Assignment 1

Task 1

```
                    List of relations
 Schema |         Name          |          Type          |  Owner
--------+-----------------------+------------------------+----------
 public | departments           | table                  | s4775476
 public | dept_emp              | table                  | s4775476
 public | dept_manager          | table                  | s4775476
 public | employees             | table                  | s4775476
 public | employees_public      | table                  | s4775476
 public | salaries              | table                  | s4775476
```

-- 1.1

```sql
--Task 1
--1.1
SELECT COUNT(*)
FROM  employees;
```

```
  count
--------
 300024
(1 row)
```

--1.2

```sql
--1.2
SELECT COUNT(DISTINCT emp_no)
FROM dept_emp
WHERE dept_no = (SELECT dept_no
                FROM departments
                WHERE dept_name = 'Marketing');
```

```
 count
-------
 20211
(1 row)
```

Task 2

--2.1

```sql
--2.1
CREATE TABLE IF NOT EXISTS salaries_horizontal
(
  emp_no int NOT NULL,
  salary int NOT NULL,
  from_date date NOT NULL,
  to_date date NOT NULL,
  PRIMARY KEY (emp_no, from_date),
  CONSTRAINT salaries_emp_no_fk FOREIGN KEY (emp_no) REFERENCES employees (emp_no)
) partition by range (from_date);

Create table IF NOT EXISTS salaries_h1 Partition of salaries_horizontal for values from (MINVALUE) to ('1990-01-01');
Create table IF NOT EXISTS salaries_h2 Partition of salaries_horizontal for values from ('1990-01-01') to ('1992-01-01');
Create table IF NOT EXISTS salaries_h3 Partition of salaries_horizontal for values from ('1992-01-01') to ('1994-01-01');
Create table IF NOT EXISTS salaries_h4 Partition of salaries_horizontal for values from ('1994-01-01') to ('1996-01-01');
Create table IF NOT EXISTS salaries_h5 Partition of salaries_horizontal for values from ('1996-01-01') to ('1998-01-01');
Create table IF NOT EXISTS salaries_h6 Partition of salaries_horizontal for values from ('1998-01-01') to ('2000-01-01');
Create table IF NOT EXISTS salaries_h7 Partition of salaries_horizontal for values from ('2000-01-01') to (MAXVALUE);

TRUNCATE TABLE salaries_horizontal;
INSERT INTO salaries_horizontal
SELECT *
FROM salaries;
```

```
psql:a1_s4775476.sql:23: NOTICE:  relation "salaries_horizontal" already exists, skipping
CREATE TABLE
psql:a1_s4775476.sql:25: NOTICE:  relation "salaries_h1" already exists, skipping
CREATE TABLE
psql:a1_s4775476.sql:26: NOTICE:  relation "salaries_h2" already exists, skipping
CREATE TABLE
psql:a1_s4775476.sql:27: NOTICE:  relation "salaries_h3" already exists, skipping
CREATE TABLE
psql:a1_s4775476.sql:28: NOTICE:  relation "salaries_h4" already exists, skipping
CREATE TABLE
psql:a1_s4775476.sql:29: NOTICE:  relation "salaries_h5" already exists, skipping
CREATE TABLE
psql:a1_s4775476.sql:30: NOTICE:  relation "salaries_h6" already exists, skipping
CREATE TABLE
psql:a1_s4775476.sql:31: NOTICE:  relation "salaries_h7" already exists, skipping
CREATE TABLE
TRUNCATE TABLE
INSERT 0 2844047
```

--2.2

```sql
--2.2
SELECT AVG(salary) AS average_salary
FROM salaries_horizontal
WHERE from_date >= '1996-06-30' AND from_date <= '1996-12-31';

EXPLAIN SELECT AVG(salary) AS average_salary
FROM salaries_horizontal
WHERE from_date >= '1996-06-30' AND from_date <= '1996-12-31';
```

```
 average_salary
--------------------
 63724.924418447694
(1 row)
```

Query Plan:

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=7588.64..7588.65 rows=1 width=32)
   ->  Gather  (cost=7588.52..7588.63 rows=1 width=32)
         Workers Planned: 1
         ->  Partial Aggregate  (cost=6588.52..6588.53 rows=1 width=32)
               ->  Parallel Seq Scan on salaries_h5 salaries_horizontal  (cost=0.00..6425.57 rows=65179 width=4)
                     Filter: ((from_date >= '1996-06-30'::date) AND (from_date <= '1996-12-31'::date))
(6 rows)
```

Descriptions the optimizes the queries:

PostgreSQL starts by performing a parallel sequential scan on the salaries table, splitting the work across 1 worker. Each worker scans its portion of the table, reading all rows and applying the filter from_date > '1996-06-30' AND from_date <= '1996-12-31'. The cost of the scan is 0.00 to 6425.57, is for reading the table and applying the filter.

Next, for the partial count part. The cost increases slightly to 6589.69 to 6589.70 due to the aggregation.

It then collects the partial counts from all workers (1 worker in this case) resulting a cost of 7589.69 to 7589.80.

Finally, summing all of partial counts together COUNT(*), giving final cost of 7589.80 to 7589.81.

--2.3

rule 1 : 'emp_no', 'first_name', 'last_name', and 'hire_date' should be stored in table 'employees_public'

```sql
--2.3 verticle fragmentation

DROP TABLE IF EXISTS employees_public;

create table employees_public(emp_no integer NOT NULL,
    first_name varchar(50) NOT NULL,
    last_name varchar(50) NOT NULL,
    hire_date date NOT NULL,
    PRIMARY KEY (emp_no));

INSERT INTO employees_public (emp_no, first_name, last_name, hire_date)
SELECT emp_no, first_name, last_name, hire_date
FROM employees;
```

```
DROP TABLE
CREATE TABLE
INSERT 0 300024
```

rule 2 : 'emp_no', 'birth_date', and 'gender' should be stored in table 'employees_confidential'

rule 3 : The 'employees_ confidential' table should be stored in a new database called 'EMP_Confidential'.

This part first is to make the database required, then export sql file of employees_confidential. including the 3 attributes as tasked. then switch to new DB

```sql
--make DB
CREATE DATABASE IF NOT EXISTS emp_confidential;

-- Generate SQL file
COPY (
    SELECT
        'INSERT INTO employees_confidential (emp_no, birth_date, gender) VALUES (' ||
        emp_no || ', ''' || birth_date || ''', ''' || gender || ''');'
    FROM employees
) TO '/tmp/employees_confidential_data.sql' WITH (FORMAT TEXT);

-- Switch to new db
\c emp_confidential
```

Once in emp_confidential DB, we then create employees_confidential table with correct schema.

Create a temporary table to store the INSERT statements

Next we import sql file copy INSERT statement to populate the employees_confidential table.

```sql
DROP TABLE IF EXISTS employees_confidential;

create table employees_confidential(emp_no integer NOT NULL,
    birth_date date NOT NULL,
    gender char(1) NOT NULL,
    PRIMARY KEY (emp_no)
    );

-- Import SQL file
CREATE TEMPORARY TABLE temp_sql_statements (statement text);

COPY temp_sql_statements (statement)
FROM '/tmp/employees_confidential_data.sql' WITH (FORMAT TEXT);

DO $$
DECLARE
    sql_stmt text;
BEGIN
    FOR sql_stmt IN (SELECT statement FROM temp_sql_statements)
    LOOP
        EXECUTE sql_stmt;
    END LOOP;
END $$;


\c emp_s4775476
```

Output :

```
DROP TABLE
CREATE TABLE
INSERT 0 300024
psql:a1_s4775476.sql:63: ERROR:  database "emp_confidential" already exists
COPY 300024
You are now connected to database "emp_confidential" as user "s4775476".
DROP TABLE
CREATE TABLE
CREATE TABLE
COPY 300024
DO
You are now connected to database "emp_s4775476" as user "s4775476".
```

additional checks:

```
emp_s4775476=# \c emp_confidential
You are now connected to database "emp_confidential" as user "s4775476".
emp_confidential=# \d
                 List of relations
 Schema |          Name          | Type  |  Owner
--------+------------------------+-------+----------
 public | employees_confidential | table | s4775476
(1 row)

emp_confidential=# \d employees_confidential
           Table "public.employees_confidential"
   Column   |     Type     | Collation | Nullable | Default
------------+--------------+-----------+----------+---------
 emp_no     | integer      |           | not null |
 birth_date | date         |           | not null |
 gender     | character(1) |           | not null |
Indexes:
    "employees_confidential_pkey" PRIMARY KEY, btree (emp_no)
```

Task 3

--3.1

ANSWER:

i) **Full Replication**:

- Design: Each server (S1 to S5) stores all 10 fragments (Fragments 1 to 10, covering emp_no 1 to 1,000,000).
- Pros:
    1. High availability: Any server can serve any emp_no (S3 can handle emp_no 150,000 if S1 fails).
    2. Low latency for reads: Queries are executed locally on any server.
    3. Load balancing: Read queries can be distributed across all servers.
- Cons:
    1. High storage cost: Each server stores 10 fragments (10 GB per server, 50 GB total if each fragment is 1 GB).
    2. High update overhead: Updates must be propagated to all servers
    3. Consistency challenges: Synchronization across all servers can introduce delays or conflicts.

ii) **Partial Replication**:

- Design:
    - S1: Fragments 1, 2, 3, 4 (emp_no 1 to 400,000)
    - S2: Fragments 3, 4, 5, 6 (emp_no 200,001 to 600,000)
    - S3: Fragments 5, 6, 7, 8 (emp_no 400,001 to 800,000)
    - S4: Fragments 7, 8, 9, 10 (emp_no 600,001 to 1,000,000)
    - S5: Fragments 9, 10, 1, 2 (emp_no 800,001 to 1,000,000 and 1 to 200,000)
- Pros:

1. Balanced storage: Each server stores 4 fragments (total 20GB so 4 each)
2. Lower update overhead: Updates are propagated to fewer servers (updating emp_no 250,000 in Fragment 3 only syncs S1 and S2).
3. Redundancy: Each fragment is on 2 servers, providing fault tolerance (if S1 fails, S2 can serve Fragment 3).

- Cons:
    1. Increased latency for some queries: Servers may need to query remote servers (S3 querying S1 for emp_no 150,000 in Fragment 2).
    2. Complex query routing: Requires a mapping to route queries to the correct server.
    3. Load imbalance: Servers with popular fragments may be overloaded.

iii) **No Replication**:

- Design:
    - S1: Fragments 1, 2 (emp_no 1 to 200,000)
    - S2: Fragments 3, 4 (emp_no 200,001 to 400,000)
    - S3: Fragments 5, 6 (emp_no 400,001 to 600,000)
    - S4: Fragments 7, 8 (emp_no 600,001 to 800,000)
    - S5: Fragments 9, 10 (emp_no 800,001 to 1,000,000)
- Pros:
    1. Minimal storage: Each server stores 2 fragments (2 GB per server, 10 GB total).
    2. Low update overhead: Updates affect only one server (updating emp_no 250,000 in Fragment 3 only affects S2).
    3. Simple consistency: No synchronization needed.
- Cons:
    1. Low availability: If a server fails, its fragments are inaccessible (if S2 fails, Fragment 3 is unavailable).
    2. High latency for remote queries: Queries must be routed to the correct server (S1 querying S2 for emp_no 250,000).
    3. Load imbalance: Servers with popular fragments may be overloaded

--3.2

Replication Design: Partial replication (each fragment on 2 servers).

Fragment Allocation:

- S1: Fragments 1, 2, 3, 4 (emp_no 1 to 400,000)
- S2: Fragments 3, 4, 5, 6 (emp_no 200,001 to 600,000)
- S3: Fragments 5, 6, 7, 8 (emp_no 400,001 to 800,000)
- S4: Fragments 7, 8, 9, 10 (emp_no 600,001 to 1,000,000)
- S5: Fragments 9, 10, 1, 2 (emp_no 800,001 to 1,000,000 and 1 to 200,000)

Master Server: S1.

Process for updating(for emp_no 250,000, changing first_name to 'John'):

1. Identify the fragment: emp_no 250,000 belongs to Fragment 3 (emp_no 200,001 to 300,000), stored on S1 and S2.
2. Next, update on S1: S1 updates its local copy of Fragment 3 (UPDATE employee SET first_name = 'John' WHERE emp_no = 250000).

3. After updating, is time to propagate to S2: S1 sends the update to S2, which applies it to its copy of Fragment 3.
4. To ensure consistency, S1 confirms the update on S2, using synchronous or asynchronous replication.
5. Finally, Resolving conflicts case can be, for example if S2 had a conflicting update, resolve using “last writer wins” .

Task 4

--4.1

```
-- 4.1: Establish FDW to sharedb

Create extension  IF NOT EXISTS postgres_fdw;

Create server sharedb_server
    Foreign data wrapper postgres_fdw
    OPTIONS (host 'infs3200-sharedb.zones.eait.uq.edu.au', port '5432', dbname 'sharedb');

--user mapping for the sharedb user
create user mapping for s4775476
    server sharedb_server
    OPTIONS (user 'sharedb', password 'Y3Y7FdqDSM9.3d47XUWg');

-- Create a foreign table to map the titles table
Create foreign table titles_f (
    emp_no integer NOT NULL,
    title varchar(50) NOT NULL,
    from_date date NOT NULL,
    to_date date
)
    server sharedb_server
    OPTIONS (schema_name 'public', table_name 'titles');

select count(*) from titles_f;
```

```
psql:a1_s4775476.sql:124: NOTICE:  extension "postgres_fdw" already exists, skipping
CREATE EXTENSION
psql:a1_s4775476.sql:128: ERROR:  server "sharedb_server" already exists
psql:a1_s4775476.sql:133: ERROR:  user mapping for "s4775476" already exists for server "sharedb_server"
psql:a1_s4775476.sql:143: ERROR:  relation "titles_f" already exists
 count
--------
 443308
(1 row)
```

--4.2

```sql
-- 4.2 Calculate average current salary per unique title

SELECT
    t.title,
    AVG(s.salary) AS avg_current_salary
FROM public.titles_f t
JOIN (
    SELECT emp_no, salary, from_date
    FROM salaries s1
    WHERE from_date = (
        SELECT MAX(from_date)
        FROM salaries s2
        WHERE s2.emp_no = s1.emp_no
    )
) s
ON t.emp_no = s.emp_no
WHERE t.to_date = '9999-01-01'
GROUP BY t.title
ORDER BY avg_current_salary DESC;
```

```
       title        | avg_current_salary
--------------------+--------------------
 Senior Staff       | 80706.495879254852
 Manager            | 77723.666666666667
 Senior Engineer    | 70823.437647633787
 Technique Leader   | 67506.590294483617
 Staff              | 67330.665204105618
 Engineer           | 59602.737759416454
 Assistant Engineer | 57317.573578595318
(7 rows)
```

--4.3

```sql
--4.3 Establish FDW to emp_confidential
create server emp_confidential_server
    Foreign data wrapper postgres_fdw
    OPTIONS (host 'localhost', port '5432', dbname 'emp_confidential');

create user mapping for s4775476
    SERVER emp_confidential_server
    OPTIONS (user 's4775476', password '');

create foreign table employees_confidential_f (
    emp_no integer NOT NULL,
    birth_date date NOT NULL,
    gender char(1) NOT NULL
)
    server emp_confidential_server
    OPTIONS (schema_name 'public', table_name 'employees_confidential');
```

```
CREATE SERVER
CREATE USER MAPPING
CREATE FOREIGN TABLE
```

```
psql:a1_s4775476.sql:147: NOTICE:  server "emp_confidential_server" already exists, skipping
CREATE SERVER
psql:a1_s4775476.sql:152: ERROR:  user mapping for "s4775476" already exists for server "emp_confidential_server"
psql:a1_s4775476.sql:158: NOTICE:  relation "employees_confidential_f" already exists, skipping
CREATE FOREIGN TABLE
You are now connected to database "emp_confidential" as user "s4775476".
CREATE ROLE
GRANT
GRANT
GRANT
```

```
You are now connected to database "emp_s4775476" as user "s4775476".
 first_name | last_name
------------+-----------
(0 rows)
```

--4.4

transmission cost (not the join cost).

```sql
--4.4
-- Semi-join
EXPLAIN
SELECT ep.first_name, ep.last_name, fr.birth_date
FROM employees_public ep,
    (SELECT ec.emp_no, ec.birth_date
     FROM employees_confidential_f ec, (SELECT emp_no FROM employees_public) ep
     WHERE ec.emp_no = ep.emp_no
     AND ec.birth_date >= '1970-01-01' AND ec.birth_date < '1975-01-01') fr
WHERE ep.emp_no = fr.emp_no;

-- Inner join
EXPLAIN
SELECT ep.first_name, ep.last_name, ec.birth_date, ec.gender
FROM employees_public ep
INNER JOIN employees_confidential_f ec
ON ep.emp_no = ec.emp_no
WHERE ep.emp_no IN (
    SELECT ec2.emp_no
    FROM employees_confidential_f ec2
    WHERE ec2.birth_date >= '1970-01-01' AND ec2.birth_date < '1975-01-01'
);
```

```
                                              QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=100.83..268.98 rows=13 width=240)
   Join Filter: (employees_public.emp_no = ec.emp_no)
   ->  Nested Loop  (cost=100.42..260.75 rows=13 width=248)
         ->  Foreign Scan on employees_confidential_f ec  (cost=100.00..151.13 rows=13 width=8)
         ->  Index Scan using employees_public_pkey on employees_public ep  (cost=0.41..8.43 rows=1 width=240)
               Index Cond: (emp_no = ec.emp_no)
   ->  Index Only Scan using employees_public_pkey on employees_public  (cost=0.41..0.62 rows=1 width=4)
         Index Cond: (emp_no = ep.emp_no)
(8 rows)
```

```
                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------------
Hash Join  (cost=375.45..843.70 rows=1 width=248)
  Hash Cond: (ec.emp_no = ep.emp_no)
  ->  Foreign Scan on employees_confidential_f ec  (cost=100.00..560.56 rows=2048 width=16)
  ->  Hash  (cost=275.26..275.26 rows=15 width=244)
        ->  Nested Loop  (cost=157.48..275.26 rows=15 width=244)
              ->  HashAggregate  (cost=157.06..157.20 rows=14 width=4)
                    Group Key: ec2.emp_no
                    ->  Foreign Scan on employees_confidential_f ec2  (cost=100.00..157.03 rows=15 width=4)
              ->  Index Scan using employees_public_pkey on employees_public ep  (cost=0.41..8.43 rows=1 width=240)
                    Index Cond: (emp_no = ec2.emp_no)
(10 rows)
```

The query plan matches the expected structure for an inner join, with a Foreign Scan on employees_confidential_f and a direct join with employees_public. PostgreSQL prefers a Nested Loop over a Hash Join. If the estimate were higher, it might have used a Hash Join.

Semi-Join Transmission Cost

- Rows: 13 from the Foreign Scan.
- Width: 8 bytes from emp_no 4 bytes + birth_date 4 bytes.
- Transmission Cost: 13 rows × 8 bytes = 104 bytes.

Inner Join Transmission Cost:

- Rows: 16 from Foreign Scan.
- Expected width is 9 bytes ; emp_no 4 bytes + birth_date 4 bytes + gender 1 byte. Hence the Transmission Cost would be10 rows × 9 bytes = 90 bytes.
- While the actual Width is 16 bytes as you can see from query plan. Resulting in Transmission Cost of 10 rows × 16 bytes = 160 bytes.

In Conclusion, using the expected width, the inner join appears more efficient (90 bytes vs. 104 bytes).

However, the actual width suggests the inner join transfers more data (160 bytes vs. 104 bytes), making the semi-join more efficient in practice.