AgensGraph Developer Manual

Copyright Notice
Copyright © 2016, Bitnine Inc. All Rights Reserved.

Restricted Rights Legend
PostgreSQL is Copyright © 1996-2018 by the PostgreSQL Global Development Group.
Postgres95 is Copyright © 1994-5 by the Regents of the University of California.
AgensGraph is Copyright © 2016 by Bitnine Inc.

소스코드와 바이너리 형태의 재배포와 사용은 수정 여부와 관계없이 다음 조건을 충족할 때 가능하다.
소스코드를 재배포하기 위해서는 반드시 위의 저작권 표시, 지금 보이는 조건들과 다음과 같은 면책조항을 유지하여야만 한다.
바이너리 형태의 재배포는 배포판과 함께 제공되는 문서 또는 다른 형태로 위의 저작권 표시, 지금 보이는 조건들과 다음과 같은 면책조항을 명시해야 한다.
사전 서면 승인 없이는 저자의 이름이나 기여자들의 이름을 이 소프트웨어로부터 파생된 제품을 보증하거나 홍보할 목적으로 사용할 수 없다. 본 SW는 저작권자와 기여자들에 의해 "있는 그대로" 제공될 뿐이며, 상품가치나 특정한 목적에 부합하는 묵시적 보증을 포함하여(단, 이에 제한되지 않음), 어떠한 형태의 보증도 하지 않는다.
어떠한 경우에도 재단과 기여자들은 제품이나 서비스의 대체 조달, 또는 데이터, 이윤, 사용상의 손해, 업무의 중단 등을 포함하여(단, 이에 제한되지 않음), 본 소프트웨어를 사용함으로써 발생한 직접적이거나, 간접적 또는 우연적이거나, 특수하거나, 전형적이거나, 결과적인 피해에 대해, 계약에 의한 것이든, 엄격한 책임, 불법행위(또는 과실 및 기타 행위를 포함)에 의한 것이든, 이와 여타 책임 소재에 상관없이, 또한 그러한 손해의 가능이 예견되어 있었다 하더라도 전혀 책임을 지지 않는다.

Trademarks
AgensGraph®는 Bitnine Inc.의 등록 상표입니다. 기타 모든 제품들과 회사 이름은 각각 해당 소유주의 상표로서 참조용으로만 사용됩니다.

Open Source Software Notice
본 제품의 일부 파일 또는 모듈은 다음의 라이선스를 준수합니다. : OpenSSL, RSA Data Security, Inc., Apache Foundation, Jean-loup Gailly and Mark Adler, Paul Hsieh's hash

기술문서 정보

제목 : AgensGraph Developer Manual
발행일 : 2019년 2월 18일
소프트웨어 버전 : AgensGraph v2.1.0, based on PostgreSQL 10.4
기술문서 버전 : v1.0

차 례

1 Introduction 7
- 1.1 AgensGraph Highlights 7
- 1.2 Graph Database Concetps 8

2 Get Started 13
- 2.1 Install AgensGraph 13
- 2.2 Get started with Cypher 15

3 Cypher Query Language 17
- 3.1 Introduction 17
- 3.2 Data Type 20
- 3.3 Syntax 21
- 3.4 Clauses 24
- 3.5 functions 41

4 SQL Language 53
- 4.1 Introduction 53
- 4.2 Data Type 53
- 4.3 functions 102

5 Hybrid Query Language 257
- 5.1 Introduction 257
- 5.2 Syntax 257

6 Drivers 259
- 6.1 Introduction 259
- 6.2 Usage of the Java Driver 259
- 6.3 Usage of the Python Driver 263

7 Procedural language 265
- 7.1 Procedural language 265
- 7.2 PL/pgSQL 267
- 7.3 PL/Python 308

8 Appendix 326
- 8.1 AgensGraph Error Codes 326
- 8.2 Terminology 335
- 8.3 FAQ 337

기술문서에 대하여

기술문서의 대상

본 기술문서는 AgensGraph®(이하 AgensGraph)에서 제공하는 각종 애플리케이션 라이브러리를 이용하여 프로그램을 개발하려는 애플리케이션 프로그램 개발자를 대상으로 기술한다.

기술문서의 전제 조건

본 기술문서를 원할히 이해하기 위해서는 다음과 같은 사항을 미리 알고 있어야 한다.

- 그래프 데이터베이스의 이해
- 관계형 데이터베이스의 이해
- 기본 프로그래밍 관련 이해

기술문서의 제한 조건

본 안내서는 AgensGraph를 실무에 적용하거나 운용하는 데 필요한 모든 사항을 포함하고 있지 않으므로 설치, 환경설정 등 운용 및 관리에 대해서는 각 기술서를 참고한다.

기술문서 규약

<table>
<tr><th>표기</th><th>의미</th></tr>
<tr><td><AaBbCc123></td><td>프로그램 소스 코드의 파일명, 디렉터리</td></tr>
<tr><td>[Button]</td><td>GUI의 버튼 또는 메뉴 이름</td></tr>
<tr><td>진하게</td><td>강조</td></tr>
<tr><td>“ ”(따옴표)</td><td>다른 관련 안내서 또는 안내서 내의 다른 장 및 절 언급</td></tr>
<tr><td>‘입력항목’</td><td>화면 UI에서 입력 항목에 대한 설명</td></tr>
<tr><td>하이퍼링크</td><td>메일계정, 웹 사이트</td></tr>
<tr><td>></td><td>메뉴의 진행 순서</td></tr>
<tr><td>+—</td><td>하위 디렉터리 또는 파일 있음</td></tr>
<tr><td>|—</td><td>하위 디렉터리 또는 파일 없음</td></tr>
<tr><td>참고</td><td>참고 또는 주의사항</td></tr>
<tr><td>[Figure 1.1]</td><td>그림 이름</td></tr>
<tr><td>[Table 1.1]</td><td>표 이름</td></tr>
<tr><td>AaBbCc123</td><td>명령어, 명령어 수행 후 화면에 출력된 결과물, 예제코드</td></tr>
<tr><td>{}</td><td>필수 인수 값</td></tr>
<tr><td>[]</td><td>옵션 인수 값</td></tr>
<tr><td>|</td><td>선택 인수 값</td></tr>
</table>

연락처

Korea

Bitnine Inc.

A1201 GangSeo Hangang Xi Tower 401,

Yangcheon-ro, Gangseo-gu, Seoul,

South Korea

Tel : +82-70-4800-3517

Fax : +82-70-8677-2552

Email : agens@bitnine.net

Web : bitnine.net

USA

Bitnine Global Inc.

3945 Freedom Cir., Suite 260,

Santa Clara, CA 95054

U.S.A

Tel : +1 (408) 352-5165

Email : agens@bitnine.net

Web : bitnine.net

1 Introduction

1.1 AgensGraph Highlights

AgensGraph는 ACID 트랜잭션을 보장하는 그래프 데이터 모델을 기반으로 만들어진 데이터베이스이다.
오픈소스 데이터베이스인 PostgreSQL의 주요 기능들을 활용하여 구현되었다.
주요 특징은 다음과 같다.

- Multi-model 데이터베이스
  - Graph, Relational, Document 모델 지원.
  - Graph와 JSON 문서를 이용한 직관적이고 유연한 데이터 모델링.
- 고수준의 질의 기능
  - ANSI SQL 및 Cypher 질의 지원.
  - ACID 트랜잭션 지원.
  - 질의문 작성시 SQL과 Cypher 구문을 혼합한 hybrid 질의문 작성 가능.
  - 계층적 그래프 라벨 생성 가능.
- 고성능의 질의문 처리
  - 신속한 Graph Traversal 수행을 위한 Graph indexing 지원.
  - Vertex와 egde index 생성 가능.
  - JSON 문서 처리를 위한 full-text Search 지원.
- 제약조건
  - Unique, Mandatory, Check 제약조건 지원.
- High Availability
  - Active-standby 구조로 설정 가능.
- 시각화 도구
  - Graph 질의문의 결과 데이터를 시각화를 통해 보여줌.
- 고급 보안 기능
  - Kerberos와 LDAP을 이용한 인증시스템 구현.
  - SSL/TLS 규약을 통한 암호화.

- Connectivity
  - JDBC, Hadoop등 드라이버 제공.

1.2 Graph Database Concetps

이번 장에서는 그래프 데이터 모델에 관해 소개한다.

1.2.1 AgensGraph Database

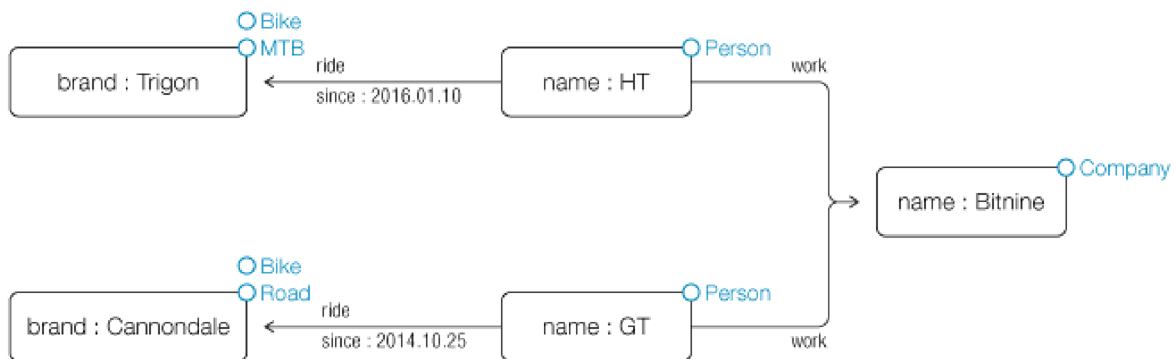
그래프 데이터베이스에서는 현실 모델의 객체들을 그래프 형태로 저장하고 관리한다.

각 객체 (Vertex) 간에 관계 (Edge) 를 맺고 있고, 유사한 객체끼리의 묶음을 그룹 (Label) 으로 표현할 수도 있다.

객체와 관계는 데이터 (Property) 를 가지고 있기 때문에 Property Graph Model이라고 부르기도 한다.

이같은 그래프 데이터 모델의 구성요소들에 관해 좀 더 자세히 알아보도록 하겠다.

아래의 예시는 그래프에 관한 구성요소를 보여준다.



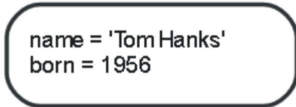
Vertices

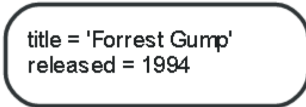
Vertices는 그래프 데이터 모델에서 가장 기본이 되는 요소로서 현실 세계의 개체 (entity) 를 나타내며, 속성 값인 Properties를 가진다.

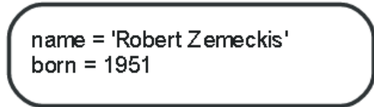
그래프에서는 vertices와 edges를 기본 단위로 가진다. AgensGraph에서는 vertices와 edges 모두 Properties를 포함할 수 있다. 보통 vertices로 개체를 나타내지만 edges로 개체를 나타내는 경우도 있다. Edges, Properties와는 달리 vertices는 label 값을 가지지 않거나, 여러 label 값을 가질 수 있다.

가장 단순한 형태의 그래프는 단일 vertex로 구성된다. Vertex는 0개 이상의 property를 가질 수 있다.

다음 단계로 여러 vertices를 갖는 그래프를 구성해본다. 이전 단계의 그래프에 2개 이상의 vertices를 추가하고 기존 vertex에 하나 이상의 property를 추가한다.







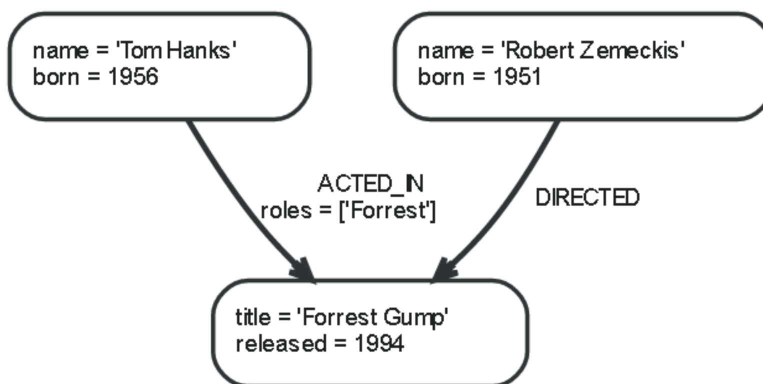
Edges

Edges는 vertices 연결하는 역할을 수행한다. 또한, 두 개의 vertex를 연결하는데 각 vertex는 edge의 방향성에 따라 start vertex와 end vertex로 나뉜다. Vertices와 마찬가지로 properties를 가진다.

Vertices 사이의 edges는 그래프 데이터베이스에서 중요한 역할을 하는데, 특히 연결된 데이터를 찾기 위해서 반드시 필요하다.

Edges는 두 개의 vertex를 연결하는데 각 vertex는 edge의 방향성에 따라 start vertex와 end vertex로 나뉜다.

Edges를 활용한다면 vertices를 list, tree, map, 복합 엔티티와 같은 다양한 자료구조 형태로 만들어줄 수 있다. 우리가 만들어가고 있는 예제에 edges를 추가한다면 더욱 의미있는 데이터를 나타낼 수 있다.



예제에서는 *ACTED_IN*과 *DIRECTED*를 edge 타입으로 사용했다. *ACTED_IN*의 property인 *roles*는 배열 타입의 값을 저장한다.

ACTED_IN edge는 *Tom Hanks* vertex를 start vertex로, *Forrest Gump* vertex를 end vertex로 가진다. 이를 다르게 설명하면 *Tom Hanks* vertex는 outgoing edge를 가지고 있고, *Forrest Gump* vertex는 incomming edge를 가지고 있다고 말할 수도 있다.

한 방향으로 edge가 형성되어 있다면 굳이 반대방향으로 중복해서 edge를 추가할 필요가 없으며, 이는 Graph 순회나 성능과도 관련이 있다.

Edges는 항상 방향성을 가지고 있지만, 사용하는 애플리케이션에서 불필요하다면 방향성을 무시할 수도 있다. 아래는 vertex가 자기 스스로에게 edges를 가지고 있는 모습을 나타내고 있다.

Graph 순회를 보다 효율적으로 수행하기 위해서는 모든 edges가 edge type을 가지는 것이 좋다.

Properties

Vertices와 edges 모두 properties를 가질 수 있다. Properties는 속성 값으로 각 속성명은 string 타입으로만 정의해야 한다.

Property 값으로 사용가능한 데이터 타입은 아래와 같다.

- Numeric 타입
- String 타입
- Boolean 타입
- List 타입 (다양한 데이터 타입들의 집합)

> *NULL*값은 property값으로 사용할 수 없다. *NULL*이 입력된다면 해당 property 자체는 없는 것으로 간주한다. 단, List에서는 *NULL* 값을 사용할 수 있다.

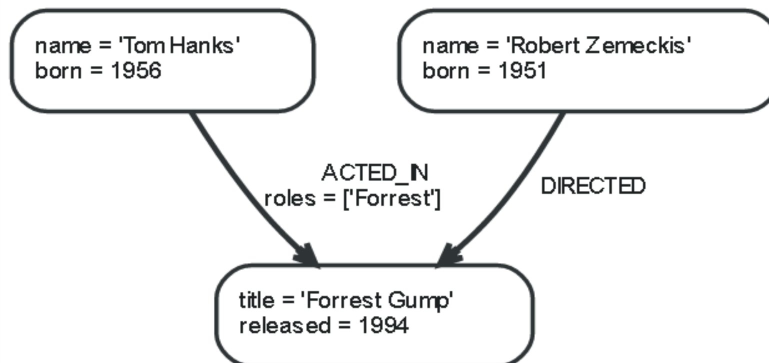
Type	Description	Value range
boolean		true/false
byte	8-bit integer	-128 to 127, inclusive
short	16-bit integer	-32768 to 32767, inclusive
int	32-bit integer	-2147483648 to 2147483647, inclusive
long	64-bit integer	-9223372036854775808 to 9223372036854775807, inclusive
float	variable-precision, inexact	15 decimal digits precision
char	16-bit unsigned integers representing Unicode characters	u0000 to uffff (0 to 65535)
String	sequence of Unicod characters	infinite

Labels

Label을 이용해서 vertex나 edge의 역할이나 타입을 정의할 수 있다. 특징이 유사한 vertex들이나 edge들을 그룹화하고 해당 그룹의 이름을 정의할 수 있는데 이를 label이라고 한다. 즉 유사한 label을 가진 모든 vertex들이나 edge들은 동일한 그룹에 속함을 나타낸다.

데이터베이스 질의문들은 label을 사용함으로서 전체 그래프가 아닌 해당 그룹에 관해서만 수행할 수 있고, 이는 질의들을 보다 효율적으로 수행할 수 있도록 도움을 준다.

Vertex들에 label을 사용하는 것은 선택사항이며, label을 가지지 않거나 오직 하나의 label만을 갖는다.
또한 label는 properties에 constraints를 정의하거나 index를 추가할 경우에도 사용된다.
Edge에도 vertex와 유사한 label을 지정할 수 있는데, vertex와는 달리 label이 없는 edge는 존재하지 않는다. 모든 edge들은 항상 하나의 label을 갖는다.
기존 예제 그래프에는 *Person*과 *Movie* labels를 추가해보도록 하겠다.



Label names

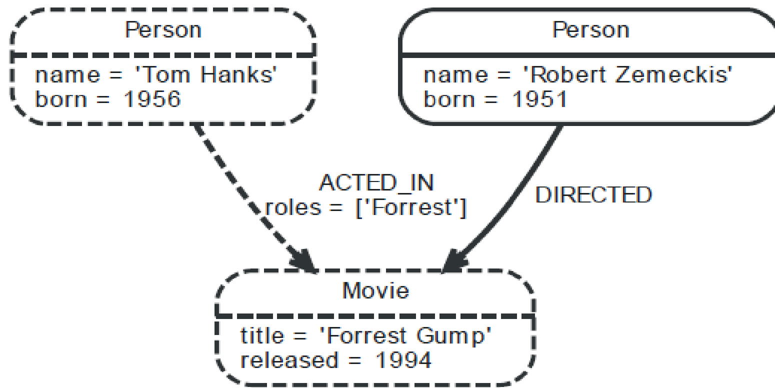
Label명은 글자와 숫자를 사용하여 표현할 수 있으며 모두 소문자로 변환 처리된다.

Labels는 int형의 unique한 id를 저장하며, 이는 데이터베이스가 최대 2^{16}-1(65535)개의 labels를 포함할 수 있음을 의미한다.

Traversal

요청받은 질의에 응답하기 위해 graph를 탐색하며 paths를 찾아가는 것을 traversal이라고 한다. Traversal은 시작 verticies에서 관련 vertices를 탐색하며 요청받은 질의에 대한 답변을 찾아가는 과정이다. 즉, 그래프를 순회한다는 해당 vertices와 파생되는 edges를 특정 규칙에 따라 찾아가는 것을 의미한다.

지금까지의 예제에서 Tom Hanks가 출현한 영화를 찾고자 한다. *Tom Hanks* vertex를 시작으로 그와 연결된 *ACTED_IN* edge를 따라서 *Forrest Gump*라는 vertex에서 종료되는 모든 과정을 traversal로 볼 수 있다.

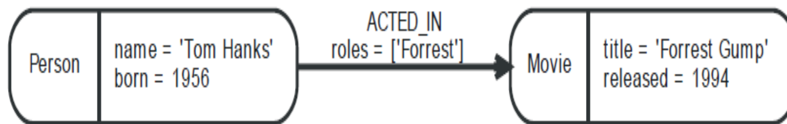


그래프 데이터베이스에서 cypher 질의문의 traversal과 추가적인 기술들을 활용한다면 보다 뛰어난 결과 데이터를 도출할 수 있다. 자세한 사항은 Cypher Query Language를 참고하면 된다.

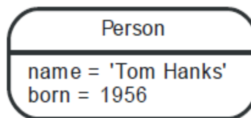
Paths

Path는 질의문이나 traversal의 결과 데이터로서, 한 개 이상의 vertices와 이에 연결된 edges를 나타낸다.

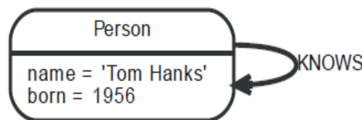
이전 예제에서 traversal의 결과 데이터인 path는 아래와 같다.



위 path의 길이는 1이다. Path중 가장 짧은 길이는 0인데, 단일 vertex가 edges를 가지고 있지 않은 경우가 이에 해당한다.



Vertex 자신을 가리키는 edge를 가진 경우 path의 길이는 1이다.



2 Get Started

2.1 Install AgensGraph

2.1.1 Pre-Installation on Linux

Get the pre-compiled binary

AgensGraph는 Linux/Windows에서 동작한다. AgensGraph는 두 가지 방법으로 설치가 가능하다. 하나는 바이너리 패키지를 다운로드 하는 방법이고, 또 다른 하나는 소스코드를 다운받아 패키지를 컴파일 하는 방법이다. 바이너리 패키지는 비트나인 홈페이지에서 다운이 가능하며, 소스코드는 github에서 다운이 가능하다. 자신의 시스템 환경을 알고 싶다면 다음 명령을 명령어 창에 입력한다.

```
uname -sm
```

Extract the package

다운받은 파일을 원하는 위치에서 압축을 해제한다. (e.g.: /usr/local/AgensGraph/)

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

2.1.2 Post-Installation Setup and Configuration

환경변수 설정 (선택사항)

쉘 시작파일에 다음 3줄을 추가한다. (e.g. .bash_profile)

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

데이터베이스 클러스터 생성

다음 명령을 통해 데이터베이스 클러스터를 생성한다. -D 옵션을 지정하지 않을 경우 .bash_profile에 지정한 AGDATA를 사용한다.

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

서버시작

아래의 명령어를 이용하여 AgensGraph를 시작할 수 있다.

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

데이터베이스 생성

```
createdb [dbname]
```

데이터베이스 이름과 사용자 이름을 명시하지 않으면 기본값으로 설정된다. 기본값은 현재 사용자의 이름이다.

터미널 실행

```
agens [dbname]
```

위와 같이 터미널 실행시 다음과 같은 화면을 볼 수 있다.

```
username=#
```

슈퍼유저인 경우 프롬프트에 ``=#''으로 표시되고 그외의 유저는 ``=>''으로 표시된다.

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

쿼리 입력 중간인 경우는 `-'으로 표시하고 괄호 내부에 입력하는 중이면'('로 표시한다. 다중 괄호인 경우도 그에 맞춰 표현할 수 있다. 입력 프롬프트를 커스터마이징 할 수 있으며 상세옵션은 다음 링크에서 확인 할 수 있다.

2.1.3 서버 매개변수 설정

AgensGraph는 성능 향상을 위한 서버 설정이 가능하다. 서버 변수는 데이터의 크기 그리고 서버의 자원 (메모리, CPU, 디스크 크기와 속도) 등에 따라 서버 매개 변수를 설정하는 것은 성능 향상을 위해 매우 중요하다. 다음의 서버 변수들은 AgensGraph의 그래프 질의 성능에 중요한 영향을 미친다. `$AGDATA/postgresql.conf` 파일을 변경함으로써 서버 매개 변수를 변경 가능하다. `$AGDATA/postgresql.conf` 파일을 수정하면 서버를 재시작 해야 한다.

- `shared_buffers` : 데이터 오브젝트 캐싱을 위한 메모리 크기이다. 이 변수는 제품의 환경에 마춰야 한다. 데이터의 크기만큼 클때 최적이다. 동시 세션 및 각 질의에 할당 된 메모리 크기를 고려하여 신중하게 설정해야 한다. 권장 값은 물리 메모리의 절반이다.
- `work_mem` : 물리 메모리와 실행되는 질의의 속성에 따라 크기가 증가한다.
- `random_page_cost` : 질의 최적화를 위한 매개변수이다. 그래프 질의를 위해 1 또는 0.005로 값을 줄여야 한다 (메모리에 그래프 데이터가 완전히 캐시된 경우)

2.2 Get started with Cypher

이번절은 Cypher의 소개와 아래의 내용에 대하여 소개한다.

- 그래프와 패턴에 대한 기본적인 이해
- 간단한 문제 해결
- Cypher 구문 작성법

2.2.1 Cypher 소개

AgensGraph는 그래프 데이터를 검색과 처리하는데 Cypher 질의 언어를 지원한다. Cypher는 SQL과 유사한 선언적인 언어이다.

2.2.2 패턴

AgensGraph의 그래프는 vertex와 edge로 구성되어 있다. Vertex와 edge에는 많은 속성들을 가질 수 있다. 실제 데이터는 단순한 그래프가 패턴으로 이루어져 있다. AgensGraph는 cypher를 통해 그래프의 패턴으로 검색과 처리를 수행한다.

그래프 생성

AgensGraph는 단일 데이터베이스에 다중 그래프 저장이 가능하다. Cypher는 다중 그래프를 파악할 수 없다. AgensGraph는 DDL과 Cypher를 사용하여 그래프를 생성, 관리를 위한 변수들을 지원한다. 다음 구문은 network라 불리는 그래프를 생성하고 현재 그래프를 설정하는 구문이다.

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

이 예제에서 `graph_path` 변수는 network로 설정한다. 그러나 그래프를 만들기 전에 `graph_path`가 설정되어 있지 않으면 그래프를 만든 후 자동으로 설정한다.

사용자 생성

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

새로운 사용자를 생성하여 데이터베이스 오브젝트에 대한 소유권과 기타 권한을 관리할 수 있다. 새 사용자가 그래프에서 label을 생성하고 싶다면 반드시 graph를 생성한 사용자와 같은 그룹에 포함되어 있어야한다. 사용자 생성에 관한 자세한 옵션은 링크를 참조하기 바란다.

레이블 생성

그래프 데이터를 생성하기 전에 레이블을 생성하는 것이 원칙이지만 cypher의 CREATE 문 수행시 label을 지정하면 자동으로 생성해주는 편의 기능을 지원하고 있다. AgensGraph에서는 vertex와 edge에는 하나의 레이블을 가져야 한다. 다음 구문은 person라는 vertex label과 knows라는 edge label을 생성하는 예제이다.

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

Vertex와 Edge 생성

이번절에서는 cypher의 `CREATE`절을 사용하여 *person* vertex와 *knows* edge를 생성한다. `CREATE`절은 vertex와 edge로 구성된 패턴을 생성한다. `(variable:label {property: value, ...})`는 vertex의 형태이고, `-[variable:label {property: value, ...}]-`는 edge의 형태이다. 그리고 edge의 방향성은 <와 >으로 edge의 방향을 나타낼 수 있다. vertex와 edge의 형태에서 `variable`의 경우는 있어도 되고, 없어도 된다.

> Note : AgensGraph는 edge 패턴에서 `--`을 지원하지 않는다. `--`는 문장 끝의 주석을 의미한다.

다음 구문은 ``Tom knows Summer'', ``Pat knows Nikki'' 그리고 ``Olive knows Todd'' 패턴을 생성하는 예제이다.

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

AgensGraph에서 vertex와 edge의 속성의 경우 *jsonb* 타입을 사용한다. 속성 값들의 경우 JSON 객체로 속성들을 표현한다.

3 Cypher Query Language

3.1 Introduction

3.1.1 Cypher is

Cypher는 그래프 데이터를 대상으로 질의를 수행하는 그래프 질의 언어이다. 주요 특징으로는 다음과 같다.

- **Declarative(선언형)**

  Cypher는 어떤 **방법**으로 해야 하는 지를 나타내기 보다 **무엇**과 같은지를 설명하는 선언형 언어이다. C, Java와 같이 실행될 알고리즘을 명시하는 명령형 언어와는 대조적으로 Cypher는 목표를 명시한다. 이런 처리 방식은 사용자가 질의를 함에 있어서 세부적인 구현에 대한 짐을 덜어준다.

- **Pattern Matching(패턴 일치)**

  Cypher는 찾고자 하는 그래프 데이터를 그리듯 표현하는 언어이다. 찾고자 하는 그래프 패턴을 ASCII Art 처럼 괄호와 대시 등을 사용하여 표현하며 해당 패턴과 일치하는 그래프 데이터를 찾는다. 찾고자 하는 형태를 직접 그리기 때문에 직관적으로 질의문을 작성할 수 있다.

- **Expressive(표현력 있는)**

  Cypher는 표현이 풍부한 질의를 위해 다양한 처리 방식들을 차용하였다. WHERE와 ORDER BY 같은 대부분의 키워드는 SQL에서, 패턴 매칭은 SPARQL에서, collection 개념은 Haskell, Python과 같은 언어에서 빌려왔다. 방식들을 차용해 익숙하면서도 간단하게 질의를 표현할 수 있다.

3.1.2 Elements of Cypher

Cypher의 기본 요소로서 vertex, edge, vlabel, elabel, property, variable이 있다.

- **Vertex**

  Vertex는 그래프를 구성하고 있는 가장 기본적인 요소이며 entity를 나타낸다. 대부분의 경우 vertex는 entity를 나타내는데 사용되지만, 목적에 따라 그 쓰임새는 달라질 수 있다.

- **Edge**

  Edge는 각 vertex 사이의 관계를 나타내며, edge 단독으로 존재할 수 없다.

- **Vlabel**

  Vlabel은 vertex들을 분류하는 기준이 되도록 사용자가 부여한 특정 이름이다.

- **Elabel**

  Elabel은 edge의 이름이다. vertex와 vertex 사이의 관계를 나타내는 역할을 한다.

- **Property**

  Property는 vertex 혹은 edge에 개별적, 차별적으로 부여할 수 있는 속성이다.

- **Variable**

  Variable은 vertex나 edge에 임의로 부여되는 식별자이다.

3.1.3 Handling Graph

Cypher를 자세히 설명하기에 앞서 사용 방법을 간단하게 살펴본다. 그래프를 생성하여, 그래프를 대상으로 질의를 하고, 그래프를 수정하는 방법에 대해 다루어 본다.

Creating Graph

Agens Graph는 단일 데이터베이스 내에 여러 그래프들을 저장할 수 있다. 따라서 그래프를 생성하고, 사용할 그래프를 선택해야 cypher를 이용한 그래프 질의가 가능하다.

- Create Graph

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

  CREATE GRAPH는 그래프를 생성하는 명령이다. 해당 명령과 함께 생성할 그래프의 이름을 함께 명시하여 사용한다(v1.3. 버전 이후부터 graphName에 공백 사용이 가능하다). graph_path는 현재 다룰 그래프를 의미하는 변수이다. 다루고자 하는 그래프의 이름을 SET을 사용하여 설정한다.

- Drop Graph

```
DROP GRAPH graphname CASCADE;
```

  DROP GRAPH는 그래프를 삭제하는 명령이다. 그래프 생성시 vertex와 edge에 대한 초기 label이 생성되므로 DROP GRAPH시 CASCADE를 반드시 사용해야 한다.

- Create Labels

```
CREATE VLABEL vlabelName;
CREATE ELABEL elabelName;

CREATE VLABEL childVlabelName inherits (parentVlabelName);
CREATE ELABEL childElabelName inherits (parentElabelName1, parentElabelName2);
```

`CREATE VLABEL`은 vlabel을, `CREATE ELABEL`은 elabel을 생성하는 명령이다. 각 명령은 생성할 vlabel 혹은 elabel의 이름을 함께 명시하여 사용한다.

`inherits()`는 다른 label을 상속하는 명령이다. Label을 생성할 때 자식 label 이름 뒤에 해당 해당 키워드와 함께 부모label의 이름을 명시하면 다른 label을 상속할 수 있다. 상속의 대상으로 부모 label은 여러 개가 될 수 있다.

- Drop Labels

```
DROP VLABEL vlabelName;
DROP ELABEL elabelName;

DROP ELABEL elabelName CASCADE;
```

`DROP VLABEL`은 vlabel을, `DROP ELABEL`은 elabel을 삭제하는 명령이다. 각 명령은 삭제할 vlabel 혹은 elabel의 이름을 함께 명시하여 사용한다. 상속관계에 있는 vlabel은 직접적으로 삭제가 되지않으므로 CASCADE를 사용하여 의존관계의 모든 데이터를 삭제한다.

- Create vertices and edges

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

`CREATE`절은 vertex와 edge를 생성하는 명령이다. 해당 명령은 생성할 vertex 혹은 edge를 pattern으로 올바르게 작성하여 사용한다. (`CREATE`절을 더불어 다양한 절과 vertex/edge를 표현할 pattern에 대해서는 이후에 자세히 설명한다.)

Querying Graph

그래프를 대상으로 질의를 한다는 것은 찾고자 하는 그래프를 pattern으로 표기를 하여 찾아낸 뒤 해당 그래프 내에서 원하는 정보를 추출해내는 것이다.

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

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

`MATCH` 절은 해당 절에 표기된 pattern에 부합하는 그래프 데이터를 찾아낸다. 찾은 그래프에서 반환하고자 하는 요소만을 `RETURN` 절에 명시한다.

Manipulating Graph

기존에 있던 그래프 데이터를 수정하기 위해서도 MATCH 절의 pattern을 표기하여 해당 그래프를 찾아낸 뒤에 수정 작업을 해야 한다.

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

SET 절을 이용하여 vertex나 edge의 property 값을 설정할 수 있다.

3.2 Data Type

Graph Data Type에 대한 설명이다. Graph Data Type에는 graphid, graphpath, vertex, edge가 있으며 type 상세는 아래와 같다.

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

3.2.1 graphid

vertex/edge의 label 생성 시 부여되는 고유 ID이다. graphid는 ag_label의 labid와 vertex 및 edge의 sequece 값으로 생성된다.

3.2.2 graphpath

연속된 vertex와 edge의 array 타입이다.

Column	Type	Description
vertices	vertex[]	
edges	edge[]	

3.2.3 vertex

vertex id와 속성값인 properties를 가진다.

Column	Type	Description
id	graphid	vertex id
properties	jsonb	vertex property

3.2.4 edge

vertex를 연결하는 역할을 하며, edge id, start vertex id, end vertex id, properties 를 가진다.

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

3.3 Syntax

3.3.1 Pattern

Pattern은 그래프를 표현하는 expression이다. Pattern은 하나 이상의 vertex 혹은 edge의 조합으로 나타내기 때문에 vertex와 edge를 pattern으로 어떻게 작성하는지는 매우 중요하다.

Vertex

Vertex는 괄호를 이용하여 표현하며 vlabel, property, variable과 함께 표기하면 찾고자 하는 vertex를 더욱 구체화 할 수 있다.

```
( )
```

Vertex는 ( )로 표현한다. 위 예제와 같이 괄호 안에 vlabel이나 property 등을 표기하지 않은 pattern은 모든 vertex를 의미한다.

```
(:person)
```

Vertex에 vlabel을 표현하고자 한다면 `(:vlabelName)` 으로 표기한다. Vertex를 나타내는 괄호 안에 콜론과 함께 vlabel의 이름을 표기하여 나타낸다. 위 예제는 person이라는 vlabel을 가지고 있는 vertex를 의미한다.

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

Vertex에 variable을 부여하고자 한다면 (**variableName**)로 표기한다. Variable은 alphanumeric(a~z, 0~9), underbar의 조합으로 명명할 수 있다(단 숫자로 시작할 수 없다). Vertex에 variable과 vlabel을 동시에 나타내고자 할 때는 (**variableName:vlabelName**) 으로 표기한다.

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

Vertex에 property를 표기하여 더욱 구체화 할 수 있다. Property를표현하고자 한다면 (**{propertyName:propertyValue}**)로 표기한다. 값이 string인 경우에는 `' 로 값을 감싸야 한다. Property를 여러 개 표현하고자 한다면 **,**를 이용한다.

Edge

Edge는 2개의 대시를 이용하여 표현하며 Edge는 elabel, property, variable과 함께 표기할 수 있다.

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

Edge는 -[]-로 표현한다. < >를 이용하면 방향성을 표현할 수 있다. 위 예에서 대시만으로 표현한 **-[ ]-**는 아무런 제약조건을 표시하지 않았기에 모든 edge를 의미한다. 화살괄호를 추가로 표현한 **-[ ]-> <-[ ]-**는 한쪽 방향성을 가진 모든 edge를 의미한다.

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

Edge에 다른 요소들을 표현하고자 한다면 [ ] 내부에 해당 요소들을 명시한다. Edge에 elabel을 표현하고자 한다면 -[**:elabelName**]-> 으로 표기한다. 위 예제는 knows라는 elabel을 가지고 있는 edge를 의미한다.

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

Edge에 variable을 부여하고자 한다면 -[**variableName**]->로 표기한다. Edge에 variable과 elabel을 동시에 나타내고자 할 때는 -[**variableName:elabelName**]-> 으로 표기한다.

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

Edge에 property를 표현하고자 한다면 `-[{propertyName:propertyValue}]->`로 표기한다. 값이 string인 경우에는 `` ` ' `` 를 이용한다. Property를 여러 개 표현하고자 한다면 `,`를 이용한다.

Vertices and Edges

Pattern은 vertex와 edge를 각각 표현 할 수도 있지만 함께 어우러져 표현할 수 있다.

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

Vertex와 edge의 조합으로 된 pattern의 기본틀은 `( )-[]->( )`이다. Vertex와 edge에 property, label을 명시하여 pattern에 의미를 부여하는 것이 좋다. Variable은 vertex와 edge 개별로 부여할 수도 있지만, pattern 전체를 대상으로 부여할 수도 있다. Pattern 전체에 variable을 부여하고자 한다면 `=`를 이용한다. 위 예제에서 **p**는 variable 이고, `=`를 이용함으로써 pattern 전체를 대상으로 적용하였다.

Path

Pattern 내에서 vertex와 edge의 개수는 지속적으로 증가할 수 있다. Pattern 내 일련 경로의 단위를 path라고 한다.

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


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


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

길이가 긴 path의 경우, 가독성을 높이고 작성의 편의성을 위해서 축약된 형태로 바꿔 작성할 수 있다. Vertex n 개가 edge를 n-1번 거쳐서 간다면 대괄호 내에 별표 (*) 와 함께 n-1을 표기하면 된다.(위의 예시를 참고한다.)

Flexible Length

상황에 따라 한 질의 내에서 거쳐야 하는 edge의 개수에 동적인 변화를 줘야 하는 경우가 생길 수 있다.

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

만약 path의 길이에 동적인 변화를 주고자 한다면 ..를 표기하면 된다. 위 예제는 차례대로 edge의 개수를 2개 이상 지닌 path, 7개 이하 지닌 path, 3개 이상 5개 이하 지닌 path, 무한 길이의 path를 뜻한다.

3.4 Clauses

3.4.1 Read Clauses

MATCH

MATCH절은 database에서 찾고자 하는 그래프 pattern을 기술하는 절이다. 데이터를 가져오는 가장 기본적인 방법이다. Pattern 명시에 대한 자세한 내용은 앞서 설명한 Pattern을 참고한다.

- Mathing

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

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

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

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

  찾고자 하는 그래프 pattern을 MATCH절에 기술한 뒤 RETURN절을 이용하여 반환한다.

- Various Notation

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

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

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

위 3개의 질의는 Jack의 지인이면서 Emily의 지인이기도 한 사람들의 이름을 출력한다. 모두 동일한 결과를 출력하지만 MATCH절의 개수나 MATCH절내 pattern 표기 방법이 다르다. 첫 번째 질의는 MATCH절을 두 번 사용하였고, 두 번째 질의는 하나의 MATCH절에 콤마를 이용하여 두 개의 pattern을 나타내었으며, 세 번째 질의는 하나의 MATCH절에 하나의 pattern을 이용하여 나타내었다. MATCH절을 다양하게 사용하여 같은 결과를 도출해낼 수 있다.

그래프 데이터를 생성하는 경우, 조회하는 경우, 삭제하는 경우, 추가하는 경우, 수정하는 경우 등 pattern과 부합하는 그래프를 우선적으로 찾아야 하는 경우가 대다수이다. 따라서 MATCH절은 매우 기본적이면서 중요한 절이다.

[참조]

Agens에서 대량의 데이터가 여러 페이지에 걸쳐서 출력될 경우 화면이동에 대한 도움말을 ? 키워드를 이용하여 볼 수 있다.

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

```
v                       Start up /usr/bin/vi at current line
ctrl-L                  Redraw screen
:n                      Go to kth next file [1]
:p                      Go to kth previous file [1]
:f                      Display current file name and line number
.                       Repeat previous command
---------------------------------------------------------------------------
```

Shortest Path

- Single Shortest Path

  Shortestpaths 함수를 사용하여 두 개의 vertex 사이에서 하나의 Shortest Paths 를 찾는다.

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

  두 개의 Vertex `Jack`과 `Alice` 사이에서 15개의 Releations 내 (안) 에서 하나의 Shortest Paths를 찾는다. 괄호 안에 start vertex, edge, end vertex로 구성된 단일 링크의 패스를 기술한다. Edge는 다른 edge와 마찬가지로 max hops 및 direction을 기술할 수 있다.

- All Shortest Paths

  두개의 vertex 사이에서 모든 Shortest Paths를 찾는다.

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

  두 개의 Vertex `Jack`과 `Alice` 사이에서 모든 Shortest Paths를 찾는다.

OPTIONAL MATCH

OPTIONAL MATCH절은 MATCH절과 마찬가지로 찾고자 하는 그래프 pattern을 기술하는 절이다. OPTIONAL MATCH절은 반환할 결과가 없을 경우 NULL을 반환한다는 점이 MATCH절과 다르다.

- Optional Matching

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

  OPTIONAL MATCH절의 사용방법은 MATCH절과 같다. 찾고자 하는 pattern을 OPTIONAL MATCH절에 표기한 뒤 RETURN절을 통해 결과를 반환한다. 반환되는 결과에는 NULL이 포함되어 있을 수 있다.

MATCH ONLY

Only 키워드를 사용하면 MATCH쿼리를 사용하여 자식 라벨을 제외한 결과를 반환할 수 있다.

- Match only

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

3.4.2 Projecting Clauses

RETURN

RETURN절은 cypher 질의의 결과를 명시하는 절이다. 찾고자 하는 그래프 pattern과 일치하는 데이터를 반환하는데 vertex, edge, property등을 반환할 수 있다.

- Vertex Return

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

  Vertex를 반환하고자 한다면 MATCH절에 찾고자 하는 vertex를 표기하고 variable을 부여한다. 이후 RETURN 절에서 해당 variable을 명시하면 vertex를 반환할 수 있다.

- Edge Return

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

  Edge를 반환하고자 한다면 MATCH절에 찾고자 하는 edge를 표기하고 variable을 부여한다. 이후 RETURN 절에서 해당 variable을 명시하면 edge를 반환할 수 있다.

- Property Return

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

  Vertex나 edge의 property를 반환하고자 한다면 MATCH절에 찾고자 하는 vertex나 edge를 표기하고 variable을 부여한다. 이후 RETURN절에서 해당 variable과 property를 .과 함께 명시하면 property를 반환할 수 있다.

- All Return

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

MATCH절에 표기한 요소들을 모두 반환하고자 한다면 RETURN절에 *를 명시하면 된다. 반환되는 요소들은 variable이 부여된 요소들이다. Variable이 부여되어 있지 않은 요소들은 반환되지 않는다.

- Path Return

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

MATCH절에 표기된 pattern과 일치하는 path를 반환하고자 할 때는 pattern 전체에 variable을 적용해야 한다. Pattern 앞에 variable을 =와 함께 표기하면 pattern 전체에 variable을 적용할 수 있다. 그 후 RETURN 절에 해당 variable을 명시하면 path를 반환할 수 있다.

- Alias Return

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

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

반환 되는 결과의 컬럼에 alias를 부여하여 출력할 수 있다. 반환 되는 요소 뒤에 AS 키워드와 함께 alias를 표기하면 된다. Alias를 큰따옴표와 함께 표기하지 않으면 소문자로 출력이 되고, 함께 표기하면 표기한 그대로 출력이 된다.

- Function Return

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

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

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

Vertex와 Edge에 대해서 id, properties만 얻을 수 있는 함수를 제공하며 path에 대해서 길이 및 각 요소들을 따로 구할 수 있도록 함수를 제공하고 있다. 또한 PostgreSQL이 지원하는 aggregation 함수를 사용할 수 있다.

- Constant Return

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

상수값들과 그에 대한 연산자를 표현 할 수 있다. SQL SELECT 구문에 올 수 있는 expression 중에 테이블, 컬럼 표현을 제외하고 모두 RETURN 구문과 함께 사용할 수 있다. 더 자세한 내용은 PostgreSQL Expression을 참조한다.

- Unique Return

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

반환 되는 결과의 중복된 레코드를 제거하고 유일한 값에 대해서만 결과를 출력한다. 반환 되는 요소 앞에 `DISTINCT` 키워드를 표기하면 된다.

WITH

WITH절은 여러 cypher 질의들을 연결하는 절이다. WITH절 선행 질의의 결과를 WITH절 후행 질의로 전달한다. 즉, WITH절은 후행 질의의 input으로 선행 질의의 값을 전달하는 RETURN절이라고 볼 수 있다.

- WITH

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

WHERE count(common) > 1
RETURN other;

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

위 3개의 질의를 예로 WITH절에 대해서 설명한다. 첫 번째 질의는 Jack과 other이 공통으로 알고있는 지인들이 몇명인지를 그 other과 함께 반환한다. 두 번째 질의는 첫 번째 질의와 연관지어 생각하면 공통의 지인 수가 1명보다 많은 경우에 해당되는 other를 반환하는 질의임을 알 수 있다. 두 번째 질의는 단독으로 실행될 수 없으며, 첫 번째 질의와 조합을 이루어야 두 번째 질의가 의미가 생기는데 두 질의를 이어주는 역할을 WITH절이 한다. 세 번째 질의는 첫 번째, 두 번째 질의를 연결시켜 주는 WITH절을 포함한 질의이다. 첫 번째 질의의 반환되는 결과들을 variable, alias 형태로 붙잡아두고 있다가 두 번째 질의로 그 결과들을 넘겨준다. 이러한 경우처럼 여러 질의들을 연결하고 선행 질의의 output이 후행 질의의 input으로 사용되는 때에는 WITH절로 연결시키면 된다. WITH절에서는 variable이나 alias형태로 붙잡고 있어야 한다.

- Partitioning

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

WHERE count(common) > 1
RETURN other;

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

WITH절을 사용하는 데에 있어서 전체 질의를 partitioning 하는 것이 중요하다. WITH절이 RETURN절이라 생각하고 RETURN 후에 올 수 있는 ORDER BY, LIMIT까지를 하나의 partition이라고 생각하면 된다. 따라서 선행 질의에 ORDER BY, LIMIT절 등이 있으면 해당 절들을 WITH절 뒤에 작성한 뒤 후행 질의를 작성하면 된다.

3.4.3 Reading sub-Clauses

WHERE

WHERE 절은 단독으로 사용할 수 없고, MATCH, OPTIONAL MATCH, START, WITH에 종속되어 사용한다. MATCH와 OPTIONAL MATCH절의 패턴에 제약조건을 추가하거나 START 및 WITH의 경우 결과를 필터링 한다.

Basic Usage

- Boolean operations

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

`AND`, `OR`과 같은 boolean operator를 사용할 수 있다.

- Filter on vertex label

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

`WHERE label(n) = 'foo'` 와 같은 형식으로 vertex를 label 별로 필터링 할 수 있다.

- Filter on vertex property

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

vertex property로 필터링 할 수 있다.

- Filter on edge property

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

edge property로 필터링 할 수 있다.

String matching

STARTS WITH, ENDS WITH를 이용하여 문자열에서 접두사 및 접미사와 일치하는 문자를 조회할 수 있다. 문자열의 위치와 상관없이 일치하는 문자열을 조회하기 위해서는 CONTAINS를 사용할 수 있다. 대/소문자를 구분하며, 문자열이 아닌 값을 조회하면 null이 반환된다.

- Prefix string search using `STARTS WITH`

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

STARTS WITH 연산은 문자열 시작 부분에 대/소문자 구분 일치를 수행하는데 사용된다.

- Suffix string search using `ENDS WITH`

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

ENDS WITH 연산은 문자열 끝에 대/소문자 구분 일치를 수행하는데 사용된다.

- Substring search using `CONTAINS`

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

CONTAINS 연산은 문자열 내의 위치에 관계없이 대/소문자 구분 일치를 수행하는데 사용된다.

ORDER BY

ORDER BY절은 결과 값을 정렬하는 절이다. RETURN절이나 WITH절과 함께 사용되며 컬럼을 기준으로 오름차순/내림차순으로 정렬을 할 것인지 결정한다.

- Order by Property

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

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

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

ORDER BY절은 기본적으로 vertex나 edge의 property를 기준으로 정렬한다. 정렬의 기준이 될 property의 alias를 ORDER BY절에 명시하면 된다. 즉 RETURN절에서 정렬의 기준이 될 property에게 alias를 부여한 뒤에 ORDER BY절에서 해당 alias를 명시한다. ORDER BY절에 여러 개의 기준들을 명시하면 첫 번째 기준으로 우선 정렬을 한 뒤 첫 번째 요소의 중복되는 값에 한해서만 두 번째 기준으로 정렬을 한다. 하지만 RETURN 절에 property가 명시되어 있지 않다면 ORDER BY절에 alias가 아닌 property를 직접적으로 명시해도 된다.

- Descending Order

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

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

ORDER BY절은 기본적으로 오름차순으로 정렬을 한다. 내림차순으로 정렬을 하고자 한다면 `DESC` 키워드를 표기하면 된다. ORDER BY절에 여러 개의 기준들을 명시하였을 경우에는 내림차순으로 정렬되기 원하는 요소 뒤에 `DESC`를 표기하면 된다.

SKIP

SKIP절은 검색 시작 위치를 변경하는 절이다.

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

MATCH절에 표기된 pattern을 찾을 때 SKIP절에 명시된 숫자만큼을 건너뛰고 검색을 시작한다.

LIMIT

LIMIT절은 result set의 결과 수를 제한 시키는 절이다.

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

출력하고자 하는 결과 수를 제한하고자 한다면 LIMIT절에 그 수를 명시하면 된다. Result set의 결과 수가 LIMIT 절에 명시된 숫자보다 작거나 같다면 모든 결과가 출력될 것이고, 크다면 LIMIT절에 명시된 숫자만큼만 출력하게 된다.

3.4.4 Writing Clauses

CREATE

CREATE절은 vertex나 edge 등 그래프 요소들을 생성하는 절이다.

- Create Vertex

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

Vertex를 생성하고자 할 때는 CREATE절에 vertex 관련 pattern을 표기하면 된다. Vertex 생성시 마지막 예제와 같이 앞서 명시한 vertex를 참조하여 생성할 수 있다.

- Create Edge

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

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

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

Edge는 두 vertex 사이를 연결하는 역할을 한다. 따라서 MATCH절을 통해 두 vertex를 찾은 뒤 edge를 생성해야 한다. 참고로 두 vertex라는 것은 서로 다른 vertex를 의미하는 것은 아니다. 위의 세 번째 질의처럼 self-edge도 가능하다.

- Create Path

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

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

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

각 요소들을 개별로 생성할 수도 있지만 CREATE절에 path를 표기하면 원하는 pattern을 한 번에 생성할 수도 있다. Path를 한번에 생성할 때는 주의해야 할 점이 있다. 첫 번째 질의처럼 MATCH절과 함께 사용하지 않은 경우에는 해당 pattern과 똑같은 데이터가 존재하더라도 새롭게 생성한다. 즉, 중복되어 만들어지는 것이다. 두 번째 질의처럼 MATCH절과 함께 사용한다면 Edward vertex를 찾은 뒤 해당 vertex에 likes edge와 Alice vertex를 새롭게 만든다. 만약 이미 존재하고 있는 Edward와 Alice vertex에 likes edge만을 새롭게 생성하고자 한다면 세 번째 질의처럼 사용하면 된다.

DELETE

DELETE절은 vertex나 edge를 제거하는 절이다.

- Delete vertex

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

MATCH절을 통해 제거하고자 하는 vertex를 찾고 DELETE절에 variable을 표기하여 제거한다. 단, 제거하려는 vertex가 다른 vertex와 edge로 연결되어 있다면 해당 edge를 먼저 제거하여야 vertex가 제거된다.

- Delete edge

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

MATCH절을 통해 제거하려는 edge를 찾고 DELETE절에 variable을 표기하여 해당 edge를 제거한다.

DETACH DELETE

- Delete a vertex with all its relationships

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

DETACH 키워드를 함께 이용하면 해당 vertex와 연결되어 있는 edge도 함께 제거된다. 제거하려는 vertex가 다른 vertex와 edge로 연결된 경우 해당 edge를 먼저 제거하는 과정을 생략할 수 있다.

SET

SET절은 property를 추가, 설정, 제거를 하거나 vlabel을 추가하는 절이다.

- Add Property

  ```
  MATCH (E:person {name: 'Edward'})
  SET E.age = 20;
  ```

  MATCH절을 통해 property를 추가하려는 vertex나 edge를 찾고 SET절에 추가할 property의 이름과 값을 표기한다. Property 관련 사항을 표기할 때는 작은 따옴표, 큰 따옴표와 함께 표기한다.

- Modify Property

  ```
  MATCH (E:person {name: 'Edward'})
  SET E.age = 30;
  ```

  MATCH절을 통해 property를 변경하려는 vertex나 edge를 찾고 SET절에 변경할 property의 이름과 값을 표기한다. Property 관련 사항을 표기할 때는 작은 따옴표, 큰 따옴표와 함께 표기한다.

- Remove Property

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

  MATCH절을 통해 property를 제거하려는 vertex나 edge를 찾고 SET절에 제거할 property의 이름과 NULL을 표기한다.

- Copy properties between nodes and relationships

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

  SET은 하나의 vertex/edge를 다른 vertex/edge로부터 모든 properties를 복사할 수 있다. at의 properties가 pn의 properties로 모두 복사되고 기존의 at의 properties는 모두 제거 된다.

- Replace all properties using a map and =

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

= 연산자를 사용하여 vertex/edge의 기존 properties를 map에의해 제공된 properties로 대체 할 수 있다.

- Remove all properties using an empty map and =

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

= 연산자를 사용하여 vertex/edge의 기존 properties를 모두 제거할 수 있다.

- Mutate specific properties using a map and +=

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

+= 연산자를 사용하여 vertex/edge의 기존 properties를 변경하거나, 새로운 properties를 추가할 수 있다. 아래와 같이 빈 map은 vertex 및 edge의 properties 가 제거 되지 않는다.

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

- Set multiple properties using one SET clause

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

여러 properties을 쉼표(,)로 구분하여 한번에 설정한다.

REMOVE

REMOVE절은 property를 제거하는 절이다.

- Remove property

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

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

SET절을 이용하여 property를 제거하는 방법은 이미 SET 에서 설명한 바 있다. REMOVE절을 이용해서도 property를 제거할 수 있다.

MATCH절을 통해 property를 제거하려는 요소를 찾고 REMOVE절에 제거할 property의 이름을 표기한다.

3.4.5 Reading/Writing Clauses

MERGE

MERGE절은 명시된 pattern이 graph 내에 존재하지 않으면 CREATE절처럼 해당 pattern을 추가하고, graph 내에 이미 존재한다면 단순히 MATCH절처럼 해당 pattern이 존재한다는 확인시켜주는 절이다. MERGE절이 인식하는 대상은 해당 절에 명시된 pattern 전체이다.

- Merge

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

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

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

MERGE절에 pattern을 명시하고 실행을 하면 해당 pattern이 graph 내에 존재하는지 아닌지를 알 수 있다. 이미 graph 내에 존재한다면 MATCH절처럼 기능을 수행하고, graph 내에 없다면 CREATE절처럼 해당 pattern을 새롭게 생성한다.

- Merge Path

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

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

MERGE절이 인식하는 대상은 해당 pattern 전체이다. 명시된 pattern의 일부만 graph 내에 존재한다고 해서 존재하지 않는 부분만 새롭게 생성하는 것이 아니다. 위의 질의를 예로 들어 설명한다. 첫 번째 질의는 Edward, Alice vertex가 이미 graph 내에 존재하고 likes edge로 연결되어 있지 않다면 edge만 새롭게 생성되는 것이 아니다. 새롭게 Edward, Alice vertex가 생성되고 그 두 vertex 사이에 edge가 생성된다.

Graph 속에 pattern의 일부 요소들이 이미 있는지 없는지 확실하게 알지 못한다면 두 번째 질의처럼 각 요소들을 나누어서 MERGE절을 사용하는 것이 좋다.

- ON CREATE SET and ON MATCH SET

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

ON CREATE SET, ON MATCH SET절을 활용하면 MERGE절의 동작 방식에 따른 property 설정이 가능하다. MERGE절이 CREATE절처럼 동작하였을 경우에는 ON CREATE SET절에 명시한 사항이 반영되고, MERGE절이 MATCH절처럼 동작하였을 경우에는 ON MATCH SET절에 명시한 사항이 반영된다.

3.4.6 Set operations

UNION and UNION ALL

UNION절은 여러 질의의 결과들을 하나로 결합하는 역할을 수행한다.

- UNION and UNION ALL

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

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

두 질의 사이에 UNION 키워드를 표기하면 두 결과를 하나로 결합할 수 있다. UNION은 중복되는 값들을 한 번씩만 출력한다. UNION과 ALL 키워드를 함께 사용하면 두 결과를 하나로 결합할 때 중복되는 값들을 중복하는 그대로 출력한다.

3.4.7 LOAD Clauses

LOAD FROM

LOAD FROM절을 사용하여 테이블 단위로 data를 loading 할 수 있다.

- create vertex

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

- create edge

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

3.4.8 Schema Clauses

CREATE and DROP CONSTRAINT

property에 제약사항을 설정하여 데이터를 제어하는 기능을 제공한다.

- CREATE and DROP CONSTRAINT

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

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

constraint name은 생략하면 자동으로 지정되어 생성한다. UNIQUE는 해당 레이블에서 필드의 값이 유일하도록 제한한다. check_expr은 새로 입력되거나 수정으로 변경되는 property에 대해 참 또는 거짓 값을 반환한다. 결과가 거짓이면 해당 입력/변경은 실패한다. `\dGv`, `\dG` 명령어를 이용하여 vertex 및 edge의 정보 조회시 constraint를 함께 확인할 수 있다.

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

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

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



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

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

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

3.5 functions

3.5.1 Aggregation functions

먼저 예제에 사용 될 데이터를 생성한다.

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

- avg()

  avg 함수는 numeric 값에 대한 평균을 반환한다.

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

- collect()

  collect 함수는 표현식에 의해 반환된 값을 포함하는 목록을 반환한다. 이 함수를 사용하면 여러 레코드 또는 값을 단일 목록으로 병합하여 데이터를 집계한다.

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

- count()

  count 함수는 결과 row의 수를 반환한다. count 함수는 vertex, edge의 수 혹은 그것들의 property의 수 등을 출력할 수 있으며 alias를 부여할 수도 있다.

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

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

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

- min()/max()

  min/max 함수는 numeric 속성을 입력으로 사용하며, 해당 열에서 최소/최대값을 반환한다.

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

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

- stDev()

  stDev 함수는 표준편차를 반환하는 함수이다. stDev 함수는 표본집단의 표준편차를 반환하며 property를 형변환해야 한다.

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

- stDevP()

  stDevP 함수는 표준편차를 반환하는 함수이다. stDevP 함수는 표본집단의 표준편차를 반환하며 property를 형변환해야 한다.

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

- sum()

  sum 함수는 numeric 값에 대한 합을 반환한다. numeric 값에 대한 합이기 때문에 property를 형변환해야 한다.

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

3.5.2 Predicates functions

- all( )

  all 함수는 리스트내 모든 요소가 조건을 만족하면, true를 반환한다.

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

- any( )

  any 함수는 리스트내 적어도 한 요소가 조건을 만족하면, true를 반환한다.

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

- none( )

  none 함수는 리스트내 모든 요소가 조건을 만족하지 않으면, true를 반환한다.

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

- single( )

  single 함수는 리스트내에서 한 요소만 조건을 만족하면, true를 반환한다.

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

3.5.3 Scalar functions

- coalesce()

  coalesce 함수는 목록에서 첫번째 null이 아닌 값을 반환한다.

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


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

- endNode()

  endNode 함수는 relationship의 마지막 노드를 반환한다.

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


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

- head()

  head 함수는 목록의 첫번째 요소를 반환한다.

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


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

- id()

  id 함수는 relationship 또는 노드의 id를 반환한다. 인자에 명시한 모든 노드에 대한 노드 id를 반환한다.

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

- last()

  last 함수는 목록의 마지막 요소를 반환한다.

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

- length()

  length 함수는 문자열이나 path의 길이를 반환한다. 인자로 문자열이나 문자열 타입의 property를 명시하면 해당 문자열의 글자수를 반환한다.

```
RETURN length('string');

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

- properties()

  properties 함수는 인자를 key/value를 매핑한 목록으로 변환한다. 인자가 이미 key/value가 매핑된 목록인 경우에는 변경되지 않고 반환된다.

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

- startNode()

  startNode 함수는 relationship의 시작 노드를 반환한다.

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

- toBoolean()

  toBoolean 함수는 문자열 값을 boolean으로 변환한다.

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

- type()

  type 함수는 인자로 전달된 edge의 elabel을 string으로 반환한다. type 함수에 인자를 전달할 시에는 주의 사항이 있다. MATCH 절을 통해 해당 pattern과 일치하는 edge를 찾고 variable을 부여한 뒤 해당 variable을 인자로서 전달해야 한다. edge 자체를 type 함수의 인자로 전달할 수는 없고 항상 variable로 전달하여야 한다.

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

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

- label() label 함수는 인자로 전달된 vertex 또는 edge의 label을 string으로 반환한다. label 함수도 labels/type 함수와 마찬가지로 vertex 및 edge에 variable을 부여한 뒤 해당 variable을 인자로서 전달해야한다.

```
MATCH (:person {name: 'Adam'})-[r]->(p:person {name: 'Eve'})
RETURN label(p), label(r);
```

3.5.4 List functions

- keys()

  key 함수는 node, relationship, map의 모든 속성 이름에 대한 문자열을 포함하는 목록을 반환한다.

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

- labels()

  labels 함수는 인자로 전달된 vertex의 부모 vertex까지 포함한 모든 vlabel의 목록을 반환한다. label 함수에 인자를 전달할 시에는 주의 사항이 있다. MATCH 절을 통해 해당 pattern과 일치하는 vertex를 찾고 variable을 부여한 뒤 해당 variable을 인자로서 전달해야 한다. vertex 자체를 label 함수의 인자로 전달할 수는 없고 항상 variable로 전달하여야 한다.

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

- nodes()

  nodes 함수는 인자로 전달된 path 내에 존재하는 vertex를 반환한다. nodes 함수에 인자를 전달할 시에는 주의 사항이 있다. MATCH 절을 통해 해당 pattern과 일치하는 path를 찾고 variable을 부여한 뒤 해당 variable을 인자로서 전달해야 한다. path 자체를 nodes 함수의 인자로 전달할 수는 없고 항상 variable로 전달하여야 한다. length 함수와 함께 사용하면 해당 path 내에 vertex의 개수를 알아낼 수 있다.

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

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

- relationships()

  relationships 함수는 인자로 전달된 path 내에 존재하는 edge를 반환한다. relationships 함수에 인자를 전달할 시에는 주의 사항이 있다. MATCH 절을 통해 해당 pattern과 일치하는 path를 찾고 variable을 부여한 뒤 해당 variable을 인자로서 전달해야 한다. path 자체를 relationships 함수의 인자로 전달할 수는 없고 항상 variable로 전달하여야 한다. count 함수와 함께 사용하면 해당 path 내에 edge의 개수를 알아낼 수 있다.

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

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

- tail()

  tail 함수는 목록에서 첫번째 요소를 제외한 모든 요소를 포함하는 목록 결과를 반환한다.

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

3.5.5 Mathematics functions

Number

- abs()

  abs 함수는 인자로 전달된 numeric 값을 절댓값으로 반환한다. 소수점이 있는 수나 뺄셈식을 인자로 전달해도 된다. MATCH 절을 통해 특정 요소를 찾고, 해당 요소들의 property 중에서 numeric 값인 property들의 뺄셈식을 인자로 전달하여도 된다.

```
RETURN abs(-3.14);

RETURN abs(20-45);

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

- ceil(), floor(), round()

  ceil 함수는 인자로 전달된 numeric 값을 소수점 첫째자리에서 올림하여 반환한다. floor 함수는 인자로 전달된 numeric 값을 소수점 첫째자리에서 내림하여 반환한다. round 함수는 인자로 전달된 numeric 값을 소수점 첫째자리에서 반올림하여 반환한다.

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

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

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

- rand()

  rand 함수는 0부터 1사이의 임의 부동 소수점 숫자를 반환한다.

```
RETURN rand();
```

- sign()

  sign 함수는 인자로 전달된 numeric 값의 부호를 반환한다. 전달된 인자가 양수이면 `1'을, 음수이면'-1'을, 0이면 '0'을 반환한다.

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

Logarithmic

- log()

  log 함수는 숫자의 자연 로그(로그의 밑이 e임)를 반환한다.

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

  27의 자연 로그가 반환된다.

- log10()

  log10 함수는 숫자의 상용 로그(로그의 밑이 10임)를 반환한다.

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

  27의 상용 로그가 반환된다.

- exp()

  exp 함수는 자연상수(e)를 밑으로 하고, 인자로 전달된 numeric 값을 지수로 하는 거듭제곱 값을 반환한다. 즉, exp(1)은 e^1 ≒ 2.718281828459O5을, exp(2)은 e^2 ≒ 7.38905609893065을, exp(-1)은 e^{-1} ≒ 0.3678794411714442을 반환한다.

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

- sqrt()

  sqrt 함수는 인자로 전달된 numeric 값의 제곱근을 반환한다. sqrt 함수는 음수를 인자로 전달할 수 없다.

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

Trigonometric

- sin()/cos()/tan()

  sin 함수는 인자로 전달된 numeric 값의 sine값을, cos 함수는 인자로 전달된 numeric 값의 cosine값을, tan 함수는 인자로 전달된 numeric 값의 tangent값을 반환한다. sin(),cos(),tan() 은 radians 단위로 값을 출력하며, degrees 단위로 값을 출력하고자 할때는 sind(), cosd(), tand() 를 사용한다.

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

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

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

- cot()/asin()/acos()/atan()/atan2()

  cot 함수는 인자로 전달된 numeric 값의 cotangent값(tangent의 역수)을, asin 함수는 인자로 전달된 numeric 값의 arcsine값(sine의 역)을, acos 함수는 인자로 전달된 numeric 값의 arccosine값(cosine의 역)을, atan, atan2 함수는 인자로 전달된 numeric 값의 arctangent값(tangen의 역)을 반환한다. asin, acos함수의 인자 범위는 -1 ~ 1 사이의 numeric 값이다. atan2는 인자가 두개인데 이는 atan 함수를 더욱 세밀하게 하기 위해서이다. 개념적으로 atan2(a, b) 는 atan(a/b) 와 같다. 하지만 atan(-5) 는 atan(-15/3) 과 atan(15/(-3)) 인지 확실히 알 수가 없다. 삼각함수에서 인자는 라디안 값이기 때문에 확실한 구분이 필요하다. 따라서 atan 보다는 atan2를 사용하는 것이 더욱 값을 정확하게 해준다.

```
RETURN cot(1.5);

RETURN asin(1);

RETURN acos(0.5);

RETURN atan(-1);

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

- pi()/degrees()/radians()

  pi 함수는 pi를 숫자로 반환한다. degrees 함수는 전달된 인자를 라디안값으로 인식하여 디그리(각도)로 변환하여 반환하며, radians 함수는 전달된 인자를 디그리(각도)로 인식하여 라디안으로 변환하여 반환한다.

```
RETURN pi();

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

RETURN radians(180);
```

3.5.6 String functions

- replace()

  replace 함수는 전달받은 첫 번째 인자에서 두 번째 인자가 존재할 경우 그 부분을 세 번째 인자로 바꾸는 함수이다.

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

- substring()

  substring 함수는 전달받은 첫 번째 인자를 두 번째 인자 번째부터 출력하는 함수이다. 세 번째 인자는 몇 글자를 출력할지를 나타낸다. 만약 세 번째 인자가 없는 경우와 첫번째 인자의 글자수보다 큰 경우에는 끝까지 출력이 된다.

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

- left()/right()

  left 함수는 첫 번째 인자를 왼쪽에서부터, right 함수는 오른쪽에서부터 두 번째 인자 글자수 만큼 출력하는 함수이다. 만약 남아있는 글자수보다 두 번째 인자의 값이 더 크면 남아있는 글자수만큼만 출력한다.

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

- lTrim()/rTrim()

  lTrim 함수는 전달받은 인자에서 왼쪽 공백을, rTrim 함수는 오른쪽 공백을 모두 없애고 출력하는 함수이다.

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

- toLower()/toUpper()

  toLower 함수는 전달받은 인자를 모두 소문자로, toUpper함수는 모두 대문자로 변환하여 출력하는 함수이다.

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

- reverse()

  reverse 함수는 전달받은 인자를 역순으로 출력하는 함수이다.

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

- toString()

  toString 함수는 정수, 부동 또는 boolean 값을 문자열로 변환한다.

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

- trim()

  trim 함수는 처음과 끝의 공백이 제거된 상태로 원래 문자열을 반환한다.

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

4 SQL Language

4.1 Introduction

AgensGraph는 관계형 데이터 질의를 위한 SQL을 지원한다. Table, Column, constraints, schema 등 object를 생성, 수정, 삭제하는 DDL (create, alter, drop 등)과 Data를 삽입, 수정, 삭제하는 DML(insert, update, delete 등)을 지원한다.

SQL Syntax는 PostgreSQL의 SQL Syntax와 동일하며 PostgreSQL-The SQL Language를 참고한다.

4.2 Data Type

AgensGraph에서는 다양한 데이터 타입을 제공한다. 또한 CREATE TYPE 명령을 사용하여 새로운 타입을 추가할 수 있다. CREATE TYPE에 대한 설명은 User-defined Type절을 참조한다.

아래의 표는 기본적으로 제공되는 범용 데이터 타입이며, 내부적으로 사용되거나 사용되지 않는 타입 중 일부는 나열되지 않은 것도 있다. (``Alias'' 항목은 내부적으로 사용되는 이름이다.)

Name	Alias	Description
bigint	int8	부호 있는 8바이트 정수
bigserial		자동 증분 8바이트 정수
bit [(n)]	16-bit integer	고정 길이 비트 스트링
bit varying [(n)]	varbit	가변 길이 비트 스트링
boolean	bool	논리 Boolean(true/false)
box		평면상의 사각 상자
bytea		바이너리 데이터 (``바이트 배열'')
character [(n)]	char [(n)]	고정 길이 문자 스트링
character varying [(n)]	varchar [(n)]	가변 길이 문자 스트링
cidr		IPv4 또는 IPv6 네트워크 주소
circle		평면상의 원
date		달력 날짜 (년, 월, 일)
double precision	float8	배정도 부동 소수점 수 (8바이트)
inet		IPv4 또는 IPv6 호스트 주소
integer	int, int4	부호 있는 4바이트 정수
interval [fields] [(p)]		시간 범위
json		텍스트 JSON 데이터

Name	Alias	Description
jsonb		바이너리 JSON 데이터, 분해됨
line		평면상의 무한 직선
lseg		평면상의 선분
macaddr		MAC(Media Access Control) 주소
money		통화량
numeric [(p, s)]	decimal [(p, s)]	선택 가능한 전체 자릿수의 정확한 숫자
path		평면상의 기하학적 경로
pg_lsn		AgensGraph 로그 시퀀스 번호
point		평면상의 기하학적 점
polygon		평면상의 기하학적 닫힌 경로
real	float4	단정도 부동 소수점 수(4바이트)
smallint	int2	부호 있는 2바이트 정수
smallserial	serial2	자동 증분 2바이트 정수
serial	serial4	자동 증분 4바이트 정수
text		가변 길이 문자 스트링
time [(p)] [without time zone]		시각(시간대 없음)
time [(p)] with time zone	timetz	시각, 시간대 포함
timestamp [(p)] [without time zone]		날짜 및 시간(시간대 없음)
timestamp [(p)] with time zone	timestamptz	날짜 및 시간, 시간대 포함
tsquery		텍스트 검색 쿼리
tsvector		텍스트 검색 도큐먼트
txid_snapshot		사용자 레벨 트랜잭션 ID 스냅샷
uuid		보편적 고유 식별자
xml		XML 데이터

4.2.1 Numeric Types

숫자 타입은 2/4/8byte 정수, 4/8byte 부동 소수점 수 및 선택 가능한 전체 자릿수로 구성된다.

다음은 Numeric Types에서 사용 가능한 타입이다.

Name	Storage Size	Description	Range
smallint	2 bytes	작은 범위의 정수	-32768 to +32767
integer	4 bytes	일반적인 정수	-2147483648 to +2147483647

Name	Storage Size	Description	Range
bigint	8 bytes	큰 범위의 정수	-9223372036854775808 to +9223372036854775807
decimal	variable	사용자 지정 전체 자릿수, 정확	소수점 앞 최대 131072자리, 소수점 이하 최대 16383자리
numeric	variable	사용자 지정 전체 자릿수, 정확	소수점 앞 최대 131072자리, 소수점 이하 최대 16383자리
real	4 bytes	가변 전체 자릿수, 부정확	6자리 전체 자릿수
double precision	8 bytes	가변 전체 자릿수, 부정확	15자리 전체 자릿수
smallserial	2 bytes	자동 증분 정수(small)	1 to 32767
serial	4 bytes	자동 증분 정수	1 to 2147483647
bigserial	8 bytes	자동 증분 정수(large)	1 to 9223372036854775807

Integer Types

smallint, integer 및 bigint type은 소수부가 없는 다양한 범위의 정수를 저장한다. 허용된 범위를 벗어난 값을 저장하려고 하면 에러가 발생한다.

- integer type : 범위, 저장소 크기 및 성능에서 최고의 균형점을 제공하므로 일반적으로 선택된다.
- smallint type : 일반적으로 디스크 공간이 부족한 경우에만 사용된다.
- bigint type : integer 타입의 범위가 불충분한 경우에 사용하기 위한 것이다.

SQL은 정수 타입 integer(또는 int), smallint 및 bigint만 지정한다. (int2, int4 및 int8로도 사용이 가능하다.)

Arbitrary Precision Numbers

numeric type은 자릿수의 매우 큰 숫자를 저장할 수 있고, 계산을 정확하게 수행한다. 특히 금액 및 정확성이 요구되는 기타 수량을 저장할 때 권장된다. 그러나, numeric 값의 산술은 다음 절에서 설명되는 정수 타입 또는 부동 소수점 타입에 비해 매우 느리다.
numeric의 scale은 소수점 오른쪽에 있는 소수부의 자릿수이다. numeric의 precision은 전체 숫자의 유효 자릿수의 총 개수이다. 즉, 소수점을 기준으로 양쪽 모두의 자릿수이다. 따라서 숫자 23.5141의 precision은 6이고, scale은 4이다. 정수는 Scale 0으로 간주할 수 있다.

numeric 컬럼의 최대 전체 자릿수 및 최대 소수 자릿수를 모두 구성할 수 있다.

타입 numeric의 컬럼을 선언하려면 다음 구문을 사용한다.

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

전체 자릿수는 양의 값이어야 하고, 소수 자릿수는 0 또는 양의 값이어야 한다.

```
NUMERIC(precision)
```

전체 자릿수 또는 소수 자릿수 없이 아래와 같이 지정하면, 이것은 소수 자릿수 0을 선택한다.

```
NUMERIC
```

Precision, scale값 명시 없이 precision, scale 값을 저장할 수 있는 numeric 컬럼을 생성하면, precision의 값까지 저장할 수 있다. 이런 종류의 컬럼은 특정 scale을 명시하지 않으면 허용 가능한 크기로 제약없이 사용가능하나 scale 값이 명시된 numeric 컬럼은 scale 값에 제약을 받게 된다. (데이터 이관시에는 전체 자리수와 소수 자리수를 항상 지정하도록 한다.)

> 참고 : 타입 선언에서 명시적으로 지정할 경우 최대 허용 전체 자릿수는 1000이다. 전체 자릿수를 지정하지 않은 NUMERIC은 표에 설명된 범위로 제한된다.

저장할 값의 소수 자릿수가 컬럼에서 선언된 소수 자릿수보다 큰 경우 시스템은 지정된 소수 자릿수로 값을 반올림한다. 그런 다음, 소수점 이하 자릿수가, 선언된 전체 자릿수에서 선언된 소수 자릿수를 뺀 것을 초과하면 에러가 발생한다. 숫자 값은 여분의 `0' 값 없이 물리적으로 저장된다. 따라서 선언된 컬럼의 전체 자릿수 및 소수 자릿수는 고정 할당이 아닌 최대이다. 이런 의미에서 numeric 타입은 char(n) 보다는 varchar(n)에 가깝다. 실제 저장소 요구 사항은 4자리 그룹별로 2바이트에, 8바이트 오버헤드에 대해 3을 더한 것이다.

타입 decimal 및 numeric은 동일하다. 두 가지 타입 모두 SQL 표준이다.

Floating-Point Types

데이터 타입 real 및 double precision은 부정확한 가변 전체 자릿수 숫자 타입이다. 부정확이란, 일부 값을 내부 형식으로 정확하게 변환할 수 없고 근사값으로 저장되므로 값의 저장 및 검색이 약간 불일치할 수 있다는 의미이다.

- 정확한 저장소 및 계산이 필요한 경우 (예 : 금액), numeric 타입을 대신 사용한다.
- 어떤 중요한 이유로 이 타입을 사용하여 복잡한 계산을 해야 할 경우, 특히 boundary cases(infinity, underflow)에서 특정 동작이 필요할 경우 구현을 신중하게 해야 한다.
- 두 개의 부동 소수점 값이 동등한지에 대한 비교가 항상 예상한 대로 작동하지 않을 수 있다.

대부분의 플랫폼에서 real 타입은 최소 범위가 1E-37 ~ 1E+37이고, 전체 자릿수는 최소 6자리이다. double precision 타입은 일반적으로 최소 15자릿수의 전체 자릿수로 1E-307 ~ 1E+308 정도의 범위이다. 너무 크거나 작은 값은 에러가 발생한다. 입력 숫자의 전체 자릿수가 너무 클 경우 반올림될 수도 있다. 0은 아닌 것으로 표시하기 어려울 정도로 숫자가 0에 근접하면 underflow 에러가 발생한다.

Serial Types

데이터 타입 smallserial, serial 및 bigserial은 실제 타입은 아니지만 고유 식별자 컬럼을 생성하는 국제적인 표기법이다(일부 다른 데이터베이스에서 지원되는 AUTO_INCREMENT 속성과 유사).

아래 두 구문은 동일하게 동작한다.

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

-- SEQUENCE
CREATE SEQUENCE tablename_colname_seq;

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

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

integer 컬럼을 생성하여 sequence에서 할당될 기본값으로 정렬한다. NOT NULL 제약 조건은 null 값이 삽입되지 않도록 하기 위해 적용된다. (대부분의 경우 UNIQUE 또는 PRIMARY KEY 제약 조건으로 중복 값이 우연히 입력되지 않게 할 수 있으나 자동으로 생성되지 않는다.)
마지막으로, sequence는 컬럼 ``소유자(owned by)'' 로 마킹되므로 컬럼 또는 테이블을 삭제하지 않으면 삭제되지 않는다.

sequence의 다음 값을 serial 컬럼에 삽입하기 위해 serial 컬럼 기본값을 지정한다. 이것은 INSERT문의 컬럼 목록에서 컬럼을 제외하거나 DEFAULT 키워드를 사용함으로써 가능하다.

- serial type은 serial4과 동일하다. 둘 다 integer 컬럼을 생성한다.
- bigserial type 및 serial8 type은 bigint 컬럼을 생성할 때 외에는 동일하게 작동된다. bigserial은 테이블 수명 내내 식별자를 231개 이상 사용할 것으로 예상하는 경우에 사용해야 한다.
- smallserial type 및 serial2 type은 smallint 컬럼을 생성할 때 외에는 동일하게 작동된다.

serial 컬럼에 대해 생성된 시퀀스는 소유 컬럼이 삭제되면 자동으로 삭제된다. 컬럼을 삭제하지 않고 시퀀스를 삭제할 수 있지만 그럴 경우 컬럼 기본 표현식이 강제로 삭제된다.

4.2.2 Character Types

다음은 Character Types에서 사용 가능한 타입이다.

Name	Description
character varying(n), varchar(n)	제한이 있는 가변
character(n), char(n)	고정 길이, 빈칸 채움
text	무제한의 가변 길이

2개의 기본 character type은 character varying(n)과 character(n)을 정의한다. n의 값은 양의 정수이며, 둘다 최대 n길이의 문자(byte가 아닌)를 저장할 수 있다.
n길이 보다 긴 문자열을 저장하면 초과하는 문자열이 공백이 아닐 경우에는 에러가 발생하며, 공백일 때는 지정한 n값의 길이로 잘린다. n길이 보다 짧은 문자열을 저장하면 character type의 경우 공백으로 채워지며, character varying의 경우 공백을 제외한 문자열을 저장한다.
character type의 후행 공백은 구문상 중요하지 않은 것으로 처리되며, character type의 값 2개를 비교할 때 무시된다. 후행 공백은 character varying, text 및 LIKE 같은 패턴 일치 정규 표현식을 사용하는 경우 구문상 중요하다.
char(n)와 varchar(n)는 character varying(n)과 character(n)의 별칭이다. 지정자가 없는 character 는 character(1)과 동일하며, 지정자가 없는 character varying은 크기에 상관없이 문자열을 저장한다. 또한 길이와 무관하게 문자열을 저장하기 위해 text type을 제공한다.
Character Type의 사용 예는 아래와 같다.

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

create table test2 (b varchar(5));
CREATE TABLE
insert into test2 values ('ok');
INSERT 0 1
insert into test2 values ('good ');
INSERT 0 1
```

```
insert into test2 values ('too long');
오류: character varying(5) 자료형에 너무 긴 자료를 담으려고 합니다.
insert into test2 values ('too long'::varchar(5));
INSERT 0 1
select b, char_length(b) from test2;
   b    | char_length
-------+-------------
 ok     |           2
 good   |           5
 too l  |           5
```

4.2.3 Date/Time Types

다음은 Date/Time Types에서 사용 가능한 타입이다.

Name	Storage Size	Description	Low Value	High Value	Resolution
timestamp [(p)] [without time zone]	8 bytes	날짜 및 시간 모두 (시간대 없음)	4713 BC	294276 AD	1 microsecond / 14 digits
timestamp [(p)] with time zone	8 bytes	날짜 및 시간 모두, 시간대 포함	4713 BC	294276 AD	1 microsecond / 14 digits
date	4 bytes	날짜(시각 없음)	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	8 bytes	시각(날짜 없음)	00:00:00	24:00:00	1 microsecond / 14 digits
time [(p)] with time zone	12 bytes	시각만, 시간대 포함	00:00:00 +1459	24:00:00-1459	1 microsecond / 14 digits
interval [fields] [(p)]	16 bytes	시간 간격	-178000000 years	178000000 years	1 microsecond / 14 digits

> 참고: SQL 표준에서는 timestamp와 timestamp without time zone을 동등하게 사용하며, timestamptz는 timestamp with time zone의 약어로 사용된다.

time, timestamp 및 interval은 두 번째 필드인 소수부 자릿수로 p값을 지정할 수 있으며, 기본적으로 전체 자리수

에 대한 명시적인 제약은 없다. p값의 허용 범위는 timestamp 및 interval type의 경우 0~6이다.

time type의 경우 8바이트 정수 저장소를 사용할 때는 p의 허용 범위가 0~6이고, 부동 소수점 저장소를 사용할 때는 0~10이다.

interval type은 아래 문구 중 하나를 작성함으로써 저장된 필드 집합을 제한하는 옵션이 추가로 있다.

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

Date/Time Input

날짜 및 시간 입력은 일, 월, 년의 순서를 아래와 같이 지정 할 수 있다.

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

월-일-년 해석을 선택하려면 DateStyle 파라미터를 MDY로 설정하고, 일-월-년 해석을 선택하려면 DMY로 설정하고, 년-월-일 해석을 선택하려면 YMD로 설정한다.

날짜 또는 시간 입력은 텍스트 문자열처럼 앞뒤에 작은따옴표를 사용해야 한다.

- 날짜

  Date types 에서 사용 가능한 타입이다.

Example	Description
1999-01-08	ISO 8601. 임의 모드에서 1월 8일 (권장 형식)
January 8, 1999	datestyle 입력 모드에서 애매함

Example	Description
1/8/1999	MDY 모드에서 1월 8일. DMY 모드에서 8월 1일
1/18/1999	MDY 모드에서 1월 18일. 다른 모드에서는 거부됨
01/02/03	MDY 모드에서 2003년 1월 2일
	DMY 모드에서 2003년 2월 1일
	YMD 모드에서 2001년 2월 3일
1999-Jan-08	임의 모드에서 1월 8일
Jan-08-1999	임의 모드에서 1월 8일
08-Jan-1999	임의 모드에서 1월 8일
99-Jan-08	YMD 모드에서 1월 8일. 그 외에는 에러
08-Jan-99	1월 8일, YMD 모드에서 except 에러
Jan-08-99	1월 8일, YMD 모드에서 except 에러
19990108	ISO 8601. 임의 모드에서 1999년 1월 8일
990108	ISO 8601. 임의 모드에서 1999년 1월 8일
1999.008	년 및 연중 일
J2451187	율리우스력 날짜
January 8, 99 BC	99년 BC

- 시각

  시각 타입은 time [(p)] without time zone 및 time [(p)] with time zone이다. time 단독으로는 time without time zone과 동일하다. 이 타입의 유효 입력은 시각으로 구성되며, 그 뒤에 선택적 시간대가 올 수 있다. (아래 표 참조.) 시간대가 time without time zone에 대한 입력으로 지정되면 무시된다. 날짜를 지정할 수도 있지만 America/New_York 같이 서머타임과 관련된 시간대 이름을 사용할 때 외에는 무시된다. 이 경우, 표준시 또는 서머타임 시가 적용되는지 여부를 결정하기 위해 날짜 지정이 필요하다. 적절한 시간대 오프셋은 time with time zone 값에 기록된다.

  표 - [시각 입력]

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	04:05와 동일. AM은 값에 영향을 주지 않음
04:05 PM	16:05와 동일. 입력 시는 <= 12여야 함
04:05:06.789-8	ISO 8601

Example	Description
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	약어로 지정된 시간대
2003-04-12 04:05:06 America/New_York	전체 이름으로 지정된 시간대

표 - [시간대 입력]

Example	Description
PST	약어 (태평양 표준시의 경우)
America/New_York	전체 시간대 이름
PST8PDT	POSIX 스타일 시간대 사양
-8:00	PST에 대한 ISO-8601 오프셋
-800	PST에 대한 ISO-8601 오프셋
-8	PST에 대한 ISO-8601 오프셋
zulu	UTC에 대한 군사 약어

시간대를 지정하는 방법은 아래 Time Zones 절 참조한다.

- 타임 스탬프

  타임 스탬프 타입에 대한 유효 입력은 날짜 및 시간의 연결로 구성되며, 그 뒤에 선택적 시간대, 그 뒤에 선택적 AD 또는 BC가 온다. (또한, AD/BC는 시간대 전에 나타나지만, 이것은 선호되는 순서가 아니다.)

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

  이것은 ISO 8601 표준을 준수하는 유효 값이다. 또한, 아래와 같은 명령 형식도 지원된다.

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

  SQL 표준은 ``+'' 또는 ``-'' 부호의 존재 및 해당 시간 이후에 시간대 오프셋에 의해 timestamp without time zone 및 timestamp with time zone 리터럴을 구분 짓는다.

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

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

timestamp with time zone를 지정하지 않으면 timestamp without time zone으로 처리하므로 timestamp with time zone를 사용할 경우 명시적 타입을 지정해야 한다.

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

- 특수 값
  표에 표시된 대로 편의를 위해 몇 가지 특수 날짜/시간 입력 값을 지원한다. 값 infinity 및 -infinity는 시스템 내부에서 특별히 표현되며, 변경 없이 표시되지만, 그 외에는 판독 시 기본 날짜/시간 값으로 변환되는 간단한 축약어이다. (특히, now 및 관련 문자열은 판독되는 순간에 특정한 시간 값으로 변환된다.) 이 값을 모두 SQL 명령에서 상수로 사용하는 경우 앞뒤로 작은따옴표를 사용해야 한다.

Input String	Valid Type	Description
epoch	date, timestamp	1970-01-01 00:00:00+00 (Unix 시스템 시간 0)
infinity	date, timestamp	다른 모든 타임 스탬프보다 이후
-infinity	date, timestamp	다른 모든 타임 스탬프보다 이전
now	date, time, timestamp	현재 트랜잭션의 시작 시간
today	date, timestamp	오늘 자정
tomorrow	date, timestamp	내일 자정
yesterday	date, timestamp	어제 자정
allballs	time	00:00:00.00 UTC

다음 SQL 호환 함수를 사용하면 해당 데이터 타입 CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP에 현재 시간값을 얻을 수도 있다.

Date/Time Output

날짜/시간 타입의 출력 형식은 4가지 스타일 ISO 8601, SQL(Ingres), 전형적인 POSTGRES(Unix 날짜 형식) 또는 German 중 하나로 설정할 수 있다. 기본값은 ISO 형식이다.
다음 표는 각 출력 스타일의 예제를 보여준다. date 및 time 타입의 출력은 제시된 예제에 따라 일반적으로 날짜 또는 시간 부분만 있다.

Style Specification	Description	Example
ISO	ISO 8601, SQL 표준	1997-12-17 07:37:16-08
SQL	전통적인 스타일	12/17/1997 07:37:16.00 PST

Style Specification	Description	Example
Postgres	원본 스타일	Wed Dec 17 07:37:16 1997 PST
German	지역 스타일	| 17.12.1997 07:37:16.00 PST

DMY 필드 순서가 지정된 경우 월, 일 순으로 출력되며, 그 외에는 일, 월 순으로 출력된다. 아래 표는 예제이다.

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

날짜/시간 스타일은 SET datestyle 명령, postgresql.conf 구성 파일의 DateStyle 파라미터 또는 서버나 클라이언트의 PGDATESTYLE 환경 변수를 사용하여 사용자가 선택할 수 있다. 형식 지정 함수 to_char는 좀 더 유연한 방식으로 날짜/시간 출력 형식을 지정할 수 있다.

Time Zones

Time Zone 및 표기법은 지형적인 것 외에도 정치적 결정의 영향을 받는다. 전세계의 시간대는 1900년대에 표준화되었지만 서머타임 규정 등에 의해 계속 바뀌고 있다. AgensGraph는 IANA(Olson) 시간대 데이터베이스를 이용한다. 미래의 시간의 경우, 지정된 시간대에 대해 알려진 최신 규칙은 먼 미래에도 무기한으로 계속 준수될 것으로 간주한다. 그러나 날짜, 시간 타입 및 기능의 특이한 혼합을 갖고 있다.
2가지 명백한 문제는 다음과 같다.

- date 타입은 연결된 시간대가 없지만 time 타입은 가능하다. 실세계에서 Time Zone은 서머타임 경계가 포함된 연도를 통해 오프셋이 달라질 수 있으므로 날짜 및 시간과 연결되지 않는 한 의미가 없다.
- 기본 Time Zone는 UTC로부터 상수 숫자 오프셋으로 지정된다. 따라서, DST 경계를 가로지르는 날짜/시간 계산 수행 시 서머타임에 적응하기 불가능할 수 있다.

이러한 문제를 해결하기 위해 Time Zone을 사용할 경우 우리는 날짜 및 시간이 모두 포함된 날짜/시간 타입의 사용을 권장한다(호환을 위해 지원은 하지만 time with time zone 타입 사용을 권장하지 않는다). 날짜 또는 시간만 포함하고 있는 타입에 대해 지역 시간대를 가정한다. Time Zone의 날짜 및 시간은 내부적으로 UTC로 저장된다. 이것은 클라이언트에게 표시되기 전에 TimeZone 구성 파라미터에 의해 지정되는 Time Zone에서 지역 시간으로 변환된다. 3가지 서로 다른 형식으로 Time Zone을 지정할 수 있도록 허용한다.

- 전체 시간대 이름. 예를 들면, America/New_York. 인식된 시간대 이름은 pg_timezone_names 뷰에 나열된다. (동일 시간대 이름은 기타 다수의 소프트웨어에서도 인식된다.)

- 시간대 약어. 예를 들면, PST. 서머타임 전환 날짜 규칙 집합을 암시할 수 있는 전체 시간대 이름과 대조적으로, 해당 사양은 단순히 UTC로부터 특정 오프셋을 정의한다. 인식된 약어는 pg_timezone_abbrevs 뷰에 나열된다. 구성 파라미터 TimeZone 또는 log_timezone을 시간대 약어로 설정할 수 없지만 날짜/시간 입력값에서 약어 및 AT TIME ZONE 연산자를 사용할 수 있다.
- 시간대 이름 및 약어 외에, STDoffset 또는 STDoffsetDST의 POSIX 스타일 시간대 사양을 수용한다. 여기서, STD는 지역 약어이고, offsetUTC로부터 서쪽의 시간에 대한 숫자 오프셋이고, DST는 지정된 오프셋에서 한 시간 앞선 것으로 간주되는 선택적 서머타임 지역 약어이다. 예를 들어, EST5EDT는 아직 인식된 지역 이름이 아닌 경우에 이것이 수용되고, 미국 동해안 시간과 기능적으로 동일해진다. 이 구문에서, 지역 약어는 글자의 문자열 또는 꺾쇠괄호 (<>) 를 사용한 임의의 문자열일 수 있다. 서머타임 지역 약어가 존재하는 경우 IANA 시간대 데이터베이스의 posixrules 항목에서 사용되는 것과 동일한 서머타임 전환 규칙에 따라 사용되는 것으로 간주된다. 표준 AgensGraph 설치에서, posixrules는 US/Eastern과 동일하므로 POSIX 스타일 시간대 사양은 USA 서머타임 규칙을 준수한다. 필요한 경우 posixrules 파일을 교체함으로써 이러한 행동을 조절할 수 있다.

요약하면, 이것은 약어 및 전체 이름 간의 차이이다. 약어는 UTC로부터 특정한 오프셋을 대표한다. 반면, 전체 이름 다수는 지역 서머타임 규칙을 암시하므로 2개의 가능한 UTC 오프셋을 갖는다. 예를 들면, 2014-06-04 12:00 America/New_York은 뉴욕 지역의 정오를 나타내는데, 이것은 특정 날짜의 동부 일광 절약 시간이었다 (UTC-4). 따라서 2014-06-04 12:00 EDT는 동일한 시간 인스턴스를 지정한다. 그러나 2014-06-04 12:00 EST는 해당 날짜에 서머타임이 명목상 발효되었는지와 무관하게 정오 동부 표준시 (UTC-5) 를 지정한다.

문제를 복잡하게 하기 위해 일부 지역은 서로 다른 시간에 서로 다른 UTC 오프셋을 의미하는 동일한 시간대 약어를 사용하고 있다. 예를 들면, 모스크바 MSK는 몇 년간은 UTC+3, 그 외에는 UTC+4를 의미했다. 이러한 약어를 지정된 날짜에 대한 의미에 따라 해석 (또는 가장 최근의 의미) 하지만, 위의 EST 예제에서 이것은 해당 날짜의 지역 시간과 반드시 일치하는 것은 아니다. POSIX 스타일 시간대 기능은 지역 약어의 온당함에 대해 검사하지 않으므로 가짜 입력을 조용히 수락하게 된다는 점을 조심해야 한다. 예를 들면, SET TIMEZONE TO FOOBAR0 는 시스템이 UTC에 대한 특유의 약어를 사용하게 하면서 작동된다. 유념해야 할 또 다른 문제는, POSIX 시간대 지역 이름에서 그리니치의 위치 west에 대해 양의 오프셋이 사용된다는 것이다. 그 외 지역에서는, AgensGraph 는 양의 시간대 오프셋이 그리니치의 east인 ISO-8601 표기법을 준수한다. 시간대 이름 및 약어는 대소문자를 구분하여 인식된다. 시간대 이름 또는 약어는 서버에 내장되지 않는다. 설치 디렉토리의 …/share/timezone/ 및 …/share/timezonesets/ 아래에 저장된 구성 파일로부터 구한다. TimeZone 구성 파라미터는 postgresql.conf 다른 표준 방법으로 설정 가능하다. 이것을 설정하는 특수한 방법이 몇 가지 있다.

- SQL 명령 SET TIME ZONE은 세션에 대한 시간대를 설정한다. SQL 사양에 좀 더 호환되는 구문인 SET TIME-ZONE TO를 대신 사용할 수 있다.
- PGTZ 환경 변수는 연결시 SET TIME ZONE 명령을 서버에 전송하기 위해 libpq 클라이언트에서 사용된다.

Interval Input

interval 값은 다음과 같은 자세한 구문을 사용하여 작성 가능하다.

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

여기서 quantity는 숫자(부호 사용)이고, unit는 microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium 또는 이러한 단위의 약어나 복수형이며, direction은 ago이거나 비어 있을 수 있다. at 부호(@)는 선택형 노이즈이다. 서로 다른 단위의 양은 적절한 부호 어카운팅을 사용하여 암시적으로 추가된다. 이 구문은 IntervalStyle이 postgres_verbose로 설정된 경우에 간격 출력에도 사용된다.

일, 시, 분 및 초의 양은 명시적 단위 마킹 없이 지정할 수 있다. 예를 들어, '1 12:59:10'은 '1 day 12 hours 59 min 10 sec'과 동일하게 판독된다.

또한, 년 및 월의 조합은 대시를 사용하여 지정할 수 있다. 예를 들면, '200-10'은 '200 years 10 months'와 동일하게 판독된다. (이러한 단축 형식은 사실 SQL 표준에서 허용되는 유일한 것이며, IntervalStyle이 sql_standard로 설정된 경우 출력 시 사용된다.)

지정자를 사용한 형식은 다음과 같다.

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

문자열은 P를 포함해야 하며, 시각 단위를 넣기 위해 T를 포함할 수 있다. 사용 가능한 단위 약어는 아래의 표에 나와 있다. 단위는 생략할 수 있으며, 임의의 순서로 지정할 수 있지만 1일 미만의 단위는 T 뒤에 나타나야 한다. 특히, M의 의미는 이것이 T 앞에 있는지, 아니면 뒤에 있는지에 따라 달라진다.

표 [ISO 8601 간격 단위 약어]

Abbreviation	Meaning
Y	년
M	월(날짜 부분에서)
W	주
D	일
H	시
M	분(시간 부분에서)
S	초

대체 형식 :

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

문자열은 P로 시작되어야 하며, T는 간격의 날짜와 시간을 분리한다. 값은 ISO 8601 날짜와 유사한 숫자로 지정된다. fields 사양으로 간격 상수를 작성하는 경우 또는 fields 사양으로 정의된 간격 컬럼에 문자열을 할당하는 경우

마킹되지 않은 수량의 해석은 fields에 따라 달라진다. 예를 들어 INTERVAL `1' YEAR는 1년으로 해석되는 반면, INTERVAL '1'은 1초를 의미한다. 또는 fields 사양에서 허용하는 최하위 오른쪽 필드 값은 무시된다.
예를 들면, INTERVAL `1 day 2:03:04' HOUR TO MINUTE는 결과적으로 초 필드를 삭제하지만 날짜 필드는 삭제하지 않는다. SQL 표준에 따라 간격 값의 모든 필드는 부호가 동일해야 하므로 선행 음의 부호는 모든 필드에 적용된다. 예를 들어, 간격 리터럴 `-1 2:03:04'에서 음의 부호는 날짜 및 시/분/초 부분에 모두 적용된다. 필드가 서로 다른 부호를 갖는 것을 허용하며, 전통적으로 텍스트 표현에서 각 필드의 부호를 독립적으로 처리하므로 시/분/초 부분은 이 예제에서 양의 값으로 간주된다. IntervalStyle이 sql_standard로 설정되면 선행 부호는 모든 필드에 적용되는 것으로 간주된다(단, 추가 부호가 없을 경우).

모호성을 피하려면 필드가 음인 경우 명시적으로 부호를 첨부하는 것이 좋다. 내부적으로 interval 값은 월, 일 및 초로 저장된다. 이것은 월의 날짜 수가 다르기 때문이고, 하루는 서머타임에 따라 23 또는 25시간일 수 있다. 월 및 일 필드는 정수이고, 초 필드는 소수로 저장할 수 있다. 간격은 보통 상수 문자열 또는 timestamp 감산에서 생성되므로 이 저장 방법은 대부분의 경우에 문제가 없다. 함수 justify_days 및 justify_hours는 일반 범위에서 오버플로되는 일 및 시를 조절할 때 유용하다. 자세한 입력 형식 및 좀 더 간략한 입력 필드의 일부 필드에서 필드 값은 소수부를 가질 수 있다. 예를 들면, '1.5 week' 또는 `01:02:03.45'. 이러한 입력은 저장을 위해 적절한 월, 일 및 초 수로 변환된다. 이것은 결과적으로 월 또는 일의 소수일 경우 소수가 변환 계수 1월 = 30일 및 1일 = 24시간을 사용하여 하위 필드에 추가된다. 예를 들면, '1.5 month'는 1개월 15일이다. 출력의 소수로서 초만 표시된다. 아래의 표는 유효 interval 입력의 몇 가지 예시를 보여준다.
표 [간격 입력]

Example	Description
1-2	SQL 표준 형식: 1년 2개월
3 4:05:06	SQL 표준 형식: 3일 4시간 5분 6초
1년 2개월 3일 4시간 5분 6초	전형적인 Postgres 형식: 1년 2개월 3일 4시간 5분 6초
P1Y2M3DT4H5M6S	ISO 8601 ``format with designators'': 위와 의미 동일
P0001-02-03T04:05:06	ISO 8601 ``alternative format'': 위와 의미 동일

Interval Output

간격 타입의 출력 형식은 명령 SET intervalstyle을 사용하여 4가지 스타일 sql_standard, postgres, postgres_verbose 또는 iso_8601 중에서 하나로 설정할 수 있다. 기본값은 postgres 형식이다. 표는 각 출력 스타일의 예제를 보여준다. sql_standard 스타일은 간격 값이 표준 제한을 만족하는 경우(양의 값 및 음의 값을 혼용하지 않는 경우 년-월만 또는 일-시만 해당)에 간격 리터럴 문자열에 대한 SQL 표준 사양을 준수하는 출력을 생성한다. 그 외에는, 부호 혼합 간격의 혼란을 없애기 위해 부호가 명시적으로 추가되고 표준 년-월 리터럴 문자열 뒤에 일-시 리터럴 문자열이 오는 것처럼 출력이 보인다.

표 [간격 출력 스타일 예제]

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

4.2.4 Boolean Type

표준 SQL 타입 boolean을 제공하며, ``true'', ``false'' 상태 및 SQL null 값으로 표현되는 세 번째 상태인 ``unknown'' 상태를 갖는다.

Name	Storage Size	Description
boolean	1 byte	True 또는 False 상태

- true 상태의 유효 리터럴 값은 다음과 같다.

```
TRUE
't'
'true'
'y'
'yes'
'on'
'1'
```

- false 상태의 경우 다음 값을 사용할 수 있다.

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

선행 또는 후행 공백은 무시되고 대소문자는 중요하지 않다. 키워드 TRUE 및 FALSE의 사용이 선호된다. 다음 예제는 boolean 값이 문자 t 및 f를 사용하여 출력되는 것을 보여준다.

boolean 타입 사용예는 다음과 같다.

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

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

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

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

4.2.5 Geometric Types

Geometric Types은 2차원 공간 개체를 표시하며 사용가능한 Geometric Types 은 아래와 같다.

Name	Storage Size	Description	Representation
point	16바이트	평면상의 점	(x,y)
line	32바이트	무한 직선	{A,B,C}
lseg	32바이트	무한 선분	((x1,y1),(x2,y2))
box	32바이트	사각 상자	((x1,y1),(x2,y2))
path	16+16n 바이트	닫힌 경로(다각형과 유사)	((x1,y1),⋯)
path	16+16n 바이트	열린 경로	[(x1,y1),⋯]
polygon	40+16n 바이트	다각형(닫힌 경로와 유사)	((x1,y1),⋯)
circle	24바이트	원	<(x,y),r> (중심점 및 반경)

Points(점)

Point는 Geometric Types에 대한 기본적인 2차원 빌딩 블록이다. point type의 값은 다음 구문 중 하나를 사용하여 지정된다.

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

여기서 x 및 y는 각각 부동 소수점 좌표이다. 점은 첫 번째 구문을 사용하여 출력된다.

Lines(선)

Line은 선형 방정식 Ax + By + C = 0으로 표현된다. 여기서 A 및 B는 둘 다 0이 아니다. line type의 값은 다음 양식의 입력 및 출력이다.

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

또는, 다음 형식 중 하나를 입력에 사용할 수 있다.

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

여기서 (x1,y1) 및 (x2,y2)는 직선상에 있는 두 개의 서로 다른 점이다.

Line Segments(선분)

Line Segment은 Line Segment의 끝점을 구성하는 점의 쌍으로 표현된다. lseg type의 값은 다음 구문 중 하나를 사용하여 지정된다.

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

여기서 (x1,y1) 및 (x2,y2)는 Line Segment의 끝점이다. Line Segment은 첫 번째 구문을 사용하여 출력된다.

Boxes(상자)

Box는 Box의 맞은편 모서리를 구성하는 점의 쌍으로 표현된다. box type의 값은 다음 구문 중 하나를 사용하여 지정된다.

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

여기서 (x1,y1) 및 (x2,y2)는 Box의 맞은편 모서리 2개이다. Box는 두 번째 구문을 사용하여 출력된다. 맞은편 모서리 두 개는 입력으로 제공되지만 값은 오른쪽 위 모서리와 왼쪽 아래 모서리로 저장하기 위해 필요 시 재정렬된다.

Paths(경로)

Path는 연결된 점의 목록으로 표현된다. 목록의 첫 번째 점과 마지막 점이 연결되지 않은 것으로 생각되면 열린 Path이고, 첫 번째 점과 마지막 점이 연결된 것으로 생각되면 닫힌 Path이다. path type의 값은 다음 구문 중 하나를 사용하여 지정된다.

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

여기서, 점들은 경로를 구성하는 선분의 끝점이다. 각괄호([])는 열린 path를 나타내고 소괄호(())는 닫힌 path를 나타낸다. 세 번째부터 다섯 번째 구문에서처럼 맨 바깥쪽 소괄호가 생략된 경우 닫힌 path로 간주된다. path는 두 번째 구문을 적절하게 사용하여 출력된다.

Polygons(다각형)

Polygon은 점 목록(Polygon 꼭지점)으로 표현된다. Polygon은 닫힌 경로와 매우 유사하지만 다르게 저장되며 자체적으로 지원되는 루틴 집합이 있다. polygon type의 값은 다음 구문 중 하나를 사용하여 지정된다.

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

여기서, 점들은 Polygon의 경계를 구성하는 선분의 끝점이다. Polygon은 첫 번째 구문을 사용하여 출력된다.

Circles(원)

Circle은 중심점과 반경으로 표현된다. circle type의 값은 다음 구문 중 하나를 사용하여 지정된다.

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

여기서 (x,y)는 Circle의 중심점이고 r은 반경이다. Circle은 첫 번째 구문을 사용하여 출력된다.

4.2.6 XML Type

XML Type은 XML데이터를 저장할 때 사용한다. 이 type의 장점은 텍스트 필드에 XML 데이터를 저장하는 것보다 잘 정리된 (well-formed) 형식으로 입력 값을 검사하며, 안전한 연산을 수행하도록 지원하는 함수가 있다. 이 데이터 타입을 사용하려면 configure --with-libxml로 빌드된 설치가 필요하다.

Creating XML Values

문자 데이터에서 xml 유형의 값을 생성하려면 아래의 함수를 사용한다.

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

사용예는 다음과 같다.

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

XML type은 입력값이 문서 형 선언(DTD)로 지정되더라도 DTD에 대한 입력값을 확인하지 않는다. XML스키마와 같은 다른 XML스키마 언어에 대한 유효성을 확인하는 기능도 지원하지 않는다.
xml에서 문자열 값을 생성하는 역 연산은 아래의 함수를 사용한다.

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

type은 character, character varying, text(또는 그중의 alias) 일 수 있다. SQL 표준에 따르면 XML과 문자 유형을 변환하는 유일한 방법이지만 값을 단순히 캐스팅 할 수도 있다.
문자열 값이 XMLPARSE 또는 XMLSERIALIZE를 통하지 않고 xml 유형으로 캐스팅 되거나 XMLSARIALIZE를 통과할 때 DOCUMENT와 CONTENT의 선택은 표준 명령을 사용하여 설정할 수 있는 ``XML 옵션'' 세션 구성 매개 변수에 의해 결정된다.

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

기본값은 CONTENT 이므로 XML 데이터의 모든 형식이 허용된다.

> 참고 : 기본 XML 옵션 설정을 사용하면 XML 내용 단편의 정의가 XML을 허용하지 않기 때문에 문서 유형 선언이 포함 된 문자열을 XML 형식으로 직접 변환 할 수 없다. XMLPARSE를 사용 하거나 XML 옵션을 변경해야 한다.

Encoding Handling

클라이언트에서 복수의 문자 인코딩을 처리할 경우와 이를 통해 XML 데이터가 전달될 경우 주의해야 한다. 쿼리를 서버에 전달하고 쿼리 결과를 클라이언트에 전달하기 위해 텍스트 모드를 사용하는 경우(정상 모드), 클라이언트와 서버 사이에 전달되는 모든 문자 데이터를 변환하고, 각각의 끝에서 문자 인코딩을 역으로도 변환한다. 이것은 위의 예제에 있는 것처럼 XML 값의 문자열 표현을 포함한다. 보통 이것은, 임베드된 인코딩 선언은 바뀌지 않으므로 클라이언트와 서버 사이의 전송 중에 문자 데이터가 다른 인코딩으로 변환되면 XML 데이터에 포함된 인코딩 선언이 무효화될 수 있다는 것을 의미한다. 이러한 행동에 대처하기 위해 XML 타입 입력에 대해 존재하는 문자 문자열에 포함된 인코딩 선언은 무시되고 CONTENT는 현재 서버 인코딩인 것으로 간주된다. 결과적으로, 올바른 처리를 위해 XML 데이터의 문자열은 현재 클라이언트 인코딩으로 클라이언트로부터 전송되어야 한다. 도큐먼트를 서버에 전송하기 전에 현재 클라이언트 인코딩으로 변환할 것인지, 아니면 클라이언트 인코딩을 적절히 조절할 것인지는 클라이언트의 책임이다. 출력에서 타입 XML의 값은 인코딩 선언이 없으며, 클라이언트는 모든 데이터가 현재 클라이언트 인코딩인 것으로 간주한다.

쿼리 파라미터를 서버에 전달하고 쿼리 결과를 클라이언트에 다시 전달하기 위해 바이너리 모드를 사용하는 경우 문자 집합 변환이 수행되지 않으므로 상황이 어렵게 된다. 이 경우, XML 데이터의 인코딩 선언이 준수되고, 부재 시 데이터가 UTF-8인 것으로 간주된다(XML 표준에서 요구하는 바이며 UTF-16을 지원하지 않는다는 점에 유의). 출력에서, 데이터는 클라이언트 인코딩이 UTF-8가 아닌 한, 클라이언트 인코딩을 지정하는 인코딩 선언을 갖는다. 그럴 경우 생략된다.

XML 데이터를 처리하는 것은 에러 발생 가능성이 적고, XML 데이터 인코딩, 클라이언트 인코딩 및 서버 인코딩이 동일한 경우에 좀 더 효율적이다. XML 데이터는 내부적으로 UTF-8로 처리되므로 서버도 UTF-8인 경우에 계산이 가장 효율적이다.

> 참고: 일부 XML 관련 함수는 서버 인코딩이 UTF-8가 아닐 경우 non-ASCII 데이터에서 작동하지 않을 수 있다. 이것은 특히 xpath()에서 문제로 알려져 있다.

Accessing XML Values

XML 데이터 타입은 비교 연산자를 제공하지 않는다는 점에서 특이하다. 이것은 잘 정의(well-defined)되지 않고 보편적으로 유용한 XML 데이터에 대한 비교 알고리즘이 없기 때문이다. 이것의 결론 중 하나는 xml 컬럼과 검색

값을 비교하여 행을 검색할 수 없다는 것이다. 따라서, ID 같은 별개의 키 필드로로 XML과 병용해야 한다. XML 값 비교를 위한 대체 솔루션은 먼저 문자열로 변환하는 것이지만, 문자열 비교는 유용한 XML 비교 방법과 아무 관계가 없다는 점에 유의해야 한다.

XML 데이터 타입에 대한 비교 연산자가 없으므로 이 타입의 컬럼에 인덱스를 직접 생성하는 것도 불가능하다. XML 데이터의 신속 검색이 필요한 경우에 가능한 해결책에는 표현식을 문자 스트링 타입으로 캐스트하고 인덱싱하거나 XPath 표현식을 인덱싱하는 것이 포함된다. 물론, 실제 쿼리는 인덱스된 표현식에 의해 검색이 조절되도록 해야 한다.

텍스트 검색 기능은 XML 데이터의 전체 문서 검색 속도를 올리는 데에도 사용할 수 있다. 그러나 필요한 전처리 지원은 아직 사용 가능한 것이 없다.

4.2.7 JSON Types

JSON 데이터 타입은 RFC 71591에 지정된 대로 JSON(JavaScript Object Notation) 데이터를 저장하기 위한 것이다. 해당 데이터는 text로도 저장할 수 있지만, JSON 데이터 타입은 각 저장된 값이 JSON 규칙에 따라 유효하도록 강제하는 장점이 있다. 이러한 데이터 타입으로 저장된 데이터에 사용할 수 있는 여러 가지 JSON 특정 함수 및 연산자도 있다.

JSON 데이터 타입은 JSON 및 JSONB의 두 가지가 있다. 이것은 거의 동일한 값 집합을 입력으로 수용한다. 실질적인 주요 차이는 효율성 중 하나이다. JSON 데이터 타입은 입력 텍스트의 정확한 사본을 저장하는데, 프로세싱 함수는 각 실행별로 다시 파싱해야 한다. JSONB 데이터는, 추가된 변환 오버헤드 때문에 입력 시에는 약간 느려지지만 재파싱이 불필요하므로 프로세싱 시에는 상당히 빨라지는 분해된 바이너리 형식으로 저장된다. JSONB 역시 인덱싱을 지원하는데, 이것은 상당한 장점일 수 있다.

JSON 타입은 입력 텍스트의 정확한 사본을 저장하므로 토큰 사이의 구문상 중요하지 않은 공백 및 JSON 개체 내의 키 순서를 보존한다. 또한, 값 내부의 JSON 개체가 동일한 키를 한 번 이상 포함하는 경우 모든 키/값 쌍이 유지된다. (프로세싱 함수는 마지막 값을 유효한 것으로 간주한다.) 반대로, JSONB는 공백을 보존하지 않으며, 개체 키의 순서를 보존하지 않으며, 중복 개체 키를 유지하지 않는다. 중복 키가 입력에 지정되면 마지막 값만 유지된다. 일반적으로 대부분의 어플리케이션은 개체 키의 순서 지정에 대한 기존의 가정과 같은 특별한 이유가 없을 경우에는 JSON 데이터를 JSONB로 저장해야 한다. 데이터베이스당 문자 집합 인코딩을 하나만 허용한다. 따라서 데이터베이스 인코딩이 UTF8가 아닐 때는 JSON 타입이 엄격하게 JSON 사양을 준수하는 것이 불가능하다. 데이터베이스 인코딩으로 나타낼 수 없는 문자를 직접 포함하려는 시도는 실패한다. 역으로, UTF8로는 안 되지만 데이터베이스 인코딩으로는 표현할 수 있는 문자는 허용된다. RFC 7159는 \uXXXX로 표시된 유니코드 이스케이프 시퀀스를 JSON 문자열이 포함하는 것을 허용한다. JSON 타입의 입력 함수에서, 유니코드 이스케이프는 데이터베이스 인코딩과 무관하게 허용되며, 구문상 올바른지에 대해서만 검사한다(즉, \u 뒤 4자리 16진수). 그러나 JSONB에 대한 입력 함수가 더 엄격하다. 데이터베이스 인코딩이 UTF8이 아닐 때는 이것은 비 ASCII 문자(U+007F 이상)에 대한 유니코드 이스케이프를 허용하지 않는다. JSONB 타입은 또한 \u0000를 거부하고(text 타입으로 표현할 수 없기 때문), 유니코드 BMT(Basic Multilingual Plane) 을 벗어난 문자를 지정하는 유니코드 대리 쌍의 사용이 정확해야 한다. 유효한 유니코드 이스케이프에 상응하는 ASCII 또는 UTF8 문자로 변환되어 저장된다. 이것은

겹치는 대리 쌍을 단일 문자로 포함한다.

> 참고: 다수의 JSON 프로세싱 함수는 유니코드 이스케이프를 일반 문자로 변환하고, 따라서 입력이 jsonb가 아닌 json 유형 인 경우에도 에러를 리턴한다. 일반적으로 JSON에서 유니코드 이스케이프를 UTF8 데이터베이스 인코딩과 가능하면 혼용하지 않는 것이 최선이다.

텍스트 JSON 입력을 JSONB로 변환하면, 표 8-23에 나타난 대로 RFC 7159에서 묘사된 기본 타입은 원시 AgensGraph 타입에 효과적으로 매핑된다. 따라서, JSON 타입에 적용되지 않고, 추상적으로 JSON에도 적용되지 않는 유효 JSONB 데이터가 구성되는 지엽적인 제약 조건 몇 가지가 추가되는데, 이것은 기본 데이터 타입으로 표현될 수 있는지에 대한 제한에 해당된다. 특히, JSONB는 AgensGraph numeric 데이터 타입의 범위를 벗어나고 JSON는 벗어나지 않는 수를 거부한다. 해당 구현이 정의된 제약 조건은 RFC 7159로 허용된다. 단, JSON의 number 기본 타입을 IEEE 754 double 부동 소수점 (RFC 7159에서 명시적으로 예측 및 허용) 으로 표현할 때 일반적이므로 실제로 이러한 문제는 다른 구현에서 훨씬 빈번하다. JSON을 해당 시스템과 상호 교환 형식으로 사용할 경우 AgensGraph에 저장된 원본 데이터와 비교했을 때 숫자 정밀도가 손실될 위험을 생각해봐야 한다. 역으로, 테이블에서 표시된 대로, 해당 AgensGraph 타입에 적용되지 않는 JSON 기본 타입의 입력 형식에 대한 지엽적인 제약 조건이 몇 가지 있다.
[JSON 기본 타입 및 해당 AgensGraph 타입]

JSON primitive type	Type	Notes
string	text	| 데이터베이스 인코딩이 UTF8이 아닌 경우 비 ASCII 유니코드 이스케이프이므로 \u0000는 허용되지 않음
number	numeric	NaN 및 infinity 값은 허용되지 않음
boolean	boolean	소문자 스펠링 true 및 false만 수용됨
null	(없음)	SQL NULL은 개념 상이

JSON Input and Output Syntax

JSON 데이터 타입의 입력/출력 구문은 RFC 7159에 지정된 대로 이다. 다음은 모두 유효한 JSON(또는 JSONB) 표현식이다.

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

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

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

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

앞에서 언급했듯이, JSON 값은 입력이고 추가 프로세싱 없이 인쇄되는 경우, JSON은 입력과 동일한 텍스트를 출력하고 JSONB는 공백처럼 구문상 중요하지 않은 내용은 보존하지 않는다. 예를 들면, 차이점은 다음과 같다.

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

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

구문상 중요하지 않지만 주의할 것 중 하나는 JSONB에 있는 것으로, 숫자는 기본 numeric 타입의 양식에 따라 인쇄된다. 실제로, 이것은 E 표시로 입력된 숫자는 인쇄될 때 누락된다는 것을 의미한다. 예를 들면 다음과 같다.

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

그러나, 이 예제에 나타나 있듯이, 구문상 중요하지 않지만 상동 검사 같은 목적으로 JSONB은 끝자리 0을 보존한다.

Designing JSON documents effectively

데이터를 JSON으로 표현하는 것은 요구 사항이 가변적인 환경에서 강제하게 되는 전통적인 관계형 데이터 모델보다 훨씬 유연할 수 있다. 두 방법이 동일 어플리케이션에서 공존 및 상호 보완할 수 있다. 단, 최대한의 유연성이 필요한 어플리케이션에 대해서도 JSON 도큐먼트가 약간은 고정된 구조를 갖는 것이 권장된다. 구조는 일반적으로 적용되지 않지만 (선언적으로 일부 비즈니스 규칙을 적용 할 수도 있음) 예측 가능한 구조를 사용 하면 테이블 의 ``문서'' (데이텀) 집합을 유용하게 요약하는 쿼리를 작성하는 것이 더 용이해진다. JSON 데이터는 테이블에 저장될 때 다른 데이터 유형과 동일한 동시성 제어 고려 사항의 영향을 받는다. 대형 도큐먼트를 저장하는 것이 실행 가능하더라도 업데이트는 전체 행에 대한 행수준 잠금을 획득한다는 점에 유의해야 한다. 업데이트 트랜잭션 간에 잠금 경합을 줄이기 위해 JSON 도큐먼트를 관리 가능한 크기로 제한을 고려하도록 한다. 원칙적으로 JSON 문서는 비즈니스 규칙이 지시하는 원자 데이터를 각각 독립적으로 수정할 수 있는 더 작은 데이텀으로 세분화 할 수 없다는 것을 나타낸다.

JSONB Containment and Existence

포함 조건 (containment) 을 테스트하는 것은 JSONB의 중요 기능이다. json 타입과 유사한 기능 집합은 없다. 포함 조건은 JSONB 도큐먼트 하나가 다른 도큐먼트 내에 포함되었는지를 테스트한다. 이 예제는 언급된 경우 외에는 true를 리턴한다.

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

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

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

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

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

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

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

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

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

일반적인 원칙은 포함 된 객체가 구조 및 데이터 내용에 대해 포함 된 객체와 일치한다. 일치하지 않는 배열 엘리먼트 또는 객체 키/값 쌍을 포함 객체에서 버린 후에 가능할 수 있다. 그러나 포함 조건을 비교 할 때는 배열 엘리먼트의 순서가 중요하지 않으며 중복된 배열 엘리먼트는 실제로 한 번만 고려된다.
구조가 일치해야한다는 일반적인 원칙에 대한 특별한 예외로서, 배열은 프리미티브 값을 포함 할 수 있다.

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

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

JSONB는 포함 테마에 대한 변형인 존재 (existence) 연산자가 있다. 이것은 문자열 (text 값으로 지정) 이 개체 키 또는 배열 엘리먼트로 JSONB 값의 최상위 레벨에 나타나는지를 테스트한다. 이 예제는 언급된 경우 외에는 true 를 리턴한다.

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

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

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

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

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

JSON 개체는, 배열과 달리 내부적으로 검색용으로 최적화되어 있고 선형 탐색하지 않으므로, 관련 키와 엘리먼트가 많은 경우 포함 또는 존재를 테스트하는 배열보다 훨씬 적합하다.

JSONB Indexing

GIN 인덱스는 다수의 JSONB 도큐먼트 (데이텀) 내의 키 또는 키/값 쌍을 효율적으로 검색하는 데 사용할 수 있다. 성능과 유연성 트레이드 오프가 서로 다른 2개의 GIN ``연산자 클래스''가 제공된다. JSONB에 대한 기본 GIN 연산자 클래스는 @>, ?, ?& 및 ?| 연산자를 사용한 쿼리를 지원한다. 이 연산자 클래스 내에서 인덱스를 생성하는 예제는 다음과 같다.

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

기본 GIN 연산자 클래스가 아닌 jsonb_path_ops는 @> 연산자만의 인덱싱을 지원한다. 이 연산자 클래스 내에서 인덱스를 생성하는 예제는 다음과 같다.

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

도큐먼트화된 스키마 정의를 사용하여 타사 웹 서비스에서 검색된 JSON 도큐먼트를 저장하는 테이블 예제를 생각해 보자. 일반적인 도큐먼트는 다음과 같다.

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

```
        "qui"
    ]
}
```

이 도큐먼트를 api라는 테이블에 jdoc라는 JSONB 컬럼으로 저장한다. 이 컬럼에서 GIN 인덱스가 생성되는 경우 다음과 같은 쿼리는 인덱스를 활용한다.

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

그러나, 연산자 ?를 인덱싱할 수 있더라도 인덱싱된 컬럼 jdoc에 직접 적용되지는 않으므로 다음과 같은 쿼리에서 인덱스는 사용할 수 없다.

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

그래도, 표현식 인덱스를 적절히 사용하면 위의 쿼리는 인덱스를 사용할 수 있다. ``tags'' 키 내부에서 특정 항목에 대한 쿼리가 공통된 경우 이것과 같은 인덱스 정의는 유용하다.

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

WHERE 절 jdoc -> `tags' ? 'qui' 는 인덱스 가능한 연산자 ?의 어플리케이션으로, 인덱스된 표현식 jdoc -> 'tags' 에 대해 인식된다.

쿼리에 대한 또 다른 방법은 포함을 최대한 잘 활용하는 것이다. 예를 들면 jdoc 컬럼의 간단한 GIN 인덱스는 이 쿼리를 지원할 수 있다.

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

그러나, 이러한 인덱스는 jdoc 컬럼의 모든 키와 값의 사본을 저장하는 반면, 이전 예제의 표현식 인덱스는 tags 키 아래에서 발견된 데이터만 저장한다. 간단 인덱스 접근법은 훨씬 더 유연하지만(임의의 키에 대한 쿼리를 지원하므로), 타깃이 되는 표현식 인덱스는 간단 인덱스보다 훨씬 작고 검색이 빠르다.

jsonb_path_ops 연산자 클래스는 @> 연산자를 사용한 쿼리만 지원하지만 기본 연산자 클래스 jsonb_ops 이상의 우수한 성능상 장점이 있다. 동일한 데이터를 놓고 봤을 때 jsonb_path_ops 인덱스는 일반적으로 jsonb_ops 인덱스보다 훨씬 작으며, 검색의 특정성은 훨씬 낫다. 데이터에 빈번하게 등장하는 키가 포함된 쿼리일 경우에 특히 그렇다. 따라서 검색 작업은 일반적으로 기본 연산자 클래스를 사용하는 것보다 훨씬 낫다.

jsonb_ops와 jsonb_path_ops GIN 인덱스 사이의 기술적 차이는, 전자는 데이터의 키와 값별로 독립적인 인덱스 항목을 생성한다는 것이고, 후자는 데이터의 값에 대해서만 인덱스 항목을 생성한다는 것이다. 기본적으로 각 jsonb_path_ops 인덱스 항목은 이것으로 이어지는 값과 키의 해시이다. 인덱스 {``foo'': {``bar'': ``baz''}}를 예로

들면, 단일 인덱스 항목은 foo, bar 및 baz의 3가지 모두 해시 값으로 통합하면서 생성된다. 따라서, 이러한 구조를 찾는 제약 조건 쿼리는 결과적으로 극단적인 특정 인덱스 검색이 된다. 그러나 foo가 키로 나타날 것인지 여부는 전혀 알아낼 방법이 없다. 반대로, jsonb_ops 인덱스는 foo, bar 및 baz를 각각 나타내는 3가지 인덱스 항목을 생성한다. 그런 다음, 제약 조건 쿼리를 실행하려면 이러한 항목을 세 가지 모두 포함하는 행을 조회한다. GIN 인덱스는 해당 AND 검색을 매우 효율적으로 수행할 수 있는 반면, 특히 3개의 인덱스 항목 중 하나를 포함하는 행이 수가 매우 많은 경우 동등한 jsonb_path_ops 검색보다는 덜 구체적이고 느리다. jsonb_path_ops 접근법의 단점은 {``a'': {}} 같은 값을 포함하지 않는 JSON 구조에 대해 인덱스 항목을 생성하지 않는다는 것이다. 해당 구조가 포함된 도큐먼트 검색이 요청되는 경우 전체 인덱스 스캔이 필요한데, 매우 느리다. 따라서 jsonb_path_ops는 해당 검색을 빈번하게 수행하는 어플리케이션에 부적합하다. JSONB도 btree 및 hash 인덱스를 지원한다. 이것은 전체 JSON 도큐먼트의 동등성을 검사하는 경우에만 일반적으로 유용하다. JSONB 데이터의 btree 순서 지정은 그다지 중요하지 않지만, 완성도를 위해서 필요하다.

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

```
Object with n pairs > object with n - 1 pairs
```

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

쌍의 수가 동일한 개체는 다음과 같은 순서로 비교된다.

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

개체 키는 저장 순서로 비교되며, 특히 짧은 키는 긴 키 앞에 저장되므로 이는 다음과 같이 직관적이지 않은 결과로 이어진다.

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

비슷하게, 엘리먼트의 수가 동일한 배열은 다음과 같은 순서로 비교된다.

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

원시 JSON 값은 기본 데이터 타입에 대한 것과 동일한 비교 규칙을 사용하여 비교된다. 문자열은 기본 데이터베이스 콜레이션을 사용하여 비교된다.

4.2.8 Arrays

테이블의 컬럼 타입이 가변 길이 다차원 배열로 정의되는 것을 허용한다. 기본 제공 또는 사용자가 정의한 기본 타입, enum 타입 또는 컴포지트 타입의 배열을 생성할 수 있다. 도메인의 배열은 아직 지원되지 않는다.

Declaration of Array Types

배열 타입의 사용을 설명하기 위해 아래와 같은 테이블을 생성해보자.

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

표시된 대로 배열 데이터 타입은 대괄호 ([]) 가 배열 엘리먼트의 데이터 타입 이름에 추가되어 명명된다. 위의 명령은 타입 text (name) 의 컬럼을 갖는 sal_emp라는 테이블을 생성하는데, 타입 integer (pay_by_quarter) 의 1 차원 배열은 직원의 분기별 급료를 나타내고, text (schedule) 의 2차원 배열은 직원의 주간 스케줄을 나타낸다. CREATE TABLE에 대한 구문은 지정할 배열의 정확한 크기를 허용한다. 예를 들면 아래와 같다.

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

다만, 현재의 구현은 제공된 배열 크기 제한을 무시한다. 즉, 동작은 지정되지 않은 길이의 배열과 동일하다. 현재 구현에서는 선언 된 차원 수를 강제하지 않는다. 특정 엘리먼트 유형의 배열은 크기 또는 차원 수에 관계없이 모두 동일한 유형으로 간주된다. 따라서 CREATE TABLE 에서 배열 크기 또는 차원 수를 선언하는 것은 단순히 문서화한 것이며 런타임 동작에 영향을 미치지 않는다.

키워드 ARRAY를 사용하여 SQL표준을 준수하는 대체 구문을 1차원 배열에 사용할 수 있다. pay_by_quarter는 다음과 같이 정의되었을 수 있다.

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

배열 크기가 지정되지 않은 경우는 다음과 같다.

```
pay_by_quarter  integer ARRAY,
```

단, 어떤 경우든 크기 제한을 강제하지 않는다.

Array Value Input

배열 값을 리터럴 상수로 작성하려면 중괄호 내에 값을 넣고 쉼표로 구분한다. 값 앞뒤로 큰따옴표를 사용할 수 있으며, 쉼표 또는 중괄호가 포함된 경우에 이렇게 해야 한다. 배열 상수의 일반적인 형식은 다음과 같다.

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

여기서 delim는 pg_type 항목에 기록된 타입에 대한 구분 문자이다. AgensGraph 배포에서 제공되는 표준 데이터 타입 중에 세미콜론 (;) 을 사용하는 타입 box 외에는 모두 쉼표를 사용한다. 각 val은 배열 엘리먼트 타입 또는 서브 배열의 상수이다. 배열 상수 예제는 다음과 같다.

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

이 상수는 3개의 정수 서브 배열로 구성된 2차원 3x3 배열이다. 배열 상수의 엘리먼트를 NULL로 설정하려면 엘리먼트 값에 대해 NULL을 작성한다. (NULL의 대문자 또는 소문자 변동이 가능하다.) 실제 문자열 값 ``NULL"을 원할 경우 앞뒤에 큰따옴표를 사용한다. (상수는 처음에 문자열으로 처리되고 배열 입력 변환 루틴으로 전달된다. 명시적 타입 사양이 필요할 수도 있다.)
INSERT문 예제 및 수행 결과는 다음과 같다.

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

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

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

다차원 배열에는 각 차원에 대해 일치하는 범위가 있어야하며, 불일치가 발생하면 다음과 같은 오류가 발생한다.

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

ARRAY의 생성자 구문도 사용할 수 있다.

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

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

배열 엘리먼트는 일반적인 SQL 상수 또는 표현식을으로 문자열 리터럴은 배열 리터럴에서와 같이 큰 따옴표 대신 작은 따옴표로 묶는다.

Accessing Arrays

위에서 생성된 테이블에서 몇가지 쿼리를 실행할 수 있다. 먼저 배열의 단일 엘리먼트에 액세스 하는 방법이다. 아래의 쿼리는 2분기에 급료가 바뀐 직원의이름을 검색한다.

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

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

배열 번호는 대괄호 안에 작성된다. 1부터 시작되는 배열 숫자 표기를 사용한다. 즉, n 엘리먼트의 배열은 array[1]부터 시작하고 array[n]에서 끝난다.
이 쿼리는 모든 직원의 3분기 급료를 검색한다.

```
SELECT pay_by_quarter[3] FROM sal_emp;

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

배열 또는 서브 배열의 임의의 사각형 슬라이스에 액세스할 수도 있다. 배열 슬라이스는 하나 이상의 배열 차원에 대해 lower-bound:upper-bound를 사용해서 나타낸다.

예를 들어, 이 쿼리는 주 중 첫 2일의 Bill의 스케줄에서 첫 번째 항목을 검색한다.

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

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

차원을 슬라이스로 작성하는 경우 (예 : 콜론 포함) 모든 차원이 슬라이스로 처리된다. 단일 숫자 (콜론 없음) 만 있는 차원은 1부터 지정된 숫자까지인 것으로 처리된다. 예를 들면, 이 예제처럼 [2]는 [1:2]로 처리된다.

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

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

non-slice 경우에 혼동을 피하기 위해서는 [2][1:1]이 아닌 [1:2][1:1]과 같은 모든 차원에 대해 slice 구문을 사용하는 것이 가장 좋다.

슬라이스 지정자의 lower-bound 또는 upper-bound를 생략 할 수 있다. 다음 예와 같이 누락 된 바운드는 배열의 lower 또는 upper로 대체된다.

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

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

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

        schedule
------------------------
```

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

배열 첨자 표현식은 배열 자체 또는 첨자 표현식이 null인 경우 null을 리턴한다. 또한, null은 첨자가 배열 경계를 벗어나는 경우 리턴된다(이 경우에 에러가 나지는 않음). 예를 들어, schedule이 현재 차원 [1:3][1:2]인 경우 참조하는 schedule[3][3]은 NULL을 출력한다. 유사하게, 첨자 수가 잘못된 배열 참조는 에러가 아니라 null을 출력한다.

배열 슬라이스 표현식은 비슷하게, 배열 자체 또는 첨자 표현식이 null인 경우 null을 리턴한다. 그러나 현재 배열 경계를 완전히 벗어난 배열 슬라이스를 선택하는 것 같은 다른 사례에서 슬라이스 표현식은 null 대신 비어 있는 (0차원) 배열을 출력한다. 요청된 슬라이스가 부분적으로 배열 경계를 오버랩하는 경우 null을 리턴하는 대신 오버랩 영역으로 줄어든다.

배열 값의 현재 차원은 array_dims 함수를 사용하여 검색할 수 있다.

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

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

array_dims은 text 결과를 생성하는데, 이것은 사람에게는 가독성을 높이지만 프로그램에는 불편을 초래한다. 차원은 지정된 배열 차원의 상한계 및 하한계를 각각 리턴하는 array_upper 및 array_lower를 사용하여 검색할 수도 있다.

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

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

array_length는 지정된 배열 차원의 길이를 리턴한다.

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

 array_length
--------------
```

```
            2
(1 row)
```

cardinality는 모든 차원의 배열에서 엘리먼트의 총 수를 리턴한다. 사실상 unnest에 대한 호출이 생성하는 행의 수이다.

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

 cardinality
-------------
           4
(1 row)
```

Modifying Arrays

배열 값은 전체적으로 update 가능하다.

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

또는 ARRAY 식 구문을 사용한다.

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

배열은 단일 엘리먼트를 update 할 수 있다.

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

또는 일부만 update 가능하다.

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

생략 된 lower-bound 또는 upper-bound 슬라이스 구문 은 NULL이나 0이 아닌 배열 값을 업데이트 할 때만 사용할 수 있다.

저장된 배열 값은 아직 존재하지 않는 엘리먼트에 할당함으로써 확대 가능하다. 이전에 존재한 엘리먼트와 새로 할당된 엘리먼트 사이의 위치는 null로 채워진다. 예를 들면, 배열 myarray에 현재 4개의 엘리먼트가 있을 경우 myarray[6]에 할당하는 업데이트 후에 6개의 엘리먼트를 갖는다. myarray[5]는 null을 포함한다. 현재, 이러한

방식으로의 확대는 다차원 배열이 아닌 1차원 배열에만 허용된다. 첨자를 붙인 할당은 첨자가 1부터 시작되지 않는 배열의 생성을 허용한다. 예를 들면, 첨자 값이 -2~7인 배열을 생성하려면 myarray[-2:7]에 할당할 수 있다.

새 배열 값은 연결 (concatenation) 연산자 ||를 사용하여 생성할 수도 있다.

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

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]] as array;
        array
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

연결 (concatenation) 연산자를 사용하면 단일 엘리먼트를 1 차원 배열의 처음 또는 끝에 밀어 넣을 수 있다. 이것은 2개의 N 차원 배열 또는 N 차원 및 N+1 차원 배열을 수용한다.

다음 예와 같이 1차원 배열의 처음 또는 끝까지 단일 엘리먼트를 밀어서 넣을 경우 결과는 배열 피연산자와 동일한 하한계 첨자로 채워진 배열이다.

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

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

차원 수가 동일한 2개의 배열을 연결 (concatenation)하는 경우 결과는 왼쪽 피연산자의 외부 차원의 하한계 첨자를 보유한다. 다음 예와 같이 결과는 왼쪽 피연산자의 모든 엘리먼트를 구성하는 배열 다음에 오른쪽 피연산자의 모든 엘리먼트가 오는 것이다.

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

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

N+1 차원 배열의 처음부터 끝까지 N 차원 배열을 밀어 넣을 경우 결과는 위의 엘리먼트 배열 사례와 유사하다. 다음 예와 같이 각 N 차원 서브 배열은 본질적으로 N+1 차원 배열의 외부 차원의 엘리먼트이다.

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

함수 array_prepend, array_append 또는 array_cat을 사용하여 함수를 생성할 수도 있다. 처음 2개는 1차원 배열만 지원하지만, array_cat는 다차원 배열을 지원한다. 위에서 논의된 연결 (concatenation) 연산자는 이러한 함수의 직접 사용을 선호한다. 사실, 이 함수들은 기본적으로 연결 (concatenation) 연산자를 구현할 때 사용하기 위해 존재한다. 그러나, 사용자 정의된 집계를 생성할 때에도 직접적으로 유용할 수 있다. 함수의 사용은 아래와 같다.

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

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

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

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

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

간단한 경우, 위에서 설명한 연결 연산자가 이러한 함수를 직접 사용하는 것보다 선호된다. 그러나 연결 연산자가 세 가지 경우 모두를 처리하기 위해 오버로드되므로 함수 중 하나를 사용하면 모호성을 피하는 데 도움이 될 수도 있다.

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

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

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

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

위의 예에서 parser는 연결 연산자의 한쪽에 정수 배열을보고 다른 연산자에는 불확정 유형의 상수를 본다. 상수 유형을 분석하기 위해 사용하는 경험적 방법은 연산자의 다른 입력과 동일한 유형 (이 경우 정수 배열)이라고 가정하는 것이다. 그래서 연결 연산자는 array_append 가 아닌 array_cat으로 간주한다. 그게 잘못된 선택 인 경우, 상수를 배열의 엘리먼트 유형으로 캐스팅하여 수정할 수 있다. 그러나 array_append를 사용하는것이 바람직한 해결책일 수 있다.

Searching in Arrays

배열의 값을 검색하려면 각각의 값을 검사해야 한다. 배열 크기를 알고 있는 경우에는 수동으로도 가능하다. 예를 들면 다음과 같다.

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

단, 대형 배열일 때는 빨리 지루해지고, 배열 크기를 알 수 없는 경우에는 도움이 되지 않는다. 위의 쿼리는 다음으로 대체할 수 있다.

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

또한, 배열 값이 10000과 동일한 모든 행을 다음과 같이 찾을 수 있다.

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

또는, generate_subscripts 함수를 사용할 수 있다.

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

왼쪽 피연산자가 오른쪽 피연산자와 오버랩되는지 검사하는 && 연산자를 사용하여 배열을 검색할 수도 있다.

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

array_position와 array_positions함수를 사용하여 배열의 특정 값을 검색 할 수도 있다. 전자는 배열에서 첫 번째 값의 첨자를 반환하고 후자는 배열에있는 모든 값의 첨자가있는 배열을 반환한다.

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

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

Array Input and Output Syntax

배열 값의 외부 텍스트 표현은 배열 엘리먼트 타입에 대한 I/O 변환 규칙 및 배열 구조를 나타내는 데코레이션에 따라 해석되는 항목으로 구성된다. 데코레이션은 배열 값 앞뒤의 중괄호 및 인접 항목 사이의 구분자로 구성된다. 구분자는 보통 쉼표(,)이지만, 다른 것일 수 있다. 배열의 엘리먼트 타입에 대한 typedelim 설정에 의해 결정된다. 표준 데이터 타입 중에 세미콜론(;)을 사용하는 타입 box 외에는 모두 쉼표를 사용한다. 다차원 배열에서 각 차원(행, 평면, 큐브 등)은 자체 레벨의 중괄호를 갖고 있으며, 구분자는 동일 레벨의 인접한 중괄호 엔티티 사이에서 사용해야 한다.

배열 출력 루틴은 비어 있는 문자열이거나, 중괄호, 구분자, 큰따옴표, 역슬래시 또는 공백이 포함되어 있거나 단어 NULL과 일치할 경우에 엘리먼트 값 앞뒤에 큰따옴표를 사용한다. 엘리먼트 값에 임베드된 큰따옴표 및 역슬래시는 역슬래시로 이스케이프된다. 숫자 데이터 타입의 경우 큰따옴표가 절대 나타나지 않는다고 간주하는 것이 안전하지만, 텍스트 데이터 타입의 경우 인용부호의 존재 또는 부재에 대처해야 한다. 기본적으로, 배열의 차원에서 하한계 인덱스 값은 1로 설정된다. 다른 하한계를 사용하여 배열을 나타낼 경우 배열 첨자 범위는 배열 콘텐츠를 작성하기 전에 명시적으로 지정할 수 있다. 이러한 데코레이션은 각 배열 차원의 하한계 및 상한계 앞뒤에 대괄호([])가 구성되며, 사이 구분자는 콜론(:)이다. 배열 차원 데코레이션 뒤에는 등호(=)가 온다.

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

 e1 | e2
----+----
```

```
  1 |   6
(1 row)
```

배열 출력 루틴은 서로 다른 하한계가 하나 이상 있을 경우에만 결과에 명시적 차원이 포함된다.

엘리먼트에 대해 작성된 값이 NULL(변동의 경우) 인 경우 엘리먼트는 NULL로 간주된다. 인용부호 또는 역슬래시의 존재는 이것을 비활성화하고 리터럴 문자열 값 ``NULL'' 의 입력을 허용한다. 또한, array_nulls 구성 파라미터는 off로 설정해서 NULL이 NULL로 인식되는 것을 막을 수 있다. 앞에서 표시된 대로, 배열 값을 작성하는 경우 개별 배열 엘리먼트 앞뒤로 큰따옴표를 사용할 수 있다. 엘리먼트 값이 배열 값 파서를 잘못 혼동할 수 있을 경우에 이렇게 해야 한다. 예를 들면, 중괄호, 쉼표 (또는 데이터 타입의 구분자), 큰따옴표, 역슬래시, 선행 또는 후행 공백이 포함된 엘리먼트는 큰따옴표를 사용해야 한다. 비어 있는 문자열 및 NULL과 일치하는 문자열도 인용부호를 사용해야 한다. 큰따옴표 또는 역슬래시를 인용된 배열 엘리먼트 값에 넣으려면 이스케이프 문자열 구문을 사용하고 역슬래시를 앞에 넣어야 한다. 또는, 배열 구문으로 잘못 처리될 수도 있는 모든 데이터 문자를 보호하기 위해 인용부호를 피하고 역슬래시 이스케이핑을 사용할 수 있다.

왼쪽 중괄호 앞 또는 오른쪽 중괄호 뒤에 공백을 추가할 수 있다. 개별 항목 문자열 앞 또는 뒤에 공백을 추가할 수도 있다. 이 모든 사례에서 공백은 무시된다. 그러나, 큰따옴표 엘리먼트 내의 공백 또는 엘리먼트의 비 공백 문자 양쪽에 있는 공백은 무시된다.

4.2.9 Range Types

범위 타입은 일부 엘리먼트 타입의 값의 범위를 표현하는 데이터 타입이다. 예를 들면, 회의실이 예약된 시간 범위를 나타내기 위해 timestamp의 범위를 사용할 수 있다. 이 경우 데이터 타입은 tsrange(``timestamp range'' 의 단축어) 이고 timestamp은 서브타입이다. 서브타입은 엘리먼트 값이 값 범위 이내인지, 이전인지, 아니면 이후인지가 잘 정의 (well-defined) 되도록 총 순서를 갖고 있어야 한다.

다수의 엘리먼트 값을 하나의 범위 값으로 나타내고, 오버랩 범위 같은 개념을 명확하게 표현할 수 있기 때문에 범위 타입이 유용하다. 스케줄링을 위해 시간 및 날짜 범위를 사용하는 것은 가장 확실한 예제가 된다. 그러나 가격 범위, 기구의 측정 범위 등도 유용할 수 있다.

Built-in Range Types

다음과 같은 기본 제공 범위 타입이 제공된다.

int4range - Range of integer

int8range - Range of bigint

numrange - Range of numeric

tsrange - Range of timestamp without time zone

tstzrange - Range of timestamp with time zone

daterange - Range of date

또한, 자체 범위 타입을 정의할 수 있다. 자세한 내용은 User-defined Type을 참조 바란다.

Examples

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

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

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

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

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

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

Inclusive and Exclusive Bounds

비어 있지 않은 모든 범위는 2개의 경계, 하한계와 상한계를 갖는다. 이 값 사이의 모든 점은 범위에 포함된다. 경계 포함은 경계 점 자체는 범위에 포함된다는 것을 의미하며, 경계 제외는 경계 점이 범위에 포함되지 않음을 의미한다. 범위의 텍스트 양식에서, 하한계 포함은 ``[``로 표현되고 하한계 제외는''(``로 표현된다. 비슷하게, 상한계 포함은'']'' 로 표현되고, 상한계 제외는 ``)'' 로 표현된다.

함수 lower_inc 및 upper_inc는 범위 값의 하한계 및 상한계를 각각 테스트한다.

Infinite (Unbounded) Ranges

범위의 하한계는 생략할 수 있는데, 상한계 미만의 모든 점들이 범위에 포함된다는 것을 의미한다. 비슷하게, 범위의 상한계가 생략된 경우 하한계 이상의 모든 점들이 범위에 포함된다. 하한계 및 상한계 모두 생략되면

엘리먼트 타입의 모든 값이 범위에 포함되는 것으로 간주된다.
이것은 하한계가 ``음의 무한'' 또는 상한계가 ``양의 무한''인 것으로 간주하는 것과 동일하다. 그러나, 이러한 무한 값은 범위의 엘리먼트 타입의 값이 절대 아니며, 범위의 일부일 수 없다는 점에 유의하라. (따라서, 무한 경계 포함 같은 류가 없다. 한 가지를 작성하려고 하면 자동으로 경계 제외로 변환된다.)
또한, 일부 엘리먼트 타입은 ``무한'' 표시법을 갖고 있지만, 범위 타입 메커니즘에 관한 한 이것은 또 다른 값이다. 예를 들면, 타임 스탬프 범위에서 [today,]는 [today,) 와 동일한 것을 의미한다. 그러나 [today,infinity]는 [today,infinity) 와 의미가 다를 때도 있다. 후자는 특수 timestamp 값 infinity가 제외된다.
함수 lower_inf 및 upper_inf는 각각 범위의 무한 하한계 및 상한계를 테스트한다.

Range Input/Output

범위 값에 대한 입력은 다음 패턴 중 하나를 따라야 한다.

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

소괄호 또는 대괄호는 앞에서 설명한 대로 하한계 및 상한계가 제외 또는 포함인지를 나타낸다. 마지막 패턴은 empty이며, 이것은 비어 있는 범위를 나타낸다(아무런 점이 없는 범위). lower-bound는 서브타입에 대한 유효 입력인 문자열이거나, 하한계가 없을 경우 비워둘 수 있다. 비슷하게, upper-bound는 서브타입에 대한 유효 입력인 문자열이거나, 상한계가 없을 경우 비워둘 수 있다. 각 경계 값은 ``(큰따옴표) 문자를 사용하여 인용할 수 있다. 경계 값에 소괄호, 대괄호, 쉼표, 큰따옴표 또는 역슬래시가 포함된 경우, 이러한 문자가 범위 구문의 일부로 잘못 간주될 수도 있으므로 이것은 필수이다. 큰따옴표 또는 역슬래시를 인용된 경계 값에 넣으려면 역슬래시를 앞에 넣어야 한다. (또한, 큰따옴표로 인용된 경계 값 내의 큰따옴표 쌍은 SQL 리터럴 문자열의 작은따옴표에 대한 규칙과 비슷하게 큰따옴표 문자로 간주된다.) 또는, 범위 구문으로 잘못 처리될 수도 있는 모든 데이터 문자를 보호하기 위해 인용을 피하고 역슬래시 이스케이핑을 사용할 수 있다. 또한, 비어 있는 문자열인 경계 값을 작성할 경우, 아무것도 안 쓰면 무한 경계를 의미하게 되므로''``를 작성한다. 공백은 범위 값 전후에 허용되지만, 소괄호 또는 대괄호 사이의 공백은 하한계 또는 상한계 값의 일부로 간주된다. (엘리먼트 타입에 따라 중요할 수도 있고 중요하지 않을 수도 있다.)
Examples:

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


-- does not include either 3 or 7, but includes all points in between
```

```
SELECT '(3,7)'::int4range;

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

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

Constructing Ranges

각 범위 타입은 범위 타입과 동일한 이름의 생성자 함수를 갖는다. 생성자 함수를 사용하는 것은 경계 값의 추가 인용을 할 필요가 없으므로 범위 리터럴 상수를 작성하는 것보다 좀 더 편리하다. 생성자 함수는 2 또는 3개의 인수를 수용한다. 3개 인수 양식은 세 번째 인수에서 지정된 양식의 경계로 범위를 생성하는 반면, 2개 인수 양식은 표준 양식의 범위를 생성한다(하한계 포함, 상한계 제외). 세 번째 인수는 문자열 ``()'', ``(]'', ``[)'' 또는 ``[]'' 중 하나여야 한다.

Examples:

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

-- If the third argument is omitted, '[)' is assumed.
SELECT numrange(1.0, 14.0);

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

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

Discrete Range Types

불연속 범위는 엘리먼트 타입이 integer 또는 date 같이 잘 정의된(well-defined) ``단계별'' 타입이다. 이러한 타입에서, 2개의 엘리먼트 사이에 유효 값이 없을 경우 인접하다고 말할 수 있다. 연속 범위와는 대조적으로, 이것은 항상(또는 거의 대부분) 2개의 주어진 값 사이에서 다른 엘리먼트 값을 인식하는 것이 가능하다. 예를

들면, timestamp를 벗어난 범위와 마찬가지로 numeric 타입을 벗어난 범위는 연속이다. (timestamp는 정밀도에 제한이 있어서 이론적으로 불연속으로 처리될 수 있지만 단계의 크기가 관심사가 아닐 때는 연속으로 간주하는 것이 낫다.) 불연속 범위 타입에 대해 생각하는 또 다른 방법은 각 엘리먼트 값에 대해 ``다음'' 또는 ``이전'' 값에 대한 명확한 방안이 있는 것이다. 주어진 원래의 것 대신, 다음 또는 이전 엘리먼트 값을 선택함으로써 범위의 경계를 포함하는 표현과 제외하는 표현 사이의 변환이 가능하다는 점을 알고 있어야 한다. 예를 들면, 정수 범위 타입 [4,8] 및 (3,9) 는 동일한 값 집합을 나타낸다. 그러나 수치상 범위일 경우에는 그렇지 않다. 불연속 범위 타입은 엘리먼트 타입에 대해 원하는 단계 크기를 인식하는 정규화 (canonicalization) 함수를 가져야 한다. 정규화 (canonicalization) 함수는 특히 일관된 포함 또는 제외 범위의 항등 표현을 갖는 범위 타입으로 값을 동등하게 변환할 책임이 있다. 정규화 (canonicalization) 함수가 지정되지 않으면 사실상 동일한 값 집합을 나타내더라도 형식이 서로 다른 범위는 항상 비등가로 처리된다. 기본 제공 범위 타입 int4range, int8range 및 daterange는 모두 하한계는 포함하고 상한계는 제외하는, 즉 [) 인 정규형 (canonical) 양식을 사용한다. 그러나, 사용자가 정의한 범위 타입은 다른 표기법을 사용할 수 있다.

Defining New Range Types

사용자는 자신만의 범위 타입을 정의할 수 있다. 이렇게 하는 가장 일반 이유는 기본 제공 범위 타입으로 제공되지 않는 서브타입의 범위를 사용하기 위해서이다. 서브타입 float8의 새 범위 타입을 정의하는 예이다.

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

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

float8은 의미가 있는 ``단계''가 아니므로 우리는 이 예제에서 정규화 (canonicalization) 함수를 정의하지 않는다. 서브타입이 연속 값이 아닌 불연속 값을 갖는 경우 CREATE TYPE 명령은 canonical 함수를 지정해야 한다. 정규화 (canonicalization) 함수는 입력 범위 값을 취하고, 경계와 형식이 다를 수 있는 등가의 범위 값을 리턴해야 한다. 정수 범위 [1, 7] 및 [1, 8) 처럼, 동일한 값 집합을 나타내는 2개의 범위에 대한 정규형 (canonical) 출력은 동일해야 한다. 형식이 서로 다른 2개의 등가 값이 항상 동일한 형식의 동일한 값으로 매핑되는 한, 어떤 표현을 정규형 (canonical) 으로 선택하든 상관 없다. 경계 포함/제외 형식을 조절하는 것 외에, 서브타입이 저장 가능한 크기 보다 원하는 단계 크기가 클 경우에 정규화 (canonicalization) 함수는 경계 값을 잘 처리할 수 있다. 예를 들면, timestamp를 벗어난 범위 타입은 단계 크기가 시간인 것으로 정의할 수 있다. 이때 정규화 (canonicalization) 함수는 시간의 배수가 아닐 경우 반올림이 필요할 수 있거나 그 대신 에러가 발생할 수 있다. 자신만의 범위 타입을 정의하면, 주어진 범위에 어떤 값이 속하는지를 결정하는 정렬 순서를 변경하기 위해 서로 다른 서브 타입 B-tree 연산자 클래스 또는 사용할 콜레이션을 지정할 수 있다. 또한, GiST 또는 SP-GiST 인덱스에서 사용하기 위한 범위

타입은 서브타입 차이 또는 subtype_diff, 함수를 정의해야 한다. (인덱스는 subtype_diff 없이도 작동되지만 차이 함수가 제공된 경우보다 효율이 상당히 떨어진다.) 서브타입 차이 함수는 서브타입의 입력 값 2개를 취하고 float8 값으로 표현된 차이를 리턴한다 (예 : X 빼기 Y). 위의 예제에서 일반 float8 빼기 연산자의 바탕이 되는 함수를 사용할 수 있지만 그 외 서브타입의 경우 일부 타입 변환이 필요할 수 있다. 차이를 숫자로 표현하는 방법에 대한 몇 가지 창의적 생각이 필요할 수도 있다. 가능한 한 가장 큰 범위까지 subtype_diff 함수는 선택된 연산자 클래스 및 콜레이션에서 암시하는 정렬 순서에 동의해야 한다. 즉, 이것의 결과는 정렬 순서에 따라 첫 번째 인수가 두 번째 인수보다 클 때에는 항상 양의 값이어야 한다.

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

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

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

범위 타입 생성에 대한 자세한 내용은 User-defined Type을 참조 바란다.

Indexing

GiST 및 SP-GiST 인덱스는 범위 타입의 테이블 컬럼을 생성할 수 있다. GiST 인덱스를 생성하는 예이다.

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

GiST 또는 SP-GiST 인덱스는 이러한 범위 연산자 =, &&, <@, @>, <<, >>, -|-, &< 및 &>와 관련된 쿼리를 가속화할 수 있다.

또한, B-tree 및 해시 인덱스는 범위 타입의 테이블 컬럼을 생성할 수 있다. 이러한 인덱스 타입의 경우 기본적으로 유일하게 유용한 범위 연산은 같음이다. 해당 < 및 > 연산자를 사용하고 범위 값에 대해 정의되는 B-tree 정렬 순서가 있지만, 순서가 제멋대로라서 실세계에서는 별로 유용하지 않다. 범위 타입의 B-tree 및 해시 지원은 실제 인덱스를 생성하기 보다는 주로 쿼리 내부적으로 정렬 및 해싱을 허용하기 위한 것이다.

Constraints on Ranges

UNIQUE는 스칼라 값에 대해 자연스러운 제약 조건이지만, 범위 타입에는 일반적으로 적절하지 않다. 대신, 주로 제외 제약 조건이 좀 더 적절하다. 제외 제약 조건은 범위 타입에서 ``비 오버랩'' 같은 제약 조건의 사양을 허용한다.

Examples:

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

해당 제약 조건은 오버랩되는 값이 테이블에 동시에 존재하지 못하게 한다.

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

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

btree_gist 확장을 사용하여 일반 스칼라 데이터 타입에서 제외 제약 조건을 정의할 수 있는데, 그럴 경우 최대한의 유연성으로 범위 제외와 결합이 가능하다. 예를 들어, btree_gist가 설치된 후 다음 제약 조건은 회의실 수가 동일한 경우에만 오버랩되는 범위를 거부한다.

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

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

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint "room_reservation_room_during_excl"
```

```
DETAIL:  Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01 15:30:00")) conflicts
with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01 15:00:00")).

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

4.2.10 User-defined Type

CREATE TYPE 명령을 사용하여 새로운 타입을 추가할 수 있다. CREATE TYPE은 아래의 구문과 같이 Composite Type, Enum Type, Range Type, Base Type, Shell Type의 다섯가지 형식이 있다.

Syntex :

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

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

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

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
```

```
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name
```

Examples :

Composite Type을 작성하고 이를 함수 정의에 사용하는 예제이다.

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

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

Enum Type을 만들고 테이블 정의에 사용하는 예제이다.

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

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

Range Type을 만드는 예제이다.

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

Base Type을 만든 다음 테이블 정의에 형식을 사용하는 예제이다.

```
CREATE TYPE box;

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

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

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

4.3 functions

4.3.1 Comparison functions

- num_nonnulls(VARIADIC ``any'')

  Null값이 아닌 인자들의 수를 리턴한다.

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

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

- num_nulls(VARIADIC ``any'')

  Null값을 가지는 인자들의 수를 리턴한다.

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


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

4.3.2 Mathematics functions

수에 관련된 다양한 함수를 제공하며, dp라고 명시되어 있는 인자는 double precision을 뜻한다.

- abs(x)

  abs() 함수는 인자의 절대값을 반환한다.

```
SELECT abs(-17.4);


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

- cbrt(dp)

  cbrt() 함수는 인자의 세제곱근 값을 반환한다.

```
SELECT cbrt(27.0);


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

- ceil(dp or numeric) 또는 ceiling(dp or numeric)

  ceil() 함수는 인자보다 크거나 같은 가장 가까운 정수값을 반환한다.

```
SELECT ceil(-42.8);


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

- degrees(dp)

  degrees() 함수는 radian값을 degree값으로 변환하여 반환한다.

```
SELECT degrees(0.5);


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

- div(y numeric, x numeric)

  div() 함수는 y/x값의 정수 지수를 반환한다.

```
SELECT div(9,4);


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

- exp(dp or numeric)

  exp() 함수는 지수함수 값을 반환한다.

```
SELECT exp(1.0);


Result:
        exp
```

```
--------------------
 2.7182818284590452
            (1 row)
```

- floor(dp or numeric)

  floor() 함수는 인자보다 작거나 같은 가장 가까운 정수값을 반환한다.

```
SELECT floor(-42.8);

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

- ln(dp or numeric)

  ln() 함수는 인자의 자연로그 값을 반환한다.

```
SELECT ln(2.0);

Result:
        ln
-------------------
 0.693147180559945
           (1 row)
```

- log(dp or numeric)

  log() 함수는 밑이 10인 대수값으로 반환한다.

```
SELECT log(100.0);

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

- log(b numeric, x numeric)

이 함수는 인자 b를 밑으로 하는 대수값을 반환한다.

```
SELECT log(2.0, 64.0);

Result:
        log
--------------------
 6.0000000000000000
             (1 row)
```

- mod(y, x)

  mod() 함수는 y/x의 나머지 값을 반환한다.

```
SELECT mod(9,4);

Result:
 mod
-----
   1
(1 row)
```

- pi()

  pi() 함수는 ∏의 상수값을 반환한다.

```
SELECT pi();

Result:
        pi
------------------
 3.14159265358979
           (1 row)
```

- power(a dp, b dp) 또는 power(a numeric, b numeric)

  power() 함수는 인자 a를 인자 b만큼 제곱한 값을 반환한다.

```
SELECT power(9.0, 3.0);

Result:
```

```
        power
--------------------
 729.000000000000000
            (1 row)
```

- radians(dp)

  radians() 함수는 degree값을 radian값으로 변환하여 반환한다.

```
SELECT radians(45.0);

Result:
      radians
-------------------
 0.785398163397448
            (1 row)
```

- round(dp or numeric)

  round() 함수는 인자의 가장 가까운 정수로 반올림한 값을 반환한다.

```
SELECT round(42.4);

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

- round(v numeric, s int)

  이 함수는 인자 v의 인자 s번째 소수자리로 반올림한 값을 반환한다.

```
SELECT round(42.4382, 2);

Result:
 round
-------
 42.44
(1 row)
```

- scale(numeric)

  scale() 함수는 인자의 소수자릿수를 반환한다.

```
SELECT scale(8.41);

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

- sign(dp or numeric)

  sign() 함수는 인자의 부호 (-1, 0, 1) 를 반환한다.

```
SELECT sign(-8.4);

Result:
 sign
------
   -1
(1 row)
```

- sqrt(dp or numeric)

  sqrt() 함수는 인자의 제곱근을 반환한다.

```
SELECT sqrt(2.0);

Result:
        sqrt
-------------------
 1.414213562373095
            (1 row)
```

- trunc(dp or numeric)

  trunc() 함수는 인자값에서 소수점을 버린 값을 반환한다.

```
SELECT trunc(42.8);

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

- trunc(v numeric, s int)

  이 함수는 인자 v값에서 인자 s 소수자릿수로 자른 값을 반환한다.

```
SELECT trunc(42.4382, 2);

Result:
 trunc
-------
 42.43
(1 row)
```

- width_bucket(operand dp, b1 dp, b2 dp, count int) 또는 width_bucket(operand numeric, b1 numeric, b2 numeric, count int)

  width_bucket() 함수는 범위 b1에서 b2에 걸쳐있는 동일한 폭의 버킷을 가진 막대 그래프에서 피연산자가 할당 될 버킷 번호를 반환한다. 범위를 벗어나는 입력에 대해서는 0 또는 count+1을 반환한다.

```
SELECT width_bucket(5.35, 0.024, 10.06, 5);

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

- width_bucket(operand anyelement, thresholds anyarray)

  버킷의 하한을 나열한 배열이 주어지면 피연산자가 할당될 버킷 번호를 반환한다. 첫번째 하한보다 작은 입력에 대해 0을 반환한다. thresholds 배열은 가장 작은 것부터 반드시 정렬되야한다.

```
SELECT width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[]);


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

4.3.3 String functions

- ascii(string)

  ascii() 함수는 인자의 첫 문자의 ASCII 코드를 반환한다. UTF8의 경우 문자의 유니코드 코드포인트를 반환한다. 다른 멀티바이트 인코딩의 경우 인자는 ASCII 문자여야 한다.

```
SELECT ascii('x');


Result:
 ascii
-------
   120
(1 row)
```

- btrim(string text [, characters text])

  btrim() 함수는 string의 시작과 끝에서 characters(기본적으로 공백)의 문자로만 구성되는 가장 긴 문자열을 제거한 값을 반환한다.

```
SELECT btrim('xyxtrimyyx', 'xyz');


Result:
 btrim
-------
  trim
(1 row)
```

- chr(int)

  chr() 함수는 입력된 코드를 가진 문자를 반환한다. UTF8의 경우 인자는 유니코드 코드포인트로 처리된다. 다른 멀티바이트 인코딩의 경우 인자는 ASCII 문자를 지정해야 한다. 텍스트 데이터 유형은 이러한 바이트를

저장할 수 없기 때문에 NULL(0) 문자는 허용되지 않는다.

```
SELECT chr(65);

Result:
 chr
-----
   A
(1 row)
```

- concat(str ``any'' [, str ``any'' [, ...]])

  concat() 함수는 모든 인자의 텍스트 표현을 연결한 값을 반환한다. NULL 인자는 무시된다.

```
SELECT concat('abcde', 2, NULL, 22);

Result:
  concat
----------
 abcde222
   (1 row)
```

- concat_ws(sep text, str ``any'' [, str ``any'' [, ...]])

  concat_ws() 함수는 첫 번째 인자를 제외한 모든 인자를 구분 기호로 연결한 값을 반환한다. 첫번째 인자가 구분자로 사용된다. NULL 인자는 무시된다.

```
SELECT concat_ws(',', 'abcde', 2, NULL, 22);

Result:
  concat_ws
------------
 abcde,2,22
     (1 row)
```

- convert(string bytea, src_encoding name, dest_encoding name)

  convert() 함수는 문자열을 `dest_encoding`으로 변환하여 반환한다. 원본 인코딩은 `src_encoding`에 의해 지정된다. 문자열은 이 인코딩에서 유효해야 한다. 변환은 CREATE CONVERSION으로 정의할 수 있다. 또한 사전 정의된 변환이 있으며, 그 목록은 링크에서 확인할 수 있다.

```
SELECT convert('text_in_utf8', 'UTF8', 'LATIN1');

Result:
          convert
----------------------------
 x746578745f696e5f75746638
                     (1 row)
```

- convert_from(string bytea, src_encoding name)

  convert_from() 함수는 문자열을 데이터베이스 인코딩으로 변환하여 반환한다. 원본 인코딩은 `src_encoding`에 의해 지정되며, 문자열은 이 인코딩에서 유효해야 한다.

```
SELECT convert_from('text_in_utf8', 'UTF8');

Result:
 convert_from
--------------
 text_in_utf8 (현재 데이터베이스 인코딩으로 표현된 문자열)
       (1 row)
```

- convert_to(string text, dest_encoding name)

  convert_to() 함수는 문자열을 `dest_encoding`으로 변환하여 반환한다.

```
SELECT convert_to('some text', 'UTF8');

Result:
      convert_to
----------------------
  x736f6d652074657874
               (1 row)
```

- decode(string text, format text)

  decode() 함수는 `string`의 텍스트 표현에서 바이너리 데이터를 디코딩한 값을 반환한다. format에 대한 옵션은 `encode`과 동일하다.

```
SELECT decode('MTIzAAE=', 'base64');


Result:
    decode
--------------
  x3132330001
       (1 row)
```

- encode(data bytea, format text)

  encode() 함수는 바이너리 데이터를 텍스트 표현으로 인코딩한 값을 반환한다. 지원되는 형식은 base64, hex, escape이다. escape는 0바이트 및 high-bit-set 바이트를 8진수 시퀀스(\nnn) 및 이중 역슬래쉬로 변환한다.

```
SELECT encode(E'123\\000\\001', 'base64');


Result:
  encode
----------
 MTIzAAE=
   (1 row)
```

- format(formatstr text [, formatarg ``any'' [, …]])

  format() 함수는 형식 문자열에 따라 인자를 형식화한 값을 반환한다. 이 함수는 C함수 sprintf와 유사합니다.

```
SELECT format('Hello %s, %1$s', 'World');


Result:
       format
--------------------
 Hello World, World
             (1 row)
```

- initcap(string)

  initcap() 함수는 각 단어의 첫 글자를 대문자로 변환하고 나머지는 소문자로 변환하여 반환한다. 단어는 영숫자가 아닌 문자로 구분된 일련의 영숫자 문자이다.

```
SELECT initcap('hi THOMAS');


Result:
  initcap
-----------
 Hi Thomas
    (1 row)
```

- length(string)

  length 함수는 string의 문자수를 반환한다.

```
SELECT length('jose');


Result:
 length
--------
      4
 (1 row)
```

- length(string bytea, encoding name)

  이 함수는 지정된 encoding의 string 수를 반환한다. string은 이 인코딩에서 유효해야 한다.

```
SELECT length('jose', 'UTF8');


Result:
 length
--------
      4
 (1 row)
```

- lpad(string text, length int [, fill text])

  lpad 함수는 문자열 fill(기본적으로 공백)을 앞에 추가하여 길이 length에 string을 채운 값을 반환한다. string이 이미 length보다 길면 잘린다.

```
SELECT lpad('hi', 5, 'xy');


Result:
```

```
 lpad
-------
 xyxhi
(1 row)
```

- ltrim(*string* text [, *characters* text])

  ltrim 함수는 *string*의 시작부분부터 문자열 *characters*만 포함된 가장 긴 문자열을 제거한다(기본값은 공백).

```
SELECT ltrim('zzzytest', 'xyz');

Result:
 ltrim
-------
  test
(1 row)
```

- md5(string)

  md5 함수는 *string*의 MD5 해시를 계산하여 결과를 16진수로 반환한다.

```
SELECT md5('abc');

Result:
                md5
----------------------------------
 900150983cd24fb0d6963f7d28e17f72
                            (1 row)
```

- parse_ident(*qualified_identifier* text [, *strictmode* boolean DEFAULT true])

  parse_ident 함수는 *qualified_identifier*를 식별자들의 배열로 분할하여, 개별 식별자의 인용 부호를 제거한 값을 반환한다. 기본적으로 마지막 식별자 다음의 추가 문자는 오류로 간주된다. 두번째 매개 변수가 *false* 면 이러한 추가 문자는 무시된다. 이 동작은 함수와 같은 개체의 이름을 파싱하는데 유용하다. 이 함수는 길이를 초과하는 식별자를 자르지 않는다. 잘라내기를 원하면 결과를 *name[]*로 형변환 할 수 있다.

```
SELECT parse_ident('"SomeSchema".someTable');

Result:
       parse_ident
```

```
-------------------------
 {SomeSchema,sometable}
                 (1 row)
```

- pg_client_encoding()

  pg_client_encoding 함수는 현재 클라이언트의 인코딩명을 반환한다.

```
SELECT pg_client_encoding();

Result:
 pg_client_encoding
---------------------
     SQL_ASCII
            (1 row)
```

- quote_ident(*string* text)

  quote_ident 함수는 질의문 문자열에서 식별자로 사용하기 위해 적절하게 인용된 문자열을 반환한다. 따옴표는 필요한 경우에만 추가된다(즉, 문자열에 식별자가 아닌 문자가 포함되거나 대소문자가 포함될 경우). 내장된 따옴표는 적절하게 2개씩이다.

```
SELECT quote_ident('Foo bar');

Result:
 quote_ident
-------------
   "Foo bar"
      (1 row)
```

- quote_literal(*string* text)

  quote_literal 함수는 질의문 문자열에서 문자열 리터럴로 사용하기 위해 적절하게 인용된 문자열을 반환한다. 내장된 작은 따옴표와 역슬래시는 적절하게 2개씩이다. *quote_literal*은 null 입력시 null을 반환한다. 인자가 NULL일 수 있는 경우 *quote_nullable*이 더 적합하다.

```
SELECT quote_literal(E'O\'Reilly');

Result:
 quote_literal
```

```
---------------
   'O''Reilly'
        (1 row)
```

- quote_literal(value anyelement)

  이 함수는 입력된 값을 텍스트로 강제 변환한 다음 리터럴로 인용된 값을 반환한다. 내장된 작은 따옴표와 백슬래시는 적절하게 2개씩이다.

```
SELECT quote_literal(42.5);


Result:
 quote_literal
---------------
        '42.5'
        (1 row)
```

- quote_nullable(*string* text)

  quote_nullable 함수는 질의문의 문자열에서 문자열 리터럴로 사용하기 위해 적절하게 인용된 문자열을 반환한다. 또는 인자가 NULL인 경우 NULL을 리턴한다. 내장된 작은 따옴표와 백슬래시는 적절하게 2개씩이다.

```
SELECT quote_nullable(NULL);


Result:
 quote_nullable
---------------
          NULL
        (1 row)
```

- quote_nullable(*value* anyelement)

  이 함수는 입력된 값을 텍스트로 강제 변환한 다음 리터럴로 인용된 값을 반환한다. 또는 인자가 NULL인 경우 NULL을 리턴한다. 내장된 작은 따옴표와 백슬래시는 적절하게 2개씩이다.

```
SELECT quote_nullable(42.5);


Result:
 quote_nullable
```

```
----------------
         '42.5'
         (1 row)
```

- regexp_matches(*string* text, *pattern* text [, *flags* text])

  regexp_matches 함수는 문자열에 대해 POSIX 정규식과 일치하는 결과로 캡처된 모든 부분 문자열을 반환한다.

```
SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');

Result:
 regexp_matches
----------------
    {bar,beque}
         (1 row)
```

- regexp_replace(*string* text, *pattern* text, *replacement* text [, *flags* text])

  regexp_replace 함수는 POSIX 정규식과 일치하는 부분 문자열을 대체하여 반환한다.

```
SELECT regexp_replace('Thomas', '.[mN]a.', 'M');

Result:
 regexp_replace
----------------
            ThM
         (1 row)
```

- regexp_split_to_array(string text, pattern text [, flags text])

  regexp_split_to_array 함수는 POSIX 정규식을 구분 기호로 사용하여 문자열을 분할하여 반환한다.

```
SELECT regexp_split_to_array('hello world', E'\\s+');

Result:
 regexp_split_to_array
------------------------
         {hello,world}
                (1 row)
```

- regexp_split_to_table(*string* text, *pattern* text [, *flags* text])

  regexp_split_to_table 함수는 POSIX 정규식을 구분자로 사용하여 문자열을 분할하여 반환한다.

```
SELECT regexp_split_to_table('hello world', E'\\s+');

Result:
 regexp_split_to_array
-----------------------
                 hello
                 world
               (2 rows)
```

- repeat(*string* text, *number* int)

  repeat 함수는 지정한 횟수만큼 문자열 반복하여 반환한다.

```
SELECT repeat('Pg', 4);

Result:
  repeat
----------
 PgPgPgPg
   (1 row)
```

- replace(*string* text, *from* text, *to* text)

  replace 함수는 *string*에서 발견된 모든 *from*과 같은 부분 문자열을 *to* 문자열로 바꾸어 반환한다.

```
SELECT replace('abcdefabcdef', 'cd', 'XX');

Result:
    replace
--------------
 abXXefabXXef
       (1 row)
```

- reverse(*str*)

  reverse 함수는 역순의 문자열을 반환한다.

```
SELECT reverse('abcde');

Result:
 reverse
---------
   edcba
  (1 row)
```

- right(*str* text, *n* int)

  right 함수는 문자열의 마지막 문자 *n*개를 반환한다. *n*이 음수이면 첫 |*n*|문자를 제외한 모든 문자를 반환한다.

```
SELECT right('abcde', 2);

Result:
 right
-------
    de
(1 row)
```

- rpad(*string* text, *length* int [, *fill* text])

  rpad 함수는 *string* 뒤에 길이 *length*까지 *fill*(기본값은 공백) 문자들을 덧붙여 채운 값을 반환한다. *string*이 이미 *length*보다 길면 잘린다.

```
SELECT rpad('hi', 5, 'xy');

Result:
 rpad
-------
 hixyx
(1 row)
```

- rtrim(*string* text [, *characters* text])

  rtrim 함수는 *string*의 끝부분부터 *characters*에 있는 문자만 포함된 가장 긴 문자열을 제거한다(기본값은 공백).

```
SELECT rtrim('testxxzx', 'xyz');


Result:
 rtrim
-------
  test
(1 row)
```

- split_part(*string* text, *delimiter* text, *field* int)

  split_part 함수는 *delimiter*로 문자열을 분할하고, 입력된 *field*번째 문자열을 반환한다(1부터 카운팅).

```
SELECT split_part('abc~@~def~@~ghi', '~@~', 2);


Result:
 split_part
------------
        def
     (1 row)
```

- strpos(*string*, *substring*)

  strpos 함수는 지정된 *substring*의 위치를 반환한다(*position*(*string*의 *substring*) 과 같으나 인자의 순서가 반대임).

```
SELECT strpos('high', 'ig');


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

- substr(*string*, *from* [, *count*])

  substr 함수는 *from*번째에서 *count*만큼의 *substring*을 추출하여 반환한다.

```
SELECT substr('alphabet', 3, 2);


Result:
```

```
 substr
--------
    ph
 (1 row)
```

- to_ascii(*string* text [, *encoding* text])

  to_ascii 함수는 다른 인코딩으로부터 *string*을 ASCII로 변환하여 반환한다 (LATIN1, LATIN2, LATIN9 및 WIN1250 인코딩의 변환 만 지원).

```
SELECT to_ascii('Karel');


Result:
 to_ascii
----------
   Karel
   (1 row)
```

- to_hex(*number* int or bigint)

  to_hex 함수는 숫자를 16진수 표현으로 변환하여 반환한다.

```
SELECT to_hex(2147483647);


Result:
  to_hex
----------
 7fffffff
   (1 row)
```

- translate(*string* text, *from* text, *to* text)

  translate 함수는 *from* 집합의 문자와 일치하는 문자열의 모든 문자는 *to* 집합의 해당 문자로 바뀌어 반환한다. *from*이 *to*보다 긴 경우 *from*의 여분의 문자는 제거된다.

```
SELECT translate('12345', '143', 'ax');


Result:
 translate
-----------
```

```
a2x5
(1 row)
```

4.3.4 Binary String functions

인자를 구분하기 위해 쉼표가 아닌 키워드를 사용하는 일부 문자열 함수를 정의한다.

- octet_length(string)

  octet_length 함수는 바이너리 문자열의 바이트 수를 반환한다.

```
SELECT octet_length(E'jo\\000se'::bytea);


Result:
 octet_length
--------------
            5
       (1 row)
```

- overlay(string placing string from int [for int])

  overlay 함수는 substring을 대체하여 변환한다.

```
SELECT overlay(E'Th\\000omas'::bytea placing E'\\002\\003'::bytea FROM 2 for 3);


Result:
     overlay
----------------
  x5402036d6173
         (1 row)
```

- position(substring in string)

  position 함수는 지정된 substring의 위치를 반환한다.

```
SELECT position(E'\\000om'::bytea in E'Th\\000omas'::bytea);


Result:
 position
----------
```

```
        3
   (1 row)
```

- substring(string [from int] [for int])

  substring 함수는 substring을 추출한 값을 반환한다.

```
SELECT substring(E'Th\\000omas'::bytea FROM 2 for 3);


Result:
 substring
-----------
  x68006f
    (1 row)
```

- trim([both] bytes from string)

  trim 함수는 문자열의 시작과 끝에서 바이트로 나타나는 바이트만 포함된 가장 긴 문자열을 제거하여 반환한다.

```
SELECT trim(E'\\000\\001'::bytea FROM E'\\000Tom\\001'::bytea);


Result:
   trim
----------
  x546f6d
    (1 row)
```

- btrim(*string* bytea, *bytes* bytea)

  btrim 함수는 문자열의 시작과 끝에서 바이트로 나타나는 바이트만 포함된 가장 긴 문자열을 제거하여 반환한다.

```
SELECT btrim(E'\\000trim\\001'::bytea, E'\\000\\001'::bytea);


Result:
    btrim
------------
 x7472696d
     (1 row)
```

- decode(*string* text, *format* text)

  decode 함수는 *string*의 텍스트 표현에서 바이너리 데이터를 디코딩하여 반환한다. 형식에 대한 옵션은 인코딩과 동일하다.

```
SELECT decode(E'123\\000456', 'escape');

Result:
      decode
------------------
 x31323300343536
           (1 row)
```

- encode(*data* bytea, *format* text)

  encode 함수는 바이너리 데이터를 텍스트 표현으로 인코딩한 값을 반환한다. 지원되는 형식은 `base64`, `hex`, `escape`이다. `escape`는 0바이트 및 high-bit-set 바이트를 8진수 시퀀스(*nnn*) 및 이중 역슬래쉬로 변환한다.

```
SELECT encode(E'123\\000456'::bytea, 'escape');

Result:
   encode
------------
 123\000456
     (1 row)
```

- get_bit(*string*, *offset*)

  get_bit 함수는 문자열에서 비트를 추출하여 반환한다.

```
SELECT get_bit(E'Th\\000omas'::bytea, 45);

Result:
 get_bit
---------
       1
  (1 row)
```

- get_byte(*string*, *offset*)

  get_byte 함수는 문자열에서 바이트를 추출하여 반환한다.

```
SELECT get_byte(E'Th\\000omas'::bytea, 4);


Result:
 get_byte
----------
      109
   (1 row)
```

- length(*string*)

  length 함수는 바이너리 문자열의 길이를 반환한다.

```
SELECT length(E'jo\\000se'::bytea);


Result:
 length
--------
      5
 (1 row)
```

- md5(*string*)

  md5 함수는 *string*의 MD5 해시를 계산하여 결과를 16진수로 반환한다.

```
SELECT md5(E'Th\\000omas'::bytea);


Result:
                md5
----------------------------------
 8ab2d3c9689aaf18b4958c334c82d8b1
                          (1 row)
```

- set_bit(*string*, *offset*, *newvalue*)

  set_bit 함수는 문자열에 비트를 설정한 값을 반환한다.

```
SELECT set_bit(E'Th\\000omas'::bytea, 45, 0);


Result:
      set_bit
```

```
------------------
 x5468006f6d4173
           (1 row)
```

- set_byte(*string, offset, newvalue*)

  set_byte 함수는 문자열에 바이트를 설정한 값을 반환한다.

```
SELECT set_byte(E'Th\\000omas'::bytea, 4, 64);


Result:
     set_byte
------------------
 x5468006f406173
           (1 row)
```

4.3.5 Date Type Formatting functions

Formatting function은 다양한 데이터 타입(날짜/시간, 정수, 부동 소수점, 숫자)을 형식화된 문자열로 변환하고 형식화된 문자열을 특정 데이터 타입으로 변환하기 위한 강력한 도구 세트를 제공한다. 이 함수는 모두 공통적인 호출 규칙을 따르며, 첫 번째 인자는 형식화 할 값이고 두 번째 인자는 출력 또는 입력 형식을 정의하는 템플릿을 따른다.

- to_char(*timestamp, text*)

  이 함수는 *timestamp*를 문자열로 변환한 값을 반환한다.

```
SELECT to_char(current_timestamp, 'HH12:MI:SS');


Result:
 to_char
----------
 04:56:02
   (1 row)
```

- to_char(*interval, text*)

  이 함수는 *interval*을 문자열로 변환한 값을 반환한다.

```
SELECT to_char(interval '15h 2m 12s', 'HH24:MI:SS');


Result:
 to_char
----------
 15:02:12
   (1 row)
```

- to_char(*int*, *text*)

  이 함수는 *integer*를 문자열로 변환한 값을 반환한다.

```
SELECT to_char(125, '999');


Result:
 to_char
----------
      125
   (1 row)
```

- to_char(*double* precision, *text*)

  이 함수는 *real/double* precision을 문자열로 변환한 값을 반환한다.

```
SELECT to_char(125.8::real, '999D9');


Result:
 to_char
----------
    125.8
   (1 row)
```

- to_char(*numeric*, *text*)

  이 함수는 *numeric*을 문자열로 변환한 값을 반환한다.

```
SELECT to_char(-125.8, '999D99S');


Result:
 to_char
```

```
----------
  125.80-
   (1 row)
```

- to_date(*text*, *text*)

  to_date 함수는 문자열을 날짜로 변환한 값을 반환한다.

```
SELECT to_date('05 Dec 2000', 'DD Mon YYYY');

Result:
   to_date
------------
 2000-12-05
     (1 row)
```

- to_number(*text*, *text*)

  to_number 함수는 문자열을 *numeric*으로 변환한 값을 반환한다.

```
SELECT to_number('12,454.8-', '99G999D9S');

Result:
 to_number
-----------
  -12454.8
    (1 row)
```

- to_timestamp(*text*, *text*)

  to_timestamp 함수는 문자열을 timestamp로 변환한 값을 반환한다.

```
SELECT to_timestamp('05 Dec 2000', 'DD Mon YYYY');

Result:
      to_timestamp
------------------------
 2000-12-05 00:00:00+09
                  (1 row)
```

4.3.6 Date/Time functions

time 또는 timestamp 입력을 받는 아래에 설명된 모든 함수에는 두 가지 변종 (time with time zone 또는 timestamp with time zone과 time without time zone 또는 timestamp without time zone) 이 있다. 간결성을 위해 이러한 변형은 별도로 표시되지 않으며, 또한 + 와 * 연산자는 교환 할 수 있는 쌍 (예 : date + integer 및 integer + date) 으로 나타낸다. 그러한 각 쌍 중 오직 하나만 기술한다.

- age(*timestamp*, *timestamp*)

  age 함수는 인자끼리의 뺄셈을 통해 일뿐 아니라 년과 월을 사용하는 결과를 반환한다.

```
SELECT age(timestamp '2001-04-10', timestamp '1957-06-13');

Result:
          age
-------------------------
 43 years 9 mons 27 days
                  (1 row)
```

- age(*timestamp*)

  이 함수는 현재 날짜 (자정 기준) 에서 인자값을 뺀 값을 반환한다.

```
SELECT age(timestamp '1957-06-13');

Result:
          age
-------------------------
 60 years 3 mons 15 days
                  (1 row)
```

- clock_timestamp()

  clock_timestamp 함수는 현재 날짜와 시간 (명령문 실행 중 변경) 을 반환한다.

```
SELECT clock_timestamp();

Result:
        clock_timestamp
-------------------------------
```

```
 2017-09-28 17:47:31.208076+09
                                (1 row)
```

- current_date

  current_date 함수는 현재 날짜를 반환한다.

```
SELECT current_date;


Result:
    date
------------
 2017-09-28
     (1 row)
```

- current_time

  current_time 함수는 현재 시간을 반환한다.

```
SELECT current_time;


Result:
       timetz
--------------------
 17:53:23.972231+09
             (1 row)
```

- current_timestamp

  current_timestamp 함수는 현재 날짜와 시간을 반환한다.

```
SELECT current_timestamp;


Result:
              now
-------------------------------
 2017-09-28 18:01:43.890977+09
                        (1 row)
```

- date_part(*text, timestamp*)

  date_part 함수는 *text*에 명시된 하위필드값을 반환한다.

```
SELECT date_part('hour', timestamp '2001-02-16 20:38:40');

Result:
 date_part
-----------
        20
    (1 row)
```

- date_part(*text*, *interval*)

  date_part 함수는 *text*에 명시된 하위필드값을 반환한다.

```
SELECT date_part('month', interval '2 years 3 months');

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

- date_trunc(*text*, *timestamp*)

  date_trunc 함수는 지정된 정밀도로 잘라낸 값을 반환한다.

```
SELECT date_trunc('hour', timestamp '2001-02-16 20:38:40');

Result:
      date_trunc
---------------------
 2001-02-16 20:00:00
              (1 row)
```

- date_trunc(*text*, *interval*)

  date_trunc 함수는 지정된 정밀도로 잘라낸 값을 반환한다.

```
SELECT date_trunc('hour', interval '2 days 3 hours 40 minutes');

Result:
    date_trunc
```

```
------------------
 2 days 03:00:00
          (1 row)
```

- extract(field from timestamp)

  extract 함수는 명시한 필드를 추출하여 반환한다.

```
SELECT extract(hour FROM timestamp '2001-02-16 20:38:40');

Result:
 date_part
-----------
        20
    (1 row)
```

- extract(field from interval)

  extract 함수는 명시한 필드를 추출하여 반환한다.

```
SELECT extract(month FROM interval '2 years 3 months');

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

- isfinite(*date*)

  isfinite 함수는 입력된 인자가 유한한지를 테스트한 결과를 반환한다(+/- 무한 아님).

```
SELECT isfinite(date '2001-02-16');

Result:
 isfinite
----------
        t
   (1 row)
```

- isfinite(*timestamp*)

isfinite 함수는 입력된 인자가 유한한지를 테스트한 결과를 반환한다(+/- 무한 아님).

```
SELECT isfinite(timestamp '2001-02-16 21:28:30');


Result:
 isfinite
----------
       t
   (1 row)
```

- isfinite(*interval*)

  isfinite 함수는 입력된 인자가 유한한지를 테스트한 결과를 반환한다.

```
SELECT isfinite(interval '4 hours');


Result:
 isfinite
----------
       t
   (1 row)
```

- justify_days(*interval*)

  justify_days 함수는 간격을 조정하여 30일을 월로 표현하여 반환한다.

```
SELECT justify_days(interval '35 days');


Result:
 justify_days
--------------
 1 mon 5 days
       (1 row)
```

- justify_hours(*interval*)

  justify_hours 함수는 간격을 조정하여 24시간을 일로 표현하여 반환한다.

```
SELECT justify_hours(interval '27 hours');


Result:
```

```
 justify_hours
----------------
 1 day 03:00:00
        (1 row)
```

- justify_interval(*interval*)

  justify_interval 함수는 `justify_days`와 `justify_hours`를 이용하여 간격을 조정하고 추가로 기호를 수정한 값을 반환한다.

```
SELECT justify_interval(interval '1 mon -1 hour');

Result:
 justify_interval
------------------
 29 days 23:00:00
          (1 row)
```

- localtime

  localtime 함수는 현재시간을 반환한다.

```
SELECT localtime;

Result:
      time
----------------
 11:27:04.72722
        (1 row)
```

- localtimestamp

  localtimestamp 함수는 현재 날짜 및 시간(현재 트랜잭션 시작)을 반환한다.

```
SELECT localtimestamp;

Result:
          timestamp
----------------------------
 2017-09-29 11:29:52.230028
                    (1 row)
```

- make_date(*year* int, *month* int, *day* int)

  make_date 함수는 년/월/일 필드에서 날짜를 생성하여 반환한다.

```
SELECT make_date(2013, 7, 15);

Result:
 make_date
------------
 2013-07-15
     (1 row)
```

- make_interval(*years* int DEFAULT 0, *months* int DEFAULT 0, *weeks* int DEFAULT 0, *days* int DEFAULT 0, *hours* int DEFAULT 0, *mins* int DEFAULT 0, *secs* double precision DEFAULT 0.0)

  make_interval 함수는 년/월/주/일/시/분 및 초 필드에서 interval을 생성하여 반환한다.

```
SELECT make_interval(days => 10);

Result:
 make_interval
---------------
       10 days
        (1 row)
```

- make_time(*hour* int, *min* int, *sec* double precision)

  make_time 함수는 시/분/초 필드에서 시간을 생성하여 반환한다.

```
SELECT make_time(8, 15, 23.5);

Result:
 make_time
------------
 08:15:23.5
     (1 row)
```

- make_timestamp(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec* double precision)

  make_timestamp 함수는 년/월/일/시/분 및 초 필드에서 시간을 생성하여 반환한다.

```
SELECT make_timestamp(2013, 7, 15, 8, 15, 23.5);


Result:
    make_timestamp
-----------------------
 2013-07-15 08:15:23.5
                (1 row)
```

- make_timestamptz(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec* double precision, [*timezone* text])

  make_timestampz 함수는 년/월/일/시/분 및 초 필드에서 시간대가 있는 타임스탬프를 반환한다. 시간대가 지정되지 않으면 현재 시간대가 사용된다.

```
SELECT make_timestamptz(2013, 7, 15, 8, 15, 23.5);


Result:
      make_timestampz
--------------------------
 2013-07-15 08:15:23.5+01
                   (1 row)
```

- now()

  now 함수는 현재 날짜와 시간(현재 트랜잭션 시작)을 반환한다.

```
SELECT now();


Result:
              now
-------------------------------
 2017-10-11 16:09:51.154262+09
                        (1 row)
```

- statement_timestamp()

  statement_timestamp 함수는 현재 날짜와 시간을 반환한다.

```
SELECT statement_timestamp();


Result:
```

```
      statement_timestamp
-------------------------------
 2017-10-11 16:08:59.641426+09
                        (1 row)
```

- timeofday()

  timeofday 함수는 현재 날짜와 시간을 반환한다.

```
SELECT timeofday();


Result:
              timeofday
-------------------------------------
 Wed Oct 11 16:09:26.934061 2017 KST
                              (1 row)
```

- transaction_timestamp()

  transaction_timestamp 함수는 현재 날짜와 시간을 반환한다.

```
SELECT transaction_timestamp();


Result:
     transaction_timestamp
-------------------------------
 2017-10-11 16:10:21.530521+09
                        (1 row)
```

- to_timestamp(*double* precision)

  to_timestamp 함수는 Unix epoch(1970-01-01 00:00:00+00부터 초)를 timestamp로 변환하여 반환한다.

```
SELECT to_timestamp(1284352323);


Result:
      to_timestamp
------------------------
 2010-09-13 13:32:03+09
                 (1 row)
```

4.3.7 Enum Support functions

enum 타입은 enum 타입에의 특정 값을 하드 코딩하지 않고 보다 깔끔한 프로그래밍을 허용하는 몇 가지 함수가 있다.

함수설명에 나오는 예제를 실행하기 위해 아래와 같이 먼저 enum 타입을 생성한다.

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

- enum_first(*anyenum*)

  enum_first 함수는 입력된 enum 타입의 첫번째 값을 반환한다.

```
SELECT enum_first(null::rainbow);


Result:
 enum_first
------------
     red
     (1 row)
```

- enum_last(*anyenum*)

  enum_last 함수는 입력된 enum 타입의 마지막 값을 반환한다.

```
SELECT enum_last(null::rainbow);


Result:
 enum_last
-----------
  purple
    (1 row)
```

- enum_range(*anyenum*)

  enum_range 함수는 정렬 된 배열에서 입력 enum 타입의 모든 값을 반환한다.

```
SELECT enum_range(null::rainbow);


Result:
              enum_range
---------------------------------------
```

```
{red,orange,yellow,green,blue,purple}
                                (1 row)
```

- enum_range(*anyenum, anyenum*)

  이 함수는 지정된 2개의 enum값들 사이의 범위를 순차적인 배열로 반환한다. 값은 동일한 enum 타입의 값이어야 한다. 첫번째 매개 변수가 null의 경우, 결과는 enum 타입의 첫번째 값으로부터 시작된다. 두번째 매개 변수가 null이면 결과는 enum 타입의 마지막 값으로 끝난다.

```
SELECT enum_range('orange'::rainbow, 'green'::rainbow);


Result:
      enum_range
-----------------------
 {orange,yellow,green}
                (1 row)

SELECT enum_range(NULL, 'green'::rainbow);
Result:
        enum_range
---------------------------
 {red,orange,yellow,green}
                    (1 row)

SELECT enum_range('orange'::rainbow, NULL);
Result:
            enum_range
-----------------------------------
 {orange,yellow,green,blue,purple}
                            (1 row)
```

4.3.8 Geometric Functions

- area(*object*)

  area 함수는 영역을 반환한다.

```
SELECT area(box '((0,0),(1,1))');


Result:
 area
------
    1
(1 row)
```

- center(*object*)

  center 함수는 입력된 객체의 센터 좌표를 반환한다.

```
SELECT center(box '((0,0),(1,2))');


Result:
 center
--------
 (0.5,1)
 (1 row)
```

- diameter(*circle*)

  diameter 함수는 입력된 원의 지름을 반환한다.

```
SELECT diameter(circle '((0,0),2.0)');


Result:
 diameter
----------
        4
   (1 row)
```

- height(*box*)

  height 함수는 box의 높이를 반환한다.

```
SELECT height(box '((0,0),(1,1))');


Result:
 height
```

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

- isclosed(*path*)

  isclosed 함수는 입력된 path가 닫힌 경로인지에 대한 논리값을 반환한다.

```
SELECT isclosed(path '((0,0),(1,1),(2,0))');


Result:
 isclosed
----------
        t
   (1 row)
```

- isopen(*path*)

  isopen 함수는 입력된 path가 열린 경로인지에 대한 논리값을 반환한다.

```
SELECT isopen(path '[(0,0),(1,1),(2,0)]');


Result:
 isopen
--------
      t
 (1 row)
```

- length(*object*)

  length 함수는 경로의 길이를 반환한다.

```
SELECT length(path '((-1,0),(1,0))');


Result:
 length
--------
      4
 (1 row)
```

- npoints(*path*)

npoints 함수는 입력된 경로의 point 수를 반환한다.

```
SELECT npoints(path '[(0,0),(1,1),(2,0)]');


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

- npoints(*polygon*)

  이 함수는 다각형 점의 수를 반환한다.

```
SELECT npoints(polygon '((1,1),(0,0))');


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

- pclose(*path*)

  pclose 함수는 입력된 path를 닫힌 경로로 반환한다.

```
SELECT pclose(path '[(0,0),(1,1),(2,0)]');


Result:
       pclose
---------------------
 ((0,0),(1,1),(2,0))
              (1 row)
```

- popen(*path*)

  popen 함수는 입력된 path를 열린 경로로 반환한다.

```
SELECT popen(path '((0,0),(1,1),(2,0))');


Result:
```

```
        popen
---------------------
 [(0,0),(1,1),(2,0)]
             (1 row)
```

- radius(*circle*)

  radius 함수는 원의 반지름을 반환한다.

```
SELECT radius(circle '((0,0),2.0)');


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

- width(*box*)

  width 함수는 box의 가로 크기를 반환한다.

```
SELECT width(box '((0,0),(1,1))');


Result:
 width
-------
     1
(1 row)
```

- box(*circle*)

  box 함수는 입력된 원에 외접하는 box를 반환한다.

```
SELECT box(circle '((0,0),2.0)');


Result:
                                   box
-----------------------------------------------------------------------------
 (1.41421356237309,1.41421356237309),(-1.41421356237309,-1.41421356237309)
                                                                     (1 row)
```

- box(*point*)

  이 함수는 입력된 point를 중심으로 하는 넓이가 0인 box를 반환한다.

```
SELECT box(point '(0,0)');


Result:
     box
-------------
 (0,0),(0,0)
      (1 row)
```

- box(*point, point*)

  이 함수는 입력된 두 점을 꼭지점으로 하는 box를 반환한다.

```
SELECT box(point '(0,0)', point '(1,1)');


    Result:
     box
-------------
 (1,1),(0,0)
      (1 row)
```

- box(*polygon*)

  이 함수는 입력된 polygon에 외접하는 box를 반환한다.

```
SELECT box(polygon '((0,0),(1,1),(2,0))');


Result:
     box
-------------
 (2,1),(0,0)
      (1 row)
```

- bound_box(*box, box*)

  bound_box 함수는 입력된 두 box를 포함하는 최소 넓이의 box를 반환한다.

```
SELECT bound_box(box '((0,0),(1,1))', box '((3,3),(4,4))');


Result:
  bound_box
-------------
 (4,4),(0,0)
      (1 row)
```

- circle(*box*)

  circle 함수는 입력된 box의 외접원을 반환한다.

```
SELECT circle(box '((0,0),(1,1))');


Result:
            circle
-------------------------------
 <(0.5,0.5),0.707106781186548>
                        (1 row)
```

- circle(*point, double precision*)

  이 함수는 입력된 원의 중심 좌표와 반지름을 이용하여 생성된 circle을 반환한다.

```
SELECT circle(point '(0,0)', 2.0);


Result:
  circle
-----------
 <(0,0),2>
    (1 row)
```

- circle(*polygon*)

  이 함수는 입력된 좌표쌍의 평균값을 원의 중심으로, 그리고 이 점으로부터 입력된 좌표쌍까지의 거리 평균을 반지름으로 하는 원을 반환한다.

```
SELECT circle(polygon '((0,0),(1,1),(2,0))');


    Result:
```

```
                  circle
-------------------------------------------
 <(1,0.333333333333333),0.924950591148529>
                                    (1 row)
```

- line(*point, point*)

  line 함수는 선의 값을 반환한다.

```
SELECT line(point '(-1,0)', point '(1,0)');


Result:
   line
----------
 {0,-1,0}
  (1 row)
```

- lseg(*box*)

  lseg 함수는 box의 대각선을 선분 값으로 반환한다.

```
SELECT lseg(box '((-1,0),(1,0))');


Result:
      lseg
----------------
 [(1,0),(-1,0)]
        (1 row)
```

- lseg(*point, point*)

  이 함수는 입력된 두 점을 시작점과 끝점으로 하는 선분 값을 반환한다.

```
SELECT lseg(point '(-1,0)', point '(1,0)');


Result:
      lseg
----------------
 [(1,0),(-1,0)]
        (1 row)
```

- path(*polygon*)

  path 함수는 다각형 경로를 반환한다.

```
SELECT path(polygon '((0,0),(1,1),(2,0))');

Result:
        path
---------------------
 ((0,0),(1,1),(2,0))
              (1 row)
```

- point(*double precision, double precision*)

  point 함수는 점을 구성하는 값을 반환한다.

```
SELECT point(23.4, -44.5);

Result:
    point
--------------
 (23.4,-44.5)
       (1 row)
```

- point(*box*)

  이 함수는 box의 중심값을 반환한다.

```
SELECT point(box '((-1,0),(1,0))');

Result:
 point
-------
 (0,0)
(1 row)
```

- point(*circle*)

  이 함수는 원의 중심값을 반환한다.

```
SELECT point(circle '((0,0),2.0)');

Result:
 point
-------
 (0,0)
(1 row)
```

- point(*lseg*)

  이 함수는 선분의 중심값을 반환한다.

```
SELECT point(lseg '((-1,0),(1,0))');

Result:
 point
-------
 (0,0)
(1 row)
```

- point(*polygon*)

  이 함수는 다각형의 중심값을 반환한다.

```
SELECT point(polygon '((0,0),(1,1),(2,0))');

Result:
        point
-----------------------
 (1,0.333333333333333)
               (1 row)
```

- polygon(*box*)

  polygon 함수는 4개의 점을 가진 다각형으로 box값을 반환한다.

```
SELECT polygon(box '((0,0),(1,1))');

Result:
          polygon
```

```
---------------------------
 ((0,0),(0,1),(1,1),(1,0))
                   (1 row)
```

- polygon(*circle*)

  이 함수는 12개의 점을 가진 다각형으로 원의 값을 반환한다.

```
SELECT polygon(circle '((0,0),2.0)');


Result:
                                        polygon
-----------------------------------------------------------------------------------------
 ((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),
 (1,1.73205080756888),(1.73205080756888,1),(2,2.44929359829471e-16),
 (1.73205080756888,-0.999999999999999),(1,-1.73205080756888),(3.67394039744206e-16,-2),
 (-0.999999999999999,-1.73205080756888),(-1.73205080756888,-1))
                                                                                  (1 row)
```

- polygon(*npts, circle*)

  이 함수는 npts-point 다각형으로 원의 값을 반환한다.

```
SELECT polygon(12, circle '((0,0),2.0)');


                                        polygon
-----------------------------------------------------------------------------------------
 ((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),
 (1,1.73205080756888),(1.73205080756888,1),(2,2.44929359829471e-16),
 (1.73205080756888,-0.999999999999999),(1,-1.73205080756888),(3.67394039744206e-16,-2),
 (-0.999999999999999,-1.73205080756888),(-1.73205080756888,-1))
                                                                                  (1 row)
```

- polygon(*path*)

  이 함수는 입력된 path를 다각형으로 변환한 값을 반환한다.

```
SELECT polygon(path '((0,0),(1,1),(2,0))');


Result:
```

```
       polygon
---------------------
 ((0,0),(1,1),(2,0))
              (1 row)
```

4.3.9 Network Address Functions

- abbrev(*inet*)

  abbrev 함수는 텍스트 형식의 약어를 반환한다.

```
SELECT abbrev(inet '10.1.0.0/16');


Result:
   abbrev
-------------
 10.1.0.0/16
      (1 row)
```

- abbrev(*cidr*)

  이 함수는 텍스트 형식의 약어를 반환한다.

```
SELECT abbrev(cidr '10.1.0.0/16');


Result:
 abbrev
---------
 10.1/16
   (1 row)
```

- broadcast(*inet*)

  broadcast() network의 broadcase 주소를 반환한다.

```
SELECT broadcast('192.168.1.5/24');


Result:
    broadcast
------------------
```

```
192.168.1.255/24
           (1 row)
```

- family(*inet*)

  family 함수는 주소의 family 값(IPv4일 경우 4, IPv6일 경우 6)을 추출하여 반환한다.

```
SELECT  family('::1');


Result:
 family
--------
      6
 (1 row)
```

- host(*inet*)

  host 함수는 text형식의 IP 주소를 추출하여 반환한다.

```
SELECT  host('192.168.1.5/24');


Result:
    host
-------------
 192.168.1.5
      (1 row)
```

- hostmask(*inet*)

  hostmask 함수는 network의 host mask를 반환한다.

```
SELECT hostmask('192.168.23.20/30');


Result:
 hostmask
----------
  0.0.0.3
   (1 row)
```

- masklen(*inet*)

  masklen 함수는 netmask의 길이를 반환한다.

```
SELECT masklen('192.168.1.5/24');


Result:
 masklen
---------
      24
  (1 row)
```

- netmask(*inet*)

  netmask 함수는 network의 netmask를 반환한다.

```
SELECT netmask('192.168.1.5/24');


Result:
    netmask
---------------
 255.255.255.0
        (1 row)
```

- network(*inet*)

  network 함수는 주소의 network 부분을 추출하여 반환한다.

```
SELECT network('192.168.1.5/24');


Result:
    network
----------------
 192.168.1.0/24
         (1 row)
```

- set_masklen(*inet, int*)

  set_masklen 함수는 *inet* 값에 대한 netmask 길이 설정 값을 반환한다.

```
SELECT set_masklen('192.168.1.5/24', 16);


Result:
  set_masklen
```

```
-----------------
 192.168.1.5/16
         (1 row)
```

- set_masklen(*cidr, int*)

  set_masklen 함수는 *cidr* 값에 대한 netmask 길이 설정 값을 반환한다.

```
SELECT set_masklen('192.168.1.0/24'::cidr, 16);


Result:
  set_masklen
-----------------
 192.168.0.0/16
         (1 row)
```

- text(*inet*)

  text 함수는 IP address와 netmask 길이를 텍스트 값으로 반환한다.

```
SELECT text(inet '192.168.1.5');


Result:
      text
-----------------
 192.168.1.5/32
         (1 row)
```

- inet_same_family(*inet, inet*)

  inet_same_family 함수는 주소가 같은 family 값인지에 대한 논리값을 반환한다.

```
SELECT inet_same_family('192.168.1.5/24', '::1');


Result:
 inet_same_family
-------------------
        f
           (1 row)
```

- inet_merge(*inet, inet*)

inet_merge 함수는 입력된 네트워크 모두를 포함하는 가장 작은 네트워크 값을 반환한다.

```
SELECT inet_merge('192.168.1.5/24', '192.168.2.5/24');

Result:
   inet_merge
----------------
 192.168.0.0/22
        (1 row)
```

- trunc(*macaddr*)

  trunc 함수는 마지막 3 바이트를 0으로 설정하여 반환한다.

```
SELECT trunc(macaddr '12:34:56:78:90:ab');

Result:
       trunc
-------------------
 12:34:56:00:00:00
           (1 row)
```

4.3.10 Text Search Functions

- array_to_tsvector(text[])

  array_to_tsvector 함수는 어휘의 배열을 *tsvector*로 변환한다.

```
SELECT array_to_tsvector('{fat,cat,rat}'::text[]);

Result:
 array_to_tsvector
-------------------
 'cat' 'fat' 'rat'
           (1 row)
```

- get_current_ts_config()

  get_current_ts_config 함수는 텍스트를 검색하는 기본 구성값을 반환한다.

```
SELECT get_current_ts_config();


Result:
 get_current_ts_config
-----------------------
               english
                (1 row)
```

- length(tsvector)

  length 함수는 tsvector에 있는 어휘수를 반환한다.

```
SELECT length('fat:2,4 cat:3 rat:5A'::tsvector);


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

- numnode(tsquery)

  numnode 함수는 tsquery의 어휘수+연산자수를 반환한다.

```
SELECT numnode('(fat & rat) | cat'::tsquery);


Result:
 numnode
---------
       5
 (1 row)
```

- plainto_tsquery([*config* regconfig ,] *query* text)

  plainto_tsquery 함수는 구두점 (punctuation) 을 무시하고 tsquery를 반환한다.

```
SELECT plainto_tsquery('english', 'The Fat Rats');


Result:
 plainto_tsquery
```

```
------------------
  'fat' & 'rat'
           (1 row)
```

- phraseto_tsquery([*config* regconfig ,] *query* text)

  phraseto_tsquery 함수는 구두점 (punctuation) 을 무시하고 구문을 검색하는 tsquery를 반환한다.

```
SELECT phraseto_tsquery('english', 'The Fat Rats');

Result:
 phraseto_tsquery
-------------------
 'fat' <-> 'rat'
           (1 row)
```

- querytree(*query* tsquery)

  querytree 함수는 *tsquery*의 색인이 가능한 부분을 반환한다.

```
SELECT querytree('foo & ! bar'::tsquery);

Result:
 querytree
-----------
   'foo'
    (1 row)
```

- setweight(*vector* tsvector, *weight* ``char'')

  setweight 함수는 *vector*의 각 원소에 *weight*를 할당하여 반환한다.

```
SELECT setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A');

Result:
           setweight
-------------------------------
 'cat':3A 'fat':2A,4A 'rat':5A
                         (1 row)
```

- setweight(*vector* tsvector, *weight* ``char'', *lexemes* text[])

이 함수는 *lexemes*에 나열된 *vector*에 *weight*를 할당하여 반환한다.

```
SELECT setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A', '{cat,rat}');

Result:
          setweight
------------------------------
 'cat':3A 'fat':2,4 'rat':5A
                      (1 row)
```

- strip(tsvector)

  strip 함수는 tsvector에서 위치 및 weight를 제거하여 반환한다.

```
SELECT strip('fat:2,4 cat:3 rat:5A'::tsvector);

Result:
       strip
-------------------
 'cat' 'fat' 'rat'
            (1 row)
```

- to_tsquery([*config* regconfig ,] *query* text)

  to_tsquery 함수는 단어를 표준화하고 tsquery로 변환하여 반환한다.

```
SELECT to_tsquery('english', 'The & Fat & Rats');

Result:
  to_tsquery
---------------
 'fat' & 'rat'
        (1 row)
```

- to_tsvector([*config* regconfig ,] *document* text)

  to_tsvector 함수는 document 값을 tsvector로 줄여서 반환한다.

```
SELECT to_tsvector('english', 'The Fat Rats');

Result:
```

```
   to_tsvector
-----------------
 'fat':2 'rat':3
          (1 row)
```

- ts_delete(*vector* tsvector, *lexeme* text)

  ts_delete 함수는 *vector*에서 *lexeme*에 할당된 값을 제거하고 반환한다.

```
SELECT ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat');

Result:
    ts_delete
------------------
 'cat':3 'rat':5A
           (1 row)
```

- ts_delete(*vector* tsvector, *lexemes* text[])

  이 함수는 *vector*에서 *lexeme*에 할당된 값들을 제거하고 반환한다.

```
SELECT ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat']);

Result:
 ts_delete
-----------
 'cat':3
    (1 row)
```

- ts_filter(*vector* tsvector, *weights* ``char''[])

  ts_filter 함수는 *vector*에서 *weights*에 할당된 원소만 선택하여 반환한다.

```
SELECT ts_filter('fat:2,4 cat:3b rat:5A'::tsvector, '{a,b}');

Result:
     ts_filter
-------------------
 'cat':3B 'rat':5A
            (1 row)
```

- ts_headline([*config* regconfig,] *document* text, *query* tsquery [, *options* text])

  ts_headline 함수는 일치하는 쿼리를 표시하여 반환한다.

```
SELECT ts_headline('x y z', 'z'::tsquery);


Result:
 ts_headline
--------------
 x y <b>z</b>
       (1 row)
```

- ts_rank([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])

  ts_rank 함수는 query에 대한 문서 순서를 반환한다.

```
SELECT ts_rank(to_tsvector('This is an example of document'), to_tsquery('example'));


Result:
  ts_rank
-----------
 0.0607927
    (1 row)
```

- ts_rank_cd([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])

  ts_rank_cd 함수는 커버 밀도를 사용하여 문서 순서를 반환한다.

```
SELECT ts_rank_cd(to_tsvector('This is an example of document'), to_tsquery('example'));


Result:
 ts_rank_cd
------------
        0.1
     (1 row)
```

- ts_rewrite(*query* tsquery, *target* tsquery, *substitute* tsquery)

  ts_rewrite 함수는 query에서 *target*값을 *substitute*값으로 교체하여 반환한다.

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery);

Result:
       ts_rewrite
-------------------------
 'b' & ( 'foo' | 'bar' )
                  (1 row)
```

- ts_rewrite(*query* tsquery, *select* text)

  이 함수는 SELECT 결과의 첫번째 컬럼 값을 SELECT 결과의 두번째 컬럼 값으로 교체하여 반환한다.

```
create table aliases (t tsquery primary key, s tsquery);
insert into aliases values ('a', 'foo|bar');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');

Result:
       ts_rewrite
-------------------------
 'b' & ( 'bar' | 'foo' )
                  (1 row)
```

- tsquery_phrase(*query1* tsquery, *query2* tsquery)

  tsquery_phrase 함수는 *query1*을 검색 한 다음 *query2*를 검색하는 쿼리를 만들어 반환한다 (<-> 연산자 와 동일).

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'));

Result:
 tsquery_phrase
-----------------
 'fat' <-> 'cat'
          (1 row)
```

- tsquery_phrase(*query1* tsquery, *query2* tsquery, *distance* integer)

  이 함수는 distance 값의 거리로 *query1*을 검색 한 다음 *query2*를 검색하는 쿼리를 만들어 반환한다.

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);

Result:
  tsquery_phrase
------------------
 'fat' <10> 'cat'
           (1 row)
```

- tsvector_to_array(tsvector)

  tsvector_to_array 함수는 `tsvector`를 어휘의 배열로 변환하여 반환한다.

```
SELECT tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector);

Result:
 tsvector_to_array
-------------------
   {cat,fat,rat}
            (1 row)
```

- tsvector_update_trigger()

  ttsvector_update_trigger 함수는 자동으로 `tsvector` 컬럼을 업데이트 하기위한 기능을 제공한다.

```
CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE ON messages
FOR EACH ROW EXECUTE PROCEDURE
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES ('title here', 'the body text is here');

SELECT * FROM messages;

Result:
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1
                                                             (1 row)
```

- tsvector_update_trigger_column()

tsvector_update_trigger_column 함수는 자동으로 `tsvector` 컬럼을 업데이트 하기위한 기능을 제공한다.

```
CREATE TRIGGER ... tsvector_update_trigger_column(tsv, configcol, title, body);
```

- unnest(tsvector, OUT *lexeme* text, OUT *positions* smallint[], OUT *weights* text)

  unnest 함수는 rows의 집합으로 tsvector를 확장하여 반환한다.

```
SELECT unnest('fat:2,4 cat:3 rat:5A'::tsvector);


Result:
        unnest
-----------------------
 (cat,{3},{D})
 (fat,"{2,4}","{D,D}")
 (rat,{5},{A})
                (3 row)
```

- ts_debug([*config* regconfig,] *document* text, OUT *alias* text, OUT *description* text, OUT *token* text, OUT *dictionaries* regdictionary[], OUT *dictionary* regdictionary, OUT *lexemes* text[])

  ts_debug 함수는 구성을 테스트한다.

```
SELECT ts_debug('english', 'The Brightest supernovaes');


Result:
                                     ts_debug
-----------------------------------------------------------------------------------
 (asciiword,"Word, all ASCII",The,{english_stem},english_stem,{})
 (blank,"Space symbols"," ",{},,)
 (asciiword,"Word, all ASCII",Brightest,{english_stem},english_stem,{brightest})
 (blank,"Space symbols"," ",{},,)
 (asciiword,"Word, all ASCII",supernovaes,{english_stem},english_stem,{supernova})
                                                                           (5 row)
```

- ts_lexize(*dict* regdictionary, *token* text)

  ts_lexize 함수는 dictionary를 테스트한다.

```
SELECT ts_lexize('english_stem', 'stars');

Result:
 ts_lexize
-----------
  {star}
   (1 row)
```

- ts_parse(*parser_name* text, *document* text, OUT *tokid* integer, OUT *token* text)

  ts_parse 함수는 text로 parser를 테스트한다.

```
SELECT ts_parse('default', 'foo - bar');

Result:
 ts_parse
-----------
 (1,foo)
 (12," ")
 (12,"- ")
 (1,bar)
   (4 row)
```

- ts_parse(*parser_oid* oid, *document* text, OUT *tokid* integer, OUT *token* text)

  이 함수는 oid로 parser를 테스트한다.

```
SELECT ts_parse(3722, 'foo - bar');

Result:
 ts_parse
-----------
 (1,foo)
 (12," ")
 (12,"- ")
 (1,bar)
   (4 row)
```

- ts_token_type(*parser_name* text, OUT *tokid* integer, OUT *alias* text, OUT *description* text)

ts_token_type 함수는 text로 parser에 정의된 토큰 타입을 반환한다.

```
SELECT ts_token_type('default');


Result:
         ts_token_type
---------------------------------
 (1,asciiword,"Word, all ASCII")
 ...
                         (23 row)
```

- ts_token_type(*parser_oid* oid, OUT *tokid* integer, OUT *alias* text, OUT *description* text)

  이 함수는 oid로 parser에 정의된 토큰 타입을 반환한다.

```
SELECT ts_token_type(3722);


Result:
         ts_token_type
---------------------------------
 (1,asciiword,"Word, all ASCII")
 ...
                         (23 row)
```

- ts_stat(*sqlquery* text, [*weights* text,] OUT *word* text, OUT *ndoc* integer, OUT *nentry* integer)

  ts_stat 함수는 `tsvector` 컬럼의 통계를 반환한다.

```
SELECT ts_stat('SELECT vector FROM apod');


Result:
  ts_stat
------------
(foo,10,15)
...
     (4 row)
```

4.3.11 JSON Functions

- to_json(anyelement), to_jsonb(anyelement)

to_json(), to_jsonb 함수는 값을 json 또는 jsonb로 리턴하며, 배열과 복합체는 배열과 객체로 변환한다. 그렇지 않으면 json으로 형 변환이있는 경우 형 변환 함수를 사용하여 변환이 수행되거나 스칼라 값이 생성된다. number, Boolean, null이 아닌 모든 스칼라 유형의 경우 유효한 json 또는 jsonb 값 방식으로 텍스트 표현이 사용된다.

```
SELECT to_json('Fred said "Hi."'::text);

Result:
       to_json
---------------------
 "Fred said \"Hi.\""
              (1 row)
```

- array_to_json(anyarray [, pretty_bool])

  array_to_json 함수는 배열을 JSON 배열로 반환한다. pretty_bool이 true이면 배열의 차원 -1 만큼의 원소마다 줄바꿈이 추가된다.

```
SELECT array_to_json('{{1,5},{99,100}}'::int[]);

Result:
  array_to_json
------------------
 [[1,5],[99,100]]
           (1 row)

SELECT array_to_json('{{1,5},{99,100}}'::int[], true);

Result:
 array_to_json
---------------
 [[1,5],      +
  [99,100]]
         (1 row)
```

- row_to_json(record [, pretty_bool])

  row_to_json 함수는 행을 JSON 객체로 반환한다. pretty_bool이 true이면 배열의 차원 -1 만큼의 원소마다 줄바꿈이 추가된다.

```
SELECT row_to_json(row(1,'foo'));


Result:
     row_to_json
---------------------
 {"f1":1,"f2":"foo"}
              (1 row)
```

- json_build_array(VARIADIC ``any''), jsonb_build_array(VARIADIC ``any'')

  json_build_array 함수는 가변 인자 목록에서 혼합 가능한 형식의 JSON 배열을 만든다.

```
SELECT json_build_array(1,2,'3',4,5);


Result:
 json_build_array
-------------------
 [1, 2, "3", 4, 5]
            (1 row)
```

- json_build_object(VARIADIC ``any''), jsonb_build_object(VARIADIC ``any'')

  json_build_object 함수는 가변 인자 목록에서 JSON 객체를 만든다. 관례 상, 인자 목록은 키와 값이 교대로 구성된다.

```
SELECT json_build_object('foo',1,'bar',2);


Result:
   json_build_object
------------------------
 {"foo" : 1, "bar" : 2}
                (1 row)
```

- json_object(text[]), jsonb_object(text[])

  json_object 함수는 텍스트 배열에서 JSON 객체를 만든다. 배열은 짝수 개의 멤버가있는 정확히 하나의 차원을 가져야하며, 이 경우 key/value 쌍이 교대로 사용되거나 각 내부 배열에 key/value 쌍으로 사용되는 원소가 두 개만있는 두 차원이 사용된다.

```
SELECT json_object('{a, 1, b, "def", c, 3.5}');
SELECT json_object('{{a, 1},{b, "def"},{c, 3.5}}');

Result:
              json_object
---------------------------------------
 {"a" : "1", "b" : "def", "c" : "3.5"}
                                (1 row)
```

- json_object(keys text[], values text[]), jsonb_object(keys text[], values text[])

  json_object 함수는 `json_object` 형식에서 두 개의 개별 배열에서 키와 값을 쌍으로 가져온다. 다른 모든 측면에서 이것은 하나의 인자 형식과 동일하다.

```
SELECT json_object('{a, b}', '{1,2}');

Result:
      json_object
------------------------
 {"a" : "1", "b" : "2"}
                 (1 row)
```

- json_array_length(json), jsonb_array_length(jsonb)

  json_array_length 함수는 가장 바깥 쪽 JSON 배열에있는 원소의 수를 반환한다.

```
SELECT json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]');

Result:
 json_array_length
-------------------
                 5
            (1 row)
```

- json_each(json), jsonb_each(jsonb)

  json_each 함수는 가장 바깥 쪽 JSON 객체를 key/value 쌍으로 확장한다.

```
SELECT * FROM json_each('{"a":"foo", "b":"bar"}');


Result:
 key | value
-----+-------
 a   | "foo"
 b   | "bar"
      (2 row)
```

- json_each_text(json), jsonb_each_text(jsonb)

  json_each_text 함수는 가장 바깥 쪽 JSON 객체를 key/value 쌍으로 확장하고, 반환된 값은 `text` type이 된다.

```
SELECT * FROM json_each_text('{"a":"foo", "b":"bar"}');


Result:
 key | value
-----+-------
 a   | foo
 b   | bar
      (2 row)
```

- json_extract_path(from_json json, VARIADIC path_elems text[]), jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])

  json_extract_path 함수는 `path_elems`가 가리키는 JSON 값을 반환한다. (#> 연산자와 같다.)

```
SELECT * FROM json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4');


Result:
  json_extract_path
----------------------
 {"f5":99,"f6":"foo"}
                 (1 row)
```

- json_extract_path_text(from_json json, VARIADIC path_elems text[]), jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])

  json_extract_path_text 함수는 `path_elems`가 가리키는 JSON 값을 `text`로 반환한다. (#>> 연산자와 같다.)

```
SELECT json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4', 'f6');

Result:
 json_extract_path_text
------------------------
        foo
               (1 row)
```

- json_object_keys(json), jsonb_object_keys(jsonb)

  json_object_keys 함수는 json 객체의 가장 바깥 쪽에 있는 키 세트를 반환한다.

```
SELECT json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}');

Result:
 json_object_keys
------------------
      f1
      f2
          (2 row)
```

- json_populate_record(base anyelement, from_json json), jsonb_populate_record(base anyelement, from_json jsonb)

  json_populate_record 함수는 `from_json` 오브젝트를 `base`로 정의된 레코드 타입과 일치하는 행으로 확장하여 반환한다.

```
CREATE TABLE myrowtype (a int, b int);

SELECT * FROM json_populate_record(null::myrowtype, '{"a":1,"b":2}');

Result:
 a | b
---+---
 1 | 2
(1 row)
```

- json_populate_recordset(base anyelement, from_json json), jsonb_populate_recordset(base anyelement, from_json jsonb)

json_populate_recordset 함수는 from_json 오브젝트의 바깥쪽 배열을 base로 정의된 레코드 타입과 일치하는 열 세트로 확장하여 반환한다.

```
SELECT * FROM json_populate_recordset(null::myrowtype, '[{"a":1,"b":2},{"a":3,"b":4}]');

Result:
 a | b
---+---
 1 | 2
 3 | 4
(2 row)
```

- json_array_elements(json), jsonb_array_elements(jsonb)

  json_array_elements 함수는 JSON 배열을 JSON 값의 세트로 확장하여 반환한다.

```
SELECT * FROM json_array_elements('[1,true, [2,false]]');

Result:
   value
-----------
     1
   true
[2,false]]
  (3 row)
```

- json_array_elements_text(json), jsonb_array_elements_text(jsonb)

  json_array_elements_text 함수는 JSON 배열을 text값의 세트로 확장하여 반환한다.

```
SELECT * FROM json_array_elements_text('["foo", "bar"]');

Result:
 value
-------
  foo
  bar
(2 row)
```

- json_typeof(json), jsonb_typeof(jsonb)

json_typeof 함수는 가장 바깥 쪽 JSON 값의 타입을 반환한다. 변환 가능한 유형은 object , array , string , number , boolean 및 null 이다.

```
SELECT json_typeof('-123.4');


Result:
 json_typeof
-------------
    number
      (1 row)
```

- json_to_record(json), jsonb_to_record(jsonb)

  json_to_record 함수는 JSON 객체에서 임의의 레코드를 만든다. record를 반환하는 모든 함수와 마찬가지로 호출자는 AS절이 있는 레코드의 구조를 명시 적으로 정의해야한다.

```
SELECT * FROM json_to_record('{"a":1,"b":[1,2,3],"c":"bar"}') as x(a int, b text, d text);


Result:
 a |     b     | d
---+-----------+---
 1 | [1,2,3]  |
          (1 row)
```

- json_to_recordset(json), jsonb_to_recordset(jsonb)

  json_to_recordset 함수는 JSON 배열의 객체에서 임의의 레코드 세트를 만든다. record를 반환하는 모든 함수와 마찬가지로 호출자는 AS 절이 있는 레코드의 구조를 명시 적으로 정의해야 한다.

```
SELECT * FROM json_to_recordset('[{"a":1,"b":"foo"},{"a":"2","c":"bar"}]')
as x(a int, b text);


Result:
 a |  b
---+-----
 1 | foo
 2 |
 (2 row)
```

- json_strip_nulls(from_json json), jsonb_strip_nulls(from_json jsonb)

json_strip_nulls 함수는 from_json에서 null 값이 없는 모든 오브젝트 필드를 리턴한다. null 값은 변경되지 않는다.

```
SELECT json_strip_nulls('[{"f1":1,"f2":null},2,null,3]');

Result:
  json_strip_nulls
---------------------
 [{"f1":1},2,null,3]
              (1 row)
```

- jsonb_set(target jsonb, path text[], new_value jsonb[, create_missing boolean])

  jsonb_set 함수는 target의 path에 해당하는 부분을 new_value로 대체되거나, create_missing 값이 true(기본값은 true)이면서 path인자가 없는 경우new_value가 추가 된다. 경로 기반 연산자와 마찬가지로,path에 나타나는 음의 정수는 JSON 배열의 끝에서부터 계산됩니다.

```
SELECT jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', false);

Result:
                  jsonb_set
---------------------------------------------
 [{"f1": [2, 3, 4], "f2": null}, 2, null, 3]
                                       (1 row)

SELECT jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}','[2,3,4]');

Result:
                  jsonb_set
---------------------------------------------
 [{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]
                                       (1 row)
```

- jsonb_insert(target jsonb, path text[], new_value jsonb, [insert_after boolean])

  jsonb_insert 함수는 new_value가 삽입된 target을 반환한다.path에 의해 지정된 target섹션이 JSONB 배열 내에 있으면, insert_after가 true(기본값은false) 일때 new_value가 타겟 뒤에 입력되고 아니라면 앞에 입력된다. path에 의해 지정된 target섹션이 JSONB 객체 내에 있으면, target이 존재하지 않는 경우에만 new_value가 입력된다. 경로 기반 연산자와 마찬가지로,path에 나타나는 음의 정수는 JSON 배열의 끝에서

부터 계산됩니다.

```
SELECT jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"');

Result:
         jsonb_insert
-------------------------------
 {"a": [0, "new_value", 1, 2]}
                         (1 row)

SELECT jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true);

Result:
         jsonb_insert
-------------------------------
 {"a": [0, 1, "new_value", 2]}
                         (1 row)
```

- jsonb_pretty(from_json jsonb)

  jsonb_pretty 함수는 `from_json`을 들여 쓰기 된 JSON 텍스트로 반환한다.

```
SELECT jsonb_pretty('[{"f1":1,"f2":null},2,null,3]');

Result:
    jsonb_pretty
--------------------
 [                 +
     {             +
         "f1": 1,  +
         "f2": null+
     },            +
     2,            +
     null,         +
     3             +
 ]
            (1 row)
```

4.3.12 Sequence Manipulation Functions

Sequence에서 사용하는 함수에 대한 설명이다. CREATE SEQUENCE로 Sequence를 생성 할 수 있다.

```
CREATE SEQUENCE serial increment by 1 start 101;
```

- nextval(regclass)

  nextval 함수는 시퀀스의 다음 값을 반환한다.

```
SELECT nextval('serial');


Result:
 nextval
---------
     101
  (1 row)
```

- currval(regclass)

  currval 함수는 지정된 시퀀스의 `nextval`로 지정된 가장 최근 값을 반환한다.

```
SELECT currval('serial');


Result:
 currval
---------
     101
  (1 row)
```

- lastval()

  lastval 함수는 어떤 시퀀스의 `nextval`로 지정된 가장 최근 값을 반환한다.

```
SELECT lastval();


Result:
 lastval
---------
     101
  (1 row)
```

- setval(regclass, bigint)

  setval 함수는 시퀀스의 현재값을 지정한다.

```
SELECT setval('serial', 101);

Result:
 setval
--------
    101
 (1 row)
```

- setval(regclass, bigint, boolean)

  이 함수는 지정된 시퀀스의 시퀀스의 현재값과 `is_called` flag를 지정한다.

```
-- true
SELECT setval('serial', 101, true);

Result:
 setval
--------
    101
 (1 row)

SELECT nextval('serial');

Result:
 nextval
---------
     102
 (1 row)

-- false
SELECT setval('serial', 101, false);

Result:
 setval
```

```
---------
    101
 (1 row)


SELECT nextval('serial');


Result:
 nextval
---------
     101
 (1 row)
```

4.3.13 Array Functions

- array_append(anyarray, anyelement)

  array_append 함수는 배열의 끝에 `anyelement`를 추가한다.

```
SELECT array_append(ARRAY[1,2], 3);


Result:
 array_append
--------------
   {1,2,3}
       (1 row)
```

- array_cat(anyarray, anyarray)

  array_cat 함수는 두 배열을 합친 결과를 반환한다.

```
SELECT array_cat(ARRAY[1,2,3], ARRAY[4,5]);


Result:
  array_cat
-------------
 {1,2,3,4,5}
      (1 row)
```

- array_ndims(anyarray)

array_ndims 함수는 배열의 차원을 반환한다.

```
SELECT array_ndims(ARRAY[[1,2,3], [4,5,6]]);


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

- array_dims(anyarray)

  array_dims 함수는 배열의 차수에 대한 텍스트 표현을 반환합니다.

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


Result:
 array_dims
------------
 [1:2][1:3]
     (1 row)
```

- array_fill(anyelement, int[], [, int[]])

  array_fill 함수는 선택적으로 제공된 값과 차원을 사용하여 초기화 된 배열을 반환한다.

```
SELECT array_fill(7, ARRAY[3], ARRAY[2]);


Result:
  array_fill
---------------
 [2:4]={7,7,7}
        (1 row)
```

- array_length(anyarray, int)

  array_length 함수는 요청 된 배열 차원의 길이를 반환한다.

```
SELECT array_length(array[1,2,3], 1);


Result:
```

```
 array_length
--------------
            3
       (1 row)
```

- array_lower(anyarray, int)

  array_lower 함수는 요청한 배열 차원의 최소값을 반환한다.

```
   SELECT array_lower('[0:2]={1,2,3}'::int[], 1);


   Result:
array_lower
   -------------
         0
     (1 row)
```

- array_position(anyarray, anyelement [, int])

  array_position 함수는 세 번째 인자에 지정된 원소부터, 또는 첫 번째 원소에서 시작하여 배열에서 두 번째 인자가 처음으로 발견되는 항목의 인덱스를 반환한다(배열은 1차원이어야한다).

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


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

- array_positions(anyarray, anyelement)

  array_positions 함수는 첫번째 인자인 배열에서 두번째 인자에 해당하는 배열의 인덱스를 반환한다(배열은 1차원이어야 한다).

```
SELECT array_positions(ARRAY['A','A','B','A'], 'A');


Result:
 array_positions
-----------------
```

```
     {1,2,4}
          (1 row)
```

- array_prepend(anyelement, anyarray)

  array_prepend 함수는 배열의 시작부분에 `anyelement`를 추가한다.

```
SELECT array_prepend(1, ARRAY[2,3]);


Result:
 array_prepend
---------------
    {1,2,3}
        (1 row)
```

- array_remove(anyarray, anyelement)

  array_remove 함수는 입력된 값과 같은 원소를 배열에서 모두 제거한다.

```
SELECT array_remove(ARRAY[1,2,3,2], 2);


Result:
 array_remove
--------------
     {1,3}
       (1 row)
```

- array_replace(anyarray, anyelement, anyelement)

  array_replace 함수는 지정된 값과 동일한 배열의 모든 원소를 새 값으로 대체하여 반환한다.

```
SELECT array_replace(ARRAY[1,2,5,4], 5, 3);


Result:
 array_replace
---------------
   {1,2,3,4}
        (1 row)
```

- array_to_string(anyarray, text [, text])

array_to_string 함수는 지정된 구분 기호와 null 문자열(필요시 지정가능)을 사용하여 배열 원소를 연결한 text를 반환한다.

```
SELECT array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*');


Result:
 array_to_string
-----------------
    1,2,3,*,5
          (1 row)
```

- array_upper(anyarray, int)

  array_upper 함수는 지정한 배열 차원에서의 상한값을 반환한다.

```
SELECT array_upper(ARRAY[1,8,3,7], 1);


Result:
 array_upper
-------------
           4
      (1 row)
```

- cardinality(anyarray)

  cardinality 함수는 배열의 총 원소 수를 반환하거나 배열이 비어 있으면 0을 반환한다.

```
SELECT cardinality(ARRAY[[1,2],[3,4]]);


Result:
 cardinality
-------------
           4
      (1 row)
```

- string_to_array(text, text [, text])

  string_to_array 함수는 제공된 구분 기호와 null 문자열(필요시 지정가능)을 사용하여 문자열을 배열 원소로 나눈다.

```
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'yy');

Result:
 string_to_array
-----------------
   {xx,NULL,zz}
         (1 row)
```

- unnest(anyarray)

  unnest 함수는 배열을 행의 집합으로 펼친다.

```
SELECT unnest(ARRAY[1,2]);

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

- unnest(anyarray, anyarray [, …])

  이 함수는 여러 배열(서로 다른 type도 가능)을 행의 집합으로 확장한다. 이것은 FROM 절에서만 허용된다.

```
SELECT * FROM unnest(ARRAY[1,2],ARRAY['foo','bar','baz']);

Result:
 unnest | unnest
--------+--------
      1 | foo
      2 | bar
        | baz

         (1 row)
```

4.3.14 Range Functions and Operators

- lower(anyrange)

lower 함수는 입력된 숫자범위의 하한 값을 반환한다.

```
SELECT * FROM lower(numrange(1.1,2.2));

Result:

 lower
-------
   1.1
 (1 row)
```

- upper(anyrange)

  upper 함수는 입력된 숫자범위의 상한 값을 반환한다.

```
SELECT * FROM upper(numrange(1.1,2.2));

Result:

 upper
-------
   2.2
 (1 row)
```

- isempty(anyrange)

  isempty 함수는 입력된 숫자범위가 빈 값인지에 대한 논리값을 반환한다.

```
SELECT * FROM isempty(numrange(1.1,2.2));

Result:

 isempty
---------
 f
 (1 row)
```

- lower_inc(anyrange)

  lower_inc 함수는 입력된 숫자범위의 하한 값이 존재하는지에 대한 논리값을 반환한다.

```
SELECT * FROM lower_inc(numrange(1.1,2.2));


Result:


 lower_inc
-----------
 t
 (1 row)
```

- upper_inc(anyrange)

  upper_inc 함수는 입력된 숫자범위의 상한 값이 존재하는지에 대한 논리값을 반환한다.

```
SELECT * FROM upper_inc(numrange(1.1,2.2));


Result:


 lower_inc
-----------
 f
 (1 row)
```

- lower_inf(anyrange)

  lower_inf 함수는 입력된 숫자범위의 하한값이 무한(infinite) 인지에 대한 논리값을 반환한다.

```
SELECT * FROM lower_inf('(,)'::daterange);


Result:


 lower_inf
-----------
 t
 (1 row)
```

- upper_inf(anyrange)

  upper_inf 함수는 입력된 숫자범위의 상한값이 무한(infinite) 인지에 대한 논리값을 반환한다.

```
SELECT * FROM upper_inf('(,)'::daterange);

Result:

 upper_inf
-----------
 t
(1 row)
```

- range_merge(anyrange, anyrange)

  range_merge 함수는 입력된 두 숫자 범위를 포함하는 최소의 숫자범위를 반환한다.

```
SELECT * FROM range_merge('[1,2)'::int4range, '[3,4)'::int4range);

Result:

 range_merge
-------------
 [1,4)
(1 row)
```

4.3.15 Aggregate Functions

- array_agg(expression)

  array_agg 함수는 null을 포함한 입력값들을 array로 반환한다.

  인자 타입 : `any non-array type`

  반환 타입 : `array of the argument type`

- array_agg(expression)

  array_agg 함수는 입력되는 array들을 1차원 증가된 array에 연결하여 반환한다(주의 : 입력된 array는 동일한 차원을 갖으며, null 또는 empty이면 안된다).

  인자 타입 : `any array type`

  반환 타입 : `same as argument data type`

- avg(expression)

  avg 함수는 모든 인자의 산술평균값을 반환한다.

  인자 타입 : `smallint, int, bigint, real, double precision, numeric, or interval`

반환 타입 : numeric for any integer-type argument, double precision for a floating-point argument, otherwise the same as the argument data type

- bit_and(expression)

  bit_and 함수는 null이 아닌 모든 입력값의 AND 비트 연산값을 반환한다. (연산값이 존재하지 않을 경우엔 null을 반환한다)

  인자 타입 : smallint, int, bigint, or bit

  반환 타입 : same as argument data type

- bit_or(expression)

  bit_or 함수는 null이 아닌 모든 입력값의 OR 비트 연산값을 반환한다. (연산값이 존재하지 않을 경우엔 null을 반환한다)

  인자 타입 : smallint, int, bigint, or bit

  반환 타입 : same as argument data type

- bool_and(expression)

  bool_and 함수는 모든 입력값이 true인 경우엔 true, 그 외에는 false를 반환한다.

  인자 타입 : bool

  반환 타입 : bool

- bool_or(expression)

  bool_or 함수는 입력값 중 하나라도 true값이 있으면 true, 그 외에는 false를 반환한다.

  인자 타입 : bool

  반환 타입 : bool

- count(anything)

  count 함수는 입력된 행의 개수를 반환한다.

  인자 타입 : any

  반환 타입 : bigint

- count(expression)

  count 함수는 입력된 값들 중 null이 아닌 행의 개수를 반환한다.

  인자 타입 : any

  반환 타입 : bigint

- every(expression)

  every 함수는 bool_and 연산자와 동일한 연산을 수행한다.

  인자 타입 : bool

  반환 타입 : bool

- json_agg(expression)

  json_agg 함수는 입력된 값들을 모아 JSON array로 반환한다.

  인자 타입 : any

  반환 타입 : json

- jsonb_agg(expression)

  jsonb_agg 함수는 입력된 값들을 모아 JSON array로 반환한다.

  인자 타입 : any

  반환 타입 : jsonb

- json_object_agg(name, value)

  json_object_agg 함수는 입력된 이름/값 쌍을 JSON 객체로 반환한다.

  인자 타입 : (any, any)

  반환 타입 : json

- jsonb_object_agg(name, value)

  jsonb_object_agg 함수는 입력된 이름/값 쌍을 JSON 객체로 반환한다.

  인자 타입 : (any, any)

  반환 타입 : jsonb

- max(expression)

  max 함수는 모든 입력값 중 최대값을 반환한다.

  인자 타입 : any numeric, string, date/time, network, or enum type, or arrays of these types

  반환 타입 : same as argument type

- min(expression)

  min 함수는 모든 입력값 중 최소값을 반환한다.

  인자 타입 : any numeric, string, date/time, network, or enum type, or arrays of these types

  반환 타입 : same as argument type

- string_agg(expression, delimiter)

  string_agg 함수는 입력된 값을 delimiter로 구분해 연결한 하나의 문자열로 반환한다.

  인자 타입 : (text, text) or (bytea, bytea)

  반환 타입 : same as argument types

- sum(expression)

  sum 함수는 모든 입력 값의 합을 반환한다.

  인자 타입 : smallint, int, bigint, real, double precision, numeric, interval, or money

반환 타입 : `bigint for smallint or int arguments, numeric for bigint arguments, otherwise the same as the argument data type`

- xmlagg(expression)

  xmlagg 함수는 입력된 XML 값들을 하나의 XML로 반환한다.

  인자 타입 : `xml`

  반환 타입 : `xml`

- corr(Y, X)

  corr 함수는 입력된 두 수의 피어슨 상관계수 (correlation coefficient) 를 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- covar_pop(Y, X)

  covar_pop 함수는 입력된 두 수의 공분산 (population covariance) 을 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- covar_samp(Y, X)

  covar_samp 함수는 입력된 두 수의 표본공분산 (sample covariance) 을 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- regr_avgx(Y, X)

  regr_avgx 함수는 입력된 독립변수 (independent variable) X의 평균값을 반환한다 (sum(X)/N).

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- regr_avgy(Y, X)

  regr_avgy 함수는 입력된 종속변수 (dependent variable) Y의 평균값을 반환한다 (sum(X)/N).

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- regr_count(Y, X)

  regr_count 함수는 입력된 쌍이 모두 null이 아닌 쌍의 수를 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `bigint`

- regr_intercept(Y, X)

  regr_intercept 함수는 입력된 (X, Y) 쌍에 의해 구해진 최소 제곱법 선형방정식 (least-squares-fit linear equation) 의 y절편값을 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- regr_r2(Y, X)

  regr_r2 함수는 입력된 두 수열의 상관계수 (correlation coefficient) 의 제곱 값을 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- regr_slope(Y, X)

  regr_slope 함수는 입력된 (X, Y) 쌍에 의해 구해진 최소 제곱법 선형방정식 (least-squares-fit linear equation) 의 y절편값을 반환한다. 기울기값을 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- regr_sxx(Y, X)

  regr_sxx 함수는 sum(X^2) - sum(X)^2/N (독립변수 (independent variable) 의 제곱의 합) 을 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- regr_sxy(Y, X)

  regr_sxy 함수는 sum(X*Y) - sum(X)*sum(Y)/N (독립변수 (independent variable) 와 종속변수 (dependent variable) 의 제곱의 합) 을 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- regr_syy(Y, X)

  regr_syy 함수는 sum(Y^2) - sum(Y)^2/N (종속변수 (dependent variable) 의 제곱의 합) 을 반환한다.

  인자 타입 : `double precision`

  반환 타입 : `double precision`

- stddev(expression)

  stddev 함수는 stddev_samp의 historical alias로서 stddev_samp와 동일한값을 반환한다.

  인자 타입 : `smallint, int, bigint, real, double precision, or numeric`

  반환 타입 : `double precision for floating-point arguments, otherwise numeric`

- stddev_pop(expression)

  stddev_pop 함수는 입력된 값들의 모표준편차값을 반환한다.

  인자 타입 : `smallint, int, bigint, real, double precision, or numeric`

  반환 타입 : `double precision for floating-point arguments, otherwise numeric`

- stddev_samp(expression)

  stddev_samp 함수는 입력된 값들의 표본표준편차값을 반환한다.

  인자 타입 : `smallint, int, bigint, real, double precision, or numeric`

  반환 타입 : `double precision for floating-point arguments, otherwise numeric`

- variance(expression)

  variance 함수는 var_samp의 historical alias로서 var_samp와 동일한값을 반환한다.

  인자 타입 : `smallint, int, bigint, real, double precision, or numeric`

  반환 타입 : `double precision for floating-point arguments, otherwise numeric`

- var_pop(expression)

  var_pop 함수는 입력된 값들의 모분산값(모표준편차의 제곱)을 반환한다.

  인자 타입 : `smallint, int, bigint, real, double precision, or numeric`

  반환 타입 : `double precision for floating-point arguments, otherwise numeric`

- var_samp(expression)

  var_samp 함수는 입력된 값들의 표본분산값(표본표준 편차의 제곱)을 반환한다.

  인자 타입 : `smallint, int, bigint, real, double precision, or numeric`

  반환 타입 : `double precision for floating-point arguments, otherwise numeric`

- mode() WITHIN GROUP (ORDER BY sort_expression)

  mode 함수는 입력값 중 최빈값을 반환한다(만약 최빈값이 여러개일 경우, 그 중 임의의 값 1개만 출력된다).

  인자 타입 : `any sortable type`

  반환 타입 : `same as sort expression`

- percentile_cont(fraction) WITHIN GROUP (ORDER BY sort_expression)

  percentile_cont 함수는 입력된 fraction에 해당하는 정렬값을 반환한다(만약 딱 나눠떨어지지 않을 경우, 보간법을 이용하여 값을 반환한다).

  인자 타입 : `double precision or interval`

  반환 타입 : `same as sort expression`

- percentile_cont(fractions) WITHIN GROUP (ORDER BY sort_expression)

  percentile_cont 함수는 입력된 배열에 대해 각각의 fractions에 해당하는 정렬값들을 반환한다.

인자 타입 : `double precision or interval`

반환 타입 : `array of sort expression's type`

- percentile_disc(fraction) WITHIN GROUP (ORDER BY sort_expression)

  percentile_disc 함수는 입력된 fraction에 정렬의 처음값과 같거나 초과하는 위치의 값을 반환한다.

  인자 타입 : `any sortable type`

  반환 타입 : `same as sort expression`

- percentile_disc(fractions) WITHIN GROUP (ORDER BY sort_expression)

  percentile_disc 함수는 입력된 배열에 대해 각각의 fractions 원소와 매칭되는 결과의 배열을 반환한다. 각 null이 아닌 원소는 입력값의 백분위에 해당하는 값으로 변경된다.

  인자 타입 : `any sortable type`

  반환 타입 : `array of sort expression's type`

- rank(args) WITHIN GROUP (ORDER BY sorted_args)

  rank 함수는 인자의 (중복되는 행들의 격차를 포함한) rank값을 반환한다.

  인자 타입 : `VARIADIC "any"`

  반환 타입 : `bigint`

- dense_rank(args) WITHIN GROUP (ORDER BY sorted_args)

  dense_rank 함수는 인자의 (중복되는 행들의 격차를 제외한) rank값을 반환한다.

  인자 타입 : `VARIADIC "any"`

  반환 타입 : `bigint`

- percent_rank(args) WITHIN GROUP (ORDER BY sorted_args)

  percent_rank 함수는 인자의 상대적 rank값을 0~1사이의 값으로 반환한다.

  인자 타입 : `VARIADIC "any"`

  반환 타입 : `double precision`

- cume_dist(args) WITHIN GROUP (ORDER BY sorted_args)

  cume_dist 함수는 인자의 상대적 rank값을 1/N~1사이의 값으로 반환한다.

  인자 타입 : `VARIADIC "any"`

  반환 타입 : `double precision`

- GROUPING(args…)

  GROUPING 함수는 현재 그룹화된 집합에서 어떤 args가 속하지 않는지 나타내는 정수형 비트 마스크 (bit mask) 를 반환한다.

  반환 타입 : `integer`

4.3.16 Window Functions

- row_number()

  row_number 함수는 처리중인 파티션 내에서의 현재 행 번호를 반환한다(count는 1부터 시작한다).

  반환 타입 : `bigint`

- rank()

  rank 함수는 격차들을 포함한 현재 행의 rank값을 반환한다. 즉, first peer의 row_number와 동일하다.

  반환 타입 : `bigint`

- dense_rank()

  dense_rank 함수는 gap을 포함하지 않는 현재 행의 rank값을 반환한다. 즉, peer group을 센다.

  반환 타입 : `bigint`

- percent_rank()

  percent_rank 함수는 현재 행의 상대적 순위 ((rank-1)/(전체열-1)) 을 반환한다.

  반환 타입 : `double precision`

- cume_dist()

  cume_dist 함수는 현재 행의 상대적 순위 ((현재 행 번호)/(전체열의 수)) 을 반환한다.

  반환 타입 : `double precision`

- ntile(num_buckets integer)

  ntile 함수는 1부터 입력된 숫자범위까지의 구간을 가능한 동일하게 나눈 결과를 반환한다.

  반환 타입 : `integer`

- lag(value anyelement [, offset integer [, default anyelement]])

  lag 함수는 작업 구역 (partition) 내에 있는 행들 중에서, 현재 행보다 offset 만큼 앞에 있는 행이라고 평가되는 값을 반환한다. 해당되는 행이 없다면 default를 반환한다. offset과 default가 생략되면, offset은 1로 default는 null이 된다.

  반환 타입 : `same type as value`

- lead(value anyelement [, offset integer [, default anyelement]])

  lead 함수는 작업 구역 (partition) 내에 있는 행들 중에서, 현재 행보다 offset 만큼 뒤에 있는 행이라고 평가되는 값을 반환한다. 해당되는 행이 없다면 default를 반환한다. offset과 default가 생략되면, offset은 1로 default는 null이 된다.

  반환 타입 : `same type as value`

- first_value(value any)

  first_value 함수는 window frame 내의 행들 중, 첫번째 행인 것으로 평가되는 값을 반환한다.

  반환 타입 : `same type as value`

- last_value(value any)

  last_value 함수는 window frame 내의 행들 중, 마지막 행인 것으로 평가되는 값을 반환한다.

  반환 타입 : `same type as value`

- nth_value(value any, nth integer)

  nth_value 함수는 window frame 내의 행들 중, n번째 행인 것으로 평가되는 값을 반환한다(값은 1부터 세며, 만약 n번째 행이 없을 시 null을 반환한다).

  반환 타입 : `same type as value`

4.3.17 System Information Functions

- current_catalog

  current_catalog() 함수는 현재 데이터베이스의 이름을 반환한다. (SQL 표준에서 ``카탈로그'' 라고 함.)

```
SELECT current_catalog;

Result:
 current_database
------------------
       test
           (1 row)
```

- current_database()

  current_database() 함수는 현재 데이터베이스의 이름을 반환한다.

```
SELECT current_database();

Result:
 current_database
------------------
       test
           (1 row)
```

- current_query()

current_query() 함수는 클라이언트가 요청하여 현재 실행중인 쿼리의 텍스트를 반환한다.(두 개 이상의 명령문을 포함 할 수 있음)

```
SELECT current_query();

Result:
      current_query
--------------------------
 SELECT current_query();
                  (1 row)
```

- current_role

current_role() 함수는 `current_user`와 동일한 값을 반환한다.

```
SELECT current_role;

Result:
 current_user
--------------
    agens
       (1 row)
```

- current_schema[()]

current_schema 함수는 현재 스키마의 이름을 반환한다.

```
SELECT current_schema();

Result:
 current_schema
----------------
     public
         (1 row)
```

- current_schemas(boolean)

current_schemas() 함수는 검색경로에서 스키마의 이름을 반환한다(선택적으로 묵시적 스키마를 포함할지 선택할 수 있다).

```
SELECT current_schemas(true);

Result:
   current_schemas
---------------------
 {pg_catalog,public}
              (1 row)
```

- current_user

  current_user() 함수는 현재 수행상의 사용자 이름을 반환한다.

```
SELECT current_user;

Result:
 current_user
--------------
    agens
       (1 row)
```

- inet_client_addr()

  inet_client_add() 함수는 원격 연결의 주소를 반환한다.

```
SELECT inet_client_addr();

Result:
 inet_client_addr
------------------
       ::1
           (1 row)
```

- inet_client_port()

  inet_client_port() 함수는 원격 연결의 port를 반환한다.

```
SELECT inet_client_port();

Result:
 inet_client_port
```

```
-------------------
     64427
          (1 row)
```

- inet_server_addr()

  inet_server_addr() 함수는 로컬 연결의 주소를 반환한다.

```
SELECT inet_server_addr();

Result:
 inet_server_addr
-------------------
      ::1
          (1 row)
```

- inet_server_port()

  inet_server_port 함수는 연결되어 있는 서버 port 번호를 반환한다.

```
SELECT inet_server_port();

Result:
 inet_server_port
-------------------
      5432
          (1 row)
```

- pg_backend_pid()

  pg_backend_pid 함수는 현재 세션에 연결된 서버 프로세스의 프로세스 ID를 반환한다.

```
SELECT pg_backend_pid();

Result:
 pg_backend_pid
-----------------
         61675
        (1 row)
```

- pg_blocking_pids(int)

pg_blocking_pids 함수는 지정된 서버 프로세스 ID를 차단하는 프로세스 ID를 반환한다.

```
SELECT pg_blocking_pids(61675);


Result:
 pg_blocking_pids
------------------
        {}
           (1 row)
```

- pg_conf_load_time()

  pg_conf_load_time 함수는 구성 load의 시간을 반환한다.

```
SELECT pg_conf_load_time();


Result:
      pg_conf_load_time
------------------------------
 2017-10-18 13:36:51.99984+09
                       (1 row)
```

- pg_my_temp_schema()

  pg_my_temp_schema 함수는 세션의 임시 스키마의 OID를 반환한다. 임시 스키마가 없는 경우 0을 반환한다.

```
SELECT pg_my_temp_schema();


Result:
 pg_my_temp_schema
-------------------
                 0
            (1 row)
```

- pg_is_other_temp_schema(oid)

  pg_is_other_temp_schema 함수는 스키마가 다른 세션의 임시 스키마인지 여부를 확인하여 값을 반환한다.

```
SELECT pg_is_other_temp_schema(61675);


Result:
```

```
 pg_is_other_temp_schema
-------------------------
           f
                  (1 row)
```

- pg_listening_channels()

  pg_listening_channels 함수는 세션에서 현재 수신 대기 중인 채널 이름을 반환한다.

```
SELECT pg_listening_channels();

Result:
 pg_listening_channels
-----------------------
                (0 row)
```

- pg_notification_queue_usage()

  pg_notification_queue_usage 함수는 현재 점유된 비동기식 알림 queue의 일부분을 반환한다(0-1).

```
SELECT pg_notification_queue_usage();

Result:
 pg_notification_queue_usage
-----------------------------
                           0
                      (1 row)
```

- pg_postmaster_start_time()

  pg_postmaster_start_time 함수는 서버의 시작 시간을 반환한다.

```
SELECT pg_postmaster_start_time();

Result:
   pg_postmaster_start_time
-------------------------------
 2017-10-18 13:36:52.019037+09
                        (1 row)
```

- pg_trigger_depth()

pg_trigger_depth 함수는 트리거의 현재 중첩 수준을 반환한다(트리거 내부에서 직접 또는 간접적으로 호출되지 않으면 0을 반환).

```
SELECT pg_trigger_depth();

Result:
 pg_trigger_depth
------------------
                0
           (1 row)
```

- session_user

session_user 함수는 세션 사용자 이름을 반환한다.

```
SELECT session_user;

Result:
 session_user
--------------
     agens
       (1 row)
```

- user

user 함수는 `current_user`와 동일한 값을 반환한다.

```
SELECT user;

Result:
 current_user
--------------
     agens
       (1 row)
```

- version()

version 함수는 AgensGraph의 버전 정보를 반환한다.

```
SELECT version();

Result:
                                version
--------------------------------------------------------------------------------
 PostgreSQL 9.6.2 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.4.7 20120313
 (Red Hat 4.4.7-17), 64-bit
```

- has_any_column_privilege(user, table, privilege)

  has_any_column_privilege 함수는 user가 table의 모든 컬럼에 대해 다른 것들과 동일한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_any_column_privilege('agens', 'myschema.mytable', 'SELECT');

Result:
 has_any_column_privilege
--------------------------
            t
                   (1 row)
```

- has_any_column_privilege(table, privilege)

  이 함수는 현재 사용자가 table의 모든 컬럼에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_any_column_privilege('myschema.mytable', 'SELECT');

Result:
 has_any_column_privilege
--------------------------
            t
                   (1 row)
```

- has_column_privilege(user, table, column, privilege)

  has_column_privilege 함수는 user가 table의 컬럼에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_column_privilege('agens', 'myschema.mytable', 'col1', 'SELECT');


Result:
 has_column_privilege
----------------------
          t
               (1 row)
```

- has_column_privilege(table, column, privilege)

  이 함수는 현재 사용자가 table의 컬럼에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_column_privilege('myschema.mytable', 'col1', 'SELECT');


Result:
 has_column_privilege
----------------------
          t
               (1 row)
```

- has_database_privilege(user, database, privilege)

  has_database_privilege 함수는 user가 database에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_database_privilege('agens', 'test', 'connect');


Result:
 has_database_privilege
------------------------
           t
                (1 row)
```

- has_database_privilege(database, privilege)

  이 함수는 현재 사용자가 database에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_database_privilege('test', 'connect');


Result:
 has_database_privilege
```

```
------------------------
           t
               (1 row)
```

- has_foreign_data_wrapper_privilege(user, fdw, privilege)

  has_foreign_data_wrapper_privilege 함수는 user가 foreign-data wrapper에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
CREATE EXTENSION postgres_fdw;

SELECT has_foreign_data_wrapper_privilege('agens', 'postgres_fdw', 'usage');

Result:
 has_foreign_data_wrapper_privilege
------------------------------------
                 t
                             (1 row)
```

- has_foreign_data_wrapper_privilege(fdw, privilege)

  이 함수는 현재 사용자가 foreign-data wrapper에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_foreign_data_wrapper_privilege('postgres_fdw', 'usage');

Result:
 has_foreign_data_wrapper_privilege
------------------------------------
                 t
                             (1 row)
```

- has_function_privilege(user, function, privilege))

  has_function_privilege 함수는 user가 function에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_function_privilege('agens', 'getfoo()', 'execute');

Result:
 has_function_privilege
------------------------
```

```
           t
                 (1 row)
```

- has_function_privilege(function, privilege)

  이 함수는 현재 사용자가 function에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_function_privilege('getfoo()', 'execute');


Result:
 has_function_privilege
------------------------
           t
                 (1 row)
```

- has_language_privilege(user, language, privilege)

  has_language_privilege 함수는 user가 language에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_language_privilege('agens', 'c', 'usage');


Result:
 has_language_privilege
------------------------
           t
                 (1 row)
```

- has_language_privilege(language, privilege)

  이 함수는 현재 사용자가 language에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_language_privilege('c', 'usage');


Result:
 has_language_privilege
------------------------
           t
                 (1 row)
```

- has_schema_privilege(user, schema, privilege)

  has_schema_privilege 함수는 user가 schema에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_schema_privilege('agens', 'myschema', 'usage');


Result:
 has_schema_privilege
----------------------
          t
               (1 row)
```

- has_schema_privilege(schema, privilege)

  이 함수는 현재 사용자가 schema에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_schema_privilege('myschema', 'usage');


Result:
 has_schema_privilege
----------------------
          t
               (1 row)
```

- has_sequence_privilege(user, sequence, privilege)

  has_sequence_privilege 함수는 user가 sequence에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
 SELECT has_sequence_privilege('agens', 'serial', 'usage');


 Result:
  has_sequence_privilege
 ------------------------
            t
                 (1 row)
```

- has_sequence_privilege(sequence, privilege)

  이 함수는 현재 사용자가 sequence에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_sequence_privilege('serial', 'usage');


 Result:
  has_sequence_privilege
```

```
------------------------
           t
                 (1 row)
```

- has_server_privilege(user, server, privilege)

  has_server_privilege 함수는 user가 server에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
CREATE SERVER app_database_server
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host '127.0.0.1', dbname 'agens');

SELECT has_server_privilege('agens', 'app_database_server', 'usage');

Result:
 has_server_privilege
----------------------
         t
              (1 row)
```

- has_server_privilege(server, privilege)

  이 함수는 현재 사용자가 server에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_server_privilege('app_database_server', 'usage');

Result:
 has_server_privilege
----------------------
         t
              (1 row)
```

- has_table_privilege(user, table, privilege)

  has_table_privilege 함수는 user가 table에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_table_privilege('agens', 'myschema.mytable', 'SELECT');

Result:
 has_table_privilege
```

```
---------------------
          t
               (1 row)
```

- has_table_privilege(table, privilege)

  이 함수는 현재 사용자가 table에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_table_privilege('myschema.mytable', 'SELECT');

Result:
 has_table_privilege
---------------------
          t
               (1 row)
```

- has_tablespace_privilege(user, tablespace, privilege)

  has_tablespace_privilege 함수는 user가 tablespace에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_tablespace_privilege('agens', 'pg_default', 'create');

Result:
 has_tablespace_privilege
--------------------------
             t
                   (1 row)
```

- has_tablespace_privilege(tablespace, privilege)

  이 함수는 현재 사용자가 tablespace에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_tablespace_privilege('pg_default', 'create');

Result:
 has_tablespace_privilege
--------------------------
             t
                   (1 row)
```

- has_type_privilege(user, type, privilege)

has_type_privilege 함수는 user가 type에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_type_privilege('agens', 'rainbow', 'usage');


Result:
 has_type_privilege
---------------------
         t
             (1 row)
```

- has_type_privilege(type, privilege)

  이 함수는 현재 사용자가 type에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT has_type_privilege('rainbow', 'usage');


Result:
 has_type_privilege
---------------------
         t
             (1 row)
```

- pg_has_role(user, role, privilege)

  pg_has_role 함수는 user가 role에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT pg_has_role('agens', 'agens', 'usage');


Result:
 pg_has_role
-------------
      t
      (1 row)
```

- pg_has_role(role, privilege)

  이 함수는 현재 사용자가 role에 대한 권한을 가지고 있는지 논리값을 반환한다.

```
SELECT pg_has_role('agens', 'usage');


Result:
```

```
 pg_has_role
-------------
     t
     (1 row)
```

- row_security_active(table)

  row_security_active 함수는 현재 사용자가 table에 대해 row level의 보안이 활성화 되어있는지 논리값을 반환한다.

```
SELECT row_security_active('myschema.mytable');


Result:
 row_security_active
---------------------
        f
             (1 row)
```

- pg_collation_is_visible(collation_oid)

  pg_collation_is_visible 함수는 검색경로에 collation이 존재하는지에 대한 정보를 반환한다.

```
SELECT pg_collation_is_visible(100);


Result:
 pg_collation_is_visible
-------------------------
           t
                 (1 row)
```

- pg_conversion_is_visible(conversion_oid)

  pg_conversion_is_visible 함수는 search path에 conversion 존재 여부 정보를 반환한다.

```
SELECT pg_conversion_is_visible(12830);


Result:
 pg_conversion_is_visible
--------------------------
           t
                  (1 row)
```

- pg_function_is_visible(function_oid)

  pg_function_is_visible 함수는 search path에 function 존재 여부 정보를 반환한다.

```
SELECT pg_function_is_visible(16716);


Result:
 pg_function_is_visible
------------------------
            t
                 (1 row)
```

- pg_opclass_is_visible(opclass_oid)

  pg_opclass_is_visible 함수는 search path에 opclass 존재 여부 정보를 반환한다.

```
SELECT pg_opclass_is_visible(10007);


Result:
 pg_opclass_is_visible
-----------------------
            t
                 (1 row)
```

- pg_operator_is_visible(operator_oid)

  pg_operator_is_visible 함수는 search path에 operator 존재 여부 정보를 반환한다.

```
SELECT pg_operator_is_visible(15);


Result:
 pg_operator_is_visible
------------------------
            t
                 (1 row)
```

- pg_opfamily_is_visible(opclass_oid)

  pg_opfamily_is_visible 함수는 search path에 opfamily 존재 여부 정보를 반환한다.

```
SELECT pg_opfamily_is_visible(421);


Result:
 pg_opfamily_is_visible
------------------------
            t
                 (1 row)
```

- pg_table_is_visible(table_oid)

  pg_table_is_visible 함수는 search path에 table 존재 여부 정보를 반환한다.

```
SELECT pg_table_is_visible(16553);


Result:
 pg_table_is_visible
---------------------
          t
              (1 row)
```

- pg_ts_config_is_visible(config_oid)

  pg_ts_config_is_visible 함수는 search path에 text search configuration 존재 여부 정보를 반환한다.

```
SELECT pg_ts_config_is_visible(3748);


Result:
 pg_ts_config_is_visible
-------------------------
            t
                  (1 row)
```

- pg_ts_dict_is_visible(dict_oid)

  pg_ts_dict_is_visible 함수는 search path에 text search dictionary 존재 여부 정보를 반환한다.

```
SELECT pg_ts_dict_is_visible(3765);


Result:
 pg_ts_dict_is_visible
```

```
------------------------
           t
                (1 row)
```

- pg_ts_parser_is_visible(parser_oid)

  pg_ts_parser_is_visible 함수는 search path에 text search parser 존재 여부 정보를 반환한다.

```
SELECT pg_ts_parser_is_visible(3722);

Result:
 pg_ts_parser_is_visible
-------------------------
             t
                  (1 row)
```

- pg_ts_template_is_visible(template_oid)

  pg_ts_template_is_visible 함수는 search path에 text search template 존재 여부 정보를 반환한다.

```
SELECT pg_ts_template_is_visible(3727);

Result:
 pg_ts_template_is_visible
---------------------------
             t
                    (1 row)
```

- pg_type_is_visible(type_oid)

  pg_type_is_visible 함수는 search path에 type 또는 domain 존재 여부 정보를 반환한다.

```
SELECT pg_type_is_visible(16);

Result:
 pg_type_is_visible
--------------------
          t
               (1 row)
```

- format_type(type_oid, typemod)

format_type 함수는 데이터 타입명을 반환한다.

```
SELECT format_type(16, 1);

Result:
 format_type
-------------
   boolean
      (1 row)
```

- pg_get_constraintdef(constraint_oid)

  pg_get_constraintdef 함수는 constraint의 정의를 반환한다.

```
SELECT pg_get_constraintdef(13096);

Result:
 pg_get_constraintdef
----------------------
 CHECK ((VALUE >= 0))
              (1 row)
```

- pg_get_constraintdef(constraint_oid, pretty_bool)

  이 함수는 constraint의 정의를 반환한다.

```
SELECT pg_get_constraintdef(13096, true);

Result:
 pg_get_constraintdef
----------------------
 CHECK ((VALUE >= 0))
              (1 row)
```

- pg_get_functiondef(func_oid)

  pg_get_functiondef 함수는 function의 정의를 반환한다.

```
SELECT pg_get_functiondef(16716);

Result:
```

```
             pg_get_functiondef
--------------------------------------------
 CREATE OR REPLACE FUNCTION public.getfoo()+
 ...
                                       (1 row)
```

- pg_get_function_arguments(func_oid)

  pg_get_function_arguments 함수는 함수 정의의 인자 목록(기본값)을 반환한다.

```
SELECT pg_get_function_arguments(16739);


Result:
     pg_get_function_arguments
------------------------------------
 double precision, double precision
                              (1 row)
```

- pg_get_function_identity_arguments(func_oid)

  pg_get_function_identity_arguments 함수는 함수를 식별하는 인자 목록(기본값 없음)을 반환한다.

```
SELECT pg_get_function_identity_arguments(16739);


Result:
 pg_get_function_identity_arguments
------------------------------------
 double precision, double precision
                              (1 row)
```

- pg_get_function_result(func_oid)

  pg_get_function_result 함수는 함수의 RETURN구문을 반환한다.

```
SELECT pg_get_function_result(16739);


Result:
 pg_get_function_result
------------------------
       float8_range
              (1 row)
```

- pg_get_indexdef(index_oid)

  pg_get_indexdef 함수는 index에 대한 `CREATE INDEX`명령어를 반환한다.

```
SELECT pg_get_indexdef(828);


    Result:
                                                pg_get_indexdef
------------------------------------------------------------------------------------
 CREATE UNIQUE INDEX pg_default_acl_oid_index ON pg_default_acl USING btree (oid)
                                                                            (1 row)
```

- pg_get_indexdef(index_oid, column_no, pretty_bool)

  pg_get_indexdef 함수는 index에 대한 `CREATE INDEX`명령어를 반환한다. `column_no` 값이 0이 아닌 경우 단 하나의 인덱스열을 정의한다.

```
SELECT pg_get_indexdef(828, 1, true);


Result:
 pg_get_indexdef
-----------------
       oid
          (1 row)
```

- pg_get_keywords()

  pg_get_keywords 함수는 SQL 키워드의 목록이나 카테고리를 반환한다.

```
SELECT pg_get_keywords();


Result:
       pg_get_keywords
----------------------------------
 (abort,U,"예약되지 않음")
 (absolute,U,"예약되지 않음")
 ...
                          (433 row)
```

- pg_get_ruledef(rule_oid)

  pg_get_ruledef 함수는 규칙에 대한 `CREATE RULE`문을 보여준다.

```
SELECT pg_get_ruledef(11732);


Result:
            pg_get_ruledef
----------------------------------------
 CREATE RULE "_RETURN" AS
 ...
                                 (1 row)
```

- pg_get_ruledef(rule_oid, pretty_bool)

  이 함수는 규칙에 대한 CREATE RULE명령어를 반환한다.

```
SELECT pg_get_ruledef(11732, true);


Result:
            pg_get_ruledef
----------------------------------------
 CREATE RULE "_RETURN" AS
 ...
                                 (1 row)
```

- pg_get_serial_sequence(table_name, column_name)

  pg_get_serial_sequence 함수는 serial, smallserial, bigserial 컬럼을 사용하는 시퀀스의 이름을 반환한다.

```
SELECT pg_get_serial_sequence('serial_t', 'col1');


Result:
  pg_get_serial_sequence
--------------------------
 public.serial_t_col1_seq
                     (1 row)
```

- pg_get_triggerdef(trigger_oid)

  pg_get_triggerdef 함수는 trigger에 대한 CREATE [CONSTRAINT] TRIGGER문을 보여준다.

```
SELECT pg_get_triggerdef(16887);

Result:
                          pg_get_triggerdef
------------------------------------------------------------------------
 CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE ON messages
 FOR EACH ROW EXECUTE PROCEDURE
 tsvector_update_trigger_column('tsv', 'configcol', 'title', 'body')
                                                                   (1 row)
```

- pg_get_triggerdef(trigger_oid, pretty_bool)

  이 함수는 trigger에 대한 CREATE [CONSTRAINT] TRIGGER문을 보여준다.

```
SELECT pg_get_triggerdef(16887, true);

Result:
                          pg_get_triggerdef
------------------------------------------------------------------------
 CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE ON messages
 FOR EACH ROW EXECUTE PROCEDURE
 tsvector_update_trigger_column('tsv', 'configcol', 'title', 'body')
                                                                   (1 row)
```

- pg_get_userbyid(role_oid)

  pg_get_userbyid 함수는 지정된 OID로 role명을 반환한다.

```
SELECT pg_get_userbyid(13096);

Result:
 pg_get_userbyid
-----------------
      agens
          (1 row)
```

- pg_get_viewdef(view_oid)

  이 함수는 view 또는 mview에 대한 기본 SELECT문을 보여준다.

```
SELECT pg_get_viewdef(17046);

Result:
          pg_get_viewdef
-----------------------------------
  SELECT pg_class.relname,         +
     pg_class.relnamespace,        +
     ...
    FROM pg_class;
                             (1 row)
```

- pg_get_viewdef(view_oid, pretty_bool)

  이 함수는 view 또는 mview에 대한 기본 SELECT문을 보여준다.

```
SELECT pg_get_viewdef(17046, true);

Result:
          pg_get_viewdef
-----------------------------------
  SELECT pg_class.relname,         +
     pg_class.relnamespace,        +
     ...
    FROM pg_class;
                            (1 row)
```

- pg_get_viewdef(view_oid, wrap_column_int)

  이 함수는 view 또는 mview에 대한 기본 SELECT문을 보여준다. 필드가 있는 행은 지정된 열 수로 줄 바꿈되어 정리된 값을 보여준다.

```
SELECT pg_get_viewdef(17046,50);

Result:
                   pg_get_viewdef
-----------------------------------------------------
  SELECT pg_class.relname, pg_class.relnamespace,   +
     pg_class.reltype, pg_class.reloftype,          +
```

```
    pg_class.relowner, pg_class.relam,             +
    pg_class.relfilenode, pg_class.reltablespace,  +
    pg_class.relpages, pg_class.reltuples,         +
    pg_class.relallvisible, pg_class.reltoastrelid,+
    pg_class.relhasindex, pg_class.relisshared,    +
    pg_class.relpersistence, pg_class.relkind,     +
    pg_class.relnatts, pg_class.relchecks,         +
    pg_class.relhasoids, pg_class.relhaspkey,      +
    pg_class.relhasrules, pg_class.relhastriggers, +
    pg_class.relhassubclass,                       +
    pg_class.relrowsecurity,                       +
    pg_class.relforcerowsecurity,                  +
    pg_class.relispopulated, pg_class.relreplident,+
    pg_class.relfrozenxid, pg_class.relminmxid,    +
    pg_class.relacl, pg_class.reloptions           +
   FROM pg_class;
                                              (1 row)
```

- pg_index_column_has_property(index_oid, column_no, prop_name)

  pg_index_column_has_property 함수는 인덱스 열에 지정된 속성이 있는지 여부를 판단하여 값을 반환한다.

```
SELECT pg_index_column_has_property(17134, 1, 'orderable');

Result:
 pg_index_column_has_property
------------------------------
             t
                     (1 row)
```

- pg_index_has_property(index_oid, prop_name)

  pg_index_has_property 함수는 색인에 지정된 속성이 있는지 여부를 판단하여 값을 반환한다.

```
SELECT pg_index_has_property(17134, 'clusterable');

Result:
 pg_index_has_property
```

```
-----------------------
          t
               (1 row)
```

- pg_indexam_has_property(am_oid, prop_name)

  pg_indexam_has_property 함수는 인덱스 액세스 메서드가 지정된 속성을 가지고 있는지 여부를 판단하여 값을 반환한다.

```
SELECT pg_indexam_has_property(403, 'can_order');

Result:
 pg_indexam_has_property
-------------------------
          t
                  (1 row)
```

- pg_options_to_table(reloptions)

  pg_options_to_table 함수는 스토리지 옵션의 name/value set을 반환한다.

```
SELECT pg_options_to_table(reloptions) FROM pg_class;

Result:
   pg_options_to_table
-------------------------
 (security_barrier,true)
                  (1 row)
```

- pg_tablespace_databases(tablespace_oid)

  pg_tablespace_databases 함수는 테이블 스페이스에 객체가 있는 데이터베이스 OID 집합을 반환한다.

```
SELECT pg_tablespace_databases(1663);

Result:
 pg_tablespace_databases
-------------------------
                       1
                   13372
```

```
                 13373
                 16384
                 16482
                (5 row)
```

- pg_tablespace_location(tablespace_oid)

  pg_tablespace_location 함수는 테이블 공간이 위치한 파일 시스템의 경로를 반환한다.

```
SELECT pg_tablespace_location(1663);

Result:
      pg_tablespace_location
----------------------------------
 /home/agens/AgensGraph/db_cluster
                          (1 row)
```

- pg_typeof(any)

  pg_typeof 함수는 어떤 값의 data type을 반환한다.

```
SELECT pg_typeof(1);

Result:
 pg_typeof
-----------
  integer
     (1 row)
```

- collation for (any)

  collation for 함수는 인자의 데이터 정렬을 반환한다.

```
SELECT collation for ('foo' COLLATE "de_DE");

Result:
 pg_collation_for
------------------
     "de_DE"
           (1 row)
```

- pg_describe_object(catalog_id, object_id, object_sub_id)

  pg_describe_object 함수는 database object의 설명을 반환한다.

```
SELECT pg_describe_object(1255, 16716, 0);


Result:
 pg_describe_object
--------------------
    getfoo() 함수
             (1 row)
```

- pg_identify_object(catalog_id oid, object_id oid, object_sub_id integer)

  pg_identify_object 함수는 database object의 식별 정보를 반환한다.

```
SELECT pg_identify_object(1255, 16716, 0);


Result:
          pg_identify_object
--------------------------------------
 (function,public,,"public.getfoo()")
                                (1 row)
```

- pg_identify_object_as_address(catalog_id oid, object_id oid, object_sub_id integer)

  pg_identify_object_as_address 함수는 database object의 주소에 대한 식별자로 반환한다.

```
SELECT pg_identify_object_as_address(1255, 16716, 0);


Result:
  pg_identify_object_as_address
---------------------------------
 (function,"{public,getfoo}",{})
                           (1 row)
```

- pg_get_object_address(type text, name text[], args text[])

  pg_get_object_address 함수는 식별자로부터 데이터베이스 객체의 주소를 반환한다.

```
SELECT pg_get_object_address('type', '{public.comp}', '{}');


Result:
 pg_get_object_address
-----------------------
     (1247,17063,0)
                (1 row)
```

- col_description(table_oid, column_number)

  col_description 함수는 테이블 컬럼의 주석을 반환한다.

```
SELECT col_description(17064, 1);


Result:
 col_description
-----------------
   code_number
          (1 row)
```

- obj_description(object_oid, catalog_name)

  obj_description 함수는 database object의 주석을 반환한다.

```
SELECT obj_description(16887, 'pg_trigger');


Result:
  obj_description
--------------------
 comment on trigger
             (1 row)
```

- obj_description(object_oid)

  이 함수는 database object의 주석을 반환한다.(더이상 사용되지 않음)

```
SELECT obj_description(16887);


Result:
  obj_description
```

```
--------------------
 comment on trigger
            (1 row)
```

- shobj_description(object_oid, catalog_name)

  이 함수는 공유 database object의 주석을 반환한다.

```
SELECT shobj_description(1262,'pg_database');


Result:
 shobj_description
-------------------


           (1 row)
```

- txid_current()

  txid_current 함수는 현재 트랜잭션 ID를 가져오고, 현재 트랜잭션이 없는 경우 새 트랜잭션 ID를 할당한다.

```
SELECT txid_current();


Result:
 txid_current
--------------
         2061
      (1 row)
```

- txid_current_snapshot()

  txid_current_snapshot 함수는 현재 스냅샷 값을 반환한다.

```
SELECT txid_current_snapshot();


Result:
 txid_current_snapshot
-----------------------
      2062:2062:
               (1 row)
```

- txid_current()

txid_current 함수는 현재 트랜잭션 ID를 가져오고, 현재 트랜잭션이 없는 경우 새 트랜잭션 ID를 할당한다.

```
SELECT txid_current();


Result:
 txid_current
--------------
         2061
       (1 row)
```

- txid_current_snapshot()

  txid_current_snapshot 함수는 현재 스냅샷 값을 반환한다.

```
SELECT txid_current_snapshot();


Result:
 txid_current_snapshot
-----------------------
       2062:2062:
                (1 row)
```

- txid_snapshot_xip(txid_snapshot)

  txid_snapshot_xip 함수는 스냅샷에서 진행중인 트랜잭션 ID를 반환한다.

```
SELECT txid_snapshot_xip('2095:2095:');


Result:
 txid_snapshot_xip
-------------------


            (1 row)
```

- txid_snapshot_xmax(txid_snapshot)

  txid_snapshot_xmax 함수는 스냅샷의 최대값을 반환한다.

```
SELECT txid_snapshot_xmax('2094:2095:');


Result:
```

```
 txid_snapshot_xmax
--------------------
               2095
             (1 row)
```

- txid_snapshot_xmin(txid_snapshot)

  txid_snapshot_xmin 함수는 스냅샷의 최소값을 반환한다.

```
SELECT txid_snapshot_xmin('2094:2095:');

Result:
 txid_snapshot_xmin
--------------------
               2094
             (1 row)
```

- txid_visible_in_snapshot(bigint, txid_snapshot)

  txid_visible_in_snapshot 함수는 스냅샷의 트랜잭션 ID가 표시되는지 여부를 반환한다.(서브트랜잭션 ID를 사용하지 않는다.)

```
SELECT txid_visible_in_snapshot(2099, '2100:2100:');

Result:
 txid_visible_in_snapshot
--------------------------
            t
                   (1 row)
```

- pg_xact_commit_timestamp(xid)

  pg_xact_commit_timestamp 함수는 트랜잭션의 커밋된 시간을 반환한다.(`track_commit_timestamp` 파라미터가 on으로 설정되어야 한다.)

```
SELECT pg_xact_commit_timestamp('2097'::xid);

Result:
   pg_xact_commit_timestamp
-------------------------------
```

```
2017-10-18 13:38:09.738211+09
                        (1 row)
```

- pg_last_committed_xact()

  pg_last_committed_xact 함수는 최신 커밋된 트랜잭션의 트랜잭션 ID와 커밋된 시간을 반환한다. (track_commit_timestamp 파라미터가 on으로 설정되어야 한다.)

```
SELECT pg_last_committed_xact();


Result:
         pg_last_committed_xact
----------------------------------------
 (2097,"2017-10-18 13:38:09.738211+09")
                                (1 row)
```

- pg_control_checkpoint()

  pg_control_checkpoint 함수는 현재 체크포인트 상태 정보를 반환한다.

```
SELECT pg_control_checkpoint();


Result:
                     pg_control_checkpoint
-------------------------------------------------------------------
 (0/1D4B0D0,0/1D4B038,0/1D4B0D0,000000010000000000000001,
 1,1,t,0:2063,24576,1,0,1751,1,0,1,1,0,0,"2017-10-16 16:26:21+09")
                                                            (1 row)
```

- pg_control_system()

  pg_control_system 함수는 현재 컨트롤 파일 상태 정보를 반환한다.

```
SELECT pg_control_system();


Result:
                      pg_control_system
--------------------------------------------------------------
 (960,201608131,6469891178207434037,"2017-10-16 16:26:21+09")
                                                       (1 row)
```

- pg_control_init()

  pg_control_init 함수는 클러스터 초기화 상태 정보를 반환한다.

```
SELECT pg_control_init();


Result:
                    pg_control_init
-------------------------------------------------------
 (8,8192,131072,8192,16777216,64,32,1996,2048,t,t,t,0)
                                               (1 row)
```

- pg_control_recovery()

  pg_control_recovery 함수는 복구 상태 정보를 반환한다.

```
SELECT pg_control_recovery();


Result:
 pg_control_recovery
---------------------
  (0/0,0,0/0,0/0,f)
              (1 row)
```

4.3.18 System Administration Functions

- current_setting(setting_name [, missing_ok])

  current_setting 함수는 설정된 현재 값을 반환한다.

```
SELECT current_setting('datestyle');


Result:
 current_setting
-----------------
 ISO, YMD
         (1 row)
```

- set_config(setting_name, new_value, is_local)

  set_config 함수는 매개변수를 지정하고 새로운 값을 반환한다.

```
SELECT set_config('log_statement_stats', 'off', false);


Result:
 set_config
------------
    off
     (1 row)
```

- pg_cancel_backend(pid int)

  pg_cancel_backend 함수는 backend의 현재 질의를 취소한다. 이것은 호출하는 role이 backend가 취소되는 role의 멤버이거나 호출하는 role에pg_signal_backend가 부여 된 경우에도 허용된다. 그러나 superuser backend는 superuser만 취소 할 수 있다.

```
SELECT pg_cancel_backend(30819);
오류: 사용자 요청에 의해 작업을 취소합니다.
```

- pg_reload_conf()

  pg_reload_conf 함수는 모든 서버 프로세스에서 구성 파일을 다시 로드한다.

```
SELECT pg_reload_conf();


Result:
 pg_reload_conf
----------------
       t
         (1 row)
```

- pg_rotate_logfile()

  pg_rotate_logfile 함수는 즉시 새로운 로그 파일로 전환가능하도록 로그 파일 관리자에게 신호를 보낸다 (내장 로그 콜렉터가 실행 중일 때만 동작한다).

```
SELECT pg_rotate_logfile();


 pg_rotate_logfile
-------------------
         f
            (1 row)
```

- pg_terminate_backend(pid int)

  pg_terminate_backend 함수는 backend를 종료한다. 이것은 호출하는 role이 backend가 종료되는 role의 멤버이거나 호출하는 role에`pg_signal_backend`가 부여 된 경우에도 허용된다. 그러나 superuser backend는 superuser만 종료 할 수 있다.

```
SELECT pg_terminate_backend(30819);


Result:
치명적오류:  관리자 요청에 의해서 연결을 끝냅니다
            서버가 갑자기 연결을 닫았음
            이런 처리는 클라이언트의 요구를 처리하는 동안이나
            처리하기 전에 서버가 갑자기 종료되었음을 의미함
            서버로부터 연결이 끊어졌습니다. 다시 연결을 시도합니다: 성공.
```

- pg_create_restore_point(name text)

  pg_create_restore_point 함수는 복원 수행을 위해 명명 된 지점을 만든다.(기본적으로 수퍼 유저로 제한되지만 다른 사용자가 이 기능을 실행하려면 EXECUTE 권한 부여가 필요할 수 있다.)

```
SELECT pg_create_restore_point( 'important_moment' );


Result:
 pg_create_restore_point
-------------------------
       0/1D72DC0
                  (1 row)
```

- pg_current_xlog_flush_location()

  pg_current_xlog_flush_location 함수는 트랜잭션 로그 플러시 위치를 반환한다.

```
SELECT pg_current_xlog_flush_location();


Result:
 pg_current_xlog_flush_location
--------------------------------
            0/1D72ED8
                          (1 row)
```

- pg_current_xlog_insert_location()

pg_current_xlog_insert_location 함수는 현재 트랜잭션 로그를 입력하는 위치를 반환한다.

```
SELECT pg_current_xlog_insert_location();


Result:
 pg_current_xlog_insert_location
---------------------------------
            0/1D72ED8
                          (1 row)
```

- pg_current_xlog_location()

  pg_current_xlog_location 함수는 현재 트랜잭션 로그를 쓰는 위치를 반환한다.

```
SELECT pg_current_xlog_location();


Result:
 pg_current_xlog_location
--------------------------
         0/1D72ED8
                   (1 row)
```

- pg_start_backup(label text [, fast boolean [, exclusive boolean]])

  pg_start_backup 함수는 온라인 백업 수행을 위한 준비를 한다.(기본적으로 수퍼 유저로 제한되지만 다른 사용자가 이 기능을 실행하려면 EXECUTE 권한 부여가 필요할 수 있다.)

```
SELECT pg_start_backup('my_backup', true, false);


Result:
 pg_start_backup
----------------
     0/2000028
          (1 row)
```

- pg_stop_backup()

  pg_stop_backup 함수는 exclusive 온라인 백업 수행을 종료한다.(기본적으로 수퍼 유저로 제한되지만 다른 사용자가 이 기능을 실행하려면 EXECUTE 권한 부여가 필요할 수 있다.)

```
SELECT pg_stop_backup();


Result:
NOTICE: pg_stop_backup 작업이 끝났습니다. 모든 필요한 WAL 조각들이 아카이브 되었습니다.
 pg_stop_backup
----------------
 (0/50000F8,,)
         (1 row)
```

- pg_stop_backup(exclusive boolean)

  pg_stop_backup 함수는 exclusive 또는 일부 온라인 백업 수행을 종료한다.(기본적으로 수퍼 유저로 제한되지만 다른 사용자가 이 기능을 실행하려면 EXECUTE 권한 부여가 필요할 수 있다.)

```
SELECT pg_stop_backup(false);


Result:
NOTICE:  WAL archiving is not enabled; you must ensure that all required WAL
segments are copied through other means to complete the backup

                              pg_stop_backup
---------------------------------------------------------------------------
 (0/3000088,"START WAL LOCATION: 0/2000028 (file 000000010000000000000002)+
 CHECKPOINT LOCATION: 0/2000060                                           +
 BACKUP METHOD: streamed                                                  +
 BACKUP FROM: master                                                      +
 START TIME: 2017-10-17 10:00:18 KST                                      +
 LABEL: my_backup                                                         +
 ","17060 /home/agens/AgensGraph/db_cluster/data                          +
 ")
                                                                     (1 row)
```

- pg_is_in_backup()

  pg_is_in_backup 함수는 온라인 exclusive 백업이 진행중이라면 true 값을 반환한다.

```
SELECT pg_is_in_backup();
```

```
Result:
 pg_is_in_backup
-----------------
        t
          (1 row)
```

- pg_backup_start_time()

  pg_backup_start_time 함수는 진행중인 온라인 exclusive 백업이 시작 시간을 반환한다.

```
SELECT pg_backup_start_time();


Result:
  pg_backup_start_time
------------------------
 2017-10-17 10:29:26+09
                 (1 row)
```

- pg_switch_xlog()

  새로운 트랜잭션 로그 파일로 강제 전환한다.(기본적으로 수퍼 유저로 제한되지만 다른 사용자가 이 기능을 실행하려면 EXECUTE 권한 부여가 필요할 수 있다.)

```
SELECT pg_switch_xlog();


Result:
 pg_switch_xlog
----------------
    0/9000120
         (1 row)
```

- pg_xlogfile_name(location pg_lsn)

  pg_xlogfile_name 함수는 트랜잭션 로그의 위치 문자열을 파일명으로 전환한다.

```
SELECT pg_xlogfile_name('0/9000028');


Result:
     pg_xlogfile_name
--------------------------
```

```
 000000010000000000000009
                   (1 row)
```

- pg_xlogfile_name_offset(location pg_lsn)

  pg_xlogfile_name_offset 함수는 트랜잭션 로그의 위치 문자열을 파일명과 파일 내에서 십진수 바이트 오프셋을 변환한다.

```
SELECT pg_xlogfile_name_offset('0/9000028');


Result:
    pg_xlogfile_name_offset
--------------------------------
 (000000010000000000000009,40)
                        (1 row)
```

- pg_xlog_location_diff(location pg_lsn, location pg_lsn)

  pg_xlog_location_diff 함수는 서로 다른 두개의 트랜잭션 로그 위치를 계산한다.

```
SELECT pg_xlog_location_diff('0/9000120', '0/9000028');


Result:
 pg_xlog_location_diff
------------------------
                (1 row)
```

- pg_is_in_recovery()

  pg_is_in_recovery 함수는 복구가 진행중 일때 true 값을 반환한다.

```
SELECT pg_is_in_recovery();


Result:
 pg_is_in_recovery
-------------------
        t
            (1 row)
```

- pg_last_xlog_receive_location()

pg_last_xlog_receive_location 함수는 스트리밍 복제를 통해 디스크로 수신 및 동기화 된 마지막 트랜잭션 로그 위치를 반환한다. 스트리밍 복제를 하는 중이라면 로그는 점차적으로 증가한다. 복구가 완료되면 디스크로 복구하는 동안 수신된 마지막 WAL 레코드의 값에 고정되어 디스크로 동기화 된다. 스트리밍 복제가 사용되지 않거나 아직 시작되지 않은 경우 함수는 NULL을 반환한다.

```
SELECT pg_last_xlog_receive_location();


Result:
 pg_last_xlog_receive_location
-------------------------------


                        (1 row)
```

- pg_last_xlog_replay_location()

  pg_last_xlog_replay_location 함수는 복구 중 재생 된 마지막 트랜잭션 로그 위치를 반환한다. 스트리밍 복제를 하는 중이라면 로그는 점차적으로 증가한다. 복구가 완료되면 이 값은 복구 중에 적용된 마지막 WAL 레코드의 값에 정적으로 유지된다. 복구없이 정상적으로 시작된 경우 NULL을 반환한다.

```
SELECT pg_last_xlog_replay_location();


Result:
 pg_last_xlog_replay_location
------------------------------


                       (1 row)
```

- pg_last_xact_replay_timestamp()

  pg_last_xact_replay_timestamp 함수는 복구 중 재생 된 마지막 트랜잭션의 타임 스템프를 반환한다. 이것은 해당 트랜잭션에 대한 WAL 레코드의 커밋 또는 중단이 발생한 시점이다. 복구 중에 재생 된 트랜잭션이 없으면 NULL을 반환한다. 복구가 아직 진행 중이면 로그는 점차적으로 증가한다. 복구가 완료되면이 값은 해당 복구 중에 적용된 마지막 트랜잭션의 값에서 정적으로 유지된다. 서버가 복구없이 정상적으로 시작된 경우 NULL을 반환합니다.

```
SELECT pg_last_xact_replay_timestamp();


Result:
 pg_last_xact_replay_timestamp
```

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

- pg_export_snapshot()

  pg_export_snapshot 함수는 현재 스냅샷을 저장하고 이것의 식별하는 문자열을 반환한다.

```
SELECT pg_export_snapshot();

Result:
 pg_export_snapshot
--------------------
     00000816-1
          (1 row)
```

- pg_create_physical_replication_slot(slot_name name [, immediately_reserve boolean])

  pg_create_physical_replication_slot 함수는 `slot_name`의 새로운 물리적 복제 slot을 생성한다. 선택적 두 번째 파라미터가 `true`일 때이 복제 슬롯의 LSN을 즉시 예약하도록 지정한다. 그렇지 않으면 스트리밍 복제 클라이언트의 첫 번째 연결에서 LSN이 예약된다. 물리적 슬롯에서 스트리밍 변경은 streaming-replication 프로토콜을 통해서만 가능하다. (링크에 자세히 기술)이 함수는 복제 프로토콜 명령 `CREATE_REPLICATION_SLOT ... PHYSICAL`에 해당한다.

```
SELECT pg_create_physical_replication_slot('test_slot', 'true');

Result:
 pg_create_physical_replication_slot
-------------------------------------
        (test_slot,0/D000220)
                              (1 row)
```

- pg_drop_replication_slot(slot_name name)

  pg_drop_replication_slot 함수는 `slot_name`의 물리적 또는 논리적인 복재 slot을 삭제한다. 복제 프로토콜 `DROP_REPLICATION_SLOT` 명령과 같다.

```
SELECT pg_drop_replication_slot('test_slot');

Result:
```

```
 pg_drop_replication_slot
--------------------------


                   (1 row)
```

- pg_create_logical_replication_slot(slot_name name, plugin name)

  pg_create_logical_replication_slot 함수는 출력 `plugin`을 사용하여 `slot_name`의 새로운 논리적 (디코딩) 복제 slot을 생성한다. 이 함수는 복제 프로토콜 명령 `CREATE_REPLICATION_SLOT ... LOGICAL`과 동일한 효과가 있다.

```
SELECT pg_create_logical_replication_slot('test_slot', 'test_decoding');


Result:
 pg_create_logical_replication_slot
------------------------------------
      (test_slot,0/D000338)
                             (1 row)
```

- pg_logical_slot_get_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])

  pg_logical_slot_get_changes 함수는 마지막으로 변경이 적용된 이후부터 `slot_name` 슬롯의 변경사항을 반환한다. `upto_lsn`과 `upto_nchanges`이 NULL이라면 논리적 디코딩은 WAL의 끝까지 진행된다. `upto_lsn`이 NULL이 아니면 디코딩은 지정된 LSN보다 먼저 커밋한 트랜잭션만 포함한다. `upto_nchanges`가 NULL이 아니면 디코딩으로 생성 된 row의 수가 지정된 값을 초과하면 디코딩이 중지된다. 그러나 이 제한은 새 트랜잭션 커밋을 디코딩 할 때 생성된 행을 추가 한후에만 확인 되기 때문에 반환되는 실제 행 수는 더 클수 있다.

```
SELECT pg_logical_slot_get_changes('regression_slot',null, null);


Result:
  pg_logical_slot_get_changes
--------------------------------
 (0/F000190,2079,"BEGIN 2079")
 (0/F028360,2079,"COMMIT 2079")
                         (2 row)
```

- pg_logical_slot_peek_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])

  pg_logical_slot_peek_changes 함수는 변경사항이 사용되지 않는 것을 제외하고

pg_logical_slot_get_changes() 함수와 동일하게 동작한다.

```
SELECT pg_logical_slot_peek_changes('regression_slot',null, null);

Result:
                        pg_logical_slot_peek_changes
-----------------------------------------------------------------------------
 (0/F028398,2080,"BEGIN 2080")
 (0/F028468,2080,"table public.data: INSERT: id[integer]:1 data[text]:'3'")
 (0/F028568,2080,"COMMIT 2080")
                                                                      (3 row)
```

- pg_logical_slot_get_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])
  pg_logical_slot_get_binary_changes 함수는 변경사항이 bytea로 반환된다는 점을 제외하고 pg_logical_slot_get_changes() 함수와 동일하게 동작한다.

```
SELECT pg_logical_slot_get_binary_changes('regression_slot',null, null);

Result:
               pg_logical_slot_get_binary_changes
---------------------------------------------------------------------
 (0/F028398,2080,"\\x424547494e2032303830")
 (0/F028468,2080,"\\x7461626c65207075626c69632e646174613a20494e5345
 52543a2069645b696e74656765725d3a3120646174615b746578745d3a273327")
 (0/F028568,2080,"\\x434f4d4d49542032303830")
                                                              (3 row)
```

- pg_logical_slot_peek_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])
  pg_logical_slot_peek_binary_changes 함수는 변경사항이 bytea로 반환되고 변경 사항이 사용되지 않는다는 점을 제외하고 pg_logical_slot_get_changes() 함수와 동일하게 동작한다.

```
SELECT pg_logical_slot_peek_binary_changes('regression_slot',null, null);

Result:
               pg_logical_slot_peek_binary_changes
```

```
------------------------------------------------------------------------
 (0/F028398,2080,"\\x424547494e2032303830")
 (0/F028468,2080,"\\x7461626c65207075626c69632e646174613a20494e5345
52543a2069645b696e74656765725d3a3120646174615b746578745d3a273327")
 (0/F028568,2080,"\\x434f4d4d49542032303830")
                                                              (3 row)
```

- pg_replication_origin_create(node_name text)

  pg_replication_origin_create 함수는 주어진 이름으로 복원 원점을 만들고 할당 된 내부 ID를 반환한다.

```
SELECT pg_replication_origin_create('test_decoding: regression_slot');

Result:
 pg_replication_origin_create
------------------------------
                            1
                      (1 row)
```

- pg_replication_origin_drop(node_name text)

  pg_replication_origin_drop* 함수는 재생된 진행 사항을 포함하여 이전에 생성된 복제 원점을 삭제한다.

```
SELECT pg_replication_origin_drop('test_decoding: temp');

Result:
 pg_replication_origin_drop
----------------------------

                    (1 row)
```

- pg_replication_origin_oid(node_name text)

  pg_replication_origin_oid 함수는 명명된 복원 원점을 찾아 내부 id를 반환한다. 상응하는 복원 원점을 찾지 못하면 오류가 발생한다.

```
SELECT pg_replication_origin_oid('test_decoding: temp');

Result:
 pg_replication_origin_oid
```

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

- pg_replication_origin_session_setup(node_name text)

  pg_replication_origin_session_setup 함수는 현재 세션을 주어진 원점에서 재생 진행 사항을 표시하여 이를 추적할 수 있다. 되돌리기 위해서 `pg_replication_origin_session_reset`를 사용한다. 이전의 원점이 구성되지 않은 경우에만 사용이 가능하다.

```
SELECT pg_replication_origin_session_setup('test_decoding: regression_slot');

Result:
 pg_replication_origin_session_setup
-------------------------------------

                              (1 row)
```

- pg_replication_origin_session_reset()

  pg_replication_origin_session_reset 함수는 `pg_replication_origin_session_setup()`의 설정을 취소한다.

```
SELECT pg_replication_origin_session_reset();

Result:
 pg_replication_origin_session_reset
-------------------------------------

                              (1 row)
```

- pg_replication_origin_session_is_setup()

  pg_replication_origin_session_is_setup 함수는 현재 세션에서 복제 원본이 구성되 었는지 여부를 반환한다.

```
SELECT pg_replication_origin_session_is_setup();

Result:
 pg_replication_origin_session_is_setup
----------------------------------------
                  t
                                 (1 row)
```

- pg_replication_origin_session_progress(flush bool)

  pg_replication_origin_session_progress 함수는 현재 세션에서 복제 원본의 재생 위치를 반환한다. `flush` 파라미터는 해당 로컬 트랜잭션이 디스크로 flush 되었는지 여부를 결정한다.

```
SELECT pg_replication_origin_session_progress(false);


Result:
 pg_replication_origin_session_progress
----------------------------------------
               0/AABBCCDD
                                    (1 row)
```

- pg_replication_origin_xact_setup(origin_lsn pg_lsn, origin_timestamp timestamptz) pg_replication_origin_xact_setup 함수는 현재 트랜잭션을 지정된 LSN과 타임스템프로 커밋된 트랜잭션을 재생하는 것으로 표시한다. 이전에 복원 원점을 사용하여 구성한 경우에만 `pg_replication_origin_session_setup()`을 사용하여 설정하여 호출 할 수 있다.

```
SELECT pg_replication_origin_xact_setup('0/AABBCCDD', '2017-01-01 00:00');


Result:
 pg_replication_origin_xact_setup
----------------------------------


                             (1 row)
```

- pg_replication_origin_xact_reset()

  pg_replication_origin_xact_reset 함수는 `pg_replication_origin_xact_setup()`의 설정을 취소한다.

```
SELECT pg_replication_origin_xact_reset();


Result:
 pg_replication_origin_xact_reset
----------------------------------


                             (1 row)
```

- pg_replication_origin_advance(node_name text, pos pg_lsn)

pg_replication_origin_advance 함수는 주어진 노드에 대한 복제 진행을 주어진 위치로 설정한다. 이 기능은 주로 초기 위치 또는 구성 변경 후 새 위치를 설정하는 것 등에 유용하다. 이 기능을 부주의하게 사용하면 데이터가 일관성없이 복제 될 수 있다.

```
SELECT pg_replication_origin_advance('test_decoding: regression_slot', '0/1');


Result:
 pg_replication_origin_advance
-------------------------------


                        (1 row)
```

- pg_replication_origin_progress(node_name text, flush bool)

  pg_replication_origin_progress 함수는 지정된 복제 원점의 재생 위치를 반환한다. `flush`파라미터는 해당 로컬 트랜잭션이 디스크로 flush 되었는지 여부를 결정한다.

```
SELECT pg_replication_origin_progress('test_decoding: temp', true);


Result:
 pg_replication_origin_progress
--------------------------------
           0/AABBCCDD
                         (1 row)
```

- pg_logical_emit_message(transactional bool, prefix text, content text)

  텍스트 논리적 디코딩 메세지를 보낸다. 이것은 WAL을 통해 논리 디코딩 플러그인에 일반 메세지를 전달하는데 사용할 수 있다. `transactional` 파라미터는 메세지가 현재 트랜잭션의 일부인지, 논리 디코딩이 레코드를 읽자마자 즉시 기록되고 디코딩되어야 하는지 여부를 판단한다. `prefix`는 논리적 디코딩 플러그인을 사용자가 쉽게 인식할 수 있도록 논리적 디코딩 플러그인이 사용하는 텍스트 형식의 prefix이다. `content`는 메세지의 텍스트이다.

```
SELECT pg_logical_emit_message(false, 'test', 'this message will not be decoded');


Result:
 pg_logical_emit_message
-------------------------
        0/F05E1D0
```

```
(1 row)
```

- pg_logical_emit_message(transactional bool, prefix text, content bytea)

  이진 논리적 디코딩 메세지를 보낸다. 이것은 WAL을 통해 논리 디코딩 플러그인에 일반 메세지를 전달하는데 사용할 수 있다. transactional 파라미터는 메세지가 현재 트랜잭션의 일부인지, 논리 디코딩이 레코드를 읽자마자 즉시 기록되고 디코딩되어야 하는지 여부를 판단한다. prefix는 논리적 디코딩 플러그인을 사용자가 쉽게 인식할 수 있도록 논리적 디코딩 플러그인이 사용하는 텍스트 형식의 prefix이다. content는 메세지의 바이너리 콘텐츠이다.

```
SELECT pg_logical_emit_message(false, 'test', '0/F05E1D0');


Result:
 pg_logical_emit_message
--------------------------
        0/F05E2C8
                  (1 row)
```

- pg_column_size(any)

  pg_column_size 함수는 특정 값을 저장하는데 사용된 byte의 수를 반환한다.

```
SELECT pg_column_size('SELECT fooid FROM foo');


Result:
 pg_column_size
----------------
             22
         (1 row)
```

- pg_database_size(oid)

  pg_database_size 함수는 지정된 OID를 가진 데이터베이스에서 사용한 disk 공간을 반환한다.

```
SELECT pg_database_size(16482);


Result:
 pg_database_size
------------------
          9721508
           (1 row)
```

- pg_database_size(name)

  이 함수는 지정된 이름을 가진 데이터베이스에서 사용한 disk 공간을 반환한다.

```
SELECT pg_database_size('test');


Result:
 pg_database_size
------------------
          9721508
           (1 row)
```

- pg_indexes_size(regclass)

  pg_indexes_size 함수는 지정된 테이블에 추가된 인덱스가 사용한 총 disk 공간을 반환한다.

```
SELECT pg_indexes_size(2830);


Result:
 pg_indexes_size
-----------------
            8192
          (1 row)
```

- pg_relation_size(relation regclass, fork text)

  pg_relation_size 함수는 지정된 테이블 또는 인덱스의 특정 fork(`main', `fsm', `vm', or `init')에 의해 사용된 disk 공간을 반환한다.

```
SELECT  pg_relation_size(16881, 'main');


Result:
 pg_relation_size
------------------
                0
           (1 row)
```

- pg_relation_size(relation regclass)

  이 함수는 `pg_relation_size(..., 'main')`의 단축형이다.

```
SELECT  pg_relation_size(16881);


Result:
 pg_relation_size
------------------
                0
           (1 row)
```

- pg_size_bytes(text)

  pg_size_bytes 함수는 사람이 읽을 수 있는 크기 단위인 byte 단위로 변환한다.

```
SELECT pg_size_bytes('100');


Result:
 pg_size_bytes
---------------
          100
        (1 row)
```

- pg_size_pretty(bigint)

  pg_size_pretty 함수는 64비트 정수로 표현된 바이트 단위의 크기를 사람이 읽을 수 있는 형식으로 크기 단위를 반환한다.

```
SELECT pg_size_pretty(10::bigint);


Result:
 pg_size_pretty
----------------
       10 bytes
         (1 row)
```

- pg_size_pretty(numeric)

  이 함수는 숫자로 표현된 바이트 단위의 크기를 사람이 읽을 수 있는 형식으로 크기 단위를 반환한다.

```
SELECT pg_size_pretty(10::numeric);


Result:
```

```
 pg_size_pretty
----------------
       10 bytes
         (1 row)
```

- pg_table_size(regclass)

  pg_table_size 함수는 인덱스를 제외한 특정 테이블에서 사용한 disk 공간을 반환한다.(TOAST, 여유공간 map, visibility map은 포함)

```
SELECT pg_table_size('myschema.mytable');

Result:
 pg_table_size
---------------
          8192
        (1 row)
```

- pg_tablespace_size(oid)

  pg_tablespace_size 함수는 특정 OID를 가진 tablespace에서 사용한 disk 공간을 반환한다.

```
SELECT pg_tablespace_size(1663);

Result:
 pg_tablespace_size
--------------------
           40859636
             (1 row)
```

- pg_tablespace_size(name)

  pg_tablespace_size 함수는 특정 이름을 가진 tablespace에서 사용한 disk 공간을 반환한다.

```
SELECT pg_tablespace_size('pg_default');

Result:
 pg_tablespace_size
--------------------
           40859636
             (1 row)
```

- pg_total_relation_size(regclass)

  pg_total_relation_size 함수는 모든 index와 TOAST 데이터를 포함한 특정 테이블이 사용한 총 disk 공간을 반환한다.

```
SELECT  pg_total_relation_size(16881);


Result:
 pg_total_relation_size
------------------------
                   8192
                 (1 row)
```

- pg_relation_filenode(relation regclass)

  pg_relation_filenode 함수는 지정된 복제 filenode 번호를 반환한다.

```
SELECT pg_relation_filenode('pg_database');


Result:
 pg_relation_filenode
----------------------
                 1262
               (1 row)
```

- pg_relation_filepath(relation regclass)

  pg_relation_filepath 함수는 지정된 복제 파일 경로 이름을 반환한다.

```
SELECT pg_relation_filepath('pg_database');


Result:
 pg_relation_filepath
----------------------
          global/1262
               (1 row)
```

- pg_filenode_relation(tablespace oid, filenode oid)

  pg_filenode_relation 함수는 주어진 테이블스페이스 공간 및 filenode와 관련된 관계를 찾는다.

```
SELECT pg_filenode_relation(1663, 16485);


Result:
 pg_filenode_relation
----------------------
  test.ag_label_seq
             (1 row)
```

- brin_summarize_new_values(index regclass)

  brin_summarize_new_values 함수는 이전에 요약되지 않은 페이지 범위를 요약한다.

```
SELECT brin_summarize_new_values('brinidx');


Result:
 brin_summarize_new_values
---------------------------
                         0
                  (1 row)
```

- gin_clean_pending_list(index regclass)

  gin_clean_pending_list 함수는 GIN 보류 목록 항목을 기본 인덱스 구조로 이동한다.

```
SELECT gin_clean_pending_list('gin_test_idx');


Result:
 gin_clean_pending_list
------------------------
                      0
               (1 row)
```

- pg_ls_dir(dirname text [, missing_ok boolean, include_dot_dirs boolean])

  pg_ls_dir 함수는 경로의 내용을 나열한다.

```
SELECT pg_ls_dir('.');


Result:
      pg_ls_dir
```

```
----------------------
 pg_xlog
 global
 ...
             (28 row)
```

- pg_read_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])

  pg_read_file 함수는 텍스트 파일의 내용을 반환한다.

```
SELECT pg_read_file('test.sql');


Result:
 pg_read_file
--------------
 test        +
      (1 row)
```

- pg_read_binary_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])

  pg_read_binary_file 함수는 파일의 내용을 반환한다.

```
SELECT pg_read_binary_file('test');


Result:
 pg_read_binary_file
---------------------
      x6161610a
             (1 row)
```

- pg_stat_file(filename text[, missing_ok boolean])

  pg_stat_file 함수는 파일에 대한 정보를 반환한다.

```
SELECT pg_stat_file('test');


Result:
                                   pg_stat_file
-----------------------------------------------------------------------------------
 (4,"2017-10-18 11:05:09+09","2017-10-18 11:04:55+09","2017-10-18 11:04:55+09",,f)
                                                                             (1 row)
```

- pg_advisory_lock(key bigint)

  pg_advisory_lock 함수는 exclusive 세션 레벨의 advisory lock을 획득한다.

```
SELECT pg_advisory_lock(1);
SELECT locktype, classid, objid, mode FROM pg_locks where objid=1;

Result:
 locktype | classid | objid |     mode
----------+---------+-------+---------------
 advisory |       0 |     1 | ExclusiveLock
                                      (1 row)
```

- pg_advisory_lock(key1 int, key2 int)

  이 함수는 exclusive 세션 레벨의 advisory lock을 획득한다.

```
SELECT pg_advisory_lock(1,2);
SELECT locktype, classid, objid, mode FROM pg_locks where objid=2;

Result:
 locktype | classid | objid |     mode
----------+---------+-------+---------------
 advisory |       1 |     2 | ExclusiveLock
                                      (1 row)
```

- pg_advisory_lock_shared(key bigint)

  pg_advisory_lock_shared 함수는 shared 세션 레벨의 advisory lock을 획득한다.

```
SELECT pg_advisory_lock_shared(10);
SELECT locktype, classid, objid, mode FROM pg_locks where objid=10;

Result:
 locktype | classid | objid |   mode
----------+---------+-------+-----------
 advisory |       0 |    10 | ShareLock
                                  (1 row)
```

- pg_advisory_lock_shared(key1 int, key2 int)

  이 함수는 shared 세션 레벨의 advisory lock을 획득한다.

```
SELECT pg_advisory_lock_shared(10,20);
SELECT locktype, classid, objid, mode FROM pg_locks where objid=20;

Result:
 locktype | classid | objid |   mode
----------+---------+-------+-----------
 advisory |      10 |    20 | ShareLock
                                 (1 row)
```

- pg_advisory_unlock(key bigint)

  pg_advisory_unlock 함수는 exclusive 세션 레벨의 advisory lock을 해제한다.

```
SELECT pg_advisory_unlock(1);

Result:
 pg_advisory_unlock
--------------------
         t
             (1 row)
```

- pg_advisory_unlock(key1 int, key2 int)

  이 함수는 exclusive 세션 레벨의 advisory lock을 해제한다.

```
SELECT pg_advisory_unlock(1,2);

Result:
 pg_advisory_unlock
--------------------
         t
             (1 row)
```

- pg_advisory_unlock_all()

  pg_advisory_unlock_all 함수는 현재 세션에서 보유하고 있는 모든 세션 레벨의 advisory lock을 해제한다.

```
SELECT pg_advisory_unlock_all();

Result:
```

```
 pg_advisory_unlock_all
-------------------------


                 (1 row)
```

- pg_advisory_unlock_shared(key bigint)

  pg_advisory_unlock_shared 함수는 share 세션 레벨의 advisory lock을 해제한다.

```
SELECT pg_advisory_unlock_shared(10);


Result:
 pg_advisory_unlock_shared
----------------------------
             t
                    (1 row)
```

- pg_advisory_unlock_shared(key1 int, key2 int)

  이 함수는 share 세션 레벨의 advisory lock을 해제한다.

```
SELECT pg_advisory_unlock_shared(10,20);


Result:
 pg_advisory_unlock_shared
----------------------------
             t
                    (1 row)
```

- pg_advisory_xact_lock(key bigint)

  pg_advisory_xact_lock 함수는 exclusive 트랜잭션 레벨 advisory lock을 획득한다.

```
SELECT pg_advisory_xact_lock(1);


Result:
 pg_advisory_xact_lock
-------------------------


                 (1 row)
```

- pg_advisory_xact_lock(key1 int, key2 int)

  이 함수는 exclusive 트랜잭션 레벨 advisory lock을 획득한다.

```
SELECT pg_advisory_xact_lock(1,2);


Result:

 pg_advisory_xact_lock

------------------------



                 (1 row)
```

- pg_advisory_xact_lock_shared(key bigint)

  pg_advisory_xact_lock_shared 함수는 share 세션 레벨의 advisory lock을 획득한다.

```
SELECT pg_advisory_xact_lock_shared(10);


Result:

 pg_advisory_xact_lock_shared

------------------------------



                       (1 row)
```

- pg_advisory_xact_lock_shared(key1 int, key2 int)

  이 함수는 share 세션 레벨의 advisory lock을 획득한다.

```
SELECT pg_advisory_xact_lock_shared(10,20);


Result:

 pg_advisory_xact_lock_shared

------------------------------



                       (1 row)
```

- pg_try_advisory_lock(key bigint)

  pg_try_advisory_lock 함수는 가능한 경우 exclusive 트랜잭션 레벨 advisory lock을 획득한다.

```
SELECT pg_try_advisory_lock(100);


Result:
 pg_try_advisory_lock
----------------------
          t
               (1 row)
```

- pg_try_advisory_lock(key1 int, key2 int)

  이 함수는 가능한 경우 exclusive 트랜잭션 레벨 advisory lock을 획득한다.

```
SELECT pg_try_advisory_lock(100,200);


Result:
 pg_try_advisory_lock
----------------------
          t
               (1 row)
```

- pg_try_advisory_lock_shared(key bigint)

  pg_try_advisory_lock_shared 함수는 가능한 경우 share 세션 레벨의 advisory lock을 획득한다.

```
SELECT pg_try_advisory_lock_shared(1000);


Result:
 pg_try_advisory_lock_shared
-----------------------------
              t
                     (1 row)
```

- pg_try_advisory_lock_shared(key1 int, key2 int)

  이 함수는 가능한 경우 share 세션 레벨의 advisory lock을 획득한다.

```
SELECT pg_try_advisory_lock_shared(1000,2000);


Result:
 pg_try_advisory_lock_shared
```

```
-----------------------------
              t
                      (1 row)
```

- pg_try_advisory_xact_lock(key bigint)

  pg_try_advisory_xact_lock 함수는 가능한 경우 exclusive 트랜잭션 레벨 advisory lock을 획득한다.

```
SELECT pg_try_advisory_xact_lock(1000);


Result:
 pg_try_advisory_xact_lock
---------------------------
             t
                    (1 row)
```

- pg_try_advisory_xact_lock(key1 int, key2 int)

  이 함수는 가능한 경우 exclusive 트랜잭션 레벨 advisory lock을 획득한다.

```
SELECT pg_try_advisory_xact_lock(1000,2000);


Result:
 pg_try_advisory_xact_lock
---------------------------
             t
                    (1 row)
```

- pg_try_advisory_xact_lock_shared(key bigint)

  pg_try_advisory_xact_lock_shared 함수는 가능한 경우 share 트랜잭션 레벨 advisory lock을 획득한다.

```
SELECT pg_try_advisory_xact_lock_shared(10000);


Result:
 pg_try_advisory_xact_lock_shared
----------------------------------
                t
                           (1 row)
```

- pg_try_advisory_xact_lock_shared(key1 int, key2 int)

이 함수는 가능한 경우 share 트랜잭션 레벨 advisory lock을 획득한다.

```
SELECT pg_try_advisory_xact_lock_shared(10000,20000);

Result:
 pg_try_advisory_xact_lock_shared
----------------------------------
                t
                           (1 row)
```

4.3.19 User-defined function

AgensGraph에서는 사용자가 원하는 함수를 생성하여 사용할 수 있는 기능을 제공한다.

AgensGraph는 SQL과 Cypher 쿼리문을 동시에 사용 가능하기 때문에 이를 이용하여 쉽게 함수를 생성할 수 있으며, 생성된 Function은 `\df` 명령으로 확인이 가능하다. 또한 생성된 함수는 SELECT 또는 RETURN 구문을 이용하여 호출 할 수 있다.

사용자 정의 함수 생성 및 문법들은 PostgreSQL 문서에서 자세한 내용을 확인할 수 있다.

- User-defined function

```
CREATE FUNCTION func_name (integer, integer) RETURNS integer
    AS 'SELECT $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;

SELECT func_name (1, 1);

Result:
 add
-----
   2
(1 row)

RETURN func_name (1, 1);

Result:
```

```
 add
-----
   2
(1 row)


DROP FUNCTION func_name (integer, integer);
```

5 Hybrid Query Language

5.1 Introduction

이번 절에서는 아래의 예제와 같이 SQL 쿼리문과 Cypher 쿼리문을 함께 사용하는 방법을 설명한다.
RDB에서 사용하는 SQL문을 통해 테이블과 컬럼 단위의 집계, 통계 처리와 함께 GDB에서 사용하는 Cypher구문을 통해 RDB의 Join연산을 대체하여 좀 더 성능이 좋은 데이터 쿼리를 지원한다.

```
CREATE GRAPH bitnine;
CREATE VLABEL dev;
CREATE (:dev {name: 'someone', year: 2015});
CREATE (:dev {name: 'somebody', year: 2016});

CREATE TABLE history (year, event)
AS VALUES (1996, 'PostgreSQL'), (2016, 'AgensGraph');
```

5.2 Syntax

5.2.1 Cypher in SQL

그래프DB에 저장된 vertex 또는 edge의 데이터 셋을 SQL문 안에서의 데이터 집합 셋으로 활용하기 위해 FROM절 내부에 Cypher구문 사용이 가능하다.
Syntex :

```
SELECT [column_name]
FROM ({table_name|SQLquery|CYPHERquery})
WHERE [column_name operator value];
```

다음의 예제와 같이 사용 가능하다.

```
SELECT n->>'name' as name
FROM history, (MATCH (n:dev) RETURN n) as dev
WHERE history.year > (n->>'year')::int;

  name
---------
```

```
 someone
(1 row)
```

5.2.2 SQL in Cypher

Cypher 구문을 통한 그래프DB의 내용을 조회 시 RDB의 특정 데이터를 활용하여 조회하고자 할때 Match구문, Where 구문 등 모두 활용 가능하다. 단, SQL구문의 결과 데이터 셋은 단일행의 결과값을 되돌려 주도록 구성해야 한다.

Syntex :

```
MATCH [table_name]
WHERE (column_name operator {value|SQLquery|CYPHERquery})
RETURN [column_name];
```

다음의 예제와 같이 사용 가능하다.

```
MATCH (n:dev)
WHERE n.year < (SELECT year FROM history WHERE event = 'AgensGraph')
RETURN properties(n) AS n;

                  n
-----------------------------------
 {"name": "someone", "year": 2015}
(1 row)
```

6 Drivers

6.1 Introduction

AgensGraph는 client driver를 이용한 DB 접속을 지원하며, Bitnine에서 공식 제공하는 드라이버 및 기존 PostgreSQL 드라이버를 이용하여 다양한 언어 사용이 가능하다.

AgensGraph는 JDBC 및 Python 드라이버를 공식 제공한다.

6.2 Usage of the Java Driver

AgensGraph의 그래프 데이터를 Java application에서 어떻게 처리하고 다루는지에 대해 설명한다.

AgensGraph의 JDBC 드라이버는 PostgreSQL의 JDBC 드라이버를 기반으로 하며, Java application에서 AgensGraph database에 접근하는 할 수 있도록 한다. AgensGraph Java 드라이버와 Postgres JDBC 드라이버의 API는 매우 비슷하다. 단 한가지 차이점은 AgensGraph JDBC 드라이버는 SQL 뿐만이 아니라 Cypher 질의 언어를 사용할 수 있으며, 그래프 데이터 (vertices, edges and paths)를 데이터 타입으로 활용할 수 있다는 것이다.

이 섹션에서는 예를 통해 AgensGraph JDBC Driver를 사용하는 방법을 보여준다.

6.2.1 Get the Driver

AgensGraph JDBC Download에서 드라이버 (jar)를 다운로드한다.

```
<dependency>
  <groupId>net.bitnine</groupId>
  <artifactId>agensgraph-jdbc</artifactId>
  <version>1.4.2</version>
</dependency>
```

6.2.2 Connection

Java 드라이버를 사용하여 AgensGraph를 접속하기 위해 2가지를 고려해야 한다. Java 드라이버로 로드되기 위한 class name과 connection string이다.

- class name : `net.bitnine.agensgraph.Driver`.
- sub-protocol, server, port, database로 구성된 connection string.
  - sub-protocol : `jdbc:agensgraph://`.
  - connection string(sub-protocol을 포함) : `jdbc:agensgraph://server:port/database`.

다음 코드는 AgensGraph에 접속하는 한 예이다. Connection object를 통해 AgensGraph에 접속한다.

```
import java.sql.DriverManager;
import java.sql.Connection;

public class AgensGraphTest {
  public static void main(String[] args) {
  try{
        Class.forName("net.bitnine.agensgraph.Driver");
        Connection conn = DriverManager.getConnection
        ("jdbc:agensgraph://127.0.0.1:5432/agens","agens","agens");
        System.out.println("connection success");
     } catch (Exception e) {
          e.printStackTrace();
          }
  }
}
```

6.2.3 Retrieving Data

MATCH절을 이용하여 AgensGraph 내 그래프 데이터를 조회하는 방법에 대해 설명한다.

쿼리의 결과는 AgensGraph의 vertex 이다. 결과를 vertex 객체로 가져 와서 속성을 얻을 수 있다.

```
import net.bitnine.agensgraph.util.*;
import java.sql.*;
import net.bitnine.agensgraph.graph.Vertex;

public class AgensGraphSelect {
      public static void main(String[] args) throws SQLException, ClassNotFoundException {

        Class.forName("net.bitnine.agensgraph.Driver");
        Connection conn = DriverManager.getConnection
        ("jdbc:agensgraph://127.0.0.1:5432/agens","agens","agens");

        try{
            Statement stmt = conn.createStatement();
```

```
            ResultSet rs = stmt.executeQuery
            ("MATCH (:person {name: 'John'})-[:knows]-(friend:person) RETURN friend");
            while (rs.next()) {
              Vertex person = (Vertex) rs.getObject(1);
              System.out.println(person.getString("name"));
                      }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

6.2.4 Creating Data

AgensGraph로 그래프 데이터를 insert하는 방법에 대해 설명한다.

아래 예는 Person이라는 vlabel을 가진 vertex를 생성하는 코드이다. 해당 vertex의 property를 추가하기 위해 JsonObject를 사용하였다.

```
import net.bitnine.agensgraph.util.*;
import java.sql.*;

public class AgensGraphCreate {
      public static void main(String[] args) throws SQLException, ClassNotFoundException {

        Class.forName("net.bitnine.agensgraph.Driver");
        Connection conn = DriverManager.getConnection
        ("jdbc:agensgraph://127.0.0.1:5432/agens","agens","agens");

        try{
            PreparedStatement pstmt = conn.prepareStatement("CREATE (:person ?)");
            Jsonb j = JsonbUtil.createObjectBuilder()
                             .add("name", "John")
                             .add("from", "USA")
                             .add("age", 17)
                             .build();
            pstmt.setObject(1, j);
```

```
            pstmt.execute();
        } catch(Exception e) {
            e.printStackTrace();
        }
        finally{
            conn.close();
        }
    }
}
```

최종적으로 만들어지는 문자열은 다음과 같다:

```
CREATE (:Person {name: 'John', from: 'USA', age: 17})
```

> 참고
> JDBC에서 물음표 (?)는 PreparedStatement의 위치 매개 변수에 대한 자리 표시이다. 그러나 다른 sql 연산자에도 사용되고 있으므로 혼란을 피하기 위해 물음표 (?)를 PreparedStatement에 사용 할 때는 문자에 붙이지 말고 띄어서 사용해야 한다.
> ex) conn.prepareStatement(``create ({name ? : ?});''

6.2.5 Graph Object Classes

Class	Description
GraphId	AgensGraph graphid type에 해당하는 java class 이다.
Vertex	AgensGraph vertex type에 해당하는 java class 이다.
	label과 properties에 대한 액세스 방법을 지원한다.
Edge	AgensGraph edge type에 해당하는 java class 이다.
	label과 properties에 대한 액세스 방법을 지원한다.
Path	AgensGraph graphpath type에 해당하는 java class 이다.
	Path 길이와 path의 vertex 및 edge에 대한 액세스 방법을 지원한다.
Jsonb	AgensGraph jsonb type에 해당하는 java class 이다.
	Vertex와 Edge는 Jsonb를 사용하여 속성을 저장한다.
	JSON scalar, array, object type에 대한 액세스 방법을 지원한다.
JsonbUtil	위의 CREATE 예제와 같이 Jsonb 객체를 만드는 다양한 방법을 제공한다.

6.3 Usage of the Python Driver

이 섹션에서는 예를 통해 AgensGraph Python Driver를 사용하는 방법을 보여준다. AgensGraph를위한 Psycopg2 타입 확장 모듈이며, 그래프 데이터를 표현하기 위해 Vertex, Edge, Path와 같은 추가 데이터 타입을 지원한다.

6.3.1 Get the Driver

AgensGraph Python 드라이버는 비트나인 홈페이지 Download - Developer Resources 또는 깃허브 agensgraph-python로 다운로드한다.

```
git clone https://github.com/bitnine-oss/agensgraph-python.git
```

6.3.2 Install

```
$ pip install -U pip
$ pip install psycopg2

$ python /path/to/agensgraph/python/setup.py install
```

6.3.3 Connection

Python 드라이버를 사용하여 AgensGraph를 접속하기 위한 connection string의 한 예이다.

```
import psycopg2
import agensgraph

conn = psycopg2.connect("dbname=agens host=127.0.0.1 user=agens")
print "Opened database successfully"
```

6.3.4 Creating Data

AgensGraph로 그래프 데이터를 insert하는 방법에 대해 설명한다. 아래 예는 ``Person''이라는 vlabel을 가진 vertex 를 생성하는 코드이다.

```
import psycopg2
import agensgraph

conn = psycopg2.connect("dbname=agens host=127.0.0.1 user=agens")
print "Opened database successfully"
```

```
cur = conn.cursor()

cur.execute("DROP GRAPH IF EXISTS test CASCADE")
cur.execute("CREATE GRAPH test")
cur.execute("SET graph_path = test")
cur.execute("CREATE (:person {name: 'John', from: 'USA', age: 17})")
cur.execute("CREATE (:person {name: 'Daniel', from: 'Korea', age: 20})")
cur.execute("MATCH (p:person {name: 'John'}),(k:person{name: 'Daniel'}) CREATE (p)-[:knows]->(k)")

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

6.3.5 Retrieving Data

MATCH절을 이용하여 AgensGraph 내 그래프 데이터를 조회하는 방법에 대해 설명한다. 쿼리의 결과는 AgensGraph 의 vertex 이다. 결과를 vertex 객체로 가져 와서 속성을 얻을 수 있다.

```
import psycopg2
import agensgraph

conn = psycopg2.connect("dbname=agens host=127.0.0.1 user=agens")
print "Opened database successfully"

cur = conn.cursor()
cur.execute("SET graph_path = test")
cur.execute("MATCH (:person {name: 'John'})-[:knows]-(friend:person) RETURN friend")

friend = cur.fetchone()
print str(friend[0])

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

7 Procedural language

7.1 Procedural language

AgensGraph는 SQL과 C 이외의 다른 언어로도 사용자 정의 함수를 작성할 수 있다. 이러한 다른 언어들을 일반적으로 procedural languages(PLs)하고 한다. Procedural language로 작성된 함수의 경우 데이터베이스 서버는 함수의 소스 텍스트를 해석할 수 없다. 대신 작업은 언어의 세부 정보를 알고 있는 특수 처리기(handler)로 전달된다. 처리기는 parsing, 구문 분석, 실행 등 모든 작업을 자체적으로 수행한다. 처리기 자체는 다른 C함수와 마찬가지로 공유 객체로 컴파일되고 필요할 때 로드되는 C언어 함수이다. 현재 AgensGraph에는 PL/pgSQL, PL/Tcl, PL/Perl 및 PL/Python의 네 가지 절차 언어가 있다.

7.1.1 Installing Procedural Languages

Procedural Language는 사용될 각 데이터베이스에 설치 되어야 한다. 그러나 template1 데이터베이스에 설치된 procedural Language는 template1의 항목이 CREATE DATABASE에 의해 복사되므로 이후에 작성되는 모든 데이터베이스에서 자동으로 사용 가능하다. 따라서 데이터베이스 관리자는 어떤 데이터베이스에서 어떤 언어를 사용할 수 있는지 결정할 수 있으며 원하는 경우 일부 언어만을 사용할 수 있다.

표준 배포본과 함께 제공되는 언어의 경우, 현재 데이터베이스에 언어를 설치하기 위해 `CREATE EXTENSION` *language_name*을 실행하기만 하면 된다. 또는 프로그램 createlang을 사용하여 쉘 명령행에서 이 작업을 수행 할 수 있다. 예를 들어 PL/Python이라는 언어를 `template1` 데이터베이스에 설치하려면 다음 예제를 참고한다.

```
createlang plpython template1
```

아래 설명된 수동 procedure는 extension으로 패키지되지 않은 언어를 설치하는 경우에만 권장한다.

Manual Procedural Language Installation

Procedural Language는 5단계로 데이터베이스에 설치되며, 데이터베이스 슈퍼유저가 수행해야 한다. 대부분의 경우 필요한 SQL명령은 extension의 설치 스크립트로 패키지화하여 CREATE EXTENSION으로 이를 실행할 수 있다.

1. 언어 처리기(Language Handler)의 공유 오브젝트는 컴파일되어 적절한 라이브러리 디렉토리에 설치 되어야 한다. 이것은 일반 사용자 정의 C함수로 모듈을 빌드하고 설치하는 것과 같은 방식으로 작동한다. 종종 언어 처리기는 실제 프로그래밍 언어 엔진을 제공하는 외부 라이브러리에 의존한다. 만일 그렇다면 반드시 설치가 되어야 한다.

2. 처리기는 반드시 명령어로 선언되어야 한다.

```
CREATE FUNCTION handler_function_name()
    RETURNS language_handler
    AS 'path-to-shared-object'
    LANGUAGE C;
```

language_handler의 특별한 return 타입은 이 함수가 정의된 SQL데이터 유형 중 하나를 리턴하지 않고 SQL 문에서 직접 사용할 수 없다는 것을 데이터베이스 시스템에 알린다.

3. 선택적으로 언어 처리기는 이 언어로 작성된 익명 코드 블록(DO 명령)을 실행하는 ``inline'' 처리기 기능을 제공 할 수 있다. inline 핸들러 함수가 언어에 의해 제공되는 경우 다음과 같은 명령으로 선언해야 한다.

```
CREATE FUNCTION inline_function_name(internal)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C;
```

4. 선택적으로 언어 처리기는 함수 정의를 실제로 실행하지 않고 정확성을 검사하는 ``유효성 검사기'' 기능을 제공 할 수 있다. 유효성 검사기 함수는 CREATE FUNCTION에 의해 호출된다. validator 함수가 언어에 의해 제공되면, 다음과 같은 명령으로 선언해야 한다.

```
CREATE FUNCTION validator_function_name(oid)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C STRICT;
```

5. 마지막으로 PL은 다음 명령으로 선언해야 한다.

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language-name
    HANDLER handler_function_name
    [INLINE inline_function_name]
    [VALIDATOR validator_function_name] ;
```

선택적 키워드 TRUSTED는 사용자가 사용하지 않을 데이터에 대한 액세스 권한을 언어가 부여하지 않도록 지정한다. 신뢰할 수 있는 언어는 일반 데이터베이스 사용자(슈퍼유저 권한이 없는 사용자)를 위해 설계되었으며 함수 및 트리거 절차를 안전하게 만들 수 있다. PL함수는 데이터베이스 서버 내에서 실행되므로, TRUSTED 플래그는 데이터베이스 서버 내부 또는 파일 시스템에 대한 액세스를 허용하지 않는 언어에 대해서만 제공되어야 한다. PL/pgSQL, PL/Tcl 및 PL/Perl 언어는 신뢰할 수있는 것으로 간주된다. 언어 PL/TclU, PL/PerlU 및 PL/PythonU는 무제한 기능을 제공하도록 설계되었으므로 신뢰할 수 있는 것으로 표시하면 안된다.

기본 AgensGraph 설치에서 PL/pgSQL 언어의 처리기가 빌드되어 ``library'' 디렉토리에 설치된다. 또한 PL/pgSQL 언어 자체가 모든 데이터베이스에 설치된다. Tcl support가 구성되어 있으며, PL/Tcl 및 PL/TclU용 핸들러가 빌드되어 라이브러리 디렉토리에 설치되어 있어도 언어 자체는 기본적으로 데이터베이스에 설치되지 않는다. 마찬가지로, Perl support가 구성되어 있으며, PL/Perl 및 PL/PerlU 처리기가 빌드 및 설치되어 있고, Python support가 구성되어 있으며 PL/PythonU 처리기가 설치되어 있어도 기본적으로 이들 언어는 설치되지 않는다.

7.2 PL/pgSQL

7.2.1 소개

PL/pgSQL은 AgensGraph에서 로드가능한 프로시저 언어다. PL/pgSQL의 설계 목적은 다음과 같은 특징을 가지는 로드가능한 절차적 언어다.

- 함수와 트리거 프로시저를 만들기 위해 사용할 수 있다.
- SQL 언어에 제어 구조를 추가한다.
- 복잡한 연산을 수행 할 수 있다.
- 모든 사용자 정의 타입, 함수, 연산자를 상속한다.
- 서버가 신뢰할 수 있게 정의할 수 있다.
- 사용하기 쉽다.

PL/pgSQL로 만든 함수는 내장 함수가 사용될 수 있는 곳이라면 어디에서든 사용할 수 있다. 예를들어, 복잡한 조건을 처리하는 함수를 만들고, 그 함수를 나중에 연산자를 정의하거나 인덱스 표현에서 사용할 수 있다.
AgensGraph에서 PL/pgSQL은 기본으로 설치된다. 하지만 여전히 로드가능한 모듈이라서, 보안을 엄격하게 생각하는 관리자는 PL/pgSQL을 제거할 수 있다.

PL/pgSQL의 장점

SQL은 데이터베이스에서 질의 언어로 사용하는 언어다. SQL은 배우기 쉽긴 하지만, 모든 SQL구문은 데이터베이스 서버에서 반드시 구문마다 개별적으로 실행된다.
다시 말하자면 클라이언트 애플리케이션은 데이터베이스 서버에 반드시 쿼리를 개별적으로 보내고, 각 쿼리가 처리될 때까지 기다린 다음, 결과를 받아서 연산을 한 후에 다음 쿼리를 서버로 보낸다. 이런 과정들은 내부 처리 과정을 발생시키고, 데이터베이스와 클라이언트가 다른 장비에 있다면 네트워크 부하도 일으킨다.
PL/pgSQL은 절차적 언어에서 쉽게 사용할 수 있고 SQL을 사용하기 쉽기 때문에, 데이터베이스 서버 내부에서의 쿼리와 연산을 그룹화 할 수 있으며 클라이언트/서버 통신 부하를 절약할 수 있다.

- 클라이언트와 서버간의 불필요한 통신을 제거한다.
- 클라이언트가 불필요한 중간 결과물들을 가지고 있거나 서버/클라이언트간에 전송하지 않아도 된다.
- 반복되는 쿼리 파싱 처리를 하지 않아도 된다.

이런 요소들 때문에 저장 함수들을 이용하지 않는 애플리케이션보다 눈에 띄는 성능 향상을 기대할 수 있다. 또한, PL/pgSQL은 SQL의 모든 데이터 자료형, 연산, 함수를 이용할 수 있다.

지원되는 인수 및 결과데이터 형식

PL/pgSQL로 작성된 함수는 서버가 지원하는 스칼라 또는 배열 데이터 유형을 인수로 사용할 수 있으며 결과로 반환할 수 있다. 또한 이름으로 지정된 복합 유형 (row 유형) 을 사용하거나 반환할 수 있다. PL/pgSQL 함수는 레코드를 반환 할 수도 있다.

PL/pgSQL 함수는 VARIADIC 마커를 사용하여 수가 변하는 인수들을 허용하도록 선언할 수 있다. 이것은 SQL 함수와 똑같은 방식으로 작동한다.

PL/pgSQL 함수는 `anyelement`, `anyarray`, `anynonarray`, `anyenum` 및 `anyrange` 등과 같이 다양한 유형을 받아들이고 반환하도록 선언할 수도 있다. 다형 (polymorphic) 함수에 의해 처리되는 실제 데이터 유형은 호출마다 다를 수 있다.

7.2.2 PL/pgSQL의 구조

PL/pgSQL로 작성된 함수는 CREATE FUNCTION 명령을 실행하여 서버에 정의된다. 그런 명령은 일반적으로 아래와 같다.

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

함수 본문은 단순히 CREATE FUNCTION에 관련된 글자 그대로의 문자열이다. 일반적인 작은 따옴표 구문보다는 함수 본문을 쓰는데 달러 ($) 인용 부호를 사용하는 것이 도움이 된다. 달러 인용 부호가 없으면 함수 본문의 작은 따옴표 나 백슬래시를 두 배로 늘려 이스케이프 해야 한다. 이 장의 거의 모든 예제는 함수 본문에 달러 인용 리터럴을 사용한다.

PL/pgSQL은 블록구조의 언어다. 함수 정의의 전체 텍스트는 블록이어야 한다. 블록은 아래처럼 정의된다.

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

블록 내의 각 선언 및 각 명령문은 세미콜론으로 끝난다. 위에 표시된 것처럼 다른 블록 내에 나타나는 블록에는 END 다음에 세미콜론이 있어야 한다. 그러나 함수 본문을 끝내는 마지막 END에는 세미콜론이 필요하지 않다.

> Tip : 일반적으로 BEGIN 직후에 세미콜론을 쓰는 실수가 많다. 이것은 올바르지 않으며 구문 오류가 발생한다.

*Label*은 EXIT 문에서 사용할 블록을 식별하거나 블록에서 선언된 변수의 이름을 한정하려는 경우에만 필요하다. Label이 END 다음에 위치한다면, 블록 시작 부분의 label과 일치해야 한다. 식별자는 일반 SQL명령과 마찬가지로 큰 따옴표가 없으면 기본적으로 소문자로 변환된다.

주석은 일반 SQL에서와 마찬가지로 PL/pgSQL 코드에서도 동일한 방식으로 작동한다. 이중 대시(--)는 시작 부분부터 그 행의 끝까지 주석으로 인식한다. 블럭으로 주석을 처리하기 위해서는 /*와 */ 사이에 주석을 기술한다.

블록의 명령문 섹션 내에 있는 모든 명령문은 서브 블록이 될 수 있다. 서브 블록은 논리적 그룹화 또는 변수의 작은 그룹에 대한 localization을 위해 사용될 수 있다. 서브 블록에서 선언된 변수는 서브 블록의 지속 시간동안 외부 블록의 유사한 이름의 변수를 가린다. 블록 이름을 지정하면 외부 변수에 액세스 할 수 있다.

```
CREATE FUNCTION somefunc() RETURNS integer AS $$

DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

> 참고 : 실제로는 PL/pgSQL 함수의 본문을 둘러싼 숨겨진 ``외부 블록''이 있다. 이 블록은 함수의 매개 변수 선언(있는 경우)과 FOUND와 같은 특수 변수를 제공한다. 외부 블록은 함수의 이름으로 레이블이 지정된다. 즉, 매개 변수와 특수 변수를 함수의 이름으로 한정할 수 있다.

PL/pgSQL에서 명령문을 그룹화하기 위해 BEGIN/END를 사용하는 것을 트랜잭션 제어를 위해 유사하게 명명된

SQL명령과 혼동하지 않는 것이 중요하다. PL/pgSQL의 BEGIN/END는 그룹화에만 사용된다. 트랜잭션이 시작되거나 종료되지 않는다. 함수와 트리거 프로시저는 외부 쿼리에 의해 설정된 트랜잭션 내에서 항상 실행된다. 트랜잭션을 시작할 컨텍스트가 없으므로 트랜잭션을 시작하거나 commit할 수 없다. 그러나 블록은 EXCEPTION 절을 포함하고 있어서 외부 트랜잭션에 영향을 미치지 않고 롤백될 수 있는 하위 트랜잭션을 구성할 수 있다.

7.2.3 Declarations

블록에서 사용되는 모든 변수는 블록의 선언 섹션에서 선언되어야 한다. (유일한 예외는 정수 값의 범위를 반복하는 FOR 루프의 루프 변수는 자동으로 정수 변수로 선언되고, cursor의 결과를 조회하는 FOR 루프의 루프 변수는 자동으로 레코드 변수로 선언된다는 것이다.) PL/pgSQL 변수는 `integer`, `varchar` 및 `char`과 같은 모든 SQL데이터 유형을 가질 수 있다.
아래는 변수 선언의 몇가지 예제이다.

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

변수 선언의 일반 구문은 다음과 같다.

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | := | = } expression ];
```

`DEFAULT`절이 주어지면 블록 입력시 변수에 할당된 초기 값을 지정한다. `DEFAULT`절이 제공되지 않으면 변수는 SQL null 값으로 초기화된다. `CONSTANT` 옵션은 초기화 후 변수가 할당되지 않도록 하여 해당 값이 블록 지속기간동안 일정하게 유지되도록 한다. `COLLATE` 옵션은 변수에 사용할 데이터 정렬을 지정한다. `NOT NULL`이 지정되면, 실행시 null 값이 할당되면 런타임 오류가 발생한다. `NOT NULL`로 선언된 모든 변수는 null이 아닌 기본값을 지정해야 한다. 등호(=)는 PL/SQL과 호환되는 ``:='' 대신에 사용할 수 있다.
변수의 기본값은 블록이 입력될 때마다 (함수 호출당 한 번이 아니라) 평가되고 변수에 지정된다. 예를 들어, `now()`를 `timestamp` 유형의 변수에 지정하면 함수는 사전 컴파일된 시간이 아닌 현재 함수 호출 시간을 갖게 된다.

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

함수 매개변수 선언

함수에 전달된 매개 변수는 식별자 $1, $2 등으로 이름이 지정된다. 선택적으로 $$n$ 매개변수 이름에 대해 별칭을 선언하여 가독성을 높일 수 있다. 그런 다음 별칭 또는 숫자 식별자를 사용하여 매개변수 값을 참조할 수 있다. 별칭을 만드는 방법에는 두 가지가 있다. 선호되는 방법은 CREATE FUNCTION 명령에서 매개변수에 이름을 지정하는 것이다.

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

다른 방법은 선언 구문을 사용하여 별칭을 명시적으로 선언하는 것이다.

```
name ALIAS FOR $n;
```

아래는 이 스타일과 동일한 예제이다.

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> 참고 : 이 두 예제는 완전히 동일하지 않다. 첫 번째 경우 subtotal은 sales_tax.subtotal로 참조될 수 있지만, 두 번째 경우에는 subtotal으로 표시될 수 없다. (내부 블록에 레이블을 첨부했다면 대신 부분 레이블이 해당 레이블로 한정될 수 있다.)

PL/pgSQL 함수가 출력 매개변수로 선언될 때, 출력 매개변수는 일반적인 입력 매개변수와 같은 방식으로 $$n$ 이름과 선택적 별칭을 갖는다. 출력 매개변수는 NULL로 시작하는 변수를 효과적으로 나타낸다. 이것은 함수 실행 중에 할당되어야 한다. 매개변수의 최종 값은 반환되는 값이다. 예를 들어, sales-tax의 예도 다음과 같이 할 수 있다.

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

출력 매개변수는 여러 값을 반환할 때 가장 유용하다. 간단한 예제는 다음과 같다.

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

이것은 함수의 결과에 대한 익명 record 유형을 효과적으로 생성한다. RETURNS절이 제공되면 RETURNS record를 반드시 기재해야 한다.

PL/pgSQL 함수를 선언하는 또 다른 방법은 RETURNS TABLE을 사용하는 것이다. 예를 들면 다음과 같다.

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

이는 하나 이상의 OUT 매개변수를 선언하고 RETURNS SETOF *sometype*을 지정하는 것과 정확히 같다.

Alias

함수 매개변수뿐만 아니라 모든 변수에 대한 alias를 선언할 수 있다. 이에 대한 실제적인 사용은 트리거 절차 내에서 NEW 또는 OLD와 같이 미리 정해진 이름을 가진 변수에 다른 이름을 할당하는 것이다. 예를 들면 다음과 같다.

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

ALIAS는 동일한 객체의 이름을 지정하는 두 가지 방법을 생성하기 때문에 제약없는 사용은 혼란스러울 수 있다. 미리 정해진 이름을 무시할 목적으로만 사용하는 것이 가장 좋다.

7.2.4 Expressions

PL/pgSQL문에서 사용되는 모든 표현식은 서버의 주요 SQL실행기를 사용하여 처리된다. PL/pgSQL문은 다음과 같이 기술할 수 있다.

```
IF expression THEN ...
```

PL/pgSQL은 주요 SQL엔진과 같은 쿼리를 제공하여 표현식을 평가한다. SELECT 명령을 구성하는 동안 PL/pgSQL 변수 이름은 매개변수로 바뀐다. 이렇게하면 SELECT에 대한 쿼리 계획을 한번만 준비한 다음 변수의 다른 값을 사용하여 후속 평가에 재사용할 수 있다. 예를 들어, 두 개의 정수 변수 x와 y를 다음과 같이 기술하면

```
IF x < y THEN ...
```

아래와 같이 사용된다.

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

이 준비된 명령문은 IF 문이 매번 실행될때 제공된 PL/pgSQL 변수의 값과 함께 수행된다. 보통 이러한 세부사항은 PL/pgSQL 사용자에게는 유용하지 않지만 문제 진단시 알아두면 유용하다.

7.2.5 Basic Statements

이 섹션과 다음 섹션에서는 PL/pgSQL이 명시적으로 이해하는 모든 명령문 유형을 설명한다. 이러한 명령문 유형 중 하나로 인식되지 않는 것은 SQL명령으로 간주되며 실행을 위해 기본 데이터베이스 엔진으로 전송된다.

Assignment

PL/pgSQL 변수에 값을 할당하는 방법은 다음과 같다.

```
variable { := | = } expression;
```

앞서 설명했듯이 이러한 명령문의 표현식은 기본 데이터베이스 엔진에 전송된 SQL SELECT 명령을 사용하여 구한다. 표현식은 단일 값을 가져야 한다(변수가 행 또는 레코드 변수인 경우 행 값일 수 있음). 대상 변수는 단순 변수(선택적으로 블록 이름으로 규정됨), 행 또는 레코드 변수 필드 또는 단순 변수 또는 필드인 배열 요소일 수 있다. 등호(=)는 PL/SQL과 호환되는 ``:='' 대신에 사용할 수 있다.

표현식의 결과 데이터 유형이 변수의 데이터 유형과 일치하지 않거나 특정한 길이나 정밀도에 일치하지 않으면, PL/pgSQL 인터프리터는 출력 함수의 결과 유형과 입력 함수의 변수 유형을 사용하여 결과 값을 묵시적으로 변환하려고 시도한다. 결과값의 문자열 형식이 입력 함수에 허용되지 않는 형식이라면 입력 함수에서 런타임 오류가 발생할 수 있다.

예를 들면 다음과 같다.

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

결과가 없는 명령 실행

행을 반환하지 않는 SQL명령(예 : RETURNING 절이 없는 INSERT)의 경우 명령을 작성하기만 하면 PL/pgSQL 함수 내에서 명령을 실행할 수 있다.

명령 텍스트에 나타나는 PL/pgSQL 변수 이름은 매개변수로 취급되며, 런타임에 변수의 현재 값이 매개변수 값으로 제공된다. 이는 표현식에 대해 앞서 설명한 처리와 정확히 같다.

이 방법으로 SQL명령을 실행할 때 PL/pgSQL은 명령 실행 계획을 캐싱(caching)하고 재사용 할 수 있다.

때로는 유용한 결과값이 없는 함수를 호출하는 것은 표현식이나 SELECT 쿼리를 평가하는데 유용하며 결과는 무시할 수 있다. PL/pgSQL에서 이렇게 하려면 PERFORM문을 사용하기를 권장한다.

```
PERFORM query;
```

그러면 쿼리가 실행되고 결과는 버려진다. SQL SELECT 명령을 쓰는 것과 같은 방법으로 쿼리를 작성하고, 초기 키워드 SELECT를 PERFORM으로 대체해야 한다. WITH 쿼리의 경우 PERFORM을 사용하고 쿼리를 괄호 안에 넣는다.(이 경우 쿼리는 한 행만 반환할 수 있다.) 결과를 반환하지 않는 명령과 마찬가지로 PL/pgSQL 변수가 쿼리로 대체되며, 계획은 같은 방식으로 캐시된다. 또한 특수 변수 FOUND는 쿼리가 적어도 하나의 행을 생성한 경우 true로 설정되고 행이 생성되지 않은 경우 false로 설정된다.

> 참고 : SELECT를 직접 작성하면 이 결과를 얻을 것이라고 기대할 수 있지만, 현재로서는, 그것을 행하는 유일한 방법은 PERFORM뿐이다. SELECT와 같은 행을 반환할 수있는 SQL명령은 다음 절에서 설명하는 INTO 절이 없으면 오류로 취급되어 거부된다.

예를 들면 다음과 같다.

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

단일 행 결과로 쿼리 실행

SQL명령의 결과로 발생하는 단일 행(여러 열일 수도 있음)은 생성하는 레코드 변수, 행 유형 변수 또는 스칼라 변수 목록에 대입될 수 있다. 이것은 기본 SQL명령을 작성하고 INTO절을 추가하여 수행된다. 예를 들어, target은 레코드 변수, 행 변수 또는 쉼표로 구분된 단순 변수 및 레코드/행 필드 목록일 수 있다.

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

행을 반환하지 않는 명령에 대해 위에서 설명한대로 PL/pgSQL 변수는 나머지 쿼리로 대체되고 계획이 캐싱된다. 이는 RETURNING을 사용하는 SELECT, INSERT/UPDATE/DELETE에서 동작하고, 행 집합 결과(예 : EXPLAIN)를 반환하는 유틸리티 명령에 대해 작동한다. INTO절을 제외하면 SQL명령은 PL/pgSQL 외부에서 작성된 것과 동일하다.

> Tip : PL/pgSQL 함수 내에서 SELECT 결과로 테이블을 만들려면 CREATE TABLE ... AS SELECT 구문을 사용해야 한다.

행 또는 변수 목록을 대상으로 사용하는 경우 쿼리의 결과 열은 숫자 및 데이터 형식에 대한 대상 구조와 정확히 일치해야 한다. 그렇지 않으면 런타임 오류가 발생한다. 레코드 변수가 타겟일 때, 자동으로 쿼리 결과 열들의 행 유형을 구성한다.

INTO절은 SQL명령의 거의 모든 위치에 나타날 수 있다. 일반적으로 SELECT 명령의 *select_expressions* 목록 직전 또는 직후에 작성되거나 다른 명령 유형에 대한 명령의 끝 부분에 작성된다.

STRICT가 INTO절에 지정되지 않은 경우, 타겟은 쿼리에 의해 리턴된 첫 번째 행으로 설정되거나, 쿼리가 행을 리턴하지 않은 경우 null로 설정된다. (``첫 번째 행''은 ORDER BY를 사용하지 않는한 잘 정의되어 있지 않다.) 첫 번째 행 뒤의 결과 행은 버려진다. 특수 FOUND 변수를 검사하여 행이 반환되었는지 여부를 확인할 수 있다.

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

STRICT 옵션이 지정되면 쿼리는 정확히 하나의 행을 리턴해야 한다. 그렇지 않으면 NO_DATA_FOUND(행 없음) 또는 TOO_MANY_ROWS(둘 이상의 행) 중 하나가 런타임 오류로 보고된다. 다음과 같이 오류를 찾고자 한다면, 예외 블록을 사용할 수 있다.

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

STRICT 명령을 성공적으로 실행하면 항상 FOUND가 true로 설정된다.

RETURNING을 사용하는 INSERT/UPDATE/DELETE의 경우, STRICT가 지정되지 않은 경우에도 PL/pgSQL은 둘 이상의 반환된 행에 대해 오류를 보고한다. 이것은 영향을 받는 행을 리턴해야 하는지 판별하는 ORDER BY와 같은 옵션이 없기 때문이다.

함수에 대해 print_strict_params가 사용 가능한 경우, STRICT의 요구 사항이 충족되지 않아 오류가 발생하면 오류 메시지의 DETAIL 부분에 쿼리에 전달된 매개변수에 대한 정보가 포함된다. plpgsql.print_strict_params 를 설정하여 모든 함수에 대한 print_strict_params 설정을 변경할 수 있지만 후속(subsequence) 함수 편집에만 영향을 받는다. 컴파일러 옵션을 사용하여 함수 단위로 활성화할 수도 있다. 예를 들면 다음과 같다.

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END
$$ LANGUAGE plpgsql;
```

실패하면 이 함수는 다음과 같은 오류 메시지를 생성할 수 있다.

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL statement
```

> 참고 : STRICT 옵션은 Oracle PL/SQL의 SELECT INTO 및 관련 명령문의 동작과 일치한다.

동적 명령 실행

흔히 PL/pgSQL 함수 안에서 동적인 명령, 즉 실행될 때마다 다른 테이블이나 다른 데이터 타입을 포함하는 명령을 생성하기를 원할 것이다. 명령에 대한 계획을 캐시하는 PL/pgSQL의 일반적인 시도는 이 시나리오에서는 작동하지 않는다. 이런 종류의 문제를 처리하기 위해 EXECUTE문이 제공된다.

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

여기서 *command-string*은 실행할 명령을 포함하는 문자열(유형 텍스트)을 생성하는 표현식이다. 선택적 타겟은 명령의 결과가 저장될 레코드 변수, 행 변수 또는 쉼표로 구분된 단순 변수 및 레코드/행 필드 목록이다. 선택적 USING 표현식은 명령에 삽입할 값을 제공한다.

계산된 명령 문자열에서는 PL/pgSQL 변수를 대체할 수 없다. 필요한 모든 변수 값은 구성된대로 명령 문자열에 삽입해야 한다. 또는 아래의 설명대로 매개변수를 사용할 수 있다.

또한 EXECUTE를 통해 실행되는 명령에 대한 캐싱 계획도 없다. 대신 명령문이 실행될 때마다 항상 명령이 계획된다. 따라서 명령 문자열은 다른 테이블과 컬럼에서 action을 수행하기 위해 함수 내에서 동적으로 생성될 수 있다.

INTO절은 반환되는 행의 결과를 할당하는 SQL명령의 결과를 명시한다. 행 또는 변수 목록이 제공되면 쿼리 결과의 구조와 정확히 일치해야 한다.(레코드 변수가 사용되면 자동으로 결과 구조와 일치하도록 구성된다.) 여러 행이 반환되면 첫 번째 행만 INTO 변수에 할당된다. 행이 반환되지 않으면 NULL이 INTO 변수에 할당된다. INTO 절을 지정하지 않으면, 쿼리결과는 버려진다.

STRICT 옵션이 주어지면, 쿼리가 정확히 하나의 행을 생성하지 않으면 오류가 보고된다.

명령 문자열은 명령에서 $1, $2 등으로 참조되는 매개변수 값을 사용할 수 있다. 이 기호는 USING절에 제공된 값을 참조한다. 이 방법은 데이터 값을 텍스트로 명령 문자열에 삽입하는 것보다 선호되는 경우가 많다. 즉, 값을 텍스트 및 백 (back) 으로 변환하는 런타임 오버 헤드를 피할 수 있으며, 따옴표나 이스케이프가 필요 없으므로 SQL-injection 공격이 덜 발생한다. 예를 들면 다음과 같다.

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

매개변수 기호는 데이터 값에 대해서만 사용할 수 있다. 동적으로 결정된 테이블 또는 열 이름을 사용하려면 문자를 명령 문자열에 삽입해야 한다. 예를 들어 앞의 쿼리를 동적으로 선택한 테이블에 대해 수행해야 하는 경우 다음과 같이 할 수 있다.

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tabname)
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

더 깔끔한 접근법은 테이블이나 컬럼 이름 (개행으로 구분된 문자열이 연결됨) 에 대해 format() 의 %I 사양을 사용하는 것이다.

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE inserted_by = $1 AND inserted <= $2', tabname)
   INTO c
   USING checked_user, checked_date;
```

매개변수 기호에 대한 또다른 제한 사항은 SELECT, INSERT, UPDATE 및 DELETE 명령에서만 작동한다는 것이다. 다른 구문 유형 (일반적으로 유틸리티 구문이라고 함) 에서는 데이터 값일지라도 텍스트로 값을 삽입해야 한다.

위의 첫 번째 예제에서와 같이 단순한 상수 명령 문자열과 일부 USING 매개변수가 있는 EXECUTE는 PL/pgSQL에서 직접 명령을 작성하고 PL/pgSQL 변수가 자동으로 대체되도록 허용하는 것과 기능상 동일하다. 중요한 차이점은 EXECUTE가 각 실행시 명령을 다시 계획하여 현재 매개변수 값과 관련된 계획을 생성한다는 것이다. 반면에 PL/pgSQL은 일반 계획을 만들어 재사용을 위해 캐싱 할 수 있다. 최상의 계획이 매개변수 값에 크게 의존하는 상황에서는 EXECUTE를 사용하여 일반 계획이 선택되지 않았는지 확실하게 확인하는 것이 좋다.

SELECT INTO는 현재 EXECUTE내에서 지원되지 않는다. 대신에 일반 SELECT 명령을 실행하고 EXECUTE 자체의 일부로 INTO를 지정해야 한다.

7.2.6 Control Structures

제어 구조는 아마도 PL/pgSQL에서 가장 유용하고 중요한 부분일 것이다. PL/pgSQL의 제어 구조를 사용하면 매우 유연하고 강력한 방법으로 AgensGraph 데이터를 조작할 수 있다.

함수에서 반환하기

함수에서 데이터를 반환할 수 있는 명령으로는 `RETURN` 및 `RETURN NEXT`, 두가지가 있다.

1. RETURN

```
RETURN expression;
```

표현식이 있는 `RETURN`은 함수를 종료하고 표현식의 값을 호출자에게 리턴한다. 이 형식은 집합을 반환하지 않는 PL/pgSQL 함수에 사용된다.

스칼라 형식을 반환하는 함수에서 식에 대한 결과는 할당에 대해 설명한대로 함수의 반환 형식으로 자동 변환된다. 그러나 복합(행) 값을 반환하려면 요청된 열 집합을 정확하게 전달하는 표현식을 작성해야 한다. 이를 위해서는 명시적 형변환을 사용해야 할 수도 있다.

출력 매개변수를 사용하여 함수를 선언한 경우 식이 없는 `RETURN`만 작성해야 한다. 출력 매개변수 변수의 현재 값이 리턴된다.

`void`를 반환하는 함수를 선언한 경우 `RETURN`문을 사용하여 함수를 일찍 종료할 수 있다. `RETURN` 다음에는 표현식을 쓰지 말아야 한다.

함수의 반환 값은 정의되지 않은 채로 남아있을 수 없다. 컨트롤이 `RETURN`문을 사용하지 않고 함수의 최상위 블록 끝에 도달하면 런타임 오류가 발생한다. 그러나 이 제한은 출력 매개변수가 있는 함수와 `void`를 반환하는 함수에는 적용되지 않는다. 이 경우 최상위 블록이 완료되면 `RETURN`문이 자동으로 실행된다.

예를 들면 다음과 같다.

```
-- functions returning a scalar type
RETURN 1 + 2;
RETURN scalar_var;


-- functions returning a composite type
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- must cast columns to correct types
```

2. RETURN NEXT and RETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

PL/pgSQL 함수가 SETOF *sometype*을 반환하도록 선언되면, 따라야 할 프로시저가 약간 다르다. 이 경우 리턴할 개별 항목은 RETURN NEXT 또는 RETURN QUERY 명령의 순서로 지정되며, 인수가 없는 최종 RETURN 명령은 함수의 실행이 완료 되었음을 나타내기 위해 사용된다. RETURN NEXT는 스칼라 및 복합 데이터 유형 모두와 함께 사용할 수 있다. 복합 결과 유형으로 결과의 전체 ``테이블''이 리턴된다. RETURN QUERY는 조회를 실행한 결과를 함수의 결과 세트에 추가한다. RETURN NEXT와 RETURN QUERY는 단일 set-returning 함수에서 자유롭게 섞일 수 있으며, 이 경우 결과는 연결된다.

RETURN NEXT 및 RETURN QUERY는 실제로 함수에서 반환되지 않는다. 함수의 결과 집합에 0개 이상의 행을 추가하기만 하면 된다. 실행은 PL/pgSQL 함수의 다음 명령문으로 계속된다. 연속 RETURN NEXT 또는 RETURN QUERY 명령이 실행될 때 결과 세트가 빌드된다. 인수가 없어야 하는 마지막 리턴은 제어 기능이 함수를 종료하게 한다.(또는 제어 기능이 함수의 끝에 도달하게 할 수 있다).

RETURN QUERY에는 동적으로 실행될 조회를 지정하는 변형 RETURN QUERY EXECUTE가 있다. 매개변수 식은 EXECUTE 명령과 마찬가지로 USING을 통해 계산된 쿼리 문자열에 삽입할 수 있다.

출력 매개변수를 사용하여 함수를 선언한 경우 표현식 없이 RETURN NEXT를 작성해야 한다. 각 실행시 출력 매개변수 변수의 현재 값이 결과 행에 대한 최종 반환을 위해 저장된다. 여러 개의 출력 매개변수가 있는 경우 SETOF 레코드를 반환하는 함수를 선언하거나, *sometype* 유형의 출력 매개변수가 하나만 있는 경우에는 출력 매개변수를 사용하여 set-returning 함수를 만들기 위해 SETOF *sometype*을 선언해야 한다.

다음은 RETURN NEXT를 사용하는 함수의 예제이다.

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END
```

```
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

다음은 RETURN QUERY를 사용하는 함수의 예제이다.

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);

    -- Since execution is not finished, we can check whether rows were returned
    -- and raise exception if not.
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No flight at %.', $1;
    END IF;

    RETURN;
 END
$BODY$
LANGUAGE plpgsql;

-- Returns available flights or raises exception if there are no
-- available flights.
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

> 참고 : RETURN NEXT와 RETURN QUERY의 현재 구현은 위에서 설명한대로 함수에서 리턴하기 전에 전체 결과 세트를 저장한다. 즉, PL/pgSQL 함수가 매우 큰 결과 집합을 생성하는 경우 성능이 떨어질 수 있다. 메모리 고갈을 피하기 위해 데이터가 디스크에 기록되지만 함수 자체는 전체 결과 집합이 생성될 때까지 반환되지 않는다. 현재 데이터가 디스크에 기록되기 시작한 시점은 work_mem 구성 변수에 의해 제어된다. 더 큰 결과 세트를 메모리에 저장할 수 있는 충분한 메모리를 가진 관리자는 이 매개변수를 늘려야 한다.

단순 루프

LOOP, EXIT, CONTINUE, WHILE, FOR 및 FOREACH문을 사용하면 PL/pgSQL 함수가 일련의 명령을 반복하도록 정렬할 수 있다.

1. LOOP

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

LOOP는 EXIT 또는 RETURN문에 의해 종료될 때까지 무기한 반복되는 무조건 (unconditional) 루프를 정의한다. 선택적 *label*은 중첩 루프 내에서 EXIT 및 CONTINUE문에 의해 사용되어 해당 문이 참조하는 루프를 지정한다.

2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

*label*이 지정되지 않으면 가장 안쪽의 루프가 종료되고 END LOOP 다음의 명령문이 다음에 실행된다. *label*이 주어지면 중첩 루프 또는 블록의 현재 또는 일부 외부 레벨의 label이어야 한다. 그런 다음 명명 된 루프 또는 블록이 종료되고 루프/블록의 해당 END 다음에 명령문이 계속된다.
WHEN이 지정되면, *boolean-expression*이 true인 경우에만 루프 종료가 발생한다. 그렇지 않으면 제어 기능은 EXIT 후에 명령문으로 넘어간다.
EXIT는 모든 유형의 루프에서 사용할 수 있다. 그것은 무조건 루프와 함께 사용하는 것에 국한되지 않는다.
BEGIN 블록과 함께 사용될 때, EXIT는 블록 종료 후에 제어를 다음 명령문으로 전달한다. Label은 이 용도로 사용해야 한다. Label이 없는 EXIT는 BEGIN 블록과 일치하는 것으로 간주되지 않는다.
예제는 다음과 같다.

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;

LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
```

```
END LOOP;



BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT ablock;  -- causes exit from the BEGIN block
    END IF;
    -- computations here will be skipped when stocks > 100000
END;
```

3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

*label*이 주어지지 않으면 가장 안쪽 루프의 다음 반복이 시작된다. 즉, 루프 본문에 남아있는 모든 문을 건너 뛰고 다른 루프 반복이 필요한지 여부를 결정하기 위해 루프 제어식이 반환된다(있는 경우). *label*이 있으면 실행을 계속할 루프의 label을 지정한다.

`WHEN`을 지정하면, *boolean-expression*이 true인 경우에만 루프의 다음 반복이 시작된다. 그렇지 않으면 `CONTINUE` 다음의 명령문으로 제어가 넘어간다.

`CONTINUE`는 모든 유형의 루프에서 사용할 수 있다. 그것은 무조건 루프와 함께 사용하는 것에 국한되지 않는다. 예제는 다음과 같다.

```
LOOP
    -- some computations
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- some computations for count IN [50 .. 100]
END LOOP;
```

4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

`WHILE`문은 *boolean-expression*이 true로 평가되는한 일련의 명령문을 반복한다. 표현식은 루프 본문에 대한 각 항목 바로 전에 검사된다.

예제는 다음과 같다.

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
END LOOP;

WHILE NOT done LOOP
    -- some computations here
END LOOP;
```

5. FOR(Integer Variant)

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

FOR의 이 형식은 정수 값 범위를 반복하는 루프를 만든다. 변수 이름은 자동으로 정수 유형으로 정의되며 루프 내에만 존재한다(변수 이름의 기존 정의는 루프 내에서 무시된다). 범위의 상한과 하한을 나타내는 두 표현식은 루프에 들어갈 때 한 번 평가된다. BY절이 지정되지 않은 경우 반복 단계는 1이고, 그렇지 않으면 BY절에 지정된 값이며 루프 항목에서 한번 평가된다. REVERSE를 지정하면 각 반복 후에 단계 값이 추가되지 않고 감산된다. integer FOR 루프의 몇 가지 예는 다음과 같다.

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

하한이 상한보다 큰 경우(또는 REVERSE의 경우보다 작으면) 루프 본문은 전혀 실행되지 않는다. 오류는 발생하지 않는다.

*label*이 FOR 루프에 연결되면 integer 루프 변수는 해당 *label*을 사용하여 규정된 이름으로 참조 될 수 있다.

쿼리 결과를 통한 루프

다른 유형의 FOR 루프를 사용하면 쿼리 결과를 반복하고 그에 따라 데이터를 조작할 수 있다. 구문은 다음과 같다.

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

*target*은 레코드 변수, 행 변수 또는 쉼표로 구분된 스칼라 변수 목록이다. *target*은 쿼리의 결과를 각 행에 연속적으로 할당되고, 루프 본문은 각 행에 대해 실행된다. 다음은 그 예이다.

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing materialized views...';

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- Now "mviews" has one record from cs_materialized_views

        RAISE NOTICE 'Refreshing materialized view %s ...', quote_ident(mviews.mv_name);
        EXECUTE format('TRUNCATE TABLE %I', mviews.mv_name);
        EXECUTE format('INSERT INTO %I %s', mviews.mv_name, mviews.mv_query);
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

루프가 EXIT문에 의해 종료되면, 루프 다음에 마지막으로 할당된 행 값에 계속 액세스할 수 있다.

FOR문의 유형에서 사용되는 쿼리는 호출자에게 행을 리턴하는 모든 SQL명령이 될 수 있다. SELECT가 가장 일반적인 경우이지만 RETURNING과 함께 INSERT, UPDATE 또는 DELETE를 사용할 수도 있다. EXPLAIN과 같은 일부 유틸리티 명령도 작동한다.

PL/pgSQL 변수가 쿼리 텍스트로 대체되고 가능한 재사용을 위해 쿼리 계획이 캐싱된다.

FOR-IN-EXECUTE문은 행을 반복하는 또 다른 방법이다.

```
[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    statements
END LOOP [ label ];
```

이는 소스 쿼리가 FOR 루프의 각 항목에서 평가되고 다시 검색되는 문자열 표현으로 지정된다는 점을 제외하면 이전 양식과 같다. 이를 통해 프로그래머는 일반 EXECUTE문과 마찬가지로 미리 계획된 쿼리의 속도 또는 동적 쿼리의 유연성을 선택할 수 있다. EXECUTE와 마찬가지로 매개변수 값을 USING을 통해 동적 명령에 삽입할 수 있다.

결과를 반복해야하는 쿼리를 지정하는 또 다른 방법은 이를 커서로 선언하는 것이다.

배열을 통한 루프

FOREACH 루프는 FOR 루프와 매우 유사하지만 SQL쿼리에서 반환 된 행을 반복하는 대신 배열 값의 요소를 반복한다. 일반적으로 FOREACH는 복합값 표현식의 구성 요소를 반복 할 때 사용된다. 배열 이외의 복합체를 반복하는 변형은 나중에 추가 될 수 있다. 배열을 반복하는 FOREACH문은 다음과 같습니다.

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

SLICE가 없거나 SLICE0이 지정된 경우 루프는 *expression*을 평가하여 생성된 배열의 개별 요소를 반복한다. *target* 변수에는 각 요소 값이 순서대로 지정되고 루프 본문이 각 요소에 대해 실행된다. 다음은 정수 배열 요소를 반복하는 예제이다.

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

요소들은 배열의 차원 수와는 관계없이 스토리지 순서로 저장된다. *target*은 일반적으로 단일 변수이지만 복합 값 배열(records)을 반복할 때 변수들의 목록이 될 수 있다. 이 경우 각 배열 요소에 대해 변수들은 복합 값의 연속열에서 할당된다.

SLICE 값이 양수이면, FOREACH는 단일 요소가 아닌 배열의 슬라이스를 반복한다. SLICE 값은 배열의 차원 수보다 크지 않은 정수 상수여야 한다. *target* 변수는 배열이어야 하며 배열 값의 연속된 슬라이스를 받는다. 각 슬라이스는 SLICE에 의해 지정된 차원 수이다. 다음은 1차원 슬라이스를 반복하는 예제이다.

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

7.2.7 Cursors

한 번에 전체 쿼리를 실행하는 대신 쿼리를 캡슐화하는 커서를 설정한 다음 쿼리 결과를 한 번에 몇 줄씩 읽을 수 있다. 이렇게 하는 이유 중 하나는 결과에 많은 수의 행이 포함되어 있을 때 메모리 오버런(overrun)을 피하기 위해서이다.(그러나 FOR 루프는 메모리 문제를 피하기 위해 내부적으로 커서를 사용하기 때문에 PL/pgSQL 사용자는 일반적으로 걱정할 필요가 없다.) 보다 흥미로운 사용법은 함수가 생성된 커서에 대한 참조를 반환하여 호출자가 행을 읽을 수 있도록 하는 것이다. 이렇게 하면 함수에서 큰 행 집합을 반환하는 효율적인 방법을 제공한다.

커서 변수 선언

PL/pgSQL의 커서에 대한 모든 액세스는 특수 데이터 유형 refcursor인 커서 변수를 통해 항상 수행된다. 커서 변수를 만드는 한 가지 방법은 refcursor 유형의 변수로 선언하는 것이다. 또 다른 방법은 일반적으로 다음과 같은 커서 선언 구문을 사용하는 것이다.

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

SCROLL이 지정되면 커서는 뒤로 스크롤 할 수 있다. NO SCROLL이 지정되면 역방향 페치(fetch)가 거부된다. 옵션이 지정되지 않으면 역방향 페치가 허용되는지 여부에 관계없이 쿼리에 따라 다르다. 인수가 지정된 경우 쉼표로 구분된 이름 데이터 유형 쌍의 목록이며 지정된 쿼리에서 매개변수 값으로 대체될 이름을 정의한다. 이 이름을 대체할 실제 값은 커서가 열릴 때 나중에 지정된다.

예제는 다음과 같다.

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

이 세 변수는 모두 refcursor 데이터 유형을 갖지만, 첫 번째 쿼리는 모든 쿼리와 함께 사용할 수 있으며, 두 번째 쿼리는 이미 완전히 지정된 쿼리를 *bound*하고, 마지막 쿼리에는 매개변수가 있는 쿼리가 바인딩 되어 있다. (key는 커서가 열릴 때 정수 매개변수 값으로 대체된다.) 변수 curs1은 특정 쿼리에 바인딩되지 않았기 때문에 *unbound*라고 한다.

커서 열기

행을 검색하는데 커서를 사용하려면 먼저 커서는 열려있어야 한다. PL/pgSQL은 3가지 형태의 OPEN문을 지원하는데, 그중 두 문항은 unbound 커서 변수를 사용하고 세 번째는 바운드 커서 변수를 사용한다.

> 참고 : 바운드 커서 변수는 명시적으로 커서를 열지 않고도 사용할 수 있다.

1. OPEN FOR *query*

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

커서 변수가 열리고 지정된 쿼리가 실행된다. 커서는 더이상 열 수 없으며, unbound 커서 변수(즉, 간단한 refcursor 변수)로 선언되어 있어야 한다. 쿼리는 SELECT이거나 행을 반환하는 다른 것(EXPLAIN과 같은)이어야 한다. 쿼리는 PL/pgSQL의 다른 SQL명령과 같은 방식으로 처리된다. PL/pgSQL 변수 이름이 대체되고, 쿼리 계획은 있을 수 있는 재사용을 위해 캐싱된다. PL/pgSQL 변수가 커서 쿼리로 대체될 때, 대체되는 값은 OPEN 시점의 값이다. 변수에 대한 후속 변경 사항은 커서의 동작에 영향을 미치지 않는다. SCROLL 및 NO SCROLL 옵션은 바운드 커서에서와 동일한 의미를 갖는다.

예제는 다음과 같다.

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
                                       [ USING expression [, ... ] ];
```

커서 변수가 열리고 지정된 쿼리가 실행된다. 커서는 더이상 열 수 없으며, unbound 커서 변수(즉, 간단한 refcursor 변수)로 선언되어 있어야 한다. 쿼리는 EXECUTE 명령에서와 같은 방식으로, 문자열 표현으로 지정된다. 늘 그렇듯이 이것은 유연성을 제공하므로 쿼리 계획은 실행마다 달라질 수 있으며, 또한 명령 문자열에서 변수 대체가 수행되지 않음을 의미한다. EXECUTE와 마찬가지로 매개변수 값을 format() 및 USING을 통해 동적 명령에 삽입할 수 있다. SCROLL 및 NO SCROLL 옵션은 바운드 커서와 동일한 의미를 갖는다.
예제는 다음과 같다.

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING keyvalue;
```

이 예제에서 테이블 이름은 format()을 통해 쿼리에 삽입된다. col1의 비교 값은 USING 매개변수를 통해 삽입되므로 따옴표가 필요하지 않다.

3. 바운딩 커서 열기

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

이 OPEN 형식은 쿼리가 선언될 때 쿼리가 바인드된 커서 변수를 여는데 사용된다. 커서를 더이상 열 수는 없다. 실제 인수값 표현식의 목록은 커서가 인수를 취하도록 선언된 경우에만 사용될 수 있다. 이 값들은 쿼리에 대체된다.
바운드 커서에 대한 쿼리 계획은 항상 캐시 가능하다고 간주된다. 이 경우 EXECUTE는 없다. 커서의 스크롤 동작이 이미 결정 되었으므로 SCROLL 및 NO SCROLL을 OPEN에 지정할 수 없다.
인수값은 위치 또는 명명된(named) 표기법을 사용하여 전달할 수 있다. 위치 표기법에서 모든 인수는 순서대로 지정된다. 명명된 표기법에서 각 인수의 이름은 :=을 사용하여 지정되어 인수 표현식과 구분된다. 호출 함수와 마찬가지로 위치 표기법과 명명된 표기법을 함께 사용할 수 있다.
예제는 다음과 같다.(위의 커서 선언 예제 사용)

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

변수 대체는 바운드 커서의 쿼리에서 수행되기 때문에 커서에 값을 전달하는데는 OPEN에 대한 명시적 인수를 사용하거나 쿼리에서 PL/pgSQL 변수를 참조하여 묵시적으로 값을 전달하는 두 가지 방법이 있다. 그러나 바운드 커서가 선언되기 전에 선언된 변수만이 변수로 대체한다. 두 경우 모두 전달할 값은 OPEN 시점에 결정된다. 예를 들어, 위의 curs3 예제와 같은 효과를 얻는 다른 방법은 다음과 같다.

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

커서 사용하기

커서가 열리면 아래에 설명된 명령문을 사용하여 조작할 수 있다.

이러한 조작은 커서를 처음으로 연 것과 동일한 기능으로 수행할 필요가 없다. 함수에서 refcursor값을 반환하고 호출자가 커서에서 작업하도록 할 수 있다.(내부적으로 refcursor값은 단순히 커서에 대한 활성 쿼리를 포함하는, 소위 포털의 문자열 이름이며, 이 이름은 포털을 방해하지 않고 전달할 수 있고 다른 refcursor 변수에 할당할 수 있다.)

모든 포털은 트랜잭션 종료시 묵시적으로 닫힌다. 따라서 refcursor값은 트랜잭션이 끝날 때까지 열린 커서를 참조하는 데 사용할 수 있다.

1. FETCH

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

FETCH는 SELECT INTO와 같이 행의 변수, 레코드 변수 또는 쉼표로 구분된 단순 변수의 목록에 커서의 대상으로 다음 행을 가져온다. 다음 행이 없으면 대상은 NULL(s)로 설정된다. SELECT INTO와 마찬가지로 특수 변수 FOUND를 검사하여 행을 얻었는지 여부를 확인할 수 있다.

*direction*절은 하나의 행을 가져올 수 있는 것을 제외하고는 SQL FETCH 명령에서 허용되는 변형 중 하나일 수 있다. 즉, NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*, RELATIVE *count*, FORWARD 또는 BACKWARD가 될 수 있다. *direction*을 생략하면 NEXT를 지정하는 것과 같다. 커서가 SCROLL 옵션으로 선언되거나 열려있지 않으면 뒤로 이동하는 *direction*값이 실패할 수 있다.

*cursor*는 열린 커서 포털을 참조하는 refcursor 변수의 이름이어야 한다.

예제는 다음과 같다.

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

2. MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

MOVE는 데이터를 검색하지 않고 커서의 위치를 재지정 한다. MOVE는 커서만 재배치하고 이동된 행을 리턴하지 않는다는 점을 제외하고는 FETCH 명령과 동일하게 작동한다. SELECT INTO와 마찬가지로 특수 변수 FOUND를 검사하여 이동할 다음 행이 있는지 확인할 수 있다.

*direction*절은 SQL FETCH 명령에서 사용되는 것과 같으며, 예를 들어 NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*, RELATIVE *count*, ALL, FORWARD [*count* | ALL] 또는 BACKWARD [*count* | ALL]과 같은 것들이다. *direction*의 생략은 NEXT를 지정하는 것과 같다. 커서가 SCROLL 옵션으로 선언되거나 열려있지 않으면 뒤로 이동하는 *direction*값이 실패할 수 있다.

예제는 다음과 같다.

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

커서가 테이블 행에 위치하면, 해당 행을 커서로 업데이트하거나 삭제하도록 식별할 수 있다. 커서의 쿼리 (특히, 그룹화 하지 않은) 에 대한 제한이 있으며 커서에서 FOR UPDATE를 사용하는 것이 가장 좋다.

예제는 다음과 같다.

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

4. CLOSE

```
CLOSE cursor;
```

CLOSE는 열린 커서의 기본 포털을 닫는다. 트랜잭션 종료 이전에 자원을 해제하거나 다시 열 수 있는 커서 변수를 해제하기 위해서 사용할 수 있다.

예제는 다음과 같다.

```
CLOSE curs1;
```

5. Returning Cursors

PL/pgSQL 함수는 커서를 호출자에게 반환할 수 있다. 이 방법은 여러 행이나 열을 반환할 때, 특히 매우 큰 결과 집합을 반환할 때 유용하다. 이를 위해 함수는 커서를 열고 호출자에게 커서 이름을 반환한다(또는 단순히 호출자가 지정한 또는 포털 이름을 사용하여 커서를 연다). 호출자는 커서에서 행을 가져올 수 있다. 커서는 호출자에 의해 닫히거나 커서가 닫히면 자동으로 닫힌다.

커서에 사용되는 포털 이름은 프로그래머가 지정하거나 자동으로 생성될 수 있다. 포털 이름을 지정하려면 `refcursor` 변수를 열기 전에 먼저 문자열을 할당해야 한다. `refcursor` 변수의 문자열 값은 `OPEN`에서 기본 포털의 이름으로 사용된다. 그러나 `refcursor` 변수가 null인 경우 `OPEN`은 기존 포털과 충돌하지 않는 이름을 자동으로 생성하여 `refcursor` 변수에 지정한다.

> 참고 : bound 커서 변수는 이름을 나타내는 문자열 값으로 초기화되므로, 프로그래머가 커서를 열기 전에 할당에 의해 무시하지 않는한 포털 이름은 커서 변수 이름과 동일하다. 그러나 unbound 커서 변수의 기본값은 처음에는 null 값으로 설정되므로, 무시되지 않는 한 자동으로 생성된 고유한 이름을 받는다.

다음 예제는 호출자가 커서 이름을 제공 할 수 있는 방법 중 하나이다.

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

다음 예제에서는 자동 커서 이름 생성을 사용한다.

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
```

```
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

다음 예제에서는 단일 함수에서 여러 커서를 반환하는 하나의 방법이다.

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

커서 결과를 통한 루프

커서로 반환된 행을 반복할 수 있는 FOR문이 있으며, 구문은 다음과 같다.

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ] LOOP
    statements
END LOOP [ label ];
```

커서 변수는 선언될 때 일부 쿼리에 바인딩 되어야하며 **열려 있을 수 없다.** FOR문은 자동으로 커서를 열고 루프가 종료될 때 커서를 닫는다. 실제 인수값 표현 목록은 커서가 인수를 취하도록 선언된 경우에만 나타나야 한다. 이 값은 OPEN과 동일한 방식으로 쿼리에서 대체된다.
recordvar 변수는 자동으로 유형 record로 정의되며 루프 내에만 존재한다(변수 이름의 기존 정의는 루프 내에서 무시된다). 커서에 의해 반환된 각 행은 이 레코드 변수에 할당되고 루프 본문이 실행된다.

7.2.8 Errors and Messages

오류 및 메세지 보고

RAISE문을 사용하여 메시지를 보고하고 오류를 발생시킨다.

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

level 옵션은 오류 심각도를 지정한다. 허용되는 수준은 DEBUG, LOG, INFO, NOTICE, WARNING 및 EXCEPTION이며 EXCEPTION이 기본값이다. EXCEPTION은 오류를 발생시킨다(일반적으로 현재 트랜잭션을 중단한다). 다른 레벨은 다른 우선순위 레벨의 메세지만을 생성한다. 특정 우선순위의 메시지가 클라이언트에 보고되거나, 서버 로그에 기록되는지, 또는 둘 다 보고되는지는 log_min_messages 및 client_min_messages 구성 변수에 의해 제어된다.
*level*이 있는 경우, *format*을 작성할 수 있다(표현식이 아닌 단순 문자열 리터럴이어야 한다). 형식 문자열(format string)은 보고할 오류 메시지 텍스트를 지정한다. 형식 문자열 다음에는 선택적 인수 표현식이 메시지 뒤에 삽입될 수 있다. 형식 문자열 내에서 %는 다음 선택적 인수값의 문자열 표현으로 대체된다. 리터럴 %를 출력하려면 %%로 작성해야 한다. 인수의 수는 형식 문자열의 %자리 표시자 수와 일치해야 하며, 그렇지 않으면 함수를 컴파일 하는 중에 오류가 발생한다.
이 예제에서 v_job_id의 값은 문자열의 %를 대체한다.

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

USING 뒤에 *option = expression* 항목을 쓰면 오류 보고서에 추가 정보를 첨부할 수 있다. 각 *expression*은 문자열 값 표현식이 될 수 있다. 허용되는 *option* 키워드는 다음과 같다.

- MESSAGE

  오류 메시지 텍스트를 설정한다. 이 옵션은 USING 이전의 형식 문자열을 포함하는 RAISE 형식으로 사용할 수 없다.

- DETAIL

  오류 세부 메시지를 제공한다.

- HINT

  힌트 메세지를 제공한다.

- ERRCODE

  조건 이름별 또는 5자리 SQLSTATE 코드로 직접 보고할 오류 코드(SQLSTATE)를 지정한다.

- COLUMN, CONSTRAINT, DATATYPE, TABLE, SCHEMA

  관련 객체의 이름을 제공한다.

이 예제는 주어진 오류 메시지와 힌트로 트랜잭션을 중단한다.

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user ID';
```

이 두 예제는 SQLSTATE를 설정하는 동일한 방법을 보여준다.

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

주(main) 인수가 보고될 조건 이름 또는 SQLSTATE인 두 번째 RAISE 구문이 있다. 예를 들면 다음과 같다.

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

또 다른 방식은 RAISE USING 또는 RAISE *level* USING을 작성하고 다른 모든 것을 USING 목록에 넣는 것이다. RAISE의 마지막 변형에는 매개변수가 전혀 없다. 이 형식은 BEGIN 블록의 EXCEPTION절 내에서만 사용할 수 있다. 현재 처리 중인 오류를 다시 발생시킨다.

RAISE EXCEPTION 명령에 조건 이름이나 SQLSTATE가 지정되지 않은 경우, 기본값으로 RAISE_EXCEPTION (P0001)이 사용된다. 메시지 텍스트가 지정되지 않은 경우, 기본값으로 메시지 이름은 조건 이름 또는 SQLSTATE이 된다.

> 참고 : SQLSTATE 코드로 오류 코드를 지정하는 경우, 사전 정의된 오류 코드에 국한되지 않고 5자리 숫자 그리고/또는 00000 이외의 대문자 ASCII 문자로 구성된 오류 코드를 선택할 수 있다. 3개의 0으로 끝나는 오류 코드는 카테고리 코드이므로, 전체 카테고리를 트래핑하여만 트랩될 수 있기 때문에 오류 코드가 발생한다.

평가(assertion) 확인

ASSERT문은 PL/pgSQL 함수에 디버깅 검사를 삽입하기 위한 편리한 속기법(shorthand)이다.

```
ASSERT condition [ , message ];
```

*condition*은 항상 참으로 평가될 것으로 예상되는 Boolean expression이다. 참이면 ASSERT문은 수행하지 않는다. 결과가 거짓 또는 null인 경우, ASSERT_FAILURE 예외가 발생한다(*condition*을 평가하는 중에 오류가 발생하면 정상적인 오류로 보고된다.)

선택적인 *message*가 제공되면, *condition*이 실패할 경우 결과(null이 아닌 경우)가 기본 오류 메시지 텍스트 ``assertion failed''을 대체한다. 평가가 성공하는 정상적인 경우에는 *message* 표현식이 수행되지 않는다.

ASSERT 테스트는 boolean값을 사용하는 설정 매개변수 plpgsql.check_asserts를 통해 활성화 또는 비활성화 할 수 있다. 기본값은 on 상태이다. 이 매개변수가 off되어 있으면 ASSERT문은 아무 작업도 수행하지 않는다.

ASSERT는 일반적인 오류 상황을 보고하는 것이 아니라 프로그램 버그를 탐지하기 위한 것이다. 오류 상황 보고를 위해서는 위에서 설명한 RAISE문을 사용하여야 한다.

7.2.9 Trigger Procedures

PL/pgSQL은 데이터 변경 또는 데이터베이스 이벤트에 대한 트리거 프로시저를 정의하는데 사용할 수 있다. 트리거 프로시저는 CREATE FUNCTION 명령으로 작성되며, 인수가 없고 trigger의 리턴 유형(데이터 변경 트리거의 경우) 또는 event_trigger(데이터베이스 이벤트 트리거의 경우)로 선언된다. PG_*something*이라는 특수 로컬 변수는 호출을 트리거한 조건을 설명하기 위해 자동으로 정의된다.

데이터 변경 트리거

데이터 변경 트리거는 인수없이 트리거의 반환 유형이 있는 함수로 선언된다. 함수는 CREATE TRIGGER에 지정된 인수를 받을 것으로 예상 되더라도 인수없이 선언해야 한다. 이러한 인수는 아래 설명된 대로 TG_ARGV를 통해 전달된다.

PL/pgSQL 함수가 트리거로 호출되면 최상위 블록에 여러 개의 특수 변수가 자동으로 생성된다. 그 항목들은 아래와 같다.

- NEW

  데이터 유형 RECORD; 행 레벨 트리거에서 INSERT/UPDATE 작업에 대해 새 데이터베이스 행을 보유하는 변수. 이 변수는 명령문 레벨(statement-level) 트리거 및 DELETE 작업에 대해 지정되지 않는다.

- OLD

  데이터 유형 RECORD; 행 레벨 트리거의 UPDATE/DELETE 작업에 대해 이전 데이터베이스 행을 보유하는 변수. 이 변수는 명령문 레벨 트리거 및 INSERT 작업에 대해 지정되지 않는다.

- TG_NAME

  데이터 유형 name; 실제로 트리거된 트리거의 이름이 들어있는 변수.

- TG_WHEN

  데이터 유형 text; 트리거의 정의에 따라 BEFORE, AFTER 또는 INSTEAD OF의 문자열.

- TG_LEVEL

  데이터 유형 text; 트리거의 정의에 따라 ROW 또는 STATEMENT의 문자열.

- TG_OP

  데이터 유형 text; 트리거가 시작된 작업을 알려주는 INSERT, UPDATE, DELETE 또는 TRUNCATE 문자열.

- TG_RELID

  데이터 유형 oid; 트리거 호출을 일으킨 테이블의 오브젝트 ID.

- TG_TABLE_NAME

  데이터 유형 name; 트리거 호출을 일으킨 테이블의 이름.

- TG_TABLE_SCHEMA

  데이터 유형 name; 트리거 호출을 일으킨 테이블의 스키마 이름.

- TG_NARGS

  데이터 유형 integer; CREATE TRIGGER문에서 트리거 프로시저에 제공된 인수의 수.

- TG_ARGV[]

  데이터 유형 text. CREATE TRIGGER문의 인수 인덱스는 0부터 시작한다. 유효하지 않은 인덱스(0보다 작거나 tg_nargs보다 크거나 같음)는 NULL값을 갖는다.

트리거 함수는 트리거가 실행된 테이블의 구조와 정확히 일치하는 NULL 또는 레코드/행 값을 반환해야 한다.

BEFORE에 의해 트리거된 행 레벨의 트리거는 트리거 관리자에게 이 행에 대한 나머지 작업을 건너뛰도록 신호를 보내기 위해 null을 반환할 수 있다(즉, 후속 트리거가 실행되지 않고 이 행에 대해 INSERT/UPDATE/DELETE가 발생하지 않는다). nonnull값이 리턴되면 작업은 해당 행 값으로 진행한다. NEW의 원래 값과 다른 행 값을 리턴하면 삽입되거나 갱신될 행이 변경된다. 따라서 트리거 함수가 행 값을 변경하지 않고 트리거링 조치가 정상적으로 성공하기를 원하면 NEW(또는 이에 상응하는 값)가 리턴되어야 한다. 저장할 행을 변경하려면 NEW에서 직접 단일값을 바꾸고 수정된 NEW를 반환하거나 반환할 완전한 새 레코드/행을 작성하는 것이 가능하다. DELETE의 before-trigger의 경우, 반환 값은 직접적인 영향을 미치지 않지만 트리거 동작을 계속 진행하려면 null이 아니어야 한다. DELETE 트리거에서는 NEW가 null이므로 반환하는 것은 일반적으로 의미가 없다. DELETE 트리거의 일반적인 관용어는 OLD를 반환하는 것이다.

INSTEAD OF 트리거(항상 row-level의 트리거이며, 뷰에서만 사용할 수 있다.)는 업데이트를 수행하지 않았음을 알리기 위해 null을 반환할 수 있으며, 이 행에 대한 나머지 작업을 건너 뛸 수 있어야 한다(예 : 트리거는 시작되지 않으며 행은 주변 INSERT/UPDATE/DELETE에 대한 행 영향 상태에서 계산되지 않는다. 그렇지 않으면 트리거가 요청된 조작을 수행했음을 알리기 위해 null이 아닌 값이 리턴되어야 한다. INSERT 및 UPDATE 작업의 경우, 리턴 값은 NEW여야 하며, 트리거 기능이 INSERT RETURNING 및 UPDATE RETURNING을 지원하도록 수정할 수 있다(이는 후속 트리거에 전달된 행 값에 영향을 미치거나 또는 INSERT문의 ON CONFLICT DO UPDATE절에 포함된 특정 EXCLUDED 별명 참조로 전달된다). DELETE 작업의 경우, 리턴 값은 OLD이어야 한다.

BFORE 또는 AFTER 후에 실행된 row-level 트리거의 리턴 값은 항상 무시된다. 그것은 null일 수도 있다. 그러나 이러한 유형의 트리거 중 하나라도 오류가 발생하면 전체 작업이 중단될 수 있다.

아래의 예제는 PL/pgSQL에서 트리거 프로시저의 예를 보여준다.
이 예제 트리거는 테이블에 행이 삽입되거나 업데이트 될 때마다 현재 사용자 이름과 시간이 행에 스탬프 처리되도록 한다. 그리고 직원의 이름이 주어졌으며 급여가 양수인지 확인한다.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when they must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
```

```
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

테이블에 변경 사항을 기록하는 또 다른 방법은 각 삽입, 업데이트 또는 삭제에 대한 행을 보유하는 새 테이블을 만드는 것이다. 이 접근법은 테이블 변경을 감사하는 것으로 생각할 수 있다. 아래의 예제는 PL/pgSQL에서의 audit 트리거 프로시저의 예를 보여준다.

이 예제 트리거는 emp 테이블의 행 삽입, 업데이트 또는 삭제가 emp_audit 테이블에 기록되도록 한다. 현재 시간과 사용자 이름은 수행된 작업 유형과 함께 행에 스탬프 처리된다.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- make use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
```

```
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

이전 예제의 변형은 주 테이블을 audit 테이블에 조인하는 뷰를 사용하여 각 항목이 마지막으로 수정된 시점을 표시한다. 이 접근법은 여전히 변경 사항의 전체 audit 추적을 테이블에 기록하지만, audit 추적에 대한 간략한 뷰를 제공하며, 각 항목에 대한 audit 추적에서 파생된 마지막 수정된 timestamp만을 표시한다. 아래의 예제는 PL/pgSQL에서 뷰에 대한 audit 트리거의 예를 보여준다.

이 예제는 뷰에서 트리거를 사용하여 뷰를 갱신할 수 있게 하고, 뷰의 행 삽입, 갱신 또는 삭제가 `emp_audit` 테이블에 기록되도록 한다. 현재 시간과 사용자 이름이 수행된 작업 유형과 함께 기록되며 뷰에는 각 행의 마지막 수정 시간이 표시된다.

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
```

```
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
        --
        -- Perform the required operation on emp, and create a row in emp_audit
        -- to reflect the change made to emp.
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE PROCEDURE update_emp_view();
```

트리거의 용도 중 하나는 다른 테이블의 요약 테이블을 유지 보수하는 것이다. 결과 요약은 특정 쿼리에 대해 원래 테이블 대신 사용할 수 있다(대개 실행 시간이 크게 단축된다). 이 기술은 측정 또는 관측된 데이터 테이블(fact 테이블이라 불리는)이 매우 클 수 있는 데이터웨어 하우징에서 일반적으로 사용된다. 아래의 예제는 데이터웨어 하우스의 fact 테이블에 대한 요약 테이블을 유지 관리하는 PL/pgSQL의 트리거 프로시저의 예를 보여준다.

여기에 설명된 스키마는 부분적으로 Ralph Kimball의 **The Data Warehouse Toolkit**의 **식료품점 저장소** 예제를 기반으로 한다.

```
--
-- Main tables - time dimension and sales fact.
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);
```

```
--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- Work out the increment/decrement amount(s).
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- forbid updates that change the time_key -
            -- (probably not too onerous, as DELETE + INSERT is how most
```

```
        -- changes will be made).
        IF ( OLD.time_key != NEW.time_key) THEN
            RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                        OLD.time_key, NEW.time_key;
        END IF;

        delta_time_key = OLD.time_key;
        delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
        delta_units_sold = NEW.units_sold - OLD.units_sold;
        delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

    ELSIF (TG_OP = 'INSERT') THEN

        delta_time_key = NEW.time_key;
        delta_amount_sold = NEW.amount_sold;
        delta_units_sold = NEW.units_sold;
        delta_amount_cost = NEW.amount_cost;

    END IF;

    -- Insert or update the summary row with the new values.

    LOOP
        UPDATE sales_summary_bytime
            SET amount_sold = amount_sold + delta_amount_sold,
                units_sold = units_sold + delta_units_sold,
                amount_cost = amount_cost + delta_amount_cost
            WHERE time_key = delta_time_key;

        EXIT insert_update WHEN found;

        BEGIN
            INSERT INTO sales_summary_bytime (
                        time_key,
                        amount_sold,
```

```
                             units_sold,
                             amount_cost)
                     VALUES (
                             delta_time_key,
                             delta_amount_sold,
                             delta_units_sold,
                             delta_amount_cost
                            );

                 EXIT insert_update;

             EXCEPTION
                 WHEN UNIQUE_VIOLATION THEN
                     -- do nothing
             END;
         END LOOP insert_update;

         RETURN NULL;

     END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

이벤트 트리거

PL/pgSQL은 이벤트 트리거를 정의하는데 사용할 수 있다. AgensGraph에서 이벤트 트리거로 호출될 프로시저는 인수가 없으며 `event_trigger`가 반환 유형으로 선언되어야 한다.

PL/pgSQL 함수가 이벤트 트리거로 호출되면 최상위 블록에 여러 개의 특수 변수가 자동으로 생성된다.

- TG_EVENT

  데이터 유형 `text`; 트리거가 실행되는 이벤트를 나타내는 문자열

- TG_TAG

  데이터 유형 `text`; 트리거가 실행되는 명령 태그를 포함하는 변수.

이 예제 트리거는 지원되는 명령이 실행될 때마다 `NOTICE` 메시지를 발생시킨다.

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE PROCEDURE snitch();
```

7.2.10 Tips for Developing in PL/pgSQL

PL/pgSQL에서 개발하는 좋은 방법 중 하나는 사용자가 선택한 텍스트 편집기를 사용하여 함수를 작성하고 다른 창에서 psql을 사용하여 해당 함수를 로드하고 테스트하는 것이다. 이런 식으로 작업하는 경우 `CREATE OR REPLACE FUNCTION`을 사용하여 함수를 작성하는 것이 좋다. 그렇게 하면 파일을 다시 로드하여 함수 정의를 업데이트할 수 있다. 예제는 아래와 같다.

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

psql을 실행하는 동안 다음과 같은 함수 정의 파일을 로드하거나 다시 로드할 수 있다.

```
\i filename.sql
```

PL/pgSQL에서 개발할 수 있는 또 다른 좋은 방법은 절차적 언어로 개발을 용이하게 하는 GUI 데이터베이스 액세스 도구를 이용하는 것이다. 이러한 도구는 종종 작은 따옴표를 제거하고 기능을 더 쉽게 재생할 수 있는 편리한 기능을 제공한다.

따옴표 처리

PL/pgSQL 함수의 코드는 CREATE FUNCTION에 문자열 리터럴로 지정된다. 문자열을 작은 따옴표로 묶는 일반적인 방법을 사용하는 경우, 함수 본체 내에 있는 단일 따옴표를 두배로 늘려야 한다. 마찬가지로 모든 백슬래시도 두 개를 사용해야 한다(이스케이프 문자열 구문이 사용된다고 가정). 따옴표를 두 번 사용하는 것은 코드를 이해하는데 어려움을 줄 수 있다. 왜냐하면 사용자가 다수의 인접한 따옴표가 필요하다는 것을 쉽게 알 수 있기 때문이다. 따라서 함수 본문을 ``달러 인용'' 문자열 리터럴으로 작성하는 것이 좋다. 달러 인용 방식에서는 절대로 따옴표를 두 번 사용하지 않고 대신 필요한 각 중첩 수준에 대해 다른 달러 인용 구분 기호를 선택하도록 주의해야 한다. 예를 들어, CREATE FUNCTION 명령을 다음과 같이 작성할 수 있다.

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
$PROC$ LANGUAGE plpgsql;
```

이 경우 SQL명령의 간단한 리터럴 문자열에 따옴표를 사용하고 문자열로 어셈블할 SQL명령을 구분하려면 $$를 사용할 수 있다. $$가 포함된 텍스트를 인용해야 하는 경우 Q 등을 사용할 수 있다.

다음 차트는 달러 인용부호 없이 따옴표를 쓸 때해야 할 일을 보여준다. 달러 인용부호를 더 이해하기 쉬운 것으로 변환할 때 유용할 수 있다.

1 quotation mark

함수 본문을 시작하고 끝내려면 다음과 같이 한다.

```
CREATE FUNCTION foo() RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

작은 따옴표로 묶인 함수 본문 내에서, 따옴표는 쌍으로 나타나야 한다.

2 quotation marks

함수 본문 내부의 문자열 리터럴을 표현할때 사용하며, 예를 들면 다음과 같다.

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

달러 인용 방식에서는 다음과 같이 쓸 수 있다.

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

두 경우 모두 PL/pgSQL 파서가 정확히 알 수 있다.

4 quotation marks

함수 본문 내부의 문자열 상수에 작은 따옴표가 필요한 경우에 사용하며, 예를 들면 다음과 같다.

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

실제로 a_output에 추가된 값은 다음과 같다. : `AND name LIKE 'foobar'AND xyz`.

달러 인용 방식에서는 다음과 같이 쓸 수 있다.

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

6 quotation marks

함수 본문 내부의 문자열에서 작은 따옴표가 해당 문자열 상수 끝에 인접한 경우에 사용하며, 예를 들면 다음과 같다.

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

a_output에 추가된 값은 다음과 같다. : `AND name LIKE 'foobar'`.

달러 인용 방식에서는 다음과 같이 쓸 수 있다.

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 quotation marks

문자열 상수(8개의 인용 부호를 나타냄)에 두 개의 작은 따옴표가 필요하고 문자열 상수의 끝에 인접한다(2 개 이상). 다른 함수를 생성하는 함수를 작성하는 경우에만 필요할 것이다.

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

a_output의 값은 다음과 같다.

```
if v_... like ''...'' then return ''...''; end if;
```

달러 인용 방식에서는 다음과 같이 쓸 수 있다.

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

7.3 PL/Python

7.3.1 PL/Python Functions

PL/Python 함수는 표준 CREATE FUNCTION 구문을 통해 선언한다.

```
CREATE FUNCTION funcname (argument-list)
  RETURNS return-type
AS $$
  # PL/Python function body
$$ LANGUAGE plpythonu;
```

함수의 본문은 단순 python 스크립트이다. 함수가 호출될 때, 인수는 리스트 args로 전달된다. 명명 된 인수는 일반 변수로 Python 스크립트에 전달된다. 명명 된 인수의 사용이 일반적으로 더 읽기 쉽다. 결과는 return 또는 yield(결과 집합 문의 경우) Python 코드에서 반환된다. 반환 값을 제공하지 않으면, 파이썬은 기본값 None을 반환한다 . PL/Python은 Python의 None을 SQL Null 값으로 변환한다.
예를 들어, 두 정수 중 큰 값을 반환하는 함수는 다음과 같이 정의 할 수 있다.

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

함수 정의의 본문으로 제공된 Python코드는 Python함수로 변환된다. 예를 들어 위의 결과는 다음과 같다.

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

23456은 함수에 할당된 OID라고 가정한다.

인수는 전역변수로 설정된다. 변수가 블록에서 전역으로 다시 선언되지 않는 한, Python의 범위 지정 규칙에 의해 변수 이름 자체를 포함하는 표현식 값으로 인수 변수를 함수내에서 재할당 할 수 없는 결과를 가진다. 예를 들어 아래의 경우 동작하지 않을 수 있다.

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

x에 대입하면 x가 전체 블록에 대한 지역 변수가되기 때문에 PL/Python 함수 파라미터가 아닌 오른쪽 x는 아직 할당되지 않은 지역 변수 x를 참조한한다. global문을 사용하여 다음을 수행 할 수 있다.

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

그러나 PL/Python의 세부 구현 사항에 의존하지 않는 것이 좋으며, 함수 매개 변수를 읽기 전용으로 사용하는 것이 좋다.

7.3.2 Data Values

일반적으로 PL/Python의 목적은 AgensGraph와 Python 간 ``자연스러운'' 매핑을 제공하는 것이다. 아래의 설명은 데이터 매핑 규칙에 대한 정보이다.

Data Type Mapping

PL/Python 함수가 호출되면 인수는 AgensGraph 데이터 타입에서 해당 Python 타입으로 변환된다.

- AgensGraph boolean은 Python boolean으로 변환된다.
- AgensGraph smallint와 int는 Python int로 변환된다. AgensGraph bigint와 oid는 Python2에서는 long, Python3에서는 int로 변환된다.
- AgensGraph real과 double은 Python float로 변환된다.
- AgensGraph numeric은 Python Decimal로 변환된다. 이 타입을 사용할 수 있는 경우 cdecimal패키지에서 가지고 온다. 그렇지 않으면 표준 라이브러리의 decimal.Decimal이 사용된다. cdecimal은 decimal 보다 상당히 빠르다. Python3.3이상에서는 cdecimal이 decimal이라는 이름으로 표준 라이브러리에 통합되었으므로 더이상 차이점은 없다.
- AgensGraph bytea는 Python2에서 str로 Python3에서 bytes로 변환된다. Python2에서 문자열을 문자 인코딩없이 바이트 시퀀스로 처리해야한다.
- AgensGraph 문자열 형식을 포함한 다른 모든 데이터 형식은 Python str로 변환된다. Python2에서는 이 문자열이 AgensGraph 서버 인코딩에 있게된다. Python3에서는 모든 문자열과 같은 유니코드 문자열이 된다.
- nonscalar 데이터 타입은 아래내용을 참조한다.

PL/Python 함수를 반환할때 반환 값은 다음과 같이 함수의 선언된 AgensGraph 반환 데이터 형식으로 변환된다.

- AgensGraph 반환 타입이 boolean일 때 반환 값은 Python 규칙에 따라 사실 여부가 평가 된다. 즉 0과 빈 문자열은 false이지만 'f' 는 true에 해당한다.
- AgensGraph 반환 타입이 bytea일 때 반환 값은 각각의 Python 내장 함수를 사용하여 string(Python2) 또는 bytes(Python3) 로 변환되며 결과는 bytea로 변환된다.
- 다른 모든 AgensGraph 반환 타입의 경우 변환 값은 Python 내장 str을 사용하여 문자열로 변환되고 결과는 AgensGraph 데이터 타입의 입력함수로 전달된다. (Python 값이 float인 경우, 정밀도의 손실을 피하기 위해 str 대신 repr built-in을 사용하여 변환한다.)

Python2에서 문자열은 AgensGraph에 전달 될때 AgensGraph 서버 인코딩에 있어야한다. 현재 서버 인코딩에서 유효하지 않은 문자열은 오류를 발생시키지만 모든 인코딩 불일치를 감지 할 수 없기 때문에 이것이 올바르게 수행되지 않으면 garbage 값이 계속 발생할 수 있다. 유니코드 문자열은 올바른 인코딩으로 자동 변환되므로 이를 사용하는 것이 더 안전하고 편리 할 수 있다. Python3에서 모든 문자열은 유니코드 문자열이다.

nonscalar 데이터 타입은 아래의 내용을 참조한다.

선언된 AgensGraph 반환 타입과 실제 반환 객체의 Python 데이터 타입 사이의 논리적인 불일치는 표시되지 않으며, 어떤 경우에도 값은 반환될 것이다.

Null, None

함수에 SQL null 값이 전달되면 Python에서 인수값은 none으로 표시된다. 예를 들어, PL/Python Functions pymax 나와 있는 함수 정의는 null 입력에 대해 잘못된 대답을 반환한다. 함수 정의에 STRICT를 추가하여 좀 더 합리적인 작업을 수행 하도록 할 수 있다. null 값이 전달되면 함수는 전혀 호출되지 않고 null 결과를 자동으로 반환한다. 또는 함수 본문에서 null 입력을 확인할 수 있다.

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

위에서 볼 수 있듯이 PL/Python 함수에서 SQL Null값을 반환하려면 None값을 반환한다. 함수의 strict 여부에 관계없이 작업을 수행 할 수 있다.

Arrays, Lists

SQL 배열 값은 Python 목록으로 PL/Python에 전달된다. PL/Python 함수에서 SQL 배열 값을 반환하려면 Python 시퀀스(예: 목록 또는 튜플)를 반환한다.

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return (1, 2, 3, 4, 5)
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

Python에서 문자열은 시퀀스이므로 익숙하지 않은 Python 프로그래머는 바람직하지 못한 결과를 가져올 수 있다.

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

Composite Types

복합 타입 인수는 Python 매핑으로 함수에 전달된다. 맵핑의 요소 이름은 복합 타입의 속성 이름이다. 전달 된 행의 속성에 Null 값이 있으면 매핑에 None값이 있다. 예제는 다음과 같다.

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

Python 함수에서 행 또는 복합 타입을 반환하는 여러 가지 방법이 있다. 복합 타입 결과 예제를 수행하기 위해 아래와 같이 TYPE을 생성한다.

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

복합 결과는 다음과 같이 반환 될 수 있다.

- Sequence type (튜플 또는 목록, 인덱스를 만들 수 없기 때문에 집합은 아님) 반환된 시퀀스 오브젝트는 결과 타입이 필드와 동일한 수의 항목을 가져야한다. 인덱스가 0인 항목은 복합 유형의 첫번째 입력란에 할당되고 1은 두번째 입력란에 할당된다. 예제는 다음과 같다.

  ```
  CREATE FUNCTION make_pair (name text, value integer)
    RETURNS named_value
  AS $$
    return [ name, value ]
    # or alternatively, as tuple: return ( name, value )
  $$ LANGUAGE plpythonu;
  ```

  모든 열에 대해 SQL Null을 반환하려면 해당 위치에 none을 입력한다.

- Mapping (dictionary)

  각 결과 타입 컬럼에 대한 값은 컬럼 이름을 키로 맵핑된 값에서 검색된다. 예제는 다음과 같다.

  ```
  CREATE FUNCTION make_pair (name text, value integer)
    RETURNS named_value
  AS $$
    return { "name": name, "value": value }
  $$ LANGUAGE plpythonu;
  ```

  추가 dictionary 키/값 쌍은 무시된다. 누락된 키는 오류로 처리된다. 모든 컬럼에 대해 SQL Null 값을 반환하려면 해당 컬럼 이름을 키로 사용하여 none을 입력한다.

- Object (any object providing method `__getattr__`)

  이것은 맵핑과 동일하게 동작한다. 예제는 다음과 같다.

  ```
  CREATE FUNCTION make_pair (name text, value integer)
    RETURNS named_value
  AS $$
  ```

```
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # or simply
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
$$ LANGUAGE plpythonu;
```

OUT 파라미터가 있는 함수도 지원된다. 예제는 다음과 같다.

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple();
```

Set-returning Functions

PL/Python 함수는 칼라 또는 복합 타입의 세트를 반환할 수 있다. 반환된 객체가 내부적으로 반복자로 바뀌기 때문에 이것을 달성하는 데는 여러가지 방법이 있다. 다음 예제에서는 복합 타입이 있다고 가정한다.

```
CREATE TYPE greeting AS (
how text,
who text
);
```

설정된 결과는 다음과 같이 반환할 수 있다.

- Sequence type (tuple, list, set)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
```

```
AS $$
  # return tuple containing lists as composite types
  # all other combinations work also
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

- Iterator (any object providing `__iter__` and `next` methods)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

- Generator (`yield`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
    yield ( how, who )
```

```
    $$ LANGUAGE plpythonu;
```

OUT 파라미터 (RETURNS SETOF record 사용)가 있는 set-returning 함수도 지원된다. 예제는 다음과 같다.

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer)
 RETURNS SETOF record
AS $$
 return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

7.3.3 Sharing Data

global dictionary SD는 함수 호출 간에 데이터를 저장하는데 사용할 수 있다. 이 변수는 개별적인 정적 데이터이다. global dictionary GD는 pubilc 데이터이며, 세션 내 모든 Python 함수에서 사용할 수 있으므로 사용에 주의해야 한다.

각 함수는 Python 인터프리터에서 고유 한 실행 환경을 가지므로 global 데이터와 myfunc의 함수 인수는 myfunc2 에서 사용할 수 없다. 위에서 언급한 바와 같이 예외는 GD dictionary의 데이터 이다.

7.3.4 Anonymous Code Blocks

PL / Python은 DO문과 함께 호출되는 익명 코드 블록도 지원한다.

```
DO $$
    # PL/Python code
$$ LANGUAGE plpythonu;
```

익명 코드 블록은 인수를 받지 않으며 반환되는 값은 모두 버려진다. 그 외에는 함수처럼 동작한다.

7.3.5 Trigger Functions

함수가 트리거로 사용될 때 dictionary TD는 트리거 관련 값을 포함한다.

TD[``event'']

이벤트를 문자열 (INSERT, UPDATE, DELETE 또는 TRUNCATE) 로 포함한다.

TD[``when'']

BEFORE, AFTER 또는 INSTEAD OF 중 하나를 포함한다.

TD[``level'']

ROW 또는 STATEMENT를 포함한다.

TD[``new''], TD[``old'']

행 레벨 트리거의 경우 필드 중 하나 또는 두가지 모두 트리거 이벤트에 따라 각각의 트리거 행을 포함한다.

TD[``name'']

트리거 이름을 포함한다.

TD[``table_name'']

트리거가 발생한 테이블의 이름을 포함한다.

TD[``table_schema'']

트리거가 발생한 테이블의 스키마를 포함한다.

TD[``relid'']

트리거가 발생한 테이블의 OID를 포함한다.

TD[``args'']

인수가 포함된 CREATE TRIGGER 명령어의 경우, TD["args"][0]에서 TD["args"][n-1]까지 사용할 수 있다.

TD["when"]이 BEFORE 또는 INSTEAD OF이고 TD["level"]이 ROW 일때, Python 함수에서 None 또는 ``OK''를 반환하여 행이 변경되지 않았음을 나타낼 수 있다. ``SKIP''은 이벤트를 중단 한다. TD["event"]가 INSERT 또는 UPDATE를 적용하면 ``MODIFY''를 반환하여 새로운 행을 수정할 수 있다. 그렇지 않으면 반환 값이 무시된다.

7.3.6 Database Access

PL/Python 언어 모듈은 plpy로 불리는 Python 모듈을 자동으로 가져온다. 이 모듈의 함수와 상수는 Python 코드에서 plpy.foo로 사용할 수 있다.

Database Access Functions

plpy 모듈은 데이터베이스 명령을 실행하는 여러가지 기능을 제공한다.

plpy.execute(query [, max-rows])

쿼리 문자열에서 plpy.execute를 호출하고 선택적 행 제한 인수를 선택하면 쿼리가 실행되고 결과 개체에서 결과가 반환된다.

결과 오브젝트는 목록 또는 dictionary 오브젝트를 에뮬레이션한다. 결과 오브젝트는 행 번호와 컬럼 이름으로 엑세스 할 수 있다. 예제는 다음과 같다.

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

my_table에서 최대 5개의 행을 반환한다. my_table에 my_column 컬럼이 있으면 컬럼명으로 엑세스 하는 예는 다음과 같다.

```
foo = rv[i]["my_column"]
```

반환된 행의 수는 내장 `len` 함수를 사용하여 획득 할 수 있다.

결과 오브젝트는 다음과 같은 추가 메서드가 있다.

- nrows() 명령에 의해 처리 된 행 수를 반환한다. 이것은 반드시 반환 된 행의 수와 같지는 않다. 예를 들어 `UPDATE` 명령은 값을 설정하지만 `RETURNING`을 사용하지 않는 한 모든 행을 리턴하지 않는다.

- status() `SPI_execute()` 값을 반환한다.

- colnames(), coltypes(), coltypmods() 컬럼 이름의 목록, 컬럼 타입 OID의 목록, 컬럼에 대한 타입별 형식 수정 목록을 각각 반환한다. 이러한 메소드는 결과 셋을 생성하지 않는 명령 (예 : `RETURNING`이 없는 `UPDATE`, `DROP TABLE`)에서 결과 오브젝트를 호출 할때 예외를 발생시킨다. 그러나 0 행을 포함하는 결과 셋에서 이러한 메소드 사용하는 것은 문제 없다.

- **str**() 표준 `__str__` 메서드는 예를 들어 `plpy.debug(rv)`를 사용한 쿼리 실행 결과를 디버그 할 수 있도록 정의된다.

결과 오브젝트는 수정할 수 있다.

`plpy.execute`를 호출하면 전체 결과 집합이 메모리로 읽혀진다. 결과 집합이 상대적으로 작을 때만 해당 함수를 사용한다. 큰 결과를 가져올 때 과도한 메모리 사용을 원하지 않으면 `plpy.excute`대신 `plpy.cusor`를 사용한다.

plpy.prepare(query [, argtypes]) plpy.execute(plan [, arguments [, max-rows]])

`plpy.prepare`은 쿼리 실행 계획을 준비한다. 쿼리내에 팜조하는 파라미터가 있는 경우 쿼리 문자열과 파라미터 타입의 목록과 함께 호출된다. 예제는 다음과 같다.

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", ["text"])
```

`text`는 $1에 전달 할 변수의 유형이다. 매개 변수를 쿼리에 전달하지 않으려는 경우 두번째 인수는 선택 사항이다. 명령문을 준비한 후에는 함수의 변형을 사용하여 명령문 `plpy.execute`을 실행한다.

```
rv = plpy.execute(plan, ["name"], 5)
```

플랜을 첫번째 인수 (쿼리 문자열 대신) 로 전달하고, 대체할 값 목록을 두번째 인수로 전달한다. 파라미터를 예상할 수 없다면 두번째 인수는 선택적이다. 세 번째 인수는 이전과 같이 선택적으로 행을 제한한다.

쿼리 파라미터와 결과 행 필드는 Data Values에서 설명한 대로 AgensGraph와 Python 데이터 타입이 변환된다.

PL/Python 모듈을 사용하여 플랜을 준비할때 이것은 자동으로 저장된다. 자세한 설명은 링크를 참조한다. 이 기능을 통해 이 함수를 효과적으로 사용하려면 영구 저장 사전 `SD` 또는 `GD` 중 하나를 사용해야 한다. (Sharing Data 참조.) 예제는 다음과 같다.

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if "plan" in SD:
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # rest of function
$$ LANGUAGE plpythonu;
```

plpy.cursor(query) plpy.cursor(plan [, arguments])

`plpy.cursor` 함수는 `plpy.execute`(행 제한 제외) 와 같은 인수를 받아 들이고 작은 조각으로 큰 결과 셋을 처리할 수 있는 커서 오브젝트를 반환한다. `plpy.execute`와 마찬가지로 쿼리 문자열 또는 plan 객체와 함께 인수 목록을 사용할 수 있다.

커서 오브젝트는 정수 파라미터를 받아 들여 결과 오브젝트를 반환하는 `fetch` 메서드를 제공한다. `fetch`를 호출할 때마다 반환 된 객체에는 파라미터 값보다 크지 않은 다음 행의 배치가 포함된다. 모든 행을 사용하면 `fetch`는 빈 결과 개체를 반환하기 시작한다. 또한 커서 객체는 반복 인터페이스를 제공히여 모든 행이 사용 될때까지 한번에 하나의 행을 생성한다. 데이터를 가져온 결과 오브젝트를 반환하는 방법이 아닌 dictionary에 각각의 단일 결과 행에 해당하는 dictionary로 추가된다.

큰 테이블에서 데이터를 처리하는 두 가지 방법의 예는 다음과 같다.

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
```

```
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 <> 0", ["integer"])
rows = list(plpy.cursor(plan, [2]))

return len(rows)
$$ LANGUAGE plpythonu;
```

커서는 자동으로 삭제되나 커서가 보유한 모든 자원을 명시 적으로 해제하려면 close 메소드를 사용한다. 한번 닫힌 커서는 더이상 가져올 수 없다.

Trapping Errors

데이터베이스에 액세스하는 함수에 오류가 발생하면 중단되고 예외가 발생한다. plpy.execute와 plpy.prepare 모두 기본적으로 함수를 종료하는 plpy.SPIError의 하위 클래스 인스턴스를 발생할 수 있다. 이 오류는 다른 Python 예외와 마찬가지로 try/except 구조를 사용하여 처리 할 수 있다. 예제는 다음과 같다.

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
        return "something went wrong"
    else:
        return "Joe added"
$$ LANGUAGE plpythonu;
```

제기 된 예외의 실제 클래스는 오류를 유발 한 특정 조건에 해당한다. 가능한 조건 목록은 링크을 참조한다. plpy.spiexceptions 모듈은 각 조건에 대한 예외 클래스를 정의하여 조건 이름에서 그것의 이름을 파생시킨다. division_by_zero는 DivisionByZero, unique_violation는 UniqueViolation, fdw_error는 FdwError와 같은 예가 있다. 이러한 예외 클래스 각각은 SPIError를 상속한다. 이러한 분류는 특정 에러를 쉽게 다룰수 있다. 예를 들면 다음과 같다.

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int", "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpythonu;
```

plpy.spiexceptions 모듈의 모든 예외는 SPIError를 상속하므로 except 처리 절은 모든 데이터베이스 액세스 오류를 catch 한다.
다른 오류 조건을 처리하는 다른 방법으로 SPIError 예외를 catch 할 수 있고 예외 오브젝트의 sqlstate 속성을 보고 except 블록 내부의 특정 오류 조건을 판별 할 수 있다. 이 속성은 ``SQLSTATE'' 오류 코드가 들어있는 문자열 값이다. 이 방법은 거의 동일한 기능을 제공한다.

7.3.7 Explicit Subtransactions

Trapping Errors에 설명된 대로 데이터베이스 액세스에 의해 발생한 오류를 복구 하면 일부 작업이 실패하기 전에 일부 작업이 성공하는 바람직하지 않은 상황이 발생할 수 있으며, 해당 오류를 복구 한 후 데이터의 일관성이 유지된다. PL/Python은 명시적 서브 트랜젝션의 형태로 이 문제의 솔루션을 제공한다.

Subtransaction Context Managers

두 계정간에 전송을 구현하는 함수를 살펴본다.

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
```

```
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

두번째 UPDATE문에서 예외가 발생하면 함수는 에러를 보고하지만 첫번째 UPDATE의 결과는 커밋된다. 즉, fund는 Joe의 계좌에서 인출되지만 Mary의 계좌로 이체되지 않는다.

이러한 문제를 피하기 위해 plpy.execute 호출을 명시적 서브 트랜잭션으로 래핑 할 수 있다. plpy 모듈은 plpy.subtransaction()로 생성되는 명시적 서브 트랜젝션을 관리하기 위한 도우미 오브젝트를 제공한다. 이 함수로 만들어진 오브젝트는 컨텍스트 매니저 인터페이스를 구현한다. 명시적 서브 트랜잭션을 사용하여 함수를 다음과 같이 작성할 수 있다.

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

try/catch의 사용은 여전히 필요하다. 그렇지 않으면 예외는 Python 스택의 맨위로 전달되고, AgensGraph 오류로 전체 함수가 중단되어 operation 테이블에 아무런 행이 입력되지 않는다. 서브 트랜잭션 컨텍스트 매니저는 오류를 잡아내지 않으며 범위 내에서 실행되는 모든 데이터베이스 조작이 단일적 (atomically) 으로 커밋되거나 롤백된다. 서브 트랜잭션 블록의 롤백은 예외 종료의 종류로 발생하며, 데이터베이스 액세스에서 비롯된 오류로 발생한다. 명시적 서브 트랜잭션 블록 내에서 제기되는 일반적인 Python 예외 또한 롤백을 되기 위해 서브 트랜잭션을 발생한다.

Older Python Versions

WITH 키워드를 사용하는 컨텍스트 관리자 구문은 기본적으로 Python 2.6에서 사용할 수 있다. PL/Python을 이전 버전의 Python과 함께 사용하는 경우 명시적 서브 트랜잭션을 명료하게 사용할 수는 없지만 사용가능 하다. 편리한 enter 및 exit 별칭을 사용하여 서브 트랜젝션 매니저의 __enter__ 및 __exit__ 함수를 호출 할 수 있다. FUND를 이전하는 함수 예제는 다음과 같이 작성할 수 있다.

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
try:
    subxact = plpy.subtransaction()
    subxact.enter()
    try:
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
    except:
        import sys
        subxact.exit(*sys.exc_info())
        raise
    else:
        subxact.exit(None, None, None)
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

7.3.8 Utility Functions

plpy 모듈은 함수를 제공한다.

```
plpy.debug(msg, **kwargs)
plpy.log(msg, **kwargs)
plpy.info(msg, **kwargs)
```

plpy.notice(msg, **kwargs)

plpy.warning(msg, **kwargs)

plpy.error(msg, **kwargs)

plpy.fatal(msg, **kwargs)

plpy.error와 plpy.fatal은 실제로 호출되지 않은 Python예외를 발생하여 현재 트랜잭션 또는 하위 트랜잭션이 중단되도록 한다. raise plpy.Error(msg)와 raise plpy.Fatal(msg)는 각각 plpy.error(msg)와 plpy.fatal(msg)를 호출하는 것과 동일하지만 raise형식에서는 키워드 인수를 전달 할 수 없다. 다른 함수는 다른 우선순위 수준의 메세지만 발생한다.특정 우선 순위 메세지를 클라이언트에 보고할지 서버 로그에 기록할지, 둘다 구성변수의 log_min_messages와 client_min_messages로 제어를 할지 결정할 수 있다. 자세한 내용은 링크를 참조한다.
msg인수는 위치 인수로 제공된다.이전 버전과의 호환성을 위해 두 개 이상의 위치 인수가 제공 될 수 있다. 이 경우 위치 인수의 튜플 문자열 표현은 클라이언트에 보고된 메세지가 된다.
다음 키워드 전용 인수가 허용된다.

detail

hint

sqlstate

schema_name

table_name

column_name

datatype_name

키워드 전용 인수로 전달 된 객체의 문자열 표현은 클라이언트에 보고된 메세지를 강화 하는데 사용된다. 예는 다음과 같다.

```
CREATE FUNCTION raise_custom_exception() RETURNS void AS $$
plpy.error("custom exception message",
           detail="some info about exception",
           hint="hint for users")
$$ LANGUAGE plpythonu;


=# SELECT raise_custom_exception();
ERROR:  plpy.Error: custom exception message
DETAIL:  some info about exception
HINT:  hint for users
CONTEXT:  Traceback (most recent call last):
```

```
  PL/Python function "raise_custom_exception", line 4, in <module>
    hint="hint for users")
PL/Python function "raise_custom_exception"
```

유틸리티 함수의 또 다른 세트는 `plpy.quote_literal(string)`, `plpy.quote_nullable(string)`, `plpy.quote_ident(string)`이다. 내장 인용 함수와 같다. 임시 쿼리를 작성할 때 유용하며 동적 PL/Python은 다음과 예와 같다.

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

7.3.9 Environment Variables

Python 인터프리터에서 허용되는 환경변수 중 일부는 PL/Python 동작에 영향을 줄 수 있다. 환경변수는 시작 스크립트의 예와 같이 주요 AgensGraph 서버 프로세스의 환경에서 설정해야 한다. 사용 가능한 환경 변수는 Python 버전에 따라 다르다. 자세한 내용은 Python 설명서를 참조한다.

- PYTHONHOME
- PYTHONPATH
- PYTHONY2K
- PYTHONOPTIMIZE
- PYTHONDEBUG
- PYTHONVERBOSE
- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE
- PYTHONHASHSEED

(이것은 PL/Python의 제어를 넘어서는 Python 구현 세부 사항으로, python man 페이지에 나열된 환경 변수 중 일부 는 명령행 인터프리터에서만 유효하며 임베디드 Python 인터프리터에서는 유효하지 않다.)

8 Appendix

8.1 AgensGraph Error Codes

AgensGraph 서버에 의해 생성된 모든 메세지는 ``SQLSTATE'' 코드에 대한 SQL 표준 규칙을 따르는 5 문자 오류 코드가 지정된다.

표준에 따르면 오류 코드의 처음 두 문자는 오류 클래스를 나타내며 마지막 세 문자는 해당 클래스 내의 특정 조건을 나타낸다. 따라서 특정 오류 코드를 인식하지 못하는 응용프로그램은 오류를 인식하지 못하여 여전히 오류 클래스에서 수행할 작업을 예상할 수 있다.

아래의 표는 AgensGraph에 정의 된 모든 오류 코드이다. (현재 일부는 실제로 사용되지 않지만 SQL 표준에 의해 정의된다.) 오류 클래스도 표시된다. 각 오류 클래스에는 마지막 세 문자 000을 갖는 ``표준'' 오류 코드가 있다. 이 코드는 클래스 내에 있지만 더 이상 특정 코드가 할당되지 않은 오류 조건에만 사용된다.

``Condition Name''에 표시된 기호는 PL/pgSQL에서 사용하는 조건 이름이다. 조건 이름은 대문자 또는 소문자로 작성할 수 있다. (PL/pgSQL은 오류와 달리 00,01,02 클래스의 조건 이름은 경고를 인식하지 못한다.)

일부 오류 유형의 경우, 서버는 오류와 연관된 데이터베이스 오브젝트 (테이블, 테이블 컬럼, 데이터 유형 또는 제한 조건) 의 이름을 보고한다. unique_isolation 오류를 발생시킨 고유한 제약 조건의 이름을 예로 들 수 있다. 이러한 이름은 오류 리포트 메세지의 별도 필드에 제공되므로 응용프로그램에서 사람이 읽을 수 있는 메세지의 텍스트로 해당 이름을 추출하지 않아도 된다.

Error Code	Condition Name
Class 00 - Successful Completion	
00000	successful_completion
Class 01 - Warning	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
Class 02 - No Data	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
Class 03 - SQL Statement Not Yet Complete	

Error Code	Condition Name
03000	sql_statement_not_yet_complete
Class 08 - Connection Exception	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
Class 09 - Triggered Action Exception	
09000	triggered_action_exception
Class 0A - Feature Not Supported	
0A000	feature_not_supported
Class 0B - Invalid Transaction Initiation	
0B000	invalid_transaction_initiation
Class 0F - Locator Exception	
0F000	locator_exception
0F001	invalid_locator_specification
Class 0L - Invalid Grantor	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Class 0P - Invalid Role Specification	
0P000	invalid_role_specification
Class 0Z - Diagnostics Exception	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Class 20 - Case Not Found	
20000	case_not_found
Class 21 - Cardinality Violation	
21000	cardinality_violation
Class 22 - Data Exception	
22000	data_exception
2202E	array_subscript_error

Error Code	Condition Name
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
22026	string_data_length_mismatch
22001	string_data_right_truncation

Error Code	Condition Name
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
Class 23 - Integrity Constraint Violation	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24 - Invalid Cursor State	
24000	invalid_cursor_state
Class 25 - Invalid Transaction State	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction

Error Code	Condition Name
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
Class 26 - Invalid SQL Statement Name	
26000	invalid_sql_statement_name
Class 27 - Triggered Data Change Violation	
27000	triggered_data_change_violation
Class 28 - Invalid Authorization Specification	
28000	invalid_authorization_specification
28P01	invalid_password
Class 2B - Dependent Privilege Descriptors Still Exist	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Class 2D - Invalid Transaction Termination	
2D000	invalid_transaction_termination
Class 2F - SQL Routine Exception	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34 - Invalid Cursor Name	
34000	invalid_cursor_name
Class 38 - External Routine Exception	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39 - External Routine Invocation Exception	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned

Error Code	Condition Name
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Class 3B - Savepoint Exception	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Class 3D - Invalid Catalog Name	
3D000	invalid_catalog_name
Class 3F - Invalid Schema Name	
3F000	invalid_schema_name
Class 40 - Transaction Rollback	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class 42 - Syntax Error or Access Rule Violation	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation

Error Code	Condition Name
42809	wrong_object_type
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Class 44 - WITH CHECK OPTION Violation	
44000	with_check_option_violation
Class 53 - Insufficient Resources	
53000	insufficient_resources
53100	disk_full

Error Code	Condition Name
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Class 54 - Program Limit Exceeded	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Class 55 - Object Not In Prerequisite State	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
Class 57 - Operator Intervention	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
Class 58 - System Error	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Class 72 - Snapshot Failure	
72000	snapshot_too_old
Class F0 - Configuration File Error	
F0000	config_file_error
F0001	lock_file_exists
Class HV - Foreign Data Wrapper Error	
HV000	fdw_error
HV005	fdw_column_name_not_found

Error Code	Condition Name
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
Class P0 - PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Class XX - Internal Error	
XX000	internal_error

Error Code	Condition Name
XX001	data_corrupted
XX002	index_corrupted

8.2 Terminology

8.2.1 Database Cluster

- Server (or Node), 서버 (혹은 노드)

  서버는 AgensGraph가 설치된 (실제 또는 가상)하드웨어를 의미한다.

- Cluster (or `Database Cluster'), 클러스터 (혹은 데이터베이스 클러스터)

  클러스터란 파일시스템에서 저장공간(디렉터리, 서브디렉터리, 파일)을 의미한다. 데이터베이스 클러스터에는 또한 ``사용자 및 권한''과 같은 전역객체의 정의도 있다. 이런것들은 데이터베이스 전체에 영향을 끼친다. 그리고 데이터베이스 클러스터에는 최소 3개의 데이터베이스(`template0', `template1', `postgres')가 존재한다. 각각의 역할은 다음과 같다.

  template0': CREATE DATABASE 명령으로 사용할 수있는 템플릿 데이터베이스 (template0은 절대로 수정해서는 안 됨)

  template1': CREATE DATABASE 명령에 의해 사용될 수있는 템플릿 데이터베이스 (template1은 DBA에 의해 수정 될 수 있음)

  postgres': 주로 유지 보수 목적을 위한 빈 데이터베이스

- Instance (or `Database Server Instance' or `Database Server' or `Backend'), 인스턴스(혹은 데이터베이스 서버 인스턴스, 데이터베이스 서버, 또는 백엔드)

  인스턴스는 UNIX서버의 경우 프로세스의 그룹을 의미하며, Windows서버의 경우 서비스와 하나의 클러스터를 제어하고 관리하는 공유메모리를 의미한다. IP 관점에서는, 하나의 인스턴스가 하나의 IP/포트 조합을 점유한다고 생각할 수 있다. eg. http://localhost:5432. 이는 같은 서버의 다른 포트에서 다른 인스턴스를 실행할 수 있다는 의미이다. 이외에도 서버에 클러스터가 여러 개있는 경우 클러스터 당 하나의 동일한 시스템에서 많은 인스턴스를 실행하는것 또한 가능하다.

- Database, 데이터베이스

  데이터베이스는 오브젝트 컬렉션이 파일에 저장되는 파일 시스템의 저장 영역을 의미한다. 오브젝트는 데이터, 메타 데이터 (테이블 정의, 데이터 형식, 제약 조건, 뷰 등) 및 인덱스와 같은 다른 데이터로 구성되며, 이러한 객체는 기본 데이터베이스 'postgres' 또는 새로 생성된 데이터베이스에 저장된다. 하나의 데이터베이스에 대한 저장 영역은 데이터베이스 클러스터의 저장 영역 내에 하나의 서브 디렉토리 트리로 구성된다. 따라서 데이터베이스 클러스터에는 여러 데이터베이스가 포함될 수 있다.

- Schema, 스키마

  스키마는 데이터베이스 내의 이름 공간으로, 이 데이터베이스의 다른 스키마에 있는 오브젝트 이름을 복제할 수있는 이름이 지정된 오브젝트 (테이블, 데이터 유형, 함수 및 연산자) 로 구성된다. 모든 데이터베이스는 기본 스키마 `public' 을 포함하며 더 많은 스키마를 포함 할 수 있다. 한 스키마의 모든 오브젝트는 동일한 데이터베이스 내에 위치해야하며, 동일한 데이터베이스 내 다른 스키마의 오브젝트는 같은 이름을 가져도 무방하다. 그리고 각 데이터베이스에는 특수 스키마인 'pg_catalog' 가 있다. 'pg_catalog' 에는 모든 시스템 테이블, 내장 데이터 유형, 함수 및 연산자가 들어 있다.

- Search Path (or `Schema Search Path'), 검색경로 (혹은 스키마 검색 경로)

  검색 경로는 스키마 이름 목록을 의미한다. 응용 프로그램이 규정되지 않은 오브젝트 이름 (예 : 테이블 이름에 `employee_table') 을 사용하는 경우 검색 경로는 지정된 스키마 순서에 따라 이 오브젝트를 찾는 데 사용됩니다. 그리고 'pg_catalog' 스키마는 검색 경로에 명시하고지는 않지만 항상 검색 경로의 첫 번째 부분이다. 이 동작은 AgensGraph가 시스템 객체를 찾도록 한다.

- initdb (OS command)

  initdb 는 (`template0', `template1', `postgres' 를 포함한) 새 클러스터를 생성한다.

8.2.2 Consistent Writes

- Checkpoint, 체크포인트

  체크포인트란 데이터베이스 파일이 일관된 상태에 있음을 보장하는 시점을 의미한다. 체크 포인트 시간에 모든 변경 기록은 WAL 파일로, 공유버퍼의 모든 dirty data page는 디스크로의) 는 디스크로 플러시되고, 마지막으로 체크포인트 기록이 WAL 파일에 기록된다. 인스턴스의 체크포인트 프로세스는 정기적인 스케쥴로 트리거된다. 그리고 클라이언트 프로그램에서 CHECKPOINT 명령을 통해 강제 실행 또한 가능하다. 데이터베이스 시스템의 경우 디스크에 물리적으로 기록하기 때문에 체크포인트 명령을 수행하는 데 많은 시간이 필요하다.

- WAL File, WAL 파일

  WAL 파일은 INSERT, UPDATE, DELETE 또는 CREATE TABLE과 같은 명령을 통해 데이터에 적용되는 변경 사항으로 구성된다. 이는 (더 나은 성능을 위해) 데이터 파일에 기록되는 중복 정보이기도하다. 인스턴스의 설정에 따라 WAL 파일 내에 더 많은 정보가 기록되어 있을 수 있다. WAL 파일은 16MB의 고정된 크기인 바이너리 형식으로 pg_wal 디렉토리 (버전 10 이전일 경우 pg_xlog) 에 있다. 그리고 WAL 파일 내의 단일 정보 단위는 로그 레코드라고 명명한다.

- Logfile, 로그파일

  인스턴스는 특수 상황에 대한 경고 및 오류 메시지를 읽을 수있는 텍스트 파일로 기록하고 리포트한다. 여기서 기록된 로그 파일은 클러스터를 제외한 서버 내 임의의 위치에 저장할 수 있다.

- Log Record, 로그레코드

로그 레코드는 WAL 파일 내 단일 정보 단위를 의미한다.

8.3 FAQ

8.3.1 사용가능한 디버깅 기능들이 어떤 것이 있습니까?

컴파일 시간

첫째, 새로운 C 코드로 개발을 한다면 항상 --enable-cassert와 --enable-debug 옵션으로 구성된 빌드환경에서 작업해야 한다. 디버그 심볼을 활성화하면 디버거 (예. gdb) 를 사용하여 오작동하는 코드를 추적할 수 있다. gcc로 컴파일을 할 때, -ggdb -Og -g3 -fno-omit-frame-pointer라는 추가 cflag는 많은 디버깅 정보를 삽입하기 때문에 유용하다. 다음과 같이 configure에 전달할 수 있다.

```
./configure --enable-cassert --enable-debug CFLAGS="-ggdb -Og -g3 -fno-omit-frame-pointer"
```

-Og 대신에 -O0를 사용하면 인라인을 포함한 대부분의 컴파일러 최적화가 비활성화 되지만, -Og는 더 많은 디버그 정보를 제공하면서 -O2나 -Os와 같은 일반적인 최적화 플래그와 거의 비슷하게 수행된다. <value optimized out> 변수가 훨씬 적고, 실행 순서를 변경하는데 혼란이나 어려움이 덜 생기지만, 성능은 꽤 유용할 것이다. -ggdb -g3는 gcc가 매크로 정의와 같은 것들을 포함하여, 생성된 바이너리에 디버그 정보의 최대량을 포함하도록 지시한다. -fno-omit-frame-pointer는 이러한 도구가 스택의 맨 위 함수뿐만 아니라 호출 스택을 캡처 할 수 있도록 하는 프레임 포인터처럼 perf와 같은 추적 및 프로파일링 도구를 사용할 때 유용하다.

실행 시간

postgres 서버는 자세한 정보를 기록 할 수있는 -d 옵션을 가지고 있다 (elog 또는 ereport DEBUGn 출력물). -d 옵션은 디버그 레벨을 지정하는 숫자를 취한다. 높은 디버그 레벨 값은 큰 로그 파일을 생성한다는 것에 주의해야 한다. 이 옵션은 ag_ctl을 통해 서버를 시작할 때 사용할 수 없지만 대신 -o log_min_messages = debug4 또는 유사한 것을 사용할 수 있다.

gdb

postmaster가 실행 중이면, 한 창에서 agens를 시작한 다음, agens가 SELECT pg_backend_pid()를 사용하여 사용하는 postgres 프로세스의 PID를 찾는다. postgres PID -gdb -p 1234 또는 실행중인 gdb 내에서 attach 1234를 첨부하기 위해 디버거를 사용한다. 또한 gdblive 스크립트가 유용 할 수도 있다. 디버거에서 중단점을 설정 한 다음 agens세션에서 쿼리를 실행할 수 있다.
오류 또는 로그 메시지를 생성하는 위치를 찾으려면 errfinish에 중단점을 설정한다. 이렇게하면 사용 가능한 로그 수준에 대한 모든 elog 및 ereport 호출이 트랩되므로 많은 트리거가 발생할 수 있다. ERROR/FATAL/PANIC

에만 관심이 있다면,errordata[errordata_stack_depth].elevel >= 20에 대해 gdb 조건부 중단점을 사용하거나,errfinish에 PANIC, FATAL, ERROR의 경우에 소스 라인 중단점을 설정한다. 모든 오류가 errfinish를 통과하는 것은 아니다. 특히 권한 검사는 별도로 발생한다. 중단점이 트리거되지 않으면 오류 텍스트에 대해 git grep을 실행하고 어디서 던져졌는지 확인한다.

세션을 시작할 때 일어나는 일을 디버깅 할 경우 PGOPTIONS="-W n"을 설정한 다음 agens를 시작할 수 있다. 그러면 n초 동안 시작이 지연되어 디버거를 이용하여 프로세스에 연결하고 적절한 중단점을 설정한 다음 시작 순서를 계속 진행할 수 있다.

pg_stat_activity, 로그,pg_locks, pg_stat_replication 등을 보면서 디버깅을 위한 타겟 프로세스를 번갈아 가려 낼 수 있다.

8.3.2 initdb를 깨뜨렸습니다. 어떻게 디버깅 해야 합니까?

때때로 패치가 initdb 실패를 일으킬 수 있다. 이것들은 initdb 자체에서는 드물다. 더 자주 initdb에 의해 시작된 postgres 백엔드에서 몇몇 설정 작업을하기 위해 실패가 발생한다.

이들 중 하나가 충돌하는 경우, gdb를 initdb에 붙이면 그 자체로는 괜찮으며, initdb 자체가 충돌하지 않아서 gdb가 깨지지 않는다.

당신이해야 할 일은 gdb에서 initdb를 실행하고,fork에 중단점을 설정한 다음, 실행을 계속하는 것이다. 중단점을 트리거 할 때, 함수를 마치게 된다. gdb는 자식 프로세스가 생성되었다고 보고할 것이다. 그러나 이것은 당신이 원하는 것이 아니다. 그것은 실제 postgres 인스턴스를 시작한 쉘이다.

initdb가 일시정지 되어있는 동안 ps를 사용하여 시작한 postgres 인스턴스를 찾는다. pstree -p는 이것에 유용할 수 있다. 그것을 찾았으면 별도의 gdb 세션을 gdb -p $ the_postgres_pid로 첨부한다. 이 시점에서 안전하게 gdb를 initdb에서 분리하고 실패한postgres 인스턴스를 디버깅 할 수 있다.

8.3.3 구문 분석 쿼리를 변경해야합니다. 간략하게 파서 파일을 설명 할 수 있습니까?

파서 파일은 `src/backend/parser' 디렉토리에 있다.

scan.l은 문자열 (SQL문 포함) 을 토큰 스트림으로 분할하는 알고리즘인 렉서 (lexer) 를 정의한다. 토큰은 대개 단일 단어이므로 공백을 포함하지 않지만 공백으로 구분된다. 예를 들어 전체 또는 작은 따옴표로 묶은 문자열 일 수도 있다. 렉서는 기본적으로 다양한 토큰 유형을 설명하는 정규 표현식으로 정의된다.

gram.y는 렉서가 기본 빌딩 블록으로 생성한 토큰을 사용하여 SQL문의 문법 (구문 구조)을 정의한다. 문법은 BNF 표기법으로 정의된다. BNF는 정규식과 비슷하지만 문자가 아닌 토큰 수준에서 작동한다. 또한 패턴 (BNF에서 규칙 또는 제작이라고 함) 은 이름이 지정되며 재귀적일 수 있다. 즉 자체를 하위 패턴으로 사용할 수 있다.

실제 렉서는 flex라는 도구로 scan.l에서 생성된다. 매뉴얼은 http://flex.sourceforge.net/manual/에서 찾을 수 있다.

실제 구문 분석기는 bison이라는 도구로 gram.y에서 생성된다. 이 안내서는 http://www.gnu.org/s/bison/에서 찾을 수 있다.

하지만 이전에 flex나 bison을 사용해 본적이 없다면 학습하기가 조금 어려울 것이다.

8.3.4 백엔드 코드에서 시스템 카탈로그의 정보에 효율적으로 어떻게 액세스 합니까?

먼저 관심있는 튜플 (행) 을 찾아야 한다. 두 가지 방법이 있다. 첫째, SearchSysCache() 및 관련 함수를 사용하면 카탈로그의 미리 정의된 인덱스를 사용하여 시스템 카탈로그를 쿼리 할 수 있다. 캐시에 대한 첫 번째 호출이 필요한 행을 로드하고 이후 요청이 기본 테이블에 액세스하지 않고 결과를 리턴 할 수 있기 때문에 시스템 테이블에 액세스하는 기본 방법이다. 사용 가능한 캐시 목록은 src/backend/utils/cache/syscache.c에 있다. src/backend/utils/cache/lsyscache.c에는 많은 열-특정 캐시 조회 기능이 포함되어 있다.

리턴된 행은 캐시 소유 버전의 heap 행이다. 따라서 SearchSysCache() 가 반환한 튜플을 수정하거나 삭제하면 안된다. ReleaseSysCache() 를 사용하여 끝내면 릴리즈 해야 한다. 이것은 필요한 경우 해당 튜플을 삭제할 수 있음을 캐시에 알린다. ReleaseSysCache() 를 호출하지 않으면 캐시 엔트리가 트랜잭션이 끝날 때까지 캐시에 잠겨 있다. 이는 개발 중에는 용인될 수 있지만 출시 가치가 있는 코드에서는 수용 가능하지 않다.

시스템 캐시를 사용할 수 없으면 모든 백엔드가 공유하는 버퍼 캐시를 사용하여 heap 테이블에서 직접 데이터를 검색해야 한다. 백엔드는 행을 버퍼 캐시에 자동으로 로드한다. 이렇게 하려면 heap_open() 을 사용하여 테이블을 연다. 그런 다음 heap_beginscan() 을 사용하여 테이블 스캔을 시작한 다음 heap_getnext() 를 사용하고 HeapTupleIsValid() 가 true를 반환하는한 계속할 수 있다. 그런 다음 heap_endscan() 을 수행한다. 키를 스캔에 할당할 수 있다. 인덱스가 사용되지 않으므로 모든 행이 키와 비교되어 유효한 행만 반환된다.

heap_fetch() 를 사용하여 블록 번호/오프셋으로 행을 페치 할 수도 있다. 검사가 자동으로 버퍼 캐시에서 행을 잠금/잠금해제하는 동안 heap_fetch() 와 함께 버퍼 포인터를 전달해야하며 완 료되면 ReleaseBuffer() 를 전달해야 한다.

일단 행이 있으면 HeapTuple 구조 항목에 액세스하여 t_self 및 t_oid와 같은 모든 튜플에 공통된 데이터를 가져올 수 있다. 테이블 관련 열이 필요한 경우 HeapTuple 포인터를 가져 와서 GETSTRUCT() 매크로를 사용하여 테이블 관련 시작 튜플에 액세스해야 한다. 그런 다음 포인터를 캐스팅한다 (예 : pg_proc 테이블에 액세스하는 경우 Form_pg_proc 포인터 또는 pg_type에 액세스하는 경우 Form_pg_type). 그런 다음 구조체 포인터를 사용하여 튜플의 필드에 액세스 할 수 있다 : ((Form_pg_class) GETSTRUCT(tuple))->relnatts

그러나 이 방법은 고정 너비 및 null이 아닌 열과 모든 이전 열이 고정 너비 및 절대 null인 경우에만 작동한다. 그렇지 않으면 열의 위치가 가변적이므로 heap_getattr() 또는 관련 함수를 사용하여 튜플에서 추출해야 한다.

또한 라이브 튜플을 변경하는 방법으로 구조체 필드에 직접 저장하지 않는다. 가장 좋은 방법은 heap_modifytuple() 을 사용하여 원래 튜플과 변경하려는 값을 전달하는 것이다. 이것은 palloc으로 묶인 튜플을 반환하고, heap_update() 에 전달한다. 튜플의 t_self를 heap_delete() 에 전달하여 튜플을 삭제할 수 있다. heap_update() 에도 t_self를 사용한다. 튜플은 ReleaseSysCache() 를 호출한 후에 사라질 수도 있는 시스템 캐시 복사본이거나 heap_fetch() 케이

스에서 heap_getnext(), heap_endscan 또는 ReleaseBuffer() 가 사라질 때 디스크 버퍼에서 직접 읽는 시스템 캐시 복사본일 수 있다. 또는 palloc된 튜플일 수 있다. 완료되면 pfree() 를 수행해야 한다.

8.3.5 새로운 포트를 추가하려면 어떻게 해야 합니까?

새 포트를 추가하기 위해 수정해야 할 장소는 다양하다. 먼저 src/template 디렉토리에서 시작한다. 해당 OS에 적합한 항목을 추가한다. src/config.guess를 사용하여 OS를 src/template/.similar에 추가한다. OS 버전을 정확히 일치시키지 않는다. configure 테스트는 정확한 OS 버전 번호를 찾고, 발견되지 않으면 버전 번호없이 일치하는 것을 찾는다. src/configure.in을 편집하여 새 OS를 추가한다. 위의 구성 항목을 참조한다. autoconf를 실행하거나 src/configure도 패치해야 한다.

그런 다음 src/include/port를 확인하고 새 OS 파일을 적절한 값과 함께 추가한다. 다행히 이미 src/include/s-torage/s_lock.h에 CPU용 잠금 코드가 있다. 포트 별 Makefile 처리를위한 src/makefiles 디렉토리도 있다. OS에 특별한 파일이 필요한 경우 백엔드/포트 디렉토리가 있다.