Accessing Data from Program Code

Exercise 01 – Postgres architecture and cloud provisioning

Reminder

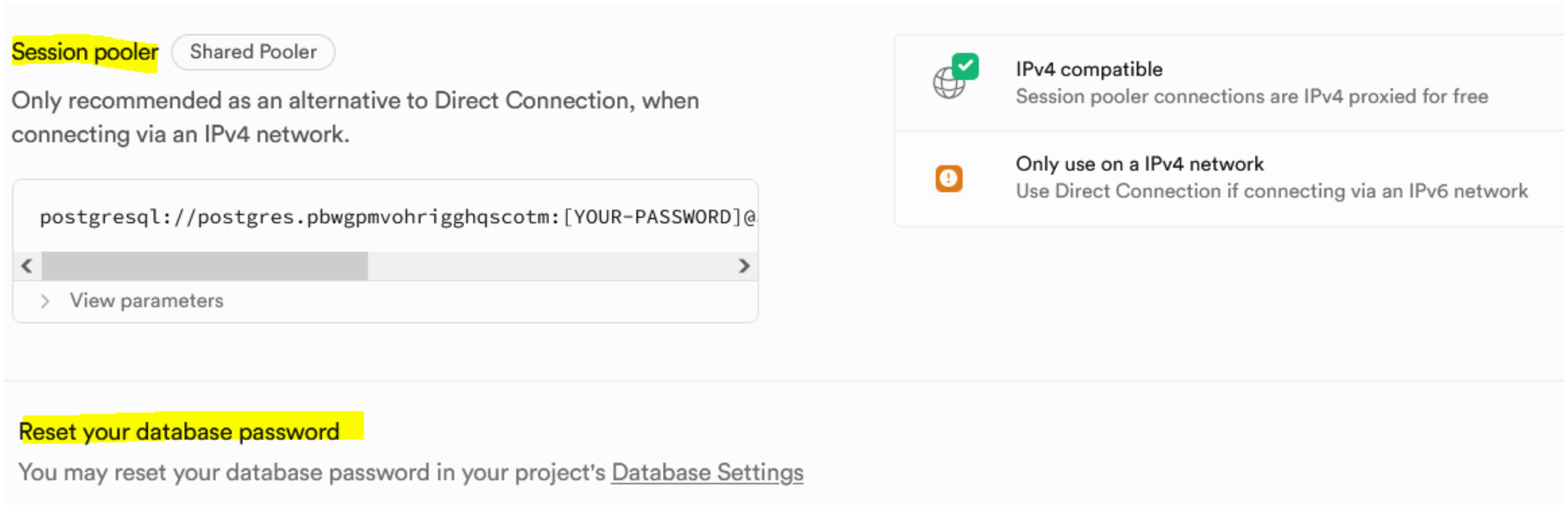
- After the introductory lecture, you should have created the account on Supabase and explored basic functionalities such as connecting to the database
- Project task was discussed and explained in the introductory lecture and formally defined in **ADPC-Project-Task.pdf** document on IE
    - if you have any further questions, you can contact me after the class or on mail/Teams

Introduction to Postgres

- Open-source relational database
  - Reliable
  - Rich ecosystem of extensions
- PostgreSQL dialect !
  - Slightly different to what you are used to with MSSQL (T-SQL)
  - Capable
- Newest versifunctionalitieson: **18**
  - 2025-09-25

Setting up Postgres cloud instance

- Follow the steps that will be described in the exercise and available on the IE afterwards
- Make sure to use `Session Pooler` and to set `postgres` user password



Session pooler

Only recommended as an alternative to Direct Connection, when connecting via an IPv4 network.

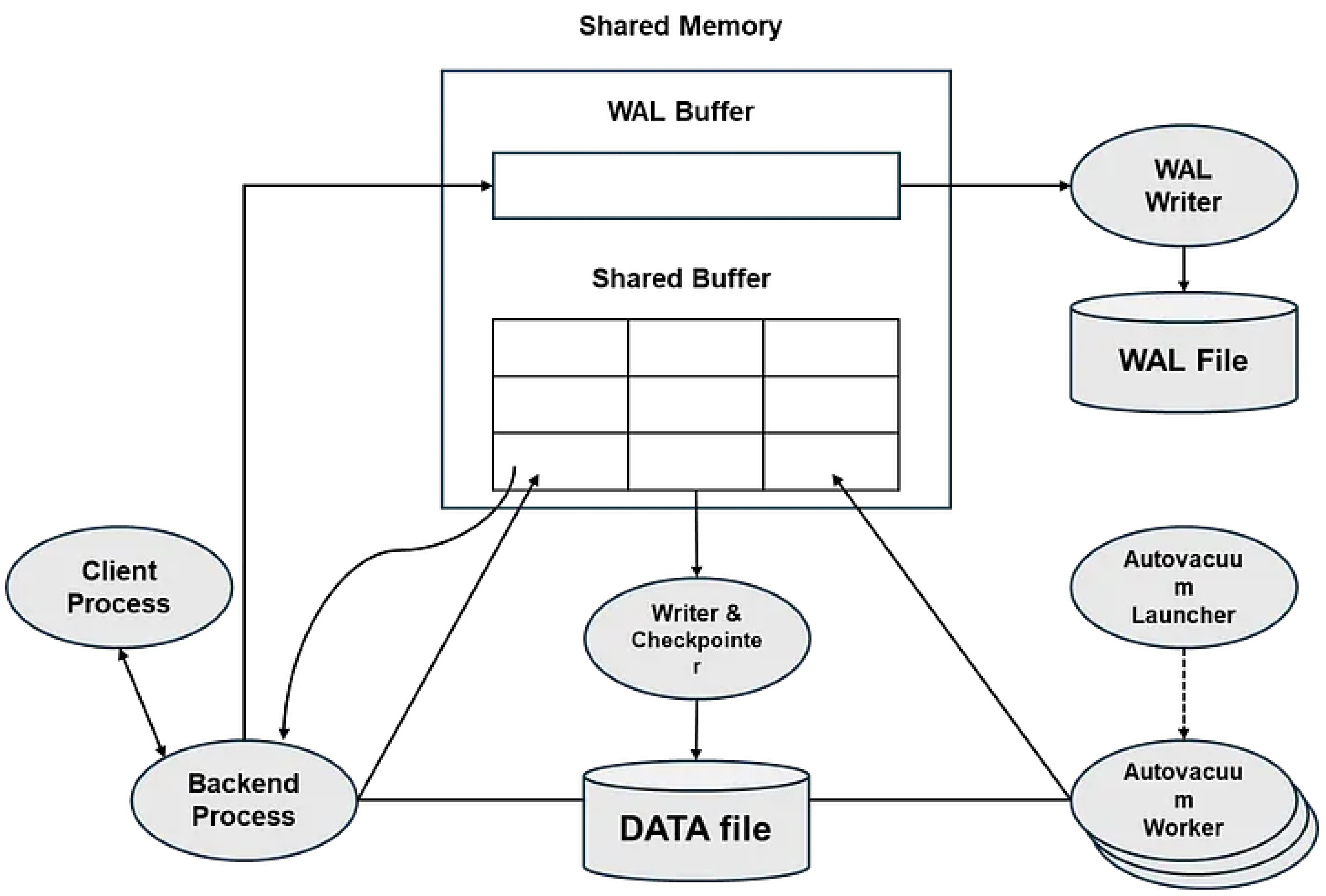
```
postgresql://postgres.pbwgpmvohrigghqscotm:[YOUR-PASSWORD]@
```

IPv4 compatible
Session pooler connections are IPv4 proxied for free

Only use on a IPv4 network
Use Direct Connection if connecting via an IPv6 network

Reset your database password
You may reset your database password in your project's Database Settings

Postgres Architecture



Postgres processes

- Postgres is a host to four main types of processes:
  - **Postmaster (Daemon) Process**
  - **Background Process**
  - Backend Process
  - Client Process

Postgres Architecture

- Postmaster (postgres)
  - the main server process, responsible for managing child processes.
- Client connection process (Backend process)
  - each client connection gets its own backend process (forked by Postmaster)
- Background processes:
  - **WAL Writer** → writes Write-Ahead Logs
  - Background Writer → flushes dirty pages to disk
  - Checkpointer → ensures data consistency by writing data periodically
  - **Autovacuum Launcher** → handles cleanup & vacuuming of dead tuples
  - Archiver (if archiving enabled)
  - Statistics Collector (tracks table/column usage)

WAL

- **Write Ahead Logging**
  - Data Integrity !
  - *WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, that is, after WAL records describing the changes have been flushed to permanent storage. If we follow this procedure, we do not need to flush data pages to disk on every transaction commit, because we know that in the event of a crash we will be able to recover the database using the log: any changes that have not been applied to the data pages can be redone from the WAL records. (This is roll-forward recovery, also known as REDO.)*
  - **significantly reduced number of disk writes**

ALGEBRA

WAL

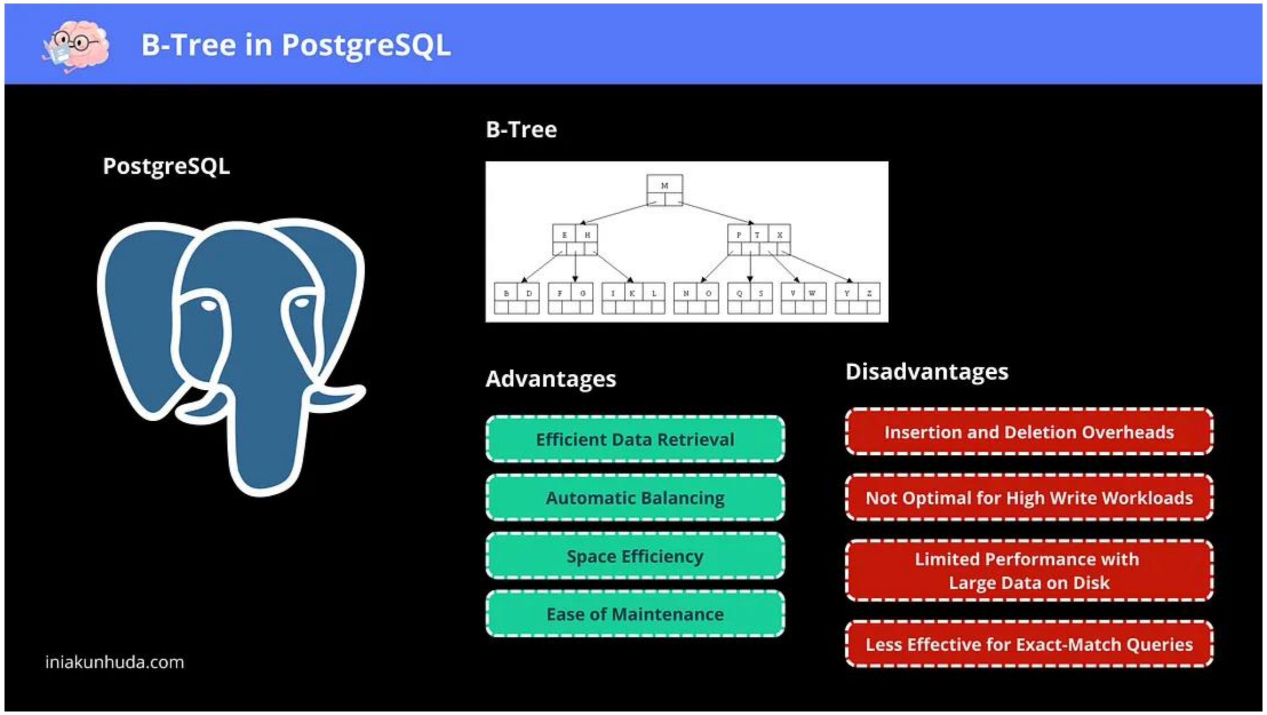
- At each **checkpoint**, all dirty buffers must be written to disk and WAL must be archived/truncated
  - checkpoints happen periodically – configurabl with `checkpoint_timeout` and `max_wal_size` or can be triggered manually
  - can set `synchronous_commit` and `commit_delay`
- Information about operations found in:
  - `pg_stat_bgwriter`
  - `pg_stat_wal`

Vacuum

- Updates or deletes in the table create new row versions, old ones remain until cleaned
- So called “dead tuples” still reside in memory after being marked as deleted
  - Similar to “soft delete” mechanism that we mentioned before
- Information can be found in **`pg_stat_user_tables`**
  - `last_autovacuum`
  - `last_vacuum`

B-Trees

- The choice of B-Tree is advantageous for
  - equality and range queries
  - sorting operations
  - data consistency and performance



B-Tree in PostgreSQL

PostgreSQL

B-Tree

Advantages
- Efficient Data Retrieval
- Automatic Balancing
- Space Efficiency
- Ease of Maintenance

Disadvantages
- Insertion and Deletion Overheads
- Not Optimal for High Write Workloads
- Limited Performance with Large Data on Disk
- Less Effective for Exact-Match Queries

iniakunhuda.com

ALGEBRA

B-Tree

- **Keys** represent indexed values, organized in ascending order within each node
- **Pointers** are links to child nodes or to the actual data if they are in leaf nodes
- Binary search is performed on keys within each node

```
CREATE INDEX index_name ON table_name (column_name);
```

Configuration files

- `postgresql.conf`
  - Main configuration file that defines parameters like memory, connections, logging, WAL, query tuning (shared_buffers, work_mem, max_connections, logging_collector)
- `pg_hba.conf`
  - controls client authentication by specifiying which users can connect, from where, using what method (md5, scram, trust, peer, etc.)
- `pg_ident.conf`
  - Used for username mapping between system users and PostgreSQL users (authentication methods like `ident`.)

Databases

- Upon executing `initdb`, three databases are created: `template0`, `template1`, and `postgres`
  - Cloud provided databases can have more premade databases
- `template0` and `template1` serve as template databases for creating user databases and include system catalog tables
- Immediately after executing `initdb`, two tablespaces are created: `pg_default` and `pg_global`
  - `pg_default` is located in **$PGDATA\base**
  - `pg_global` is located in **$PGDATA\global**

Tables

- Each table is associated with three files
    - One for storing table data, named after the table's OID
    - One for managing the table's free space, named `OID_fsm`
    - One for managing the visibility of table blocks, named `OID_vm`
- Indexes lack a vm file, thus consisting of only two files: `OID` and `OID_fsm`

Query execution example

```
SELECT * FROM users WHERE id = 10;
```

- **Parser**
  - detects table users, column id
- **Rewrite**
  - expands rules or views (if users is a view)
- **Planner considers:**
  - Sequential scan (if table is small)
  - Index scan (if id has index)
- **Execute**
  - uses best plan, fetches matching rows
- **Result**
  - rows returned to client

Examples

```
CREATE TABLE animal (
    id SERIAL PRIMARY KEY,
    heat_control TEXT
);

INSERT INTO animal (heat_control)
SELECT CASE
           WHEN random() < 0.75 THEN ‘endotherm’
           ELSE ‘ectotherm'
       END
FROM generate_series(1, 100000);
```

Execution planner

- ANALYZE will generate statistics in pg_stats
- Then use EXPLAIN to the query plan created by execution planner

```
SELECT attname, n_distinct, most_common_vals,
most_common_freqs

FROM pg_stats

WHERE tablename = 'animal’;


EXPLAIN SELECT * FROM animal WHERE heat_control =
'ectotherm’;
```

- Check the different plans before and after running ANALYZE

Noticeable specifics

- Sequence
- Dialect

Postgres Sequence

- Sequences are special single-row tables designed to generate unique values
  - created implicitly when you use `SERIAL/BIGSERIAL` or explicitly with `CREATE SEQUENCE`
  - backed by a counter stored in a catalog-managed relation and stored in pg_sequence system catalog
  - once incremented, the value is “lost” even if the transaction rolls back (ensures uniqueness)

Postgres Sequence

- Functions:
  - `nextval('seq’)`
    - increments and returns next value
  - `currval('seq’)`
    - last value used in this session
  - `setval('seq', N)`
    - set sequence to N
- Since PG 10+, identity columns (GENERATED ALWAYS AS IDENTITY) are preferred over SERIAL.

Cloud provisioning

- After you have created new Postgres instance, connect to this Postgres instance via DBeaver, DataGrip or VS Code extension
- Explore basics and underlying tables in the Postgres system schema
- Now write your own connection to the Postgres cloud instance by exploring system tables using Npgsql library!

Task #1

- Query and explore the following system databases:
  - `pg_database`
  - `pg_stat_database`
  - `pg_stats`
  - `pg_stat_user_tables`
  - `pg_stat_activity`
  - `information_schema.tables`

Task #2

- Using ADPC-Exercise-01-Seed.sql file from IE to populate the database
- Analyze the table definitions and content
- Retrieve the following information:
  - Top 5 students by average score
  - Most popular exams (by applications)
  - Pass rate (%)

Task #3

- Write a program that inserts two students inside a transaction
- Force an error on the second insert (e.g. duplicate primary key) and show that the first insert is rolled back

Task #4

- List all databases and then list all tables and their columns using
  - `pg_database`
    - `oid, datname`
  - `pg_stat_database`
    - `xact_commit`
    - `xact_rollback`

Task #5

- Create a table called `sensor_reading` that has three float fields: `humidity`, `temperature` and `AQI` alongside primary key
- Programmatically create a CSV file with 100000 rows of mock data for sensor reads
- Insert the data from CSV file using
  - `INSERT`
  - `COPY`

In the next week's episode...

- Docker setup !
- More interesting Postgres functionalities
  - Pivots and window functions
- Connecting to Postgres using C# - Npgsql