Postgres Workshop - Hands-on

Features shown during the hands-on session

- Process and Memory Architecture

- Database Cluster Data Directory Layout

- Creation the PostgreSQL Cluster

- Working with the utility psql

- Postgres Configuration

- Routine Maintenance Tasks

- Develop and tune the database application

Hand-on documentation

https://github.com/borysneselovskyi/postgres_workshop_documentation

Setup the demo env

Connect to your demo environment

- Connect to the terminal 1:
    - Run in the browser https://<your vm ip address>
    - Connect to the terminal as user **postgres** with the password **edb**
- Download the hans-on rom the GitHub:
    - Run:

        git clone https://github.com/borysneselovskyi/postgres_workshop.git

- Connect to the terminal 2:
    - Open the new browser tab.
    - Run https://<your vm ip address>
    - Connect to the terminal as user **postgres** with the password **edb**
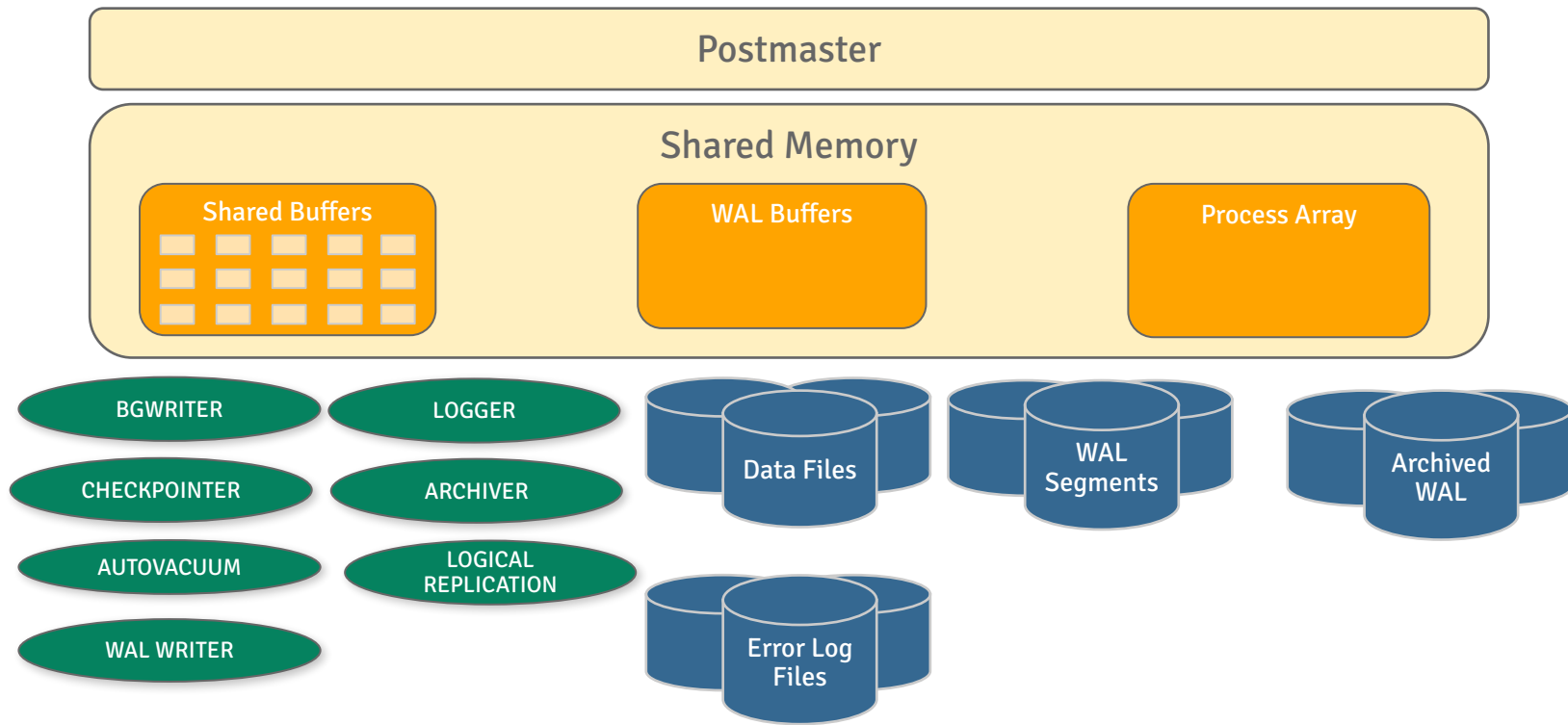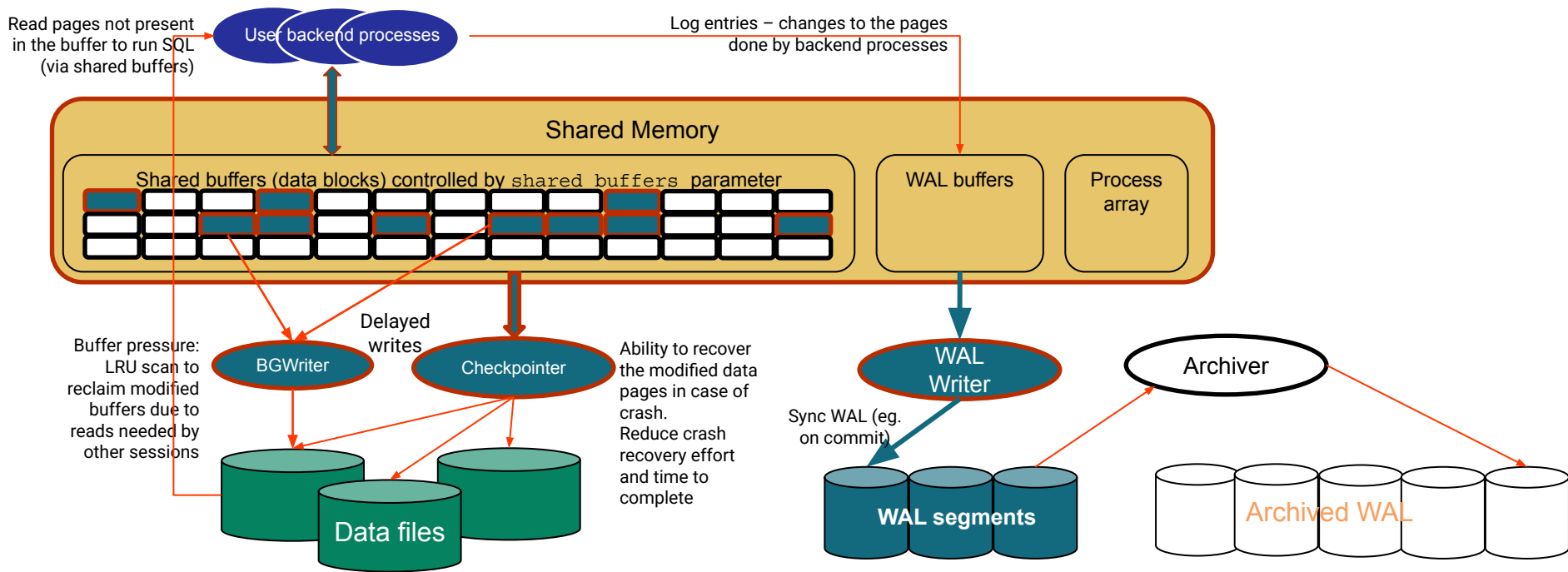
Getting started with Postgres

Objectives

- In this session we will:
  - learn about the Postgres architecture, processes and physical layout
  - create the postgres cluster using the utility initdb
  - start the postgres cluster using the utility pg_ctl
  - connect to (and disconnect from) the postgres cluster using the utility psql

Process and Memory Architecture

Key processes affecting disk and memory

Read pages not present in the buffer to run SQL (via shared buffers)

User backend processes

Log entries – changes to the pages done by backend processes

Shared Memory

Shared buffers (data blocks) controlled by `shared_buffers` parameter

WAL buffers

Process array

Buffer pressure: LRU scan to reclaim modified buffers due to reads needed by other sessions

Delayed writes

BGWriter

Checkpointer

Ability to recover the modified data pages in case of crash.
Reduce crash recovery effort and time to complete

Data files

WAL Writer

Sync WAL (eg. on commit)

WAL segments
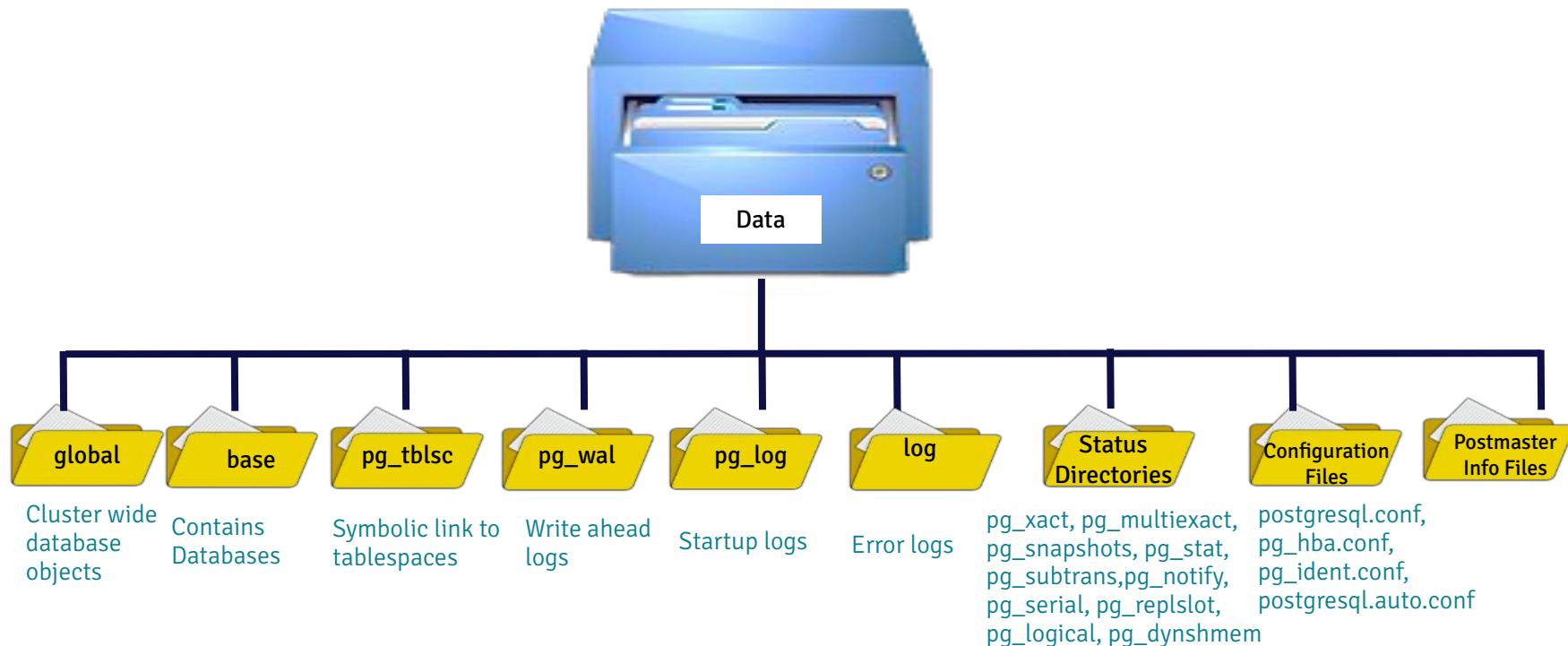
Archiver

Archived WAL

Physical Database Architecture

- Database cluster is a collection of databases managed by single server instance
- Each cluster has a separate
  - Data directory
  - TCP port
  - Set of processes
- A cluster can contain multiple databases
- Default Installation Directory Location:
  - Linux - /usr/lib/postgresql/17
    - bin - Programs
    - lib - Libraries
    - share - Shared data
- Default Data directory: /var/lib/pgsql/17/data
- Data directory used in this workshop: /opt/postgres/data

Database Cluster Data Directory Layout

Data

global
Cluster wide database objects

base
Contains Databases

pg_tblsc
Symbolic link to tablespaces

pg_wal
Write ahead logs

pg_log
Startup logs

log
Error logs

Status Directories
pg_xact, pg_multiexact, pg_snapshots, pg_stat, pg_subtrans, pg_notify, pg_serial, pg_replslot, pg_logical, pg_dynshmem

Configuration Files
postgresql.conf, pg_hba.conf, pg_ident.conf, postgresql.auto.conf

Postmaster Info Files

Lab Exercise - 1: Create the PostgreSQL Cluster

- In the terminal 1:

    - Go to the directory /var/lib/postgresql/postgres_workshop:

    cd /var/lib/postgresql/postgres_workshop

    - Create the postgres cluster - run the command:

    initdb -D /opt/postgres/data -U postgres -W

    Enter new superuser password (twice): **edb**

    - Start the postgres cluster:

    pg_ctl -D /opt/postgres/data -l logfile start

    - Verify the postgres cluster status:

    pg_ctl -D /opt/postgres/data status

Lab Exercise - 2: Verify the environment

- In the terminal 1:
  - Check the OS Processes:

    ps -ef | grep postgres | egrep 'postgres:|data'

  - Check the structure the Data directory (PGDATA):

    ls -al /opt/postgres/data

  - Check the environment variables inclusive PATH:

    cat $HOME/.bashrc | egrep 'PATH|PG'

Lab Exercise - 3: Connect to the postgres

- In the terminal 1:
    - Connect the postgres cluster via the tool psql:

        psql -h localhost -p 5432 -d postgres -U postgres

    - Check the databases in the cluster:

        \l

    - Disconnect from the cluster:

        \q

    - Run the sql script and create workshop database with some objects:

        cd $HOME/postgres_workshop

        psql -h localhost -p 5432 -d postgres -U postgres -f ./edbstore.sql

Summary

- In this session we:
  - learned about the Postgres architecture, processes and physical layout
  - created the postgres cluster using the utility initdb
  - learned about how to start the postgres cluster using the utility pg_ctl
  - connected to the postgres cluster using the utility psql

Working with the utility psql

Objectives

- In this session we will learn about psql utility:
  - Connect options
  - Listing of database objects
  - Working with SQL commands
  - Copy Command

Introduction to psql

- psql is a command line interface (CLI) to Postgres

- Can be used to execute SQL queries and psql meta commands

- psql has its own set of commands, all of which start with a backslash (\).

- Some commands accept a pattern. This pattern is a modified regex. Key points:

  - * and ? are wildcards

  - Double-quotes are used to specify an exact name, ignoring all special characters and preserving case

```
[postgres@pgsrv1 ~]$ psql -p 5432 -U postgres -d postgres
Password for user postgres:
psql (16.0)
Type "help" for help.

postgres=# \q
```

Connecting to a Database

psql Connection Options:

- -d <Database Name>
- -h <Hostname>
- -p <Database Port>
- -U <Database Username>

Environmental Variables

- PGDATABASE, PGHOST, PGPORT and PGUSER

psql: The primary CLI client

```
Usage:
  psql [OPTIONS]... [DBNAME [USERNAME]]

General options:
  -d DBNAME       specify database name to connect to (default: "oddbjorn")
  -c COMMAND      run only single command (SQL or internal) and exit
  -f FILENAME     execute commands from file, then exit
  -l              list available databases, then exit
  -v NAME=VALUE   set psql variable NAME to VALUE
  -X              do not read startup file (~/.psqlrc)
  --help          show this help, then exit
  --version       output version information, then exit

Input and output options:
  -a              echo all input from script
  -e              echo commands sent to server
  -E              display queries that internal commands generate
  -q              run quietly (no messages, only query output)
  -o FILENAME     send query results to file (or |pipe)
  -n              disable enhanced command line editing (readline)
  -s              single-step mode (confirm each query)
  -S              single-line mode (end of line terminates SQL command)

Output format options:
  -A              unaligned table output mode (-P format=unaligned)
  -H              HTML table output mode (-P format=html)
  -t              print rows only (-P tuples_only)
  -T TEXT         set HTML table tag attributes (width, border) (-P tableattr=)
  -x              turn on expanded table output (-P expanded)
  -P VAR[=ARG]    set printing option VAR to ARG (see \pset command)
  -F STRING       set field separator (default: "|") (-P fieldsep=)
  -R STRING       set record separator (default: newline) (-P recordsep=)

Connection options:
  -h HOSTNAME     database server host or socket directory (default: "local socket")
  -p PORT         database server port (default: "5432")
  -U NAME         database user name (default: "oddbjorn")
  -W              prompt for password (should happen automatically)
```

psql: \?: Listing the internal commands

General
```
\c[onnect] [DBNAME|- [USER]]
                 connect to new database
\cd [DIR]        change the current working directory
\copyright       show PostgreSQL usage and distribution terms
\encoding [ENCODING]
                 show or set client encoding
\h [NAME]        help on syntax of SQL commands, * for all commands
\q               quit psql
\set [NAME [VALUE]]
                 set internal variable, or list all if no parameters
\timing          toggle timing of commands (currently off)
\unset NAME      unset (delete) internal variable
\! [COMMAND]     execute command in shell or start interactive shell
```

Query Buffer
```
\e [FILE]        edit the query buffer (or file) with external editor
\g [FILE]        send query buffer to server (and results to file or
   |pipe)
\p               show the contents of the query buffer
\r               reset (clear) the query buffer
\s [FILE]        display history or save it to file
\w [FILE]        write query buffer to file
```

Input/Output
```
\echo [STRING]   write string to standard output
\i FILE          execute commands from file
\o [FILE]        send all query results to file or |pipe
\qecho [STRING]
                 write string to query output stream (see \o)
```

Informational
```
\d [NAME]        describe table, index, sequence, or view
\d{t|i|s|v|S} [PATTERN] (add "+" for more detail)
                 list tables/indexes/sequences/views/system tables
\da [PATTERN]    list aggregate functions
\dc [PATTERN]    list conversions
\dC              list casts
\dd [PATTERN]    show comment for object
\dD [PATTERN]    list domains
\df [PATTERN]    list functions (add "+" for more detail)
\dn [PATTERN]    list schemas
\do [NAME]       list operators
\dl              list large objects, same as \lo_list
\dp [PATTERN]    list table access privileges
\dT [PATTERN]    list data types (add "+" for more detail)
\du [PATTERN]    list users
\l               list all databases (add "+" for more detail)
\z [PATTERN]     list table access privileges (same as \dp)
```

Formatting
```
\a               toggle between unaligned and aligned output mode
\C [STRING]      set table title, or unset if none
\f [STRING]      show or set field separator for unaligned query output
\H               toggle HTML output mode (currently off)
\pset NAME [VALUE]
                 set table output option
                 (NAME := {format|border|expanded|fieldsep|footer|null|
                 recordsep|tuples_only|title|tableattr|pager})
\t               show only rows (currently off)
\T [STRING]      set HTML <table> tag attributes, or unset if none
\x               toggle expanded output (currently off)
```

Copy, Large Object
```
\copy ...        perform SQL COPY with data stream to the client host
\lo_export
\lo_import
\lo_list
\lo_unlink       large object operations
```

psql: \d: Describe

```
\d [NAME]          describe table, index, sequence, or view

\d{t|i|s|v|S} [PATTERN] (add "+" for more detail)
            list tables/indexes/sequences/views/system tables

\da [PATTERN]   list aggregate functions
\dc [PATTERN]   list conversions
\dC             list casts
\dd [PATTERN]   show comment for object
\dD [PATTERN]   list domains
\df [PATTERN]   list functions (add "+" for more detail)
\dn [PATTERN]   list schemas
\do [NAME]      list operators
\dl             list large objects, same as \lo_list
\dp [PATTERN]   list table access privileges
\dT [PATTERN]   list data types (add "+" for more detail)
\du [PATTERN]   list users
\l              list all databases (add "+" for more detail)
\z [PATTERN]    list table access privileges (same as \dp)
```

psql: \h: SQL-help

ABORT	CREATE LANGUAGE	DROP TYPE
ALTER AGGREGATE	CREATE OPERATOR CLASS	DROP USER
ALTER CONVERSION	CREATE OPERATOR	DROP VIEW
ALTER DATABASE	CREATE RULE	END
ALTER DOMAIN	CREATE SCHEMA	EXECUTE
ALTER FUNCTION	CREATE SEQUENCE	EXPLAIN
ALTER GROUP	CREATE TABLE	FETCH
ALTER LANGUAGE	CREATE TABLE AS	GRANT
ALTER OPERATOR CLASS	CREATE TRIGGER	INSERT
ALTER SCHEMA	CREATE TYPE	LISTEN
ALTER SEQUENCE	CREATE USER	LOAD
ALTER TABLE	CREATE VIEW	LOCK
ALTER TRIGGER	DEALLOCATE	MOVE
ALTER USER	DECLARE	NOTIFY
ANALYZE	DELETE	PREPARE
BEGIN	DROP AGGREGATE	REINDEX
CHECKPOINT	DROP CAST	RESET
CLOSE	DROP CONVERSION	REVOKE
CLUSTER	DROP DATABASE	ROLLBACK
COMMENT	DROP DOMAIN	SELECT
COMMIT	DROP FUNCTION	SELECT INTO
COPY	DROP GROUP	SET
CREATE AGGREGATE	DROP INDEX	SET CONSTRAINTS
CREATE CAST	DROP LANGUAGE	SET SESSION AUTHORIZATION
CREATE CONSTRAINT TRIGGER	DROP OPERATOR CLASS	SET TRANSACTION
CREATE CONVERSION	DROP OPERATOR	SHOW
CREATE DATABASE	DROP RULE	START TRANSACTION
CREATE DOMAIN	DROP SCHEMA	TRUNCATE
CREATE FUNCTION	DROP SEQUENCE	UNLISTEN
CREATE GROUP	DROP TABLE	UPDATE
CREATE INDEX	DROP TRIGGER	VACUUM

Lab Exercise - 4: Working with psql

- In the terminal 1:

  - Connect the postgres database postgres

    psql -h localhost -p 5432 -d postgres -U postgres

  - Check the connection information:

    \conninfo

  - Get help for the plsq information commands:

    \?

  - List installed extensions:

    \dx

Lab Exercise - 4: Working with psql

- In the terminal 1:
  - Install extension pg_stat_statements:

    create extension pg_stat_statements;

  - Check installed extensions again:

    \dx

  - List the Catalog tables and views:

    \dS

Lab Exercise - 4: Working with psql

- In the terminal 1:
  - Connect to the database edbstore with the user edbuser:

    \c edbstore edbuser

  - Check the connection information:

    \conninfo

  - List the tables:

    \dt

  - Get table definition of the table categories:

    \d categories

Lab Exercise - 4: Working with psql

▪ In the terminal 1:

- ○ Get the help for the SQL commands:

  \h

- ○ Get help for the Syntax of "CREATE TABLE":

  \h CREATE TABLE

- ○ Get content of the table categories:

  select * from categories;

  or

  table categories;

Lab Exercise - 4: Working with psql

- In the terminal 1:

  - Use internal functions:

    - Check the postgres version:

      select version();

    - Check the current connected user and database:

      select current_user;

      select current_database();

  - Disconnect:

    \q

Lab Exercise - 5: Use copy command

- In the terminal 1:
  - In this exercise we will put the data from the csv file into the postgres table:
    - Show the content of the csv file score.csv:

      cat $HOME/postgres_workshop/score.csv

    - Connect the postgres database postgres

      psql -h localhost -p 5432 -d postgres -U postgres

    - Create table footballranking:

      create table footballranking (team text, ranking text, played_games integer);

    - Insert data from the score.csv into the table footballranking:

      \copy footballranking from '/var/lib/postgresql/postgres_workshop/score.csv' delimiter ';';

    - Query the table:

      select * from footballranking;

Summary

- In this session we learned:
    - How to connect to postgres using psql
    - How to get information about database objects using psql
    - How to run SQL commands using psql
    - How to use the copy command using psql

Postgres Configuration

Objectives

- In this session we will learn about how to change the settings for the postgres on several levels:
  - Session
  - Database
  - Cluster

Setting Server Parameters

- There are many configuration parameters that affect the behaviour of the database system

- All parameter names are case-insensitive

- Every parameter takes a value of one of five types:

    - boolean

    - integer

    - floating point

    - string

    - enum

- One way to set these parameters is to edit the file **postgresql.conf**, which is normally kept in the data directory

The Server Parameter File - postgresql.conf

- Holds parameters used by a cluster

- Parameters are case-insensitive

- Normally stored in data directory

- **`initdb`** installs default copy

- Some parameters only take effect on server restart **(`pg_ctl restart`**)

- **`#`** used for comments

- One parameter per line

- Use include directive to read and process another file

- Can also be set using the command-line option

Viewing and Changing Server Parameters

Configuration parameters can be viewed using:
- SHOW command
- pg_settings
- pg_file_settings

Configuration parameters can be modified for:
- Single session using the SET command
- Database user using ALTER USER
- Single database using ALTER DATABASE

Changing Configuration Parameter at Cluster Level

```
[postgres@pgsrv1 ~] psql edb postgres

edb=# ALTER SYSTEM SET work_mem=20480;
ALTER SYSTEM

edb=# SELECT pg_reload_conf();

edb=# ALTER SYSTEM RESET work_mem;
ALTER SYSTEM

edb=# SELECT pg_reload_conf();
```

Use ALTER SYSTEM command to edit cluster level settings without editing **postgresql.conf**

`ALTER SYSTEM` writes new setting to **postgresql.auto.conf** file which is read at last during server reload/restarts

Parameters can be modified using `ALTER SYSTEM` when required

Lab Exercise - 6: Change parameter work_mem - Session

- In the terminal 1:
  - Connect the postgres database postgres

    psql -h localhost -p 5432 -d postgres -U postgres

  - Show settings for the parameter work_mem:

    show work_mem;

  - Change work_mem, set the value to "8MB";:

    set work_mem="8MB";

  - Show settings for the parameter work_mem:

    show work_mem;

  - Exit the psql:

    \q

Lab Exercise - 6: Change parameter work_mem - Cluster

- Open the new browser window:
  - Connect the postgres database postgres again

    psql -h localhost -p 5432 -d postgres -U postgres

  - Show settings for the parameter work_mem:

    show work_mem;

    We can see, the value is "4MB"

  - Change work_mem on the cluster level and reload the configuration:

    alter system set work_mem="8MB";

  - Show settings for the parameter work_mem (Change is not visible):

    show work_mem;

  - Reload configuration and show settings again:

    select pg_reload_conf();

    show work_mem;

Lab Exercise - 6: Change parameter work_mem - Cluster

- Open the new browser window:

  ○ Exit the psql:

     \q

  ○ Connect the postgres database postgres again

     psql -h localhost -p 5432 -d postgres -U postgres

  ○ Show settings for the parameter work_mem:

     show work_mem;

     We can see, the value is "8MB"

Summary

- In this session we learned about how to change the settings for the postgres on several levels:
  - Session
  - Database
  - Cluster

Routine Maintenance Tasks

Objectives

- In this session we will learn about how to
    - Updating Optimizer Statistics
    - Handling Data Fragmentation using Routine Vacuuming
    - Preventing Transaction ID Wraparound Failures
    - Automatic Maintenance using Autovacuum

Database Maintenance

- Data files become fragmented as data is modified and deleted

- Database maintenance helps reconstruct the data files

- If done on time nobody notices but when not done everyone knows

- Must be done before you need it

- Improves performance of the database

- Saves database from transaction ID wraparound failures

Maintenance Tools

- Maintenance thresholds can be configured using the pgAdmin Client

- Postgres maintenance thresholds can be configured in postgresql.conf

- Manual scripts can be written watch stat tables like `pg_stat_user_tables`

- Maintenance commands:

    - **`ANALYZE`**

    - **`VACUUM`**

    - **`CLUSTER`**

- Maintenance command vacuumdb can be run from OS prompt

- Autovacuum can help in automatic database maintenance

Optimizer Statistics

- Optimizer statistics play a vital role in query planning

- Not updated in real time

- Collects information for relations including size, row counts, average row size and row sampling

- Stored permanently in catalog tables

- The maintenance command `ANALYZE` updates the statistics

Lab Exercise - 7: Updating statistics

- In the terminal 1:
  - Connect the postgres database postgres again

    psql -h localhost -p 5432 -d postgres -U postgres

  - Create table testanalyze and insert 10.000 rows:

    create table testanalyze(id integer, name varchar);

    insert into testanalyze values(generate_series(1,10000), 'Sample');

  - Check the postgres data dictionary to view the current statistics for the table:

    select relname, reltuples from pg_class where relname = 'testanalyze';

  - Update statistics using the command ANALYSE:

    analyze testanalyze ;

  - Check the postgres data dictionary to view the current statistics for the table:

    select relname, reltuples from pg_class where relname = 'testanalyze';

Data Fragmentation and Bloat

- Data is stored in data file pages

- An update or delete of a row does not immediately remove the row from the disk page

- Eventually this row space becomes obsolete and causes fragmentation and bloating

Routine Vacuuming

- Obsoleted rows can be removed or reused using vacuuming

- Helps in shrinking data file size when required

- Vacuuming can be automated using autovacuum

- The `VACUUM` command locks tables in access exclusive mode

- Long running transactions may block vacuuming, thus it should be done during low usage times

Vacuuming Commands

- When executed, the **`VACUUM`** command:

  - Can recover or reuse disk space occupied by obsolete rows

  - Updates data statistics

  - Updates the visibility map, which speeds up index-only scans

  - Protects against loss of very old data due to transaction ID wraparound

- The **`VACUUM`** command can be run in two modes:

  - **`VACUUM`**

  - **`VACUUM FULL`**

Vacuum and Vacuum Full

- **`VACUUM`**

    - Removes dead rows and marks the space available for future reuse

    - Does not return the space to the operating system

    - Space is reclaimed if obsolete rows are at the end of a table

- **`VACUUM FULL`**

    - More aggressive algorithm compared to **`VACUUM`**

    - Compacts tables by writing a complete new version of the table file with no dead space

    - Takes more time

    - Requires extra disk space for the new copy of the table, until the operation completes

Lab Exercise - 7: Vacuuming

- In the terminal 1:
  - Connect the postgres database postgres:

    psql -h localhost -p 5432 -d postgres -U postgres

  - Create table key_value and insert 10.000 rows:

    create table key_value (key bigint primary key, value bigint);

    insert into key_value select i, i FROM generate_series(10, 10000) AS t(i);

  - Check the table size:

    select pg_size_pretty(pg_total_relation_size('key_value')) as table_size;

  - Update 7500 rows in the table:

    update key_value set value = value * 2 where key > 2500;

  - Check the table size:

    select pg_size_pretty(pg_total_relation_size('key_value')) as table_size;

Lab Exercise - 7: Vacuuming (continued)

○ Run vacuum for the table:

vacuum key_value;

○ Check the table size again:

select pg_size_pretty(pg_total_relation_size('key_value')) as table_size;

We can see, the table size is not changed

○ Run vacuum **full** for the table::

vacuum full key_value;

○ Check the table size:

select pg_size_pretty(pg_total_relation_size('key_value')) as table_size;

The space in the table is reclaimed

Lab Exercise - 8: Show settings for the autovacuuming

- In the terminal 1:
  - Connect the postgres database postgres:

    psql -h localhost -p 5432 -d postgres -U postgres

  - Show settings for the parameter autovacuum:

    show autovacuum;

  - Show settings for the parameter autovacuum_max_workers:

    show autovacuum_max_workers;

  - Show settings for the parameter autovacuum_vacuum_threshold:

    show autovacuum_vacuum_threshold;

  - Show settings for the parameter autovacuum_analyze_threshold:

    show autovacuum_analyze_threshold;

Working with Postgres database

For this part
we may use the
following link on
the GitHub

https://github.com/borysneselovskyi/postgres_workshop/tree/main/excersises

Before we begin

- See: https://github.com/borysneselovskyi/postgres_workshop/tree/main/excersises#1-before-we-begin

- Go to the lab terminal session connected as OS user postgres and perform:

```
cd $HOME/postgres_workshop/

ln -s excersises exersises      # fix misspelled directory name

cd exercises

chmod 0600 *

chmod 0700 *.sh
```

- **Note: All the exercises scripts are intended run from $HOME/postgres_workshop/exercises directory**

Setting up dev01 database and the role (user) u1

- In the lab terminal session connected as OS user postgres:

    ```
    cd $HOME/postgres_workshop/exersises
    ```

- Invoke psql (connecting to postgres database user) to run creation script

    ```
    psql -f 00-setup.sql
    ```

- Verify you can connect to the dev01 Postgres database as u1 by running

    ```
    ./u1-psql.sh
    ```

- Exit the psql session (type \q at the prompt)

Preload pg_stat_statements extension shared library

- In the lab terminal session connected as OS user postgres:

  ```
  cd /opt/postgres/data

  cp postgresql.conf postgresql.conf.backup

  vi postgresql.conf   # alternatively use nano editor
  ```

- Find and edit **shared_preload_libraries** parameter, uncomment it and edit as below:

  ```
  shared_preload_libraries = 'pg_stat_statements'
  ```

- Save **postgresql.conf**, exit the editor and restart the database

  ```
  cd /var/lib/postgresql

  pg_ctl -D /opt/postgres/data stop

  pg_ctl -D /opt/postgres/data -l logfile start

  cd $HOME/postgres_workshop/exercises
  ```

Make sure that pg_stat_statements library is preloaded

- Invoke psql (connecting to postgres database user)

  ```
  psql
  ```

- Run the following command

  ```
  SHOW shared_preload_libraries;
  ```

- It needs to show the string containing `pg_stat_statements`

Exercise #1: Introduction

- There is a performance degradation on the production database
- Over a time since go-live, a simple query executes longer and longer
- Caught by application timeouts or in Postgres log showing long-running queries
- The query text (Java)

```
SELECT * FROM s1.tst_bind_bigint WHERE ext_id = ?
```

- The table definition is

```
\d+ s1.tst_bind_bigint

        Table "s1.tst_bind_bigint"
 Column   |  Type   | Nullable | Storage
---------+---------+----------+----------
 id       | bigint  | not null | plain
 ext_id   | bigint  | not null | plain
 info     | text    | not null | extended

Indexes:
  "tst_bind_bigint_pk" PRIMARY KEY btree (id)
  "tst_bind_bigint_i1"        btree (ext_id)
```

Exercise #1: Step 1

- Create a testcase schema (s1) and load some data (smaller set than in production)

  ```
  ./01-create-schema.sh
  ```

- The script calls the SQL code, see on the lab machine or use browser to see at GitHub

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/01-create-schema.sql

Exercise #1: Step 2

- Review the test-case application code: TableAccess.java

  - See at the lab machine in exercises directory, OR use GitHub link

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/TableAccess.java

- Run test application invocation for ext_id=12345

```
./02-app-behavior.sh  # this invokes: java TableAccess 12345
```

- What is the execution time?

- Is this normal according to the experience? If not, why?

Exercise #1: Step 3

- Compare it with the execution in psql (03-query-psql.sql)

  - See at the lab machine in exercises directory, OR use GitHub link:

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/03-query-psql.sql

- Run test application invocation for ext_id=12345

  ```
  ./03-query-psql.sh
  ## runs: SELECT * FROM s1.tst_bind_bigint WHERE ext_id = 12345;
  ```

- What is the difference with running it from Java program?

- What could be the reasons?

Exercise #1: Step 4 and 5

- Try with a different parameter (44213) in an opposite direction - psql first, Java next

```
./04-query-psql.sh       # uses ext_id = 44213
./05-app-behavior.sh     # invokes: java TableAccess 44213
```

- Any difference?

- Any other hypothesis?

Exercise #1: Step 6

- See the query execution planner and runtime data (06-explain-analyze.sql)

  ○ See at the lab machine in exercises directory, OR use GitHub link:

  ○ https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/06-explain-analyze.sql

  ```
  ./06-explain-analyze.sh
  ```

- What plan is used?

- Does it use the index?

- Any progress with explanations and remediations?

Exercise #1: Step 7

- Inspect pg_stat_statements for query stats (07-pg-stat-statements.sql)

  - See at the lab machine in exercises directory, OR use GitHub link:

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/07-pg-stat-statements.sql

```
./07-pg-stat-statements.sh
```

- What stands out?

- Is the application's query using the index?

- How does the Java bind variable map to Postgres type? See TableAccess.java again?

Exercise #1: Step 8

- Reproduce in psql with the application's bind-variable type (08-gotit-explain-analyze.sql)

  - See at the lab machine in exercises directory, OR use GitHub link:

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/08-gotit-explain-analyze.sql

  ```
  ./08-gotit-explain-analyze.sh
  ```

- How does the SQL planner behave now?

Exercise #1: Step 9

- Demonstrate the recommended fix in the application code (ExplainTableBigInt.java)

  - See at the lab machine in exercises directory, OR use GitHub link:

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/ExplainTableBigInt.java

```
./09-difference.sh
```

Exercise #1: Step 10

- What if the application code fix will not come soon and the performance problem is more and more painful?

- Workaround at the database side - an extra function based index (10-hotfix-function-index.sql)

  ○ See at the lab machine in exercises directory, OR use GitHub link:

  ○ https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/10-hotfix-function-index.sql

  ```
  ./10-hotfix-function-index.sh
  ```

- Why we may consider it rather as a temporary workaround than the permanent fix?

Exercise #1: Step 11

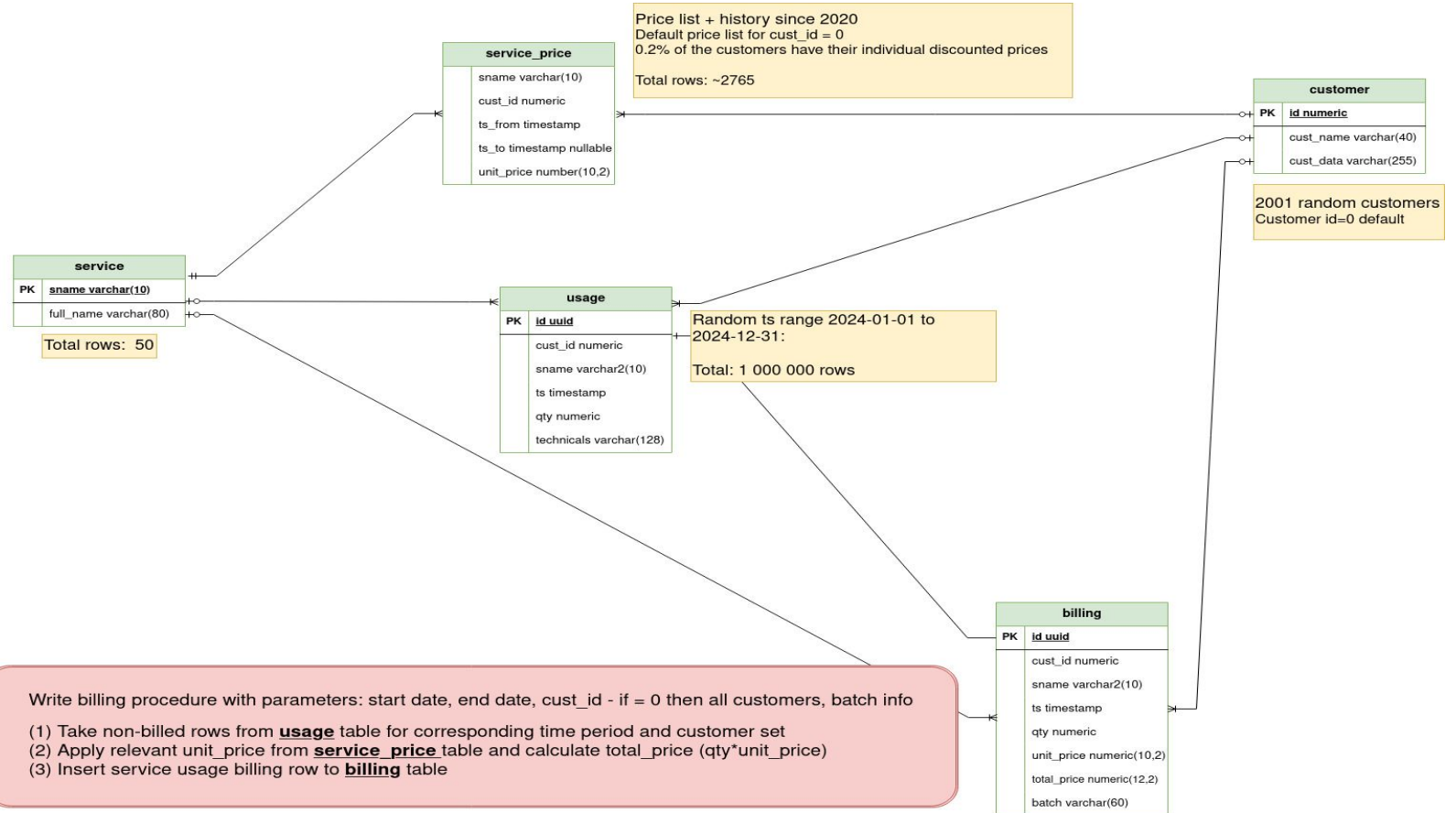- Look at the query behavior after applying the quick fix (workaround)

  ```
  ./02-app-behavior.sh
  ./07-pg-stat-statements.sh
  ./09-difference.sh
  ```

Exercise #2: Introduction

service_price

sname varchar(10)	
cust_id numeric	
ts_from timestamp	
ts_to timestamp nullable	
unit_price number(10,2)	

Price list + history since 2020
Default price list for cust_id = 0
0.2% of the customers have their individual discounted prices

Total rows: ~2765

customer

PK	**id numeric**
	cust_name varchar(40)
	cust_data varchar(255)

2001 random customers
Customer id=0 default

service

PK	**sname varchar(10)**
	full_name varchar(80)

Total rows: 50

usage

PK	**id uuid**
	cust_id numeric
	sname varchar2(10)
	ts timestamp
	qty numeric
	technicals varchar(128)

Random ts range 2024-01-01 to 2024-12-31:

Total: 1 000 000 rows

billing

PK	**id uuid**
	cust_id numeric
	sname varchar2(10)
	ts timestamp
	qty numeric
	unit_price numeric(10,2)
	total_price numeric(12,2)
	batch varchar(60)

Write billing procedure with parameters: start date, end date, cust_id - if = 0 then all customers, batch info

(1) Take non-billed rows from __usage__ table for corresponding time period and customer set
(2) Apply relevant unit_price from __service_price__ table and calculate total_price (qty*unit_price)
(3) Insert service usage billing row to __billing__ table

Exercise #2: Step 1

- Create tst schema, objects and load the data (30-create-schema.sql)

  - See at the lab machine in exercises directory, OR use GitHub link:

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/30-create-schema.sql

  - How the precision of partition keys based on timestamp data type is handled?

  ```
  ./30-create-schema.sh
  ```

- Verify objects and row count following Step 1
  https://github.com/borysneselovskyi/postgres_workshop/tree/main/excersises#3-exercise-2---batch-process-exercise

Exercise #2: Step 1

- Create tst schema, objects and load the data (30-create-schema.sql)

  - See at the lab machine in exercises directory, OR use GitHub link:

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/30-create-schema.sql

  - How the precision of partition keys based on timestamp data type is handled?

```
./30-create-schema.sh
```

- Verify objects and row count following Step 1
  https://github.com/borysneselovskyi/postgres_workshop/tree/main/excersises#3-exercise-2---batch-process-exercise

Exercise #2: Step 2

- Let's go through classical procedure implementation (31-create-proc-billing.sql)

  - See at the lab machine in exercises directory, OR use GitHub link:

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/31-create-proc-billing.sql

```
./31-create-proc-billing.sh
```

Exercise #2: Step 3

- Let's run the first procedure (32-exec-proc-billing.sql)

  - See at the lab machine in exercises directory, OR use GitHub link:

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/32-exec-proc-billing.sql

```
./32-exec-proc-billing.sh
# psql runs 32-exec-proc-billing.sql which:
# truncates tst.billing table, sets the \timing on and run
# call tst.proc_billing (date '2024-01-01', date '2024-12-31', 0);
```

- What is its execution time? Please write it down

- How can it be improved?

Exercise #2: Step 4

- Let's check one of the concepts: no lookup to tst.service_price table, use a fixed cost

- Indicate the performance limits of reading `tst.usage` and write to `tst.billing`

- Comment out the tst.service_price looku`tst.proc_billing0`p and create a new procedure (33-create-proc-null-billing.sql)

  ○ See at the lab machine in exercises directory, OR use GitHub link:

  ○ https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/33-create-proc-null-billing.sql

```
./33-create-proc-null-billing.sh
```

Exercise #2: Step 5

- Let's run "null-billing" procedure `tst.proc_billing0` (34-exec-proc-null-billing.sql)

  - See at the lab machine in exercises directory, OR use GitHub link:

  - https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/34-exec-proc-null-billing.sql

```
./34-exec-proc-null-billing.sh
# psql runs 34-exec-proc-null-billing.sql which:
# truncates tst.billing table, sets the \timing on and run
# call tst.proc_billing0 (date '2024-01-01', date '2024-12-31', 0);
```

- What is its execution time? Please write it down and compare with the time from Step 3

- Any thoughts?

Exercise #2: Step 6

- How to improve the code to be more performant BUT with the real processing?

- Cache lookup data in the procedure variable

- Option #1: Use the variable of HSTORE data type: procedure tst.proc_billing4_1
  (35-create-proc-opt1.sql)

  ○ See at the lab machine in exercises directory, OR use GitHub link:

  ○ https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/35-create-proc-opt1.sql

```
./35-create-proc-opt1.sh
```

Exercise #2: Step 7

- Let's run the procedure `tst.proc_billing4_1` using HSTORE variable to cache the relevant tst.service_price subset (36-exec-proc-billing-opt1.sql)

  ○ See at the lab machine in exercises directory, OR use GitHub link:

  ○ https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/36-exec-proc-billing-opt1.sql

```
./36-exec-proc-billing-opt1.sh
# psql runs 36-exec-proc-billing-opt1.sql which:
# truncates tst.billing table, sets the \timing on and run
# call tst.proc_billing4_1 (date '2024-01-01', date '2024-12-31', 0);
```

- What is its execution time? Please write it down and compare with the time from Step 3 and 5

Exercise #2: Step 8

- How to improve the code to be more performant BUT with the real processing - revisited ?

- Cache lookup data in the procedure variable in another way

- Option #2: Use the variable of JSONB data type: procedure tst.proc_billing5_4
  (37-create-proc-opt2.sql)

- Remark: Possible since Postgres 17

  ○ See at the lab machine in exercises directory, OR use GitHub link:

  ○ https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/37-create-proc-opt2.sql

```
./37-create-proc-opt2.sh
```

Exercise #2: Step 9

- Let's run the procedure `tst.proc_billing5_4` using JSONB variable to cache the relevant tst.service_price subset (38-exec-proc-billing-opt2.sql)

  ○ See at the lab machine in exercises directory, OR use GitHub link:

  ○ https://github.com/borysneselovskyi/postgres_workshop/blob/main/excersises/38-exec-proc-billing-opt2.sql

  ```
  ./38-exec-proc-billing-opt2.sh
  # psql runs 38-exec-proc-billing-opt2.sql which:
  # truncates tst.billing table, sets the \timing on and run
  # call tst.proc_billing5_4 (date '2024-01-01', date '2024-12-31', 0);
  ```

- What is its execution time? Please write it down and compare with the time from Step 3, 5 and 7

Exercise #2: Step 10

▪ Run procedure set for Q1 2024, not for full 2024

    ○ See at the lab machine in exercises directory, OR use GitHub workshop repo to see sh and sql scripts

```
./42-exec-proc-billing.sh          # As ./32-exec-proc-billing.sh, but for Q1 2024 only
./44-exec-proc-null-billing.sh     # As ./34-exec-proc-null-billing.sh, but for Q1 2024 only
./46-exec-proc-billing-opt1.sh     # As ./36-exec-proc-billing-opt1.sh, but for Q1 2024 only
./48-exec-proc-billing-opt2.sh     # As ./48-exec-proc-billing-opt2.sh, but for Q1 2024 only
```

▪ What are the execution times?