Apache AGE (Incubating)

Apache AGE

Apache AGE a PostgreSQL extension that provides graph database functionality. AGE is an acronym for A Graph Extension, and is inspired by Bitnine's fork of PostgreSQL 10, AgensGraph, which is a multi-model database. The goal of the project is to create single storage that can handle both relational and graph model data so that users can use standard ANSI SQL along with openCypher, the Graph query language.

Below is a brief overview of the AGE architecture in relation to the PostgreSQL architecture and backend. Every component runs on the PostgreSQL transaction cache layer and storage layer.

Setup

Getting AGE

Getting Source Code

Apache AGE releases

The release notes for each release can be found from one of the following links:

- Apache AGE 0.3.0
- Apache AGE 0.2.0
- Apache AGE 0.1.0

The source code can be downloaded from one of the following Apache release

- https://downloads.apache.org/incubator/age/

Github

The source code can be downloaded from the master branch on Github. Please note that this is the development link and the Apache AGE makes no guarantees about the stability of the source code.

- https://github.com/apache/incubator-age

Pre-Installation

Install the following essential libraries according to each OS.

Building AGE from source depends on the following Linux libraries (Ubuntu package names shown below):

CENTOS

```
$ yum install gcc glibc glib-common readline readline-devel zlib
zlib-devel flex bison
```

Fedora

```
$ dnf install gcc glibc bison flex readline readline-devel zlib
zlib-devel
```

Ubuntu

```
$ sudo apt-get install build-essential libreadline-dev zlib1g-dev
flex bison
```

Installation

The build process will attempt to use the first path in the PATH environment variable when installing AGE. If the pg_config path is located there, run the following command in the source code directory of Apache AGE to build and install the extension.

```
$ make install
```

If the path to your Postgres installation is not in the PATH variable, add the path in the arguments:

```
$ make PG_CONFIG=/path/to/postgres/bin/pg_config install
```

Post Installation

Per Installation Instructions

After the installation, run the CREATE EXTENSION command to have AGE be installed on the server.

```
CREATE EXTENSION age;
```

Per Session Instructions

```
LOAD 'age';
```

We recommend adding ag_catalog to your search_path to simplify your queries. The rest of this document will assume you have done so. If you do not, remember to add 'ag_catalog' to your cypher query function calls.

```
SET search_path = ag_catalog, "$user", public;
```

Features

Graphs

A graph consists of a set of vertices and edges, where each individual node and edge possesses a map of properties. A vertex is the basic object of a graph, that can exist independently of everything else in the graph. An edge creates a directed connection between two vertices.

Create a Graph

To create a graph, use the create_graph function, located in the ag_catalog namespace.

create_graph()

Syntax: `create_graph(graph_name);`
Returns:

```
void
```

Arguments:

Name	Description
graph_name	Name of the graph to be created

Considerations

- This function will not return any results. However if there is not an error message the graph will be created.
- Tables needed to set up the graph are automatically created.

Example:

```
SELECT * FROM ag_catalog.create_graph('graph_name');
```

Delete a Graph

To delete a graph, use the drop_graph function, located in the ag_catalog namespace.

drop_graph()

Syntax: `drop_graph(graph_name, cascade);`

Returns:

```
void
```

Arguments:

Name	Description
graph_name	Name of the graph to be created
cascade	A boolean that will not drop the graph if any data remains in the graph.

Considerations:

- This function will not return any results. However if there is not an error message the graph will be created.

- It is recommended to set the cascade option to true, otherwise everything in the graph must be manually dropped with SQL DDL commands.

Example:

```
SELECT * FROM ag_catalog.drop_graph('graph_name', true);
```

A Cypher Graph vs A Postgres Namespace

Cypher uses a Postgres namespace for every individual graph. It is recommended that no DML or DDL commands are executed in the namespace that is reserved for the graph.

Data Types - An Introduction to agtype

AGE uses a custom data type called agtype, which is the only data type returned by AGE. Agtype is a superset of Json and a custom implementation of JsonB.

Simple Data Types

Null

In Cypher, null is used to represent missing or undefined values. Conceptually, null means 'a missing unknown value' and it is treated somewhat differently from other values. For example getting a property from a vertex that does not have said property produces null. Most expressions that take null as input will produce null. This includes boolean expressions that are used as predicates in the WHERE clause. In this case, anything that is not true is interpreted as being false. null is not equal to null. Not knowing two values does not imply that they are the same value. So the expression null = null yields null and not true.

Input/Output Format

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN NULL
$$) AS (null_result agtype);
```

A null will appear as an empty space.

Result:

null_result

(1 row)

Agtype NULL vs Postgres NULL

Integer

The integer type stores whole numbers, i.e. numbers without fractional components. Integer data type is a 64-bit field that stores values between -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Attempts to store values outside of the allowed range will result in an error.

The type integer is the common choice, as it offers the best balance between range, storage size, and performance. The smallint type is generally only used if disk space is at a premium. The bigint type is designed to be used when the range of the integer type is insufficient.

Input/Output Format

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN 1
$$) AS (int_result agtype);
```

Result:

int_result
1
(1 row)

Float

The data type float is an inexact, variable-precision numeric type, conforming to the IEEE-754 Standard.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire

branch of mathematics and computer science and will not be discussed here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the numeric type instead.
- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
- Comparing two floating-point values for equality might not always work as expected.

Values that are too large or too small will cause an error. Rounding might take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

In addition to ordinary numeric values, the floating-point types have several special values:

- Infinity
- -Infinity
- NaN

These represent the IEEE 754 special values “infinity”, “negative infinity”, and “not-a-number”, respectively. When writing these values as constants in a Cypher command, you must put quotes around them and typecast them, for example SET x.float_value = '-Infinity'::float. On input, these strings are recognized in a case-insensitive manner.

> **Note**
> IEEE754 specifies that NaN should not compare equal to any other floating-point value (including NaN). However, in order to allow floats to be sorted correctly, AGE evaluates ‘NaN’::float = ‘NaN’:float to true. See the section Comparability and Equality for more details.

Input/Output Format:

To use a float, denote a decimal value.

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN 1.0
$$) AS (float_result agtype);
```

Result:

float_result
1.0
(1 row)

Numeric

The type numeric can store numbers with a very large number of digits. It is especially recommended for storing monetary amounts and other quantities where exactness is required. Calculations with numeric values yield exact results where possible, e.g., addition, subtraction, multiplication. However, calculations on numeric values are very slow compared to the integer types, or to the floating-point type.

We use the following terms below: The *precision* of a numeric is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. The *scale* of a numeric is the count of decimal digits in the fractional part, to the right of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas numeric columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. We find this a bit useless. If you're concerned about portability, always specify the precision and scale explicitly.)

Note
The maximum allowed precision when explicitly specified in the type declaration is 1000; NUMERIC without a specified precision is subject to the limits described in Table 8.2.

If the scale of a value to be stored is greater than the declared scale of the column, the system will round the value to the specified number of fractional digits. Then, if the number of digits to the left of the decimal point exceeds the declared precision minus the declared scale, an error is raised.

Numeric values are physically stored without any extra leading or trailing zeroes. Thus, the declared precision and scale of a column are maximums, not fixed allocations. (In this sense the numeric type is more akin to varchar(*n*) than to char(*n*).) The actual storage requirement is two bytes for each group of four decimal digits, plus three to eight bytes overhead.

In addition to ordinary numeric values, the numeric type allows the special value NaN, meaning “not-a-number”. Any operation on NaN yields another NaN. When writing this value as a constant in an SQL command, you must put quotes around it, for example UPDATE table SET x = 'NaN'. On input, the string NaN is recognized in a case-insensitive manner.

> **Note**
> In most implementations of the "not-a-number" concept, NaN is not considered equal to any other numeric value (including NaN). However, in order to allow floats to be sorted correctly, AGE evaluates ‘NaN’::numeric = ‘NaN’:numeric to true. See the section Comparability and Equality for more details.

When rounding values, the numeric type rounds ties away from zero, while (on most machines) the real and double precision types round ties to the nearest even number. For example:

Input/Output Format:

When creating a numeric data type, the ‘::numeric’ data annotation is required.

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN 1.0::numeric
$$) AS (numeric_result agtype);
```

Result:

numeric_result
1.0::numeric
(1 row)

Bool

AGE provides the standard Cypher type boolean. The boolean type can have several states: “true”, “false”, and a third state, “unknown”, which is represented by the Agtype null value.

Boolean constants can be represented in Cypher queries by the keywords TRUE, FALSE, and NULL.

Note that the parser automatically understands that TRUE and FALSE are of type boolean, but this is not so for NULL because that can have any type. So in some contexts you might have to cast NULL to boolean explicitly, for example NULL::boolean. Conversely, the cast can be

omitted from a string-literal Boolean value in contexts where the parser can deduce that the literal must be of type boolean.

Input/Output Format

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN TRUE
$$) AS (boolean_result agtype);
```

Unlike Postgres, AGE's boolean outputs as the full word, ie. true and false as opposed to t and f.

Result:

boolean_result
true
(1 row)

String

Agtype strings String literals can contain the following escape sequences:

Escape Sequence	Character
\t	Tab
\b	Backspace
\n	Newline
\r	Carriage Return
\f	Form Feed
\'	Single Quote
\"	Double Quote
\\	Backslash

\uXXXX	Unicode UTF-16 code point (4 hex digits must follow the \u)

Input/Output Format

Use single (‘) quotes to identify a string. The output will use double (“) quotes.

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN ‘This is a string’
$$) AS (string_result agtype);
```

Result:

string_result
“This is a string”
(1 row)

Composite Data Types

List

All examples will use the WITH clause and RETURN clause.

Lists in general

A literal list is created by using brackets and separating the elements in the list with commas.

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst
$$) AS (lst agtype);
```

Result:

lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
(1 row)

NULL in a List

A list can hold the value null, unlike when a null is an independent value, it will appear as the word ‘null’ in a list

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH [null] as lst
    RETURN lst
$$) AS (lst agtype);
```

Result:

lst
[null]
(1 row)

Access Individual Elements

To access individual elements in the list, we use the square brackets again. This will extract from the start index and up to but not including the end index.

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst[3]
$$) AS (element agtype);
```

Result:

element
3
(1 row)

Map Elements in Lists

Query

```
SELECT *
FROM cypher('graph_name', $$
   WITH [0, {key: 'key_value'}, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst
$$) AS (map_value agtype);
```

Result:

map_value
[0, {"key": "key_value"}, 2, 3, 4, 5, 6, 7, 8, 9, 10]
(1 row)

Accessing Map Elements in Lists

Query

```
SELECT *
FROM cypher('graph_name', $$
   WITH [0, {key: 'key_value'}, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst[1].key
$$) AS (map_value agtype);
```

Result:

map_value
“key_value”
(1 row)

Negative Index Access

You can also use negative numbers, to start from the end of the list instead.

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst[-3]
$$) AS (element agtype);
```

Result:

element
8
(1 row)

Index Ranges

Finally, you can use ranges inside the brackets to return ranges of the list.

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst[0..3]
$$) AS (element agtype);
```

Result:

element
[0, 1, 2]
(1 row)

Negative Index Ranges

Query

```
SELECT *
```

```
FROM cypher('graph_name', $$
    WITH [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst[0..-5]
$$) AS (lst agtype);
```

Result:

lst
[0, 1, 2, 3, 4, 5]
(1 row)

Positive Slices

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst[..4]
$$) AS (lst agtype);
```

Result:

lst
[0, 1, 2, 3]
(1 row)

Negative Slices

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst[-5..]
$$) AS (lst agtype);
```

Result:

lst

[6, 7, 8, 9, 10]
(1 row)

Out-of-bound slices are simply truncated, but out-of-bound single elements return null.

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst[15]
$$) AS (element agtype);
```

Result:

element
[0, 1, 2, 3]
(1 row)

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as lst
    RETURN lst[5..15]
$$) AS (element agtype);
```

Result:

element
[5, 6, 7, 8, 9, 10]
(1 row)

Map

Maps can be constructed using Cypher.

Literal Maps with Simple Data Types

You can construct a simple map with simple agtypes

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH {int_key: 1, float_key: 1.0, numeric_key: 1::numeric, bool_key:
true, string_key: 'Value'} as m
    RETURN m
$$) AS (m agtype);
```

Result:

m
{"int_key": 1, "bool_key": true, "float_key": 1.0, "string_key": "Value", "numeric_key": 1::numeric}
(1 row)

Literal Maps with Composite Data Types

A map can also contain Composite Data Types, i.e. lists and other maps.

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH {listKey: [{inner: 'Map1'}, {inner: 'Map2'}], mapKey: {i: 0}} as m
    RETURN m
$$) AS (m agtype);
```

Result:

m
{"mapKey": {"i": 0}, "listKey": [{"inner": "Map1"}, {"inner": "Map2"}]}
(1 row)

Property Access of a map

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH {int_key: 1, float_key: 1.0, numeric_key: 1::numeric, bool_key:
true, string_key: 'Value'} as m
    RETURN m.int_key
$$) AS (int_key agtype);
```

Result:

int_key
1
(1 row)

Accessing List Elements in Maps

Query

```
SELECT *
FROM cypher('graph_name', $$
    WITH {listKey: [{inner: 'Map1'}, {inner: 'Map2'}], mapKey: {i: 0}} as m
    RETURN m.listKey[0]
$$) AS (m agtype);
```

Result:

m
{"inner": "Map1"}
(1 row)

Simple Entities

An entity has a unique, comparable identity which defines whether or not two entities are equal. An entity is assigned a set of properties, each of which are uniquely identified in the set by the irrespective property keys.

GraphId

Simple entities are assigned a unique graphid. A graphid is a unique composition of the entity's label id and a unique sequence assigned to each label. Note that there will be overlap in ids when comparing entities from different graphs.

Labels

A label is an identifier that classifies vertices and edges into certain categories.

- Edges are required to have a label, but vertices do not.
- The names of labels between vertices and edges cannot overlap.

See CREATE clause for information about how to make entities with labels.

Properties

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

Vertex

- A vertex is the basic entity of the graph, with the unique attribute of being able to exist in and ofitself.
- A vertex may be assigned a label.
- A vertex may have zero or more outgoing edges.
- A vertex may have zero or more incoming edges.

Data Format:

Attribute Name	**Description**
Id	graphid for this vertex
label	Name of the label this vertex has
properties	Properties associated with this vertex

```
{id:1; label: ‘label_name’; properties: {prop1: value1, prop2: value2}}::vertex
```

Type Casting a Map to a Vertex

Query

```
SELECT *
FROM cypher('graph_name', $$
      WITH {id: 0, label: "label_name", properties: {i: 0}}::vertex as v
      RETURN v
$$) AS (v agtype);
```

Result:

v
{"id": 0, "label": "label_name", "properties": {"i": 0}}::vertex
(1 row)

Edge

An edge is an entity that encodes a directed connection between exactly two nodes, thesource node and the target node. An outgoing edge is a directed relationship from the point of view of its source node. An incoming edge is a directed relationship from the point of view of its target node. An edge is assigned exactly one edge type.

Data Format

Attribute Name	**Description**
id	graphid for this edge
startid	graphid for the incoming edge
endid	graphid for the outgoing edge
label	Name of the label this edge has
properties	Properties associated with this edge

Output:

```
{id: 3; startid: 1; endid: 2; label: ‘edge_label’ properties{prop1: value1, prop2: value2}}::edge
```

Type Casting a Map to an Edge

Query

```
SELECT *
FROM cypher('graph_name', $$
      WITH {id: 2, start_id: 0, end_id: 1, label: "label_name", properties: {i: 0}}::edge as e
      RETURN e
$$) AS (e agtype);
```

Result:

v
{"id": 2, "label": "label_name", "end_id": 1, "start_id": 0, "properties": {"i": 0}}::edge
(1 row)

Composite Entities

Path

A path is a series of alternating vertices and edges. A path must start with a vertex, and have at least one edge.

Type Casting a Map to a Path

Query

```
SELECT *
FROM cypher('graph_name', $$
      WITH [{id: 0, label: "label_name_1", properties: {i: 0}}::vertex,
            {id: 2, start_id: 0, end_id: 1, label: "edge_label", properties: {i: 0}}::edge,
           {id: 1, label: "label_name_2", properties: {}}::vertex
           ]::path as p
      RETURN p
$$) AS (p agtype);
```

The result is formatted to improve readability

Result:

p
[{"id": 0, "label": "label_name_1", "properties": {"i": 0}}::vertex, {"id": 2, "label": "edge_label", "end_id": 1, "start_id": 0, "properties": {"i": 0}}::edge, {"id": 1, "label": "label_name_2", "properties": {}}::vertex]::path
(1 row)

Comparability, Equality, Orderability and Equivalence

AGE already has good semantics for equality within the primitive types (booleans, strings,integers, and floats) and maps. Furthermore, Cypher has good semantics for comparability and orderability for integers, floats, and strings, within each of the types. However, working with values of different types deviates from Postgres' defined logic and the openCypher specification:

- Comparability between values of different types is defined. This deviation is particularly pronounced when it occurs as part of the evaluation of predicates (in WHERE).
- ORDER BY will not fail if the values passed to it have different types.

The underlying conceptual model is complex and sometimes inconsistent. This leads to an unclear relationship between comparison operators, equality, grouping, and ORDER BY:

- Comparability and orderability are aligned with each other consistently, as all types can be ordered and compared.
- The difference between equality and equivalence, as exposed by IN, =, DISTINCT, and grouping, in AGE is limited to testing two instances of the value null to each other
    - In equality, null = null is null.
    - In equivalence, used by DISTINCT and when grouping values, two null values are always treated as being the same value.
    - However, equality treats null values differently if they are an element of a list or a map value.

Concepts

The openCypher specification features four distinct concepts related to equality and ordering:

Comparability

Comparability is used by the inequality operators (>, <, >=, <=), and defines the underlying semantics of how to compare two values.

Equality

Equality is used by the equality operators (=, <>), and the list membership operator (IN). It defines the underlying semantics to determine if two values are the same in these contexts. Equality is also used implicitly by literal maps in node and relationship patterns, since such literal maps are merely a shorthand notation for equality predicates.

Orderability

Orderability is used by the ORDER BY clause, and defines the underlying semantics of how to order values.

Equivalence

Equivalence is used by the DISTINCT modifier and by grouping in projection clauses (WITH,RETURN), and defines the underlying semantics to determine if two values are the same in these contexts.

Comparability and equality

Comparison operators need to function as one would expect comparison operators to function - equality and comparability. But, at the same time, they need to allow the sorting of column data - equivalence and orderability.

Unfortunately, it may not be possible to implement separate comparison operators for equality and comparison operations, and, equivalence and orderability operations, in PostgreSQL, for the same query. So we prioritize equivalence and orderability over equality and comparability to allow for ordering of output data.

Comparability

Comparability is defined between any pair of values, as specified below.

- Numbers
    - Numbers of different types (excluding NaN values and the Infinities) are compared to each other as if both numbers would have been coerced to arbitrary precision big decimals(currently outside the Cypher type system) before comparing them with each other numerically in ascending order.
    - Comparison to any value that is not also Number follows the rules of orderability.
    - Floats don't have the required precision to represent all of the whole numbers in the range of agtype integer and agtype numeric. When casting an integer or numeric agtype to a float, unexpected results can occur when casting values in the high and low range
    - Integers
        - Integers are compared numerically in ascending order.
    - Floats
        - Floats (excluding NaN values and the Infinities) are compared numerically in ascending order.
        - Positive infinity is of type FLOAT, equal to itself and greater than any other number, except NaN values.
        - Negative infinity is of type FLOAT, equal to itself and less than any other number.
        - NaN values are comparable to each and greater than any other float value.
    - Numeric
        - Numerics are compared numerically in ascending order.
- Booleans
    - Booleans are compared such that false is less than true.
    - Comparison to any value that is not also a boolean follows the rules of orderability.
- Strings
    - Strings are compared in dictionary order, i.e. characters are compared pairwise in ascending order from the start of the string to the end. Characters missing in a shorter string are considered to be less than any other character. For example, 'a' < 'aa'.
    - Comparison to any value that is not also a string follows the rules of orderability.
- Lists
    - Lists are compared in sequential order, i.e. list elements are compared pairwise in ascending order from the start of the list to the end. Elements missing in a shorter list are considered to be less than any other value (including null values). For example, [1] < [1, 0]but also [1] < [1, null].
    - Comparison to any value that is not also a list follows the rules of orderability.
- Maps
    - The comparison order for maps is unspecified and left to implementations.

    - The comparison order for maps must align with the equality semantics outlined below. In consequence, any map that contains an entry that maps its key to a null value is incomparable. For example, {a: 1} <= {a: 1, b: null} evaluates to null.
    - Comparison to any value that is not also a regular map follows the rules of orderability.

Entities

- Vertices
    - The comparison order for vertices is based on the assigned graphid.
- Edges
    - The comparison order for edges is based on the assigned graphid.
- Paths
    - Paths are compared as if they were a list of alternating nodes and relationships of the path from the start node to the end node. For example, given nodes n1, n2, n3, and relationships r1and r2, and given that n1 < n2 < n3 and r1 < r2, then the path p1 from n1 to n3 via r1 would be less than the path p2 to n1 from n2 via r2.
    - Expressed in terms of lists:

```
p1 < p2
<=> [n1, r1, n3] < [n1, r2, n2]
<=> n1 < n1 || (n1 = n1 && [r1, n3] < [r2, n2])
<=> false || (true && [r1, n3] < [r2, n2])<=> [r1, n3] < [r2, n2]
<=> r1 < r2 || (r1 = r2 && n3 < n2)
<=> true || (false && false)
<=> true
```

    - Comparison to any value that is not also a path will return false.
- NULL
    - null is incomparable with any other value (including other null values.)

Orderability Between different Agtypes

The ordering of different Agtype, when using <, <=, >, >= from smallest value to largest value is:

1. Edge
2. Path
3. Map
4. Vertex
5. Edge
6. Array
7. String
8. Bool
9. Numeric, Integer, Float
10. NULL

Note: This is subject to change in future releases.

Operator Precedence

Operator precedence in AGE is shown below:

Precedence	Operator	
1	.	Property Access
2	[]	Map and List Subscripting
	()	Function Call
3	STARTS WITH	Case-sensitive prefix searching on strings
	ENDS WITH	Case-sensitive suffix searching on strings
	CONTAINS	Case-sensitive inclusion searching on strings
4	-	Unary Minus
5	IN	Checking if an element exists in a list
	IS NULL	Checking a value is NULL
	IS NOT NULL	Checking a value is not NULL
6	^	Exponentiation
7	* / %	Multiplication, division and remainder
8	+ -	Addition and Subtraction
9	= <>	For relational = and ≠ respectively
	< <=	For relational < and ≤ respectively
	> >=	For relational > and ≥ respectively
10	NOT	Logical NOT
11	AND	Logical AND

12	OR	Logical OR

The AGE Cypher Query Format

Cypher queries are constructed using a function called cypher in ag_catalog which returns a Postgres SETOF records.

Cypher()

Cypher() executes the cypher query passed as an argument.

Syntax `cypher(graph_name, query_string, parameters)`

Returns

```
A SETOF records
```

Arguments:

Argument Name	Description
graph_name	The target graph for the Cypher query.
query_string	The Cypher query to be executed.
parameters	An optional map of parameters used for stored procedure. Default is NULL. See Stored Procedures for details

Considerations:

- If a Cypher query does not return results, a record definition still needs to be defined. Example.

Query:

```
SELECT * FROM cypher('graph_name', $$
/* Cypher Query Here */
$$) AS (result1 agtype, result2 agtype);
```

Cypher in an Expression

Cypher may not be used as part of an expression, use a subquery instead. See Advanced Cypher Queries for information about how to use Cypher queries with Expressions

SELECT Clause

Calling Cypher in the SELECT clause as an independent column is not allowed. However Cypher may be used when it belongs as a conditional.

Not Allowed:

```
SELECT
    cypher('graph_name', $$
        MATCH (v:Person)
        RETURN v.name
     $$);
```

```
ERROR:  cypher(...) in expressions is not supported
LINE 3:     cypher('graph_name', $$
            ^
HINT:  Use subquery instead if possible.
```

Using CREATE with CTEs

Query:

```
WITH graph_query as (
    SELECT *
        FROM cypher('graph_name', $$
        MATCH (n), (m)
        WHERE n.name = 'A' AND m.name = 'B'
        CREATE (n)-[r:RELTYPE {name: n.name + '->' + m.name }]->(m)
        RETURN n.value, m.value, r.name
    $$) as (n_value agtype, m_value agtype, edge_name agtype)
)
SELECT * FROM graph_query
JOIN schema_name.table_name t
ON t.name = qraph_query.name;
```

Type Casting

Type Casting in Cypher

You can typecaste one agtype type to another with the syntax: `::datatype`
Query

```
SELECT * FROM cypher('graph_name', $$
    RETURN 1::float
$$) AS (float_result VARCHAR(50));
```

The float value 1.0 will be returned.

float_result
1.0
1 row(s) returned

Type Casting to Postgres Data Types

Some Agtype values can be converted to built in Postgres types, other types are currently not supported.

Agtype	**Postgres Data Type**
int	Cannot be cast
string	Varchar and Char
bool	boolean
float	float
numeric	Cannot be cast
Vertex	Cannot be cast
Edge	Cannot be cast
Path	Cannot be cast

Type Coercion With the Cypher Function

The cypher function call is capable of coercing agtype to certain postgres types.

Query:

```
SELECT * FROM cypher('graph_name', $$
    RETURN 'this is a string'
$$) AS (string_result agtype);
```

Query:

```
SELECT * FROM cypher('graph_name', $$
    RETURN 'this is a string'
$$) AS (string_result VARCHAR(50));
```

Aggregation

Aggregation does not currently support grouping by non-aggregate columns. Any reference to a non-aggregate value in a RETURN statement that contains an aggregate function will be ambiguous and non-deterministic.

> **Developers Note**
> Aggregation will be heavily updated in the next release.

See aggregation functions for more details.

Prepared Statements

Cypher can run a read query within a Prepared Statement. When using parameters with stored procedures, An SQL Parameter must be placed in the cypher function call. See The AGE Query Format for details.

```
Developers Note
CREATE, SET, and REMOVE are not currently compatible with Stored
Procedures.
```

Cypher Parameter Format

A cypher parameter is in the format of a '$' followed by an identifier. Unlike Postgres parameters, Cypher parameters start with a letter, followed by an alphanumeric string of arbitrary length.

Example: `$parameter_name`

Prepared Statements Preparation

Preparing Prepared Statements in cypher is an extension of Postgres' stored procedure system. Use the PREPARE clause to create a query with the Cypher Function call in it. Do not place Postgres style parameters in the cypher query call, instead place Cypher parameters in the query and place a Postgres parameter as the third argument in the Cypher function call.

```
PREPARE cypher_stored_procedure(agtype) AS
SELECT *
FROM cypher('expr', $$
    MATCH (v:Person)
    WHERE v.name = $name //Cypher parameter
    RETURN v
$$, $1) //An SQL Parameter must be placed in the cypher function call
AS (v agtype);
```

Prepared Statements Execution

When executing the prepared statement, place an agtype map with the parameter values where the Postgres Parameter in the Cypher function call is. The value must be an agtype map or an error will be thrown. Exclude the '$' for parameter names.

```
EXECUTE cypher_prepared_statement('{"name": "Tobias"}');
```

Clauses

MATCH

The MATCH clause allows you to specify the patterns Cypher will search for in the database. This is the primary way of getting data into the current set of bindings. It is worth reading up more on the specification of the patterns themselves in Patterns.

MATCH is often coupled to a WHERE part which adds restrictions, or predicates, to the MATCH patterns, making them more specific. The predicates are part of the pattern description, and should not be considered a filter applied only after the matching is done. This means that WHERE should always be put together with the MATCH clause it belongs to.

MATCH can occur at the beginning of the query or later, possibly after a WITH. If it is the first clause, nothing will have been bound yet, and Cypher will design a search to find the results matching the clause and any associated predicates specified in any WHERE part. Vertices and edges found by this search are available as bound pattern elements, and can be used for pattern matching of sub-graphs. They can also be used in any future clauses, where Cypher will use the known elements, and from there find further unknown elements.

Cypher is declarative, and so usually the query itself does not specify the algorithm to use to perform the search. Predicates in WHERE parts can be evaluated before pattern matching, during pattern matching, or after finding matches.

Basic vertex finding

Get all Vertices

By just specifying a pattern with a single vertex and no labels, all vertices in the graph will be returned.

Query

```
SELECT * FROM cypher('graph_name', $$
MATCH (v)
RETURN v
$$) as (v agtype);
```

Returns all the vertices in the database.

v
{id: 0; label: 'Person'; properties{name: 'Charlie Sheen'}}::vertex
{id: 1; label: 'Person'; properties{name: 'Martin Sheen'}}::vertex
{id: 2; label: 'Person'; properties{name: 'Michael Douglas'}}::vertex
{id: 3; label: 'Person'; properties{name: 'Oliver Stone'}}::vertex
{id: 4; label: 'Person'; properties{name: 'Rob Reiner'}}::vertex

{id: 5; label: ‘Movie’; properties{name: ‘Wall Street’}}::vertex
{id: 6; label: ‘Movie’; properties{title: ‘The American President’}}::vertex
7 row(s) returned

Get all vertices with a label

Getting all vertices with a label on them is done with a single node pattern where the vertex has a label on it.

Query

```
SELECT * FROM cypher('graph_name', $$
MATCH (movie:Movie)
RETURN movie.title
$$) as (title agtype);
```

Returns all the movies in the database.

title
‘Wall Street’
‘The American President’
2 row(s) returned

Related Vertices

The symbol -[]- means related to, without regard to type or direction of the edge.

Query

```
SELECT * FROM cypher('graph_name', $$
MATCH (director {name: 'Oliver Stone'})-[]-(movie)
RETURN movie.title
$$) as (title agtype);
```

Returns all the movies directed by 'Oliver Stone'

title

‘Wall Street’
1 row(s) returned

Match with labels

To constrain your pattern with labels on vertices, you add it to your vertex in the pattern, using the label syntax.

Query

```
SELECT * FROM cypher('graph_name', $$
MATCH (:Person {name: 'Oliver Stone'})-[]-(movie:Movie)
RETURN movie.title
$$) as (title agtype);
```

Returns any vertices connected with the Person 'Oliver' that are labeled Movie.

title
‘Wall Street’
1 row(s) returned

Edge basics

Outgoing Edges

When the direction of an edge is of interest, it is shown by using -> or <-.

Query

```
SELECT * FROM cypher('graph_name', $$
MATCH (:Person {name: 'Oliver Stone'})-[]->(movie)
RETURN movie.title
$$) as (title agtype);
```

Returns any vertices connected with the Person'Oliver' by an outgoing edge.

title

‘Wall Street’
1 row(s) returned

Directed Edges and variable

If a variable is required, either for filtering on properties of the edge, or to return the edge, this is how you introduce the variable.

Query

```
SELECT * FROM cypher('graph_name', $$
MATCH (:Person {name: 'Oliver Stone'})-[r]->(movie)
RETURN type(r)
$$) as (type agtype);
```

Returns the type of each outgoing edge from 'Oliver'.

title
‘DIRECTED’
1 row(s) returned

Match on edge type

When you know the edge type you want to match on, you can specify it by using a colon together with the edge type.

Query

```
SELECT * FROM cypher('graph_name', $$
MATCH (:Movie {title: 'Wall Street'})<-[:ACTED_IN]-(actor)
RETURN actor.name
$$) as (actors_name agtype);
```

Returns all actors that ACTED_IN'Wall Street'.

actors_name
‘Charlie Sheen’
‘Martin Sheen’

‘Michael Douglas’
3 row(s) returned

Match on edge type and use a variable

If you both want to introduce a variable to hold the edge, and specify the edge type you want, just add them both.

Query

```
SELECT * FROM cypher('graph_name', $$
MATCH ({title: 'Wall Street'})<-[r:ACTED_IN]-(actor)
RETURN r.role
$$) as (role agtype);
```

Returns ACTED_IN roles for 'Wall Street'.

role
‘Gordon Gekko’
‘Carl Fox’
‘Bud Fox’
3 row(s) returned

Multiple Edges

Edges can be expressed by using multiple statements in the form of ()-[]-(), or they can be strung together.

Query

```
SELECT * FROM cypher('graph_name', $$
    MATCH (charlie {name: 'Charlie
Sheen'})-[:ACTED_IN]->(movie)<-[:DIRECTED]-(director)
    RETURN movie.title, director.name
$$) as (title agtype, name agtype);
```

Returns the movie 'Charlie Sheen' acted in and its director.

title	name
‘Wall Street’	‘Oliver Stone’
1 row(s) returned	

WITH

Introduction
Using WITH, you can manipulate the output before it is passed on to the following query parts. The manipulations can be of the shape and/or number of entries in the result set.

WITH can also, like RETURN, alias expressions that are introduced into the results using the aliases as the binding name.

WITH is also used to separate the reading of the graph from updating of the graph. Every part of a query must be either read-only or write-only. When going from a writing part to a reading part, the switch can be done with an optional WITH clause.

Filter on results

Results passed through a WITH clause can be filtered on.

Query

```
SELECT *
FROM cypher('graph_name', $$
MATCH (david {name: 'David'})-[:FRIEND]-(otherPerson)
WITH otherPerson.name as name, otherPerson.age as age,
otherPerson.freetonight as free_tonight
WHERE age > 21 and free_tonight = TRUE
RETURN name
$$) as (name agtype);
```

The name of the person connected to 'David' with the at least more than one outgoing relationship will be returned by the query.

Result

name
"Anders"

1 row

RETURN

In the RETURN part of your query, you define which parts of the pattern you are interested in. It can be nodes, relationships, or properties on these.

Return nodes

To return a node, list it in the RETURN statement.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n {name: 'B'})
    RETURN n
$$) as (n agtype);
```

The example will return the node.

Result

n
{id: 0; label: ‘’ properties: {name: ‘B’}}::vertex
(1 row)

Return edges

To return n edge, just include it in the RETURN list.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n)-[r:KNOWS]->()
    WHERE n.name = 'A'
    RETURN r
$$) as (r agtype);
```

The relationship is returned by the example.

r
{id: 2; startid: 0; endid: 1; label: 'KNOWS' properties: {}}::edge
(1 row)

Return property

To return a property, use the dot separator, like this:

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n {name: 'A'})
    RETURN n.name
$$) as (name agtype);
```

The value of the property name gets returned.

Result

name
'A'
(1 row)

Variable with uncommon characters

To introduce a placeholder that is made up of characters that are not contained in the English alphabet, you can use the ` to enclose the variable, like this:

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (`This isn\'t a common variable`)
    WHERE `This isn\'t a common variable`.name = 'A'
    RETURN `This isn\'t a common variable`.happy
```

```
$$) as (happy agtype);
```

The node with name "A" is returned.

Result

happy
"Yes!"
(1 row)

Aliasing a field

If the name of the field should be different from the expression used, you can rename it by changing the name in the column list definition.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n {name: 'A'})
    RETURN n.name
$$) as (objects_name agtype);
```

Returns the age property of a node, but renames the field.

Result

objects_name
‘A’
(1 row)

Optional properties

If a property might or might not be there, you can still select it as usual. It will be treated as null if it is missing.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n)
```

```
    RETURN n.age
$$) as (age agtype);
```

This example returns the age when the node has that property, or null if the property is not there.

Result

age
55
NULL
(2 rows)

Other expressions

Any expression can be used as a return item—literals, predicates, properties, functions, and everything else.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (a)
    RETURN a.age > 30, 'I'm a literal', id(a)
$$) as (older_than_30 agtype, literal agtype, id agtype);
```

Returns a predicate, a literal and function call with a pattern expression parameter.

Result

older_than_30	literal	id
true	'I'm a literal'	1
(1 row)		

Unique results

DISTINCT retrieves only unique records depending on the fields that have been selected to output.

Query

```
SELECT *
FROM cypher('graph_name', $$
MATCH (a {name: 'A'})-[]->(b)
RETURN DISTINCT b
$$) as (b agtype);
```

The node named "B" is returned by the query, but only once.

Result

b
{id: 1; label: ‘’ properties: {name: ‘B’}}::vertex
(1 row)

ORDER BY

ORDER BY is a sub-clause following WITH, and it specifies that the output should be sorted and how.

```
Developers Note
In future releases, ORDER BY will be compatible with the RETURN clause, but
is currently not available.
```

Introduction

Note that you cannot sort on nodes or relationships, just on properties on these. ORDER BY relies on comparisons to sort the output, see Ordering and comparison of values. In terms of scope of variables, ORDER BY follows special rules, depending on if the projecting RETURN or WITH clause is either aggregating or DISTINCT. If it is an aggregating or DISTINCT projection, only the variables available in the projection are available. If the projection does not alter the output cardinality (which aggregation and DISTINCT do), variables available from before the projecting clause are also available. When the projection clause shadows already existing

variables, only the new variables are available.Lastly, it is not allowed to use aggregating expressions in the ORDER BY sub-clause if they are not also listed in the projecting clause. This last rule is to make sure that ORDER BY does not change the results, only the order of them

Order nodes by property

ORDER BY is used to sort the output.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n)
    WITH n.name as name, n.age as age
    ORDER BY n.name
    RETURN name, age
$$) as (name agtype, age agtype);
```

The nodes are returned, sorted by their name.

Result

name	age
"A"	34
"B"	34
"C"	32
(1 row)	

Order nodes by multiple properties

You can order by multiple properties by stating each variable in the ORDER BY clause. Cypher will sort the result by the first variable listed, and for equal values, go to the next property in the ORDER BY clause, and so on.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n)
    WITH n.name as name, n.age as age
```

```
    ORDER BY n.age, n.name
    RETURN name, age
$$) as (name agtype, age agtype);
```

This returns the nodes, sorted first by their age, and then by their name.

Result

name	age
"C"	32
"A"	34
"B"	34
(1 row)	

Order nodes in descending order

By adding DESC[ENDING] after the variable to sort on, the sort will be done in reverse order.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n)
    WITH n.name AS name, n.age AS age
    ORDER BY n.name DESC
    RETURN name, age
$$) as (name agtype, age agtype);
```

The example returns the nodes, sorted by their name in reverse order.

Result

name	age
"C"	32
"B"	34
"A"	34
(3 rows)	

Ordering null

When sorting the result set, null will always come at the end of the result set for ascending sorting,and first when doing descending sort.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n)
    WITH n.name AS name, n.age AS age, n.height
    ORDER BY n.height
    RETURN name, age, height
$$) as (name agtype, age agtype, height agtype);
```

The nodes are returned sorted by the length property, with a node without that property last.

Result

name	**age**	
"A"	34	170
"C"	32	185
"B"	34	<NULL>
(3 rows)		

WHERE

WHERE is a subclause of MATCH that puts restrictions on the MATCH clause.

In the case of multiple MATCH clauses, the predicate in WHERE is always a part of the patterns in the directly preceding MATCH. Both results and performance may be impacted if the WHERE is associated with the wrong MATCH clause.

Filter on a Vertex Property

To filter on a vertex property, write your clause after the WHERE keyword.

Query

```
SELECT * FROM cypher('graph_name', $$
    MATCH (n)
    WHERE n.age < 30
    RETURN n.name, n.age
$$) as (name agtype, age agtype);
```

The name and age values for the 'Tobias' node are returned because he is less than 30 years of age.

name	age
'Tobias'	26
1 row(s) returned	

Filter on an Edge Property

To filter on an edge property, write your clause after the WHERE keyword.

Query:

```
SELECT * FROM cypher('graph_name', $$
    MATCH (n)-[k:KNOWS]->(f)
    WHERE k.since < 2000
    RETURN f.name, f.age, f.email
$$) as (name agtype, age agtype, email agtype);
```

The name, age and email values for the 'Peter' node are returned because Andrés has known him since before 2000.

name	age	email
'Peter'	35	'peter_n@example.com'
1 row(s) returned		

Using Multiple Filters

You can use the boolean operators AND, OR and NOT.

Query:

```
SELECT * FROM cypher('graph_name', $$
    MATCH (n)
    WHERE (n.age < 30 AND n.name = 'Tobias')
        OR NOT (n.name ='Tobias' OR n.name = 'Peter')
    RETURN n.name, n.age
$$) as (name agtype, age agtype);
```

Results:

name	age
'Andres'	36
'Tobias'	25
'Peter'	35
3 row(s) returned	

Filter On Properties With STARTS WITH

Perform case-sensitive prefix searching on strings

Query:

```
SELECT * FROM cypher('graph_name', $$
    MATCH (n)
    WHERE n.name STARTS WITH 'P'
    RETURN n.name, n.age
$$) as (name agtype, age agtype);
```

Results:

name	age
'Peter'	35
3 row(s) returned	

Filter On Properties With ENDS WITH

Perform case-sensitive suffix searching on strings

Query:

```
SELECT * FROM cypher('graph_name', $$
    MATCH (n)
    WHERE n.name ENDS WITH 's'
    RETURN n.name, n.age
$$) as (name agtype, age agtype);
```

Results:

name	age
'Andres'	36
'Tobias'	25
3 row(s) returned	

Filter On Properties With CONTAINS

Perform case-sensitive inclusion searching in strings

Query:

```
SELECT * FROM cypher('graph_name', $$
    MATCH (n)
    WHERE n.name CONTAINS 'e'
    RETURN n.name, n.age
$$) as (name agtype, age agtype);
```

Results:

name	age
'Andres'	36
'Peter'	35
3 row(s) returned	

Filter On Properties With Exists

Use the exists() function to only include nodes or relationships in which a property exists.

Query:

```
SELECT * FROM cypher('graph_name', $$
MATCH (n)
WHERE exists(n.belt)
RETURN n.name, n.belt
$$) as (name agtype, belt agtype);
```

The name and belt for the 'Andres' node are returned because he is the only one with a belt property

name	**belt**
'Andres'	'white'
1 row(s) returned	

Filter On Patterns With Exists

Patterns are not only expressions, they are also predicates when used with the EXISTS subclause. The only limitation to your pattern is that you must be able to express it in a single path. You cannot use commas between multiple paths like you do in MATCH. You can achieve the same effect by combining multiple patterns with AND. Note that you cannot introduce new variables here.

Although it might look very similar to the MATCH patterns, the WHERE clause is all about eliminating matched subgraphs. `MATCH (a)-[]->(b)` is very different from `WHERE EXISTS((a)-[]->(b))`. The first will produce a subgraph for every path it can find between a and b, whereas the latter will eliminate any matched subgraphs where a and b do not have a directed relationship chain between them.

Query:

```
SELECT * FROM cypher('graph_name', $$
    MATCH (tobias {name: 'Tobias'}), (others)
    WHERE EXISTS((tobias)<-[]-(others))
```

```
    RETURN others.name, others.age
$$) as (name agtype, age agtype);
```

name	age
'Andres'	36
1 row(s) returned	

Filter on patterns using NOT

The NOT operator can be used to exclude a pattern.

Query:

```
SELECT * FROM cypher('graph_name', $$
    MATCH (persons), (peter {name: 'Peter'})
    WHERE NOT EXISTS((persons)-[]->(peter))
    RETURN persons.name, persons.age
$$) as (name agtype, age agtype);
```

Name and age values for nodes that do not have an outgoing relationship to the 'Peter' node are returned.

name	age
'Tobias'	25
'Peter'	35
2 row(s) returned	

Filter on patterns with properties

You can also add properties to your patterns.

Query

```
SELECT * FROM cypher('graph_name', $$
MATCH (n)
WHERE EXISTS((n)-[:KNOWS]->({name: 'Tobias'}))
RETURN n.name, n.age
```

```
$$) as (name agtype, age agtype);
```

Finds all name and age values for nodes that have a KNOWS relationship to a node with the name 'Tobias.'

name	age
'Andres'	36
1 row(s) returned	

IN Operator with Lists

To check if an element exists in a list, you can use the IN operator.

Query

```
SELECT * FROM cypher('graph_name', $$
    MATCH (a)
    WHERE a.name IN ['Peter', 'Tobias']
    RETURN a.name, a.age
$$) as (name agtype, age agtype)
```

This query shows how to check if a property exists in a literal list

name	age
'Tobias'	25
'Peter'	35
2 row(s) returned	

CREATE

The CREATE clause is used to create graph vertices and edges.

Terminal CREATE clauses

A create clause that is not followed by another clause is called a terminal clause. When a cypher query ends with a terminal clause, no results will be returned from the cypher function call. However, the cypher function call still requires a column list definition. When cypher ends with a terminal node, define a single value in the column list definition: no data will be returned in this variable.

Query

```
SELECT *
FROM cypher('graph_name', $$
    CREATE /* Create clause here, no following clause */
$$) as (a agtype);
```

a
0 row(s) returned

Create Vertices

Create single vertex

Creating a single vertex is done by issuing the following query.

Query

```
SELECT *
FROM cypher('graph_name', $$
    CREATE (n)
$$) as (v agtype);
```

Nothing is returned from this query.

v
(0 rows)

Create multiple vertices

Creating multiple vertices is done by separating them with a comma.

Query

```
SELECT *
FROM cypher('graph_name', $$
    CREATE (n), (m)
$$) as (v agtype);
```

Result

a
0 row(s) returned

Create a vertex with a label

To add a label when creating a vertex, use the syntax below.

Query

```
SELECT *
FROM cypher('graph_name', $$
    CREATE (:Person)
$$) as (v agtype);
```

Nothing is returned from this query.

Result

v
0 row(s) returned

Create vertex and add labels and properties

When creating a new vertex with labels, you can add properties at the same time.

Query

```
SELECT *
FROM cypher('graph_name', $$
    CREATE (:Person {name: ‘Andres’, title: ‘Developer’)
$$) as (n agtype);
```

Nothing is returned from this query.

Result

n
(0 rows)

Return created node

Creating a single node is done by issuing the following query.

Query

```
SELECT *
FROM cypher('graph_name', $$
    CREATE (a {name: ‘Andres’)
    RETURN a
$$) as (a agtype);
```

The newly-created node is returned.

Result

a
{id: 0; label: ‘’; properties: {name: ‘Andres’}}::vertex
(1 row)

Create Edges

Create an edge between two nodes

To create an edge between two vertices, we first get the two vertices. Once the nodes are loaded, we simply create an edge between them.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (a:Person), (b:Person)
    WHERE a.name = 'Node A' AND b.name = 'Node B'
    CREATE (a)-[e:RELTYPE]->(b)
    RETURN e
$$) as (e agtype);
```

The created edge is returned by the query.

Result

e
{id: 3; startid: 0, endid: 1; label: 'RELTYPE'; properties: {}}::edge
(1 row)

Create an edge and set properties

Setting properties on edges is done in a similar manner to how it's done when creating vertices. Note that the values can be any expression.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (a:Person), (b:Person)
    WHERE a.name = 'Node A' AND b.name = 'Node B'
    CREATE (a)-[e:RELTYPE {name:a.name + '<->' + b.name}]->(b)
    RETURN e
$$) as (e agtype);
```

The newly-created edge is returned by the example query.

Result

e
{id: 3; startid: 0, endid: 1; label: 'RELTYPE'; properties: {name: 'Node A<->Node B'}}::edge
(1 row)

Create a full path

When you use CREATE and a pattern, all parts of the pattern that are not already in scope at this time will be created.

Query

```
SELECT *
FROM cypher('graph_name', $$
    CREATE p = (andres
{name:'Andres'})-[:WORKS_AT]->(neo)<-[:WORKS_AT]-(michael {name:'Michael'})
    RETURN p
$$) as (p agtype);
```

This query creates three nodes and two relationships in one go, assigns it to a path variable, and returns it.

Result

p
[{id:0; label: ‘’; properties:{name:’Andres’}}::vertex, {id: 3; startid: 0, endid: 1; label: ‘WORKS_AT’; properties: {}}::edge, {id:1; label: ‘’; properties: {}}::vertex {id: 3; startid: 2, endid: 1; label: ‘WORKS_AT’; properties: {}}::edge, {id:2; label: ‘’; properties: {name:’Michael’}}::vertex]::path
(1 row)

SET

The SET clause is used to update labels on nodes and properties on vertices and edges

Terminal SET clauses

A set clause that is not followed by another clause is called a terminal clause. When a cypher query ends with a terminal clause, no results will be returned from the cypher function call. However, the cypher function call still requires a column list definition. When cypher ends with a terminal node, define a single value in the column list definition: no data will be returned in this variable.

Set a property

To set a property on a node or relationship, use SET.

Query

```
SELECT *
FROM cypher('graph_name', $$
   MATCH (v {name: 'Andres'})
   SET v.surname = 'Taylor'
$$) as (v agtype);
```

The newly changed node is returned by the query.

Result

v
(0 rows)

Return created vertex

Creating a single vertex is done by issuing the following query.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (v {name: 'Andres'})
    SET v.surname = 'Taylor'
    RETURN v
$$) as (v agtype);
```

The newly changed vertex is returned by the query.

Result

v
{id: 3; label: ‘Person’; properties: {surname:"Taylor", name:"Andres", age:36, hungry:true}}::vertex
(1 row)

Remove a property

Normally you remove a property by using REMOVE, but it's sometimes handy to do it using the SET command. One example is if the property comes from a parameter.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (v {name: 'Andres'})
    SET v.name = NULL
    RETURN v
$$) as (v agtype);
```

The node is returned by the query, and the name property is now missing.

Result

v
{id: 3; label: 'Person'; properties: {surname:"Taylor", age:36, hungry:true}}::vertex
(1 row)

REMOVE

The REMOVE clause is used to remove properties from vertex and edges.

Terminal REMOVE clauses

A remove clause that is not followed by another clause is called a terminal clause. When a cypher query ends with a terminal clause, no results will be returned from the cypher function call. However, the cypher function call still requires a column list definition. When cypher ends with a terminal node, define a single value in the column list definition: no data will be returned in this variable.

Remove a property

Cypher doesn't allow storing null in properties. Instead, if no value exists, the property is just not there. So, to remove a property value on a node or a relationship, is also done with REMOVE.113

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (andres {name: 'Andres'})
    REMOVE andres.age
    RETURN andres
$$) as (andres agtype);
```

The node is returned, and no property age exists on it.

Result

andres
{id: 3; label: ‘Person’; properties: {name:"Andres"}}::vertex
1 row(s) returned

Functions

Built-In functions:

All AGE functions are defined in the ag_catalog namespace. When using built-in functions in a cypher query, the default namespace is always ag_catalog.This is independent of the configuration of search_path. These functions can be used outside a cypher command. No functions in ag_catalog that are not defined in this section can be used in a cypher command.

Predicate Functions

Predicates are boolean functions that return true or false for a given set of input. They are most commonly used to filter out subgraphs in the WHERE part of a query.

Exists(Property)

exists() returns true if the specified property exists in the node, relationship or map. This is different from the EXISTS clause.

Syntax:exists(property)

Returns:
An agtype boolean

Arguments:

Name	Description
property	A property from a vertex or edge

Query

```
SELECT *
FROM cypher('graph_name', $$
     MATCH (n)
     WHERE exists(n.surname)
     RETURN n.first_name, n.last_name
$$) as (first_name agtype, last_name agtype);
```

Results:

first_name	last_name
‘John	‘Smith’
‘Patty’	‘Patterson’
2 row(s) returned	

Scalar Functions

id()

id() returns the id of a vertex or edge.

Syntax:`id(expression)`

Returns:

```
An agtype integer
```

Arguments:

Name	Description
expression	An expression that returns a vertex or edge.

Considerations:

Query:

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (a)
    RETURN id(a)
$$) as (id agtype);
```

Results

id
0
1
2
3
4 row(s) returned

start_id()

start_id() returns the id of the vertex that is the starting vertex for the edge.

Syntax: `start_id(expression)`

Returns:

```
An agtype integer
```

Arguments:

Name	Description
expression	An expression that evaluates to an edge.

Considerations:

Query:

```
SELECT *
FROM cypher('graph_name', $$
    MATCH ()-[e]->()
    RETURN start_id(e)
$$) as (start_id agtype);
```

Results

start_id
0
1
2
3
4 row(s) returned

end_id()

end_id() returns the id of the vertex that is the ending vertex for the edge.

Syntax: `end_id(expression)`

Returns:

```
An agtype integer
```

Arguments:

Name	Description
expression	An expression that evaluates to an edge.

Query:

```
SELECT *
```

```
FROM cypher('graph_name', $$
    MATCH ()-[e]->()
    RETURN end_id(e)
$$) as (end_id agtype);
```

Results

end_id
4
5
6
7
4 row(s) returned

type()

type() returns the string representation of the edge type

Syntax: `type(edge)`

Returns:

```
An agtype string
```

Arguments:

Name	Description
edge	An expression that evaluates to an edge.

Considerations:

Query:

```
SELECT *
FROM cypher('graph_name', $$
    MATCH ()-[e]->()
```

```
    RETURN type(e)
$$) as (type agtype);
```

Results

type
“KNOWS”
“KNOWS”
2 row(s) returned

properties()

Returns an agtype map containing all the properties of a vertex or edge. If the argument is already a map, it is returned unchanged.

Syntax: `properties(expression)`

Returns:

```
An agtype Map.
```

Arguments:

Name	Description
Expression	An expression that returns a vertex, an edge, or an agtype map. Considerations: properties(null) returns null.

Considerations:

- properties(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    CREATE (p:Person {name: 'Stefan', city: 'Berlin'})
    RETURN properties(p)
$$) as (type agtype);
```

Results:

properties
{name: "Stefan"; city: "Berlin"}
1 row(s) returned

head()

returns the first element in an agtype list.

Syntax: `head(list)`

Returns:

```
The type of the value returned will be that of the first element of the
list.
```

Arguments:

Name	Description
List	An expression that returns a list

Considerations:

- head(null) returns null.
- If the first element in the list is null, head(list) will return null.

Query

```
SELECT *
FROM cypher('graph_name', $$
   MATCH (a)
   WHERE a.name = 'Eskil'
   RETURN a.array, head(a.array)
$$) as (lst agtype, lst_head agtype);
```

The first element in the list is returned.

Result:

lst	lst_head
["one","two","three"]	"one"
1 row(s) returned	

last()

returns the last element in an agtype list.

Syntax:`last(list)`

Returns:

```
The type of the value returned will be that of the last element of the
list.
```

Arguments:

Name	Description
List	An expression that returns a list

Considerations:

- tail(null) returns null.
- If the last element in the list is null, last(list) will return null.

Query

```
SELECT *
FROM cypher('graph_name', $$
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.array, last(a.array)
$$) as (lst agtype, lst_tail agtype);
```

The first element in the list is returned.

Result:

lst	lst_tail
["one","two","three"]	"three"

1 row(s) returned

length()

length() returns the length of a path.

Syntax: `length(path)`

Returns:

```
An agtype Integer.
```

Arguments:

Name	Description
path	An expression that returns a path.

Considerations:length(null) returns null.
Query

```
SELECT *
FROM cypher('graph_name', $$
   MATCH p = (a)-[]->(b)-[]->(c)
   WHERE a.name = 'Alice'
   RETURN length(p)
$$) as (length_of_path agtype);
```

The length of the path p is returned.

Results:

length_of_path
2
2
2
3 row(s) returned

size()

size() returns the length of a list.

Syntax:`size(path)`

Returns:

```
An agtype Integer.
```

Arguments:

Name	Description
list	An expression that returns a list.

Considerations:

- size(null) returns null.

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN size(['Alice', 'Bob'])
$$) as (size_of_list agtype);
```

The length of the path p is returned.

Results:

size_of_list
2
1 row(s) returned

startNode()

startNode() returns the start node of an edge.

Syntax:`startNode(edge)`

Returns:

```
A vertex.
```

Arguments:

Name	Description

edge	An expression that returns an edge.

Considerations:

- startNode(null) returns null.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (x:Developer)-[r]-()
    RETURN startNode(r)
$$) as (v agtype);
```

Result

v
Node[0]{name:"Alice",age:38,eyes:"brown"}
Node[0]{name:"Alice",age:38,eyes:"brown"}
2 row(s) returned

endNode()

endNode() returns the start node of an edge.

Syntax: `endNode(edge)`

Returns:

```
A vertex.
```

Arguments:

Name	Description
edge	An expression that returns an edge.

Considerations:

- endNode(null) returns null.

Query

```
SELECT *
```

```
FROM cypher('graph_name', $$
    MATCH (x:Developer)-[r]-()
    RETURN endNode(r)
$$) as (v agtype);
```

Result

v
Node[2]{name:"Charlie",age:53,eyes:"green"}
Node[1]{name:"Bob",age:25,eyes:"blue"}
2 row(s) returned

timestamp()

timestamp() returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.

Syntax:`timestamp()`

Returns:

```
An Agtype Integer.
```

Considerations:

- timestamp() will return the same value during one entire query, even for long-running queries.

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN timestamp()
$$) as (t agtype);
```

The time in milliseconds is returned.

Results:

t
1613496720760
1 row(s) returned

toBoolean()

toBoolean() converts a string value to a boolean value.

Syntax: `toBoolean(expression)`

Returns:

```
An agtype Boolean.
```

Arguments:

Name	Description
expression	An expression that returns a boolean or string value.

Considerations:

- toBoolean(null) returns null.
- If expression is a boolean value, it will be returned unchanged.
- If the parsing fails, null will be returned.

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN toBoolean('TRUE'), toBoolean('not a boolean')
$$) as (a_bool agtype, not_a_bool agtype);
```

Result:

a_bool	not_a_bool
true	NULL
1 row(s) returned	

toFloat()

toFloat() converts an integer or string value to a floating point number.

Syntax:toFloat(expression)

Returns:A Float.

Name	Description
expression	An expression that returns an agtype number or agtype string value.

Considerations:

- toFloat(null) returns null.
- If expression is a floating point number, it will be returned unchanged.
- If the parsing fails, null will be returned.

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN toFloat('11.5'), toFloat('not a number')
$$) as (a_float agtype, not_a_float agtype);
```

Result:

a_float	not_a_float
11.5	NULL
1 row(s) returned	

toInteger()

toInteger() converts a floating point or string value to an integer value.

Syntax:`toInteger(expression)`

Returns:

```
An agtype Integer.
```

Arguments

Name	Description
expression	An expression that returns an agtype number or agtype string value.

Considerations:

- toInteger(null) returns null.
- If expression is an integer value, it will be returned unchanged.
- If the parsing fails, null will be returned.

Query

```
SELECT *
FROM cypher('graph_name', $$
     RETURN toInteger('42'), toInteger('not a number')
$$) as (an_integer agtype, not_an_integer agtype);
```

Result:

an_integer	not_an_integer
42	NULL
1 row(s) returned	

coalesce()

coalesce() returns the first non-null value in the given list of expressions.

Syntax:`coalesce(expression [, expression]*)`

Returns:

```
The type of the value returned will be that of the first non-null
expression.
```

Arguments:

Name	Description
expression	An expression which may return null.

Considerations:

- null will be returned if all the arguments are null.

Query

```
SELECT *
```

```
FROM cypher('graph_name', $$
MATCH (a)
WHERE a.name = 'Alice'
RETURN coalesce(a.hairColor, a.eyes), a.hair_color, a.eyes
$$) as (color agtype, hair_color agtype, eyes agtype);
```

Table 105.
Result

color	hair_color	eyes
“brown”	NULL	“Brown”
1 row(s) returned		

Logarithmic Functions

e()

e() returns the base of the natural logarithm, e.

Syntax:`e()`

Returns:

An agtype float.

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN e()
$$) as (e agtype);
```

Results

e
2.71828182845905
1 row(s) returned

sqrt()

sqrt() returns the square root of a number.

Syntax:`sqrt(expression)`

Returns:

An agtype float.

Query

```
SELECT *
FROM cypher('graph_name', $$
    RETURN sqrt(144)
$$) as (results agtype);
```

Results

results
12
1 row(s) returned

exp()

exp() returns e^n, where e is the base of the natural logarithm, and n is the value of the argument expression.

Syntax: `e(expression)`

Returns:

```
An agtype Float.
```

Arguments:

Name	Description
expression	An agtype number expression

Considerations:

- exp(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN e(2)
$$) as (e agtype);
```

e to the power of 2 is returned.

Result:

e
7.38905609893065
1 row(s) returned

log()

log() returns the natural logarithm of a number.

Syntax:`log(expression)`

Returns:

```
An agtype Float.
```

Arguments:

Name	Description
expression	An agtype number expression

Considerations:

- log(null) returns null.
- log(0) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN log(27)
$$) as (natural_logarithm agtype);
```

The natural logarithm of 27 is returned.

Result:

natural_logarithm
3.295836866004329
1 row(s) returned

log10()

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

Syntax:`log10(expression)`

Returns:

```
An agtype Float.
```

Arguments:

Name	Description
expression	An agtype number expression

Considerations:

- log10(null) returns null.
- log10(0) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN log(27)
$$) as (natural_logarithm agtype);
```

The common logarithm of 27 is returned.

Result:

natural_logarithm
1.4313637641589874
1 row(s) returned

Trigonometric Functions

degrees()

degrees() converts radians to degrees.

Syntax:`degrees(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression that represents the angle in radians.

Considerations:

- degrees(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN degrees(3.14159)
$$) as (deg agtype);
```

The number of degrees in something close to pi is returned.

Results:

deg
179.99984796050427
1 row(s) returned

radians()

radians() converts radians to degrees.

Syntax:`radians(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression that represents the angle in degrees.

Considerations:

- radians(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN radians(180)
$$) as (rad agtype);
```

The number of degrees in something close to pi is returned.

Results:

rad
3.14159265358979
1 row(s) returned

pi()

pi() returns the mathematical constant pi.

Syntax: `pi()`

Returns:

```
An agtype float.
```

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN pi()
$$) as (p agtype);
```

The constant pi is returned.

Result:

p
3.141592653589793
1 row(s) returned

sin()

sin() returns the sine of a number.

Syntax:`sin(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression that represents the angle in radians.

Considerations:

- sin(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN sin(0.5)
$$) as (s agtype);
```

The sine of 0.5 is returned.

Results:

s
0.479425538604203
1 row(s) returned

cos()

cos() returns the cosine of a number.

Syntax: `cos(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype expression that represents the angle in radians.

Considerations:

- cos(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN cosin(0.5)
$$) as (c agtype);
```

The cosine of 0.5 is returned.

Results:

c
0.8775825618903728
1 row(s) returned

tan()

tan() returns the tangent of a number.

Syntax: `tan(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression that represents the angle in radians.

Considerations:

- tan(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN tan(0.5)
$$) as (t agtype);
```

The tangent of 0.5 is returned.

Results:

t
0.5463024898437905
1 row(s) returned

asin()

asin() returns the arcsine of a number.

Syntax:`asin(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression that represents the angle in radians.

Considerations:

- asin(null) returns null.
- If (expression < -1) or (expression > 1), then (asin(expression)) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN asin(0.5)
$$) as (s agtype);
```

The arcsine of 0.5 is returned.

Results:

s
0.523598775598299
1 row(s) returned

acos()

acos() returns the arccosine of a number.

Syntax:`acos(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression that represents the angle in radians.

Considerations:

- acos(null) returns null.
- If (expression < -1) or (expression > 1), then (acos(expression)) returns null.

Query:

```
SELECT *
```

```
FROM cypher('graph_name', $$
    RETURN acos(0.5)
$$) as (arc_c agtype);
```

The arccosine of 0.5 is returned.

Results:

arc_c
1.0471975511965979
1 row(s) returned

atan()

atan() returns the arctangent of a number.

Syntax:`atan(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression that represents the angle in radians.

Considerations:

- atan(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN atan(0.5)
$$) as (arc_t agtype);
```

The arccosine of 0.5 is returned.

Results:

arc_t
0.463647609000806
1 row(s) returned

atan2()

atan2() returns the arctangent2 of a set of coordinates in radians.

Syntax: `atan2(expression1, expression2)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression1	An agtype number expression for y that represents the angle in radians.
expression2	An agtype number expression for x that represents the angle in radians.

Considerations:

- atan2(null, null), atan2(null, expression2) and atan(expression1, null) all return null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN atan2(0.5, 0.6)
$$) as (arc_t2 agtype);
```

The arctangent2 of 0.5 and 0.6 is returned.

Results:

arc_t2
0.694738276196703

1 row(s) returned

String Functions

replace()

replace() returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.

Syntax: `replace(original, search, replace)`

Returns:

```
An agtype String.
```

Arguments:

Name	Description
original	An expression that returns a string.
search	An expression that specifies the string to be replaced in original.
replace	An expression that specifies the replacementstring.

Considerations:

- If any argument is null, null will be returned.
- If search is not found in original, original will be returned.

Query:

```
SELECT *
FROM cypher('graph_name', $$
      RETURN replace('hello', 'l', 'w')
$$) as (str_array agtype);
```

Result:

new_str
“Hewwo”
1 row(s) returned

split()

split() returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.

Syntax: `split(original, split_delimiter)`

Returns:

```
An agtype list of agtype strings.
```

Arguments:

Name	Description
original	An expression that returns a string.
split_delimiter	The string with which to split original.

Considerations:

- split(null, splitDelimiter) and split(original, null) both return null

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN split('one,two', ',')
$$) as (split_list agtype);
```

Result:

split_list
[“one”,”two”]
1 row(s) returned

left()

left() returns a string containing the specified number of leftmost characters of the original string.

Syntax: `left(original, length)`

Returns:

```
An agtype String.
```

Arguments:

Name	Description
original	An expression that returns a string.
n	An expression that returns a positive integer.

Considerations:

- left(null, length) and left(null, null) both return null
- left(original, null) will raise an error.
- If length is not a positive integer, an error is raised.
- If length exceeds the size of original, original is returned.

Query:

```
SELECT *
FROM cypher('graph_name', $$
      RETURN left('Hello', 3)
$$) as (new_str agtype);
```

Result:

new_str
“Hel”
1 row(s) returned

right()

right() returns a string containing the specified number of rightmost characters of the original string.

Syntax: `right(original, length)`

Returns:

```
An agtype String.
```

Arguments:

Name	Description
original	An expression that returns a string.

n	An expression that returns a positive integer.

Considerations:

- right(null, length) and right(null, null) both return null
- right(original, null) will raise an error.
- If length is not a positive integer, an error is raised.
- If length exceeds the size of original, original is returned.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN right('hello', 3)
$$) as (new_str agtype);
```

Result:

new_str
“llo”
1 row(s) returned

substring()

substring() returns a substring of the original string, beginning with a 0-based index start and length.

Syntax: `substring(original, start [, length])`

Returns:

```
An agtype String.
```

Arguments:

Name	Description
original	An expression that returns a string.
start	An expression denoting the position at which the substring will begin.
length	An optional expression that returns a positive integer, denoting how many characters of the original expression will be returned.

Considerations:

- start uses a zero-based index.
- If length is omitted, the function returns the substring starting at the position given by start and extending to the end of original.
- If original is null, null is returned.
- If either start or length is null or a negative integer, an error is raised.
- If start is 0, the substring will start at the beginning of original.
- If length is 0, the empty string will be returned.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN substring('hello', 1, 3), substring('hello', 2)
$$) as (sub_str1 agtype, sub_str2 agtype);
```

Result:

sub_str1	sub_str2
'ell'	'llo'
1 row(s) returned	

rTrim()

rTrim() returns the original string with trailing whitespace removed.

Syntax: rTrim(original)

Returns:

```
An agtype String.
```

Arguments:

Name	Description
original	An expression that returns a string

Considerations:

- rTrim(null) returns null

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN rTrim(' hello ')
$$) as (trimmed_str agtype);
```

Result:

trimmed_str
" hello"
1 row(s) returned

lTrim()

lTrim() returns the original string with trailing whitespace removed.

Syntax: `lTrim(original)`

Returns:

```
A String.
```

Arguments:

Name	Description
original	An expression that returns a string

Considerations:

- lTrim(null) returns null

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN lTrim(' hello ')
$$) as (trimmed_str agtype);
```

Result:

left_trimmed_str

“hello ”
1 row(s) returned

trim()

trim() returns the original string with leading and trailing whitespace removed.

Syntax: `trim(original)`

Returns:

```
An agtype String.
```

Arguments:

Name	Description
original	An expression that returns a string

Considerations:

- trim(null) returns null

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN trim(' hello ')
$$) as (trimmed_str agtype);
```

Result:

trimmed_str
“hello”
1 row(s) returned

toLower

toLower() returns the original string in lowercase.

Syntax: `toLower(original)`

Returns:

```
An agtype String.
```

Arguments:

Name	Description
original	An expression that returns a string

Considerations:

- toLower(null) returns null

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN toLower('HELLO')
$$) as (lower_str agtype);
```

Result:

lower_str
“hello”
1 row(s) returned

toUpper()

toUpper() returns the original string in lowercase.

Syntax: `toUpper(original)`

Returns:

```
An agtype String.
```

Arguments:

Name	Description
original	An expression that returns a string

Considerations:

- toUpper(null) returns null

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN toUpper('hello')
$$) as (upper_str agtype);
```

Result:

upper_str
“HELLO”
1 row(s) returned

reverse()

reverse() returns a string in which the order of all characters in the original string have been reversed.

Syntax: `reverse(original)`

Returns:

```
An agtype String.
```

Arguments:

Name	Description
original	An expression that returns a string

Considerations:

- reverse(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN reverse("hello")
$$) as (upper_str agtype);
```

Result:

upper_str
“olleh”

1 row(s) returned

toString()

toString() converts an integer, float or boolean value to a string.

Syntax:`toString(expression)`

Returns:

```
A String.
```

Arguments:

Name	Description
expression	An expression that returns a number, a boolean, or a string.

Considerations:

- toString(null) returns null
- If expression is a string, it will be returned unchanged.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN toString(11.5),toString('a string'), toString(true)
$$) as (float_to_str agtype, str_to_str agtype, bool_to_string);
```

Result:

float_to_str	str_to_str	bool_to_str
"11.5"	"a string"	"true"
1 row(s) returned		

Numeric Functions

rand()

rand() returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e.[0,1). The numbers returned follow an approximate uniform distribution.

Syntax: `rand()`

Returns:

```
A Float.
```

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN rand()
$$) as (r agtype);
```

A random number is returned.

Result:

r
0.3586784748902053
1 row(s) returned

abs()

abs() returns the absolute value of the given number.

Syntax: `abs(expression)`

Returns:

```
The type of the value returned will be that of expression.
```

Arguments:

Name	Description
expression	An agtype number expression

Considerations:

- abs(null) returns null.
- If expression is negative, -(expression) (i.e. the negation of expression) is returned.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (a), (e) WHERE a.name = 'Alice' AND e.name = 'Eskil'
    RETURN a.age, e.age, abs(a.age - e.age)
$$) as (alice_age agtype, eskil_age agtype, difference agtype);
```

The absolute value of the age difference is returned.

Result:

alice_age	eskil_age	difference
38	41	3
1 row(s) returned		

ceil()

ceil() returns the smallest floating point number that is greater than or equal to the given number and equal to a mathematical integer.

Syntax: `ceil(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression

Considerations:

- ceil(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN ceil(0.1)
$$) as (cil agtype);
```

The ceiling of 0.1 is returned.

Result:

cil
1
1 row(s) returned

floor()

floor() returns the greatest floating point number that is less than or equal to the given number and equal to a mathematical integer.

Syntax: `floor(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression

Considerations:

- floor(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN floor(0.1)
$$) as (flr agtype);
```

The floor of 0.1 is returned.

Result:

flr
0
1 row(s) returned

round()

round() returns the value of the given number rounded to the nearest integer.

Syntax: `round(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
expression	An agtype number expression

`

Considerations:

- round(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN round(3.141592)
$$) as (rounded_value agtype);
```

3.0 is returned.

Result:

rounded_value
3.0
1 row(s) returned

sign()

sign() returns the signum of the given number: 0 if the number is 0, -1 for any negative number, and 1 for any positive number

Syntax:sign(expression)

Returns:
An Integer.

Arguments:

Name	Description
expression	An agtype number expression

Considerations:

- sign(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    RETURN sign(-17), sign(0.1), sign(0)
$$) as (negative_sign agtype, positive_sign agtype, zero_sign agtype);
```

The signs of -17 and 0.1 are returned.

Result:

negative_sign	positive_sign	zero_sign
-1	1	0
1 row(s) returned		

Aggregation Functions

min()

min() returns the minimum value in a set of values.

Syntax: `min(expression)`

Returns:

```
A property type, or a list, depending on the values returned by expression.
```

Arguments:

Name	Description

expression	An expression returning a set containing any combination of property types and lists thereof.

Considerations:

- Any null values are excluded from the calculation.
- In a mixed set, any string value is always considered to be lower than any numeric value, and anylist is always considered to be lower than any string.
- Lists are compared in dictionary order, i.e. list elements are compared pairwise in ascending order from the start of the list to the end.
- min(null) returns null.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (v:Person)
    RETURN min(v.age)
$$) as (min_age agtype);
```

Result:

min_age
21
1 row(s) returned

Using min() with Lists

Data Setup:
To clarify the following example, assume the next three commands are run first:

```
SELECT * FROM cypher('graph_name', $$
    CREATE (:min_test {val:'d'})
$$) as (result agtype);

SELECT * FROM cypher('graph_name', $$
    CREATE (:min_test {val:['a', 'b', 23]})
$$) as (result agtype);

SELECT * FROM cypher('graph_name', $$
```

```
    CREATE (:min_test {val:['a', 'b', 23]})
$$) as (result agtype);
```

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (v:min_test)
    RETURN min(v.val)
$$) as (min_val agtype);
```

The lowest of all the values in the set—in this case, the list ['a', 'c', 23]—is returned, as (i) the two lists are considered to be lower values than the string "d", and (ii) the string "a" is considered tobe a lower value than the numerical value 1.

Result:

min_age
["a", "b", 23]
1 row(s) returned

max()

max() returns the maximum value in a set of values.

Syntax: `max(expression)`

Returns:

```
A property type, or a list, depending on the values returned by expression.
```

Arguments:

Name	Description
expression	An expression returning a set containing any combination of property types and lists thereof.

Considerations:

- Any null values are excluded from the calculation.
- In a mixed set, any numeric value is always considered to be higher than any string value, and anystring value is always considered to be higher than any list.
- Lists are compared in dictionary order, i.e. list elements are compared pairwise in ascending order from the start of the list to the end.
- max(null) returns null.

Query:

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n:Person)
    RETURN max(n.age)
$$) as (max_age agtype);
```

The highest of all the values in the property age is returned.

Result:

min_age
44
1 row(s) returned

stDev()

stDev() returns the standard deviation for the given value over a group. It uses a standard two-pass method, with N - 1 as the denominator, and should be used when taking a sample of the population for an unbiased estimate. When the standard variation of the entire population is being calculated, stdDevP should be used.

Syntax: `stDev(expression)`

Returns:

```
An agtype Float.
```

Arguments:

Name	Description

expression	An agtype number expression

Considerations:

- Any null values are excluded from the calculation.
- stDev(null) returns 0.

Query

```
SELECT *
FROM cypher('graph_name', $$
   MATCH (n:Person)
   RETURN stDev(n.age)
$$) as (stdev_age agtype);
```

The standard deviation of the values in the property age is returned.

Result:

stdev_age
15.716233645501712
1 row(s) returned

stDevP()

stDevP() returns the standard deviation for the given value over a group. It uses a standard two-pass method, with N as the denominator, and should be used when calculating the standard deviation for an entire population. When the standard variation of only a sample of the population is being calculated, stDev should be used.

Syntax: `stDevP(expression)`

Returns:

```
An agtype Float.
```

Arguments:

Name	Description
expression	An agtype number expression

Considerations:

- Any null values are excluded from the calculation.
- stDevP(null) returns 0.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n:Person)
    RETURN stDevP(n.age)
$$ as (stdevp_age agtype);
```

The population standard deviation of the values in the property age is returned.

Result:

stdevp_age
12.832251036613439
1 row(s) returned

percentileCont()

percentileCont() returns the percentile of the given value over a group, with a percentile from 0.0 to 1.0. It uses a linear interpolation method, calculating a weighted average between two values if the desired percentile lies between them. For nearest values using a rounding method, see percentileDisc.

Syntax: `percentileCont(expression, percentile)`

Returns:

```
An agtype Float.
```

Arguments:

Name	Description
expression	An agtype number expression
percentile	An agtype number value between 0.0 and 1.0

Considerations:

- Any null values are excluded from the calculation.
- percentileCont(null, percentile) returns null.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n:Person)
    RETURN percentileCont(n.age, 0.4)
$$ as (percentile_cont_age agtype);
```

The 40th percentile of the values in the property age is returned, calculated with a weighted average. In this case, 0.4 is the median, or 40th percentile.

Result:

percentile_cont_age
29.0
1 row(s) returned

percentileDisc()

percentileDisc() returns the percentile of the given value over a group, with a percentile from 0.0to 1.0. It uses a rounding method and calculates the nearest value to the percentile. For interpolated values, see percentileCont.

Syntax: `percentileDisc(expression, percentile)`

Returns:

```
An agtype Float.
```

Arguments:

Name	Description
expression	An agtype number expression
percentile	An agtype number value between 0.0 and 1.0

Considerations:

- Any null values are excluded from the calculation.
- percentileDisc(null, percentile) returns null.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n:Person)
    RETURN percentileDisc(n.age, 0.5)
$$ as (percentile_disc_age agtype);
```

The 50th percentile of the values in the property age is returned.

Result:

percentile_cont_age
33
1 row(s) returned

count()

count() returns the number of values or records, and appears in two variants:

- count(*) returns the number of matching records
- count(expr) returns the number of non-null values returned by an expression.

Syntax: `count(expression)`

Returns:

```
An agtype Integer.
```

Arguments:

Name	Description
expression	An expression

Considerations:

- count(*) includes records returning null.
- count(expr) ignores null values.
- count(null) returns 0.
- Using count(*) to return the number of nodes

- count(*) can be used to return the number of nodes; for example, the number of nodes connected to some node n.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n {name: 'A'})-[]->(x)
    RETURN n.age, count(*)
$$ as (age agtype, number_of_people agtype);
```

The labels and age property of the start node n and the number of nodes related to n are returned.

Result:

age	number_of_people
13	3
1 row(s) returned	

Using count(*) to group and count relationship typescount(*) can be used to group relationship types and return the number.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n {name: 'A'})-[r]->()
    RETURN type(r), count(*)
$$ as (label agtype, count agtype);
```

The relationship types and their group count are returned.

Result:

label	count
“KNOWS”	3
1 row(s) returned	

Using count(expression) to return the number of values

Instead of simply returning the number of records with count(*), it may be more useful to return the actual number of values returned by an expression.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n {name: 'A'})-[]->(x)
    RETURN count(x)
$$) as (count agtype);
```

The number of nodes connected to the start node is returned.

Result:

count
3
1 row(s) returned

Counting non-null values

count(expression) can be used to return the number of non-null values returned by the expression.

Query

```
SELECT *
FROM cypher('graph_name', $$
    MATCH (n:Person)
    RETURN count(n.age)
$$) as (count agtype);
```

The number of :Person nodes having an age property is returned.

Result:

count

3
1 row(s) returned

avg()

avg() returns the average of a set of numeric values.

Syntax: `avg(expression)`

Returns:

```
An agtype Integer
```

Arguments:

Name	Description
expression	An expression returning a set of numeric values.

Considerations:

- Any null values are excluded from the calculation.
- avg(null) returns null.

Query

```
SELECT *
FROM cypher('graph_name', $$
MATCH (n:Person)
RETURN avg(n.age)
$$) as (avg_age agtype);
```

The average of all the values in the property age is returned.

Result:

avg_age
30.0

1 row(s) returned

sum

sum() returns the sum of a set of numeric values.

Syntax: `sum(expression)`

Returns:

```
An agtype Float
```

Arguments:

Name	Description
expression	An expression returning a set of numeric values.

Considerations:

- Any null values are excluded from the calculation.
- sum(null) returns 0.

Query

```
SELECT *
FROM cypher('graph_name', $$
MATCH (n:Person)
RETURN sum(n.age)
$$ as (total_age agtype);
```

The sum of all the values in the property age is returned.

Result:

total_age
90

1 row(s) returned

User defined functions

Users may add custom functions to the AGE. When using the Cypher function, all function calls with a Cypher query use the default namespace of: ag_catalog. However if a user want to use a function outside this namespace, they may do so by adding the namespace before the function name.

Syntax: `namespace_name.function_name`

Query

```
SELECT *
FROM cypher('graph_name', $$
RETURN pg_catalog.sqrt(25)
$$ as (result agtype);
```

Result:

result
25
1 row(s) returned

Advanced Cypher Queries

All queries so far have followed the same pattern: a SELECT clause followed by a single Cypher call in the FROM clause. However, a Cypher query can be used in many other ways. This section highlights some more advanced ways of using the Cypher call within a more complex SQL/Cypher Hybrid Query.

CTE Clause

There are no restrictions to using Cypher with CTEs.

Query:

```
WITH graph_query as (
    SELECT *
```

```
        FROM cypher('graph_name', $$
        MATCH (n)
        RETURN n.name, n.age
    $$) as (name agtype, age agtype)
)
SELECT * FROM graph_query;
```

Results:

name	age
'Andres'	36
'Tobias'	25
'Peter'	35
3 row(s) returned	

Join Clause

A Cypher query can be part of a JOIN clause.

```
Developers Note
Cypher queries using the CREATE, SET, REMOVE clauses cannot be used in sql
queries with Joins, as they affect the Postgres transaction system. One
possible solution is to protect the query with CTEs. See the subsection
Using CTEs with CREATE, REMOVE, and SET for a solution to this problem.
```

Query:

```
SELECT id,
    graph_query.name = t.name as names_match,
    graph_query.age = t.age as ages_match
FROM schema_name.sql_person AS t
JOIN cypher('graph_name', $$
        MATCH (n:Person)
        RETURN n.name, n.age, id(n)
$$) as graph_query(name agtype, age agtype, id agtype)
ON t.person_id = graph_query.id
```

Results:

id	names_match	ages_match
1	True	True
2	False	True
3	True	False
3 row(s) returned		

Using CTEs with CREATE, REMOVE, and SET

Prior to the rest of the SQL query running, all CTEs are run and the results are cached and then referenced. This functionality makes it safe to run an Cypher query with a CREATE, SET, or REMOVE clause in the CTE and join to other tables or cypher queries.

```
WITH graph_query_1 as (
    SELECT *
    FROM cypher('graph_name', $$
        MATCH (n:Person), (m:Person)
        WHERE n.name = 'Tobias' AND m.name = 'Peter'
        CREATE (n)-[:FRIEND {id: n.name + '->' + m.name}]->(m)
        RETURN m.name
    $$) as (Tobias_new_friend agtype)
),
graph_query_2 as (
    SELECT *
    FROM cypher('graph_name', $$
        MATCH (n:Person)-[:FRIEND]->(m:Person)
        WHERE n.name = 'Tobias' AND m.name <> 'Peter'
        RETURN m.name
    $$) as (Tobias_old_friend agtype)
)
SELECT * FROM graph_query_1, graph_query_2;
```

Results:

Tobias_new_friend	Tobias_old_friend
'Peter'	‘Andres’
1 row(s) returned	

Using Cypher with expressions

Cypher cannot be used in an expression, the query must exists in the FROM clause of a query. However, if the cypher query is placed in a Subquery, it will behave as any SQL style query.

Using Cypher with '='

When writing a cypher query that is known to return 1 column and 1 row, the '=' comparison operator may be used.

```
SELECT t.name FROM schema_name.sql_person AS t
where t.name = (
    SELECT a
    FROM cypher('graph_name', $$
        MATCH (v)
        RETURN v.name
    $$) as (name varchar(50))
    ORDER BY name
    LIMIT 1);
```

Results:

name	age
'Andres'	36
3 row(s) returned	

Working with Postgres's IN Clause

When writing a cypher query that is known to return 1 column, but may have multiple rows. The IN operator may be used.

Query:

```
SELECT t.name, t.age FROM schema_name.sql_person as t
where t.name in (
    SELECT *
    FROM cypher('graph_name', $$
        MATCH (v:Person)
        RETURN v.name
    $$) as (a agtype));
```

Results:

name	age
'Andres'	36
'Tobias'	25
'Peter'	35
3 row(s) returned	

Working with Postgres EXISTS Clause

When writing a cypher query that may have more than 1 column and row returned. The EXISTS operator may be used.

Query:

```
SELECT t.name, t.age
FROM schema_name.sql_person as t
WHERE EXISTS (
    SELECT *
    FROM cypher('graph_name', $$
        MATCH (v:Person)
        RETURN v.name, v.age
    $$) as (name agtype, age agtype)
    WHERE name = t.name AND age = t.age
);
```

Results:

name	age
'Andres'	36
'Tobias'	25
3 row(s) returned	

Querying Multiple Graphs

There is no restriction to the number of graphs an SQL statement can query. Allowing users to query more than one graph at the same time.

```
SELECT graph_1.name, graph_1.age, graph_2.license_number
```

```
FROM cypher('graph_1', $$
    MATCH (v:Person)
    RETURN v.name, v.age
$$) as graph_1(col_1 agtype, col_2 agtype, col_3 agtype)
JOIN cypher('graph_2', $$
    MATCH (v:Doctor)
    RETURN v.name, v.license_number
$$) as graph_2(name agtype, license_number agtype)
ON graph_1.name = graph_2.name
```

Results:

name	age	license_number
'Andres'	36	1234567890
3 row(s) returned		