Session 6

Indexing

Transactions

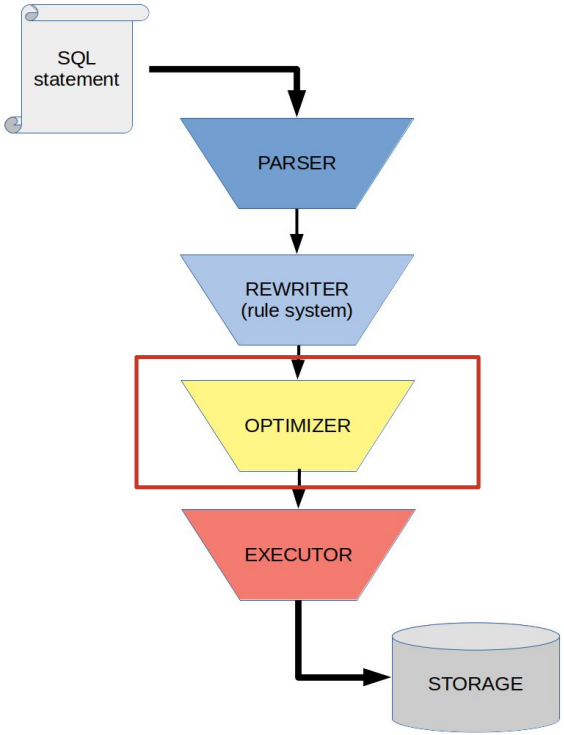
PostgreSQL Roles

sourcemind

How SQL statements execute?

```
SELECT *
    FROM book INNER JOIN topic
        ON book.topic_id = topic.id
    WHERE num_pages > 100 AND topic.id = 1;
```

Optimizer



- Optimizer is responsible to find the fastest way to reach to the data in a reasonable time.

```
SELECT *
    FROM book INNER JOIN topic
        ON book.topic_id = topic.id
    WHERE num_pages > 100 AND topic.id = 1;
```

- Each way to reach to the data has a **cost**.
- The optimizer estimates the cost and the path with least cost wins.
- Each operation has a cost (load time, processing)

Optimizer nodes

```
SELECT * FROM book ORDER BY title;
```

Set of actions to executes: Two nodes

1) Retrieve all the data

   Find where the book data is physically stored on the disk
   Seek the disk head to that location and read all the records block-by-block

2) Sort the data
   When the data is loaded in the memory, sort them by title

Sequential Scan

- Sequential nodes are those nodes that will be executed sequentially, one after the other, in order to achieve the final result.

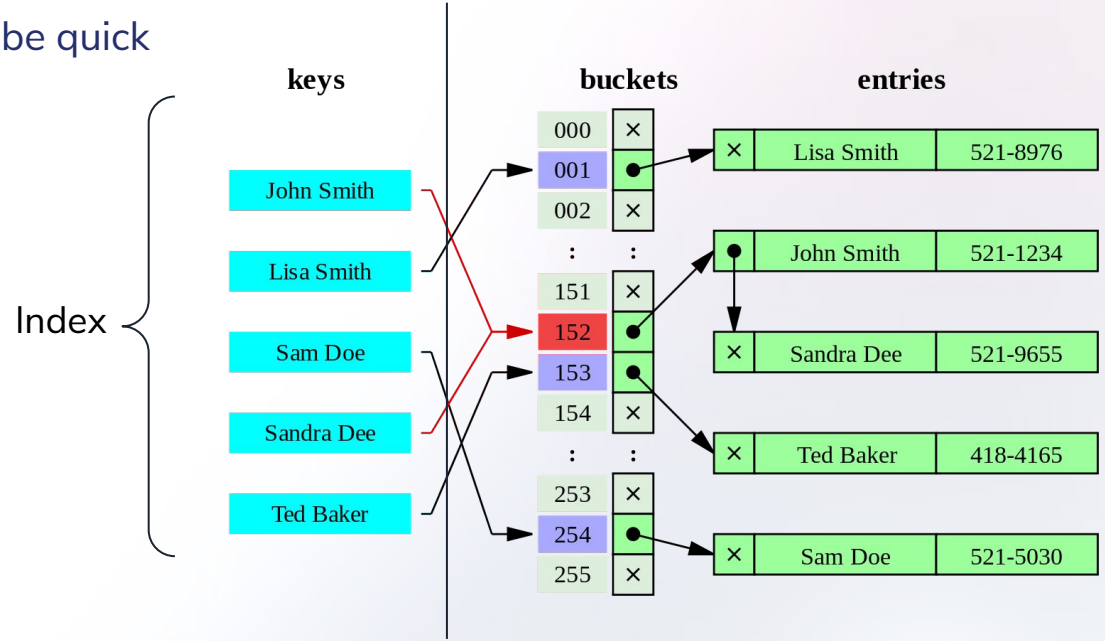
- Sequential Scan (Seq Scan)

```
SELECT * FROM book;
```

- When the entire table data is asked
- When the filterable part is too small
- Fallback option

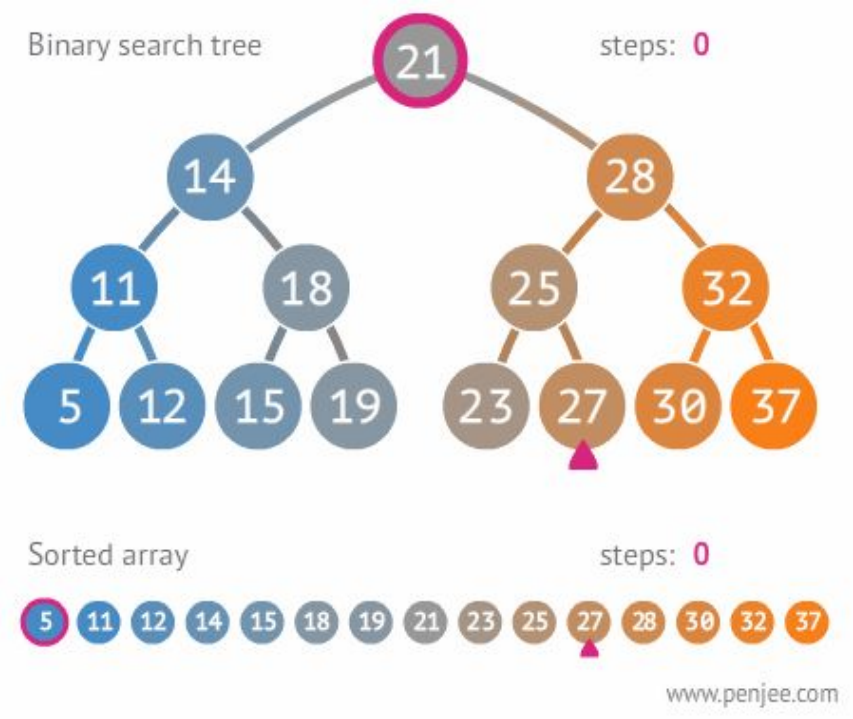
Index

- An index is an auxiliary data structure related to the main data that helps look up the required data.
- Finding an index must be quick
  - Sorted indexes
  - Hashed indexes



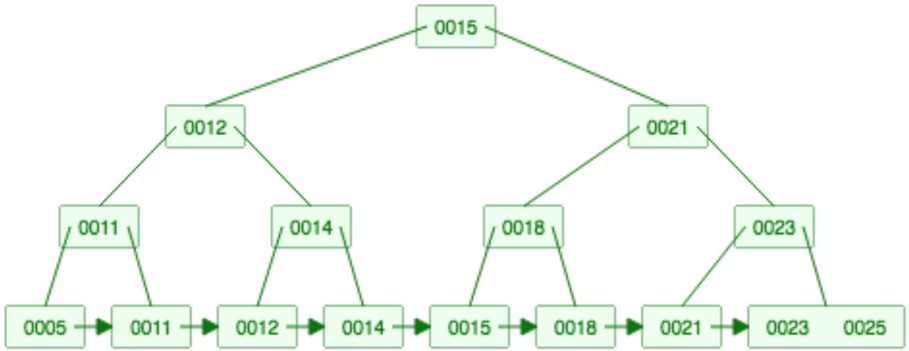
sourcemind

Index



- DBMS uses a data structure which is similar to BST, called **B-Tree**
- In a B-Tree, the leaf nodes are pointers to the blocks that contain table rows
- Therefore, using an index is faster for finding a row compared with sequential scan

B-Tree



- B-Tree is not necessarily binary
- B-Tree has higher **branching factor** to fit more data in fewer levels

Visualization

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Index Scan

- Helps find a key to data quickly, enforce uniqueness, ...
  - Example: Table of contents, indexes at the end of a book

- PostgreSQL reads the chosen index, and from that, it goes seeking the tuples, reading again from the storage.

```
SELECT * FROM book WHERE id = 3;
```

```
CREATE TABLE book (
    id SERIAL INT PRIMARY KEY,
    title VARCHAR
);
```

CREATE INDEX

- Create index on title

```
CREATE INDEX title_idx ON book (title);
CREATE UNIQUE INDEX title_idx ON book (title);
CREATE INDEX title_idx2 ON book USING hash (title);
```

- Create index on `(title, author_id)`

```
CREATE INDEX title_idx ON book (title) INCLUDE (author_id);
```

- Drop index

```
DROP INDEX title_idx2;
```

Index-only Scan

- If the requested data only involves columns that belong to the index, PostgreSQL is smart enough to avoid the second trip to storage since it can extract all the required information directly from the index.

```
SELECT count(*) FROM book;
```

EXPLAIN statement

- `EXPLAIN` is the statement that allows you to see how PostgreSQL is going to execute a specific query.

```
EXPLAIN SELECT * FROM book;
```

```
+-------------------------------------------------------+
| QUERY PLAN                                            |
|-------------------------------------------------------|
| Seq Scan on book  (cost=0.00..1.03 rows=3 width=131) |
+-------------------------------------------------------+
```

- Execution nodes (Sequential scan)
- Estimated cost (start-up cost .. final cost)
- Estimated rows to return
- Width: Average bits every tuple occupies

EXPLAIN statement

- Two nodes: Sequential scan & Sort

```
EXPLAIN SELECT * FROM book ORDER BY title DESC;
```

```
+-------------------------------------------------------------+
| QUERY PLAN                                                  |
|-------------------------------------------------------------|
| Sort  (cost=1.05..1.06 rows=3 width=131)                    |
|   Sort Key: title DESC                                      |
|   ->  Seq Scan on book  (cost=0.00..1.03 rows=3 width=131)  |
+-------------------------------------------------------------+
```

EXPLAIN statement

- More nodes

```
EXPLAIN SELECT * FROM book INNER JOIN author ON author.id = book.author_id;
```

```
+-----------------------------------------------------------------------+
| QUERY PLAN                                                            |
|-----------------------------------------------------------------------|
| Hash Join  (cost=1.07..21.14 rows=3 width=214)                        |
|   Hash Cond: (author.id = book.author_id)                             |
|   ->  Seq Scan on author  (cost=0.00..17.30 rows=730 width=83)        |
|   ->  Hash  (cost=1.03..1.03 rows=3 width=131)                        |
|         ->  Seq Scan on book  (cost=0.00..1.03 rows=3 width=131)      |
+-----------------------------------------------------------------------+
```

EXPLAIN options

- Print the explain results in JSON

```
EXPLAIN (FORMAT JSON)  SELECT title FROM book ORDER BY title DESC;
```

- Actually run and report the results

```
EXPLAIN (ANALYZE) SELECT title FROM book ORDER BY title DESC;
```

- More details: Verbose

```
EXPLAIN (ANALYZE, VERBOSE on) SELECT * FROM book;
```

Shakespeare database

1. Update repo
2. Create database shakespeare
3. Using psql, import the sql file

```
psql -h localhost -p 5433 -U postgres -d shakespeare < shakespeare.sql
```

Query tuning

- Find the paragraphs by work, e.g. Othello

```
SELECT COUNT(*), workid FROM paragraph GROUP BY workid;
```

- Inspect the paragraph table and see it's structure and indexes
- Run a query to count paragraphs in Othello
- Analyze the query plan and see how to improve it

```
SELECT COUNT(*) FROM paragraph WHERE workid = 'othello';
EXPLAIN SELECT COUNT(*) FROM paragraph WHERE workid = 'othello';
CREATE INDEX idx_par_workid ON paragraph (workid);

EXPLAIN SELECT COUNT(*) FROM paragraph WHERE workid = 'othello';
```

Transactions

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- Logically one operation, but internally consists of many steps.
- A transaction is complete if all of its steps are completed.

Transaction: Bank transfer

- Transfer $50 from account A to account B

```
1. read(A)
2. A:=A−50
3. write(A)
4. read(B)
5. B:=B+50
6. write(B)
```

Transaction: Supermarket Sales

- Complete the purchase of an item

1. Add item to order
2. Receive payment
3. Subtract from inventory

Transaction Issues

1. Failures of various kinds, such as hardware failures and system crashes
2. Concurrent execution of multiple transactions

```
1. read(A)
2. A:=A−50
3. write(A)
4. read(B)          Failure
5. B:=B+50
6. write(B)
```

Transaction Atomicity

- If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
- Failure could be due to software or hardware
- The system should ensure that updates of a partially executed transaction are not reflected in the database.
- All or nothing

```
1. read(A)
2. A:=A−50
3. write(A)
4. read(B)
5. B:=B+50
6. write(B)
```

Transaction Consistency

- *The sum of A and B is unchanged by the execution of the transaction.*
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - – e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent

```
1. read(A)
2. A:=A−50
3. write(A)
4. read(B)
5. B:=B+50
6. write(B)
```

Transaction Isolation

- Parallel transactions must not see intermediate state.
- They must appear “as if” they are running serially.

T1	T2
1. read(A)	
2. A:=A−50	
3. write(A)	
	read(A), read(B), print(A+B)
4. read(B)	
5. B:=B+50	
6. write(B)	

Concurrent Transactions

Two transactions:

T1: Transfer $50 from A to B

T2: Transfer 10% of A to B

Initially A = 100, B = 300

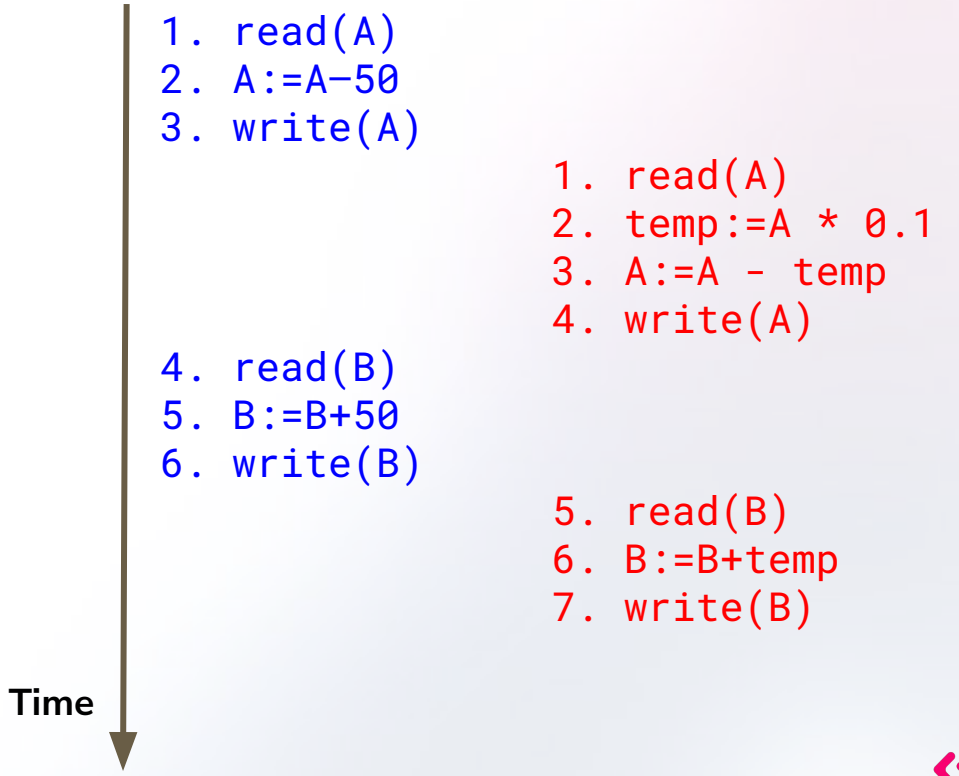
A + B must be preserved

```
1. read(A)
2. A:=A−50
3. write(A)
4. read(B)
5. B:=B+50
6. write(B)
```

```
1. read(A)
2. temp:=A * 0.1
3. A:=A - temp
4. write(A)
5. read(B)
6. B:=B+temp
7. write(B)
```

Concurrent Transactions

- T1: Transfer $50 from A to B
- T2: Transfer 10% of A to B



Transaction Durability

- Once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.
- Recovery after software / hardware failures.

```
Begin Transaction
1. read(A)
2. A:=A−50
3. write(A)
4. read(B)
5. B:=B+50
6. write(B)
End Transaction
```

ACID Transactions

A transaction is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure:

- **Atomicity**: Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency**: Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability**: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transactions Example

Reference: https://www.postgresqltutorial.com/postgresql-transaction/

```
CREATE TABLE account (
    id SERIAL INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    balance DEC(15,2) NOT NULL,
);

-- Add funds into accounts

-- Verify transactions in parallel sessions

-- Begin...commit and rollback
```

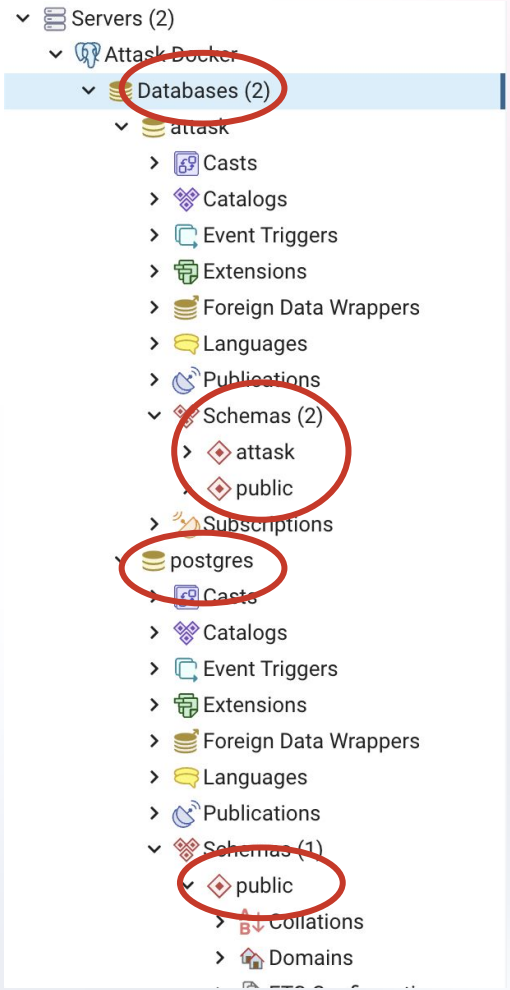
PostgreSQL Overview

- Open-source, community-driven
- Known for stability, scalability, and safeness
- ACID-compliant
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- 1970s: Ingres DB
- 1986: Post-Ingres ⇒ Postgres
- 1995: SQL Support ⇒ Postgres95
- 1996: Public-hosted source code ⇒ PostgreSQL

Databases and schemas

A single Postgres server instance may contain multiple databases and schemas.

```
Postgres > Database 1
             > Default schema (public)
             > Schema 2
             > Schema N
         > Database 2
             > Default schema (public)
         > Database 3
             Default schema (public)
             ...
```



sourcemind

PostgreSQL Concepts

- Cluster: Entire PostgreSQL service
- Postmaster and backend processes
- Database: Namespace, table, trigger, ...
- `$PGDATA`: Main storage directory
- WAL (Write-ahead log): Intent log, used for recovery

PostgreSQL Processes

```
root@127f7d66d9cf:/# ps -aef | grep postgres
postgres     1     0  0 10:13 ?        00:00:00 postgres
postgres    70     1  0 10:13 ?        00:00:00 postgres: checkpointer
postgres    71     1  0 10:13 ?        00:00:00 postgres: background writer
postgres    72     1  0 10:13 ?        00:00:00 postgres: walwriter
postgres    73     1  0 10:13 ?        00:00:00 postgres: autovacuum launcher
postgres    74     1  0 10:13 ?        00:00:00 postgres: stats collector
postgres    75     1  0 10:13 ?        00:00:00 postgres: logical replication launcher
postgres   171     1  0 10:21 ?        00:00:00 postgres: postgres postgres 172.17.0.1(57902) idle
postgres   203     1  0 10:23 ?        00:00:00 postgres: postgres postgres 172.17.0.1(57918) idle
```

Psql: List Databases

```
## List existing databases
psql -h localhost -p 5432 -U postgres -l
Password for user postgres:
                                  List of databases
   Name    |  Owner   | Encoding |  Collate   |   Ctype    |   Access privileges
-----------+----------+----------+------------+------------+-----------------------
 postgres  | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 template0 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
(3 rows)
```

Psql: Interactive SQL command line

- Install psql command-line tool
- Connect to the server process

```
## Connect via command-line client

psql -h localhost -p 5432 -U postgres

Password for user postgres:

psql (13.2)

Type "help" for help.

postgres=#
```

Psql: Connection string

psql postgres://user@host:port/database

```
psql postgres://postgres@localhost:5432/postgres
Password for user postgres:
psql (13.2)
Type "help" for help.


postgres=#
```

Psql: Listing available databases

```
psql postgres://postgres@localhost:5432/postgres -l
```

```
Password for user postgres:
                                  List of databases
   Name    |  Owner   | Encoding |  Collate   |   Ctype    |   Access privileges
-----------+----------+----------+------------+------------+-----------------------
 attask    | postgres | UTF8     | C          | C          |
 postgres  | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 template0 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
(4 rows)
```

sourcemind

Psql: Running queries

```
postgres=# SELECT current_date, current_time;
 current_date |    current_time
--------------+--------------------
 2021-04-01   | 16:08:22.171908+00
(1 row)
```

Inside PostgreSQL server

```
docker exec -it postgres bash

root@127f7d66d9cf:# su postgres

postgres@127f7d66d9cf:~$ PATH=$PATH:/usr/lib/postgresql/11/bin

postgres@127f7d66d9cf:~/data/base$ pg_ctl status

pg_ctl: server is running (PID: 1)

postgres@127f7d66d9cf:~$ echo $PGDATA

/var/lib/postgresql/data

postgres@127f7d66d9cf:~$ cd $PGDATA/

postgres@127f7d66d9cf:~/data$ ls

base    pg_commit_ts  pg_hba.conf  ...
```

User-defined objects in PostgreSQL

```
postgres@127f7d66d9cf:~/data$ ls base/
1  13066  13067  16384
```

```
postgres@127f7d66d9cf:~/data$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  16384         attask  pg_default
  13067       postgres  pg_default
  13066      template0  pg_default
      1      template1  pg_default
```

PostgreSQL stores ObjectIDs (OID) as folders

```
postgres@127f7d66d9cf:~/data/base$ cd 16384/
postgres@127f7d66d9cf:~/data/base/16384$ oid2name -d attask
From database "attask":
  Filenode                  Table Name
----------------------------------------
     16397          databasechangelog
     16403       databasechangeloglock
     16406     deleted_object_guids_1
```

sourcemind

PostgreSQL tablespaces

```
postgres@127f7d66d9cf:~/data$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  16384         attask  pg_default
  13067       postgres  pg_default
  13066      template0  pg_default
      1      template1  pg_default
```

- Objects are stored in PGDATA by default
- Using symbolic links, other storage location can be added: Tablespace
- While physically in another location, they appear to be in PGDATA for the PostgreSQL process

Managing database users

- Default user **`postgres`** has superuser privileges
- As a database admin (DBA) you can define roles, users, and groups
- Why create users with specific roles?

Setting up database with roles

```
SELECT rolname, rolsuper, rolvaliduntil FROM pg_roles;
```

```
          rolname          | rolsuper |     rolvaliduntil
---------------------------+----------+-----------------------
 pg_monitor                | f        |
 pg_read_all_settings      | f        |
 pg_read_all_stats         | f        |
 pg_stat_scan_tables       | f        |
 pg_read_server_files      | f        |
 pg_write_server_files     | f        |
 pg_execute_server_program | f        |
 pg_signal_backend         | f        |
 postgres                  | t        |
```

```
CREATE ROLE instructor
WITH PASSWORD '123' LOGIN
VALID UNTIL '2021-12-31 23:59:59';
```

```
CREATE DATABASE sourcemind WITH OWNER = instructor;
```

```
psql postgres://instructor@localhost:5433/sourcemind
```

Student and assistant roles

```
CREATE ROLE student WITH NOLOGIN;

CREATE ROLE vahe WITH LOGIN PASSWORD '123' IN ROLE student;


CREATE ROLE assistant WITH NOLOGIN;

GRANT assistant TO vahe;


CREATE ROLE john WITH LOGIN PASSWORD '123';

GRANT student TO john;


GRANT instructor TO vahe;

REVOKE instructor FROM vahe;
```

List roles overview

```
postgres-# \du
                                   List of roles
 Role name  |                         Attributes                         |      Member of
------------+------------------------------------------------------------+---------------------
 assistant  | Cannot login                                               | {}
 instructor | Password valid until 2021-12-31 23:59:59+00                | {}
 john       |                                                            | {student}
 postgres   | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 student    | Cannot login                                               | {}
 vahe       |                                                            | {student,assistant}
```

Managing connections with roles

- Try the following connections:

```
psql postgres://instructor@localhost:5433/sourcemind
psql postgres://john@localhost:5433/sourcemind
psql postgres://john@localhost:5433/postgres
```

- How to limit access to databases by roles?
- E.g. Do not allow connections from students to `postgres` database

GRANT privileges to roles

- Control access at database and tables using the **GRANT** keyword

```
GRANT CONNECT ON DATABASE database_name TO username;
GRANT USAGE ON SCHEMA schema_name TO username;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA schema_name TO username;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA schema_name TO username;
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA schema_name TO username;
GRANT ALL PRIVILEGES ON DATABASE database_name TO username;
```

Managing connections with roles

- Host-based access: `pg_hba.conf` file in `$PGDATA`

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
# "local" is for Unix domain socket connections only
local   all             all                                     trust
# IPv4 local connections:
host    all             all             127.0.0.1/32            trust
# IPv6 local connections:
host    all             all             ::1/128                 trust
```

Schema search_path

A variable **search_path** tells Postgres which schema to look for the objects used in SQL queries.

```
postgres=# SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)

postgres=# SET search_path TO
public,library;

postgres=# SHOW search_path;
   search_path
-----------------
 public, library
(1 row)
```

Homework 6

1. Explain what are “isolation levels” in a relational database
2. What can B-TREE indexes do that HASH indexes cannot?
3. Run a Postgres server Docker image on your local machine and connect to it using **`psql`** tool.

Create a database named **`Homework`**. Add the following roles with the privileges:

- **`admin`**, can connect and create tables, as well as read and write data
- **`viewer`**, can connect and only read data from existing tables
- **`restricted`**, cannot connect at all.

Write all the commands that complete the task in a `.sql` file.