","agens");

        try{
            PreparedStatement pstmt = conn.prepareStatement("CREATE (:person
 ?)");
            Jsonb j = JsonbUtil.createObjectBuilder()
                             .add("name", "John")
                             .add("from", "USA")
                             .add("age", 17)
                             .build();
            pstmt.setObject(1, j);
            pstmt.execute();
        } catch(Exception e) {
            e.printStackTrace();
        }
        finally{
            conn.close();
        }
    }
}
```

The final string created:

```
CREATE (:Person {name: 'John', from: 'USA', age: 17})
```

[Reference]
In JDBC, the question mark (?) is a placeholder for the positional parameter of the PreparedStatement. You may be confused with this mark as it is also being used in other sql operators. To avoid confusion, there should be a space between the question mark (?) and a character when used in the prepared statement.

Graph Object Classes

Class	Description
GraphId	A Java class corresponding to AgensGraph graphid type.
Vertex	A Java class corresponding to AgenceGraph vertex type. Supports access to labels and properties.
Edge	A Java class corresponding to AgensGraph edge type. Supports access to labels and properties.
Path	A Java class corresponding to AgensGraph graphpath type. Support how to access path length and vertex/edge in the path.
Jsonb	A Java class corresponding to AgensGraph jsonb type. Vertex and edge use Jsonb to store properties(attributes). Supports access to JSON scalar type, array type, and object types.
JsonbUtil	Provides various ways to create a jsonb object as in the CREATE clause examples above.

Usage of the Python Driver

This section shows how to use the AgensGraph Python driver with examples. Python driver is a Psycopg2 type extension module for AgensGraph, and supports additional data types such as Vertex, Edge and Path to express graph data.

Get the Driver

You can download AgensGraph Python driver from SKAI worldwide website's Download - Developer Resources or our Github's agensgraph-python.

```
git clone https://github.com/SKAI worldwide-oss/agensgraph-python.git
```

Install

```
$ pip install -U pip
$ pip install psycopg2

$ python /path/to/agensgraph/python/setup.py install
```

Connection

This is an example of a connection string for accessing AgensGraph using a Python driver.

```
import psycopg2
import agensgraph

conn = psycopg2.connect("dbname=agens host=127.0.0.1 user=agens")
print "Opened database successfully"
```

Creating Data

This section explains how to insert graph data into AgensGraph. The following code (example) creates a vertex with a vlabel "Person".

```
import psycopg2
import agensgraph

conn = psycopg2.connect("dbname=agens host=127.0.0.1 user=agens")
print "Opened database successfully"

cur = conn.cursor()

cur.execute("DROP GRAPH IF EXISTS test CASCADE")
cur.execute("CREATE GRAPH test")
cur.execute("SET graph_path = test")
cur.execute("CREATE (:person {name: 'John', from: 'USA', age: 17})")
cur.execute("CREATE (:person {name: 'Daniel', from: 'Korea', age: 20})")
cur.execute("MATCH (p:person {name: 'John'}),(k:person{name: 'Daniel'}) CREAT
E (p)-[:knows]->(k)")

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

Retrieving Data

This section explains how to query graph data in AgensGraph using a `MATCH` clause. The result of the query is a vertex of AgensGraph. You can get the properties by importing the result as a vertex object.

```
import psycopg2
import agensgraph

conn = psycopg2.connect("dbname=agens host=127.0.0.1 user=agens")
print "Opened database successfully"

cur = conn.cursor()
cur.execute("SET graph_path = test")
cur.execute("MATCH (:person {name: 'John'})-[:knows]-(friend:person) RETURN f
riend")

friend = cur.fetchone()
print str(friend[0])

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

Procedural language

Procedural language

AgensGraph can also write user-defined functions in languages other than SQL and C. These other languages are generally referred to as procedural languages (PLs). In the case of functions written in procedural languages, the database server cannot interpret the function's source text. Thus, the task is passed to a special handler that knows the details of the language. The handler performs all the tasks, including parsing, syntax analysis, and execution. The handler itself is a C language function that, like any C other functions, is compiled into a shared object and loaded on demand. Current AgensGraph has four procedural languages: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python.

Installing Procedural Languages

Procedural languages must be installed in each database to be used. However, the procedural languages installed in the template1 database will be automatically available in all subsequent databases created, as the entries in template1 are copied by CREATE DATABASE. The database administrator can determine which languages can be used in which databases and can make only certain languages available, if needed.

In the case of languages provided with the standard distribution, you only need to run `CREATE EXTENSION` *language_name* to install them in the current database. Alternatively, you may use the program createlang to do this from the shell command line. For example, to install a language called PL/Python in the `template1` database, see the following example:

```
createlang plpython template1
```

The manual procedure described below is recommended only if you are installing a language that is not packaged into extension.

Manual Procedural Language Installation
Procedural languages can be installed in a database in five steps and must be done by the database superuser. In most cases, the necessary SQL commands can be packaged into an extension's install script and executed using CREATE EXTENSION.

1. The shared object of the language handler should be compiled and installed in an appropriate library directory. It works the same way as building and installing modules with regular user-defined C functions. Often the language handler relies on external libraries that provide the actual programming language engine; in such a case, the share object must be installed.

2. The handler must be declared as a command.

   ```
   CREATE FUNCTION handler_function_name()
       RETURNS language_handler
       AS 'path-to-shared-object'
       LANGUAGE C;
   ```

   The special return type of `language_handler` tells the database system that this function does not return one of the defined SQL data types and cannot be used directly in the SQL statement.

3. Optionally, the language handler can provide an "inline" handler function that executes anonymous code blocks (DO commands) written in this language. If an inline handler function is provided by the language, it must be declared with the following command:

```
CREATE FUNCTION inline_function_name(internal)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C;
```

4. Optionally, the language handler can provide a "validator" function that checks accuracy of the function definition without actually executing it. The validator function is called by CREATE FUNCTION. If the validator function is provided by the language, you must declare it with the following command:

```
CREATE FUNCTION validator_function_name(oid)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C STRICT;
```

5. Finally, PL must be declared with the following command:

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language-name
    HANDLER handler_function_name
    [INLINE inline_function_name]
    [VALIDATOR validator_function_name] ;
```

The optional keyword `TRUSTED` specifies that the language does not grant access rights to data that will not be used by the user. Trusted languages are designed for general database users (i.e. users without superuser privileges) and can safely create functions and trigger procedures. As the PL function is executed within the database server, the `TRUSTED` flag should only be provided for languages that do not allow access to the database server or file system. PL/pgSQL, PL/Tcl, and PL/Perl languages are considered to be trusted. Languages PL/TclU, PL/PerlU, and PL/PythonU are designed to provide unlimited functionality and should not be marked as trusted.

In the default AgensGraph installation, a handler for the PL/pgSQL language is built and installed in the "library" directory. The PL/pgSQL language itself is installed in every database. Although Tcl support is configured and handlers for PL/Tcl and PL/TclU are built and installed in the library directory, the languages themselves are not installed by default in the database. Likewise, even if Perl support is configured, PL/Perl and PL/PerlU handlers are built and installed, Python support is configured, and a PL/PythonU handler is installed, these languages are not installed by default.

PL/pgSQL

Introduction

PL/pgSQL is a loadable procedural language in AgensGraph. The design objective of PL/pgSQL is to be a loadable procedural language with the following features:

- Can be used to create functions and trigger procedures;
- Adds control structures to the SQL language;
- Can perform complex computations;
- Inherits all user defined types, functions and operators;
- Can be defined to be trusted by the server;
- Is easy to use.

Functions created in PL/pgSQL can be used wherever built-in functions can be used. For example, you can create a function that processes complex conditions, and later define the function as an operator or use it in index expression.

In AgensGraph, PL/pgSQL is installed by default. However, as it is still a loadable module, administrators who are strictly security-conscious can remove PL/pgSQL.

Advantages of PL/pgSQL
SQL is a query language used in databases. Although SQL is easy to learn, all SQL statements must be executed separately per statement in the database server.
In other words, a client application sends a query to the database server individually, waits until each query is processed, takes the result, computes it, and then sends the next query to the server. These processes generate internal processing and cause network load if the database and client are on different machines.
As PL/pgSQL is easy to use in procedural languages and SQL is easy to use, you can group queries and operations within a database server and save on client/server communications loads.

- Eliminate unnecessary communications between client and server.
- Clients do not have to hold unnecessary intermediate results or to transfer them between client and server.

- You do not need to do repeated query parsing.

Because of these factors, you can expect a noticeable performance improvement over applications that do not use stored functions.
In addition, PL/pgSQL may use all data types, operations, and functions of SQL.

Supported argument and result data types
Functions written in PL/pgSQL can accept scalar or array data types supported by the server as arguments and can return results. You can also use or return a specified complex type (row type). PL/pgSQL functions can also return records.

PL/pgSQL functions can be declared using the VARIADIC marker to allow arguments of varying numbers. This works in exactly the same way as SQL functions.

PL/pgSQL functions can be declared to accept and return various types, such as `anyelement`, `anyarray`, `anynonarray`, `anyenum`, and `anyrange`. The actual data types handled by the polymorphic function may vary from call to call.

Structure of PL/pgSQL

Functions written in PL/pgSQL are defined in the server by executing CREATE FUNCTION (command) as follows:

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

The function body is simply a literal string associated with CREATE FUNCTION. It is more helpful to use dollar ($) quotes to write function bodies than to use a usual single quotes syntax. If there is no dollar citation mark, you should escape it by doubling single quotes or backslashes of the function body. Almost all examples in this section use dollar quote literals in function bodies.

PL/pgSQL is a block-structured language. The full text of a function definition should be a block. The block is defined as follows:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

Each declaration and each statement within a block ends with a semicolon. A block appearing within other block, as shown above, should have a semicolon after END. However, a semicolon is not required at the end of the function body.

Tip : In general, they mistakenly use a semicolon immediately after BEGIN very often. This is incorrect and causes a syntax error.

Label is needed only if you want to identify a block to use in an EXIT statement or to restrict the name of a variable declared in a block. If label is positioned after END, it must match the label at the beginning of the block. Identifiers, like regular SQL commands, are converted to lower case by default if they do not have double quotes.

Annotations work the same way in PL/pgSQL code as in regular SQL. Two dashes (--) are recognized as a comment (from the beginning to the end of the line). To process a comment as a block, include the comment between /* and */.

Every statement in a statement section of a block can be a sub-block. Subblocks may be used for logical grouping or localization of a small group of variables. A variable declared in a subblock masks a variable of a similar name in an outer block for the duration of the subblock. By specifying a block name, you may access an external variable.

```
CREATE FUNCTION somefunc() RETURNS integer AS $$

DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Pri
nts 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Note : Actually there is a hidden "outer block" surrounding the body of the PL/pgSQL function. This block provides the function's parameter declarations (if any) and special variables such as FOUND. The outer block is labeled with the name of the function. That is, you can limit parameters and special variables to the name of the function.

It is important that using BEGIN/END to group statements in PL/pgSQL is not to be confused with similarly-named SQL commands for transaction control. BEGIN/END in PL/pgSQL is only used for grouping; it does not start/end any transaction. Functions and trigger procedures are always executed within a transaction set by an outer query. You cannot start or commit a transaction because there is no context to start the transaction. However, you may construct a sub-transaction that can be rolled back without affecting external transactions since a block contains an EXCEPTION clause.

Declarations

All variables used in a block should be declared in the declaration section of the block. (The only exception is that a loop variable in a FOR loop that repeats a range of integers is automatically declared as an integer variable, and a loop variable in a FOR loop that queries the result of cursor is automatically declared as a record variable.) PL/pgSQL variables can have any SQL data type, such as `integer`, `varchar`, and `char`.

Below are some examples of variable declarations:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

The following is a general syntax of a variable declaration:

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | :
= | = } expression ];
```

If a `DEFAULT` clause is given, it specifies the initial value assigned to the variable at block input. If a `DEFAULT` clause is not provided, the variable is initialized to the SQL null value. The `CONSTANT` option prevents variables from being allocated after initialization so that the value remains constant for the duration of the block. The `COLLATE` option specifies collation to use for variables. If `NNOT NULL` is specified, a runtime error occurs if a null value is assigned at execution. All variables declared as `NOT NULL` should specify a non-null default value. The equal sign (=) can be used in place of ":=", which is compatible with PL/SQL.

The default value of a variable is evaluated and assigned to a variable whenever a block is entered (not once per function call). For example, if you assign `now()` to a variable of type `timestamp`, the function will have the current function call time, not the precompiled time.

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

Function parameter declaration
The parameters passed to the function are named identifiers $1, $2, and so on. Optionally, you can declare an alias for the $*n* parameter name to improve readability. You may then use the alias or numeric identifier to refer to the parameter value.

There are two ways to create an alias. Naming a parameter in `CREATE FUNCTION` command is a preferred way.

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Another way is to explicitly declare the alias using declaration syntax.

```
name ALIAS FOR $n;
```

Below is an example of this style:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Note: These two examples are not exactly the same. In the first case, `subtotal` can be referenced as `sales_tax.subtotal`, but in the second case it cannot be marked as `subtotal`. (If you attach label to an inner block, the partial label can be confined to that label instead.

When a PL/pgSQL function is declared as an output parameter, the output parameter has a $*n* name and an optional alias in the same way as a normal input parameter. The output parameter effectively represents a variable that starts with NULL. This should be assigned during execution of the function. The final value of the parameter is the value to be returned. For example, the sales-tax example could be:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Output parameters are most useful when returning multiple values. Here is a simple example:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

This effectively creates an anonymous record type for the result of the function. If a RETURNS clause is provided, a RETURNS record must be specified.

Another way to declare a PL/pgSQL function is to use RETURNS TABLE. For example:

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

This is exactly the same as declaring one or more OUT parameters and specifying RETURNS SETOF *sometype*.

Alias
You can declare alias for all variables as well as function parameters. Alias is used, in an actual case, to assign a different name to a variable with a predefined name, such as NEW or OLD, in the trigger procedure. For example:

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

Since ALIAS creates two methods of naming the same object, its unrestricted use can be confusing. It is best to use ALIAS only for the purpose of ignoring predefined names.

Expressions

All expressions used in PL/pgSQL statements are processed using the server's main SQL launcher. The PL/pgSQL statements can be written as follows:

```
IF expression THEN ...
```

PL/pgSQL evaluates expressions by providing the same query as the main SQL engine. During configuration of SELECT command, the PL/pgSQL variable name is changed to a parameter. This allows you to prepare a query plan for SELECT once and then reuse it for subsequent evaluations using different values of the variable. For example, if you write two integer variables x and y as

```
IF x < y THEN ...
```

it is used as follows:

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

This prepared statement is performed with the value of the PL/pgSQL variable provided each time the IF statement is executed. In general, specific information like this is not that useful for PL/pgSQL users. However, it is worth knowing in case of a problem diagnosis.

Basic Statements

This section and the subsequent sections describe all statement types that PL/pgSQL explicitly understands. Those that are not recognized as part of these statement types are considered SQL commands and are sent to the underlying database engine for execution.

Assignment
To assign a value to a PL/pgSQL variable:

```
variable { := | = } expression;
```

As mentioned earlier, the expressions in these statements are obtained using SQL `SELECT` (command) sent to the underlying database engine. An expression should have a single value (it can be a row value if the variable is a row or record variable). The target variable can be a simple variable (optionally defined by a block name), a row or record variable field, or an array element that is a simple variable or field. The equal sign (=) can be used in place of ":=", which is compatible with PL/SQL.

If the result data type of the expression does not match the data type of the variable or does not match a specific length or precision, the PL/pgSQL interpreter attempts to implicitly convert the result value using the result type of the output function and the variable type of the input function. If the string form of the result value is of a type that is not allowed in the input function, then the input function may generate a run-time error.

For example:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

Executing a command without a result
For SQL commands that do not return a row (for example, an `INSERT` without a `RETURNING` clause), you can execute the commands within a PL/pgSQL function simply by writing them.

The PL/pgSQL variable names that appear in the command text are treated as parameters, and the current values of the variables are provided as the parameter values at runtime. This is exactly the same as the process described hereinabove for expressions.

When you execute an SQL command in this way, PL/pgSQL can cache and reuse the command execution plan.

Calling a function without a useful result value can be sometimes useful for evaluating expressions or SELECT queries, and the results can be ignored. To do this in PL/pgSQL, we recommend you to use the PERFORM statement.

```
PERFORM query;
```

The query is then executed and the results are discarded. You should write your query the same way you would write an SQL SELECT command, and replace the initial keyword SELECT with PERFORM. For WITH queries, use PERFORM and put the query into parentheses (in this case, the query can only return one row). As with commands that do not return results, the PL/pgSQL variable is replaced by the query, and the plan is cached in the same manner. The special variable FOUND is set to true if the query generated at least one row, and set to false if no rows were generated.

Note: You may expect to get this result by writing your own SELECT, but for now, PERFORM is the only way to do it. An SQL command that can return a row such as SELECT is treated as an error and rejected if it is without an INTO clause described in the next section.

For example:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

Run a query as a single row result
A single row (possibly multiple columns) resulting from an SQL command can be assigned to a record variable, row-type variable, or scalar variable list that you create. This is done by writing a basic SQL command and adding an INTO clause. For example, target can be a record variable, a row variable, or a comma-delimited simple variable, and a list of record/row fields.

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

For commands that do not return rows, the PL/pgSQL variable is replaced with the rest of the query, as described above, and the plan is cached. It works in SELECT (INSERT/UPDATE/DELETE that uses RETURNING) and works according to utility commands that return row set results (e.g. EXPLAIN). Except for the INTO clause, SQL commands are the same as those written outside of PL/pgSQL.

Tip: To create a table as a result of SELECT within a PL/pgSQL function, you should use the CREATE TABLE ... AS SELECT statement.

When using a row or variable list as a target, the result column of a query should exactly match the target structure for the numbers and data types. Otherwise, a runtime error will occur. When a record variable is a target, it automatically configures row types of the query result columns.

The INTO clause may appear at almost any position in an SQL command. It is usually written immediately before or after the *select_expressions* list in SELECT command, or at the end of commands for other command types.

If STRICT is not specified in an INTO clause, the *target* is set to the first row returned by the query, or set to null if the query did not return a row. (The "first row" is not well defined unless you use ORDER BY.) The result row after the first row is discarded. You can check the special FOUND variable to see if the row has been returned.

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

In the case where a STRICT option is specified, the query should return exactly a single row. Otherwise, either NO_DATA_FOUND (no rows) or TOO_MANY_ROWS (two or more rows) is reported as a run-time error. If you want to find certain errors as follows, you may use an exception block.

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

Successful execution of STRICT always sets FOUND to true.

In the case of INSERT/UPDATE/DELETE with RETURNING, PL/pgSQL will report an error for more than one returned row, even if STRICT is not specified. This is because there is no such option as ORDER BY to determine if an affected row should be returned.

In the case where print_strict_params is enabled for a function and an error occurs due to unmet requirements of STRICT, DETAIL of the error message contains information about the parameters passed to the query. You can change the print_strict_params setting for all functions by setting plpgsql.print_strict_params; it is, however, affected only by editing of the subsequence function. It is also possible to enable it on a function-by-function basis using compiler options. For example:

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END
$$ LANGUAGE plpgsql;
```

If unsuccessful, this function may generate an error message as follows:

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL statement
```

Note: The STRICT option matches behavior of SELECT INTO and related statements in Oracle PL/SQL.

Executing dynamic commands
You will often need to create a dynamic command (i.e. a command containing different tables or different data types each time it is executed) within a PL/pgSQL function. The general attempt by PL/pgSQL to cache plans for commands does not work in this scenario. An EXECUTE statement is provided to handle this kind of problem.

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

Here, *command-string* is an expression that creates a string (type text) containing a command to execute. An optional *target* is a record variable, a row variable, or a comma-delimited simple variable, and a record/line field list where the result of the command will be stored. The optional USING expression provides a value to be inserted into the command.

In a calculated command string, PL/pgSQL variables cannot be replaced. All necessary variable values should be inserted into the command string as configured. Alternatively, you can use parameters as described below.

There is also no caching plan for commands executed via EXECUTE. Instead, commands are always scheduled each time the statement is executed. Therefore, the command strings can be generated dynamically within a function to perform actions in other tables and columns.

The INTO clause specifies the result of an SQL command that assigns the results of returned rows. When a row or variable list is provided, it should exactly match the structure of the query result (if a record variable is used, it is automatically configured to match the result structure). If multiple rows are returned, only the first row is assigned to the `INTO` variable; NULL is assigned to the `INTO` variable, if no rows are returned. If the `INTO` clause is not specified, the query result is discarded.

Given a `STRICT` option, an error is reported if the query does not generate exactly a single row.

Command strings may use parameter values referenced in commands as `$1`, `$2`, and so on. These symbols refer to values provided in the `USING` clause. This method is often preferred over inserting data values into a command string as text. That is, the runtime overhead of converting values into text and back can be avoided, and SQL-injection attacks occur less often as it does not require quotes or escapes. For example:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <=
$2'
   INTO c
   USING checked_user, checked_date;
```

Parameter symbols can only be used for data values. To use dynamically-determined table or column names, you must insert the corresponding characters into the command string. For example, if you need to perform the preceding query on a dynamically selected table, you can do the following:

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tabname)
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

A simpler approach will be to use `%I` of `format()` for the table or column name (newline-delimited, concatenated strings).

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE inserted_by = $1 AND inserted <= $2', tabname)
   INTO c
   USING checked_user, checked_date;
```

Another limitation on parameter symbols is that they only work with `SELECT`, `INSERT`, `UPDATE` and `DELETE` commands. In other syntax types (commonly referred to as utility syntax), you should insert values in text, even if they are data values.

As in the first example above, `EXECUTE` with a simple constant command string and `USING` parameters is functionally equivalent to writing commands directly in PL/pgSQL and allowing PL/pgSQL variables to be automatically replaced. The important difference is that `EXECUTE` re-plans the commands at each run to generate a plan associated with the current parameter values. On the other hand, PL/pgSQL may create a general plan and cache it for reuse. In situations where the best planning depends heavily on parameter values, it is a good idea to use EXECUTE to make sure that the general plan is not selected.

SELECT INTO is not currently supported within EXECUTE. Instead, you should issue a generic SELECT command and specify INTO as part of EXECUTE.

Control Structures

The control structure is probably the most useful and important part of PL/pgSQL. The PL/pgSQL control structure allows you to manipulate AgensGraph data in a very flexible and powerful way.

Return from a function
There are two commands that can return data from functions: RETURN and RETURN NEXT.

1. RETURN

```
RETURN expression;
```

RETURN with an expression terminates the function and returns the value of the expression to the caller. This format is used for PL/pgSQL functions that do not return sets.
In a function that returns a scalar type, the result of the expression is automatically converted to the return type of the function as described for the assignment. However, to return a composite (row) value, you should write an expression that correctly conveys the set of columns requested. This may require explicit type conversion.
If you declare a function using an output parameter, you need to write only RETURN without an expression. The current value of the output parameter variable is returned.
If you declare a function that returns void, you can use the RETURN statement to terminate the function early. Do not write expressions after RETURN.
The return value of a function cannot remain undefined. If control reaches the end of the top-level block of the function without using a RETURN statement, a run-time error occurs. However, this restriction does not apply to functions with output parameters and functions that return void. In such a case, the RETURN statement is automatically executed when the top-level block completes.
For example:

```
-- functions returning a scalar type
RETURN 1 + 2;
RETURN scalar_var;

-- functions returning a composite type
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- must cast columns to correct types
```

2. RETURN NEXT and RETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

If a PL/pgSQL function is declared to return SETOF *sometype*, you should follow a slightly-different procedure. In such a case, the individual items to be returned are specified in the order of `RETURN NEXT` or `RETURN QUERY`, and the final `RETURN` command without arguments is used to indicate that execution of the function is complete. `RETURN NEXT` can be used with both scalar and complex data types. The entire "table" of the result is returned as a composite result type. `RETURN QUERY` adds the result of executing the query to the result set of the function. `RETURN NEXT` and `RETURN QUERY` can be mixed freely in a single set-returning function, in which case the results are concatenated.

`RETURN NEXT` and `RETURN QUERY` are not actually returned by a function. You can simply add zero or more rows to the result set of the function. Execution continues with the next statement of the PL/pgSQL function. The result set is built when a sequential `RETURN NEXT` or `RETURN QUERY` command is executed. The last return that should have no argument causes control to terminate the function (or control may reach the end of the function).

`RETURN QUERY` has a variant `RETURN QUERY EXECUTE` that specifies the queries to be executed dynamically. Parameter expressions, like `EXECUTE`, can be inserted into a query string computed via `USING`.

If you declare a function using an output parameter, you should write `RETURN NEXT` without an expression. At each run, the current value of the output parameter variable is stored for the final return of the result row. If there are multiple output parameters, declare a function that returns a SETOF record; if there is only one output parameter of type *sometype*, then you should declare SETOF *sometype* to create a set-returning function using the output parameter.

The following is an example of a function that uses `RETURN NEXT`.

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

Here is an example of a function that uses `RETURN QUERY`.

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);

    -- Since execution is not finished, we can check whether rows were return
ed
    -- and raise exception if not.
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No flight at %.', $1;
    END IF;

    RETURN;
 END
$BODY$
LANGUAGE plpgsql;

-- Returns available flights or raises exception if there are no
-- available flights.
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

Note: As described hereinabove, the current implementation of RETURN NEXT and RETURN QUERY stores the entire result set before the function is returned. In other words, performance can be degraded if a PL/pgSQL function generates a very large result set. Data is written to disk to avoid memory depletion, but the function itself is not returned until the entire result set is created. The point at which the current data begins to be written to disk is controlled by the work_mem configuration variable. Administrators having memory large enough to store a bigger volume of result sets should increase this parameter.

Simple loop

By using the `LOOP`, `EXIT`, `CONTINUE`, `WHILE`, `FOR`, and `FOREACH` statements, you can collate a series of commands so that the PL/pgSQL function can repeat them.

1. LOOP

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

`LOOP` defines an unconditional loop that repeats indefinitely until terminated by an `EXIT` or `RETURN` statement. Optional *`label`* is used by `EXIT` and `CONTINUE` statements within the nested loop to specify the loop to which the statement refers.

2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

If *`label`* is not specified, the innermost loop is terminated and the statement following `END LOOP` is executed next. The given *`label`* should be a label of the current or some outer level of the nested loop or block. Then the named loop or block is terminated and the statement continues after the corresponding END of the loop/block.

If `WHEN` is specified, loop terminates only if *`boolean-expression`* is true. Otherwise, control passes to the statement after `EXIT`.

`EXIT` can be used in any type of loop. It is not limited to the use with an unconditional loop.

When used with a `BEGIN` block, `EXIT` passes control to the next statement after the block terminates. Label should be used for this purpose. `EXIT` without label is not considered to match the `BEGIN` block.

Here is an example:

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;

LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
END LOOP;
```

```
BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT ablock;  -- causes exit from the BEGIN block
    END IF;
    -- computations here will be skipped when stocks > 100000
END;
```

3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

If *label* is not given, the next iteration of the innermost loop begins. That is, loop control is returned (if any) to skip all remaining statements in the loop body and to determine if another loop iteration is needed. If *label* is present, specify label of the loop to be continued.

With `WHEN` specified, the next iteration of the loop starts only if *boolean-expression* is true. Otherwise, control passes to the statement following `CONTINUE`.

`CONTINUE` can be used in any type of loop. It is not limited to the use with an unconditional loop.

Here is an example:

```
LOOP
    -- some computations
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- some computations for count IN [50 .. 100]
END LOOP;
```

4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

The `WHILE` statement repeats a series of statements as long as *boolean-expression* is evaluated to be true. Expressions are checked just before each item in the loop body.

Here is an example:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
```

```
END LOOP;

WHILE NOT done LOOP
    -- some computations here
END LOOP;
```

5. FOR(Integer Variant)

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

This type of `FOR` creates a loop that iterates over a range of integers. Variable *names* are automatically defined as integer types and exist only inside a loop (existing definitions of variable names are ignored in the loop). Two expressions representing the upper and lower bounds of the range are evaluated once when they enter the loop. If a `BY` clause is not specified, the iteration step is 1; if specified, the value specified in the `BY` clause is evaluated once in the loop entry. If `REVERSE` is specified, step value is not added but subtracted after each time of iteration.

Some examples of integer FOR loops:

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

If the lower bound is greater than the upper bound (or less than REVERSE), the loop body is never executed. No error occurs.

When a *label* is connected to a `FOR` loop, the integer loop variable can be referenced with the specified name using that *label*.

Loop through query results
Other types of `FOR` loops allow you to iterate over the query results and manipulate the data accordingly. The syntax is as follows:

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

target is a record variable, a row variable, or a comma-delimited list of scalar variables. *target* is assigned to each row of the query result successively, and each row of the loop body is executed. Here is an example:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing materialized views...';

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- Now "mviews" has one record from cs_materialized_views

        RAISE NOTICE 'Refreshing materialized view %s ...', quote_ident(mview
s.mv_name);
        EXECUTE format('TRUNCATE TABLE %I', mviews.mv_name);
        EXECUTE format('INSERT INTO %I %s', mviews.mv_name, mviews.mv_query);
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

If the loop is terminated by an `EXIT` statement, you can still access the last-assigned row value after the loop.

The *query* used in the `FOR` statement type can be any SQL command that returns a row to the caller. `SELECT` is the most common case, but you can also use `INSERT`, `UPDATE`, or `DELETE` with `RETURNING`. Some utility commands such as `EXPLAIN` also work.

PL/pgSQL variables are replaced by query text and the query plan is cached for possible reuse.

The `FOR-IN-EXECUTE` statement is another way to iterate through rows.

```
[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    statements
END LOOP [ label ];
```

This statement is the same as the previous form, except that the source query is specified in a string expression that is evaluated and rescored in each item of the FOR loop. This allows the programmer to choose the speed of pre-planned queries or flexibility of dynamic queries, just like a normal EXECUTE statement. Like EXECUTE, parameter values can be inserted into dynamic commands via USING.

Another way to specify a query that needs to repeat the result is to declare it as cursor.

Loop through array
The FOREACH loop is very similar to the FOR loop, but it iterates through the elements of the array value, instead of repeating the rows returned by the SQL query. Typically, FOREACH is used to iterate through the components of a composite value expression. Variants that repeat complexes other than arrays can be added later.

A FOREACH statement that iterates through the array:

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

If SLICE is not present or SLICE0 is specified, the loop iterates over the individual elements of the array created by evaluating *expression*. In the *target* variable, each element value is specified in order, and the loop body is executed for each element. The following is an example of iterating over an array element.

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Elements are stored in storage order regardless of the number of dimensions of the array. *target* is usually a single variable, but it can be a list of variables when repeating composite value arrays. In such a case, for each array element, the variables are assigned in consecutive columns of composite values.

If the SLICE value is positive, FOREACH repeats the slice of the array rather than a single element. The SLICE value should be an integer constant that is not greater than the number of dimensions of the array. *target* variable should be an array and receive successive slices of the array value. Each slice is the number of dimensions specified by `SLICE`. The following is an example of repeating a one-dimensional slice.

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

Cursors

Instead of running the entire query at once, you can set *cursors* to encapsulate the query and then read a few lines of the query results at a time. A reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users generally do not have to worry about this as `FOR` loops use cursors internally to avoid memory problems.) More interesting way of use is to return a reference to the cursor where a function was created so that the caller can read the line. By doing so, it provides an efficient way to return a large row set from a function.

Declaring cursor variables

All access to PL/pgSQL cursors is always performed via cursor variables, which are special data type `refcursor`. One way to create a cursor variable is to declare it as a variable of type `refcursor`. Another way to do this is to use the following cursor declaration syntax:

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

When `SCROLL` is specified, the cursor may scroll backward. If `NO SCROLL` is specified, the reverse fetch is denied. If this option is not specified, it depends on the query whether reverse fetching is allowed or not. If *`argument`* is specified, it is a comma-separated list of *`name data type pairs`* and defines the name to be substituted for the parameter value in the specified query. The actual value to replace this name is specified later when the cursor is opened.

Here is an example:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

All three of these variables have a `refcursor` data type, but the first query can be used with all queries, the second query bounds a fully specified query, and the last query has a parameterized query *bound*. (The `key` is replaced with the integer parameter value when the cursor is opened.) The variable `curs1` is called *unbound* because it is not bound to a particular query.

Opening cursors
To use a cursor to search for a row, the cursor should be already *open*. PL/pgSQL supports three types of `OPEN` statements, two of which use unbound cursor variables and the third use bound cursor variables.

Note: Bound cursor variables can be used without explicitly opening cursors.

1. OPEN FOR *query*

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

The cursor variable is opened and the specified query is executed. The cursor can no longer be opened, and it should be declared as an unbound cursor variable (i.e. a simple `refcursor` variable). The query should be `SELECT` or something else that returns a row (such as `EXPLAIN`). Queries are handled in the same way as other PL/pgSQL SQL commands. The PL/pgSQL variable name is replaced, and the query plan is cached for possible reuse. When the PL/pgSQL variable is replaced with a cursor query, the value to be replaced is the value at the time of `OPEN`. Subsequent changes to the variable do not affect the behavior of the cursor. The `SCROLL` and `NO SCROLL` options have the same meaning as in the bound cursors.

Here is an example:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
                                     [ USING expression [, ... ] ];
```

The cursor variable is opened and the specified query is executed. The cursor can no longer be opened, and it should be declared as an unbound cursor variable (i.e. a simple `refcursor` variable). The query is specified in a string expression in the same way as in `EXECUTE`. As always, this provides flexibility; this means that the query plan can vary each time of execution and variable substitution is not performed in the command string. Like EXECUTE, parameter values can be inserted into dynamic commands via format() and USING. The SCROLL and NO SCROLL options have the same meaning as bound cursors.

Here is an example:

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USI
NG keyvalue;
```

In this example, the table name is inserted into the query via format(). As the comparison value of col1 is inserted via USING, no quotes are required.

3. Opening bounding cursors

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

This OPEN format is used when a query is declared to open a cursor variable to which the query is bound. The cursor can no longer be opened. The list of actual argument value expressions can only be used if a cursor is declared to take an argument. These values are replaced in the query.

Query plans for bound cursors are always considered cacheable. There is no EXECUTE in this case. SCROLL and NO SCROLL cannot be specified for OPEN because the cursor's scroll behavior has already been determined.

Argument values can be passed using *positional* or *named* notation. In position notation, all arguments are specified in order. In named notation, the names of each argument are specified and separated from the argument expression using :=. Like the calling function, you can use both positional notation and named notation together.

Here's an example (the cursor declaration example above is used):

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

Variable substitution is performed in the query of a bound cursor, which means there are two ways to pass the value to the cursor: explicitly using the OPEN argument, or implicitly passing the value by referencing the PL/pgSQL variable in the query. However, only variables declared before declaration of the bound cursor can be replaced with variables. In both cases, the value to pass is determined at the time of OPEN. For instance, another way to get the same effect as the curs3 example above is:

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

```
BEGIN
    key := 42;
    OPEN curs4;
```

Using cursors
Once a cursor is open, you can manipulate it using the statements described below.

You do not need to perform this manipulation since it works in the same manner as when the cursor is first opened. A function can return a `refcursor` value and allow the caller to work on the cursor. (Internally, the `refcursor` value is simply a string name of so-called a portal containing the active query for the cursor, and can be assigned to another `refcursor` variable).

All portals are closed implicitly at the end of the transaction. Thus, the value of `refcursor` can be used to reference the open cursor until the end of the transaction.

1. FETCH

```
    FETCH [ direction { FROM | IN } ] cursor INTO target;
```

   `FETCH` fetches the next row as the target of cursor in a row variable, a record variable, or a comma-separated list of simple variables, such as `SELECT INTO`. If there is no next row, the target is set to NULL(s). As with `SELECT INTO`, you can check the special variable `FOUND` to see if you obtained the row.

   The *direction* clause can be one of the variations allowed by SQL FETCH, except that it can fetch a single row (i.e. `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` *count*, `RELATIVE` *count*, `FORWARD`, or `BACKWARD`). Omitting *direction* is equivalent to specifying `NEXT`. If the cursor is not declared or open with `SCROLL` option, the *direction* value that moves backward may fail.

   The *cursor* must be the name of a `refcursor` variable that refers to the open cursor portal.

   Here is an example:

```
    FETCH curs1 INTO rowvar;
    FETCH curs2 INTO foo, bar, baz;
    FETCH LAST FROM curs3 INTO x, y;
    FETCH RELATIVE -2 FROM curs4 INTO x;
```

2. MOVE

```
    MOVE [ direction { FROM | IN } ] cursor;
```

MOVE relocates a cursor without retrieving the data. MOVE works the same as FETCH command except that it only relocates the cursor and does not return the moved rows. As with SELECT INTO, you can check the special variable FOUND to see if there is a next row to move.

The direction clause is used in the same manner as it is used in SQL FETCH command (e.g. NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*, RELATIVE *count*, ALL, FORWARD [*count*|ALL] or BACKWARD [*count*|ALL]). Omitting *direction* is equivalent to specifying NEXT. If the cursor is not declared or open with SCROLL option, the *direction* value that moves backward may fail.
Here is an example:

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

When a cursor is positioned on a table row, it can be identified to update or delete the row with the cursor. There is a restriction on the cursor's queries (especially not grouped queries) and it is best to use FOR UPDATE on the cursor.

Here is an example:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

4. CLOSE

```
CLOSE cursor;
```

CLOSE closes the default portal of the open cursor. It can be used to release cursor variables that can release or reopen resources before the end of a transaction.

Here is an example:

```
CLOSE curs1;
```

5. Returning Cursors
The PL/pgSQL function can return a cursor to the caller. This method is useful when returning multiple rows or columns, especially when returning a very large result set. To do this, the function opens a cursor and returns the cursor name to the caller (or simply opens a cursor using the name of the caller or the name of the portal). The

caller can fetch rows from the cursor. The cursor is closed by the caller or automatically closed when it is closed.

The name of the portal used for a cursor can be specified by the programmer or automatically generated. To specify a portal name, you should first assign a string before the `refcursor` variable is opened. The string value of the `refcursor` variable is used as the name of the default portal in OPEN. However, if the `refcursor` variable is null, `OPEN` automatically creates a name that does not conflict with the existing portal and assigns it to the `refcursor` variable.

Note: Since the bound cursor variable is initialized to a string value representing the name, the portal name is the same as the cursor variable name, unless the programmer ignores it through assignment before opening a cursor. However, as the default value of the unbound cursor variable is initially set to null, it gets a unique, automatically generated name unless ignored.

The following example is a way a caller can supply a cursor name.

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

The following example uses automatic cursor name generation.

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;
```

```
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

The following example is a way to return multiple cursors from a single function.

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

Loop through cursor results

There is a `FOR` statement (example below) that can iterate through the rows returned by the cursor.

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value [,
 ...] ) ] LOOP
    statements
END LOOP [ label ];
```

The cursor variable should be bound to some query when it is declared and **should not be already open**. The FOR statement automatically opens a cursor and closes it when the loop ends. The list of actual argument values should appear only if the cursor is declared to take an argument. This value is replaced in the query in the same way as `OPEN`.

The *recordvar* variable is automatically defined as type record and exists only in the loop (the existing definition of the variable name is ignored in the loop). Each row returned by the cursor is assigned to this record variable and the loop body is executed.

Errors and Messages

Reporting errors and messages
You may use the RAISE statement to report messages and causes errors.

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression
[, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

The *level* option specifies the severity of the error. Allowable levels are DEBUG, LOG, INFO, NOTICE, WARNING, and EXCEPTION (EXCEPTION is the default). EXCEPTION causes an error (usually aborting the current transaction). The other levels generate only messages of different priority levels. Whether messages of a particular priority are reported to the client, or recorded in the server log can be controlled by the log_min_messages and client_min_messages configuration variables.

If there is *level*, you can create a *format* (it must be a simple string literal, not an expression). A format string specifies the error message text to report. Following the format string, an optional argument expression can be inserted after the message. Within the format string, % is replaced by the string expressions of the following optional argument values: To print a literal %, you should write %%. The number of arguments should match the number of % placeholders in the format string; if not, an error will occur while compiling the function.

In this example, the value of v_job_id replaces % of the string.

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

You can attach additional information to the error report by writing an *option = expression* entry after USING. Each *expression* can be a string value expression. The allowable *option* keywords include:

- MESSAGE
  Sets the error message text. This option cannot be used in a RAISE format that includes format strings before USING.

- `DETAIL`
  Provides a detailed error message.

- `HINT`
  Provides a hint message.

- ERRCODE
  Specifies the error code (SQLSTATE) to be reported directly by condition name or with a 5-digit SQLSTATE code.

- COLUMN, CONSTRAINT, DATATYPE, TABLE, SCHEMA
  Provides the names of the related objects.

This example stops a transaction with a given error message and a hint.

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user ID';
```

These two examples show an identical way of setting SQLSTATE.

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

There is a second RAISE statement for which the main argument is the condition name or SQLSTATE to be reported. For example:

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

Another way is to create `RAISE USING` or `RAISE` *`level`* `USING` and put everything else in the `USING` list. The last variant of `RAISE` has no parameters at all. This format can be used only within the `EXCEPTION` clause of a `BEGIN` block. Re-generate the error currently being processed.

If no condition name or SQLSTATE is specified in `RAISE_EXCEPTION (P0001)` command, `RAISE_EXCEPTION (P0001)` is used by default. If the message text is not specified, the message name becomes the condition name or SQLSTATE by default.

Note: If you specify a SQLSTATE code as the error code, you can select an error code that is not limited to the predefined error code(s) but consists of a five-digit number and/or uppercase ASCII characters (except `00000`). As the three zero-terminated error codes are category codes, error codes are generated when they can only be trapped by trapping the entire category.

Verifying assertion

The `ASSERT` statement is a convenient shorthand for inserting debugging checks into PL/pgSQL functions.

```
ASSERT condition [ , message ];
```

`condition` is a Boolean expression that is always expected to be asserted to true. If true, no ASSERT statement is executed; if false or null, an `ASSERT_FAILURE` exception is thrown. (If an error occurs while asserting the `condition`, it is reported as a normal error.)

With an optional `message` given, if `condition` fails, the result (if not null) replaces the default error message text "assertion failed." If the assertion is successful, the `*message*` expression is not performed.

`ASSERT` tests can be enabled or disabled via the configuration parameter `plpgsql.check_asserts`, which uses boolean values; the default is `on`. If this parameter is `off`, the `ASSERT` statement does nothing.

`ASSERT` is intended to detect program bugs, not to report common error conditions. Use the `RAISE` statement described above for reporting errors.

Trigger Procedures

PL/pgSQL can be used to define trigger procedures for data changes or database events. A trigger procedure is created with CREATE FUNCTION (command); it does not have any argument and is declared with a return type of `trigger` (in the case of data change triggers) or `event_trigger` (in the case of database event triggers). A special local variable called `PG_something` is automatically defined to describe the condition that triggered the call.

Data change triggers
A data change trigger is declared as a function with a return type of trigger without arguments. The function should be declared without arguments, even if it is expected to receive the arguments specified in `CREATE TRIGGER`. These arguments are passed through `TG_ARGV` as described below.

When the PL/pgSQL function is called by trigger, several special variables are automatically created in the top-level block; the created items are as follows:

- `NEW`
  Data type `RECORD`; variable that holds new database rows on `INSERT/UPDATE` operations on row-level triggers. This variable is not specified for statement-level triggers and `DELETE` operation.
- `OLD`
  Data type `RECORD`; variable that holds old database row on `UPDATE/DELETE` operation on the row level triggers. This variable is not specified for statement-level triggers and `INSERT` operation.
- `TG_NAME`
  Data type `name`; variable that contains the name of the trigger actually fired.

- TG_WHEN
  Data type text; a string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger definition.

- TG_LEVEL
  Data type text; a string of ROW or STATEMENT, depending on the trigger definition.

- TG_OP
  Data type text; a string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger is actually fired.

- TG_RELID
  Data type oid; the object ID of the table that caused the trigger invocation.

- TG_TABLE_NAME
  Data type name; the name of the table that caused the trigger invocation.

- TG_TABLE_SCHEMA
  Data type name; the schema name of the table that caused the trigger call.

- TG_NARGS
  Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.

- TG_ARGV[]
  Data type text; the argument index of the CREATE TRIGGER statement is zero-based. Invalid indices (less than 0 or greater than or equal to tg_nargs) result in a NULL value.

A trigger function must return NULL or a record/row value that exactly matches the structure of the table on which the trigger was executed.

A row-level trigger triggered by BEFORE can return null to signal the trigger manager to skip the rest of the work for this row (i.e. no subsequent trigger is executed and INSERT/UPDATE/DELETE of this row will not occur). If nonnull is returned, the job advances to the corresponding row value. Returning a row value different from the original value of NEW changes the row to be inserted or updated. Accordingly, if the trigger function does not change the row value and you want the triggering action to succeed normally, NEW (or its corresponding value) should be returned. It is possible to change the row to be saved by modifying a single value directly from NEW and returning a modified NEW or by creating a completely-new record/row to be returned. In the case of before-trigger of DELETE, the return value does not have a direct effect, but should not be null to continue the triggering operation. As NEW is null in DELETE trigger, returning is meaningless in general. A common idiom for DELETE trigger is to return OLD.

An INSTEAD OF trigger (which is always a row-level trigger and can only be used in a view) can return null to indicate that no update has been made; skipping the rest of the work for this row should be possible (e.g. the trigger will not start and the row will not be calculated when it is affected by surrounding INSERT/UPDATE/DELETE). Otherwise, a non-null value should be returned to indicate that the trigger has performed the requested operation. The return value should be NEW in the case of INSERT and UPDATE operations, and the trigger function can be modified to support INSERT RETURNING and UPDATE RETURNING (this affects the row values passed to the subsequent trigger, or is passed by the specific EXCLUDED alias reference contained in the ON CONFLICT DO UPDATE clause of the INSERT statement). For DELETE operation, the return value should be OLD.

The return value of a row-level trigger executed after BEFORE or AFTER is always ignored. It may be null. However, if any of these types of triggers fail, the entire operation can be suspended.

The following example shows an example of a trigger procedure in PL/pgSQL.
This trigger in the example below allows the current user name and time to be stamped on a row whenever a row is inserted or updated in the table. Make sure the employee's name is given and the salary is positive (number).

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when they must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Another way to log changes to a table is to create a new table that holds rows for each insert, update, or delete. This approach can be thought of as auditing any changes of the table. The following example shows an example of an audit trigger procedure in PL/pgSQL. This example trigger causes the row insert, update, or delete in emp table to be written to the emp_audit table. The current time and user name are stamped on the row together with the type of the operation performed.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on em
p,
        -- make use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

A variation on the previous example indicates when each entry was last modified by using view that joins the main table to the audit table. This approach still logs the entire audit trail of changes to the table, but provides a brief view of the audit trail, showing only the last modified timestamp derived from the audit trail for each entry. The following example shows an example of an audit trigger on view in PL/pgSQL.
 This example allows view to use a trigger to update the view and to make inserting, updating, or deleting rows of the view to be written to the `emp_audit` table. The current time and user name are recorded with the type of operation performed, and the view shows the last modification time of each row.

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
```

```
        --
        -- Perform the required operation on emp, and create a row in emp_aud
it
        -- to reflect the change made to emp.
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE PROCEDURE update_emp_view();
```

One of the uses of triggers is to maintain a summary table of other tables. The result summary can be used in place of the original table for a particular query (usually it requires a much shorter execution time). This technique is commonly used in data warehousing where measured or observed data tables (called fact tables) can be very large. The following example shows an example of a PL/pgSQL trigger procedure that maintains a summary table for the data warehouse's fact tables.

The schemas described here are based in part on a **grocery store** example included in **The Data Warehouse Toolkit** by Ralph Kimball.

```
--
-- Main tables - time dimension and sales fact.
```

```
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_ke
y);

--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELE
TE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN
```

```
        -- Work out the increment/decrement amount(s).
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- forbid updates that change the time_key -
            -- (probably not too onerous, as DELETE + INSERT is how most
            -- changes will be made).
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                      OLD.time_key, NEW.time_
key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;

        -- Insert or update the summary row with the new values.

        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;
            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );

                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN
                    -- do nothing
            END;
        END LOOP insert_update;

        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

Event triggers

PL/pgSQL can be used to define event triggers. In AgensGraph, procedures to be called as event triggers have no arguments and `event_trigger` should be declared as return type.

When a PL/pgSQL function is called with an event trigger, several special variables are automatically created in the top-level block.

- TG_EVENT
  Data type `text`; a string representing the event under which the trigger will be executed.
- TG_TAG
  Data type `text`; a variable that contains a command tag where the trigger is executed.

The trigger in the example below generates a `NOTICE` message each time a supported command is executed.

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE PROCEDURE snitch();
```

Tips for Developing in PL/pgSQL

A good way of development in PL/pgSQL is to write a function using a text editor of your choice and load and test it using psql in another window. If you work like this, we recommend you to write a function using `CREATE OR REPLACE FUNCTION`. You can then update the function definition by reloading the file. An example is shown below:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

You can load or reload the following function definition file while running psql as follows:

```
\i filename.sql
```

Another good way to develop in PL/pgSQL is to use a GUI database access tool that facilitates development with a procedural language. These tools often provide convenient features (e.g. removing single quotes, easier play back).

Quotation processing
The code of a PL/pgSQL function is specified as a string literal in `CREATE FUNCTION`. If you use a usual way to enclose a string in single quotes, you need to double the single quotes in the body of the function. Likewise, you should use two backslashes as well (assuming escape string syntax is used). Using quotes twice can make your code difficult to understand. This is because it is easy for the user to know that many adjacent quotes are

needed.
Therefore, you are recommended to write function bodies using "dollar quoted" string literals. In the dollar citation method, you should never use quotation marks twice, but instead choose a different dollar citation delimiter for each required nesting level. For example, you can create a CREATE FUNCTION command as follows:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
$PROC$ LANGUAGE plpgsql;
```

In this case, you may use quotes around a simple literal string in SQL commands and use $$ to separate the SQL command to assemble into strings. If you need to quote text that contains $$, you can use Q.

The following chart shows what to do when writing quotes without dollar citation marks. This can be useful when converting dollar citation marks to something more understandable.

1 quotation mark
To start and end a function body:

```
CREATE FUNCTION foo() RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

Within the function body enclosed in single quotes, the quotes should appear in pairs.

2 quotation marks
It is used to express a string literal inside a function body. For example:

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

In the dollar citation scheme, you can write:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

The PL/pgSQL parser figures out both cases exactly.

4 quotation marks
This is used when a string constant inside the function body requires a single quote. For example:

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

The value that has actually been added to a_output: AND name LIKE 'foobar'AND xyz.

In the dollar citation scheme, you can write:

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

6 quotation marks

Used when a single quote of a string inside the function body is adjacent to the end of the string constant. For example:

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

The value added to a_output: `AND name LIKE 'foobar'`.

In the dollar citation scheme, you can write:

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 quotation marks
Two single quotes are needed in a string constant (representing eight quote marks) and are adjacent to the end of the string constant (two or more). You will only need it if you are writing a function that creates another function.

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

The value of a_output is as follows

```
if v_... like ''...'' then return ''...''; end if;
```

In the dollar citation scheme, you can write:

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

PL/Python

PL/Python Functions

Functions in PL/Python are declared using standard CREATE FUNCTION syntax.

```
CREATE FUNCTION funcname (argument-list)
  RETURNS return-type
AS $$
```

```
  # PL/Python function body
$$ LANGUAGE plpythonu;
```

The body of this function is a simple python script. When the function is called, its arguments are passed to the list args; named arguments are passed to the Python script as ordinary variables as well. Such named arguments are generally more readable. The result is returned in the Python code with return or yield (in case of a result-set statement). If you do not provide a return value, Python returns the default (None). PL/Python converts Python's None to SQL's Null.

For example, a function that returns the greater of two integers can be defined as:

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

The Python code that is given as the body of the function definition is transformed into a Python function. For example, the above results in:

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

Assuming that 23456 is the OID assigned to the function.

The argument is set to a global variable. Unless a variable is declared globally in a block, the argument variable cannot be reassigned within the function as an expression value containing the variable name itself according to the Python's scoping rules. For example, the following may not work:

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

When you assign it to x, x becomes a local variable for the entire block. Thus, the x on the right side, which is not a PL/Python function parameter, only refers to the local variable x that has not yet been allocated. You can use a global statement to perform the following:

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

However, it is better not to rely on the detailed implementation of PL/Python; you are recommended to use function parameters as read-only.

Data Values

In general, the purpose of PL/Python is to provide a "natural" mapping between AgensGraph and Python. Information on the data mapping rules is described below.

Data Type Mapping

When a PL/Python function is called, the argument is converted from the AgensGraph data type to the corresponding Python type.

- AgensGraph `boolean` is converted to Python `boolean`.

- AgensGraph `smallint` and `int` are converted to Python `int`. AgensGraph `bigint` and `oid` are converted to `long` in Python2 and `int` in Python3.

- AgensGraph `real` and `double` are converted to Python `float`.

- AgensGraph `numeric` is converted to Python `Decimal`. If this type is available, bring it from the `cdecimal` package. Otherwise, the standard library `decimal.Decimal` is used. `cdecimal` is considerably faster than `decimal`. In Python3.3 or later versions, there is no difference between the two as `cdecimal` is integrated into the standard library under the name `decimal`.

- AgensGraph `bytea` is converted to `str` in Python2 and `bytes` in Python3. In Python2, you need to treat the string as a sequence of bytes without character encoding.
- All other data types, including the AgensGraph string format, are converted to Python `str`. In Python2, this string is in the AgensGraph server encoding. In Python3, it becomes the same Unicode string as all strings.

- For nonscalar data type, see below.

When returning a PL/Python function, the return value is converted to the declared AgensGraph return data type of the function as follows:

- When the AgensGraph return type is `boolean`, the return value is evaluated to see if it is true according to the Python rules. That is, while 0 and an empty string are false, 'f' is true.

- When the AgensGraph return type is `bytea`, the return value is converted to string (Python2) or bytes (Python3) using each Python built-in function, and the result is converted to bytea.

- For all other AgensGraph return types, the conversion value is converted to a string using the Python built-in `str`, and the result is passed to the input function of AgensGraph data type. (If the Python value is `float`, use the `repr` built-in instead of `str` to avoid loss of precision.)

In Python2, strings should be in the AgensGraph server encoding when passed to AgensGraph. A string that is not valid in the current server encoding will cause an error; since not all encoding discrepancies can be detected, garbage values can continue to occur if this is not done correctly. As Unicode strings are automatically converted to the correct encoding, using them can be safer and more convenient. In Python3, all strings are Unicode strings.

*See below for more information on nonscalar data types.

The logical discrepancy between the declared AgensGraph return type and actual return object's Python data type is not displayed; in any case the value will be returned.

Null, None

If SQL null is passed to a function, the argument value in Python is displayed as none. For example, the function definition in PL/Python Functions `pymax` returns an incorrect answer for null input. You can add `STRICT` to your function definition to make the work more reasonable. If null is passed, the function will not be called at all and will automatically return null. You may also check for null input in the function body.

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

As shown above, to return an SQL null value from a PL/Python function, return the value None. This can be done whether the function is strict or not.

Arrays, Lists

SQL array values are passed to PL/Python as a Python list. To return a SQL array value from a PL/Python function, return a Python sequence (e.g. a list or tuple).

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return (1, 2, 3, 4, 5)
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

Strings are sequences in Python; Python programmers who are not familiar with this can have undesirable consequences.

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

Composite Types

Composite-type arguments are passed to the function as Python mappings. The element names of the mapping are the attribute names of the composite type. If an attribute in the passed row has a null value, it has a None value in the mapping. Here is an example:

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

There are multiple ways to return row or composite types from a Python function. To execute a composite type result example, create a TYPE as shown below:

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

A composite result can be returned as a:

- Sequence type (a tuple or list, but not a set because it is not indexable) Returned sequence objects must have the same number of items as the composite result type has fields. The item with index 0 is assigned to the first field of the composite type, 1 to the second and so on. For example:

  ```
  CREATE FUNCTION make_pair (name text, value integer)
    RETURNS named_value
  AS $$
    return [ name, value ]
    # or alternatively, as tuple: return ( name, value )
  $$ LANGUAGE plpythonu;
  ```

  To return a SQL null for any column, insert none at the corresponding position.

- Mapping (dictionary)

  The value for each result type column is retrieved from the mapping with the column name as key. Example:

  ```
  CREATE FUNCTION make_pair (name text, value integer)
    RETURNS named_value
  AS $$
    return { "name": name, "value": value }
  $$ LANGUAGE plpythonu;
  ```

Any extra dictionary key/value pairs are ignored. Missing keys are treated as errors. To return a SQL null value for any column, insert None with the corresponding column name as the key.

- Object (any object providing method __getattr__)

  This works the same as a mapping. Here is an example:

```
    CREATE FUNCTION make_pair (name text, value integer)
      RETURNS named_value
    AS $$
      class named_value:
        def __init__ (self, n, v):
          self.name = n
          self.value = v
      return named_value(name, value)

      # or simply
      class nv: pass
      nv.name = name
      nv.value = value
      return nv
    $$ LANGUAGE plpythonu;
```

  Functions with OUT parameters are also supported. Here is an example:

```
    CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
    return (1, 2)
    $$ LANGUAGE plpythonu;

    SELECT * FROM multiout_simple();
```

Set-returning Functions

A PL/Python function can also return sets of scalar or composite types. There are several ways to achieve this because the returned object is internally turned into an iterator. The following examples assume we have composite type:

```
CREATE TYPE greeting AS (
how text,
who text
);
```

A set result can be returned from a:

- Sequence type (tuple, list, set)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # return tuple containing lists as composite types
  # all other combinations work also
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

- Iterator (any object providing `__iter__` and next methods)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

- Generator (yield)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
    yield ( how, who )
$$ LANGUAGE plpythonu;
```

The set-returning function with `OUT` parameters (using `RETURNS SETOF record`) is also supported. Here is an example:

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer)
 RETURNS SETOF record
AS $$
```

```
 return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

Sharing Data

Global dictionary SD can be used to store data between function calls. This variable is of individual static data. You should be careful since Global dictionary GD is public data that can be used in all Python functions in the session.

As each function has its own execution environment in the Python interpreter, global data and function arguments of myfunc cannot be used in myfunc2. As mentioned above, data in the GD dictionary is an exception.

Anonymous Code Blocks

PL/Python also supports anonymous code blocks that are called together with DO statements.

```
DO $$
    # PL/Python code
$$ LANGUAGE plpythonu;
```

Anonymous code blocks do not take arguments and discard all the returned values. Otherwise, they behave like a function.

Trigger Functions

When a function is used as a trigger, the dictionary TD contains the trigger-related value.

TD["event"]
Contains an event as a string (INSERT, UPDATE, DELETE, or TRUNCATE).

TD["when"]
Contains one of BEFORE, AFTER, and INSTEAD OF.

TD["level"]
Contains ROW or STATEMENT.

TD["new"], TD["old"]
Contains each trigger row according to one or both trigger events of the fields, in the case of row-level triggers.

TD["name"]
Contains the trigger name.

TD["table_name"]
Contains the name of the table on which the trigger occurred.

TD["table_schema"]
Contains the schema of the table where the trigger occurred.

TD["relid"]
Contains the OID of the table where the trigger occurred.

TD["args"]
In the case of `CREATE TRIGGER` (command) with arguments, you can use from `TD["args"][0]` to `TD["args"][n-1]`.

When `TD ["when"]` is `BEFORE` or `INSTEAD OF` and `TD ["level"]` is `ROW`, the Python function may return `None` or "OK" to indicate that the row has not changed. "SKIP" aborts the event. When `TD ["event"]` applies `INSERT` or `UPDATE`, it can return "MODIFY" to modify the new row. Otherwise, the return value is ignored.

Database Access

The PL/Python language module automatically imports a Python module called `plpy`. The functions and constants in this module are available to you in the Python code as `plpy.foo`.

Database Access Functions

The `plpy` module provides several functions to execute database commands:

plpy.execute(query [, max-rows])

If you call `plpy.execute` with a query string and select an optional row limit argument, the corresponding query will be run and the result will be returned in a result object.

The result object emulates a list or dictionary object. The result object can be accessed by row number and column name. For example:

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

returns up to 5 rows from `my_table`. If `my_table` has a column `my_column`, it would be accessed as:

```
foo = rv[i]["my_column"]
```

The number of rows returned can be obtained using the built-in `len` function.

The result object has these additional methods:

- nrows() Returns the number of rows processed by the command. Note that this is not necessarily the same as the number of rows returned. For example, `UPDATE` command will set this value but won't return any rows (unless `RETURNING` is used).
- status() Returns the `SPI_execute()` value.
- colnames(), coltypes(), coltypmods() Returns a list of column names, list of column type OIDs, and list of type-specific type modifiers for the columns, respectively. These methods raise an exception when called on a result object from a command that did not produce a result set (e.g. `UPDATE` without `RETURNING`, or `DROP TABLE`). But it is OK to use these methods on a result set containing zero rows.
- **str**() The standard `__str__` method is defined so that it is possible, for example, to debug query execution results using `plpy.debug(rv)`.

The result object can be modified.

Calling `plpy.execute` will cause the entire result set to be read into memory. Use the function only when the result set will be relatively small. If you don't want to risk excessive memory usage when fetching large results, use `plpy.cursor` rather than `plpy.execute`.

plpy.prepare(query [, argtypes]) plpy.execute(plan [, arguments [, max-rows]])

`plpy.prepare` prepares the execution plan for a query. It is called with a query string and a list of parameter types, if you have parameter references in the query. Here is an example:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1",
["text"])
```

`text` is the type of the variable you will be passing for $1. The second argument is optional if you don't want to pass any parameters to the query. After preparing a statement, you use a variant of the function `plpy.execute` to run it:

```
rv = plpy.execute(plan, ["name"], 5)
```

Pass the plan as the first argument (instead of the query string), and a list of values to substitute into the query as the second argument. The second argument is optional if the query cannot expect any parameters. The third argument is the optional row limit as before.

Query parameters and result row fields are converted between PostgreSQL and Python data types as described in Data Values.

When you prepare a plan using the PL/Python module it is automatically saved. Refer to this link for more information. In order to use this function effectively through this feature, you need to use one of the persistent storage dictionaries `SD` or `GD` (see Sharing Data). Here is an example:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if "plan" in SD:
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # rest of function
$$ LANGUAGE plpythonu;
```

plpy.cursor(query) plpy.cursor(plan [, arguments])

The `plpy.cursor` function accepts the same arguments as `plpy.execute` (except for the row limit) and returns a cursor object, which allows you to process large result sets in smaller chunks. As with `plpy.execute`, either a query string or a plan object along with a list of arguments can be used.

The cursor object provides a `fetch` method that accepts an integer parameter and returns a result object. Each time you call `fetch`, the returned object will contain the next batch of rows, which is no larger than the parameter value. Once all rows are used, `fetch` starts returning an empty result object. Cursor objects also provide an iterator interface, creating one row at a time until all rows are exhausted. Data fetched that way is not returned as result objects, but rather as dictionaries, each dictionary corresponding to a single result row.

An example of two ways of processing data from a large table is:

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
```

```
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 <> 0", ["integ
er"])
rows = list(plpy.cursor(plan, [2]))

return len(rows)
$$ LANGUAGE plpythonu;
```

Cursors are automatically disposed of. However, if you want to explicitly release all resources held by a cursor, use the `close` method. Once closed, a cursor cannot be fetched from anymore.

Trapping Errors

Functions accessing the database might encounter errors, which will cause them to abort and raise an exception. Both `plpy.execute` and `plpy.prepare` can raise an instance of a subclass of `plpy.SPIError`, which by default will terminate the function. This error can be handled just like any other Python exception, by using the `try/except` construct. For example:

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
        return "something went wrong"
    else:
        return "Joe added"
$$ LANGUAGE plpythonu;
```

The actual class of the exception being raised corresponds to the specific condition that caused the error. Refer to this link for a list of possible conditions. The module `plpy.spiexceptions` defines an exception class for each condition, deriving their names from the condition name. For instance, `division_by_zero` becomes DivisionByZero, `unique_violation` becomes `UniqueViolation`, `fdw_error` becomes `FdwError`, and so on. Each of these exception classes inherits from `SPIError`. This separation makes it easier to handle specific errors. For example:

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text
AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["in
t", "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpythonu;
```

As all exceptions from the `plpy.spiexceptions` module inherit from `SPIError`, the `except` clause handling it will catch any database access error.

As an alternative way of handling different error conditions, you can catch the `SPIError` exception and determine the specific error condition inside the `except` block by looking at the `sqlstate` attribute of the exception object. This attribute is a string value containing the "SQLSTATE" error code. This approach provides approximately the same functionality.

Explicit Subtransactions

Recovering from errors caused by database access as described in Trapping Errors can lead to an undesirable situation where some operations succeed before one of them fails; after recovering from that error, the data is left in an inconsistent state. PL/Python offers a solution to this problem in the form of explicit subtransactions.

Subtransaction Context Managers

Consider a function that implements a transfer between two accounts:

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_n
ame = 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_n
ame = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

If the second UPDATE statement results in an exception being raised, this function will report the error, but the result of the first UPDATE will nevertheless be committed. In other words, the funds will be withdrawn from Joe's account, but will not be transferred to Mary's account.

To avoid such issues, you can wrap your plpy.execute calls in an explicit subtransaction. The plpy module provides a helper object to manage explicit subtransactions that gets created with the plpy.subtransaction() function. Objects created by this function implement the context manager interface. By using explicit subtransactions, you may rewrite the function as:

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE accou
nt_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE accou
nt_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

The use of try/catch is still required. Otherwise the exception would be transferred to the top of the Python stack and would cause the whole function to abort due to an AgensGraph error; as a result, the operations table would not have any row inserted into it. The subtransaction context manager does not trap errors; it only assures that all database operations executed inside its scope will be atomically committed or rolled back. A rollback of the subtransaction block occurs on any kind of exception exit, not only ones caused by errors originating from database access. A regular Python exception raised inside an explicit subtransaction block would also cause the subtransaction to be rolled back.

Older Python Versions

Context managers syntax using the WITH keyword is available by default in Python 2.6. If using PL/Python with an older Python version, it is still possible to use explicit subtransactions, although not as transparently. You can call the subtransaction manager's __enter__ and __exit__ functions using the enter and exit convenience aliases. The example function that transfers funds could be written as:

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
try:
    subxact = plpy.subtransaction()
    subxact.enter()
    try:
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE accou
nt_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE accou
nt_name = 'mary'")
    except:
        import sys
        subxact.exit(*sys.exc_info())
        raise
    else:
        subxact.exit(None, None, None)
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

Utility Functions

The `plpy` module also provides functions:

```
plpy.debug(msg, **kwargs)
plpy.log(msg, **kwargs)
plpy.info(msg, **kwargs)
plpy.notice(msg, **kwargs)
plpy.warning(msg, **kwargs)
plpy.error(msg, **kwargs)
plpy.fatal(msg, **kwargs)
```

`plpy.error` and `plpy.fatal` actually raise a Python exception that is not actually called, causing the current transaction or subtransaction to be aborted. raise `plpy.Error(msg)` and raise `plpy.Fatal(msg)` are equivalent to calling `plpy.error(msg)` and `plpy.fatal(msg)`, respectively, but you cannot pass keyword arguments in `raise` type. Other functions only generate messages of different priority levels. You may determine whether messages of a particular priority should be reported to the client, written to the server log, or both are controlled by the log_min_messages and client_min_messages configuration variables. See this link for more information.

The msg argument is provided as a positional argument. Two or more positional arguments may be provided for backward compatibility. In this case, the tuple string expression of the positional argument is the message reported to the client.

The following keyword-specific arguments are allowed.

```
detail
hint
sqlstate
schema_name
table_name
column_name
datatype_name
```

The string expression of the object passed as a keyword-specific argument is used to enforce the messages reported to the client. Here is an example:

```
CREATE FUNCTION raise_custom_exception() RETURNS void AS $$
plpy.error("custom exception message",
           detail="some info about exception",
           hint="hint for users")
$$ LANGUAGE plpythonu;

=# SELECT raise_custom_exception();
ERROR:  plpy.Error: custom exception message
DETAIL:  some info about exception
HINT:  hint for users
CONTEXT:  Traceback (most recent call last):
  PL/Python function "raise_custom_exception", line 4, in <module>
    hint="hint for users")
PL/Python function "raise_custom_exception"
```

Another set of utility functions are `plpy.quote_literal(string)`,
`plpy.quote_nullable(string)`,
and `plpy.quote_ident(string)`. They are equivalent to the built-in quoting functions. They are useful when constructing ad-hoc queries. Dynamic PL/Python would be:

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

Environment Variables

Some of the environment variables that are accepted by the Python interpreter may affect PL/Python behavior. They would need to be set in the environment of the main

AgensGraph server process, for example in a start script. The available environment variables depend on the version of Python; see the Python manuals for more information.

- PYTHONHOME
- PYTHONPATH
- PYTHONY2K
- PYTHONOPTIMIZE
- PYTHONDEBUG
- PYTHONVERBOSE
- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE
- PYTHONHASHSEED

(It appears to be a Python implementation detail beyond the control of PL/Python that some of the environment variables listed on the `python` man page are only effective in a command-line interpreter and not an embedded Python interpreter.)

Appendix

AgensGraph Error Codes

Every message generated by the AgensGraph server is assigned with a five-character error code that complies with the SQL standard rules for "SQLSTATE" codes.

In accordance with the standard, the first two characters of the error code indicate the error class, and the last three indicate a specific condition within the class. For this reason, we may expect that an application that does not recognize a specific error code is likely to fail to recognize the error and still work in the error class.

The following table lists all the error codes defined in AgensGraph. (Some of them are not actually used at the moment, but their definitions are still included in the SQL standard.) Error classes are also displayed. Each error class has a "standard" error code with the last three characters 000. This code is used only for error conditions that are in the class but without any specific code assigned. The symbols in the "Condition Name" column are the condition names used by PL/pgSQL. Condition names can be written in uppercase or lowercase. (Unlike errors, PL/pgSQL does not recognize warnings for condition names of class 00,01,02.)

For some types of errors, the server reports the names of database objects (table, table column, data type, or constraint) associated with the errors. For example, unique_isolation is the name of a unique constraint that caused the error. As these names are provided in a separate field in the error report message, the application does not have to extract the names to a human-readable message text.

Error Code	Condition Name
Class 00 - Successful Completion	
00000	successful_completion
Class 01 - Warning	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
Class 02 - No Data	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
Class 03 - SQL Statement Not Yet Complete	
03000	sql_statement_not_yet_complete
Class 08 - Connection Exception	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown

08P01	protocol_violation
Class 09 - Triggered Action Exception	
09000	triggered_action_exception
Class 0A - Feature Not Supported	
0A000	feature_not_supported
Class 0B - Invalid Transaction Initiation	
0B000	invalid_transaction_initiation
Class 0F - Locator Exception	
0F000	locator_exception
0F001	invalid_locator_specification
Class 0L - Invalid Grantor	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Class 0P - Invalid Role Specification	
0P000	invalid_role_specification
Class 0Z - Diagnostics Exception	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Class 20 - Case Not Found	
20000	case_not_found
Class 21 - Cardinality Violation	
21000	cardinality_violation
Class 22 - Data Exception	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict

22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception

22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
Class 23 - Integrity Constraint Violation	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24 - Invalid Cursor State	
24000	invalid_cursor_state
Class 25 - Invalid Transaction State	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction

25P03	idle_in_transaction_session_timeout
Class 26 - Invalid SQL Statement Name	
26000	invalid_sql_statement_name
Class 27 - Triggered Data Change Violation	
27000	triggered_data_change_violation
Class 28 - Invalid Authorization Specification	
28000	invalid_authorization_specification
28P01	invalid_password
Class 2B - Dependent Privilege Descriptors Still Exist	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Class 2D - Invalid Transaction Termination	
2D000	invalid_transaction_termination
Class 2F - SQL Routine Exception	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34 - Invalid Cursor Name	
34000	invalid_cursor_name
Class 38 - External Routine Exception	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39 - External Routine	

Invocation Exception	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Class 3B - Savepoint Exception	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Class 3D - Invalid Catalog Name	
3D000	invalid_catalog_name
Class 3F - Invalid Schema Name	
3F000	invalid_schema_name
Class 40 - Transaction Rollback	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class 42 - Syntax Error or Access Rule Violation	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name

42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition

Class 44 - WITH CHECK OPTION

Violation	
44000	with_check_option_violation
Class 53 - Insufficient Resources	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Class 54 - Program Limit Exceeded	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Class 55 - Object Not In Prerequisite State	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
Class 57 - Operator Intervention	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
Class 58 - System Error	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Class 72 - Snapshot Failure	

72000	snapshot_too_old
Class F0 - Configuration File Error	
F0000	config_file_error
F0001	lock_file_exists
Class HV - Foreign Data Wrapper Error	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply

HV00N	fdw_unable_to_establish_connection
Class P0 - PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Class XX - Internal Error	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted

Terminology

Database Cluster

- Server (or Node)
  Refers to hardware (actual or virtual) with AgensGraph installed.
- Cluster (or 'Database Cluster')
  Refers to storage space (directory, subdirectory, file) in the file system. Database cluster also has global object definitions, such as "Users and Privileges." These things affect the entire database. There are at least three databases ('template0', 'template1', 'postgres') in database cluster. Roles of each database:
  template0': Template database that can be used with CREATE DATABASE command (template0 should never be modified)
  template1': Template database that can be used by CREATE DATABASE command (template1 can be modified by DBA)
  postgres': an empty database primarily for maintenance purposes
- Instance (or 'Database Server Instance' or 'Database Server' or 'Backend')
  An instance is a group of processes in a UNIX server. In a Windows server, it is a shared memory that controls and manages services and a cluster. From an IP perspective, one may assume that an instance occupies a combination of IP/port (e.g. http://localhost:5432). This means you can run different instances on different ports on the same server. It is also possible to run many instances on a single system per cluster if the server has multiple clusters.
- Database
  A database is a storage area in the file system where object collections are stored in

files. An object consists of data, metadata (table definitions, data types, constraints, views, etc.) and other data such as indexes, all of which are stored in the default database 'postgres' or in a newly created database. The storage area for one database consists of a single subdirectory tree in the storage area of the database cluster. This means a database cluster may contain multiple databases.

- Schema
  A namespace within a database. A schema consists of named objects (tables, data types, functions, and operators) that can replicate object names in other schemas in the database. All databases contain the default schema 'public' and is able to contain more schemas. All objects in a schema should be in the same database, and objects in other schemas in the same database may have the same name. Each database has 'pg_catalog,' a special schema; 'pg_catalog' contains all system tables, built-in data types, functions and operators.
- Search Path (or 'Schema Search Path')
  A list of schema names. If the application uses an unqualified object name (for example, 'employee_table' in table name), the search path is used to find this object in the specified schema order. The 'pg_catalog' schema is not specified in the search path, but is always the first part of the search path. This action allows AgensGraph to find the system object.
- initdb (OS command)
  initdb creates a new cluster (including 'template0', 'template1', and 'postgres').

Consistent Writes

- Checkpoint
  A checkpoint is a time when a database file is guaranteed to be in a consistent state. At checkpoint time, all changes are written to the WAL file, all dirty data pages in the shared buffer are flushed to disk, and finally the checkpoint record is written to the WAL file. The instance's checkpoint process is triggered according to a regular schedule. You can also force it through CHECKPOINT command in the client program. In the case of a database system, it takes a lot of time to execute the checkpoint command as it is physically written to disk.
- WAL File
  The WAL file consists of changes that are applied to data through commands such as INSERT, UPDATE, DELETE, or CREATE TABLE. This is redundant information as it is also written to the data file (for better performance). Depending on the configuration of the instance, more information may be recorded in the WAL file. The WAL file is in the pg_wal directory (pg_xlog for version 10 or earlier) in binary format with a fixed size of 16MB. A single unit of information within the WAL file is called a log record.

- Logfile
  An instance writes and reports warning and error messages for special situations to a readable text file. The log file recorded here can be stored at any position in the server except for clusters.

- Log Record
  A log record is a single unit of information within the WAL file.

FAQ

What debugging features do you have?

Compile time

First, if you are developing using a new C code, you should always work in a build environment consisting of the `--enable-cassert` and `--enable-debug options`. If you enable a debug symbol, you can trace the malfunctioning code using the debugger (e.g. gdb). When compiling with gcc, an additional called `-ggdb -Og -g3 -fno-omit-frame-pointer` is useful because it inserts a lot of debugging information. You can pass it to `configure` as follows:

```
./configure --enable-cassert --enable-debug CFLAGS="-ggdb -Og -g3 -fno-omit-f
rame-pointer"
```

If you use `-O0`, instead of `-Og`, most compiler optimizations, including inlining, will be disabled. However, `-Og` performs much like a normal optimization flag such as `-O2` or `-Os`, providing more debug information; it has far fewer `<value optimized out>` variables, generates less confusion or difficulty in changing the execution order with a quite useful performance. `-ggdb -g3` tells `gcc` to include the maximum amount of debug information in the generated binaries, including things like macro definitions. `-fno-omit-frame-pointer` is useful when using trace and profiling tools (e.g. `perf`) as with the frame pointer that allows these tools to capture the call stack as well as the stack's top functions.

Run time

The postgres server has a -d option to record detailed information (elog or ereport DEBUGn output). The -d option takes a number that specifies the debug level. Note that high debug level values generate large log files. This option is not available when starting the server via `ag_ctl`, but you can use `-o log_min_messages = debug4` or something similar.

gdb

If postmaster is running, start agens in a window and find the PID of the postgres process agens using
`SELECT pg_backend_pid()`. Use the debugger to attach postgres PID `-gdb -p 1234` or `attach 1234` within the running gdb. The gdblive script can be useful as well. You can set breakpoints in the debugger and then run queries in the agens session.

Set breakpoints in `errfinish` to find a location to generate errors or log messages. As this will trap all of the `elog` and `ereport` calls on the available log levels, many triggers may be fired. If you are only interested in ERROR/FATAL/PANIC, use gdb conditional breakpoints on `errordata[errordata_stack_depth].elevel >= 20`, or set source line breakpoints on PANIC, FATAL, and ERROR in `errfinish`. Not all errors pass errfinish. In particular, authorization checks occur separately. If the breakpoints are not triggered, run `git grep` on the error text and see where it was thrown.

If you are debugging things that occurred when you start a session, you may start agens after setting `PGOPTIONS="-W n"`. You can then delay the start for n seconds, connect to the process using the debugger, set the appropriate breakpoints, and continue the startup sequence.

You may tell the target process alternately for debugging by looking at `pg_stat_activity`, log,
`pg_locks`, `pg_stat_replication`, and so on.

I broke `initdb`. How do I debug it?

Sometimes a patch can cause an initdb failure. These are rare in initdb itself. A failure occurs to do some configuration job on the postgres backend, which is started more often by initdb.

If one of them crashes, putting `gdb` in `initdb` is OK by itself, and gdb will not break because initdb itself does not crash.

All you need to do is run `initdb` in `gdb`, set a breakpoint on `fork`, and then continue execution. When you trigger a breakpoint, the function will end. `gdb` will report that a child process has been created. But this is not what you want, since it is the shell that started the actual `postgres` instance.

Find the `postgres` instance that began using ps while `initdb` is paused. `pstree -p` can be useful for this. If you find it, attach a separate `gdb` session as `gdb -p $ the_postgres_pid`. At this point, you can safely disconnect `gdb` from `initdb` and debug the failed `postgres` instance.

I need to change the parsing query. Can you briefly describe the parser file?

The parser file is in the `src/backend/parser` directory.

scan.l defines a lexer that is an algorithm for splitting a string (including SQL statements) into a token stream. Tokens are usually single words and do not contain spaces; they are instead separated by spaces. For example, it can be a string enclosed in double or single quotation marks. The lexer is basically defined as a regular expression that describes various token types.

gram.y defines the grammar (syntax structure) of the SQL statement using tokens generated by the lexer as basic building blocks. The grammar is defined in BNF notation. BNF is similar to regular expressions, but works at the non-character token level. Patterns (also called rules or productions in BNF) are named and can be recursive. That is, it can use itself as a subpattern.

The actual lexer is created in scan.l with a tool called flex; its manual can be found at http://flex.sourceforge.net/manual/.

The actual parser is created in gram.y with a tool called bison; a relevant guide can be found at http://www.gnu.org/s/bison/.

However, if you have not used flex or bison before, you may find it a bit difficult to learn.

How can I efficiently access information of the system catalogs in the backend code?

First you need to find the tuple (row) you are interested in. There are two ways. First, SearchSysCache() and related functions allow you to query system catalogs using predefined indexes of the catalog. This is the primary way to access system tables because, after loading the row that needs first call on the cache, subsequent requests can return results without accessing the base table. A list of available caches is in src/backend/utils/cache/syscache.c. Many column-specific cache lookups are contained in src/backend/utils/cache/lsyscache.c.

The row returned is the heap row of the cache-owned version. Therefore, you should not modify or delete the tuple returned by SearchSysCache(). You must release it when you have finished using ReleaseSysCache(). This tells the cache that it can delete the tuple if necessary. If you do not call ReleaseSysCache(), the cache entry is locked in the cache until the transaction ends. This can be tolerated only in the stage of development (not tolerated in a code worth releasing).

If the system cache is not available, you should retrieve the data directly from the heap table using the buffer cache shared by all backends. The backend automatically loads the row into the buffer cache. To do this, use heap_open() to open the table. You can then start the table scan using heap_beginscan() and continue it as long as you use heap_getnext() and HeapTupleIsValid() returns true. Now, run heap_endscan(). The key can be assigned to the scan. Since the index is not used, all rows are compared to the key and only valid rows are returned.

You can also fetch rows by block number/offset using heap_fetch(). While the checker automatically locks/unlocks rows in the buffer cache, it should pass the buffer pointer with heap_fetch(). When finished, releaseBuffer() should be passed.

Once there is a row, you can access the HeapTuple structure entry to get data common to all tuples such as t_self and t_oid. If you need a table-related column, you need to fetch a HeapTuple pointer and use the GETSTRUCT () macro to access the table-related start tuple. Then, cast the pointer (e.g. Form_pg_proc if accessing the pg_proc table, or Form_pg_type if accessing pg_type). You can then access the fields of the tuple using struct pointers: ((Form_pg_class) GETSTRUCT (tuple))->relnatts

However, this works only when the column is fixed-width and non-null and when all previous columns are fixed-width and absolutely null. Otherwise, as the position of the column is variable, you must extract it from the tuple using heap_getattr() or a related function.
Do not store it directly in the struct field by changing the live tuple. The best option is to use heap_modifytuple() to pass the original tuple and the value you want to change. It returns a tuple enclosed in palloc, and passes it to heap_update(). You can delete a tuple by passing tuple's t_self to heap_delete(). Use t_self for heap_update() as well. A tuple can be a copy of the system cache that may disappear after calling ReleaseSysCache(), or that reads directly from the disk buffer when heap_getnext(), heap_endscan, or ReleaseBuffer() in the heap_fetch() case disappears (Or, it can be a palloc tuple). When done, you should perform pfree().

How can I add a new port?

There are many places to modify to add a new port. Start with the src/template directory. Add an appropriate item to the OS. Use src/config.guess to add the OS to src/template/.similar. You do not have to match the OS versions exactly. The configure test finds the correct OS version number; if not found, it finds what matches without a version number. Edit src/configure.in to add the new OS. See above configuration items. You should run autoconf or patch src/configure.

Then, check src/include/port and add the new OS file with appropriate values. Fortunately, there is a lock code for CPU in src/include/storage/s_lock.h. There is also a src/makefiles directory for port-specific Makefile processing. There is a backend/port directory in case your OS needs special files.