

kt ds 국가인적자원개발 컨소시엄

PostgreSQL & PPAS 실무적용

2016

[목차]

PART 1. Tutorial

1장. PostgreSQL 소개	6
1.1 PostgreSQL	6
1.2 PostgreSQL 기능	7
1.3 PostgreSQL 에디션	7
2장. PostgreSQL 설치	7
2.1 설치 전 준비 사항	7
2.1.1 지원 플랫폼	7
2.1.2 하드웨어 및 소프트웨어 요구 사항	8
2.2 서버 설치	8
2.2.1 윈도우 설치	9
2.2.2 리눅스 설치	13
3장. 클라이언트 설치와 사용	15
3.1 pgAdmin	15
3.1.1 설치	15
3.1.2 PostgreSQL 접속	18
3.1.3 PostgreSQL 쿼리 실행	20
3.1.4 설정	21
3.2 psql	21
3.2.1 psql 실행	21

PART 2. 아키텍처

1장. 아키텍처	23
1.1 전체 도식	23
1.2 메모리	23
1.2.1 Shared Memory	23
1.2.2 Backend Memory	24
1.3 프로세스	25
1.3.1 Postmaster	25
1.3.2 Utility Process / User backend Process	26
1.3.3 Session	27

1.3.4 프로세스 관리	- 27 -
1.4 물리적인 디렉터리 구조	- 28 -
1.4.1 엔진 Directory	- 28 -
1.4.2 데이터 Directory	- 28 -
 PART 3. 서버관리	
1장. 서버 실행과 종료	- 31 -
1.1 서버 시작	- 31 -
1.2 서버 종료	- 31 -
1.3 시그널 전송	- 31 -
2장. 서버 설정	- 32 -
2.1 서버 환경 레벨	- 32 -
2.2 환경 변수 설정/확인	- 33 -
2.2.1 환경 변수 설정	- 33 -
2.2.2 환경 변수 확인	- 34 -
2.3 환경 변수	- 34 -
2.3.1 접속과 인증	- 34 -
2.3.2 메모리	- 34 -
2.3.3 쿼리 실행	- 35 -
2.3.4 트랜잭션 로그	- 35 -
2.3.5 체크 포인트	- 35 -
2.3.6 Writer 프로세스	- 36 -
2.3.7 Vacuum	- 36 -
2.3.8 Autovacuum	- 36 -
2.3.9 로그	- 36 -
2.3.10 Lock	- 37 -
2.4 접근 정보 제어	- 37 -
3장. 유틸리티	- 39 -
3.1 psql 명령어	- 39 -
3.2 데이터 Load	- 39 -
4장. Maintenance Tasks	- 40 -
4.1 Explain and Analyze	- 40 -

4.1.1	Explain.....	- 41 -
4.1.2	Analyze.....	- 42 -
4.2	Vacuum.....	- 43 -
4.2.1	Vacuum.....	- 43 -
4.2.2	Vacuum full	- 44 -
4.2.3	Autovacuum.....	- 44 -
5장.	모니터링	- 44 -
5.1	Catalog	- 44 -
5.2	일반 모니터링	- 48 -
5.2.1	프로세스.....	- 48 -
5.2.2	파일 시스템.....	- 49 -
5.2.3	로그	- 49 -
5.3	쿼리 모니터링	- 49 -
5.4	모니터링 툴 소개.....	- 52 -
6장.	백업과 복구.....	- 53 -
6.1	백업	- 53 -
6.1.1	Cold Backup.....	- 53 -
6.1.2	Hot Backup.....	- 53 -
6.2	복구	- 57 -
6.2.1	Cold Backup 복구	- 57 -
6.2.2	Hot Backup 복구	- 57 -

PART 4. SQL

1장.	SQL 종류	- 62 -
1.1	Query.....	- 62 -
1.2	DML.....	- 62 -
1.3	DDL.....	- 64 -
1.4	트랜잭션.....	- 65 -
2장.	데이터베이스 객체	- 66 -
2.1	개요.....	- 66 -
2.2	테이블	- 67 -
2.2.1	데이터 타입	- 67 -

2.2.2 의사 칼럼	- 68 -
2.2.3 제약 조건	- 69 -
2.2.4 테이블 종류	- 73 -
2.3 인덱스	- 78 -
2.3.1 인덱스 개념	- 78 -
2.3.2 인덱스 생성과 삭제	- 79 -
2.3.3 인덱스 종류	- 79 -
2.4 뷰	- 80 -
2.4.1 뷰 개념	- 80 -
2.4.2 뷰 활용	- 81 -
2.5 시노님	- 81 -
2.6 시퀀스	- 81 -
2.6.1 시퀀스 개념	- 81 -
2.6.2 시퀀스 활용	- 82 -
2.7 함수	- 83 -
2.7.1 문자형 함수	- 83 -
2.7.2 숫자형 함수	- 85 -
2.7.3 날짜형 함수	- 86 -
2.7.4 Null 관련 함수	- 88 -
2.7.5 변환 함수	- 89 -
2.7.6 조건식	- 90 -
2.8 연산자	- 92 -
2.8.1 논리 연산자	- 92 -
2.8.2 날짜 연산자	- 92 -
2.8.3 범위 조건	- 93 -
2.8.4 Null 처리	- 95 -
2.8.5 집합 연산자	- 96 -
3장. 실전 쿼리 작성	- 99 -
3.1 JOIN	- 99 -
3.2 그룹 쿼리	- 104 -
3.2.1 DISTINCT와 ALL	- 104 -
3.2.2 집계 함수	- 106 -

3.2.3 GROUP BY와 ORDER BY, HAVING.....	- 107 -
3.3 서브쿼리.....	- 111 -

PART 5 Stored procedure language

1장. PL/pgSQL	- 115 -
1.1 개요.....	- 115 -
1.2 기초 코딩	- 116 -
1.2.1 변수 및 상수 선언	- 116 -
1.2.2 커서	- 118 -
1.2.3 조건문.....	- 120 -
1.2.4 반복문.....	- 121 -
1.2.5 Null 문	- 124 -
1.2.6 예외 처리.....	- 125 -
1.2.7 주석	- 126 -
1.3 서브 프로그램.....	- 127 -
1.3.1 사용자 정의 함수	- 127 -
1.3.2 프로시저.....	- 128 -
1.3.3 패키지.....	- 128 -
1.3.4 트리거.....	- 128 -
1.3.5 를	- 131 -

PART 1. Tutorial

1부에서는 PostgreSQL과 EDB PAS(PostgreSQL Advanced Server)에 대해 알아봅니다. 또한 서버와 클라이언트를 설치해 봅니다.

1장. PostgreSQL 소개

현실 세계의 데이터를 효율적이고 체계적으로 관리하는 시스템이 바로 데이터베이스 시스템입니다. 현재까지 수 많은 데이터베이스 시스템이 개발되었으며, 상용제품에서부터 오픈소스SW 제품에 이르기까지 다양한 제품이 존재합니다. 상용제품은 오라클사의 오라클 데이터베이스, MS사의 MS-SQL SERVER DB, SYBASE 사의 SYBASE DB, IBM사의 DB2 등이 있습니다. 상용제품의 성능과 기능에 버금가는 오픈소스SW 제품들이 다수 있으며, 점차 시장 점유율을 높여가고 있습니다. 오픈소스SW 제품으로는 PostgreSQL, MySQL, CUBRID 등이 대표적입니다. 이 중에서 우리가 이 책에서 주로 다루고자 하는 제품은 PostgreSQL입니다. 추가적으로 EDB PAS(PostgreSQL Advanced Server)에 대해 같이 설명 드립니다. EDB PAS는 EnterpriseDB사에서 PostgreSQL에 성능, 기능, 오라클 호환성 등을 추가한 상용 오픈소스SW제품입니다.

1.1 PostgreSQL

PostgreSQL은 캘리포니아 대학의 버클리 컴퓨터 과학 학부에서 개발된 POSTGRES 버전 4.21을 기반으로 한 오픈소스 객체관계형 데이터베이스 관리 시스템(ORDBMS)입니다. PostgreSQL은 오랜 기간 지속적인 연구 개발을 통해 기능과 성능을 향상시켜 왔습니다. 그 결과 다양한 환경에서 사용 가능한 고성능의 오픈소스SW 데이터베이스시스템으로 발전하였습니다.

PostgreSQL의 전신인 POSTGRES 프로젝트는 The Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation(NSF), ESL, Inc. 의 후원을 받아 1986년에 시작되었습니다. 이후 몇 번의 주요 버전을 공개해 왔으며, 이후 1989년 6월 일부 외부 사용자에게 버전 1이 공개되었습니다. POSTGRES는 다양한 연구를 통해 실제 업무 어플리케이션 영역에서 사용 될 수 있었습니다. 하지만, Berkeley POSTGRES 프로젝트는 버전 4.2를 마지막으로 공식적으로 종료되었습니다. 이후 "Postgres95"라는 프로젝트명으로 개발되어 오다가 PostgreSQL라는 새로운 이름으로 변경하였습니다. 1996년에는 버전 명을 6.0으로 재설정하면서 현재의 PostgreSQL 9.5에 이르고 있습니다.

1.2 PostgreSQL 기능

PostgreSQL와 EDB PAS의 9.5 버전에서 지원하는 기능입니다.

○ 개발 기능

- Ansi SQL
- 다양한 함수(집계함수, 윈도우 함수 등)
- 파티션 테이블

○ 운영 기능

- MVCC(multi version concurrency control)
- 객체 관리
- 데이터베이스 암호화
- 온라인 백업 및 복구

○ 성능 기능

- Hint(EDB PAS 제공)

1.3 PostgreSQL 에디션

PostgreSQL은 현재 9.5 버전까지 출시가 되었으며, 다양한 플랫폼 환경을 지원하고 있습니다.

플랫폼	PostgreSQL 9.5	EDB Postgres Advanced Server 9.5
Windows x86-32	○	
Windows x86-64	○	○
Linux x86-32	○	
Linux x86-64	○	○
Apple Mac	○	

2장. PostgreSQL 설치

PostgreSQL 9.5 버전으로 설치하도록 하겠습니다. EDB PAS는 라이선스 키가 있어야 설치가 가능합니다. EDB PAS는 PostgreSQL과 설치과정이 동일하므로 해당 교재에서는 EDB PAS 설치 과정을 생략하도록 하겠습니다.

2.1 설치 전 준비 사항

2.1.1 지원 플랫폼

PostgreSQL 9.5가 지원하는 운영체제입니다.

- Linux x86-64/32: *RHEL / CentOS / OEL 7.x & 6.x, SLES 12.x, Ubuntu 14.04 LTS, Debian 7.x,*

Amazon Linux

- Windows x86-64: *Windows 2012 R2 & R1, Windows 2008 R2*

- Windows x86-32: *Windows 2012 R2, Windows 2008 R2*

- Apple Mac: *OS X 10.9-10.6*

2.1.2 하드웨어 및 소프트웨어 요구 사항

○ 하드웨어

구분	요구사항
CPU	최소 600MHz 이상
RAM	최소 512MB 이상
HDD (하드웨어 용량)	최소 512MB

○ 소프트웨어

구분	요구사항
시스템	32비트, 64비트
운영체제	Linux 계열, Window 계열, Mac OS

2.2 서버 설치

PostgreSQL 설치 파일은 Enterprise DB 사 홈페이지(<http://www.enterprisedb.com/products-services-training/pgdownload>)에서 별도 회원가입 없이 다운로드가 가능합니다.

Software Downloads

[EDB Software Platform Support and Lifecycle Policies](#)

Older releases can be accessed through the [Customer Portal](#) downloads section (accessible to customers only).

Please Note: Cookies should be enabled for the download process to function correctly.

Postgres Advanced Server	PostgreSQL	EDB Replication Server 6.0	Postgres Enterprise Manager	EDB Failover Manager	EDB Backup & Recovery Tool	Components & Other Downloads
--	----------------------------	--	---	--------------------------------------	--	--

Please make sure you upgrade your components to the latest patch / point releases using StackBuilder immediately after installation is complete.

Current Release			Benefits	
PostgresSQL 9.5				
<small>Recommended: Try EnterpriseDB tools by running StackBuilder after installing PostgreSQL, expanding Trial Products and selecting Components under EnterpriseDB</small>				
Windows-32		Download	<ul style="list-style-type: none"> World's most advanced open source database 25+ years enterprise-class development by a large independent community Most secure open source database Fully supported by EnterpriseDB's own PostgreSQL experts Active world-wide deployments in public and private sector organizations of all sizes and missions 	
Windows-64		Download		
Linux x86-32		Download		
Linux x86-64		Download		
Mac		Download		

그림 2-1 Enterprise DB 사 홈페이지

2.2.1 윈도우 설치

윈도우 운영체제에 PostgreSQL 9.5 버전으로 설치하도록 하겠습니다. 설치 파일을 실행하면

PostgreSQL Setup Wizard가 실행됩니다 'Next' 버튼을 클릭하여 설치를 진행합니다.



그림 2-2 설치 초기 화면

PostgreSQL 설치 디렉터리를 설정합니다. 기본 설정 값은 C:\Program Files\PostgreSQL\9.5

입니다. 'Next' 버튼을 클릭하여 설치를 계속 진행합니다.

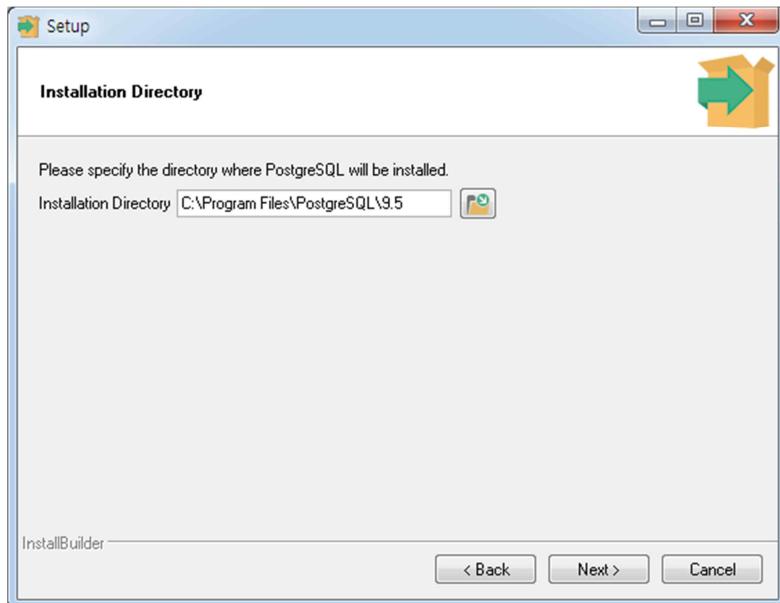


그림 2-3 설치 디렉터리 선택 화면

PostgreSQL의 사용자 데이터를 저장할 디렉터리를 설정합니다. 기본 설정 값은 PostgreSQL 설치 경로의 하위 디렉터리 data에 존재합니다. 디렉터리 값 설정 후 'Next' 버튼을 클릭하여 설치를 계속 진행합니다

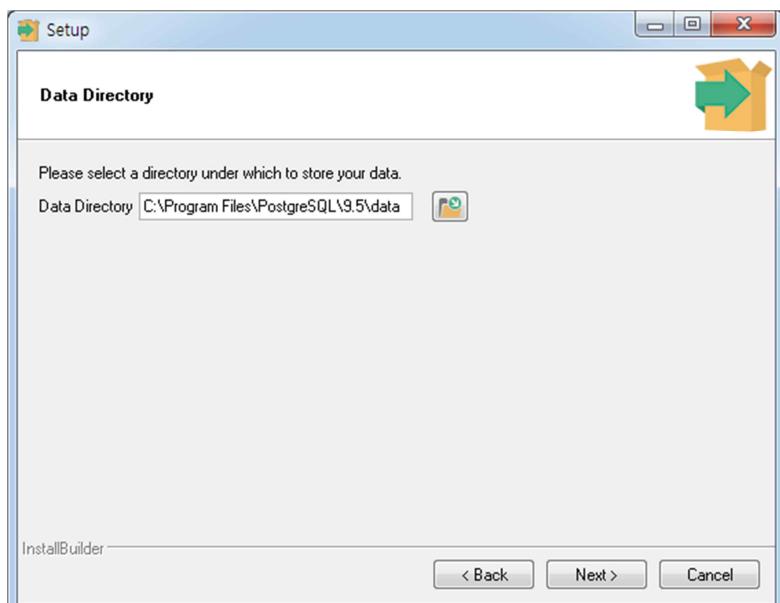


그림 2-4 설치 디렉토리 선택 화면

PostgreSQL의 슈퍼 유저 패스워드를 입력합니다. 이는 PostgreSQL에 접속하기 위한 계정 정보이므로 반드시 기억하도록 해야합니다. 입력 후 'Next' 버튼을 클릭합니다

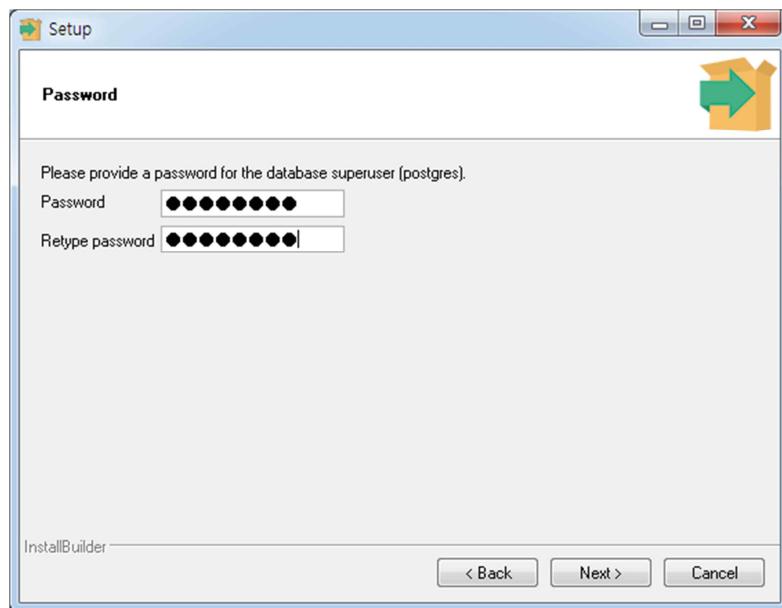


그림 2-5 패스워드 입력 화면

PostgreSQL의 서비스 포트를 설정합니다. 기본 포트 번호는 5432이며 변경 가능합니다. 포트 정보를 입력 후 'Next' 버튼을 클릭합니다.

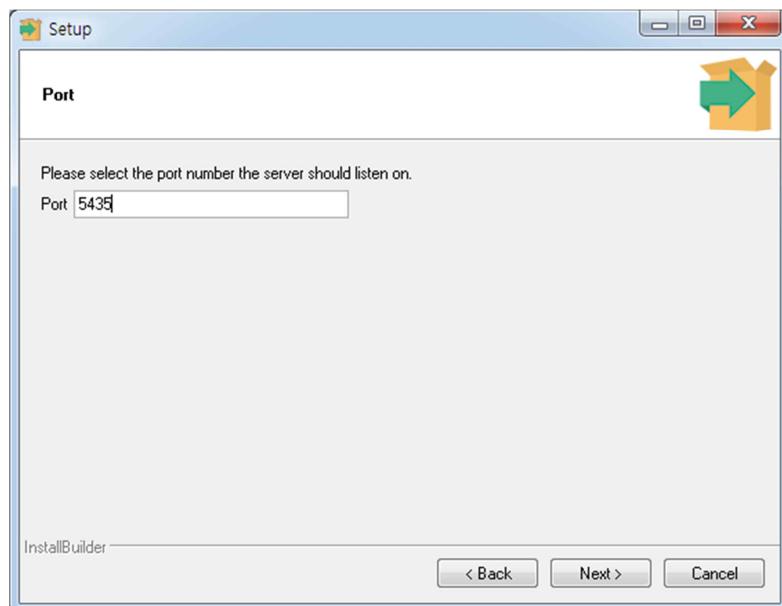


그림 2-6 포트 설정 화면

Locale을 "C"로 설정하고 'Next' 버튼을 클릭합니다.

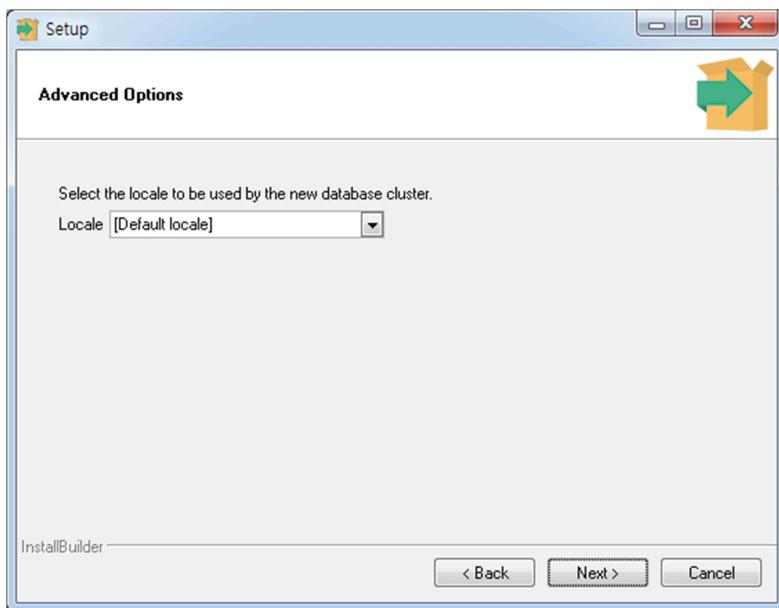


그림 2-7 Locale 설정 화면

기본 설정을 마쳤으므로 'Next' 버튼을 클릭하여 설치를 진행합니다

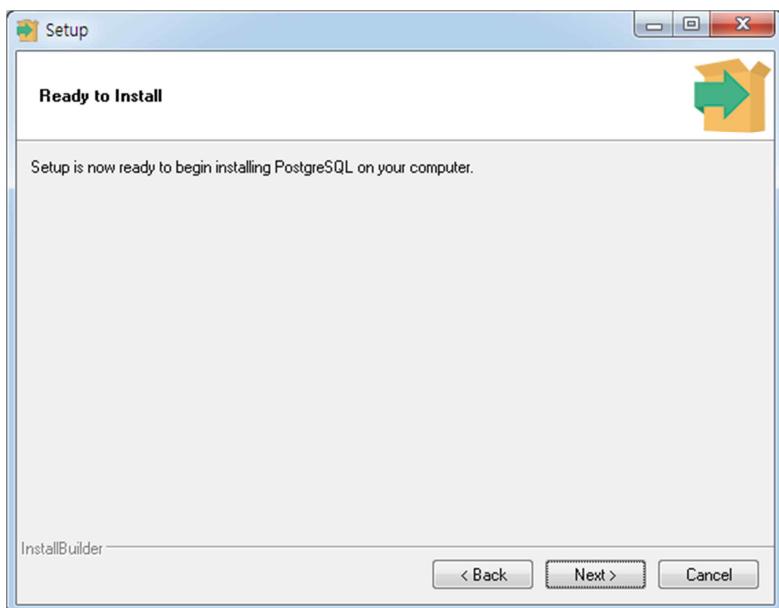


그림 2-8 설치 진행 화면

설치 과정을 progress bar로 보여주며 설치는 설치 장비의 성능에 따라 수 분이 소요될 수 있습니다

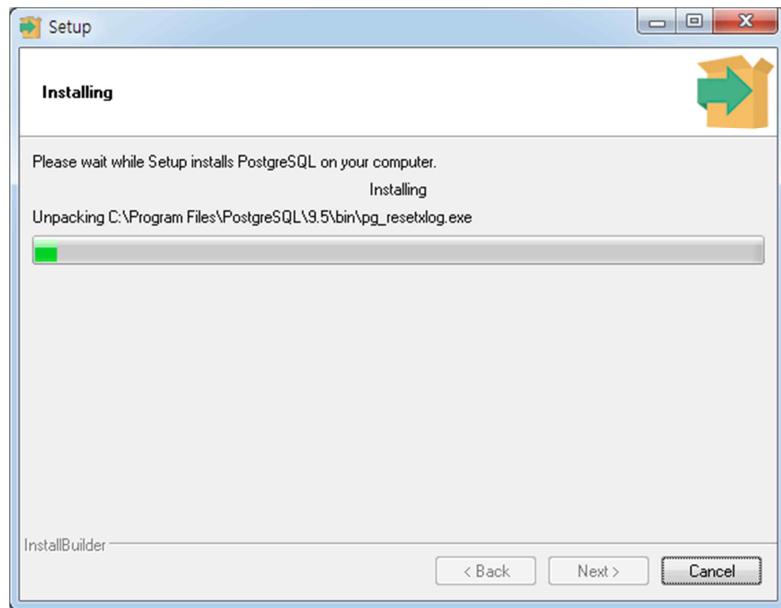


그림 2-9 설치 진행 화면

설치가 완료되었습니다. 기본적으로 PostgreSQL의 클라이언트 툴인 pgAdmin이 설치되어 있으며 필요에 따라 관련된 툴들을 추가로 설치 할 수 있습니다. 설치 완료 화면에서 “Launch Stack Builder at exit?” 항목의 체크 박스를 해제하면 추가 모듈 설치 없이 설치가 완료됩니다.

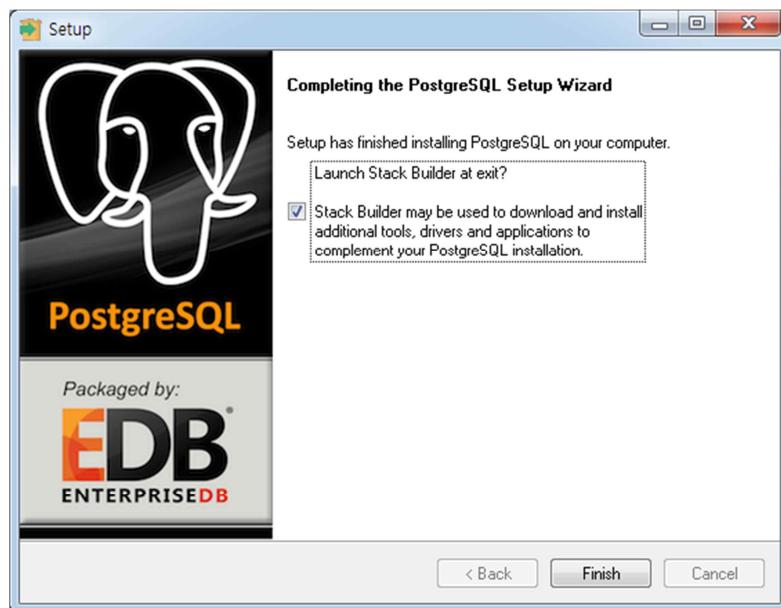


그림 2-10 추가 설치 확인 화면

2.2.2 리눅스 설치

RedHat 계열기준으로 PostgreSQL의 설치 방법을 설명 드립니다. 설치는 root 계정으로 실행해야 합니다

○ 설치

```
[root@localhost tmp]# chmod u+x postgresql-9.5.1-1-linux-x64.run
[root@localhost tmp]# ./postgresql-9.5.1-1-linux-x64.run
-----
Welcome to the PostgreSQL Setup Wizard.
-----
Please specify the directory where PostgreSQL will be installed.
Installation Directory [/opt/PostgreSQL/9.5]: Enter 엔진 디렉토리 위치 지정
-----
Please select a directory under which to store your data.
Data Directory [/opt/PostgreSQL/9.5/data]: Enter 데이터 디렉토리 위치 지정
-----
Please provide a password for the database superuser (postgres). A locked
Unix
user account (postgres) will be created if not present.
Password : 비밀번호 입력 DBA 계정 비밀번호 지정
Retype password : 비밀번호 입력
Please select the port number the server should listen on.
Port [5432]: Enter 리슨 포트 번호 설정
-----
Advanced Options
Select the locale to be used by the new database cluster.
Locale
[1] [Default locale]
[2] aa_DJ
...
Please choose an option [1] : 111 로케일 'C' 설정
-----
Setup is now ready to begin installing PostgreSQL on your computer.
Do you want to continue? [Y/n]: y
Please wait while Setup installs PostgreSQL on your computer.

Installing
0% _____ 50% _____ 100%
#####
-----
Setup has finished installing PostgreSQL on your computer
```

○ 서버 초기 설정

```
[root@localhost ~]# chkconfig --del postgresql-9.5 [os 자동 부팅 제거]
[root@localhost ~]# chown postgres.postgres /opt/PostgreSQL/9.5
[root@localhost ~]# su - postgres [사용자 계정 설정]
-bash-4.1$ cp /etc/skel/.bash* ./
-bash-4.1$ vi .bash_profile
source ./pg_env.sh
-bash-4.1$ source .bash_profile

[postgres@localhost ~]$ vi /opt/PostgreSQL/9.5/data/pg_hba.conf [접속 설정 변경]
local    all            all                                md5
host     all            all      0.0.0.0/0                md5
[postgres@localhost ~]$ pg_ctl reload [데이터베이스 설정 적용]
server signaled
```

3장. 클라이언트 설치와 사용

3.1 pgAdmin

3.1.1 설치

pgAdmin은 PostgreSQL의 Client 툴로써 PostgreSQL에 접속하여 관리하거나 SQL을 작성할 때 사용할 수 있습니다. 설치 파일은 pgAdmin.org 사이트에서 내려 받을 수 있습니다.(<http://www.pgadmin.org/download/>)

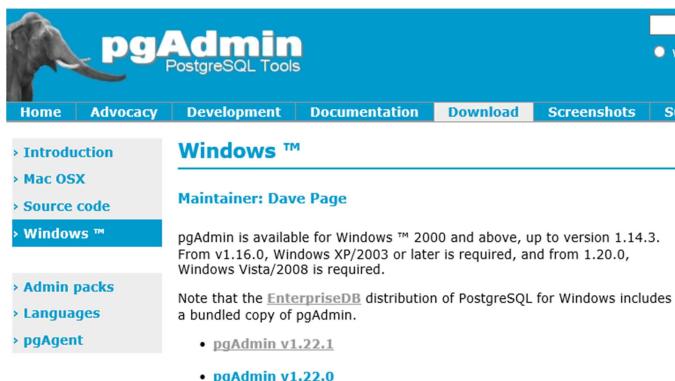


그림 3-1 pgAdmin 홈페이지 화면

사이트 왼쪽 메뉴에 자신의 플랫폼과 운영체제를 선택하여 설치파일을 내려 받으시면 됩니다. 다운로드 후 설치파일을 실행하면 pgAdmin III Setup Wizard가 실행됩니다. 'Next' 버튼을 클릭하여 설치를 진행합니다.

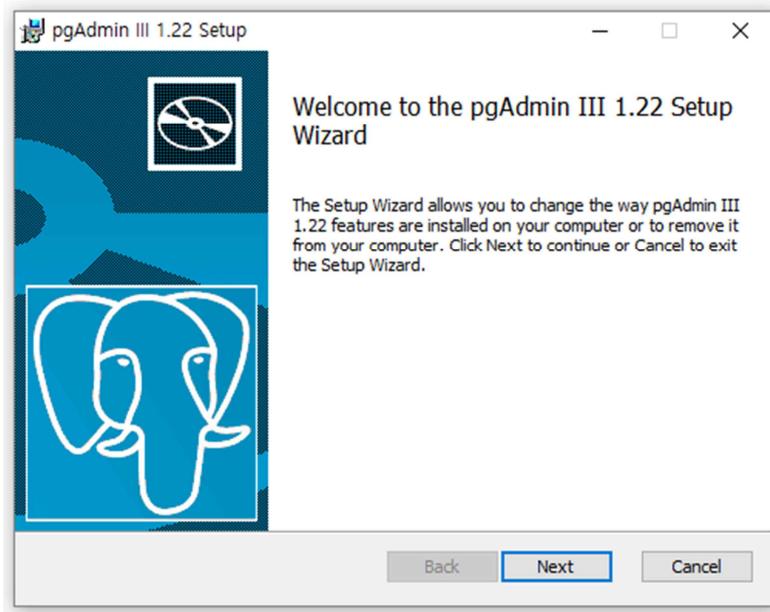


그림 3-2 초기 설치 화면

라이선스에 동의한 후 'Next' 버튼을 클릭합니다.

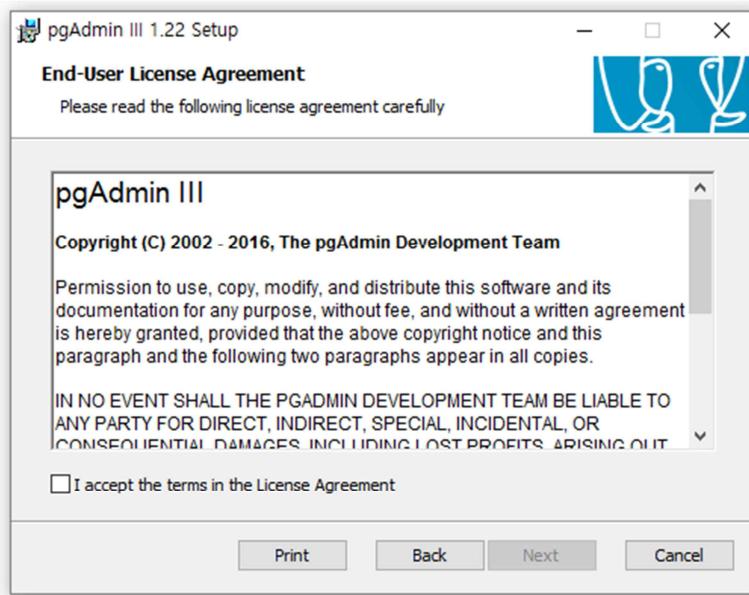


그림 3-3 라이선스 동의 선택 화면

설치 파일과 설치 경로를 설정한 후 'Next' 버튼을 클릭합니다.

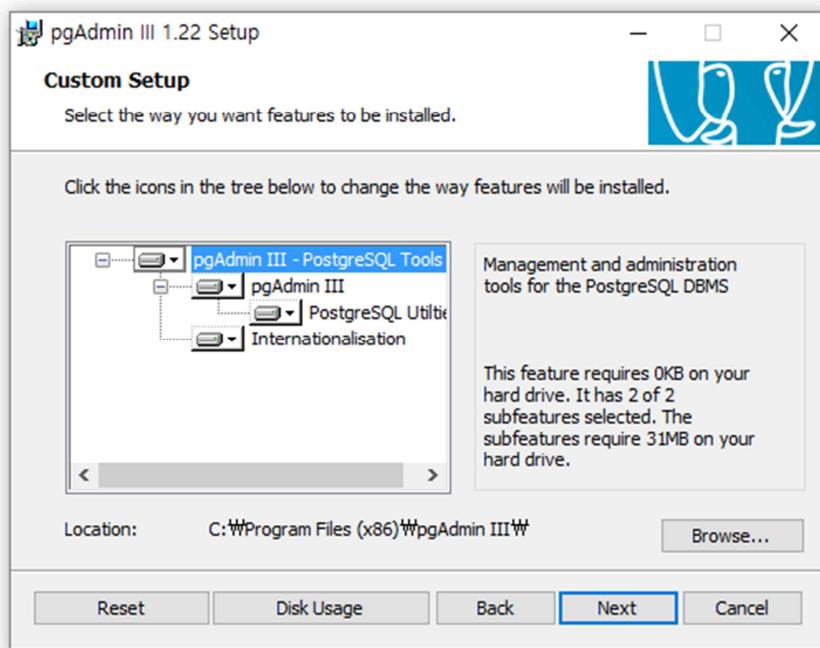


그림 3-4 설치 디렉토리 선택 화면

설치 진행에 필요한 정보를 모두 설정하였으므로 'Install' 버튼을 클릭하여 설치를 진행합니다.

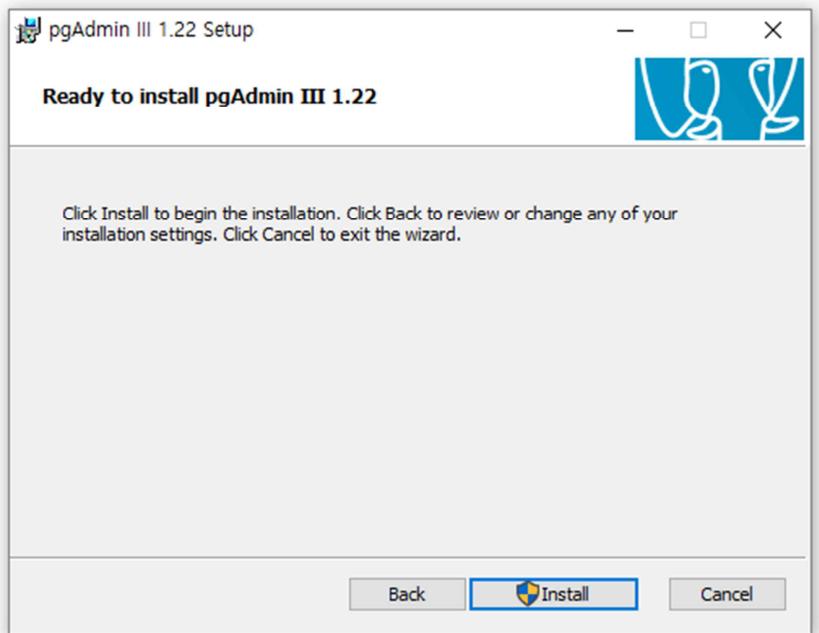


그림 3-5 설치 진행 선택 화면

설치는 약 1분 정도 소요됩니다. 설치가 완료되었으면 'Finish'를 클릭하여 설치를 종료합니다.

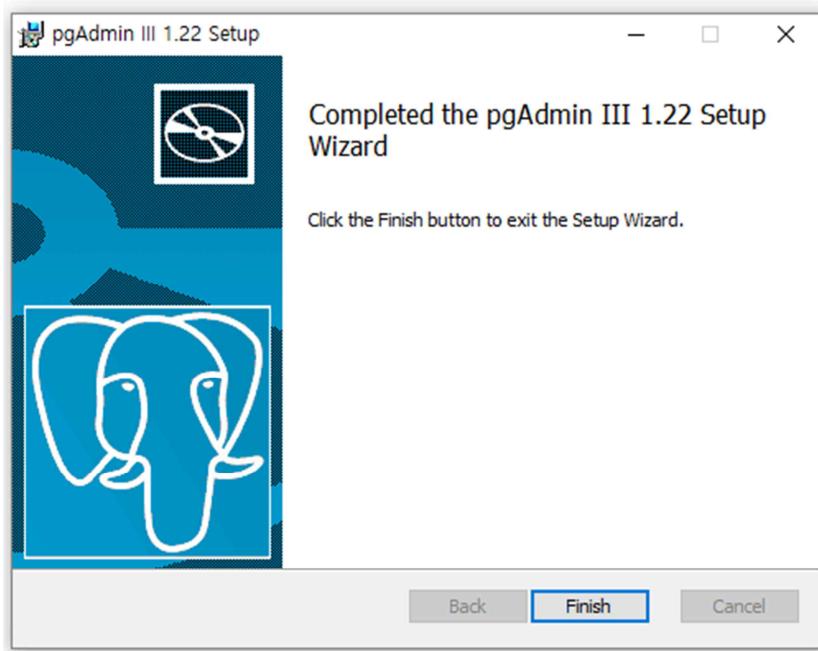


그림 3-6 설치 완료 화면

3.1.2 PostgreSQL 접속

pgAdmin을 실행하면 아래와 같은 초기 화면을 확인할 수 있습니다. PostgreSQL에 접속하기 위해

왼쪽 상단 콘센트 모양의 아이콘(Plug icon)을 클릭하거나 File→Add Server를 클릭합니다.

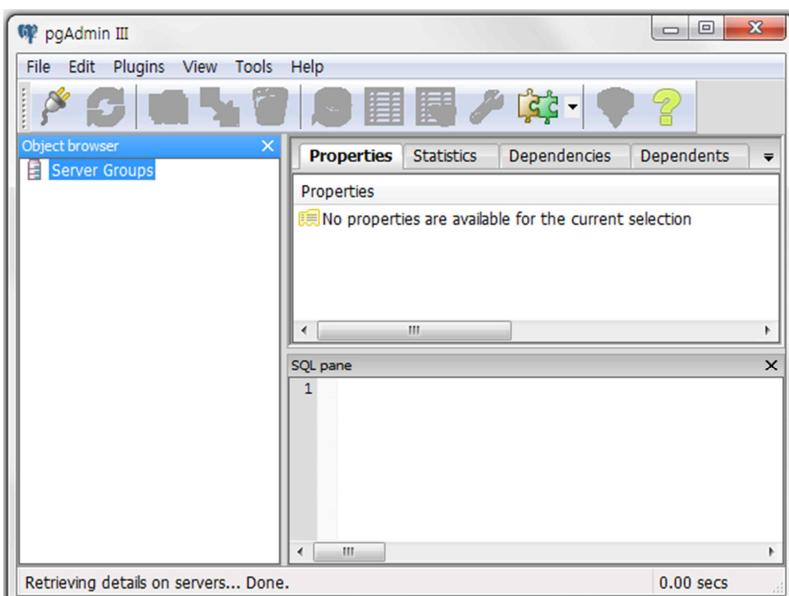


그림 3-7 pgAdmin 초기 화면

PostgreSQL의 접속 정보를 입력한 후 'OK' 버튼을 클릭합니다.

접속 정보	설명
-------	----

Name	데이터베이스 서버의 별칭
Host	데이터베이스 IP
Port	데이터베이스 Port
Service	pg_service.conf 파일에 구성된 서비스의 이름
Maintenance DB	초기 접속 데이터베이스
Username	데이터베이스 접속 계정
Password	데이터베이스 접속 계정 비밀번호
Group	사용자 지정 서버 그룹

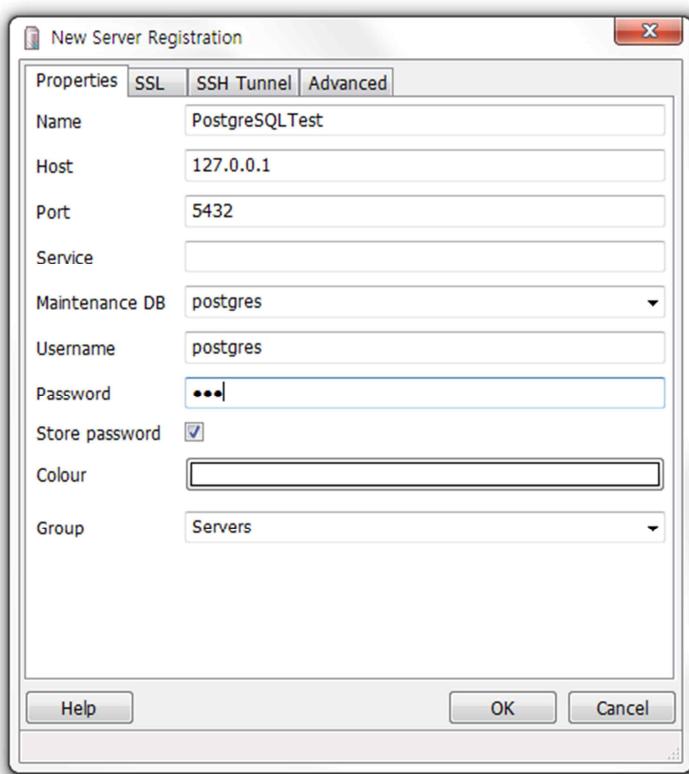


그림 3-8 접속 정보 입력 화면

pgAdmin을 통하여 PostgreSQL에 접속한 것을 확인 할 수 있습니다.

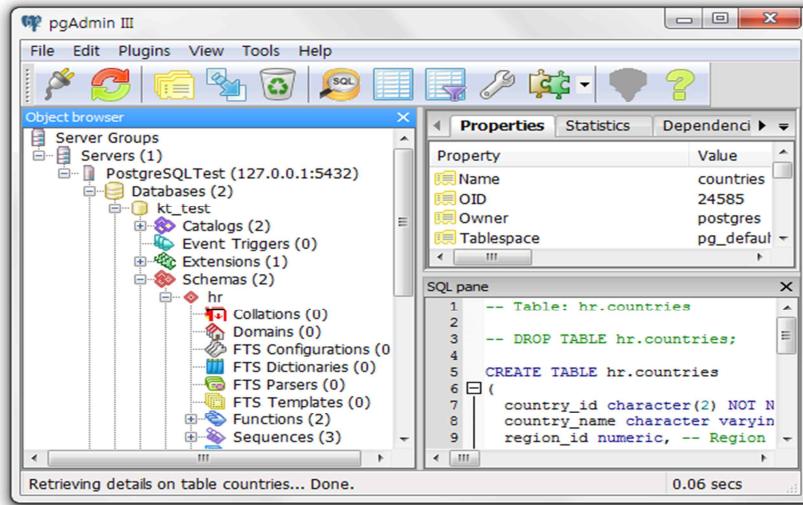


그림 3-9 접속 초기 화면

3.1.3 PostgreSQL 쿼리 실행

접속된 PostgreSQL의 데이터베이스에 SQL 문을 실행시킬 수 있습니다. 해당 데이터베이스의

스키마를 선택하고 상단의 SQL 아이콘()을 클릭합니다. SQL 문을 작성할 수 있는 팝업이 나타납니다. SQL 문을 작성하고 아이콘이나 F5를 입력하여 SQL문을 실행할 수 있으며 SQL 문 실행 결과는 하단에서 확인할 수 있습니다.

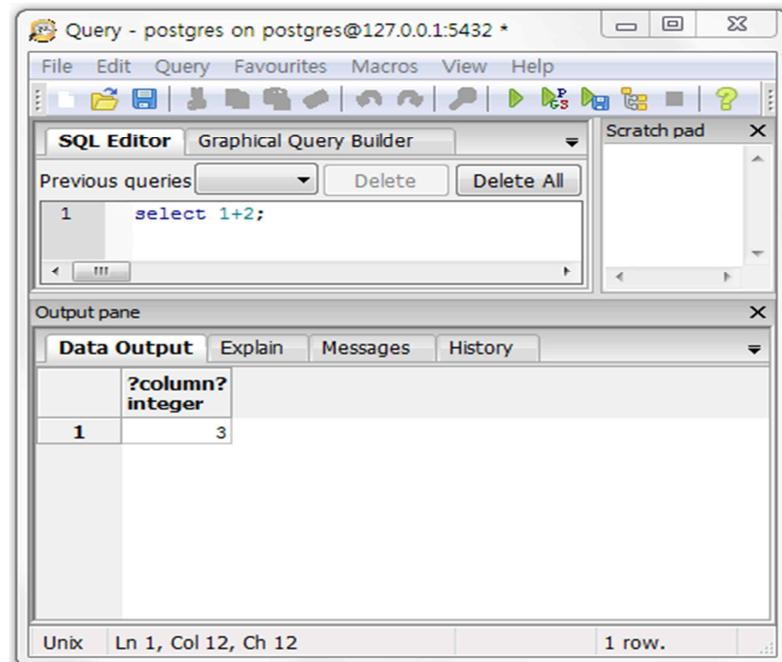


그림 3-10 쿼리 실행 화면

3.1.4 설정

PgAdmin 옵션 메뉴(File → Options)를 이용하여 pgAdmin를 사용자 편의에 맞게 설정할 수 있습니다. 여러 옵션 중 유용한 옵션에 대해 몇 가지 소개해 드리도록 하겠습니다.

항목	설명
Copy SQL from main window to query tool	쿼리 입력 창에 해당 객체의 DDL문 출력하는 옵션
Enable Auto ROLLBACK	PostgreSQL은 오토 커밋이기 때문에 모든 명령은 단일 트랜잭션으로 실행 on: 트랜잭션 안에 실행하는 쿼리 실패 시 자동으로 해당 트랜잭션을 롤백하고 다음 쿼리가 실행 off: 트랜잭션 안에 쿼리 오류가 있으면 이후 모든 쿼리 오류로 처리
Keywords in uppercase	쿼리 입력 시, 키워드는 대문자로 자동 변환

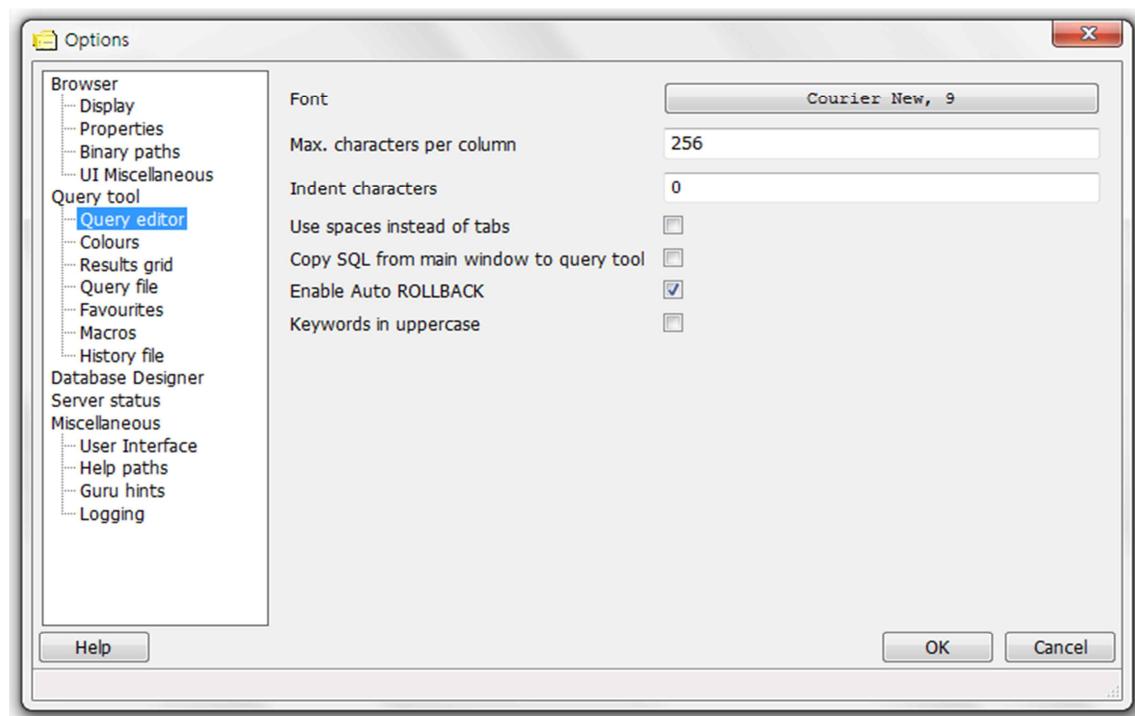


그림 3-11 pgAdmin 옵션 설정 화면

3.2 psql

3.2.1 psql 실행

PostgreSQL 설치가 정상적으로 완료되면 시작프로그램에 PostgreSQL 9.5가 등록됩니다. SQL Shell 프로그램인 psql을 사용할 수 있습니다.

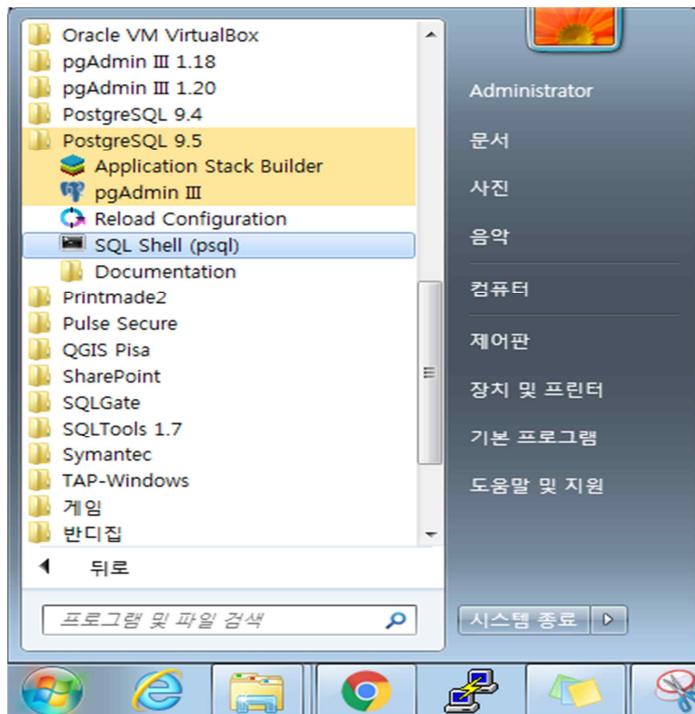


그림 3-12 psql 설치 화면

Server는 접속하고자 하는 데이터베이스 서버의 IP 정보를 입력하시면 됩니다. Database는 접속하고자 하는 데이터베이스를 지정하시면 됩니다. Port는 데이터베이스의 리슨 포트번호를 입력하시면 됩니다. Username은 데이터베이스 접속 계정을 입력하시면 됩니다. [Enter] 입력 시 []로 표기된 기본 값으로 입력 됩니다.

```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
psql <9.4.4>
Type "help" for help.

postgres=#
```

A screenshot of the psql command-line interface window. It displays the connection parameters required for a database connection: Server (localhost), Database (postgres), Port (5432), and Username (postgres). Below these parameters, the version of psql is shown as <9.4.4>. A prompt at the bottom indicates that the user can type "help" for assistance. The window has a standard Windows-style title bar and scroll bars.

그림 3-13 psql 실행 화면

psql의 자세한 사용 방법은 “유틸리티” 장에서 자세히 설명되어 있습니다.

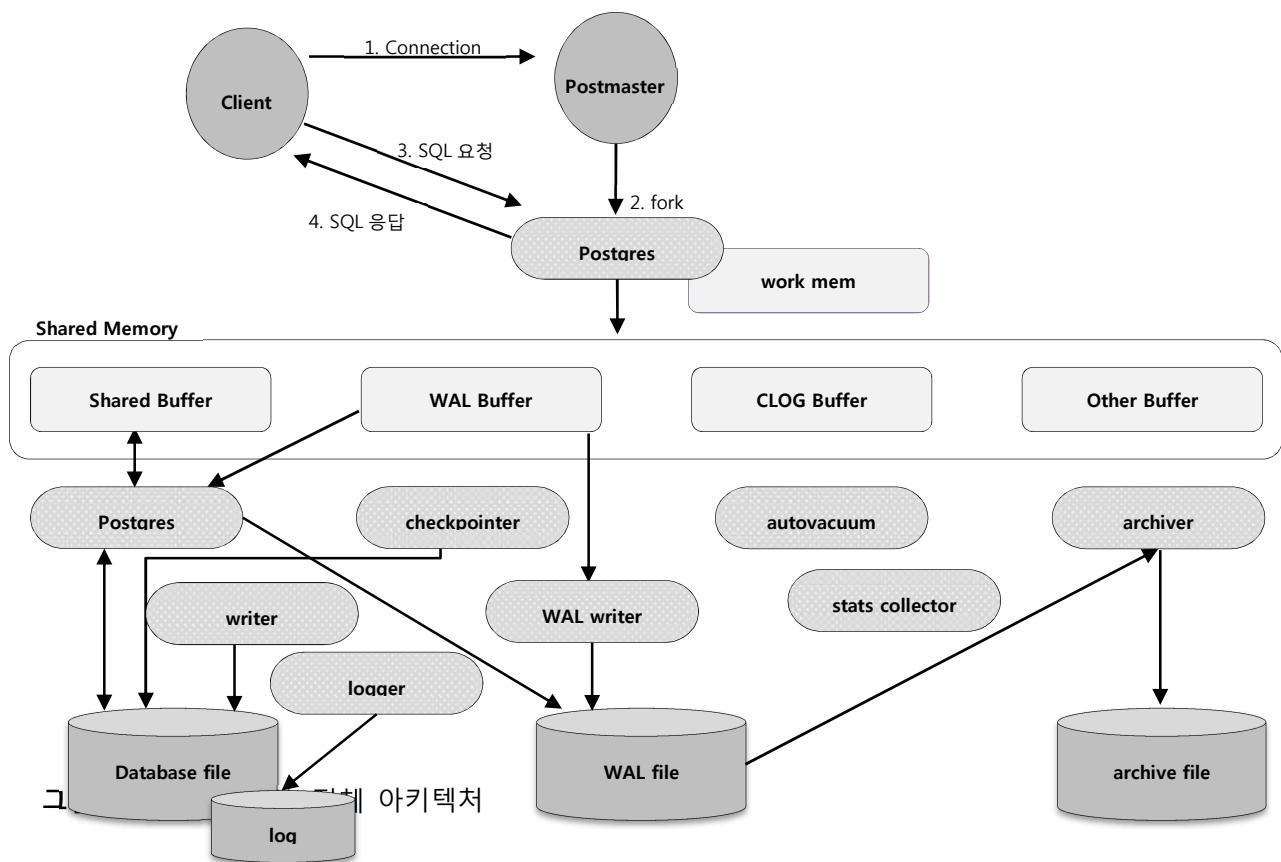
PART 2. 아키텍처

데이터베이스의 안정적인 운영과 성능 향상을 위해서는 아키텍처 이해가 선행되어야 합니다. PostgreSQL의 메모리, 프로세스, 물리적 디렉토리 구조에 대해서 알아봅니다.

1장. 아키텍처

1.1 전체 도식

PostgreSQL의 메모리, 프로세스, 물리적 디렉토리 구조로 나누어 아키텍처에 대해 설명하도록 하겠습니다.



1.2 메모리

PostgreSQL 메모리는 모든 사용자가 공유하여 사용하는 Shared Memory와 사용자마다 개별적으로 사용하는 Backend Memory로 나눌 수 있습니다.

1.2.1 Shared Memory

Shared Memory는 모든 사용자가 공유하여 사용하는 메모리 영역입니다.

메모리 항목	설명
Shared buffers	<ul style="list-style-type: none"> - 데이터를 Select, DML 작업을 하기 위해 디스크에 존재하는 데이터 블록을 읽어 저장하는 메모리 공간 - LRU 알고리즘으로 관리 - Default 값은 128MB이며(9.5기준), 데이터베이스 서버의 RAM의 25%를 할당 - 확인방법: show shared buffer
WAL (Write-Ahead Log) buffers	<ul style="list-style-type: none"> - 오브젝트 및 데이터 변경 시 변경 내역을 저장하는 메모리 공간 - 선 로그 기법: DML 작업을 수행할 경우 실제 데이터에 대한 DML을 수행하기 전에 데이터 변경에 대한 내용을 WAL에 저장 - 복구 목적으로 사용 - shared_buffer의 3% 할당
CLOG buffers	<ul style="list-style-type: none"> - 트랜잭션 커밋 정보를 저장하는 메모리 공간 <p>CLOG는 트랜잭션 커밋 정보를 MVCC에서 불필요한 DISK READ를 방지하고자 하는 기능</p>
Other buffers	<ul style="list-style-type: none"> - Lock 관련 buffers - 통계 정보 buffers - Two Phase commit 관련 buffers

1.2.2 Backend Memory

Backend Memory는 사용자마다 공유하지 않고 개별적으로 사용하는 메모리 영역입니다. 아래 표에 설명한 메모리 외에도 세션 정보와 커서 상태 정보, 변수 저장 공간 등이 메모리로 존재합니다.

메모리 항목	설명
work_mem	<ul style="list-style-type: none"> - 정렬 작업을 하기 위한(Order by, Group by) 메모리 영역 - 대량의 sort 작업이 필요하면 세션단위로 늘려도 되지만 OS 메모리를 감안해야 함 - work_mem가 부족할 경우 디스크 영역 사용 - Default 값은 4MB이며 (OS 메모리 – Shared buffer) / connection 로 계산하여 할당
maintenance_work_mem	<ul style="list-style-type: none"> - 데이터베이스를 관리하기 위한(Vacuum, INDEX 생성) 필요한 메모리 - Default 값은 16MB이며 vacuum에서 사용하는 메모리 최대 테이블 크기와 비슷하게 설정
temp_buffers	<ul style="list-style-type: none"> - 세션 단위로 임시 테이블에 접근하기 위한 메모리 - Default 값은 8MB

1.3 프로세스

PostgreSQL의 프로세스는 서버 프로세스인 Postmaster, 백그라운드 프로세스인 Utility Process, 세션 프로세스인 User Backend Process로 나눌 수 있습니다.

[예시] PostgreSQL 프로세스

```
[postgres@localhost ~]$ ps -U postgres -H -opid,ppid,cmd
 PID  PPID CMD
 9225    1 /opt/PostgreSQL/9.5/bin/postgres -D /opt/PostgreSQL/9.5/data
 9257  9225  postgres: logger process
 9259  9225  postgres: checkpointer process
 9260  9225  postgres: writer process
 9261  9225  postgres: wal writer process
 9262  9225  postgres: autovacuum launcher process
 9263  9225  postgres: archiver process   last was 000000010000016300000003D
 9264  9225  postgres: stats collector process
10323  9225  postgres: postgres postgres 127.0.0.1(52249) UPDATE waiting
10328  9225  postgres: postgres postgres 172.27.108.53(40728) UPDATE waiting
10333  9225  postgres: postgres postgres 172.27.108.53(40742) UPDATE
10337  9225  postgres: postgres postgres 172.27.108.53(40754) idle in
transaction
10352  9225  postgres: postgres postgres 127.0.0.1(52258) idle in transaction
18078  9225  postgres: postgres postgres 127.0.0.1(53004) idle
```

- 9225 프로세스가 Postmaster
- 9257부터 9264까지는 백그라운드 하위 프로세스들
- 10328, 10333, 10337 프로세스는 리모트 세션 프로세스
- 10328 프로세스는 172.27.108.53에서 리모트로 TCP 통신 연결해서 현재 UPDATE waiting 중
- 10333 프로세스는 172.27.108.53에서 리모트로 TCP 통신 연결해서 현재 UPDATE 작업 중
- 10337 프로세스는 172.27.108.53에서 리모트로 TCP 통신 연결해서 현재 idle in transaction 중
- 10323, 18078 프로세스는 세션 프로세스로 로컬호스트에서 TCP 통신으로 연결한 세션 프로세스

1.3.1 Postmaster

Postmaster 프로세스는 오라클의 listener + pmon + smon 역할을 수행합니다. 즉, 서버 LISTEN, 유ти리티 프로세스와 세션 프로세스를 실행하고 관리합니다.

수행 업무	설명
서버 LISTEN	- 관리자가 서버를 실행하면, 서버 실행 스크립트는 먼저 bin/postgres 파일을 실행해서 서버 최상위 프로세스인 postmaster 프로세스 생성

	<ul style="list-style-type: none"> 환경 설정을 파일을 읽어서 서버에 필요한 공유 메모리를 획득 세션에서 사용할 세마포어 공간을 확보 다음 필요한 데이터 파일들 (클라이언트 접속 제한 정보, 데이터베이스를 사용하기 위한 전역 컨트롤 파일들과, 트랜잭션 정보, 데이터베이스 정보 등등)을 읽어서 공유 메모리 영역에 저장 <ul style="list-style-type: none"> 유닉스 소켓 파일 생성 설정에 따라 TCP 소켓 하나를 특정 포트(일반적으로 5443)로 열어 Listen 상태 Ready - 비정상적 실행 일 때는 서버 복구 모드로 실행
유ти리티 프로세스 실행/관리	<ul style="list-style-type: none"> - 데이터베이스 운영에 필요한 여러 하위 프로세스들을 실행하고, 이 프로세스들이 비정상적으로 종료되면 다시 실행
세션 프로세스 실행	<ul style="list-style-type: none"> - 서버 운영에 필요한 모든 작업이 준비가 되면, postmaster는 주기적으로 해당 소켓 파일을 Select 하다가 소켓 연결 요청(클라이언트의 접속 시도)이 오면 fork 방식으로 postgres라는 프로세스 이름으로 세션 프로세스 생성 - 접근이 허용되는 호스트 여부를 판단하고, 허용되면 사용자 인증 작업 수행 - 그 세션 프로세스를 통해서 응용 프로그램은 작업 요청을 하고, 서버는 응답 결과를 그 세션 프로세스를 통해 응용 프로그램으로 전송

1.3.2 Utility Process / User backend Process

하위 프로세스들은 서버의 환경 설정에 따라 어떤 것은 아예 실행이 안 되는 경우도 있고, 어떤 경우는 필요할 때만 잠시 실행되고 중지 되는 경우도 있습니다.

프로세스	설명
Writer	<ul style="list-style-type: none"> - writer(8.3 이하 버전에서는 background writer로 불렸음) 프로세스는 Shared buffers에 있는 자료를 디스크에 기록
WAL Writer	<ul style="list-style-type: none"> - 메모리에 있는 데이터 블록의 변경 내역은 트랜잭션이 일어날 때마다 트랜잭션 로그(WAL log)에 먼저 기록하고, 체크 포인트가 발생하거나, 트랜잭션 로그 파일을 재활용해야 할 상황이 오면, writer가 물리적인 파일에 각 블록을 기록하는 write ahead log writing 기법에 사용 - WAL writer 프로세스는 이 트랜잭션 정보를 로그 파일에 바로 쓰는 것이 아니라, WAL 버퍼에 기록하고, 물리적 파일에 기록
Archiver	<ul style="list-style-type: none"> - 온라인 백업을 위한 트랜잭션 로그의 아카이빙 기능을 담당 - 트랜잭션 로그 스위칭이 일어나면 환경 설정에서 지정한 아카이브 명령(archive_command)을 실행하는 프로세스. 아카이빙 관련 장애는 보통 이 작업에 지정한 OS 명령어(archive_command 환경 설정 값)의 오동작에 대한 예외 처리를 제대로 하지 않아서 발생

Logger	<ul style="list-style-type: none"> - 로그 수집기 프로세스는 데이터베이스 프로세스들이 만들어 내는 로그를 파일에 기록 - 세션 프로세스가 실행 중일 때는 OS 관리자가 강제로 종료 하지 않는 한 절대로 중지 되지 않는 프로세스
Stats Collector	<ul style="list-style-type: none"> - 통계 수집기 프로세스는 쿼리 최적화에서 사용할 데이터베이스 통계 정보를 수집하는 프로세스 - autovacuum 실행기와 같이 주기적으로 체크를 하다가 통계 수집 작업이 필요하다고 판단되면 그 작업을 진행
Autovacuum	<ul style="list-style-type: none"> - autovacuum launcher가 상시 백그라운드로 실행되고 있으면서 각 객체들의 vacuum 작업이 필요하다고 판단되면, autovacuum 작업을 하는 세션을 실행시켜 작업을 하는 방식 - autovacuum 작업 세션의 최대 개수는 서버 환경 설정 파일에서 지정
WAL receiver & sender	<ul style="list-style-type: none"> - 트랜잭션 로그 스트리밍 기반의 리플리케이션 이용하는 서버에서 사용하는 프로세스. 마스터 노드에는 Sender 프로세스가 슬레이브 쪽에는 receiver 프로세스가 실행

1.3.3 Session

세션 프로세스는 User Process로부터 받은 요청을 수행하며 이를 위해 해당 데이터 블록을 데이터베이스 파일로부터 shared buffer에 올리는 작업을 합니다. 이 때 shared buffer에 free buffer가 없다면 LRU 알고리즘에 의해서 사용 빈도가 제일 낮은 checkpoint된 블록을 버리고 사용합니다. checkpoint된 블록이 없다면 세션 프로세스는 직접 트랜잭션 정보를 WAL Buffer에 기록하고 사용 빈도가 낮은 buffer를 데이터베이스 파일에 기록합니다. 만약 이 때 WAL Buffer 또한 공간이 없다면 WAL Buffer를 WAL files로 쓰는 작업을 통해 여유 WAL Buffer를 마련한 후 일련의 작업을 수행합니다.

PostgreSQL는 다른 데이터베이스와 다르게 이 모든 작업을 세션 프로세스가 직접 하기도 합니다.

1.3.4 프로세스 관리

서버 프로세스와 유ти리티 프로세스가 모두 실행되고 있는지 OS 명령어로 확인 가능합니다. 해당 포트가 정상적으로 리슨 상태인지는 OS 명령어나 클라이언트 툴로 접속하여 확인 가능합니다.

OS 작업자는 특별한 이유 없이 kill 명령으로 세션 프로세스를 종료하면 안 되는데, 그 이유는 PostgreSQL의 경우 세션 프로세스가 외부 시그널에 의해 종료되면, postmaster가 그것을 감지하고 서버를 초기화하는 특징을 갖고 있기 때문입니다. 따라서 세션 프로세스를 정리할 때는 반드시 데이터베이스 서버로 접속해서 SQL 명령으로 세션을 정리하는 것이 가장 안전합니다.

1.4 물리적인 디렉터리 구조

PostgreSQL의 서버 소프트웨어가 설치되는 엔진 디렉터리와 운영을 하면서 관리되는 데이터 디렉터리로 나누어집니다.

1.4.1 엔진 Directory

PostgreSQL를 설치하면 데이터베이스 Major 버전의 이름으로 하위 디렉터리가 생성이 됩니다. 이 디렉터리 기준으로 패치와 업그레이드가 진행됩니다. 패치는 같은 디렉터리 내에서 이뤄집니다. 업그레이드인 경우는 마이그레이션 작업이 필요하기 때문에, 새로운 디렉터리를 생성하여 진행하게 됩니다.

Directory	설명
bin	<ul style="list-style-type: none"> - 각종 실행파일 포함 <ul style="list-style-type: none"> • 서버 응용 프로그램: pg_ctl 등 • 클라이언트 응용 프로그램: psql, pg_dump 등 • 기타 Postgres 운영에 필요한 패키지 실행파일
doc	<ul style="list-style-type: none"> - PostgreSQL에서 사용하는 각종 서브 프로그램들과 확장 모듈들에 대한 설명서
include	<ul style="list-style-type: none"> - C 언어 소스 컴파일에 필요한 헤더 파일들
installer	<ul style="list-style-type: none"> - scripts Directory와 함께 설치 때 필요했던 각종 부가 설치 스크립트
lib	<ul style="list-style-type: none"> - 데이터베이스에서 사용하는 동적 라이브러리들 - 자바를 제외한 언어 개발에 필요한 정적 라이브러리들
scripts	<ul style="list-style-type: none"> - 설치 시 필요한 스크립트
share	<ul style="list-style-type: none"> - 각종 확장 모듈들을 사용하기 위한 모듈 초기화 SQL 스크립트 - extension 객체로 관리되지 않는 부가 확장 모듈, 부가 응용 프로그램 스크립트
StackBuilder	<ul style="list-style-type: none"> - 데이터베이스 엔진과 각종 부가 패키지들의 설치, 업그레이드를 관리
symbols	<ul style="list-style-type: none"> - Postgres 모듈의 디버깅 정보

1.4.2 데이터 Directory

PostgreSQL의 데이터 저장하는 최상위 개념을 데이터 클러스터라고 부릅니다. 흔히 PGDATA Directory라고 하며, 이 클러스터 안에는 데이터베이스 서버에서 사용하는 모든 데이터가 보관됩니다. 이 디렉터리는 보안상 PostgreSQL 서비스 관리자만 접근 할 수 있도록 설정되어야 합니다. 데이터베이스를 백업 한다는 말은 궁극적으로 이 데이터 클러스터 디렉터리 전체를 백업 시점으로 사진을 찍듯이 딱 찍어 다른 곳에 보관한다는 것을 뜻입니다. 데이터 디렉터리는 서버 프로세스가 접근 할 수 있는 어떤 위치에 있어도 상관은 없으나(UNIX NFS나, Windows 네트워크 드라이브는 제외) 관리 목적상 데이터베이스 엔진 디렉터리 안에 두지 않습니다. 또한 백업

도구인 pg_basebackup 같은 도구는 이 디렉터리 기준으로 작업하기 때문에, 이 디렉터리 안에는 데이터베이스와 관계되지 않는 파일들을 두지 않는 것이 운영상 좋습니다.

데이터 디렉터리의 위치를 알아보는 방법은 아래와 같습니다.

- SQL 쿼리문: show data_directory;
- OS 프로세스 목록에서 postgres 프로세스의 실행 옵션 가운데 -D 옵션 위치 확인
- OS에서 PG_VERSION 파일이나 설정 파일(postgresql.conf 또는 pg_hba.conf) 위치 확인

위에서 말한 postgresql.conf, pg_hba.conf와 같이 환경 설정을 위한 일반 텍스트 파일 외엔 어떠한 파일도 운영상 부득이한 경우를 제외하고는 임의로 조작하지 말아야 합니다.

데이터 디렉터리는 크게 자료 영역, 트랜잭션 로그 영역, 환경 설정 영역, 임시 트랜잭션 영역, 그 외 기타 영역으로 나누어 살펴볼 수 있습니다.

영역	Directory	설명
자료	\$PGDATA/base/	<ul style="list-style-type: none"> - 데이터 클러스터의 핵심 디렉터리 - 저장 규칙: \$PGDATA/base/{database_oid}/{object_id} - 테이블 별로 1:N의 데이터 파일이 생성 - 테이블 생성시 테이블 OID 이름의 데이터 파일이 생성 - object_filenode.N 형태는 테이블의 크기가 1GB 이상 생성 - filenode 접미사 의미 <ul style="list-style-type: none"> • _fsm : free space map, 블록의 빈 공간 정보 보관 • _vm : visibility map, 실제 사용하는 자료의 정보 보관 - 객체 물리 파일 확인: pg_relation_filepath()
	\$PGDATA/global/	<ul style="list-style-type: none"> - 데이터베이스 전역 정보(모든 데이터베이스에서 공통으로 참조) <ul style="list-style-type: none"> • pg_control: 데이터 클러스터의 컨트롤 정보(데이터베이스 버전, 트랜잭션, 체크포인트, 백업 데이터 블록 크기, 트랜잭션 로그 크기, 부동수형 처리 방식 등). pg_controldata 명령으로 확인 가능 <ul style="list-style-type: none"> • pg_internal.init, pg_filenode.map: 객체들의 속성 정보와 실제 파일의 맵핑 정보가 저장 • 그 외 숫자 파일: 전역 객체의(테이블 스페이스, 데이터베이스, 데이터베이스 사용자 그룹, Role 그룹, 사용자) 데이터와 인덱스 파일
	\$PGDATA/pg_tblspc/	<ul style="list-style-type: none"> - 물리적 테이블스페이스에 대한 심볼링 링크 파일 공간 - 기본 테이블 스페이스인 pg_default, pg_global을 제외한 사용자 정의 테이블 스페이스만 보관 - 테이블 스페이스 단위 백업/복구와 on/offline 상태 값 변경 기능 미지원

		- 테이블 스페이스 임계치 관리, 자동 확장 등의 기능은 OS의 파티션 관리 기능으로 맡김
트랜잭션	\$PGDATA/pg_xlog/	<ul style="list-style-type: none"> - 트랜잭션 로그 파일 보관(데이터베이스 복구용) - 트랜잭션 로그는 새로운 이름으로 변경하여 재활용 - 트랜잭션 로그 최대 개수 설정: max_wal_size 설정 값의 약 세 배(데이터베이스 9.4 이하는 checkpoint_segments)
	\$PGDATA/pg_clog/	<ul style="list-style-type: none"> - 2비트 단위로 트랜잭션 정상 Commit 정보 - 최대 트랜잭션 정보는 autovacuum_freeze_max_age 설정
환경 설정	\$PGDATA/PG_VERSION	<ul style="list-style-type: none"> - 데이터 클러스터의 데이터베이스 메이저 버전
	\$PGDATA/postgresql.conf \$PGDATA/postgresql.auto.conf	<ul style="list-style-type: none"> - 서버 환경 설정 파일
	\$PGDATA/pg_hba.conf	<ul style="list-style-type: none"> - 클라이언트의 접속을 허용과 제한을 위한 호스트 기반 접근 권한 제어 파일 - 설정 변경 적용을 위해 reload 작업 필요
임시 트랜잭션 영역	\$PGDATA/pg_multixact	<ul style="list-style-type: none"> - shared row lock 처리를 위한 트랜잭션 상태 값 보관
	\$PGDATA/pg_serial	<ul style="list-style-type: none"> - serializable 트랜잭션 상태 값 보관
	\$PGDATA/pg_subtrans	<ul style="list-style-type: none"> - 서브트랜잭션 상태 값 보관
	\$PGDATA/pg_twophase	<ul style="list-style-type: none"> - prepared 트랜잭션 상태 값 보관
	\$PGDATA/pg_snapshots	<ul style="list-style-type: none"> - 추출된 스냅샷 트랜잭션
기타	\$PGDATA/pg_notify	<ul style="list-style-type: none"> - 세션 간 이벤트 발생, 감지 처리를 위한 LISTEN, NOTIFY SQL 명령어 중 NOTIFY로 정의된 이벤트 저장
	\$PGDATA/pg_stat_tmp	<ul style="list-style-type: none"> - 통계 수집기가 사용하는 임시 파일 저장

PART 3. 서버 관리

PostgreSQL를 운영 및 관리하는 방법에 대해 알아봅니다.

1장. 서버 실행과 종료

PostgreSQL 데이터베이스를 시작과 종료를 하기 위해서는 pg_ctl 명령어를 사용합니다. PostgreSQL과 EDB PAS 서비스 관리자인 데이터베이스 운영자 계정으로 실행해야 합니다.

1.1 서버 시작

[서버 시작 방법]

- 시작: pg_ctl -l log -D \$PGDATA start
- 확인: pg_ctl -D \$PGDATA status

-l: 로그 -D: 클러스터 경로

1.2 서버 종료

[서버 종료 방법]

- 중지: pg_ctl -D \$PGDATA -mf stop
- 확인: pg_ctl -D \$PGDATA status

데이터베이스 종료 시에는 -m 뒤에 옵션을 사용할 수 있습니다. 데이터베이스 운영 중에는 fast 옵션을 사용합니다.

- smart: 모든 클라이언트 실행중인 작업 정상 중지(commit) 끝내고 종료
- fast: 모든 클라이언트 실행중인 작업 강제 중지(rollback) 하고 종료(권장)
- immediate: 바로 종료. 재 실행 시 복구 모드로 실행 됨

smart 옵션은 오라클의 shutdown immediate 명령어와 동일합니다. immediate 옵션은 오라클의 abort 옵션과 동일하므로 주의가 필요합니다.

[서버 재시작 방법]

- 재시작: pg_ctl \$PGDATA -mf restart
- 확인: pg_ctl -D \$PGDATA status

1.3 시그널 전송

PostgreSQL의 일부 서버 환경 변수 값을 적용하기 위해 데이터베이스 서버에 시그널을 보낼 수 있습니다.

[서버 시그널 전송 방법]

- 시그널 명령어: pg_ctl -D \$PGDATA reload
- 로그 확인: received SIGHUP, reloading configuration files

pg_ctl 사용법의 자세한 설명은 pg_ctl --help 명령으로 확인 가능합니다. pgplus_env.sh 파일에 환경 변수 값을 설정하여 명령어와 클러스터 경로를 생략하여 편리하게 사용 가능합니다.

2장. 서버 설정

2.1 서버 환경 레벨

PostgreSQL의 서버 환경 설정은 클러스터, 데이터베이스, 사용자, 세션 레벨에 따라 설정 가능합니다. 환경 변수의 특징에 따라 클러스터, 데이터베이스, 사용자, 세션 레벨로 설정 가능한 환경 변수가 있고 설정이 불가능한 변수가 있습니다.

같은 환경 변수가 각 레벨에 걸쳐 설정되어 있다면 Session > User > Database > Cluster 순으로 설정이 적용됩니다.

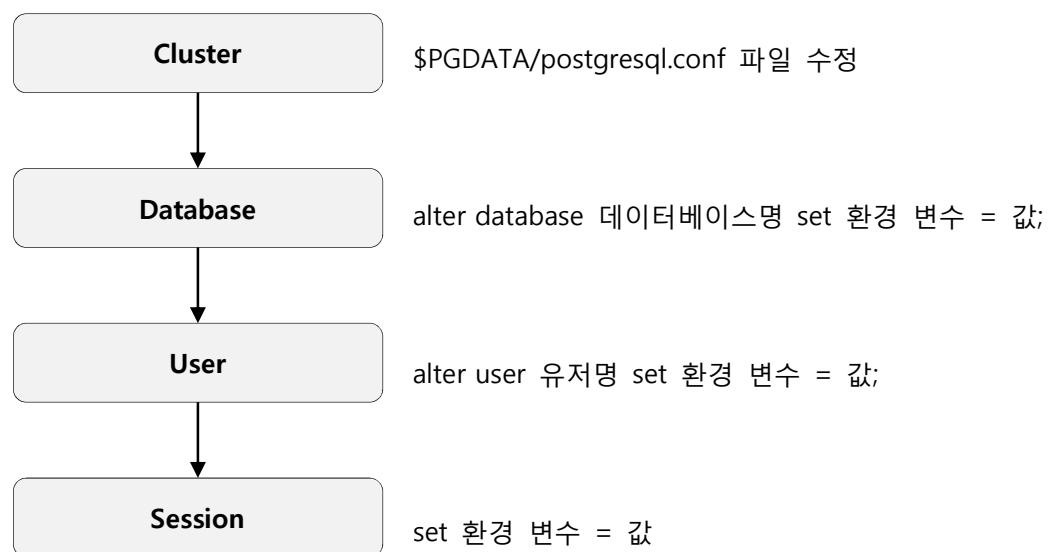


그림 2-1 서버 환경 설정 레벨

2.2 환경 변수 설정/확인

2.2.1 환경 변수 설정

환경 변수의 기본적인 설정 형식은 “환경 변수 명 = 값” 형태입니다. 값은 boolean, 숫자, 문자열, 열거형 형식으로 설정이 가능합니다.

○ Cluster

Cluster 레벨의 환경 설정을 적용하기 위해서는 \$PGDATA/postgresql.conf 파일을 수정하거나 ALTER SYSTEM 명령어를 사용하면 됩니다.

[Cluster 환경 변수 변경 방법]

```
ALTER SYSTEM SET 환경변수 = 값;
```

ALTER SYSTEM 명령어를 이용하여 환경 변수를 설정한 경우에는 바로 적용되는 것이 아니라 postgresql.auto.conf에 기록을 합니다. 그리고 데이터베이스 재기동과 Reload 작업 시 postgresql.conf 파일을 읽은 다음 postgresql.auto.conf을 읽고 적용됩니다.

환경 변수의 특징에 따라 적용을 위해서는 서버를 재실행하거나 Reload 작업을 하면 됩니다.

○ Database

Database 레벨의 환경 설정을 적용할 수 있습니다. Cluster의 환경 변수보다 우선 적용 받습니다.

[Database 환경 변수 변경 방법]

```
ALTER DATABASE 데이터베이스명 SET 환경변수 = 값;
```

○ User

사용자 레벨의 환경 설정을 적용할 수 있습니다. Cluster, Database의 환경 변수보다 우선 적용 받습니다.

[User 환경 변수 변경 방법]

```
ALTER USER 데이터베이스명 SET 환경변수 = 값;
```

○ Session

세션 레벨의 환경 설정을 적용할 수 있습니다. 세션을 벗어나면 환경 설정은 초기화됩니다.

Cluster, Database, User의 환경 변수보다 우선 적용 받습니다.

[Session 환경 변수 변경 방법]

```
SET 환경변수 = 값;
```

2.2.2 환경 변수 확인

SHOW 명령어와 시스템 카탈로그 뷰로 환경 변수 값을 확인할 수 있습니다.

확인 방법	확인 가능 대상	비고
SHOW 환경 변수 명	Cluster, Database, User, Session	
pg_settings	Cluster	setting: 현재 설정 값 short_desc: 환경 변수 설명 context: 적용 방법
pg_db_role_setting	Database, User	

2.3 환경 변수

2.3.1 접속과 인증

환경 변수명	설명
listen_addresses	- 서버 리슨 주소 - 모든 접속을 허용하려면 '*' - pg_hba.conf 파일에서 지정하기 때문에, 이 값은 일반적으로 모든 접속 허용으로 설정
port	- 서버 TCP 리슨 포트(PostgreSQL 5432, EDB PAS 5444)
max_connections	- 최대 클라이언트 접속 허용 개수. - 세션 프로세스는 세마포어와 공유 메모리, 세션 메모리를 사용하기 때문에, OS 각 설정을 함께 고려해서 설정
superuser_reserved_connections	- Super 유저 허용 접속 개수. vacuum 작업 고려하여 설정

2.3.2 메모리

환경 변수명	설명
shared_buffers	- 공유 메모리 영역에서 사용하는 데이터 블록 저장 공간, 오라클의 Buffer cache와 동일 개념
work_mem	- 각 세션 별로 사용하는 정렬과 집계 및 기타 작업에 사용하는

	메모리 - 접속하는 클라이언트 당 사용할 수 최대 메모리 - 접속하는 세션 수와 OS의 물리적 메모리 크기 고려 필요
maintenance_work_mem	- vacuum, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY 작업에서 사용하는 메모리
temp_buffers	- 각 세션 단위로 사용할 수 있는 임시 버퍼 값
max_prepared_transactions	- prepared 문장을 동시에 사용할 수 있는 트랜잭션의 최대 수
temp_file_limit	- OS의 swap과 같은 역할을 하는 디스크 영역의 제한 값. 한 트랜잭션이 이 수치를 넘으면 트랜잭션이 취소됨

2.3.3 쿼리 실행

옵티マイ저의 쿼리 실행 계획과 관련된 환경 설정입니다.

환경 변수명	설명
effective_cache_size	- 쿼리 최적화기에서 사용할 하나의 쿼리가 사용할 최대 OS 캐시 크기의 추정치
seq_page_cost	- 시퀀스 검색의 비용 추정치
random_page_cost	- 임의의 데이터 블록을 읽는데 드는 비용 - seq_page_cost 값의 몇 배의 비용이 드는지 설정
enable_관련 환경 변수	- 쿼리 실행 관련 환경 변수 - 필요 시 세션 단위로 조절 필요

2.3.4 트랜잭션 로그

환경 변수명	설명
wal_level	- 트랜잭션 로그의 용도 설정 <ul style="list-style-type: none"> • minimal: 서버 재기동할 수 있는 최소한의 정보 • archive: 트랜잭션 로그 archiving - replaying 기반 대기 서버나 백업 서버 구축 • hot_standby: 스트리밍 리플리케이션 서버 구축 • logical: Logical Decoding
wal_buffers	- WAL buffer의 메모리 크기 - 기본 값은 -1로 공유 메모리 값에 따라 자동 조절

2.3.5 체크 포인트

환경 변수명	설명
checkpoint_timeout	- 세션 체크포인트, 트랜잭션 로그 전환 작업 없어도 checkpoint 강제 수행 시간

max_wal_size / min_wal_size	<ul style="list-style-type: none"> - 트랜잭션 로그 파일 수 설정 - 9.5 이전 버전: checkpoint_segments - checkpoint_segments는 max_wal_size = (3 * checkpoint_segments) * 16MB로 계산하여 비슷하게 설정 가능
-----------------------------	--

2.3.6 Writer 프로세스

환경 변수명	설명
bgwriter_delay	- bgwriter의 활동 간의 시간 차이
bgwriter_lru_maxpages	- bgwriter 프로세스의 한 번 작업에 쓰는 최대 버퍼 수

2.3.7 Vacuum

환경 변수명	설명
vacuum_cost_delay	- vacuum 프로세스 활동간의 시간 차이
vacuum_cost_limit	- vacuum_cost_page_hit + vacuum_cost_page_miss, vacuum_cost_page_dirty 값으로 작동

2.3.8 Autovacuum

환경 변수명	설명
autovacuum	- autovacuum launcher 실행 여부
autovacuum_max_workers	- autovacuumworker 프로세스의 최대 개수

2.3.9 로그

log_로 시작하는 환경 변수는 서버 운영, 장애 사후 분석, 모니터링 등 운영에 필요한 정보 수집을 위해 사용합니다. PostgreSQL은 운영을 위한 도구나 시스템 카탈로그가 오라클에 비해 부족하기 때문에 환경 변수에 대해 꼼꼼히 설정을 해야 합니다. 로그 관련 환경 변수는 reload 명령으로 가능하기 때문에 필요에 따라서 언제든지 설정을 바꾸어 가면서 운영 할 수 있습니다.

환경 변수명	설명
logging_collector	서버 로그를 pg_log 디렉터리의 파일로 기록 여부
log_directory	서버 로그 저장 위치
log_line_prefix	로그 형식 지정
log_min_duration_statement	로그에 기록할 Slow 쿼리 시간
client_min_messages	클라이언트에 보낼 로그 메시지 레벨(error, warning, notice, log, debug1~debug5)

log_min_messages	서버에 보낼 로그 메시지 레벨(error, warning, notice, log, debug1~debug5)
log_rotation_age	로그 파일 전환 주기
log_rotation_size	로그 파일 전환 크기

2.3.10 Lock

기본적으로 특별히 설정 할 것은 없습니다만, two-phase commit 기능을 사용한다면 max_locks_per_transaction, max_pred_locks_per_transaction, max_prepared_transactions 값을 조정해야 합니다. lock 정보는 공유 메모리를 사용하는데, 최대 사용량은 (max_connections + max_prepared_transactions) * max_locks_per_transaction * 270 byte입니다. lock 관련 설정 값이 잘못 설정되면 공유 메모리 할당이 불가능하여 서버가 실행되지 않기 때문에 초기 설정 시 서버가 실행 된다면 크게 문제가 없습니다.

2.4 접근 정보 제어

서버와 어플리케이션 접근 권한 설정은 \$PGDATA/pg_hba.conf에서 설정 가능합니다. pg_hba.conf는 호스트 기반 클라이언트 접근 권한 정보입니다.

[pg_hba.conf 설정 파일]

#	TYPE	DATABASE	USER	ADDRESS	METHOD
# "local" is for Unix domain socket connections only					
local	all	all			md5
# IPv4 local connections:					
host	all	all		127.0.0.1/32	md5
# IPv6 local connections:					
host	all	all		::1/128	md5
# Allow replication connections from localhost, by a user with the # replication privilege.					
#local	replication	enterprisedb			md5
#host	replication	enterprisedb		127.0.0.1/32	md5
#host	replication	enterprisedb		::1/128	md5

표기 형식은 1. 어떤 접속 방식을 2. 어느 데이터베이스를 3. 누가 4. 어디서 접속하느냐에 따라 5. 무조건 접근을 거부/허용하거나 지정한 인증 방식으로 인증에 성공한 클라이언트만 접속을 허용합니다.

각 필드의 의미는 아래와 같습니다.

필드 구분	필드 값	설명
TYPE		접속 종류
	local	Unix 도메인 소켓 사용 접속
	host	모든 TCP/IP
	hostssl	SSL 접속 TCP/IP
	hostnossal	SSL 접속이 아닌 TCP/IP
DATABASE		데이터베이스명
	all	모든 데이터 베이스
	sameuser	데이터베이스와 같은 이름
USER		데이터베이스 유저
ADDRESS		접속 주소 0.0.0.0/0 설정 시 모든 IP 허용
METHOD		인증 방식
	trust	모든 접속 허가
	reject	모든 접속 거부
	md5	md5 암호화된 패스워드 인증
	crypt	crypt 암호화된 패스워드 인증
	password	암호화되지 않은 패스워드 인증
	krb5	TCP/IP 접속시 kerberos V5 인증
	ident	OS 사용자와 동일한 DB 사용자인 경우 인증 허용
	ldap	LDAP 인증
	pam	OS에서 제공하는 PAM(Pluggable Authentication Modules) 서비스 인증

설정은 파일 수정 후 로딩(reload)으로 변경사항 적용이 됩니다.

[예시] pg_hba.conf 설정

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	local	all	all		trust
	host	all	all	127.0.0.1/32	trust
	host	all	all	0.0.0.0/0	md5

첫 번째 줄은 데이터베이스가 운영되고 있는 호스트에서 클라이언트가 실행되고, 접속할 서버의 호스트 주소를 따로 지정하지 않는 경우, unix 소켓 파일을 이용한 접속을 할 것이며, 이 접속에 대해서 비밀번호 인증 없이 모든 접속을 허용함을 뜻입니다.

두 번째 줄은 인증이 허용되지 않는 경우 (호스트 주소로 호스트 IP를 지정 해서 TCP 소켓을 이용) 가운데, 클라이언트가 실행된 호스트의 주소가 127.0.0.1 이면 비밀번호 인증 없이 모든

접속을 허용함을 뜻입니다.

세 번째 줄은 그 외 모든 외부 접속에 대해서는 모든 DB 계정이 모든 데이터베이스에 대해서 비밀번호 인증으로 허용합니다

3장. 유ти리티

3.1 psql 명령어

psql은 command line 기반 클라이언트 툴로 PostgreSQL을 손쉽게 사용하고 관리할 수 있습니다.

구분		사용 방법 및 예시	비고
실행과 종료	실행	psql -h 127.0.0.1 -p 5432 -U opensource -d test	-h: hostname -p: 포트 -U: 사용자 -d: 데이터베이스명
	종료	\q	
데이터베이스 접속	접속	\c 접속할 데이터베이스 명 접속 사용자명	
	확인	\c	
객체 확인	데이터베이스	\l [+] [패턴]	+은 자세히 설명
	사용자	\du [+] [패턴]	
	스키마	\dn [+] [패턴]	
	객체	\d[f i s t v] [+] [패턴]	함수, 인덱스, 시노님, 테이블, 뷰
	객체 설명	\d 객체명	
쿼리 파일 저장	commend	\s 파일이름	
	query buffer	\w 파일이름	
	파일 EDIT	\e 파일이름	vi 편집기로 가능
	쿼리 결과	\o 파일이름	오라클의 spool 기능
기타	쿼리 시간	\timing on	쿼리 속도 측정
	파일 실행	\i 파일	
	shell 명령	\! [command]	
	psql	\?	psql 옵션 설명
	SQL	\h [command]	SQL syntax 설명

3.2 데이터 Load

대량의 데이터를 Load하기 위해서 copy 구문을 사용합니다. 데이터 export와 import 기능을 제공하며, directly read from or write to a file로 빠른 속도를 보장합니다. 슈퍼 사용자가 아닌 일반 사용자도 사용할 수 있습니다.

[copy 사용 방법]

```
- export: \COPY 테이블명 TO '경로/파일명';
- import: \COPY 테이블명 FROM '경로/파일명';
```

copy문에 조건문을 사용한 SELECT 구문으로 사용 가능합니다.

[예시] 조건절을 이용한 copy

```
\COPY (SELECT empno FROM emp WHERE empno = '1') TO '/tmp/emp.csv';
```

원격 서버에서 데이터베이스 서버에 있는 파일을 참조하는 경우 copy를 사용해야 합니다.

PostgreSQL 9.3 이후 버전부터는 copy와 shell 명령어를 함께 사용할 수 있는 부가 옵션을 제공하고 있습니다. 데이터의 크기가 클 경우 압축, split 명령과 함께 사용하면 유용합니다.

[예시] shell 명령어를 활용한 copy

```
\COPY emp TO PROGRAM 'gzip > /tmp/hr.emp.gz'
```

copy 명령어를 이용하여 테이블에 import를 수행 중 에러가 발생하면 import를 전체 롤백이 됩니다.

4장. Maintenance Tasks**4.1 Explain and Analyze**

사용자가 쿼리를 실행하면 쿼리를 파싱하고 옵티マイ저는 실행계획을 세워 쿼리를 수행합니다.

쿼리가 실행되기 전까지의 과정을 알아보고 explain와 analyze가 왜 필요한지 알아봅니다.

[쿼리 실행 단계]

```
SET debug_print_parse = 'on';
SET debug_print_rewritten = 'on';
SET debug_print_plan = 'on';
SET client_min_messages = 'debug1';
SELECT substr(name, 1,8) FROM mytable WHERE num = 10;

LOG: parse tree:
DETAIL: {QUERY ....}
LOG: rewritten parse tree:
DETAIL: ( {QUERY.....})
LOG: plan:
DETAIL: {PLANNEDSTMT.....}
substr
```

```
-----
rkvoawvq
(1 row)
```

하나의 쿼리가 실행되기 전까지 세 단계를 거칩니다

○ Parsing

입력한 쿼리가 SQL 문법상 문제가 없는지 확인하며, SQL 단어들에 대한 토큰 분리가 모두 성공하는지를 체크합니다.

○ Rewriting

쿼리 실행 계획을 세우기 위해서 쿼리 최적화에서 사용할 구문으로 쿼리가 재 작성됩니다. 객체, 자료형, 임계치 등을 검사합니다. 인덱스를 쉽게 사용할 수 있도록 조건절의 좌우변 변환 작업도 같이 수행됩니다.

○ Planning

쿼리 최적화 작업을 진행하면서, 최적의 실행계획을 세웁니다. 이때, pg_statistic 시스템 카탈로그 테이블을 참조해서 비용을 계산하여 가장싼 비용의 실행계획을 선택하게 됩니다.

4.1.1 Explain

옵티마이저가 세우는 쿼리 실행 계획을 확인할 수 있습니다.

[explain 사용 방법]

```
EXPLAIN (ANALYZE true, VERBOSE true, COSTS true, BUFFERS true, TIMING true) 실행
쿼리
```

옵션에 따라 explain의 실행 방법과 결과가 다릅니다.

[예시 1]

```
EXPLAIN SELECT * FROM emp;
          QUERY PLAN
-----
Seq Scan on emp  (cost=0.00..1.14 rows=14 width=45)
(1 rows)
```

[예시 1]은 성능 예측만 가능합니다. 첫 번째 value(cost)는 쿼리의 데이터가 추출되기 전까지의

추정 비용이며 두 번째 value(rows)는 리턴 될 row수, 마지막 값(width)은 한 row당 byte 값을 의미합니다.

[예시 2]

```
EXPLAIN (ANALYZE true, VERBOSE true, COSTS true, BUFFERS true, TIMING true)
SELECT * FROM mytable WHERE num = 10;

QUERY PLAN
-----
Index Scan using mytable_pkey on public.mytable
(cost=0.00..8.27 rows=1 width=22) (actual time=0.100..0.102 rows=1 loops=1)
  Output: num, name
  Index Cond: (mytable.num = 10)
  Buffers: shared hit=4
Total runtime: 0.138 ms
```

[예시 2]는 explain 명령을 자세히 보기 위해 여러 옵션을 추가하였습니다. 실행 계획 상, 위의 쿼리는 index scan 작업을 하며 비용은 8.27, 하나의 row가 반환되며, 그 row 자료를 추출하기 위한 공간이 22 정도 소요되고 공유 메모리에 있는 버퍼 캐시를 4개 읽을 것으로 추정됩니다. 주의할 점은 explain option 중 timing true 옵션이 추가되었을 때, plan이 길고 복잡한 쿼리인 경우 explain analyze를 하는 시간이 추가되므로 실제 쿼리 수행시간과 편차가 발생합니다. 따라서 정확한 시간 추출을 위해 timing false 옵션 추가해야 합니다. .

explain에서 ANALYZE를 사용할 경우 실제 쿼리가 실행 됨에 주의해야 합니다. DML이 실제 수행되므로 데이터의 변경 없이 ANALYZE 옵션을 사용하고자 한다면 트랜잭션으로 처리하여 rollback 처리해야 합니다.

4.1.2 Analyze

Analyze는 옵티マイ저가 올바른 쿼리 실행 계획을 세우기 위해 테이블의 통계 정보를 수집합니다. pg_stat_all_tables에 last analyze 정보 조회가 가능하며 pg_class에 실제 통계 정보들이 저장됩니다.

[analyze 사용 방법]

```
ANALYZE 객체명;
```

전체 데이터베이스와 테이블 단위로 통계 수집이 가능합니다.

4.2 Vacuum

PostgreSQL에서는 테이블의 Update와 Delete 수행 시 row를 바로 반환하지 않습니다. 그래서 테이블에 DML이 계속 발생하게 되면, 쓸모 없는 이전 로우들이 끊임 없이 생기게 됩니다. 그래서 데이터 공간이 계속 늘어나게 됩니다. 이를 방지하기 위해 vacuum 기능이 제공됩니다.

○ Vacuum 필요성

- update나 delete 된 rows들에 의해 차지하고 있던 디스크 공간을 복원하거나 재사용
- Query planner에 의해 사용되는 통계치를 갱신
- visibility map을 갱신
- transaction ID wraparound에 의한 데이터 손실 방지

4.2.1 Vacuum

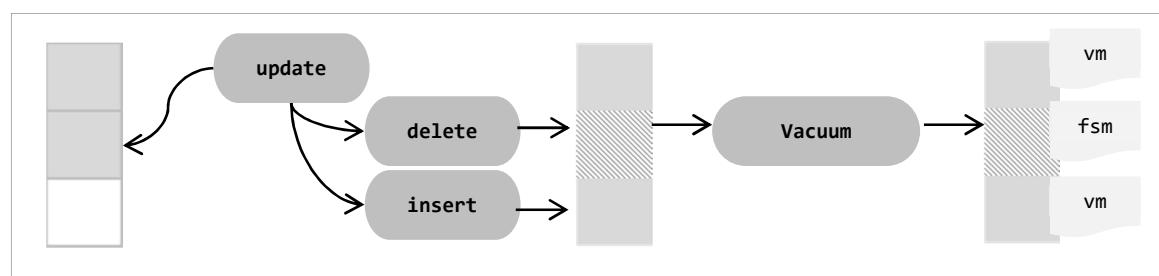


그림 4-1 Vacuum 동작 그림

vacuum 명령어로 특정 테이블의 더 이상 쓰지 않는 row의 정리 작업을 할 수 있습니다. 이 작업은 주로 대량 delete, update 작업이 일어난 경우와 autovacuum 프로세스가 다른 테이블 작업을 하고 있을 때 사용합니다.

vacuum 작업이 끝나게 되면 사용되는 row를 제외하고는 모두 free space로 정리됩니다. 그래서 다른 insert, update 작업이 일어나면, 다른 블록을 사용하지 않고 해당 블록을 재활용합니다. vacuum 작업이 완료되면 fsm 파일과, vm 파일이 만들어집니다. vm파일은 select 작업에서 full sequence scan 작업 비용을 최소화합니다. fsm 파일은 insert, update 작업에서 free space로 활용될 수 있도록 합니다.

vacuum은 ShareUpdateExclusiveLock 을 획득하기 때문에 DML이 가능합니다.

4.2.2 Vacuum full

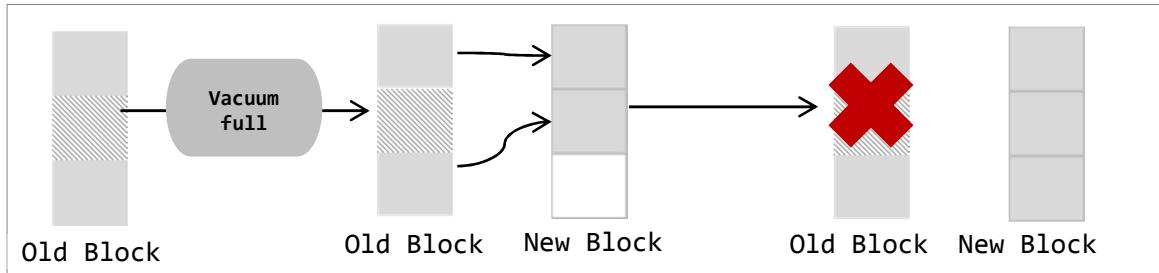


그림 4-2 Vacuum full 동작 그림

vacuum full은 실제 shrink 작업을 수행하여 실제 row를 바로 반환합니다.

vacuum full 작업을 하는 경우 데이터와 트랜잭션 로그의 여유 공간이 필요합니다

AccessExclusiveLock을 획득하므로 실제 운영 중에는 사용 불가능하며 vacuum 보다 수행시간이 많이 소요됩니다.

4.2.3 Autovacuum

통계 자료 갱신과 데이터 블록 관리를 서버가 자동 관리하는 기능입니다. 일정 조건이 충족될 경우 Autovacuum 프로세스가 자동 수행과 중단을 반복합니다.

[Autovacuum 실행 조건]

```
vacuum threshold = vacuum (base) threshold (50) + vacuum scale factor(0.2) *
number of tuples
```

update되거나 delete된 rows 수가 vacuum threshold보다 커지면 autovacuum 수행됩니다. 예를 들어 dept 테이블이 4 rows 변동이 있으면, vacuum threshold 은 $50 + 0.2 * 4$ 이 되며, update(or delete)가 51번 수행되면 vacuum threshold 50.8을 넘어서 autovacuum이 수행됩니다. autovacuum_* 으로 시작하는 환경 변수로 autovacuum on/off, 실행 주기, 작동 시점 등 조절 가능합니다.

5장. 모니터링

5.1 Catalog

데이터베이스 운영에 필요한 정보를 시스템 카탈로그 스키마를 통해 제공하고 있습니다.

이름	타입	설명	비고
edb_partdef	table	파티션 부가속성	EDB PAS 만 제공

edb_partition	table	파티션 정보	EDB PAS 만 제공
pg_attrdef	table	칼럼정보	
pg_attribute	table	그룹	
pg_auth_members	table	사용자	
pg_class	table	객체들	
pg_database	table	데이터베이스	
pg_depend	table	객체 의존정보	
pg_index	table	인덱스	
pg_namespace	table	스키마	
pg_proc	table	함수	
pg_rewrite	table	룰(rule)	
pg_statistic	table	pg_stat_* 뷰의 테이블	
pg_tablespace	table	테이블스페이스	
pg_trigger	table	트리거	
edb_dblink	view	dblink	EDB PAS 만 제공
edb_package	view	package	EDB PAS 만 제공
edb_pkgelements	view	package	EDB PAS만 제공
pg_locks	view	Lock 정보	
pg_settings	view	환경설정값	
pg_stat_activity	view	현재 실행되는 쿼리	
pg_stat_all_indexes	view	인덱스 통계 정보	
pg_stat_all_tables	view	테이블 통계 정보	
pg_stat_bgwriter	view	writer 통계 정보	
pg_stat_database	view	Database 통계 정보	
pg_statio_all_indexes	view	Index buffer hits	
pg_statio_all_sequences	view	Sequences buffer hits	
pg_statio_all_tables	view	Table buffer hits	
session_waits	view	waits	EDB PAS만 제공
system_waits	view	waits	EDB PAS만 제공

대표적인 시스템 테이블과 뷰에 대해서 알아보도록 합니다.

○ pg_class

테이블, 인덱스, 뷰 등의 객체 정보가 확인 가능합니다.

[예시] 객체 정보 확인

```
SELECT oid, relname, relkind FROM pg_class WHERE relname='address';
```

oid	relname	relkind
33143	address	r

- relkind: 객체 타입. r은 테이블, i는 인덱스, f는 함수

○ pg_settings

환경 변수의 현재 설정값, 기본값, 설명 등 확인 가능합니다.

[예시] 데이터베이스 최대 접속 설정 확인

SELECT name, setting, context FROM pg_settings WHERE name = 'max_connections';		
name	setting	context
max_connections	100	postmaster

- context: 환경 변수의 적용 방법(backend, sighup: 데이터베이스 재 시작 없이 reload 적용 가능 / internal: 직접 변경 불가능 / postmaster: 데이터베이스 재기동 필요 / superuser: superuser만 설정 가능 / user: user 세션별로 적용 가능)

○ pg_stat_activity(오라클의 V\$SESSION에 해당)

현재 실행 중인 세션 정보와 쿼리 정보를 확인할 수 있습니다. 운영 상에 가장 활용도가 큰 뷰입니다.

[예시] 현재 데이터 베이스 접속 세션 수

SELECT count(*) FROM pg_stat_activity;	
count	
3	

[예시] 세션 별 쿼리와 실행 시간 확인

pid	,waiting	,current_timestamp - least(query_start, xact_start) AS runtime	,substr(query, 1, 25) AS query
FROM pg_stat_activity			
WHERE NOT pid = pg_backend_pid();			
pid	waiting	runtime	query

26373	f	00:02:08.158115	SELECT CD_NAME, MIN(CD_ID
27868	f	00:00:02.239603	SELECT CD_NAME, MIN(C D_ID
24455	f	00:20:47.107681	SELECT PROGR_STEP_CD, SAN
27866	f	00:02:04.004775	SHOW TRANSACTION ISOLATIO
3023	f	00:00:50.196728	SELECT 1
3022	f	00:00:52.704439	SELECT 1
3020	f	00:00:50.2377	SELECT 1

○ pg_stat_all_tables

테이블의 통계 정보를 제공하며, 이 정보를 확인하여 Maintenance 작업을 진행합니다. last_vacuum, last_autovacuum, last_analyze, last_autoanalyze가 실행된 지 오래됐을 경우, 통계 정보가 정확하지 않고 FSM/VM 작업이 이루어 지지 않아 성능에 문제가 생길 수 있습니다.

○ pg_statio_user_tables / pg_statio_user_indexes

테이블의 조회에 따라 공유 메모리의 데이터 블록 히트율 확인 가능합니다. 데이터가 많이 증가하게 되면 초기에는 히트율이 떨어지나 조회가 빈번하면 점차 히트율은 점점 증가합니다. 이 정보를 통해 메모리 설정이 적정한지 확인해 볼 수 있습니다.

[예시] 테이블의 히트율 확인

relname	hit_pct	heap from cache	heap from disc
employee	0.00000000000000000000000000000000	0	1

[예시] 인덱스의 히트율 확인

indexrelname	hit_pct
,cast(idx_blk_hit AS NUMERIC) / (idx_blk_hit + idx_blk_read) * 100	

```

FROM pg_statio_user_indexes
WHERE relname = 'tb_cctv';

indexrelname | hit_pct | idx_blk_hit | idx_blk_read
-----+-----+-----+-----+
tb_cctv_pk | 0.00000000000000000000000000000000 | 0 | 3

```

○ session_waits / system_waits

EDB PAS만 제공하는 뷰로, snapshot wait 정보가 확인 가능합니다. 주기적으로 확인하여 데이터베이스 분석이 가능합니다

EDB PAS는 오라클 호환 기능으로 DBA로 시작하는 뷰를 제공하고 있습니다. 단, 오라클의 딕셔너리와 칼럼과 정보가 같지 않으므로 PostgreSQL 고유의 pg_로 시작하는 시스템 카탈로그를 활용해야 합니다.

5.2 일반 모니터링

데이터베이스를 운영하면서 상시적으로 모니터링 해야 하는 항목과 방법에 대해 알아봅니다.

5.2.1 프로세스

Postmaster 프로세스와 백그라운드 프로세스가 정상적으로 실행 중인지 확인해야 합니다. 그 외에 대량의 트랜잭션 발생을 할 경우 "autovacuum: VACUUM Table_name (to prevent wraparound"라는 프로세스가 발생하는지 확인해야 합니다. 만약, 발생할 경우 명시적으로 VACUUM 작업 수행을 해야 데이터베이스가 안정적으로 운영될 수 있습니다.

[프로세스 확인]

```

ps -ef | grep postgres
501      1799      1  0 13:55 pts/1    00:00:00 /opt/Postgres/9.5/bin/postgres
501      1800  1799  0 13:55 ?        00:00:00 postgres: logger process
501      1802  1799  0 13:55 ?        00:00:00 postgres: checkpointer process
501      1803  1799  0 13:55 ?        00:00:00 postgres: writer processes
501      1804  1799  0 13:55 ?        00:00:00 postgres: wal writer process
501      1805  1799  0 13:55 ?        00:00:00 postgres: autovacuum launcher
process
501      1806  1799  0 13:55 ?        00:00:00 postgres: stats collector process

```

[세션 프로세스 확인]

```
- 9.2 버전 이전: ps -eHf | grep "idle in transaction"
- 9.2 버전 이후: SELECT * FROM pg_stat_activity WHERE state = 'idle in
transaction';
```

5.2.2 파일 시스템

디스크 공간 관리는 OS 명령어로 관리해야 합니다. 지속적으로 데이터가 저장되는 디렉토리, 트랜잭션 로그 영역, 아카이브 로그 영역의 임계치를 확인해야 합니다. 파일 시스템 장애 중 가장 많이 일어나는 장애는 아카이브 디렉터리에 공간이 없어 트랜잭션 로그를 아카이빙 하지 못하여 트랜잭션 로그 디렉터리의 공간이 차서 데이터베이스가 비정상적으로 종료되는 경우입니다.

[파일 시스템 확인]

```
df -h
```

5.2.3 로그

서버의 로그는 \$PGDATA/pg_log 디렉토리에서 확인 가능합니다. 로그로 데이터베이스의 변경 사항, 에러, Slow query 등이 확인 가능합니다.

[서버 로그 설정]

```
logging_collector = on    #(수정 후 서버 재 시작 필요 Parameter)
log_min_duration_statement = 500 (0.5)sec    # 쿼리 duration 시간 ( -1 : error
only, 0 : all )
log_line_prefix = '%t %u@%r/%d %p'    # log prefix 형식
```

- %t: 시간 %u: 사용자 %r: remote host and port %d: 데이터베이스 이름 %p: 프로세스 ID

5.3 쿼리 모니터링

실무에서 활용 가능한 쿼리 모니터링을 제공합니다.

[long running 쿼리 확인]

```
SELECT pid
      ,waiting
```

```

, current_timestamp - least(query_start, xact_start) AS runtime
, query
, extract(epoch FROM ((timeofday()::TIMESTAMP) - query_start)) AS secs
FROM pg_stat_activity
WHERE NOT pid = pg_backend_pid()
ORDER BY secs DESC;

```

[현재 접속 가능한 세션 수]

```

SELECT (SELECT to_number(setting, '999999')
        FROM pg_settings
       WHERE NAME = 'max_connections')
      - (SELECT count(*)
        FROM pg_stat_activity);

```

[Lock 모니터링]

데이터베이스에서 Lock이란 데이터베이스의 자원을 동시 작업을 제어하기 위한 메커니즘입니다. MVCC 를 구현하기 위해 Lock 이라는 기술요소가 필요합니다. Lock은 먼저 획득한 사용자에 의해 다른 사용자의 작업이 제한되며 Lock의 종류에 따라 작업 내용이 제한됩니다.

PostgreSQL은 8개의 락이 존재합니다.

필요한 Lock 모드	현재 Lock 모드							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

표 5-1 PostgreSQL Lock(출처: postgresql.org)

- **ACCESS SHARE:** 테이블을 SELECT 시 획득하는 Lock
- **ROW SHARE:** SELECT FOR UPDATE, SELECT FOR SHARE 시 획득하는 Lock
- **ROW EXCLUSIVE:** DML 작업 시 획득하는 Lock
- **SHARE UPDATE EXCLUSIVE:** VACUUM, ANALYZE, CREATE INDEX CONCURRENTLY 작업 시에 획득하는 Lock
- **SHARE:** CREATE INDEX 작업 시 획득하는 lock
- **SHARE ROW EXCLUSIVE:** 한 세션에서 데이터 변경을 막기 위해 획득하는 lock (명령어에 의해 실행되지 않음)
- **EXCLUSIVE:** 병렬 프로세스에서 읽기만 가능
- **ACCESS EXCLUSIVE:** ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, and VACUUM FULL

```

SELECT bl.pid AS blocked_pid
      ,a.username AS blocked_user
      ,ka.query AS blocking_statement
      ,TO_CHAR(now() - ka.query_start, 'HH24:MI:SS') AS blocking_duration
      ,kl.pid AS blocking_pid
      ,ka.username AS blocking_user
      ,a.query AS blocked_statement
      ,TO_CHAR(now() - a.query_start, 'HH24:MI:SS') AS blocked_duration
FROM pg_catalog.pg_locks bl
JOIN pg_catalog.pg_stat_activity a ON bl.pid = a.pid
JOIN pg_catalog.pg_locks kl
JOIN pg_catalog.pg_stat_activity ka ON kl.pid = ka.pid ON bl.transactionid =
kl.transactionid
      AND bl.pid != kl.pid
      OR (
            bl.relation = kl.relation
            AND bl.locktype = kl.locktype
            AND bl.pid != kl.pid
      ) WHERE bl.granted = 'f'
  
```

[Lock 세션 확인]

```

SELECT pid
      ,waiting
      ,STATE
      ,query
FROM pg_stat_activity
WHERE waiting = 't';
  
```

[Lock 세션 종료]

- 세션 종료: `SELECT pg_terminate_backend(PID);` --권장
- 세션 취소 `SELECT pg_cancel_backend(PID);`

5.4 모니터링 툴 소개

pgAdmin을 이용해서 간단한 모니터링이 가능합니다. 서버 상태 및 Lock 정보, 로그 확인이 가능합니다. Tool → Server Status 메뉴에서 서버 현황이 조회 가능합니다.

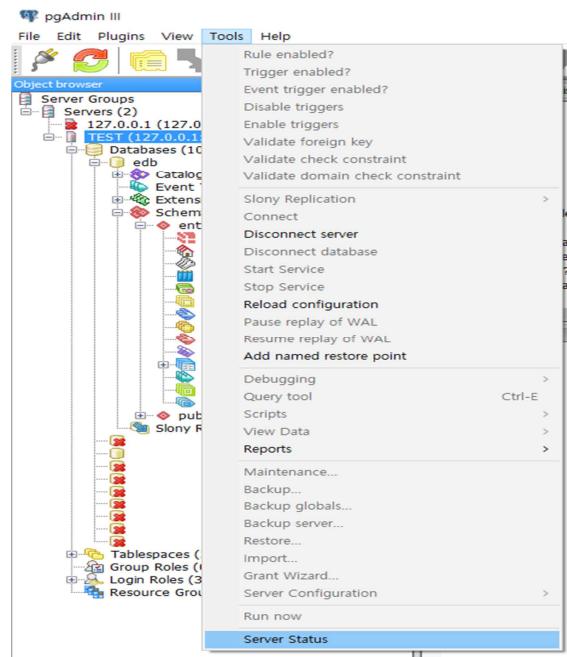


그림 5-1 pgAdmin 서버 모니터링 메뉴 그림

PID	Application name	Database	User	Client	Client start	Que
1953	pgAdmin III - ...	postgres	enterprisedb	10.0.2.2:55003	2016-06-22 09:5...	
1955	pgAdmin III - ...	edb	enterprisedb	10.0.2.2:55004	2016-06-22 09:5...	

그림 5-2 pgAdmin 모니터링 화면 그림

6장. 백업과 복구

시스템의 예기치 못한 재앙이나 사용자 실수에 대비하여 데이터베이스를 정기적으로 백업해야 합니다. 또한 백업 파일을 정확하게 복구하는 것도 중요합니다. 백업 유형에 따라 백업 방법과 복구 방법을 알아보도록 합니다.

6.1 백업

백업은 Hot Backup과 Cold Backup으로 나눠집니다. Hot Backup은 서비스 운영 중에 백업을 수행하는 것을 말하며, Cold Backup은 데이터베이스 엔진을 정지하고 데이터 디렉터리 전체를 백업하는 것을 말합니다.

6.1.1 Cold Backup

데이터베이스 서비스를 정지하고 데이터 디렉터리 전체를 백업하므로 데이터 디렉터리 사이즈를 복사하는 속도만큼 백업 시간이 소요됩니다. 별도의 작업 없이 백업 받은 파일만 있다면 바로 복구 가능합니다.

[예시] Cold Backup 백업

```
cp -a /data/pgdata /backup
```

파일 시스템 복사는 cp 외에 다른 명령어를 사용하셔도 됩니다.

6.1.2 Hot Backup

온라인 백업이라고 하며, 서비스가 운영 중에 백업을 받을 수 있는 기능입니다. SQL Dump와 아카이브 로그 파일을 이용한 백업으로 나누어집니다.

○ SQL Dump

특정 테이블, DDL Only 덤프에 주로 이용 SQL commands 명령어로 텍스트 파일을 생성 덤프 자료는 pg_dump가 실행 시작 시간에 찍은 스냅샷 자료가 됩니다.

[SQL Dump 사용 방법]

```
pg_dump [OPTION]... [DBNAME]
```

[예시] DDL 백업

```
pg_dump -s -n hr pgdb > hr.sql
```

- s: 스키마만 백업, -n: 스키마

[예시] 테이블 데이터 백업

```
pg_dump -a -t emp -n hr pgdb > data.sql
```

-a: 데이터만 백업 -t: 테이블

○ Archive Backup

아카이브 로그 파일을 이용한 백업입니다. 백업과 복구를 위해서는 아래 파일이 필요합니다.

- 데이터 백업본(full 백업)
- 아카이브 로그 파일

아카이브 로그를 이용한 백업과 복구를 수행하기 위하여 백업과 복구가 가능한 경우와 불가능한 경우를 구분해야 합니다. 아카이브 로그를 이용한 복구 같은 경우 데이터 디렉토리의 백업본(full 백업본)에 아카이브 로그를 적용하여 변경 내역을 반영하는 개념입니다. 따라서 full 백업(base 백업)본과 아카이브 파일이 있는 아래의 경우에는 데이터베이스 복구가 가능합니다.

[복구가 가능한 경우]

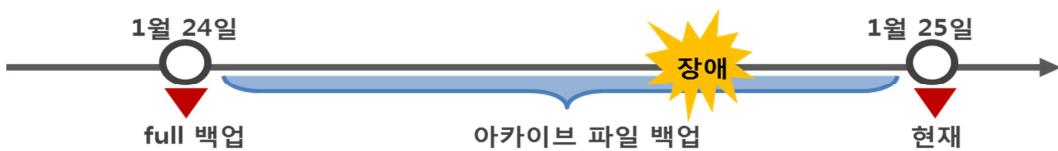


그림 6-1 복구가 가능한 백업

full 백업(base 백업)본과 아카이브 파일 중 일부가 누락된 경우 데이터베이스 복구가 불가능합니다.

[복구가 불가능한 경우]



그림 5-2 복구가 불가능한 백업

pg_dump 명령과 달리 데이터베이스 클러스터(\$PGDATA) 기반으로 작동되기 때문에, 특정 스키마나 특정 테이블 스페이스 단위로 작업을 진행해야 합니다.

Archive Backup의 경우 아카이브 모드로 데이터베이스를 설정해야 합니다. 아카이브 모드의 경우 \$PGDATA/postgresql.conf 파일을 아래와 같은 설정을 해야 합니다.

[아카이브 모드 설정]

wal_level = archive	재시작 필요
archive_mode = on	재시작 필요
archive_command = 'cp %p /opt/PostgreSQL/9.5/archive/%f'	reload로 변경 가능

%p = path of file to archive, %f = file name only

archive_command 설정을 환경설정 파일 reload로 가능하므로 운영 상 이 아카이브 기능을 잠시 중단하려면, archive_command 값으로 'true'를 설정하면 됩니다.

백업은 백업 시작 함수 호출 → 데이터 디렉토리 백업 → 백업 종료 함수 호출 순으로 수행합니다.

[백업 시작 함수]

pg_start_backup(label text [, fast boolean])

백업 시작 함수 pg_start_backup(label text [, fast boolean]) 첫번째 매개변수에는 사용자가 구분할 수 있는 임의의 텍스트를 사용하며, 일반적으로 날짜나 호스트네임을 함께 사용해서 백업 파일에서 중복되지 않도록 합니다. 두번째 매개변수 값이 true면 강제 checkpoint 작업을 합니다. false일 경우 트랜잭션이 빈번하지 않은 시스템인 경우 최대 checkpoint_timeout 값까지 이 명령의 수행 시간이 길어질 수 있습니다.

[예시] 백업 시작 함수

[postgres@localhost ~]\$ psql psql.bin (9.5.1) Type "help" for help. postgres=# SELECT pg_start_backup('ERP_2016-12-24', true); pg_start_backup -----
--

```
0/3000028
(1 row)

[postgres@localhost ~]$ ls $PGDATA
backup_label pg_ident.conf pg_snapshots pg_xlog
base          pg_log        pg_stat      postgresql.auto.conf
global         pg_logical    pg_stat_tmp postgresql.conf
pg_clog        pg_multixact pg_subtrans postmaster.opts
pg_commit_ts   pg_notify    pg_tblspc   postmaster.pid
pg_dynshmem    pg_replslot pg_twophase
pg_hba.conf    pg_serial    PG_VERSION

[postgres@localhost ~]$ cat $PGDATA/backup_label
START WAL LOCATION: 0/5000028 (file 000000010000000000000005)
CHECKPOINT LOCATION: 0/5000028
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2016-02-24 20:43:42 KST
LABEL: ERP_2016-12-24
```

[예시] 데이터 디렉터리 복사

```
[postgres@localhost ~]$ cp -a $PGDATA /tmp/backup
```

데이터 클러스터를 복사하였으면, 백업 종료 함수를 호출합니다. 백업 스냅샷 시점은 백업 종료 함수인 pg_stop_backup가 호출되는 시점입니다.

[백업 종료 함수]

```
pg_stop_backup()
```

[예시] 백업 종료 함수

```
postgres=# select pg_stop_backup;

NOTICE: pg_stop_backup complete, all required WAL segments have been archived
pg_stop_backup
-----
0/3000120
(1 row)

[postgres@localhost ~]$ ls $PGDATA
base          pg_log        pg_stat      postgresql.auto.conf
```

```

global      pg_logical    pg_stat_tmp  postgresql.conf
pg_clog     pg_multixact  pg_subtrans  postmaster.opts
pg_commit_ts pg_notify     pg_tblspc   postmaster.pid
pg_dynshmem  pg_replslot  pg_twophase
pg_hba.conf pg_serial     PG_VERSION

[postgres@localhost ~]$ ls /opt/PostgreSQL/9.5/archive
000000010000000000000003.00000028.backup
[enterprisedb@db1 archive]$ cat 000000010000000000000003.00000028.backup
START WAL LOCATION: 0/3000028 (file 000000010000000000000003)
STOP WAL LOCATION: 0/3000120 (file 000000010000000000000003)
CHECKPOINT LOCATION: 0/3000028
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2016-02-24 20:33:20 KST
LABEL: ERP_2016-12-24
STOP TIME: 2016-02-24 20:40:15 KST

```

아카이브 백업의 경우 full 백업 이후의 파일을 백업 받아두시면 됩니다.

6.2 복구

백업 방법에 따라 복구 방법이 달라집니다.

6.2.1 Cold Backup 복구

데이터베이스의 서비스를 내리고 파일 시스템을 백업 받은 경우는 별도 작업 없이 데이터 클러스터를 지정 후 데이터베이스를 기동하면 됩니다.

[예시] Cold Backup 복구

- 복구: cp -a /backup /data/pgdata
- 데이터베이스 시작: pg_ctl -D /data/pgdata start

6.2.2 Hot Backup 복구

데이터베이스의 서비스를 종료하지 않고 백업을 받았기 때문에 별도의 작업이 필요합니다.

○ SQL Dump

Plain text로 받았기 때문에 쉽게 복구가 가능합니다.

[예시] SQL Dump 복구

```
psql < dump.sql  
psql -f dump.sql
```

○ Full Recovery

아카이브 파일을 이용한 백업은 장애 직후나 특정 시점까지 복구가 가능합니다. 백업된 파일을 Recovery Directory로 이동 → Recovery Directory에 recovery.conf에 파일 생성 → 데이터베이스 시작의 절차로 진행합니다.

[recovery.conf 설정]

```
restore_command: archive 위치
```

[예시] 데이터 디렉터리 복사

```
[postgres@localhost ~]$ cp -a /tmp/backup/9.5/* $PGDATA  
  
[postgres@localhost ~]$ ls $PGDATA  
backup_label pg_ident.conf pg_snapshots pg_xlog  
base pg_log pg_stat postgresql.auto.conf  
global pg_logical pg_stat_tmp postgresql.conf  
pg_clog pg_multixact pg_subtrans postmaster.opts  
pg_commit_ts pg_notify pg_tblspc postmaster.pid  
pg_dynshmem pg_replslot pg_twophase  
pg_hba.conf pg_serial PG_VERSION  
  
[postgres@localhost ~]$ cat $PGDATA/backup_label  
START WAL LOCATION: 0/5000028 (file 000000010000000000000005)  
CHECKPOINT LOCATION: 0/5000028  
BACKUP METHOD: pg_start_backup  
BACKUP FROM: master  
START TIME: 2016-02-24 20:43:42 KST  
LABEL: ERP_2016-12-24
```

[예시] recovery.conf 파일 생성

```
[postgres@localhost ~]$ cp ~/share/recovery.conf.sample $PGDATA/recovery.conf  
  
[postgres@localhost ~]$ cat $PGDATA/recovery.conf  
restore_command = 'cp /opt/PostgreSQL/9.5/archive/%f %p'
```

[예시] 데이터베이스 기동

```
[postgres@localhost ~]$ pg_ctl start

[postgres@localhost ~]$ cd $PGDATA/pg_log

[postgres@localhost pg_log]$ vi postgresql-2016-02-24_212205.log
2016-02-24 21:22:05 KST @/ (1088) LOG: starting archive recovery
2016-02-24 21:22:05 KST @/ (1088) LOG: restored log file
"000000010000000000000003" from archive
2016-02-24 21:22:05 KST @/ (1088) LOG: redo starts at 0/3000090
2016-02-24 21:22:05 KST @/ (1088) LOG: consistent recovery state reached at
0/3000120
2016-02-24 21:22:05 KST @/ (1088) LOG: restored log file
"000000010000000000000004" from archive
.. 생략 ..
2016-02-24 21:22:05 KST @/ (1088) LOG: selected new timeline ID: 2
cp: cannot stat /opt/PostgreSQL/9.5/archive/00000001.history': No such file or
directory
2016-02-24 21:22:05 KST @/ (1088) LOG: archive recovery complete
2016-02-24 21:22:05 KST @/ (1088) LOG: MultiXact member wraparound
protections are now enabled
2016-02-24 21:22:05 KST @/ (1100) LOG: autovacuum launcher started
2016-02-24 21:22:05 KST @/ (1084) LOG: database system is ready to accept
connections

[postgres@localhost ~]$ ls $PGDATA
backup_label.old  pg_dynshmem    pg_multixact  pg_stat      PG_VERSION
base              pg_hba.conf   pg_notify     pg_stat_tmp  pg_xlog
global            pg_ident.conf pg_replslot  pg_subtrans postgresql.auto.conf
pg_clog           pg_log        pg_serial    pg_tblspc   postgresql.conf
pg_commit_ts      pg_logical   pg_snapshots pg_twophase postmaster.opts
```

○ Point-in-Time Recovery (PITR)

특정 시점까지 복구가 필요한 경우는 여러 상황이 있습니다. 대표적으로 사용자 실수로 테이블이나 데이터를 삭제한 경우에 그 직전까지 복구가 가능합니다.

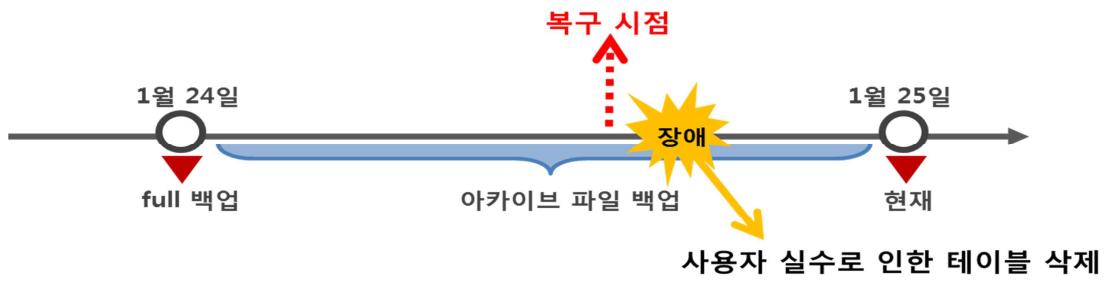


그림 5-3 시점 복구 원리

복구 방법은 Full Recovery와 동일하지만 recovery.conf에 복구 시점을 추가하여 설정하면 됩니다.

[recovery.conf 설정]

```
restore_command: archive 위치
recovery_target_timeline : 복구 시점 지정
```

[예시] 데이터 디렉터리 복사

```
[postgres@localhost ~]$ cp -a /tmp/backup/9.5/* $PGDATA

[postgres@localhost ~]$ ls $PGDATA
backup_label pg_ident.conf pg_snapshots pg_xlog
base          pg_log          pg_stat      postgresql.auto.conf
global        pg_logical      pg_stat_tmp postgresql.conf
pg_clog       pg_multixact   pg_subtrans postmaster.opts
pg_commit_ts  pg_notify      pg_tblspc   postmaster.pid
pg_dynshmem   pg_replslot    pg_twophase
pg_hba.conf   pg_serial      PG_VERSION

[postgres@localhost ~]$ cat $PGDATA/backup_label
START WAL LOCATION: 0/5000028 (file 000000100000000000000005)
CHECKPOINT LOCATION: 0/5000028
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2016-02-24 20:43:42 KST
LABEL: ERP_2016-12-24
```

[예시] recovery.conf 파일 생성

```
[postgres@localhost ~]$ cp ~/share/recovery.conf.sample $PGDATA/recovery.conf

[postgres@localhost ~]$ cat $PGDATA/recovery.conf
restore_command = 'cp /opt/PostgreSQL/9.5/archive/%f %p'
recovery_target_time = '2016-02-24 20:35:15 KST'
```

[예시] 데이터베이스 기동

```
[postgres@localhost ~]$ pg_ctl start
[postgres@localhost ~]$ cd $PGDATA/pg_log

[postgres@localhost pg_log]$ vi postgresql-2016-02-24_212205.log
2016-02-24 21:41:34 KST @/ (1258) LOG: starting point-in-time recovery to
2016-02-24 20:35:15+09
2016-02-24 21:41:34 KST @/ (1258) LOG: restored log file
"000000010000000000000003" from archive
2016-02-24 21:41:34 KST @/ (1258) LOG: redo starts at 0/3000090
2016-02-24 21:41:34 KST @/ (1258) LOG: consistent recovery state reached at
0/3000120
.. 생략 ..
2016-02-24 21:41:34 KST @/ (1258) LOG: record with zero length at 0/7000090
2016-02-24 21:41:34 KST @/ (1258) LOG: redo done at 0/7000028
2016-02-24 21:41:34 KST @/ (1258) LOG: restored log file
"000000010000000000000007" from archive
2016-02-24 21:41:34 KST @/ (1258) LOG: restored log file "00000002.history"
from archive
.. 생략 ..
2016-02-24 21:41:34 KST @/ (1258) LOG: archive recovery complete
2016-02-24 21:41:34 KST @/ (1258) LOG: MultiXact member wraparound
protections are now enabled
2016-02-24 21:41:34 KST @/ (1271) LOG: autovacuum launcher started
postmaster.opts

[postgres@localhost ~]$ ls $PGDATA
backup_label.old  pg_dynshmem    pg_multixact  pg_stat      PG_VERSION
base              pg_hba.conf   pg_notify     pg_stat_tmp  pg_xlog
global            pg_ident.conf pg_replslot  pg_subtrans  postgresql.auto.conf
pg_clog           pg_log        pg_serial    pg_tblspc   postgresql.conf
pg_commit_ts      pg_logical   pg_snapshots pg_twophase postmaster.opts
```

PART 4. SQL

데이터베이스 객체의 특징과 쿼리 문법에 대해 학습합니다.

1장. SQL 종류

SQL(Structured Query Language)은 데이터베이스에서 데이터를 조작하기 위해 사용되는 언어입니다. SQL을 통해 데이터베이스를 생성, 수정, 조회가 가능합니다.

1.1 Query

데이터베이스로부터 데이터를 추출하기 위해 사용하는 명령어입니다. C언어의 printf() 와 Java의 System.out.print() 처럼 데이터를 추출해 결과를 보여 줄 때 사용됩니다.

[SELECT 사용 방법]

```
SELECT select 리스트
FROM 테이블, 뷔
WHERE 조건;
```

select 리스트: 칼럼명, 표현식, 함수

[예시] SELECT 사용

```
SELECT emp_id
      ,first_name || last_name as NAME
      ,salary
      ,salary + 100
      ,'test'
FROM emp
ORDER BY emp_id LIMIT 1;
```

`first_name || last_name`은 칼럼결합의 표현식입니다. `as NAME` 칼럼에 별칭을 사용할 때 씁니다. 칼럼에 연산을 `salary + 100`으로 사용할 수 있으며, 칼럼에 단순 텍스트를 추가하여 표현식으로 출력할 수 있습니다. 출력된 값을 `ORDER BY`로 특정 컬럼으로 정렬 가능하며, `LIMIT` 키워드로 출력 값 수를 제한할 수 있습니다.

1.2 DML

데이터를 조작하는 SQL을 DML (Data Manipulation Language)이라고 하며 데이터를 조작, 저장, 삭제, 수정할 때 사용하는 SQL 구문입니다.

○ INSERT

테이블에 새로운 값을 삽입하기 위해 사용하는 구분입니다.

[INSERT 사용 방법]

```
INSERT INTO 테이블 (칼럼)  
VALUES 값 리스트;
```

[예시] INSERT 사용

```
INSERT INTO countries  
VALUES ('AR', 'Argentina', 2);
```

○ UPDATE

이미 데이터베이스 내에 삽입되어 있는 데이터를 수정 할 때 사용합니다.

[UPDATE 사용 방법]

```
UPDATE 테이블  
SET 칼럼1=값, 칼럼2=값 ...  
WHERE 조건;
```

[예시] UPDATE 사용

```
UPDATE jobs  
SET job_title='test', min_salary=min_salary+10  
WHERE job_id='AC_ACCOUNT';
```

○ DELETE

데이터베이스의 데이터를 삭제 할 때 사용합니다.

[DELETE 사용 방법]

```
DELETE FROM 테이블  
WHERE 조건;
```

[예시] DELETE 사용

```
DELETE FROM jobs
WHERE job_id='ACCOUNT';
```

DML 결과를 바로 출력이 가능합니다. 즉, 업데이트한 로우를 DML문 Returning 문으로 바로 출력 가능합니다.

[예시] DML 결과 바로 출력

```
INSERT INTO test VALUES (test_seq.nextval,current_timestamp) returning a,b;
a | b
-----
16167 | 07-AUG-13 11:03:31.718238

INSERT 0 1
```

1.3 DDL

DDL (Data Definition Language)은 데이터 객체들을 생성 혹은 수정하는데 사용되는 SQL 문입니다. 데이터베이스, 테이블, 인덱스, 뷰, 트리거 등이 생성 가능합니다.

○ CREATE

데이터베이스 객체를 생성. 기본 문법은 "CREATE 객체종류 객체명 ..."이며, 자세한 내용은 "데이터베이스 객체" 참고하시면 됩니다.

○ DROP

데이터베이스 객체를 삭제. 기본 문법은 "DROP 객체종류 객체명 ..."이며, 자세한 내용은 "데이터베이스 객체" 참고하시면 됩니다.

○ ALTER

기존에 이미 생성된 데이터베이스 객체를 수정하는 구문입니다. 테이블에 칼럼을 추가하거나 속성을 변경 가능합니다. 테이블 객체 같은 경우, DROP 후 새로 CREATE 를 하면 운용 중이던 데이터를 손실할 위험이 있기 때문에 이 경우 ALTER를 사용합니다. 자세한 내용은 "데이터베이스 객체" 참고하시면 됩니다.

○ TRUNCATE

DELETE 과 동일한 역할로 테이블의 데이터 삭제합니다. 단, DELETE는 WAL 파일에 작업 내용을 행 단위로 작업을 기록하는 반면 TRUNCATE는 테이블 단위로 기록하므로 수행속도가

빠릅니다. 또한 DELETE는 메모리상의 데이터를 삭제 플래그만 표시하지만 TRUNCATE은 메모리와 데이터 파일에 들어있는 데이터까지 삭제를 합니다.

1.4 트랜잭션

하나의 세션이 쿼리를 수행하는 단위를 트랜잭션이라고 합니다.

○ COMMIT

데이터 변경 작업(INSERT, UPDATE, DELETE)을 한 뒤, COMMIT 명령어 실행하면 데이터 변경 사항이 데이터베이스에 반영이 됩니다.

○ ROLLBACK

ROLLBACK은 COMMIT의 반대 되는 개념으로 변경된 사항은 모두 되돌립니다.

○ SAVEPOINT

ROLLBACK 지점 선택할 수 있습니다.

[예시] COMMIT, ROLLBACK, SAVEPOINT

```
BEGIN;
    UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Alice';
    SAVEPOINT my_savepoint;
    UPDATE accounts SET balance = balance + 100.00 WHERE name = 'Bob';
    ROLLBACK TO SAVEPOINT my_savepoint;
    UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Tom';
COMMIT
```

PostgreSQL은 기본 설정은 autocommit이며 데이터베이스 서버에서 변경 불가능 합니다. 즉, 모든 명령을 단일 트랜잭션으로 실행합니다. 여러 명령을 하나의 트랜잭션으로 처리하려면, 클라이언트 단에서 begin; ~ end; (commit or rollback으로 (명시적으로) 트랜잭션 처리를 해야 합니다. 트랜잭션 내에서 all or nothing으로 트랜잭션이 처리합니다. 즉, 결과가 일부만 반영되지 않아야 하며 모두 반영이 되거나 반영이 되지 않아야 합니다.

PostgreSQL은 DML문 뿐만이 아니라 DDL문도 롤백 가능합니다.

[예시] 트랜잭션 결과

```

kt_test=> BEGIN;
BEGIN
kt_test=> INSERT INTO jobs VALUES ('MI_MAN','MIS Manager',4000,10000);
INSERT 0 1
kt_test=> INSERT INTO jobs VALUES ('MI_MAN','MIS Manager',4000,10000);
ERROR: duplicate key value violates unique constraint "job_id_pk"
DETAIL: Key (job_id)=(MI_MAN) already exists.
kt_test=> commit;
ROLLBACK
kt_test=> SELECT * FROM jobs WHERE job_id='MI_MAN';
 job_id | job_title | min_salary | max_salary
-----+-----+-----+-----
(0 rows)

```

2장. 데이터베이스 객체**2.1 개요**

PostgreSQL 객체는 클러스터 전역 객체와 데이터베이스 지역 객체로 나눠집니다. 클러스터 전역 객체는 하나의 데이터 클러스터(하나의 DB 포트로 운영되는 한 서버를 뜻함)에서 유일합니다.

- 테이블 스페이스
- 데이터 베이스
- DB 그룹 룰과 로그인 룰
- DB 사용자

데이터베이스 지역 객체란 데이터베이스 하위의 스키마부터를 말합니다. 스키마는 논리적인 객체의 그룹이며, 스키마 안에 테이블, 인덱스, 뷰, 함수 등이 있습니다. 지역 객체의 경우 데이터베이스만 다르다면 동일한 이름으로 생성 가능합니다.

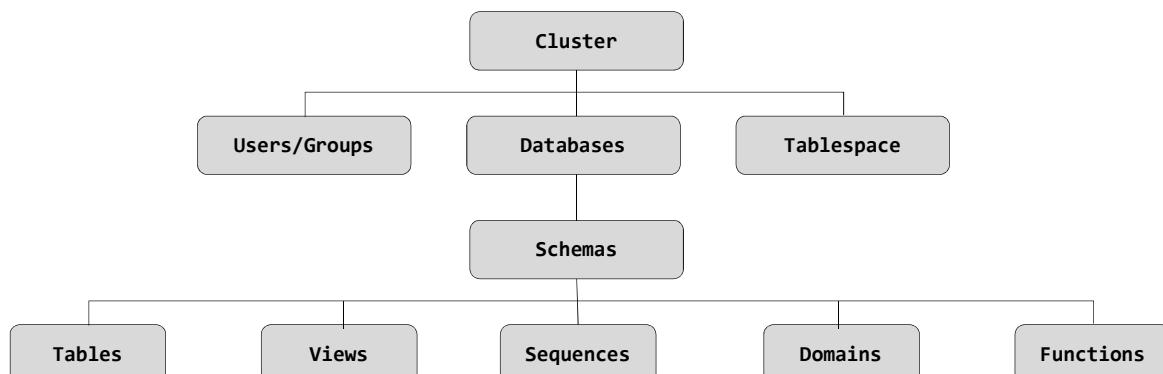


그림 2-1 객체 구조

2.2 테이블

테이블은 데이터를 저장하는 객체입니다.

2.2.1 데이터 타입

PostgreSQL은 다양한 데이터 타입을 제공하며 CREATE TYPE 명령을 사용하여 사용자 데이터 타입을 정의 가능합니다.

○ 수치 타입

구분	설명		크기
정수	smallint	- 일반적으로 디스크 용량에 제한이 있을 때 사용	2 bytes
	integer(int)	- 수치의 범위, 저장 사이즈 및 성능에 있어서 가장 많이 사용되는 타입	4 bytes
	bigint	- integer로 범위가 충분하지 않을 때 사용	8 bytes
소수	numeric	- 정확한 계산에 사용되나 정수 타입에 비해 느림 - NUMERIC(전체 자리 수, 소수 자리 수)	variable
	real	- 부정확하고 가변적인 수치 데이터 형(근사치로 저장되며 운영시스템에 따라 저장 값과 출력 값이 차이가 있을 수 있음)	4 bytes
	float, double		8 bytes
연속형 데이터 타입	smallserial	- 정확히 말하자면 데이터 형은 아니며, 테이블 열의 일련 번호를 설정하기 위한 하나의 표기법(MYSQL의 AUTO_INCREMENT와 유사한 기능) - 연속형 데이터 타입으로 데이터 타입을 설정하면, 내부적으로 시퀀스 객체를 생성되며 이 값은 임의로 값을 INSERT 할 때, 중복될 수 있으며 값을 PK로 사용하려면 제약조건으로 설정	2 bytes
	serial		4 bytes
	bigserial		8 bytes

○ 문자 타입

구분	설명	
고정	character	- char와 동일
가변	character varying	- nvarchar와 동일
	text	- 1GB까지 저장되는 가변 길이 문자 - 오라클의 LONG, NCLOB는 text로 구현 가능

utf8 환경에서의 string length 문제로 PostgreSQL은 text, varchar는 모두 nvarchar로 구현

가능합니다. 고정형인 character의 경우 앞 공백만을 포함하여 문자열을 비교하며, varchar는 앞뒤 공백 모두 포함한다는 점을 주의하셔야 합니다.

○ 날짜/시간 타입

구분	설명
date	- 날짜 저장
time	- 시간 저장
timestamp	- 날짜와 시간 저장 (timestamp with time zone, timestamp without time zone 두 개의 자료형으로 구분)
interval	- 시간 간격 출력

PostgreSQL에서 date은 날짜 정보만 저장하고 time은 시간 정보만 저장합니다. timestamp는 날짜와 시간 모두 저장하지만 EDB PAS는 오라클 호환성을 위해 date, timestamp를 같은 포맷으로 제공합니다.

○ 기타 타입

구분	설명
bytea	- 바이너리 데이터 타입 - 오라클 BLOB, BFILE, RAW, LONG RAW 구현 가능
enum	- 데이터 타입 같은 경우는 check 제약조건으로 구현 가능 - check control_c(control in ('auto','on','off'))

2.2.2 의사 칼럼

의사 칼럼을 의미대로 해석하면 “가짜 칼럼” 정도로 번역할 수 있는데, 이는 일반적으로 “칼럼처럼 보이지만 실제로는 테이블에 저장되지 않은 칼럼”이라고 할 수 있습니다.

SQL 문이 실행될 때만 그 값이 할당되어 마치 값이 존재 하는 것처럼 SELECT가 가능하지만 실제로는 존재하지 않기 때문에 의사 타입에 대한 INSERT, DELETE 등은 불가능합니다.

- oid: 행의 식별자. WITH OIDS 옵션을 사용하여 테이블을 생성할 경우만 생성
- ctid: 행의 물리적인 위치

PostgreSQL은 오라클의 Rownum과 Rowid는 지원하지 않습니다. Rownum은 쿼리의 결과로 나오게 되는 각각의 로우들에 대한 순서값을 나타내는 의사칼럼이며, Rowid는 테이블에 저장된 각각의 로우들이 저장된 주소값을 가진 의사칼럼입니다. Rownum은 윈도우 함수를 사용하여 구현 가능하며 Rowid는 PK를 생성해서 사용해야 합니다.

[예시] Rownum 구현

```
SELECT row_number() over () as rownum, * FROM dept;
```

EDB PAS는 Rownum을 제공하나 오라클과 순서가 상이하므로 ORDER BY로 순서를 조절해야 합니다.

2.2.3 제약 조건

제약조건은 데이터의 무결성을 보장하기 위하여 사용합니다. 즉, 어떤 칼럼에 정확한 데이터가 들어갈 수 있게 칼럼에 규칙을 설정하는 것입니다.

○ CHECK

CHECK 제약조건은 일정 조건에 맞는 데이터만 입력할 수 있도록 합니다.

[CHECK 제약조건 형식]

```
칼럼명 데이터형 CONSTRAINT 제약조건명 CHECK(칼럼명 조건)
```

[예시] CHECK 사용

```
CREATE TABLE adult (name CHAR(3)
    ,gender TEXT CHECK ('man', 'woman')
    ,age INTEGER CONSTRAINT adult_check CHECK (age > 20));
```

○ NOT NULL

NOT NULL 제약 조건은 칼럼에 데이터가 꼭 들어가야 하는 경우에 사용합니다. NOT NULL 조건을 CHECK 제약조건에 추가하여 사용 가능하나 NOT NULL을 직접 명시해 주는 것이 더 효율적입니다.

[NOT NULL 제약조건 형식]

```
칼럼명 데이터형 NOT NULL
```

[예시] NOT NULL 사용

```
CREATE TABLE regions (
    region_id NUMERIC NOT NULL
    ,region_name CHARACTER VARYING(25));
```

○ Unique

Unique는 단어 그대로 그 칼럼의 값이 유일해야 한다는 조건입니다. Unique 제약조건 형식에서도 볼 수 있듯이 여러 개의 칼럼을 명시하여 복합 Unique로 사용 가능합니다. 단, 주의해야 할 점은 Unique는 NULL을 허용한다는 점입니다. NULL은 비교될 수 없기 때문에 중복으로 삽입 가능하기 때문입니다.

[Unique 제약조건 형식]

- 칼럼명 데이터형 CONSTRAINT 제약조건명 UNIQUE
- UNIQUE (칼럼1, 칼럼2)

[예시] Unique 사용

```
CREATE TABLE name (
    id INT UNIQUE
    ,first_name CHAR(2) NOT NULL
    ,last_name CHAR(1) NOT NULL
    ,CONSTRAINT name_unique UNIQUE (first_name, last_name));
```

○ 기본키

기본 키는 테이블 내의 행의 유일한 식별자로 Unique 제한과 NOT NULL 제한의 제약조건 모두 포함됩니다.

[기본키 제약조건 형식]

- 칼럼명 데이터형 PRIMARY KEY
- PRIMARY KEY (칼럼1, 칼럼2)

[예시1] 기본키 사용

```
CREATE TABLE countries (
    country_id CHARACTER(2) PRIMARY KEY
    ,country_name CHARACTER VARYING(40)
    ,region_id NUMERIC);
```

[예시2] 기본키 사용

```
CREATE TABLE countries (
    country_id CHARACTER(2) NOT NULL
    ,country_name CHARACTER VARYING(40)
    ,region_id NUMERIC
    ,CONSTRAINT country_c_id_pk PRIMARY KEY (country_id);
```

○ 외래키

여러 테이블의 칼럼의 데이터들이 참조되는 제약 조건입니다. 참조되는 각 칼럼의 데이터 형과 값은 동일합니다.

외래 키 제약조건이 있는 칼럼에 데이터를 넣을 때에는 참조되는 칼럼에 데이터를 먼저 넣어야 하며 삭제할 때는 참조하는 칼럼의 데이터를 먼저 삭제해야 참조가 된 칼럼의 데이터를 삭제 가능합니다. 한 테이블에 여러 개의 외래 키가 설정 가능합니다. 외래 키를 설정하는 이유는 단순히 칼럼의 데이터를 참조하기 위함도 있으나, 칼럼의 값이 변경될 때(삭제될 때) 참조하는 칼럼에 대해 특정 ACTION을 지정해 줄 수 있기 때문입니다.

[외래키 제약조건 형식]

- 칼럼명 데이터형 REFERENCES 참고테이블(참조칼럼명) ON UPDATE/DELETE ACTION
- FOREIGN KEY (칼럼1, 칼럼2) REFERENCES 참고테이블(참조칼럼1, 참조칼럼2)

[예시 1] 외래키 사용

```
CREATE TABLE countries (
    country_id CHARACTER(2) PRIMARY KEY
    ,country_name CHARACTER VARYING(40)
    ,region_id NUMERIC
    ,CONSTRAINT countr_reg_fk FOREIGN KEY (region_id) REFERENCES
regions(region_id) MATCH FULL ON UPDATE NO ACTION ON DELETE NO ACTION);
```

[예시 2] 외래키 사용

```
CREATE TABLE countries (
    country_id CHARACTER(2) NOT NULL
    ,country_name CHARACTER VARYING(40)
    ,region_id NUMERIC REFERENCES regions(region_id) MATCH FULL ON UPDATE
NO ACTION ON DELETE NO ACTION);
```

외래 키 제약조건은 아래와 같은 Action이 존재합니다.

Action	설명
NO ACTION	어떠한 ACTION 도 하지 않음
RESTRICT	참조되는 행 수정/삭제 제한
CASCADE	참조되는 행이 수정/삭제될 때 그것을 참조하는 행도 수정/삭제
SET NULL	참조되는 행이 수정/삭제될 때 그것을 참조하는 행을 NULL로 설정
SET DEFAULT	참조되는 행이 수정/삭제될 때 그것을 참조하는 행도 DEFAULT로 설정
NO ACTION	어떠한 ACTION 도 하지 않음

○ Default

default는 데이터 제약조건에 포함되지는 않지만 칼럼 단위로 설정되는 속성입니다. default는 칼럼에 특정 값을 default 값으로 설정하면, 테이블에 데이터를 입력할 때 해당 칼럼에 값을 입력하지 않으면 default로 설정한 값이 자동 입력됩니다.

[예시] Default 사용

```
CREATE TABLE telephone_number (
    country code NUMERIC DEFAULT 82
    ,area code NUMERIC
    ,number NUMERIC));
```

○ 제약 조건 예외

제약조건 예외는 특정 칼럼 혹은 연산자에 의하여 비교되며 false 혹은 NULL을 반환합니다. UNIQUE 제약조건과 비슷하나 일반적인 제약조건을 더 추가 가능합니다. exclude는 공간 데이터가 아닌 스칼라 자료형의 경우는 check constraint 사용을 권장합니다. exclude는 index를 자동 생성하는 장점이 있습니다.

[제외 조건 예외 형식]

```
EXCLUDE USING 칼럼명 (비교칼럼명 WITH &&)
```

[예시] 제외 조건 예외

```
CREATE TABLE circles (
    C circle
    ,EXCLUDE USING gist(c WITH &&));
```

중복되는 원을 포함하지 않는 제약조건

```
INSERT INTO circles (c) VALUES ('0,0,100'); SQL 문을 두 번 실행하게 되면 다음과
```

같은 에러 메세지가 출력

```
ERROR: conflicting key value violates exclusion constraint "circles_c_excl"
DETAIL: Key (c)=(<(0,0),100>) conflicts with existing key (c)=(<(0,0),100>).
```

2.2.4 테이블 종류

○ 일반 테이블

- 테이블 생성

사용자는 데이터베이스에 새로운 테이블을 생성할 수 있으며, 생성된 테이블의 소유권은 그 테이블을 생성한 사용자가 갖게 됩니다. 스키마 명을 붙이면 지정된 스키마에 테이블이 생성되며, 스키마 명을 붙이지 않으면 사용자 이름과 동일한 스키마에, 사용자 이름과 동일한 스키마가 없다면 public으로 자동 생성됩니다.

[테이블 생성 구문]

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] 테이블명 ( [
{ 칼럼명 데이터 타입 [ COLLATE collation ] [ 칼럼 제약 조건 ]
| 테이블 제약조건 }
] )
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

Temporary Table의 Global속성과 Local 속성은 표준SQL과의 호환성을 지키기 위해서 문법적으로 받아들이고 있지만, 모두 Local로 생성되므로 의미가 없습니다. Unlogged 옵션은 오라클의 nologging과 동일한 기능이나 많은 제약 사항 존재합니다. 운영 중 변경 불가능 하며 Drop 될 때까지 로깅되지 않으며 데이터베이스 정상 shutdown 외 테이블 저장/데이터 복구 불가합니다.

오라클은 몇몇 예약어(Reserved Word)가 칼럼명으로 사용 가능. 하지만 PostgreSQL은 원칙적으로는 불가능합니다. 불가피하게 사용하고 싶을 경우에는 이중 인용 부호("")를 사용해야 합니다.

[예시] 예약어를 이용한 테이블 생성

```
(잘못된 예) CREATE TABLE TEST(id numeric, offset numeric);
```

```
(바른 예) CREATE TABLE TEST(id numeric, "offset" numeric);
```

```
SELECT * FROM test WHERE "offset"=1
```

테이블 이름은 큰 따옴표를 사용하지 않으면 소문자로 변환되어 생성됩니다. 대문자로 생성된 테이블을 조회하기 위해서는 큰 따옴표 사용해야 합니다.

[예시] 대문자를 이용한 테이블 조회

```
SELECT * FROM "SAMPLE_TABLE";
```

- 테이블 복제

원본 테이블 대상으로 쿼리 결과와 동일한 테이블 구조와 데이터를 가지는 복제 테이블 생성 가능합니다. CTAS를 이용할 경우 테이블 구조, 데이터만 복제되며 like 옵션을 사용시 테이블 구조, not null 제약조건, 데이터가 default로 복제 가능하며, 추가 옵션 설정 시 default 값, 제약조건, index, storage 등도 복제 가능(외래 키 제외)합니다.

[테이블 복제 구문]

- CREATE TABLE 테이블명 AS query;
- CREATE TABLE 테이블명 (like 대상테이블 including DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | ALL)

[예제 1] 테이블 복제

```
CREATE TABLE new_employees AS SELECT employee_id, salary FROM employees WHERE department_id = 100;
```

[예제 2] 테이블 복제

```
CREATE TABLE new_employees (LIKE employees);
```

- 테이블 수정

이미 생성된 테이블을 테이블명 변경, 스키마명 변경, 제약조건 수정, 칼럼 수정 등 테이블 수정 가능합니다.

[테이블 수정 구문]

```
ALTER TABLE [ IF EXISTS ] 테이블명 .... ;
```

[예제 1] 테이블 수정

```
ALTER TABLE locations RENAME TO city;
```

[예제 2] 테이블 수정

```
ALTER TABLE regions ADD CONSTRAINT reg_id_pk PRIMARY KEY (region_id);
```

- 테이블 삭제

데이터베이스 내의 테이블을 삭제할 수 있습니다. 테이블의 소유자만이 삭제 가능합니다. `DROP TABLE`은 연관된 인덱스, 룰, 트리거, 제약 등을 모두 삭제하지만, 뷰 또는 다른 테이블에서 외래키 제약으로 참조되고 있는 테이블까지 삭제하려면 `CASCADE`를 지정해야 합니다. 다만, `CASCADE` 지정 시에도 테이블에 의존하는 뷰는 완전하게 삭제되지만 외부 키 제약에 연관되는 테이블은 키 제약만 사라질 뿐 테이블 자체가 사라지지는 않습니다.

[테이블 삭제 구문]

```
DROP TABLE [IF EXIST] 테이블명 [CASCADE | RESTRICT]
```

○ 파티션 테이블

파티션 테이블은 용량이 큰 테이블을 여러 테이블로 나눠 저장하는 것을 말하며 성능 및 관리 편리성 향상됩니다. PostgreSQL은 파티션 테이블을 상속의 개념으로 구현됩니다.

- 파티션 테이블 생성

파티션 테이블 생성 시, `INSERT` 트리거와 각 하위 테이블에 대해서 인덱스 생성 필요합니다. 리스트 파티션 테이블과 범위 파티션 테이블은 `check` 조건에 의해서 생성됩니다.

[파티션 테이블 생성 구문]

```
CREATE TABLE 테이블명 () INHERITS (상속받을 테이블명);
```

[예시] 파티션 테이블 생성

```
-- 테이블 생성
CREATE TABLE pgbench_partition_table (aid INTEGER PRIMARY KEY,bid INTEGER,
```

```
abalance INTEGER, filler CHARACTER(84));
CREATE TABLE pgbench_partition_table_1 (CHECK (aid < 100001)) INHERITS
(pgbench_partition_table);
CREATE TABLE pgbench_partition_table_2 (CHECK (aid >= 100001 AND aid <
200001)) INHERITS (pgbench_partition_table);

-- 인덱스 생성
CREATE INDEX idx_pgbench_partition_table_1 ON pgbench_partition_table_1(aid);
CREATE INDEX idx_pgbench_partition_table_2 ON pgbench_partition_table_2(aid);

-- 트리거 함수 생성
CREATE OR REPLACE FUNCTION pgbench_partition_table_get_part()
RETURNS TRIGGER AS $$

BEGIN
    IF ( NEW.aid < 100001) THEN INSERT INTO pgbench_partition_table_1 VALUES
(NEW.*);
    ELSIF ( NEW.aid >= 100001 AND NEW.aid < 200001) THEN INSERT INTO
pgbench_partition_table_2 VALUES (NEW.*);
END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;

-- 트리거 생성
CREATE TRIGGER pgbench_partition_trigger
BEFORE INSERT ON pgbench_partition_table
FOR EACH ROW EXECUTE PROCEDURE pgbench_partition_table_get_part();
```

EDB PAS는 오라클의 파티션 생성 및 관리 구문 지원합니다.

- 파티션 테이블 수정

[예시] 파티션 테이블 수정

```
ALTER TABLE pgbench_partition_table_1 NO INHERIT pgbench_partition_table
```

- 파티션 테이블 삭제

[예시] 파티션 테이블 삭제

```
DROP TABLE pgbench_partition_table_1;
```

○ External 테이블

External 테이블은 plain 파일을 읽거나 외부 데이터베이스의 테이블을 읽고 작업할 수 있습니다. External 테이블 생성은 확장 모듈 설치, 서버 생성, 테이블 생성 순으로 진행됩니다.

[예시] External 테이블

```
-- 조회할 외부 테이블 파일(txt) 생성

$cat hr.txt

100|Steven|King|SKING@kt.com|515.123.4567|2003/06/17|24000
101|Neena|Kochhar|NKOCHHAR@kt.com|515.123.4568|2005/09/21|17000
102|Lex|De Haan|LDEHAAN@kt.com|515.123.4569|2001/01/13|17000

-- extension 설치
create extension file_fdw;

-- server 생성
create server kt_test foreign data wrapper file_fdw;

-- foreign table 생성
create foreign table test_table(EMPLOYEE_ID text, FIRST_NAME text, LAST_NAME text, EMAIL text, PHONE_NUMBER text, HIRE_DATE date, SALARY int) server kt_test options (format 'csv', header 'false', filename '/tmp/hr.txt', delimiter '|', null '');

-- 결과 확인

SELECT * FROM test_table;
 employee_id | first_name | last_name |      email       | phone_number | hire_date    | salary
-----+-----+-----+-----+-----+-----+
-----+-----+
 100      | Steven     | King      | SKING@kt.com  | 515.123.4567 | 17 - JUN-03 00:00:00 | 24000
 101      | Neena      | Kochhar   | NKOCHHAR@kt.com | 515.123.4568 | 21 - SEP-05 00:00:00 | 17000
 102      | Lex        | De Haan   | LDEHAAN@kt.com | 515.123.4569 | 13 - JAN-01 00:00:00 | 17000
(3 rows)
```

○ 클러스터 테이블

클러스터 테이블은 테이블이 인덱스 정보에 기반하여 물리적으로 재구성되어 저장되어 있습니다. 테이블의 변경 정보는 클러스터에 반영이 되지 않으며, 읽기 전용 테이블일 경우 구성에 유리합니다. 한 테이블의 하나의 로우를 랜덤 엑세스 할 때는 이점이 없으나 추출하고자 하는 데이터가 하나의 로우 이상일 때, 데이터가 그룹핑과 정렬이 되어 유리합니다.

[클러스터 테이블 생성 구문]

- CLUSTER 테이블명 USING 인덱스명;
- CLUSTER 인덱스명 ON 테이블명;

[예시] 클러스터 테이블 생성

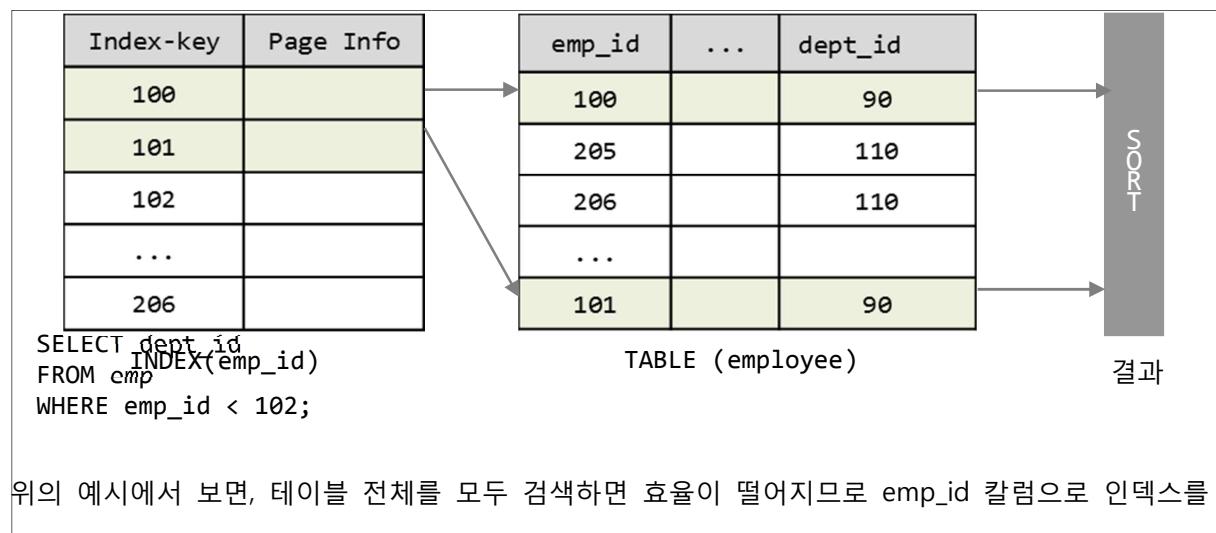
```
CLUSTER employees USING employees_ind;
```

2.3 인덱스

2.3.1 인덱스 개념

테이블의 데이터를 빠르고 효율적으로 검색하기 위해 테이블과 별도로 생성하는 객체입니다.

[예시] 인덱스



위의 예시에서 보면, 테이블 전체를 모두 검색하면 효율이 떨어지므로 emp_id 칼럼으로 인덱스를

생성해 두면 효율적으로 검색할 수 있습니다.

데이터베이스 개발자는 어떠한 칼럼이 빈번하게 사용되는지 잘 예측하여 인덱스 생성 필요합니다.

인덱스는 테이블 의존 객체로 pgAdmin에서 객체 탐색기 창에서는 테이블 항목을 선택해야 하위 분류로 보입니다.

기본키로 생성된 인덱스는 내부적으로 인덱스 생성됩니다. 또한 모든 자료형에 대해서 인덱스 생성이 가능합니다. 예를 들어, text (clob, long 자료형)에 대해서도 컬럼 사이즈가 2-3 kbyte 내에서 인덱스 생성 가능합니다.

PostgreSQL은 B-tree, Hash, GiST, SP-GiST, GIN 인덱스 지원합니다.

2.3.2 인덱스 생성과 삭제

[인덱스 생성 구문]

```
CREATE INDEX 인덱스명 ON 테이블명 USING 인덱스생성방법 (칼럼명 칼럼정렬순서 NULLS NULL정렬);
```

인덱스생성방법: B-tree, Hash, GiST, SP-GiST, GIN / 칼럼정렬순서: ASC, DESC / NULL정렬: FIRST, LAST

[예시] 인덱스 생성

```
CREATE INDEX emp_department_ix ON employees USING btree (department_id DESC NULLS FIRST);
```

[인덱스 삭제 구문]

```
DROP INDEX 인덱스명;
```

2.3.3 인덱스 종류

○ 기본 인덱스

단일 칼럼에 대해 인덱스되는 인덱스입니다.

○ 복합 인덱스

복합 인덱스는 하나 이상의 칼럼을 지정하여 인덱스를 생성하는 것을 말합니다. 복합 인덱스의 칼럼 순서와 정렬 순서가 다를 경우 두 인덱스는 다른 인덱스입니다. 복합 인덱스는 B-tree 와 GiST가 가능합니다.

[예시] 복합 인덱스 생성

```
CREATE INDEX job_inx1 (job_id, job_title);
CREATE INDEX job_inx2 (job_title, job_id);
```

○ UNIQUE 인덱스

UNIQUE 인덱스는 인덱스 생성 시 칼럼에 중복 값 허용이 불가능 합니다. UNIQUE 인덱스는 B-tree 생성 가능 가능합니다.

[UNIQUE 인덱스 생성 구문]

```
CREATE UNIQUE INDEX employees_inx ON employees(employee_id);
```

○ 부분 인덱스

테이블의 부분 값만을 인덱스로 생성할 수 있습니다. 부분 인덱스를 생성하는 이유는 공통되는 값이 인덱싱 되는 것을 방지하고 인덱스의 크기를 줄이기 위해 사용됩니다.

[예시] 부분 인덱스 생성

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

○ 함수 기반 인덱스

인덱스를 생성 시, 칼럼에 함수를 사용하여 함수의 결과값으로 인덱스를 생성할 수 있습니다. 조건절에 함수를 이용하여 칼럼을 가공하면, 인덱스를 사용할 수 없으므로 함수기반 인덱스를 생성하여 실행계획을 인덱스 사용으로 유도할 수 있습니다. 함수 기반 인덱스 함수에서 사용하는 함수는 반드시 immutable 함수여야 합니다.

[예시] 함수 기반 인덱스 생성

```
CREATE INDEX employees_inx ON employees (upper(first_name||last_name));
```

2.4 뷰

2.4.1 뷰 개념

테이블에 대한 가공 정보를 보여주는 객체를 뷰라고 합니다. 테이블처럼 취급되지만, 데이터를 가지고 있지 않은 가상의 테이블입니다. PostgreSQL은 ANSI 호환성 때문에 뷰가 존재하지 내부적으로 뷰는 select 를 rule 로 처리 됩니다. 뷰는 미리 정의된 쿼리만을 조회하게 함으로 써 편의성과 보안적인 측면 제공합니다. 예를 들면 사원 테이블 같은 경우 월급과 같은 민감한 정보를 제외하고 뷰를 생성함으로 써, 다른 부서에서 제한된 데이터만 조회 가능합니다.

2.4.2 뷰 활용

[뷰 생성 구문]

```
CREATE VIEW 뷰명 AS 쿼리
```

[예시] 뷰 생성

```
CREATE VIEW view_emp AS
SELECT first_name, last_name, email, hire_date
FROM employees
WHERE department_id = 20;
```

[뷰 삭제 구문]

```
DROP VIEW 뷰명;
```

[예시] 뷰 삭제

```
DROP VIEW view_emp;
```

PostgreSQL 9.3부터는 Updatable View 기능 제공합니다. 그러나 FROM절 뒤에 테이블, 업데이트 가능 뷰이어야 하며, 단일 테이블이어야 합니다. 또한 뷰 정의에 WITH, DISTINCT, GROUP BY, HAVING, LIMIT, OFFSET, UNION, INTERSECT, EXCEPT, 함수/상수의 사용이 불가능하므로 실무에서 사용하기 어렵습니다.

2.5 시노님

시노님(Synonym)은 스키마 오브젝트의 별칭입니다. PostgreSQL은 시노님을 제공하고 있지 않기 때문에 search_path를 이용하여 스키마명을 생략할 수 있습니다. 단, 관리 편의성을 위해 “스키마.객체”로 사용을 하셔야 합니다.

EDB PAS는 오라클 호환 기능으로 시노님을 사용할 수 있습니다.

2.6 시퀀스

2.6.1 시퀀스 개념

순차적으로 번호를 할당하는 객체입니다. 테이블 생성 시 데이터 타입을 serial/bigserial로 지정하면 내부적으로 시퀀스 객체를 생성하여 순차적으로 번호 할당 가능합니다. 정보를 제외하고 뷰를 생성함으로써, 다른 부서에서 제한된 데이터만 조회 가능합니다.

2.6.2 시퀀스 활용

[시퀀스 생성 구문]

```
CREATE SEQUENCE 시퀀스명
MINVALUE 최소값
MAXVALUE 최대값
INCREMENT BY 증가값
START WITH 최소 시작값
```

[예시] 시퀀스 생성

```
CREATE SEQUENCE seq_test;
```

시퀀스 객체를 사용하기 위해서는 오라클은 의사칼럼을 사용하는 것과 달리 PostgreSQL에서는 시퀀스 조작 함수(Sequence Manipulation Functions)를 사용합니다. 시퀀스 조작 함수는 시퀀스 테이블을 제어하기 위해 사용되는 함수입니다.

함수 이름	반환 값	설명
currval(시퀀스명)	bigint	최근의 시퀀스 값을 구함
nextval(시퀀스명)	bigint	시퀀스를 진행하여 새로운 값을 반환
setval(시퀀스명, bigint)	bigint	현재 시퀀스 값을 설정
setval(시퀀스명, bigint, Boolean)	bigint	시퀀스의 현재값과 is_called 플래그를 설정

[시퀀스 삭제 구문]

```
DROP SEQUENCE 시퀀스명;
```

[예시] 시퀀스 활용

```
-- 시퀀스 생성
kt_test=> CREATE SEQUENCE seq_test;
CREATE SEQUENCE

-- 시퀀스 호출
kt_test=> SELECT currval('seq_test');
currval
-----
1

-- 현재 시퀀스 값 확인
kt_test=> SELECT currval('seq_test');
currval
-----
1
```

```
(1 row)

-- 새로운 세션에서 시퀀스 값 호출
kt_test=> \q
[postgres@hjlee 9.3]$ psql -U hr kt_test
psql.bin (9.3.4)
Type "help" for help.

kt_test=> SELECT currval('seq_test');
ERROR: currval of sequence "seq_test" is not yet defined in this session
```

위의 예제에서는 새로운 세션에는 시퀀스가 한번도 호출되지 않았기 때문에 에러가 발생합니다. 오라클은 시퀀스의 Cache가 global cache인 반면 PostgreSQL은 세션 단위이며 시퀀스 생성 시, CACHE 옵션을 NO CACHE로 생성해야 합니다. 그렇지 않을 경우 순번대로 시퀀스가 생성이 되지 않을 수 있습니다.

2.7 함수

함수는 특정 연산을 수행하고 그 결과를 반환해 주는 기능 제공합니다. PostgreSQL에서는 다양한 함수를 기본적으로 제공합니다. 또한 필요에 따라 사용자 함수를 정의하여 사용 가능합니다.



그림2-1 내장 함수

2.7.1 문자형 함수

문자형 함수는 문자형 데이터 값을 조작하거나 검색하는 등의 역할을 하며, 연산자와 함께 사용하여 다양한 연산을 수행 가능합니다. 문자형 함수에서 사용할 수 있는 데이터 형은

character, character varying, text입니다. character형에서 공백 문자가 포함될 경우 공백은 자동으로 padding됨을 주의해야 합니다.

대표적인 문자형 함수에 대해 소개합니다.

함수명	설명	사용 방법	예시
UPPER	입력 값을 모두 대문자로 변환	UPPER('문자열' 또는 칼럼명)	SELECT upper('PostgreSQL'); upper ----- POSTGRESQL
LOWER	입력 값을 모두 소문자로 변환	LOWER('문자열' 또는 칼럼명)	SELECT lower('PostgreSQL'); lower ----- postgresql
INITCAP	입력 값의 첫 글자를 대문자로 변환하고 나머지는 소문자로 변환	INITCAP('문자열' 또는 칼럼명)	SELECT initcap('PostgreSQL'); initcap ----- Postgresql
LENGTH	입력 값의 길이를 계산해 주는 함수	LENGTH ('문자열' 또는 칼럼명)	SELECT length('PostgreSQL'); length ----- 10
CONCAT	입력 문자열을 결합하여 변환	CONCAT('문자열', '문자열')	SELECT concat('Postgre', 'SQL'); concat ----- PostgreSQL
CONCAT_WS		CONCAT('구분자', '문자열', '문자열')	SELECT concat_ws('/', 'Postgre', 'SQL' , '' , '!'); concat_ws ----- Postgre/SQL//! (1 row)
SUBSTR	입력 값의 길이의 문자를 추출할 때 사용	SUBSTR ('문자열' 또는 칼럼명, 시작위치, 추출할 글자수)	substr('PostgreSQL', 1, 3); substr ----- Pos
SPLIT_PART	입력 문자열을 구분자로나눠서 반환	SPLIT_PART ('문자열' 또는 칼럼명, '구분자', 구분자로 구분된 출력할 문자 순번)	SELECT split_part('www.PostgreSQL.or g', '.', 2); split_part ----- PostgreSQL
LPAD	입력 문자열을 자릿수만큼 채울 문자로 왼쪽부터 반환	LPAD ('문자열' 또는 칼럼명, 자리수, '채울문자')	SELECT LPAD('PostgreSQL', 15, '*'); lpad ----- *****PostgreSQL
RPAD	입력 문자열을 자릿수만큼 채울 문자로	RPAD ('문자열' 또는 칼럼명,	SELECT RPAD('PostgreSQL', 15, '*'); rpad

	오른쪽부터 반환	자리수, '채울문자')	----- PostgreSQL*****
LTRIM	입력 문자열을 삭제 문자로 왼쪽부터 삭제	LTRIM ('문자열' 또는 칼럼명, '삭제 문자')	SELECT ltrim('***PostgreSQL***','*'); ltrim ----- PostgreSQL***
RTRIM	입력 문자열을 삭제 문자로 오른쪽부터 삭제	RTRIM ('문자열' 또는 칼럼명, '삭제 문자')	SELECT rtrim('***PostgreSQL***','*'); rtrim ----- ***PostgreSQL
TRIM	입력 문자열을 삭제 문자로 오른쪽, 왼쪽 모두 삭제	TRIM ('문자열' 또는 칼럼명, '삭제 문자')	SELECT trim('***PostgreSQL***','*') btrim ----- PostgreSQL
REPLACE	입력 문자열을 문자1이 있으면 문자2로 변환	REPLACE ('문자열' 또는 칼럼명, '문자1', '문자2')	SELECT replace('Oracle DBMS','Oracle','PostgreSQL') replace ----- PostgreSQL DBMS

2.7.2 숫자형 함수

함수의 파라미터와 리턴 값이 숫자형인 함수입니다.

대표적인 문자형 함수에 대해 소개합니다.

함수명	설명	사용 방법	예시
ROUND	소수점 반올림한 결과 반환	ROUND (숫자, 원하는 자리수)	SELECT round(0.987), round(0.987,0), round(0.987,2), round(0.987,- 2); round round round round -----+-----+-----+----- 1 1 0.99 0
TRUNC	숫자를 원하는 자릿수만큼 잘라 반환	TRUNC (숫자, 원하는 자리수)	SELECT trunc(123.987), trunc(123.987,0), trunc(123.987,2), trunc(123.987,-2); trunc trunc trunc trunc -----+-----+-----+----- 123 123 123.98 100
MOD	숫자2를 숫자1로 나눈 나머지 값 반환	MOD (숫자1, 숫자2)	SELECT mod(123,4); mod ----- 3
CEIL /	숫자와 같거나 가장 큰	CEIL / FLOOR	SELECT

FLOOR	정수 / 같거나 가장 작은 정수 반환	(숫자)	<code>CEIL(12.345),FLOOR(12.345); ceil floor -----+----- 13 12</code>
POWER	숫자1의 숫자2의 제곱값 반환	POWER (숫자1, 숫자2)	<code>SELECT power(1,2), power(2,2); power power -----+----- 1 4</code>
SQRT	숫자의 제곱근 값 반환	SQRT (숫자)	<code>SELECT sqrt(1), sqrt(2); sqrt sqrt -----+----- 1 1.4142135623731</code>
ABS	숫자의 절대값 반환	ABS (숫자)	<code>SELECT ABS(123),ABS(-123); abs abs -----+----- 123 123</code>
SIGN	숫자의 양/음수 여부 판단. 양수일 경우 1을 음수일 경우 -1 그리고 0일 경우 0을 반환	SIGN (숫자)	<code>SELECT SIGN(123),SIGN(-123),SIGN(0); sign sign sign -----+-----+----- 1 -1 0</code>
GENERATE_SERIES	연속되는 숫자 발생하며 FROM 절, SELECT LIST에 모두 사용 가능	GENERATE_SERIES (시작숫자,마지막숫자,간격)	<code>SELECT generate_series(1,3) as level; level ----- 1 2 3 SELECT current_date + s AS dates FROM generate_series(0,14,7) AS s; dates ----- 2014-05-31 2014-06-07 2014-06-14</code>

2.7.3 날짜형 함수

현재 날짜를 반환해 주는 함수입니다.

구분	함수명	예시
DATE 반환	CURRENT_DATE	2012-07-26
	timeofday	Thu Jul 26 15:38:17.218000 2012 KST (text)
TIME	CURRENT_TIME	15:38:17.218+09
TIMESTAMP 반환 함수	CURRENT_TIMESTAMP	2012-07-26 15:38:17.218+09
	now()	2012-07-26 15:38:17.218+09
	transaction_timestamp()	2012-07-26 15:38:17.218+09
	statement_timestamp()	2012-07-26 15:38:17.218+09

	clock_timestamp()	2012-07-26 15:38:17.228+09
	LOCALTIMESTAMP	2012-07-26 15:38:17.218

오라클에서 자주 사용하는 sysdate는 시스템 일자를 반환하지만, PostgreSQL은 sysdate가 없으므로 current_date를 사용하면 됩니다. EDB PAS은 sysdate 지원을 지원합니다. PostgreSQL은 트랜잭션 내에서 트랜잭션 시작 시간으로 동일하게 처리되므로 clock_timestamp 함수를 이용하여 트랜잭션 내의 시간을 다르게 처리 가능합니다.

함수명	설명	사용 방법	예시
ADD_MONTHS	임의의 날짜에 개월 수를 더한 뒤 그 결과를 반환	PostgreSQL 미지원	<pre>SELECT TO_DATE('2014-12-31', 'YYYY-MM-DD') + interval '1 month'; ?column? -----+ 2015-01-31 00:00:00</pre>
DATE_TRUNC	날짜를 자리에 맞게 잘라내고 반환	DATE_TRUNC (자리, 날짜)	<pre>SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40'); date_trunc -----+ 2001-02-16 20:00:00</pre>
EXTRACT	특정한 날짜 유형을 추출하여 반환	EXTRACT (날자유형 FROM 날짜)	<pre>SELECT EXTRACT(DAY FROM TIMESTAMP '1984-09-25 12:00:00'); date_part -----+ 25</pre>
AGE	두 날짜의 차이 출력	AGE (날짜, 날짜)	<pre>SELECT age(timestamp '2001- 04-10', timestamp '1957-06- 13'); age -----+ 43 years 9 mons 27 days</pre>
JUSTIFY_DAYS / JUSTIFY_HOURS /JUSTIFY_INTERVAL	입력된 날짜형을 30일 기준으로 / 24기준으로 / 간격으로 반환	JUSTIFY_DAYS(간격) / JUSTIFY_HOUR S(간격) / JUSTIFY_INTER VAL(간격)	<pre>SELECT justify_days(interval '35 days'), justify_hours(interval '27 hours'), justify_interval(interval '1 mon -1 hour'); justify_days justify_hours justify_interval -----+-----+ 1 mon 5 days 1 day 03:00:00 29 days 23:00:00</pre>
PG_SLEEP	서버 프로세스의 실행을 지연	PG_SLEEP (초)	

2.7.4 Null 관련 함수

PostgreSQL은 Null과 empty string을 엄격히 구분하므로 주의해야 합니다. 대표적인 Null 관련 함수를 소개합니다.

○ COALESCE

인자 중에서 NULL이 아닌 첫 번째 인자를 반환합니다. 만일 모든 인자가 NULL이라면 NULL을 반환합니다. 이 함수는 추출한 데이터를 표시할 때 NULL 값을 기본 값으로 치환하기 위해 주로 사용되는 함수입니다.

[COALESCE 함수 사용 문법]

```
COALESCE ('문자' 또는 칼럼, '문자' 또는 칼럼, ... )
```

[예시] COALESCE 함수 사용

```
SELECT emp_id 사번, first_name || ' ' || last_name 성명, salary 월급, salary *  
commission_pct 커미션  
FROM emp  
WHERE salary * COALESCE (commission_pct,0) < 1000 LIMIT 2;
```

○ NULLIF

NULLIF함수는 값1과 값2가 동일할 경우 NULL을 반환하며 그 외에는 값1을 반환합니다. 값2에 NULL 설정 시 값1이 NULL이 아닐 경우 값1 반환하고 NULL일 경우 NULL 반환합니다.

[NULLIF 함수 사용 문법]

```
NULLIF ((‘문자’ 또는 칼럼, ‘문자’ 또는 칼럼)
```

[예시] NULLIF 함수 사용

```
SELECT e.first_name || ' ' || e.last_name 이름, NULLIF(e.job_id, j.job_id)  
구직업아이디  
FROM emp e, job j  
WHERE e.emp_id = j.emp_id LIMIT 2;
```

○ LNNVL

LNNVL 함수는 조건을 체크하여 조건 결과 값이 FALSE나 UNKNOWN일 경우 TRUE를, 결과가 TRUE이면 FALSE를 반환하는 Oracle 함수입니다. PostgreSQL은 이 함수를 제공하지 않으며 아래와 같이 비슷하게 구현 가능합니다.

[LNNVL 대체 함수 사용 문법]

```
COALESCE (Not 표현식, True)
```

[예시] LNNVL 대체 함수 사용

```
SELECT emp_id 사원번호, first_name || ' ' || last_name 성명, salary 월급,
       salary * commission_pct 커미션
  FROM   emp
 WHERE COALESCE (NOT salary * commission_pct >= 1000, True) LIMIT 2;
```

2.7.5 변환 함수

프로그래밍을 하다 보면 여러 가지 데이터 타입들을 서로 형 변환하여 사용해야 하는 경우 발생합니다. 여느 언어들이 이러한 형 변환을 지원하듯이 PostgreSQL도 데이터 형을 변경하는 함수 제공합니다.

데이터 타입이 결합되는 문자형에 따라 묵시적으로 해당 데이터 형 변환하는 자동 형 변환을 제공합니다. PostgreSQL은 오라클보다 자동 형 변환이 엄격한 편입니다. 예를 들면, int 데이터형과 '1.1'을 연산할 경우 문자로 변환할지, numeric으로 변환할지 명확하지 않기 때문에 에러가 발생합니다. 그러므로 데이터 타입을 명시적으로 변환을 해야 합니다.

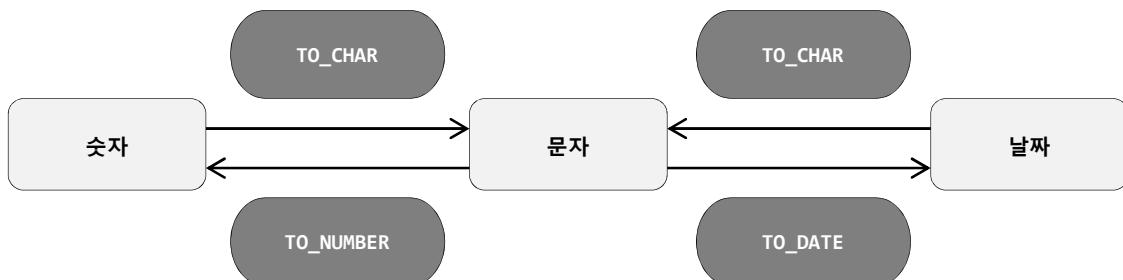


그림 2-2 형 변환 함수

함수명	설명	사용 방법	예시
TO_CHAR	숫자, 날짜를 문자형으로 변환	TO_CHAR (‘문자’ 혹은 칼럼, 표현식)	SELECT to_char(current_date, 'ddth'); ----- 29th
TO_NUMBER	문자를 숫자로 변환	TO_NUMBER (‘문자열’ 또는 칼럼명, ‘숫자 포맷’)	SELECT to_number('1','9') ; to_number ----- 1
TO_DATE / TO_TIMESTAMP	문자를 날짜나 시간으로 변환	TO_DATE / TO_TIMESTAMP P(‘문자열’ 또는 칼럼명, ‘날짜 포맷’)	SELECT to_date('19840925','YYYYMMDD') , to_timestamp('19840925','YYYY MMDD'); to_date to_timestamp -----+----- 1984-09-25 1984-09-25 00:00:00+09
CAST / ::	데이터 형을 변환하고 싶은 데이터형으로 변환	CAST(데이터 AS 원하는 데이터 형) 데이터형:: 원하는 데이터 형	SELECT '1984-09-25', cast('1984-09-25' as date), '1984-09-25'::date; ?column? date date -----+-----+----- 1984-09-25 1984-09-25 1984-09-25

2.7.6 조건식

SQL 호환 조건식 제공합니다. 조건식으로 표현이 어려운 경우는 stored procedure로 구현 가능합니다.

○ CASE

조건에 따라 값 처리합니다. IF/ELSE 구문과 유사합니다. 특정 칼럼 비교나(=만 사용 가능), 조건 비교 처리(다양한 조건 비교 가능)의 두 가지 형식 가능합니다.

[CASE 사용 문법1]

```
CASE 칼럼 WHEN 조건 THEN 처리
[WHEN ...]
[ELSE 처리]
END
```

[CASE 사용 문법2]

```
CASE WHEN 조건 비교 THEN 처리
[WHEN ...]
[ELSE 처리]
END
```

[예시1] CASE 사용

```
SELECT emp_id, last_name,
CASE dept_id WHEN 10 THEN 'Administration'
WHEN 20 THEN 'Marketing'
WHEN 30 THEN 'Purchasing'
END dept
FROM emp WHERE dept_id IN (10,20,30)limit 5;
```

[예시2] CASE 사용

```
SELECT emp_id, last_name,
CASE WHEN salary >= 2000 AND salary < 3000 THEN '2000 band'
WHEN salary >= 3000 AND salary < 4000 THEN '3000 band'
WHEN salary >= 4000 AND salary < 5000 THEN '4000 band'
WHEN salary >= 5000 AND salary < 6000 THEN '5000 band'
ELSE 'etc' END salary_band
FROM emp LIMIT 5;
```

CASE와 유사한 오라클의 DECODE 함수는 PostgreSQL에서 지원하지 않습니다. 단, EDB PAS는 오라클 호환으로 제공합니다.

○ GREATEST & LEAST

GREATEST와 LEAST 함수는 식 내에서 최대값 혹은 최소값 반환합니다. 표현식 내의 리스트에서 NULL값은 무시되며 모든 값이 NULL일 경우에도 결과는 NULL로 출력됩니다.

[GREATEST & LEAST 사용 문법]

```
GREATEST(칼럼 혹은 비교값 리스트) / LEAST (칼럼 혹은 비교값 리스트)
```

[예시] GREATEST & LEAST 사용

```
SELECT GREATEST(1,2,3,4,5,6,7), LEAST(1,2,3,4,5,6,7);

greatest | least
-----+-----
7 | 1
```

2.8 연산자

연산자는 데이터를 조작하여 그 결과를 산출하는 역할을 합니다. 연산자는 일반적인 수식, 문자, 비교, 논리 연산자가 존재하며 규칙에 따라 우선순위 존재합니다. PostgreSQL은 9.5버전 이전과 이후에 연산자의 우선순위가 변경되었습니다. 데이터베이스 버전에 따라 주의가 필요합니다.

종류	연산자	사용 예
문자 연산자		<code>SELECT * FROM emp WHERE employee_id first_name = '100Steven';</code>
수식 연산자	+, -, *, /, %	<code>SELECT * FROM emp WHERE salary * 12 = 57600;</code>
비교 연산자	>, <, =, >=, <=	<code>SELECT * FROM emp WHERE salary * 12 > 57600;</code>

2.8.1 논리 연산자

논리 값에 사용되는 연산자로서 결과값을 TRUE나 FALSE로 반환합니다. AND, OR, NOT 와 같은 논리연산자를 사용 가능합니다.

A	B	A AND B	A OR B
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

A	NOT A
TRUE	FALSE
FALSE	TRUE
NULL	NULL

2.8.2 날짜 연산자

날짜형 연산자는 수식 연산자(+, -, *, /) 가능합니다. 날짜형 연산자의 반환되는 값의 데이터 형의 주의가 필요합니다.

기존 오라클이나 MySQL 처럼 날짜 데이터 타입에 1시간 후, 1일 후 등과 같은 연산을 할 때는 interval로 형 변환을 하여 사용해야 합니다.

[예시] 날짜 연산자 사용

```
SELECT '2013-09-25'::date + interval '12 month';
?column?
-----
2014-09-25 00:00:00
```

구분	결과	사용 예시
Date vs Date	integer	SELECT pg_typeof(CURRENT_DATE - date '1984-09-25'); pg_typeof ----- integer
Date +- integer	date	SELECT pg_typeof(CURRENT_DATE - 1); pg_typeof ----- date
Date +- interval	timestamp	SELECT pg_typeof(CURRENT_DATE - interval '1 hour'); pg_typeof ----- timestamp without time zone
Date +- numeric	연산 안됨	SELECT pg_typeof(CURRENT_DATE - 1.1); ERROR: operator does not exist: date - numeric
Timestamp vs timestamp	interval	SELECT pg_typeof(timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'); pg_typeof ----- interval
Timestamp +- interval	timestamp	SELECT pg_typeof(timestamp '2001-09-29 03:00' - interval '1 hour'); pg_typeof ----- timestamp without time zone
Timestamp +- integer	연산 안됨	SELECT pg_typeof(timestamp '2001-09-29 03:00' - 1); ERROR: operator does not exist: timestamp without time zone - integer
Timestamp +- numeric	연산 안됨	SELECT pg_typeof(timestamp '2001-09-29 03:00' - 1.1); ERROR: operator does not exist: timestamp without time zone - numeric

2.8.3 범위 조건

○ BETWEEN.. AND

어느 범위에 걸쳐 있는지 검사하는 경우 BETWEEN.. AND와 비교 연산자를 사용 가능합니다.

[예시] BETWEEN.. AND 사용

```
SELECT first_name || ' ' || last_name 성명, salary 월급, dept_id 부서번호
FROM emp
WHERE salary BETWEEN 15000 AND 20000;
```

○ IN과 EXISTS

IN과 EXISTS은 여러 값을 동시에 비교할 때 사용합니다. IN은 OR 연산자와 동일한 기능을 수행하며 IN하면 OR을 사용하는 것보다 문장이 간결해지고 가독성이 높아집니다.

[예시] IN과 OR 사용

```
SELECT first_name || ' ' || last_name 성명, salary 월급, department_id
부서번호
FROM emp
WHERE salary IN (10000, 20000);

SELECT first_name || ' ' || last_name 성명, salary 월급, department_id
부서번호
FROM emp
WHERE salary = 10000 OR salary = 20000;
```

IN 연산자는 NOT과 함께 사용하여 조건에 포함되지 않게 비교하려고 할 때 사용합니다.

[예시] IN과 NOT 사용

```
SELECT first_name || ' ' || last_name 성명, salary 월급, department_id
부서번호
FROM emp
WHERE salary NOT IN (10000, 20000) LIMIT 3;
```

EXISTS은 특정 칼럼값이 존재하는지 여부를 체크하며 서브쿼리에서만 사용 가능합니다. EXISTS는 IN에 비하여 성능이 유리할 때가 있으며, NULL 비교도 가능합니다. EXISTS의 반대는 NOT EXISTS입니다.

[예시] EXISTS 사용

```
SELECT e.emp_id 사원번호, e.first_name || ' ' || last_name 성명, e.salary 월급,
e.dept_id 부서코드
FROM emp e
WHERE EXISTS (SELECT *
FROM dept d
WHERE d.dept_id IN (30,60,90) AND e.dept_id = d.dept_id) LIMIT 10;
```

○ LIKE

테이블의 칼럼 값들 중에 특정 패턴에 속하는 값을 조회할 때 사용합니다.

[LIKE 사용 문법]

검색 값 LIKE 패턴 표현식 (예. %, _)

%는 길이에 상관없이 모든 문자를 대체하며 "_"는 한 문자에서만 대체하여 검색합니다. LIKE 역시 IN이나 BETWEEN과 같이 NOT과 함께 사용 가능합니다. 대소문자를 구별하지 않으려면 LIKE 대신 ILIKE를 사용 가능합니다. LIKE를 간단히 ~~연산자로 사용할 수 있으며 ILIKE는 "~~*"연산자로 사용 가능하며 NOT LIKE 및 NOT ILIKE는 "! ~~" 및 "! ~~*" 연산자 사용 가능합니다.

[예시] LIKE 사용

SELECT first_name ' ' last_name 성명, salary 월급, dept_id 부서번호, phone_number 전화번호 FROM emp WHERE phone_number NOT LIKE '%515%' LIMIT 3;

2.8.4 Null 처리

NULL 값이 있는 테이블의 칼럼 값을 검색하고자 할 때는 IS NULL 구문 사용 가능합니다. IS NULL 또한 NOT과 함께 쓰여 IS NOT NULL과 구문으로 사용 가능합니다.

NULL 검색 시 '= null' 형식으로 사용 불가합니다. NULL 값은 비교연산자를 사용할 수 없으므로 결과 값을 알 수 없는 UNKNOWN을 반환합니다. 앞서 언급했듯이, NULL과 "" (empty string)은 다른 값임을 주의해야 합니다.

[예시1] NULL 사용

```
SELECT first_name || ' ' || last_name 성명, salary 월급, department_id 부서번호,  
phone_number 전화번호  
FROM employees  
WHERE manager_id IS NULL;
```

[예시2] NULL 사용

```
postgres=# SELECT 'PostgreSQL' || null;  
?column?  
-----
```

2.8.5 집합 연산자

집합(SET) 연산자는 여러 쿼리의 질의 결과에 대해 의도된 집합을 찾는 역할을 합니다.

○ UNION

UNION은 둘 이상의 질의 결과를 하나의 결과로 출력하는 합집합의 역할을 합니다.
UNION은 ALL과 함께 쓰여 모든 요소를 중복되게 출력 가능합니다.

[예시] UNION

```
-- 샘플 테이블과 데이터 생성  
  
CREATE TABLE song_club (student_name text, grade numeric);  
CREATE TABLE dance_club (student_name text, grade numeric);  
INSERT INTO song_club VALUES ('Steven', 1);  
INSERT INTO song_club VALUES ('David', 1);  
INSERT INTO song_club VALUES ('John', 1);  
INSERT INTO dance_club VALUES ('Ismael', 2);  
INSERT INTO dance_club VALUES ('Steven', 1);  
  
-- 노래 동아리에 가입된 학생 목록 출력  
SELECT student_name FROM song_club;  
student_name  
-----  
Steven  
David  
John
```

```
-- 춤과 동아리에 가입된 학생 목록을 중복되게 출력
SELECT student_name FROM song_club
UNION ALL
SELECT student_name FROM dance_club ;
student_name
-----
Steven
David
John
Ismael
Steven
```

○ INTERSECT

INTERSECT은 여러 테이블에 공통으로 포함된 요소를 출력하는 교집합의 역할을 합니다. INTERSECT 결과는 조인과 같지만 데이터의 양이 많을 때는 INTERSECT보다 조인을 사용하는 것이 성능에 좋습니다. INTERSECT 또한, ALL과 함께 쓰여 중복되는 데이터 출력 가능합니다.

[예시] INTERSECT

```
SELECT student_name FROM song_club
INTERSECT
SELECT student_name FROM dance_club;
student_name
-----
Steven
```

○ EXCEPT

EXCEPT는 오라클의 MINUS 연산자와 동일한 기능을 합니다. 즉, 한쪽 테이블에 포함된 요소를 출력하는 차집합의 역할을 합니다. 따라서 어느 집합이 먼저 오는지에 따라 결과 값이 다릅니다. EXCEPT 역시 ALL과 함께 쓰여 중복되는 행을 출력 가능합니다.

[예시] EXCEPT

```
SELECT student_name FROM song_club
EXCEPT
SELECT student_name FROM dance_club;
student_name
-----
```

```
David  
John
```

○ 집합 연산자와 NULL

집합(SET) 연산자를 사용할 때 칼럼의 일부가 NULL값이 있으면 NULL 값이 포함되어 결과가 추출합니다. NULL은 특정 유형의 데이터 타입이 아니므로 일부 칼럼이 NULL일 경우 더 이상 비교를 수행하지 않기 때문입니다. 즉 집합 연산자를 사용할 때는 NULL은 신경 쓰지 않아도 됩니다.

[예시] 집합 연산자의 숫자 NULL 출력

```
SELECT 1 칼럼1, 'FIRST' 칼럼2  
UNION  
SELECT NULL 칼럼1, 'SECOND' 칼럼2;  
  
칼럼1 | 칼럼2  
-----+-----  
1 | FIRST  
| SECOND
```

집합 연산자를 사용할 때 몇 가지 주의해야 합니다.

- 집합 연산자를 사용할 때는 SELECT 문장의 칼럼 개수가 동일해야 함
- 비교하려는 데이터들이 서로 호환이 되는 데이터형이어야 함(예 text-character varying)
- SET 연산자를 사용할 때 ORDER BY절은 맨 마지막에 한 번만 사용 가능. 그러나 오라클과 달리 서브쿼리 내부에 ORDER BY절을 사용 가능

[예시1] 잘못된 UNION과 ORDER BY 사용

```
SELECT e.emp_id, e.first_name, e.last_name  
FROM emp e, dept d, locations l  
WHERE e.dept_id = d.dept_id AND d.location_id = l.location_id AND l.country_id =  
'DE' ORDER BY 1  
UNION  
SELECT customer_id, cust_first_name, cust_last_name  
FROM customers  
WHERE nls_territory = 'GERMANY'
```

[예시2] 올바른 UNION과 ORDER BY 사용

```

SELECT e.emp_id, e.first_name, e.last_name
FROM emp e, dept d, locations l
WHERE e.dept_id = d.dept_id AND d.location_id = l.location_id AND l.country_id =
'DE'
UNION
SELECT customer_id, cust_first_name, cust_last_name
FROM customers
WHERE nls_territory = 'GERMANY'
ORDER BY 1

```

3장. 실전 쿼리 작성

3.1 JOIN

관계형 데이터베이스의 테이블은 중복 데이터를 최소화하고 데이터의 무결성을 보장되도록 설계해야 합니다. 사용자에게 의미 있는 데이터를 제공하기 위해서 연관성 있는 데이터를 연결하거나 조합하는 일련의 작업들을 조인이라고 합니다. 서로 다른 테이블간의 결합 및 자기 자신과의 테이블을 결합을 위해 조인을 사용합니다.

```
SELECT empno,ename,deptno FROM emp limit 5;
```

empno	ename	deptno
7369	SMITH	20
7499	ALLEN	30
7521	WARD	30
7566	JONES	20
7654	MARTIN	30

위의 예시에서 emp 테이블을 조회한 결과. 데이터를 살펴보면 deptno가 숫자로 작성되어 있어 사원의 부서명을 알기 힘듭니다. 사원들의 부서명을 확인하기 위해서는 부서명이 들어 있는 dept 테이블의 데이터들을 확인해야 합니다. 여기서 dept 테이블에 존재하는 부서명과 emp 테이블에 존재하는 부서 아이디를 결합하여 나타내면 데이터를 식별하기 쉽습니다. 이러한 기능을 수행하기 위해 조인을 사용 가능합니다. 조인 기능을 이용하면 서로 연관된 데이터간 결합하여 조회 가능합니다.

조인 대상의 테이블들에 양쪽에 데이터가 모두 있을 경우에만 출력되는 조인 형태를 내부조인 (Inner Join) 이라 합니다. Join에는 결합 조인(Cross Join), 왼쪽 외부 조인(Left Outer Join), 오른쪽

외부 조인(Right Outer Join), 완전 외부 조인(Full Outer Join), 셀프 조인(Self Join) 등 다양한 종류 존재합니다.

○ 결합 조인(Cross Join)

결합 조인은 두 테이블의 모든 레코드를 조합한 결과를 산출합니다. 즉, 테이블 간의 조합 가능한 모든 경우의 수를 계산하여 결과를 산출합니다.

[예시] 결합 조인

```
SELECT empno,ename, dname FROM emp CROSS JOIN dept;
```

empno	ename	dname
7369	SMITH	ACCOUNTING
7369	SMITH	RESEARCH
7369	SMITH	SALES
7369	SMITH	OPERATIONS
7499	ALLEN	ACCOUNTING
...		

위에 쿼리문을 실행하면 총 56개의 레코드가 출력됩니다. 이것은 emp의 14개의 레코드와 dept의 4개의 레코드의 모든 조합 수입니다. 즉 emp 테이블의 첫 번째 레코드와 dept 테이블의 첫번째 레코드를 결합하고 다시, emp 테이블의 첫 번째 레코드와 emp 테이블의 두 번째 레코드를 결합 하는 형식으로 계속하여 조합 가능한 모든 경우의 수를 구하여 56(14×4)건이 조회되었습니다.

○ 왼쪽 외부 조인(Left Outer Join)

왼쪽 외부 조인(Left Outer Join)은 내부 결합을 수행하고, emp 테이블의 레코드와 결합 조건이 맞지 않는 dept의 각 레코드에 대해서는, emp의 레코드를 NULL로 치환하여 결합합니다. 따라서 연결된 테이블은 무조건 emp의 레코드가 각각 적어도 1개 이상 존재합니다.

조인되는 조건은 WHERE 대신 ON을 이용하여 조건 명시외부 조인에서는 일치하는 값이 없는 경우에도 NULL 출력합니다.

[예시] 왼쪽 외부 조인

```

SELECT empno
      ,ename
      ,dname
FROM emp e
LEFT JOIN dept d ON e.deptno = d.deptno;

ORDER BY e.deptno empno | ename | dname
-----+-----+-----
 7934 | MILLER | ACCOUNTING
 7782 | CLARK  | ACCOUNTING
 7839 | KING   | ACCOUNTING
 7788 | SCOTT  | RESEARCH
 7566 | JONES  | RESEARCH
 7369 | SMITH  | RESEARCH
 7876 | ADAMS  | RESEARCH
 7902 | FORD   | RESEARCH
 7521 | WARD   | SALES
 7844 | TURNER | SALES
 7499 | ALLEN  | SALES
 7698 | BLAKE  | SALES
 7654 | MARTIN | SALES
 7900 | JAMES  | SALES
(14 rows)

```

○ 오른쪽 외부 조인(Right Outer Join)

오른쪽 외부 조인은 왼쪽 외부 조인과 반대로 dept의 레코드가 각각 적어도 1개 이상 존재하면 출력됩니다. 오른쪽 내부 조인과 왼쪽 내부조인은 쉽게 말해 모든 행이 출력 되는 테이블의 위치가 어디 있는지에 따라 구분됩니다.

[예시] 오른쪽 외부 조인

```

SELECT empno
      ,ename
      ,dname
FROM emp e
RIGHT JOIN dept d ON e.deptno = d.deptno
ORDER BY e.deptno

empno | ename | dname
-----+-----+-----
 7934 | MILLER | ACCOUNTING

```

7782	CLARK	ACCOUNTING
7839	KING	ACCOUNTING
7369	SMITH	RESEARCH
7566	JONES	RESEARCH
7788	SCOTT	RESEARCH
7876	ADAMS	RESEARCH
7902	FORD	RESEARCH
7521	WARD	SALES
7844	TURNER	SALES
7499	ALLEN	SALES
7900	JAMES	SALES
7698	BLAKE	SALES
7654	MARTIN	SALES
		OPERATIONS
(15 rows)		

출력 결과를 살펴 보면 인원이 할당 되지 않은 부서의 이름까지 출력되는 것을 확인 가능합니다 내부조인이 왼쪽 테이블(emp) 값을 기준으로 결합을 하고 오른쪽 외부 조인은 오른쪽 테이블(dept) 기준으로 결합했기 때문입니다.

○ 완전 외부 조인(Full Outer Join)

완전 외부 조인은 오른쪽 외부 조인과 왼쪽 내부 조인을 합쳐 놓은 형태로 두 테이블의 모든 값을 출력됩니다.

[예시] 완전 외부 조인

empno	ename	dname
7934	MILLER	ACCOUNTING
7782	CLARK	ACCOUNTING
7839	KING	ACCOUNTING
7369	SMITH	RESEARCH
7566	JONES	RESEARCH
7788	SCOTT	RESEARCH

7876	ADAMS	RESEARCH
7902	FORD	RESEARCH
7521	WARD	SALES
7844	TURNER	SALES
7499	ALLEN	SALES
7900	JAMES	SALES
7698	BLAKE	SALES
7654	MARTIN	SALES
		OPERATIONS
(15 rows)		

위의 결과를 보면 부서명이 없는 레코드와 할당되지 않은 부서명이 모두 출력되는 것을 확인 가능합니다. 즉 완전 외부 조인은 왼쪽 내부 조인과 오른쪽 내부 조인을 결합한 형태입니다.

○ 셀프 조인(Self Join)

셀프 조인(Self-join)은 자기 자신과 조인을 맺는 것입니다. 즉 하나의 테이블 안에서 조인이 발생하는 것입니다.

[예시] 셀프 조인

```
SELECT e.empno
      ,e.ename
      ,m.empno mgrno
      ,m.ename mgrname
FROM emp e
      ,emp m
WHERE e.empno = m.mgr
```

empno	ename	mgrno	mgrname
7902	FORD	7369	SMITH
7698	BLAKE	7499	ALLEN
7698	BLAKE	7521	WARD
7839	KING	7566	JONES
7698	BLAKE	7654	MARTIN
7839	KING	7698	BLAKE
7839	KING	7782	CLARK
7566	JONES	7788	SCOTT
7698	BLAKE	7844	TURNER
7788	SCOTT	7876	ADAMS
7698	BLAKE	7900	JAMES

7566 JONES 7902 FORD
7782 CLARK 7934 MILLER
(13 rows)

위의 쿼리문은 empno의 칼럼과 mgrno칼럼을 비교하여 해당 사원의 매니저가 누구인지를 나타냅니다. 셀프 조인은 위의 쿼리문과 같이 자기 자신의 테이블을 결합하여 사용하는 조인입니다. 셀프 조인을 사용하기 위해서는 반드시 테이블 별칭 (alias)를 사용해야 합니다.

3.2 그룹 쿼리

데이터베이스에 저장되어 있는 데이터를 원하는 목적에 맞게 가공하여 ‘의미 있는 데이터’ 활용할 때 평균, 총합, 차액 등과 같이 집계 데이터로 가공해야 할 필요가 있습니다. 즉, 특정 집단으로 분류를 세분화하거나 여러 집단을 합해서 데이터를 산출합니다. 이러한 목적으로 원본데이터를 가공해서 사용하는 문장을 그룹쿼리라고 합니다.

[예시] 그룹쿼리

SELECT deptno, ROUND(SUM(sal)), COUNT(sal), ROUND(AVG(sal))
FROM emp
GROUP BY deptno;
deptno round count round
-----+-----+-----+-----
20 10875 5 2175
30 9400 6 1567
10 8750 3 2917

3.2.1 DISTINCT와 ALL

○ DISTINCT

SELECT 리스트에서 칼럼 앞에 쓰여 해당 칼럼값 중 유일한 값(중복을 제거한 결과) 출력합니다.

[예시1] DISTINCT

SELECT deptno FROM emp;

deptno

```

20
30
30
20
30
30
10
20
10
30
20
30
20
10
(14 rows)

```

[예시2] DISTINCT

```

SELECT DISTINCT(deptno) FROM emp

deptno
-----
20
30
10
(3 rows)

```

[예제1]에서는 중복값을 포함되었기 때문에 14건의 데이터가 조회되었습니다. DISTINCT를 사용하였을 경우 중복값을 제외하여 3건이 조회되었습니다. DISTINCT는 NULL을 한번만 포함하여 출력합니다. DISTINCT에 두 칼럼 이상을 명시할 경우 칼럼의 조합으로 유일한 값을 반환합니다.

[예시3] DISTINCT

```

SELECT DISTINCT mgr, empno FROM emp

mgr | empno
-----+-----
7698 | 7521
7839 | 7782
7902 | 7369
7698 | 7844

```

7839		7698
7782		7934
7566		7788
7698		7499
7839		7566
7698		7654
		7839
7698		7900
7566		7902
7788		7876
(14 rows)		

DISTINCT ON 키워드를 사용해서 특정 칼럼에 대한 유일한 값 출력 가능합니다.

[예시3] DISTINCT ON

SELECT DISTINCT ON (mgr) mgr, empno FROM emp;		
mgr		empno
-----+-----		
7566		7902
7698		7521
7782		7934
7788		7876
7839		7782
7902		7369
		7839
(7 rows)		

○ ALL

ALL은 중복여부에 대해서 상관없이 전체 데이터를 조회 가능합니다. SELECT 절의 Default 값은 ALL이며 따로 명시하지 않아도 ALL 속성이 적용됩니다.

3.2.2 집계 함수

집계 함수는 입력 값으로부터 단일 결과를 계산 하는 기능을 합니다.

○ COUNT 함수

칼럼의 NULL을 제외한 행의 수를 출력합니다.

[예시1] COUNT 함수

```
SELECT count(*) FROM emp;
count
-----
14
(1 row)
```

[예시2] COUNT 함수

```
SELECT count(mgr) FROM emp;
count
-----
13
(1 row)
```

○ MAX / MIN / SUM / AVG 함수

칼럼의 최대값 / 최소값 / 칼럼 합계 / 평균 출력합니다.

[예시] MAX / MIN / SUM / AVG 함수

```
SELECT round(max(sal))
      ,round(min(sal))
      ,round(sum(sal)) AS sum
      ,round(avg(sal)) AS avg
FROM emp;

round | round | sum | avg
-----+-----+-----+-----
  5000 |    800 | 29025 |  2073
(1 row)
```

3.2.3 GROUP BY와 ORDER BY, HAVING

집계 함수에 의한 집계 데이터들을 더욱 의미 있게 나타내기 위해서는 특정 범위별로 데이터를 추출할 필요가 있습니다. 예를 들어 emp 테이블에서 월급에 대한 집계 데이터를 추출한다고 할 때, 부서별 또는 직급별 등의 그룹별 데이터를 추출할 수가 있습니다. 이렇게 특정 범위 별 집계 데이터를 추출 하는 것을 그룹핑이라고 합니다.

○ GROUP

GROUP BY를 이용하면 그룹핑이 가능합니다. GROUP BY 절의 칼럼이 나열된 순서는 상관 없으며 그룹을 대표하는 하나의 공통 값으로 묶어 주는 역할을 합니다.

[GROUP BY 사용 문법]

```
SELECT select 리스트
FROM 테이블, 뷔
WHERE 조건
GROUP BY 그룹핑할 칼럼 리스트
```

[예시1] GROUP BY 사용

```
SELECT deptno FROM emp GROUP BY deptno;

deptno
-----
20
30
10
(3 rows)
```

[예시2] GROUP BY 사용

```
SELECT deptno, round(sum(sal)) FROM emp GROUP BY deptno;

deptno | round
-----+-----
20    | 10875
30    |  9400
10    |  8750
(3 rows)
```

[예시3] GROUP BY 사용

```
SELECT deptno, job, round(sum(sal)) FROM emp GROUP BY deptno;

ERROR: column "emp.job" must appear in the GROUP BY clause or be used in an
aggregate function
LINE 2: SELECT deptno, job, round(sum(sal)) FROM emp group by deptno...
^
```

```
***** Error *****
```

```
ERROR: column "emp.job" must appear in the GROUP BY clause or be used in an
aggregate function
SQL state: 42803
Character: 17
```

[예시1]는 GROUP BY 절에 deptno를 명시하여 부서번호 별로 그룹핑 한 것입니다. 단순 GROUP BY 절만을 사용하기 보다는 일반적으로 집계 함수를 함께 사용하여야 더 의미 있는 데이터를 추출 가능합니다.

[예시2]는 부서별 급여의 합계 출력. 급여의 합계를 SUM함수로 나타내고, GROUP BY를 이용하여 부서별로 그룹핑 이 결과를 바탕으로 각 부서의 총 급여액을 비교할 수 있습니다.

[예시3]에서 볼 수 있듯이, SELECT 문장에서 GROUP BY절을 사용할 경우 SELECT 리스트에 있는 항목 중에서 집계 함수를 제외한 모든 항목을 GROUP BY 절에 명시해 주어야 합니다. 항목 누락 시 “ERROR: column “employees.job_id” must appear in the GROUP BY clause or be used in an aggregate function” 에러를 출력됩니다.

○ ORDER BY

이제까지 GROUP BY 절을 사용하여 데이터를 그룹핑 하였습니다. 하지만 해당 쿼리 결과들이 한 눈에 일목요연하게 보이지가 않는다는 문제점 있습니다. 즉, 부서별, 직급별 등으로 그룹핑된 결과를 정렬할 필요 있습니다. 이 때 ORDER BY절을 사용하여 가능합니다. ORDER BY절은 쿼리문에서 추출된 결과를 정렬하는 데 사용합니다. 특히 보고서, 통계 자료 등 데이터 정렬이 중요한 경우에 많이 사용 됩니다. ORDER BY 절에 정렬하고자 하는 기준이 되는 칼럼명을 명시하면 됩니다.

[예시] ORDER BY 사용

```
SELECT deptno, round(sum(sal)) FROM emp GROUP BY deptno ORDER BY deptno;
```

deptno	round
10	8750
20	10875
30	9400
(3 rows)	

○ HAVING

HAVING 절은 GROUP BY절과 함께 사용되며, SELECT 문장에서 집계 함수를 사용한 조건을 명시하는 목적으로 사용됩니다. 즉, 일반 조건은 WHERE절에 명시 하지만, 조건에 집계 함수의 결과가 필요한 경우에는 HAVING절에 명시 해야 합니다.

예를 들어, 부서의 부서원이 5명 이하인 부서만 조회한다고 할 때 그룹 쿼리에 WHERE 절을 사용하면 아래와 같이 에러 발생합니다.

[예시1] 그룹쿼리의 WHERE 사용

```
SELECT deptno, count(*) FROM emp WHERE count(*) <=5 GROUP BY deptno

ERROR: aggregate functions are not allowed in WHERE
LINE 1: SELECT deptno, count(*) FROM emp WHERE count(*) <=5 GROUP BY...
^
***** Error *****

ERROR: aggregate functions are not allowed in WHERE
SQL state: 42803
Character: 40
```

에러의 내용을 보면 WHERE절에서는 집계 함수를 사용할 수 없다는 내용이므로 HAVING절을 이용 조건을 명시해야 합니다.

[예시2] HAVING 사용

```
SELECT deptno, count(*) FROM emp GROUP BY deptno HAVING count(*) <=5

deptno | count
-----+-----
 20 |    5
 10 |    3
(2 rows)
```

다시 요약하면 HAVING 절은 집계 함수를 위한 조건절이라 할 수 있으며, 항상 GROUP BY 절과 같이 사용 가능합니다.

3.3 서브쿼리

서브 쿼리는 하나의 SQL 문장 내부에 존재하는 또 다른 SELECT 문장을 말합니다. 서브 쿼리는 메인쿼리와의 관계에 따라 다음과 같이 분류할 수 있습니다.

- 메인쿼리와 연관성 없는 서브쿼리: Noncorrelated Subquery
- 메인쿼리와 연관성 있는 서브쿼리: Correlated Subquery

서브쿼리의 위치에 따라 다음과 같이 분류할 수 있습니다.

- SELECT 절에 위치: 서브쿼리 (Scalar subquery)
- FROM 절에 위치: 인라인 뷰 (Inline View)
- WHERE 절에 위치: 중첩 쿼리 (Nested Query)

○ 메인 쿼리와 연관성이 없는 서브 쿼리

서브쿼리와 메인 쿼리 사이에 데이터의 연관성이 없는 서브쿼리입니다. 즉 서브쿼리의 결과로 반환되는 로우들은 메인 쿼리와는 독립적입니다.

[예시1] 단일 로우/단일 칼럼을 반환하는 서브쿼리

```
SELECT e.ename
      ,j.job
  FROM emp e
      ,jobhist j
 WHERE e.sal = (SELECT MAX(sal)
                  FROM emp)
       AND e.job = j.job;

ename | job
-----+-----
 KING | PRESIDENT
 (1 row)
```

하나의 칼럼과 로우를 반환하는 서브쿼리입니다. 집계함수가 포함된 쿼리가 대부분이며, WHERE 조건절의 비교값 형태로 오는 경우가 많습니다. 위의 예는 월급이 가장 많은 사원 정보를 조회하는 예시입니다. WHERE절 안에 집계함수인 MAX() 함수를 사용한 서브쿼리가 포함되어 있습니다. 단일 로우, 단일 칼럼을 반환하는 유형의 서브쿼리는 WHERE절 이외에도, FORM, SELECT, ORDER BY, HAVING, CONNECT BY, 절에 사용 가능합니다. 단일 로우, 단일 칼럼을 반환하는 서브쿼리는 서브쿼리 중에서 가장 기본적이고 많이 사용되는 유형 중 하나이며 주로 WHERE절에서 사용합니다.

[예시2] 다중 로우/단일 칼럼을 반환하는 서브쿼리

```

SELECT tablename
FROM pg_tables
WHERE schemaname IN (
    SELECT schemaname
    FROM pg_tables
    WHERE schemaname = 'pg_catalog'
);

tablename
-----
pg_statistic
pg_type
pg_authid
...생략

```

[예시2]는 다중 로우, 단일 칼럼을 반환하는 서브쿼리는 일반적인 쿼리 형태를 보입니다. 시스템 카탈로그에 있는 테이블의 목록을 반환하는 예시입니다. 쿼리문을 살펴보면 서브 쿼리와 메인 쿼리 간에 직접적인 연관은 없는 것을 알 수가 있습니다.

다중 로우, 단일 칼럼 서브 쿼리는 여러 로우들을 출력해야 하므로 일반적으로 동등연산자(=)를 사용하지 않고 IN 연산자를 사용합니다. 비교 연산자를 사용하기 위해서는 ANY 연산자와 ALL 연산자를 사용 가능합니다.

[예시3] ANY 연산자를 이용한 서브쿼리

```

SELECT empno
      ,deptno
      ,sal
FROM emp
WHERE sal > ANY (
    SELECT sal
    FROM emp
)
ORDER BY sal

empno | deptno |   sal
-----+-----+
  7900 |     30 | 950.00
  7876 |     20 | 1100.00
      ...

```

7839 10 5000.00
(13 rows)

총 13개의 결과물을 출력되었습니다. 서브 쿼리의 결과는 sal 값이 최소 950에서 최대 5000까지 값이 출력되었습니다. ANY 연산자는 서브쿼리의 결과값들과 조건을 비교해 어느 하나라도 조건에 만족하면 결과값을 출력됩니다. 따라서 950에서 5000사이의 sal 값을 가진 레코드들이 모두 출력됩니다.

[예시4] ALL 연산자를 이용한 서브쿼리

```

SELECT empno
      ,deptno
      ,sal
FROM emp
WHERE sal > ALL (
    SELECT sal
    FROM emp
    WHERE deptno = 30
)
ORDER BY sal

empno | deptno |   sal
-----+-----+-----
 7566 |     20 | 2975.00
 7788 |     20 | 3000.00
 7902 |     20 | 3000.00
 7839 |     10 | 5000.00
(4 rows)

```

ALL 연산자는 서브 쿼리의 결과값들과 조건을 비교해 모든 조건을 만족하는 결과값만 출력됩니다.

[예시5] 다중 칼럼을 반환하는 서브쿼리

```

SELECT deptno
      ,dname
FROM dept
WHERE (deptno
      ,mgr) IN (
    SELECT deptno
          ,empno
    FROM emp
    WHERE mgr IS NULL);

```

서브 쿼리에서 하나 이상의 칼럼을 반환합니다. 위의 쿼리문은 매니저가 없는 사원의 부서 정보를 추출하는 예입니다. 이 쿼리에서는 메인 쿼리의 deptno 와 mgr 값이 동시에 서브 쿼리 반환 값들에 포함되어야 조건을 만족해야 합니다.

○ 메인 쿼리와 연관성이 있는 서브 쿼리

연관성 없는 서브쿼리는 단순히 서브쿼리를 수행하여, 그 결과를 메인 쿼리와 비교하는 형태인 반면 연관성 있는 서브쿼리의 경우 서로 데이터의 참조가 발생하는 쿼리입니다.

[예시] 메인 쿼리와 연관성이 있는 서브 쿼리

```
SELECT count(*)  
FROM emp e  
WHERE EXISTS ( SELECT 1  
                FROM dept d  
                WHERE e.deptno = d.deptno);  
  
count  
-----  
14
```

쿼리문을 살펴보면 서브쿼리의 AND 절에서 메인 쿼리와 서브 쿼리간에 조인이 발생하는 것을 확인 가능합니다. 이와 같이 메인 쿼리와 서브쿼리가 독립적으로 실행되지 않고 서로 연관성 있다 하여 연관성 있는 쿼리입니다.

PART 5 Stored procedure language

1장. PL/pgSQL

1.1 개요

PL/pgSQL은 PostgreSQL에서 제공하는 프로그래밍 언어로 SQL과 다르게 순차적 처리가 가능합니다.

- 함수 및 트리거 생성 가능
- SQL 언어 제어 구조 추가
- 복잡한 계산 수행
- 서버에 의해 신뢰받을 수 있는 정의
- 사용 용이

○ PL/pgSQL 구조

PL/pgSQL는 선언부, 실행부, 예외처리부 구성됩니다. 선언부는 DECLARE로 선언되며, 실행부에서 사용할 변수나 상수를 선언. 변수나 상수는 반드시 DECLARE 블록 안에서 선언되어야 합니다. 실행부는 IF문, LOOP문, FOR문 등 실제 처리할 로직을 작성합니다. SQL문에서는 한번에 한 문장만 실행할 수 있었지만 PL/pgSQL에서는 블록 단위로 한번에 여러 문장을 순차적으로 처리 가능 합니다. 예외 처리부는 실행부에서 발생한 오류를 처리하는 것으로 자바나 C#에서 사용하는 try-catch 구문과 동일한 역할을 합니다.

[PL/pgSQL 사용 문법]

```
DECLARE -- 선언부
BEGIN
    statements -- 실행부
EXCEPTION -- 예외처리부
END;
```

[예시] PL/pgSQL 사용

```
CREATE FUNCTION pgsql_example() RETURNS integer AS $$  
DECLARE counter integer;  
BEGIN  
    counter := 10;  
    counter := counter / 0;
```

```

RETURN counter;

EXCEPTION WHEN division_by_zero THEN
    RAISE NOTICE 'DIV 0';
    counter := counter / 1;
    RETURN counter;
END;
$$ LANGUAGE plpgsql;

SELECT pgsql_example();

```

위의 문장은 선언부에서 counter라는 변수를 선언하고 실행부에서 0으로 나누는 연산을 수행합니다. 정수를 0으로 나눌 때 division_by_zero 에러를 발생하기 때문에 예외 처리부로 진입됩니다. 이 경우, RAISE NOTICE 문을 실행하여 'DIV 0'이라는 메시지를 출력하고 카운터를 1로 나누게 됩니다. division_by_zero는 정수를 0으로 나누었을 때 발생하는 PostgreSQL 에러 코드입니다.

1.2 기초 코딩

1.2.1 변수 및 상수 선언

블록내에서 사용되는 모든 변수는 FOR문을 제외하고 해당 블록의 선언부에 선언합니다. PL/pgSQL 변수는 SQL 데이터 형으로(integer, varchar, char 등) 선언 가능합니다.

[예시] PL/pgSQL 변수 선언

```

user_id Integer :=40;
quantity numeric(5);
url varchar;
myrow RECORD;

```

상수 선언도 변수 선언과 마찬가지로 선언부에 선언하며 CONSTANT라는 키워드로 선언합니다. 상수는 선언과 동시에 반드시 값을 할당해야 합니다.

[예시] PL/pgSQL 상수 선언

```
year CONSTANT INTEGER := 30;
```

○ %TYPE

PL/pgSQL은 연산을 수행할 때 테이블의 데이터를 이용할 수 있으나 해당 칼럼의 데이터 타입을 알고 있어야 한다는 단점을 극복하기 위해 %TYPE 속성을 사용합니다.

%TYPE이라는 속성을 이용하면 각 칼럼의 데이터 타입을 따로 정의하지 않아도 됩니다. %TYPE은 참조할 테이블 칼럼의 데이터 타입을 자동 선언됩니다.

[%TYPE 사용법]

```
변수명 테이블명.칼럼명%TYPE
```

[예시] %TYPE 사용

```
CREATE FUNCTION type_example(i float) RETURNS float AS $$  
DECLARE p_sal emp.sal%TYPE; -- %TYPE 선언  
BEGIN  
    sal = i + 10.0;  
    RETURN p_sal;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT type_example(9.5);  
type_example  
-----  
19.5
```

emp 테이블의 sal 칼럼은 numeric 타입으로 소수점 2자리까지 출력합니다. 변수 sal은 \$TPYE 형태로 선언하였고, 이 경우 sal 칼럼의 데이터 타입인 numeric 타입을 가지게 되므로 소수점 연산 가능합니다.

○ %ROWTYPE

%TYPE은 변수 하나값에 대해 적용되지만 %ROWTYPE은 하나 이상의 값에 적용 가능합니다. %ROWTYPE의 선언은 %TYPE과 동일하게 선언할 수 있습니다.

[예시] %ROWTYPE 사용

```
CREATE OR REPLACE FUNCTION rowtype_test(p_empno IN emp.empno%TYPE) RETURNS  
void AS $$  
DECLARE v_emp emp%ROWTYPE ; -- employees 테이블 속성을 그대로 사용  
BEGIN
```

```

SELECT empno, ename
INTO v_emp.empno, v_emp.ename
FROM emp
WHERE empno = p_empno;
RAISE NOTICE 'NUMBER : %', v_emp.empno;
RAISE NOTICE 'NAME : %', v_emp.ename;
END;$$ LANGUAGE plpgsql;

SELECT rowtype_test(7369) ;

NOTICE: NUMBER : 7369
NOTICE: FIRST NAME : SMITH

```

위의 함수 실행 시 사원 번호 값을 매개 변수로 전달 받아 empno를 조회하고 empno, ename 칼럼의 값을 %ROWTYPE으로 선언된 v_emp.empno, v_emp.ename에 저장하여 출력합니다. 매개 변수로 선언된 p_empno는 emp.empno%TYPE 형태로 선언이 되어 empno 칼럼의 속성 하나만을 가지지만 %ROWTYPE 으로 선언된 v_emp 변수는 emp 테이블의 있는 모든 칼럼의 속성을 사용 가능합니다.

○ RECORD

레코드(RECORD)은 테이블 형태의 데이터 타입입니다. 레코드는 테이블의 칼럼처럼 여러 개의 필드로 구성되어 있고, PL/pgSQL 블록에서 임시로 사용할 수 있는 하나의 데이터 타입입니다. 자세한 내용은 "커서"에서 자세히 설명하도록 하겠습니다.

1.2.2 커서

SELECT 문장을 실행하면 조건에 따른 결과가 추출되며, 추출되는 결과는 한 건이 될 수도 있고 여러 건이 될 수도 있습니다. 이를 결과 셋(result set) 혹은 결과집합이라고 부릅니다. 쿼리에 의해 반환되는 결과는 메모리상에 위치하게 되는데 이는 커서(cursor)를 사용하여 접근 가능합니다. 즉 커서를 사용하면 개별 데이터 집합에 접근 가능합니다. 단, 오라클과 달리 커서를 선언할 때 커서명은 반드시 커서 키워드 앞에 선언해야 합니다.

[커서 변수의 선언 문법]

커서명 CURSOR FOR 쿼리

[예시] 커서 변수의 선언

```
DECLARE curs1 refcursor;
curs2 CURSOR FOR SELECT * FROM emp
```

선언한 커서를 사용하기 위해서는 커서를 호출(OPEN), 실행(FETCH), 종료(CLOSE) 명령어 필요합니다.

[예시] 커서 변수의 사용

```
CREATE OR REPLACE FUNCTION curs_test()
RETURNS record AS $$

DECLARE
customers_record RECORD;
TestCursor CURSOR FOR SELECT * FROM emp ;
BEGIN
OPEN TestCursor ;
FETCH TestCursor INTO customers_record;
RETURN customers_record;
CLOSE TestCursor;
END;
$$ LANGUAGE plpgsql;

SELECT curs_test();
curs_test
-----
(7369,SMITH,CLERK,7902,"17 -DEC-80 00:00:00",800.00,,20)
(1 row)
```

선언부에서는 커서의 결과 값을 받는 RECORD 형 변수와 emp 테이블의 데이터를 조회하는 커서를 선언합니다. 선언한 커서를 실행하기 위해 OPEN 키워드를 사용하여 커서를 활성화시키고, FETCH 키워드로 커서를 실행합니다. 다음으로 INTO 키워드로 customers_record 에 결과값을 삽입하여 반환한 후 CLOSE 명령어를 사용하여 커서를 종료합니다. 함수를 실행해 보면 emp 테이블에 맨 첫 열의 데이터가 출력되는 것을 확인 가능합니다.

1.2.3 조건문

○ IF 문

IF 명령문의 조건에 의해 명령을 실행합니다. 일반적인 프로그래밍 언어와 비교해서 문법적인 부분 외에는 별 차이 없이 사용 가능합니다.

[IF 문법]

```
-- 처리 조건이 한 개일 경우
IF 조건 THEN 처리문 END IF;

-- 처리 조건이 두 개일 경우
IF 조건 THEN 처리문1 ELSE 처리문2 END IF;

-- IF 명령문 문법: 처리 조건이 여러 개일 경우
IF 조건 THEN 처리문1 ELSEIF 조건2 THEN 처리문2 ... ELSE 처리문 END IF;
```

[예시] IF 사용

```
CREATE OR REPLACE FUNCTION if_example(grade char) RETURNS void AS $$

DECLARE
BEGIN
IF grade= 'A' THEN
    RAISE NOTICE 'Excellent';
ELSEIF grade='B' THEN
    RAISE NOTICE 'Good';
ELSEIF grade='C' THEN
    RAISE NOTICE 'Fair';
ELSEIF grade='D' THEN
    RAISE NOTICE 'Poor';
END IF;
END;
$$ LANGUAGE plpgsql;

SELECT if_example('A');
NOTICE: Excellent
```

○ CASE 문

조건에 따라 값 처리할 수 있습니다. IF/ELSE 구문과 유사합니다.

[CASE 문법]

```
-- 특정 칼럼 비교
CASE 칼럼 WHEN 조건 THEN 처리
[WHEN ...]
[ELSE 처리]
END

-- 조건 비교
CASE WHEN 조건 비교 THEN 처리
[WHEN ...]
[ELSE 처리]
END CASE
```

[예시] CASE 사용

```
CREATE OR REPLACE FUNCTION case_example(grade char) RETURNS void AS $$

DECLARE
BEGIN
CASE grade
WHEN 'A' THEN
    RAISE NOTICE 'Excellent';
WHEN 'B' THEN
    RAISE NOTICE 'Good';
WHEN 'C' THEN
    RAISE NOTICE 'Pair';
WHEN 'D' THEN
    RAISE NOTICE 'Poor';
END CASE;
END;
$$ LANGUAGE plpgsql;
SELECT case_example('A');
NOTICE: Excellent
```

1.2.4 반복문

○ LOOP 문

LOOP문은 EXIT문 또는 RETURN문에 의해 종료될 때까지 무한하게 반복되는 문장입니다. LOOP 문에 별도의 조건을 명시하지 않으면 무한 루프에 빠지게 되므로 반드시 EXIT나 RETURN문을 명시해야 합니다. WHILE LOOP문은 일반 LOOP문과 다르게 해당 조건식이 참이여야 LOOP문이 반복됩니다. WHILE 문에 들어가는 조건식은 반드시 초기화가 필요합니다.

[LOOP 문법]

```
LOOP 처리문; END LOOP;
```

[WHILE LOOP 문법]

```
WHILE 조건 LOOP 처리문; END LOOP;
```

[예시1] LOOP 사용

```
CREATE OR REPLACE FUNCTION loop_example() RETURNS void AS $$  
DECLARE  
    test_number INTEGER;  
    result_num INTEGER ;  
BEGIN  
    test_number := 1;  
    LOOP  
        result_num := 2 * test_number;  
        IF result_num > 20 THEN  
            EXIT;  
        ELSE  
            RAISE NOTICE ' %', result_num;  
        END IF;  
        test_number := test_number + 1;  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT Loop_example();  
NOTICE: 2  
NOTICE: 4  
NOTICE: 6  
NOTICE: 8  
NOTICE: 10  
NOTICE: 12  
NOTICE: 14  
NOTICE: 16  
NOTICE: 18  
NOTICE: 20
```

위의 문장은 2의 배수를 출력하는 문장으로 result_num가 20이 넘으면 LOOP문을 빠져나오고 그렇지 않으면 test_number를 1씩 증가시키는 PL/pgSQL LOOP문 안에 있는 IF 조건문에서 EXIT 문을 실행하면 LOOP문을 종료합니다.

[예시2] WHILE LOOP 사용

```
CREATE OR REPLACE FUNCTION while_loop_example()
RETURNS void AS $$

DECLARE
    test_number INTEGER;
    result_num INTEGER ;
BEGIN
    test_number := 1;
    result_num := 0;
    WHILE result_num < 20 LOOP
        result_num := 2 * test_number;
        RAISE NOTICE '%', result_num;
        test_number := test_number + 1;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

○ FOR 문

FOR는 정수값의 범위를 반복하는 루프를 생성합니다.

[FOR 문법]

```
FOR 카운터 IN [REVERSE] 최소값[최대값] ... 최대값[최소값] LOOP 처리문;
END LOOP;
```

[예시] FOR 사용

```
CREATE OR REPLACE FUNCTION for_loop_example()
RETURNS void AS $$

DECLARE
    test_number INTEGER;
    result_num INTEGER ;
BEGIN
    test_number := 1;
    result_num := 0;
    FOR test_number IN 1..10 LOOP
        result_num := 2 * test_number;
        RAISE NOTICE '%', result_num;
    END LOOP;
    RAISE NOTICE 'second ';
```

```

result_num :=0;
FOR test_number IN REVERSE 10..1 LOOP
    result_num := 2 * test_number;
    RAISE NOTICE ' %', result_num;
END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT for_loop_example();
NOTICE:  2
NOTICE:  4
NOTICE:  6
NOTICE:  8
NOTICE:  10
NOTICE:  12
NOTICE:  14
NOTICE:  16
NOTICE:  18
NOTICE:  20
NOTICE:  second
NOTICE:  20
NOTICE:  18
NOTICE:  16
NOTICE:  14
NOTICE:  12
NOTICE:  10
NOTICE:  8
NOTICE:  6
NOTICE:  4
NOTICE:  2

```

첫 번째 FOR문은 test_number 변수가 증가하면서 연산하는 것을 보여줍니다. 두 번째 FOR문은 test_number 가 감소하면서 연산을 보여줍니다. 즉, 첫 번째 FOR문은 2의 배수가 증가하는 것을 보여주고, 두 번째 FOR문은 첫 번째 FOR문의 내용을 역으로 출력합니다. GOTO문은 특정한 곳으로 분기를 하는데 사용되는데 PostgreSQL에서는 지원하지 않습니다. GOTO 문을 사용해야 되는 경우에는 IF-THEN 구문이나 LOOP 구문으로 대체해서 사용해야 합니다.

1.2.5 Null 문

PL/pgSQL에서는 NULL문을 제공합니다. 즉, 아무것도 처리하지 않음을 의미합니다.

[예시] Null 문 사용

```

CREATE OR REPLACE FUNCTION case_example(grade char) RETURNS void AS $$

DECLARE
BEGIN
CASE grade
WHEN 'A' THEN
    RAISE NOTICE 'Excellent';
WHEN 'B' THEN
    RAISE NOTICE 'Good';
WHEN 'C' THEN
    RAISE NOTICE 'Pair';
WHEN 'D' THEN
    RAISE NOTICE 'Poor';
ELSE NULL;
END CASE;
END;

$$ LANGUAGE plpgsql;

SELECT case_example('E');
case_example
-----
(1 row)

```

CASE 문에 마지막에 ELSE 에 NULL 을 사용함으로써 A,B,C,D 를 제외한 다른 문자가 입력이 되었을 때 어떠한 내용도 출력하지 않습니다. 보통 NULL 문장은 예외 처리시 자주 사용되며 EXCEPTION 절에서 명시한 예외 이외의 오류가 발생하였을 경우에 NULL문을 명시합니다.

1.2.6 예외 처리

프로그램 실행 중 예외의 경우가 발생하여 비정상적으로 작동하는 것을 방지합니다.

[예외 처리 문법]

```

EXCEPTION
WHEN 예외처리상태 THEN
    처리방법
WHEN 예외처리상태 THEN
    처리방법

```

[예시1] 예외 처리

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

대표적인 예외 처리 코드입니다.

에러코드	상태 이름	의미
42601	syntax_error	문법 에러
53200	out_of_memory	가용 메모리 없음
P0002	no_data_found	데이터 없음
22012	division_by_zero	0 으로 나눔
22005	error_in_assignment	대입 에러

[예외와 메세지 문법]

```
RAISE 에러레벨 ...;
```

에러 레벨은 DEBUG, LOG, INFO, NOTICE, WARNING, EXCEPTION 존재합니다.

[예시2] 예외 처리

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
USING HINT = 'Please check your user ID';
```

[예시3] 예외 처리

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'uniqueViolation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

1.2.7 주석

프로그램의 설명이나 참고사항을 주석으로 처리할 수 있습니다.

[주석 사용법]

```
-- 한 문장
/* 여러 문장 */
```

PostgreSQL은 오라클과 다르게 코멘트 사용시 반드시 시작이 존재 한다면 종료 주석도 해주어야 합니다.

[예시] 잘못된 주석 사용

```
***** 인사 담당자*****
/* 조직 코드
/* Opensource S/W
*****
```

[예시] 올바른 주석 사용

```
***** 인사 담당자*****
/* 조직 코드 */
/* Opensource S/W */
*****
```

1.3 서브 프로그램

1.3.1 사용자 정의 함수

PostgreSQL에서 기본적으로 제공하는 함수 외에 사용자가 용도에 맞게 함수를 정의해서 사용 가능합니다.

[사용자 정의 함수 생성]

```
CREATE OR REPLACE FUNCTION 함수명 (파라미터1 데이터타입, 파라미터2, 데이터타입, ...)
RETURNS 데이터타입 AS
DECLARE
    변수 선언
BEGIN
    처리 내용;
    RETURN 리턴 값;
END;
```

CREATE OR REPLACE FUNCTION 명령은 함수를 생성하는 명령어로 기존의 작성된 함수를 수정할 때 DROP 시키지 않고 수정 가능합니다. 파라미터에는 함수가 받는 파라미터의 이름과 데이터 타입을 명시합니다. RETURNS은 반환하는 데이터 타입을 명시합니다. 리턴하는 데이터가 없는 경우 void 키워드를 사용합니다. RETURNS형이 void가 아니면 실행부에 RETURN을 반드시 명시해줘야 합니다.

[예시] 사용자 정의 함수 생성

```

CREATE OR REPLACE FUNCTION emp_salaries (emp_id numeric)
RETURNS numeric AS $$

DECLARE
nSalaries numeric(9);

BEGIN
nSalaries := 0;
SELECT sal
INTO nSalaries
FROM emp
WHERE empno = emp_id;
RETURN nSalaries;
END;

$$ LANGUAGE plpgsql;

SELECT empno, emp_salaries(empno) FROM emp WHERE deptno = 10;

empno | emp_salaries
-----+-----
 7782 |      2450
 7839 |      5000
 7934 |      1300
(3 rows)

```

위의 예제는 사원번호를 받아와 해당 사원의 급여를 반환하는 함수입니다. `emp_salaries()` 함수를 사용하여 부서번호가 10인 부서의 사원 번호, 연봉 정보를 찾는 문장입니다. 즉, WHERE 절에서 검색된 `empno`가 `emp_salaries()`의 파라미터로 입력되어 연봉 정보를 출력합니다.

1.3.2 프로시저

PostgreSQL에서는 프로시저를 지원하지 않습니다. 즉, `CREATE OR REPLACE PROCEDURE` 명령어를 사용하여 프로시저를 정의 불가능합니다. 프로시저는 반환 값이 없는 함수이므로, 사용자 함수에 반환 값이 없는 함수로 구현 가능합니다. EDB PAS는 프로시저 문법을 지원합니다.

1.3.3 패키지

PostgreSQL에서는 오라클의 패키지 기능을 지원하지 않습니다. EDB PAS는 패키지를 지원합니다.

1.3.4 트리거

트리거는 특정 이벤트가 발생했을 때 목시적으로 어떤 동작이나 처리를 수행합니다. 일반적으로

트리거는 임의의 테이블에 데이터가 insert, update, delete 될 때 또 다른 테이블의 데이터를 자동적으로 조작할 경우에 사용. 예를 들면 히스토리나 로그 정보를 기록할 때 사용합니다.

PostgreSQL에서 트리거를 사용하려면 트리거 함수와 트리거를 정의 해야 합니다. 트리거는 그 트리거가 실행할 함수를 지정하는 방식으로 생성 합니다. 즉, 테이블에서 지정하는 트리거의 실작업은 그 트리거에서 정의한 함수가(트리거 함수) 실행합니다.

○ 트리거 함수

트리거 함수란 트리거가 호출하는 함수로써 트리거가 생성되기 전에 정의되어 있어야 합니다. 트리거 함수의 특징은 인자가 없고 리턴 값이 트리거입니다. 트리거 인자는 TG_ARGV로 부터 받게 됩니다. 트리거 함수는 return 구문으로 함수가 종료되어야 하며 new, old, null 세 개의 해당 작업 대상이 된 row를 지칭하는 미리 정의된 record 변수를 지정해야 합니다. 트리거가 여러 개인 경우 첫 번째 트리거 함수에서 return null로 종료되는 경우, 다음 트리거가 실행해야 할 모든 작업이 중지 됩니다.

[트리거 함수 사용 문법]

```
CREATE OR REPLACE FUNCTION 함수명 (파라미터1 데이터타입, 파라미터2...)
RETURNS trigger AS
DECLARE
    변수 선언
BEGIN
    처리 내용;
    RETURN 리턴 값;
END;
```

[예시] 트리거 함수 사용

```
CREATE OR REPLACE FUNCTION add_job_history()
RETURNS trigger AS $$

DECLARE
    p_empno      numeric(6,0);
    p_start_date date;
    p_end_date   date;
    p_job_id     character varying(10);
    p_deptno     numeric(4,0);

BEGIN
```

```

IF (TG_OP = 'UPDATE') THEN
    p_empno = OLD.empno;
    p_start_date = NEW.hire_date;
    p_end_date = now();
    p_job_id = OLD.job_id;
    p_deptno= OLD.deptno;
END IF;
<<insert_update>>
LOOP
BEGIN
    INSERT INTO job_history (empno, start_date, end_date, job_id, deptno)
    VALUES(p_empno, p_start_date, p_end_date, p_job_id, p_deptno);
    EXIT insert_update;
END;
    END LOOP insert_update;
RETURN NULL;
END $$ LANGUAGE plpgsql;

```

위의 예제는 emp테이블의 job_id 칼럼 값이 변경되면, 변경 된 이력이 job_history 테이블에 기록되기 위한 트리거 함수를 구현한 예제입니다. 아래는 트리거 함수의 변수입니다.

변수명	데이터형	설명
NEW	RECORD	INSERT/UPDATE 에 의하여 생성된 새로운 로우
OLD	RECORD	INSERT/UPDATE 에 의하여 생성되기 전의 로우
TG_NAME	name	실제로 트리거 실행의 이름
TG_WHEN	text	트리거에 정의된 BEFORE, AFTER, INSTEAD OF 문자열
TG_LEVEL	text	트리거의 정의된 ROW 또는 STATEMENT 문자열
TG_OP	text	트리거가 동작하게 된 이벤트(INSERT, UPDATE, DELETE) 문자열
TG_RELID	oid	트리거 호출 테이블의 객체 ID.
TG_RELNAME	name	트리거 호출 테이블 이름
TG_TABLE_NAME	name	트리거 호출 테이블의 이름.
TG_TABLE_SCHEMA	name	트리거 호출 스키마명
TG_NARGS	integer	CREATE TRIGGER 문에서 트리거 함수의 인자의 수
TG_ARGS[]	text 형의 배열	CREATE TRIGGER 문에서의 인자
변수명	데이터형	설명

○ 트리거

트리거 함수가 생성되었으면 트리거는 그 함수를 호출하여 정의 가능합니다. 트리거의

종류에는 행 트리거와 문장 트리거가 있습니다. 행 트리거는 칼럼의 각 행에 이벤트가 발생할 때마다 실행되는 반면 문장 크리거는 이벤트에 의하여 한번만 실행됩니다.

[트리거 사용 문법]

```
CREATE TRIGGER 트리거 이름
  { BEFORE | AFTER } 트리거 이벤트 ON 테이블 명
  [ FOR [ EACH ] { ROW | STATEMENT } ]
  EXECUTE PROCEDURE 트리거 함수명(인자)
```

- BEFORE | AFTER : 이벤트의 발생 전 또는 발생 후에 트리거 함수 호출 여부
- FOR EACH ROW | FOR EACH STATEMENT: 행 트리거와 문장 트리거 여부. 디폴트는 FOR EACH STATEMENT

[예시] 트리거 사용

```
CREATE TRIGGER update_job_history
  AFTER UPDATE OF job_id, deptno
  ON emp
  FOR EACH ROW
  EXECUTE PROCEDURE add_job_history();
```

[예시]는 emp 테이블의 job_id와 deptno 칼럼이 변경될 경우 트리거 함수인 add_job_history를 호출하여 job_history 테이블에 변경된 내역을 저장하고 있음을 보여줍니다. EDB PAS는 오라클 호환 기능을 제공하므로 트리거 함수를 생성하지 않고 프로시저 생성으로 트리거 생성이 가능합니다.

1.3.5 룰

룰은 트리거와 동일한 역할을 하는 PostgreSQL의 고유 객체입니다.

[룰 사용 문법]

```
CREATE OR REPLACE RULE 룰이름 AS ON 이벤트
  TO 테이블
  DO [ ALSO | INSTEAD ] { NOTHING | 액션 }
```

[예시1] 룰사용

```
CREATE or replace RULE update_job_history AS ON UPDATE TO employees  
DO (  
    INSERT INTO t  
    VALUES(OLD.employee_id,NEW.hire_date,now(), OLD.job_id,OLD.department_id);  
)
```

[예시2] 룰사용

```
CREATE OR REPLACE RULE delete_temp AS ON DELETE TO code DO INSTEAD NOTHING;
```

[예시1]은 트리거 함수와 트리거로 구현한 앞의 예시를 룰로 구현한 예시입니다. [예시2]는 테이블에(code) 삭제 작업이 일어나면, 실제 작업을 하지 못하게 하는 작업입니다. 테이블에 트리거와 룰이 동시에 생성되어 있다면 트리거가 우선 동작합니다. 또한 트리거보다 룰이 더 빠릅니다.