Homework 2 - Advanced Database Systems (ICS424)

Alfaifi, Ammar – 201855360

March 27, 2024

First, to setup the and install the DBMS, choosing PostgreSQL, on macOS, see

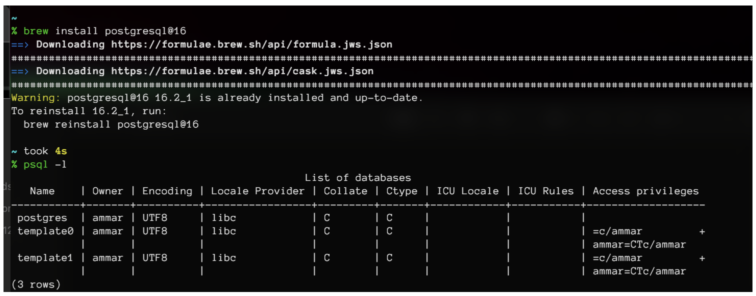


Figure 1: Install PostgreSQL package and checking it works by executing `psql -l`, to list all existing databases.

1 Startup and Shutdown Options and Procedures

Starting PostgreSQL on Linux/Unix-based systems as in Figure 2.

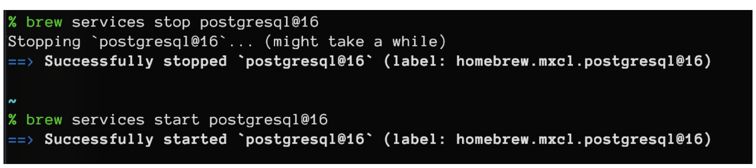


Figure 2: Start and stop PostgreSQL.

2 Logical and Physical Structures of a Database

The logical structure of the database is represented by the schema, which defines the tables, relationships, constraints, and other logical entities. The physical structure is how the data is stored on disk, which includes file organization, indexing, and storage parameters.

We can view the logical structure of the Horse database by running the following command:

```
\d
```

See Figure 3 for list of all tables of Horse DB, after I fixed all issues of compatiblity with PostgreSQL-16.

We can also inspect the physical storage parameters using the following command:

```sql
SELECT relname, relkind, relpages
FROM pg_catalog.pg_class
WHERE relkind IN ('r', 'i', 't');
```

```
postgres=# \d
          List of relations
 Schema |     Name     | Type  | Owner
--------+--------------+-------+-------
 public | horse        | table | ammar
 public | owner        | table | ammar
 public | owns         | table | ammar
 public | race         | table | ammar
 public | raceresults  | table | ammar
 public | stable       | table | ammar
 public | track        | table | ammar
 public | trainer      | table | ammar
(8 rows)
```

Figure 3: Listing all tables in the DB.

This query will display the names of relations (tables, indexes, and toast tables), their type, and the number of disk pages they occupy, see Figure 4.

```
                relname                 | relkind | relpages
----------------------------------------+---------+----------
 race_pkey                              | i       |        1
 track                                  | r       |        0
 race                                   | r       |        0
 raceresults_pkey                       | i       |        2
 raceresults                            | r       |        1
 stable_pkey                            | i       |        1
 pg_statistic                           | r       |       19
 pg_type                                | r       |       15
 horse_pkey                             | i       |        1
 stable                                 | r       |        0
 horse                                  | r       |        0
 owner_pkey                             | i       |        1
 owns_pkey                              | i       |        1
 owner                                  | r       |        0
 owns                                   | r       |        0
 trainer_pkey                           | i       |        1
 trainer                                | r       |        0
 track_pkey                             | i       |        1
```

Figure 4: The names of relations (tables, indexes, and toast tables), their type, and the number of disk pages they occupy.

3 Creating a Materialized View with a Complex Join

A materialized view is a pre-computed result set that is stored on disk for faster query performance. Here's an example of creating a materialized view that joins multiple tables:

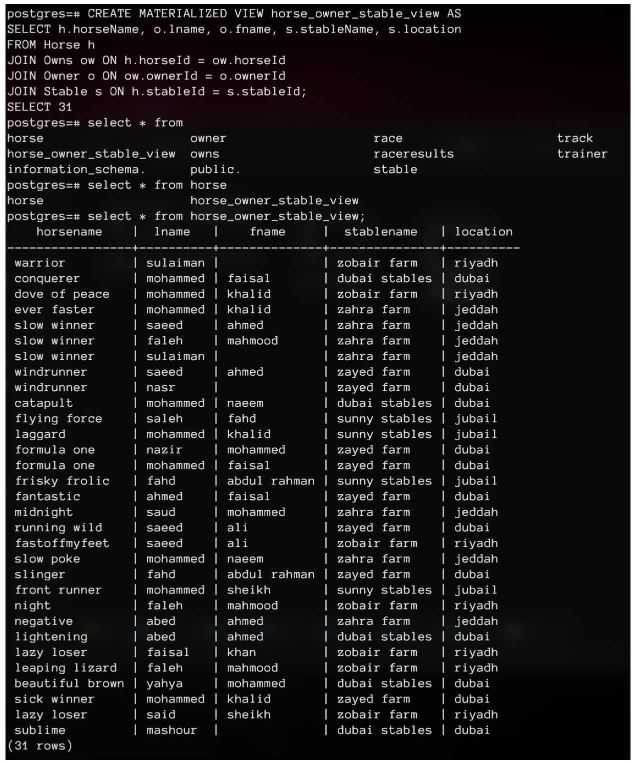
```sql
CREATE MATERIALIZED VIEW horse_owner_stable_view AS
SELECT h.horseName, o.lname, o.fname, s.stableName, s.location
FROM Horse h
JOIN Owns ow ON h.horseId = ow.horseId
JOIN Owner o ON ow.ownerId = o.ownerId
JOIN Stable s ON h.stableId = s.stableId;
```

This materialized view combines data from the Horse, Owns, Owner, and Stable tables to provide a consolidated view of horse names, owner names, and stable information. See Figure 5.

4 Developing and Demonstrating a Stored Procedure and a Trigger

To create a Stored Procedure we run this command:

```sql
CREATE OR REPLACE FUNCTION update_horse_age()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE Horse
    SET age = age + 1
    WHERE horseId = NEW.horseId;
    RETURN NEW;
```



```
postgres=# CREATE MATERIALIZED VIEW horse_owner_stable_view AS
SELECT h.horseName, o.lname, o.fname, s.stableName, s.location
FROM Horse h
JOIN Owns ow ON h.horseId = ow.horseId
JOIN Owner o ON ow.ownerId = o.ownerId
JOIN Stable s ON h.stableId = s.stableId;
SELECT 31
postgres=# select * from
horse                   owner                   race                    track
horse_owner_stable_view owns                    raceresults             trainer
information_schema.     public.                 stable
postgres=# select * from horse
horse                   horse_owner_stable_view
postgres=# select * from horse_owner_stable_view;
    horsename    |  lname   |    fname     |  stablename   | location
-----------------+----------+--------------+---------------+----------
 warrior         | sulaiman |              | zobair farm   | riyadh
 conquerer       | mohammed | faisal       | dubai stables | dubai
 dove of peace   | mohammed | khalid       | zobair farm   | riyadh
 ever faster     | mohammed | khalid       | zahra farm    | jeddah
 slow winner     | saeed    | ahmed        | zahra farm    | jeddah
 slow winner     | faleh    | mahmood      | zahra farm    | jeddah
 slow winner     | sulaiman |              | zahra farm    | jeddah
 windrunner      | saeed    | ahmed        | zayed farm    | dubai
 windrunner      | nasr     |              | zayed farm    | dubai
 catapult        | mohammed | naeem        | dubai stables | dubai
 flying force    | saleh    | fahd         | sunny stables | jubail
 laggard         | mohammed | khalid       | sunny stables | jubail
 formula one     | nazir    | mohammed     | zayed farm    | dubai
 formula one     | mohammed | faisal       | zayed farm    | dubai
 frisky frolic   | fahd     | abdul rahman | sunny stables | jubail
 fantastic       | ahmed    | faisal       | zayed farm    | dubai
 midnight        | saud     | mohammed     | zahra farm    | jeddah
 running wild    | saeed    | ali          | zayed farm    | dubai
 fastoffmyfeet   | saeed    | ali          | zobair farm   | riyadh
 slow poke       | mohammed | naeem        | zahra farm    | jeddah
 slinger         | fahd     | abdul rahman | zayed farm    | dubai
 front runner    | mohammed | sheikh       | sunny stables | jubail
 night           | faleh    | mahmood      | zobair farm   | riyadh
 negative        | abed     | ahmed        | zahra farm    | jeddah
 lightening      | abed     | ahmed        | dubai stables | dubai
 lazy loser      | faisal   | khan         | zobair farm   | riyadh
 leaping lizard  | faleh    | mahmood      | zobair farm   | riyadh
 beautiful brown | yahya    | mohammed     | dubai stables | dubai
 sick winner     | mohammed | khalid       | zayed farm    | dubai
 lazy loser      | said     | sheikh       | zobair farm   | riyadh
 sublime         | mashour  |              | dubai stables | dubai
(31 rows)
```

Figure 5: Creating a view and listing all of its contents.

```sql
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_horse_age_trigger
AFTER INSERT ON RaceResults
FOR EACH ROW
EXECUTE FUNCTION update_horse_age();
```

This stored procedure `update_horse_age()` is triggered after inserting a new row into the RaceResults table. It updates the age of the horse by incrementing it by 1 year. Now, to create a Trigger, we run instead this commmadn:

```sql
CREATE OR REPLACE FUNCTION prevent_duplicate_owners()
RETURNS TRIGGER AS $$
DECLARE
    owner_count INT;
BEGIN
    SELECT COUNT(*) INTO owner_count
    FROM Owns
    WHERE ownerId = NEW.ownerId AND horseId = NEW.horseId;
    IF owner_count > 0 THEN
        RAISE EXCEPTION 'Owner already owns this horse';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

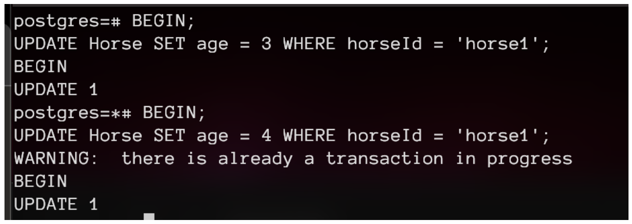


Figure 6: When we run two sessions we get a warning indicating there's an already running transaction in progress.

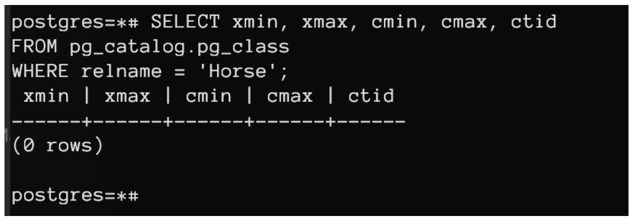


Figure 7: Here we see an empty list of the the timestamp table.

```
CREATE TRIGGER prevent_duplicate_owners_trigger
BEFORE INSERT ON Owns
FOR EACH ROW
EXECUTE FUNCTION prevent_duplicate_owners();
```

This trigger `prevent_duplicate_owners()` is executed before inserting a new row into the Owns table. It checks if the owner already owns the horse being inserted, and if so, it raises an exception to prevent the insertion.

5 Demonstrating Locking and Timestamping

PostgreSQL provides various locking mechanisms to ensure data integrity and concurrency control. We can observe locking by running two separate sessions and performing conflicting operations. For Session1:

```
BEGIN;
UPDATE Horse SET age = 3 WHERE horseId = 'horse1';
```

and for Session2:

```
BEGIN;
UPDATE Horse SET age = 4 WHERE horseId = 'horse1';
```

In Session 2, the UPDATE statement will be blocked until Session 1 commits or rolls back its transaction, demonstrating row-level locking. See Figure 6 for how we can do two transactions.

PostgreSQL also supports timestamping to track when data was last modified. We can view the timestamp of a table by running:

```
SELECT xmin, xmax, cmin, cmax, ctid
FROM pg_catalog.pg_class
WHERE relname = 'Horse';
```

See Figure 7 for the list of existing timestamps.

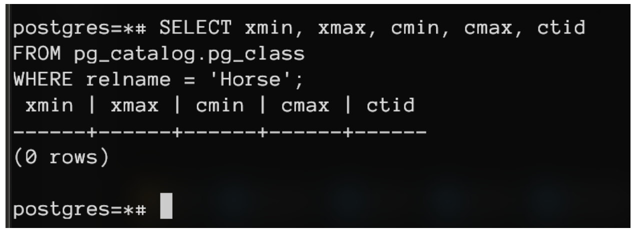


Figure 8: PostgreSQL lock table

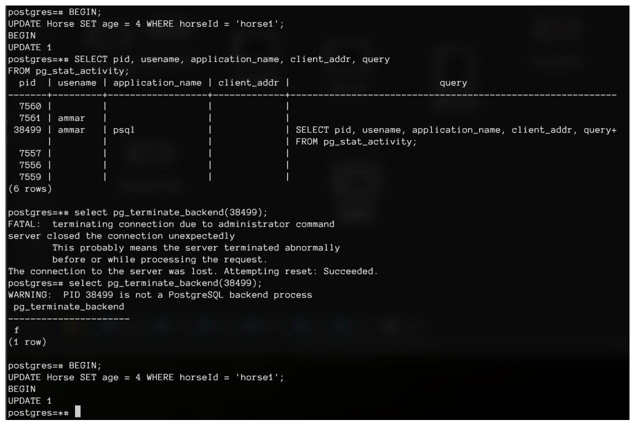


Figure 9: We searched up for a running process's PID and then terminate it.

6 Diagnosing and Resolving Locking Conflicts

To diagnose locking conflicts, We can use the `pg_locks` view, which provides information about outstanding locks in the database. For example:

```sql
SELECT l.locktype, l.relation::regclass, l.virtualtransaction, l.pid
FROM pg_locks l
JOIN pg_stat_activity a ON l.pid = a.pid;
```

This query displays the lock type, relation (table) name, virtual transaction ID, and process ID of the locks, as seen in Figure 8.

To resolve locking conflicts, we can either wait for the blocking transaction to complete or terminate the blocking process using the `pg_terminate_backend()` function. See Figure 9 for how we can terminate processes from their PID.

7 Evaluating Backup Options

PostgreSQL offers several backup options, including:

- **SQL Dump:** This creates a plain-text file containing SQL statements to recreate the database objects and data. We can create a SQL dump using the `pg_dump` utility.
- **File System Level Backup:** This involves creating a consistent backup of the PostgreSQL data directory using file system-level tools like tar or disk-level backup utilities.
- **Continuous Archiving and Point-in-Time Recovery (PITR):** This technique involves archiving Write-Ahead Log (WAL) segments, which can be used to restore the database to a specific point in time.

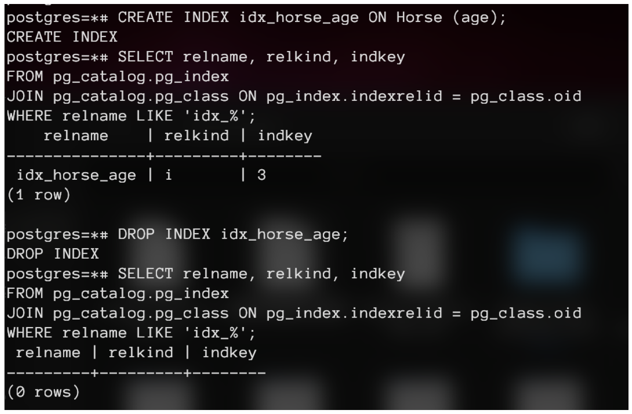


Figure 10: We created an index on `age` column, view it, and drop at the end as demonstration.

We can evaluate and choose the appropraite backup option based on our requirements, such as backup size, restore time, and the need for point-in-time recovery.

8 Recovering a Database

To recover a database from a backup, follow these general steps:

1. Stop the PostgreSQL server.
2. Replace the data directory with the backup files (for file system level backup) or restore the SQL dump (for SQL dump backup).
3. Start the PostgreSQL server.
4. If using PITR, restore the database from the archived WAL segments to the desired point in time.

9 Creating and Managing Indexes

Indexes improve query performance by allowing faster data retrieval. To create an index on a table column, use the CREATE INDEX statement:

```sql
CREATE INDEX idx_horse_age ON Horse (age);
```

This creates an index named `idx_horse_age` on the age column of the Horse table.

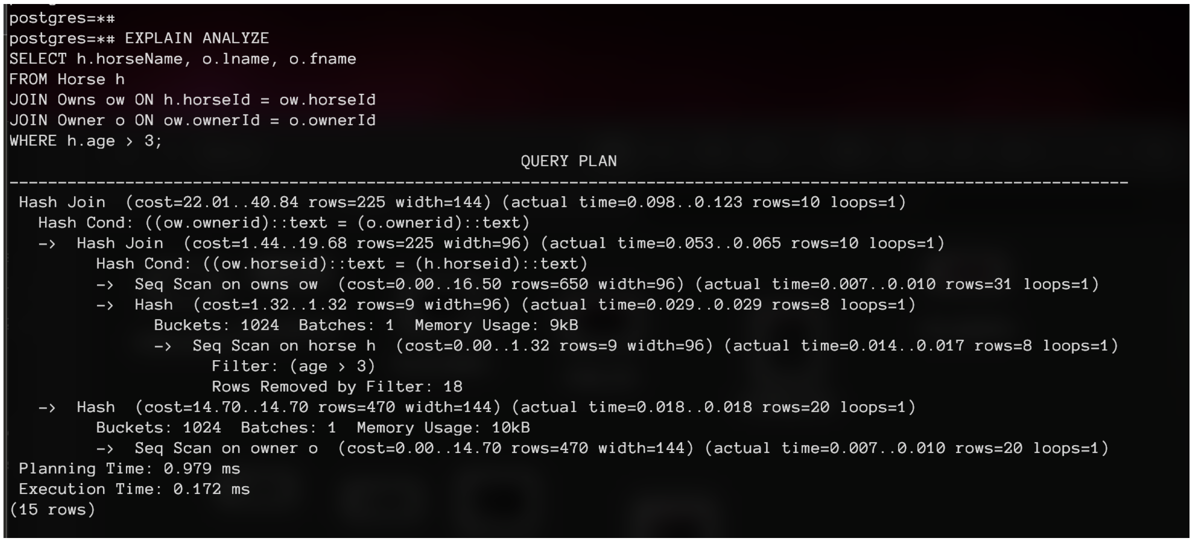
We can view existing indexes using the following query:

```sql
SELECT relname, relkind, indkey
FROM pg_catalog.pg_index
JOIN pg_catalog.pg_class ON pg_index.indexrelid = pg_class.oid
WHERE relname LIKE 'idx_%';
```

To drop an index, use the DROP INDEX statement:

```sql
DROP INDEX idx_horse_age;
```

See Figure 10, how apply all above in the `age` column in `Horse` table.



```
postgres=**
postgres=** EXPLAIN ANALYZE
SELECT h.horseName, o.lname, o.fname
FROM Horse h
JOIN Owns ow ON h.horseId = ow.horseId
JOIN Owner o ON ow.ownerId = o.ownerId
WHERE h.age > 3;
                                                     QUERY PLAN
----------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=22.01..40.84 rows=225 width=144) (actual time=0.098..0.123 rows=10 loops=1)
   Hash Cond: ((ow.ownerid)::text = (o.ownerid)::text)
   ->  Hash Join  (cost=1.44..19.68 rows=225 width=96) (actual time=0.053..0.065 rows=10 loops=1)
         Hash Cond: ((ow.horseid)::text = (h.horseid)::text)
         ->  Seq Scan on owns ow  (cost=0.00..16.50 rows=650 width=96) (actual time=0.007..0.010 rows=31 loops=1)
         ->  Hash  (cost=1.32..1.32 rows=9 width=96) (actual time=0.029..0.029 rows=8 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on horse h  (cost=0.00..1.32 rows=9 width=96) (actual time=0.014..0.017 rows=8 loops=1)
                     Filter: (age > 3)
                     Rows Removed by Filter: 18
   ->  Hash  (cost=14.70..14.70 rows=470 width=144) (actual time=0.018..0.018 rows=20 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 10kB
         ->  Seq Scan on owner o  (cost=0.00..14.70 rows=470 width=144) (actual time=0.007..0.010 rows=20 loops=1)
 Planning Time: 0.979 ms
 Execution Time: 0.172 ms
(15 rows)
```

Figure 11: An execution plan example.

10 Collecting and Analyzing Database Performance Information

PostgreSQL provides several tools and views for collecting and analyzing performance information, such as:

- `pg_stata_user_tables`: This view provides statistical information about tables, including the number of rows, disk space usage, and the last autovacuum and autoanalyze times.
- `pg_stat_user_indexes`: This view provides statistical information about indexes, including the number of index scans and the index size.
- `pg_stat_database`: This view provides statistics about database-level activity, such as the number of transactions, tuples read and written, and blocks read and written.
- `EXPLAIN` and `EXPLAIN ANALYZE`: These commands provide execution plans for SQL queries, including the estimated and actual costs, and information about the operations performed.

We can use these tools to analyze query performance, identify bottlenecks, and optimize our database for better efficiency.

Here's an example of using EXPLAIN ANALYZE to analyze the performance of a query:

```sql
EXPLAIN ANALYZE
SELECT h.horseName, o.lname, o.fname
FROM Horse h
JOIN Owns ow ON h.horseId = ow.horseId
JOIN Owner o ON ow.ownerId = o.ownerId
WHERE h.age > 3;
```

This will display the execution plan for the query, including the actual time and cost for each operation, providing insights into potential performance bottlenecks. See Figure 11, for the analysis of execution plan for the Listing 10.