

Assignment 1: INFS3200

Alexander White: s4321830

July 29, 2025

1 Question 1

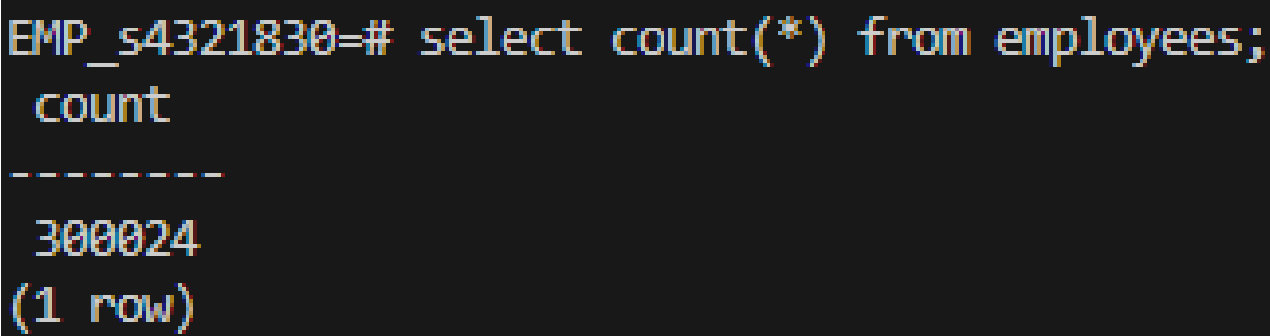
1.1 1.1

The following SQL commands were used to create the database and load the provided SQL files:

```
CREATE DATABASE "EMP_s4321830";  
\c "EMP_s4321830"  
\i ./Resources/departments.sql;  
\i ./Resources/employees.sql;  
\i ./Resources/salaries.sql;  
\i ./Resources/dept_manager.sql;
```

To count the total number of employees in the database, the following query was used:

```
SELECT COUNT(*) FROM employees;
```



The screenshot shows a terminal window with a black background and yellow text. The prompt is 'EMP_s4321830=#'. The query entered is 'select count(*) from employees;'. The output shows 'count' followed by a dashed line separator, then the value '300024', and finally '(1 row)'.

```
EMP_s4321830=# select count(*) from employees;  
count  
-----  
300024  
(1 row)
```

Figure 1: Query result for count of employees

1.2 1.2

To find the number of employees in the Marketing department, the following query was used:

```
SELECT COUNT(e.emp_no)  
FROM employees e  
JOIN dept_emp de ON e.emp_no = de.emp_no  
JOIN departments d ON de.dept_no = d.dept_no  
WHERE d.dept_name = 'Marketing';
```

```

EMP_s4321830=# select count(e.emp_no) from employees e join dept_emp de on e.emp_no = de.emp_no join
departments d on de.dept_no = d.dept_no where d.dept_name = 'Marketing';
count
-----
20211
(1 row)

```

Figure 2: Query result for number of employees in the Marketing department

2 Question 2

2.1 2.1

The query below was used to create the horizontally partitioned version of the table, and then partition it into 7 partitions based on the from_dates. The data from the old version of the salaries table was then inserted into the partitioned table.

```

CREATE TABLE salaries_horizontal (
    emp_no integer not null,
    salary integer not null,
    from_date date not null,
    to_date date not null,
    PRIMARY KEY (emp_no, from_date)
) PARTITION BY RANGE (from_date);

CREATE TABLE salaries_h1 PARTITION OF salaries_horizontal
FOR VALUES FROM ('1900-01-01') TO ('1990-01-01');

CREATE TABLE salaries_h2 PARTITION OF salaries_horizontal
FOR VALUES FROM ('1990-01-01') TO ('1992-01-01');

CREATE TABLE salaries_h3 PARTITION OF salaries_horizontal
FOR VALUES FROM ('1992-01-01') TO ('1994-01-01');

CREATE TABLE salaries_h4 PARTITION OF salaries_horizontal
FOR VALUES FROM ('1994-01-01') TO ('1996-01-01');

CREATE TABLE salaries_h5 PARTITION OF salaries_horizontal
FOR VALUES FROM ('1996-01-01') TO ('1998-01-01');

CREATE TABLE salaries_h6 PARTITION OF salaries_horizontal
FOR VALUES FROM ('1998-01-01') TO ('2000-01-01');

CREATE TABLE salaries_h7 PARTITION OF salaries_horizontal
FOR VALUES FROM ('2000-01-01') TO ('2030-01-01');

INSERT INTO salaries_horizontal SELECT * FROM salaries;

```

```

EMP_s4321830=# CREATE TABLE salaries_horizontal (
    emp_no integer not null,
    salary integer not null,
    from_date date not null,
    to_date date not null,
    PRIMARY KEY (emp_no, from_date)
) PARTITION BY RANGE (from_date);
CREATE TABLE
EMP_s4321830=# CREATE TABLE salaries_h1 PARTITION OF salaries_horizontal FOR VALUES FROM ('1900-01-01') TO ('1990-01-01');
CREATE TABLE salaries_h2 PARTITION OF salaries_horizontal FOR VALUES FROM ('1990-01-01') TO ('1992-01-01');
CREATE TABLE salaries_h3 PARTITION OF salaries_horizontal FOR VALUES FROM ('1992-01-01') TO ('1994-01-01');
CREATE TABLE salaries_h4 PARTITION OF salaries_horizontal FOR VALUES FROM ('1994-01-01') TO ('1996-01-01');
CREATE TABLE salaries_h5 PARTITION OF salaries_horizontal FOR VALUES FROM ('1996-01-01') TO ('1998-01-01');
CREATE TABLE salaries_h6 PARTITION OF salaries_horizontal FOR VALUES FROM ('1998-01-01') TO ('2000-01-01');
CREATE TABLE salaries_h7 PARTITION OF salaries_horizontal FOR VALUES FROM ('2000-01-01') TO ('2030-01-01');
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
EMP_s4321830=# 

```

Figure 3: Query for creating and partitioning salaries table

```

EMP_s4321830=# select table_name from information_schema.tables where table_name like 'salaries%';
    table_name
-----
salaries
salaries_h1
salaries_h2
salaries_h3
salaries_h4
salaries_h5
salaries_h6
salaries_h7
salaries_horizontal
(9 rows)

```

Figure 4: Showing list of tables in the database

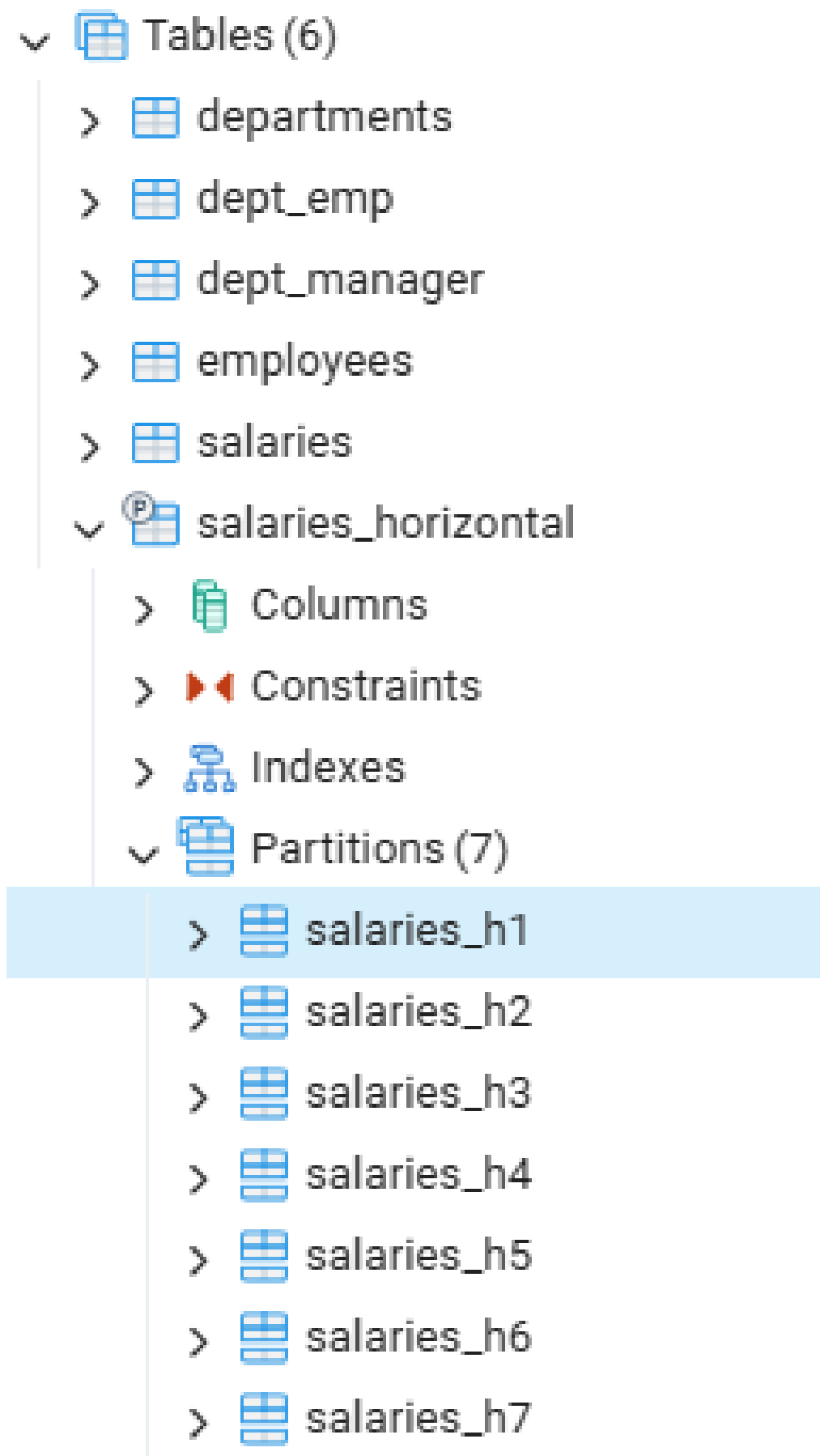


Figure 5: Partition tables confirmation

2.2 2.2

The query below was used to calculate the average salary between the specified dates:

```
SELECT AVG(salary) AS Average_Salary
FROM salaries_horizontal
WHERE from_date BETWEEN '1996-06-30' AND '1996-12-31';
```

```
EMP_s4321830=# SELECT AVG(salary) AS Average_Salary FROM salaries_horizontal WHERE from_date BETWEEN '1996-06-30' AND '1996-12-31';
average_salary
-----
63724.924418447694
(1 row)
```

Figure 6: Average salary query

This query was used to find the query execution plan:

```
EXPLAIN ANALYZE SELECT AVG(salary) AS Average_Salary
FROM salaries_horizontal
WHERE from_date BETWEEN '1996-06-30' AND '1996-12-31';
```

```
EMP_s4321830=# EXPLAIN ANALYZE SELECT AVG(salary) AS Average_Salary FROM salaries_horizontal WHERE from_date BETWEEN '1996-06-30' AND '1996-12-31';
QUERY PLAN
-----
Finalize Aggregate  (cost=7588.01..7588.02 rows=1 width=32) (actual time=21.551..31.761 rows=1 loops=1)
-> Gather  (cost=7587.89..7588.00 rows=1 width=32) (actual time=21.426..31.755 rows=2 loops=1)
    Workers Planned: 1
    Workers Launched: 1
-> Partial Aggregate  (cost=6587.89..6587.90 rows=1 width=32) (actual time=16.593..16.593 rows=1 loops=2)
    -> Parallel Seq Scan on salaries_h5 salaries_horizontal  (cost=0.00..6424.81 rows=65231 width=4) (actual time=0.031..14.449 rows=55562 loops=2)
        Filter: (((from_date >= '1996-06-30'::date) AND (from_date <= '1996-12-31'::date)))
        Rows Removed by Filter: 170187
Planning Time: 0.171 ms
Execution Time: 31.782 ms
(10 rows)
```

Figure 7: Query Plan

The query was executed using a parallel sequential scan on the salaries_h5 partition of the salaries_horizontal table. There was one additional worker used, resulting in parallel processes that filtered and aggregated data within the specified date range. The PSQL database system used parallelism, aggregation and filtering to ensure that the query was ran efficiently and optimised.

2.3 2.3

The next query was used to vertically partition the employees table. Firstly I had to create the employees_public table, and then insert the data from the original employees table into it.

```
CREATE TABLE employees_public (
    emp_no integer NOT NULL PRIMARY KEY,
    first_name varchar(40) NOT NULL,
    last_name varchar(40) NOT NULL,
    hire_date date NOT NULL
);

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

```

EMP_s4321830=# CREATE TABLE employees_public (
    emp_no INTEGER NOT NULL PRIMARY KEY,
    first_name VARCHAR(40) NOT NULL,
    last_name VARCHAR(40) NOT NULL,
    hire_date DATE NOT NULL
);

INSERT INTO employees_public
SELECT emp_no, first_name, last_name, hire_date FROM employees;
CREATE TABLE
INSERT 0 300024

```

Figure 8: Creating employees public partition

The next code chunk was used to create the EMP confidential database, and then create the employees confidential table that sits in it.

```
CREATE DATABASE "EMP_Confidential";
```

```

\c "EMP_Confidential"
\i ./Resources/employees.sql;

```

```

CREATE TABLE employees_confidential (
    emp_no integer NOT NULL PRIMARY KEY,
    birth_date date NOT NULL,
    gender char(1) not null check(gender in ('M', 'F'))
);

```

```

INSERT INTO employees_confidential
SELECT emp_no, birth_date, gender
FROM employees;

```

```

EMP_s4321830=# CREATE DATABASE "EMP_Confidential";
CREATE DATABASE

```

Figure 9: Creating confidential vertical partition

```

EMP_s4321830=# \c "EMP_Confidential"
You are now connected to database "EMP_Confidential" as user "s4321830".

```

Figure 10: Connecting to database

```
EMP_Confidential=# CREATE TABLE employees_confidential (  
    emp_no integer NOT NULL PRIMARY KEY,  
    birth_date date NOT NULL,  
    gender char(1) not null check(gender in ('M', 'F'))  
);  
CREATE TABLE
```

Figure 11: Creating confidential table

```
EMP_Confidential=# \i ./Resources/employees.sql;  
CREATE TABLE  
INSERT 0 10000  
INSERT 0 10000  
INSERT 0 10000
```

Figure 12: Duplicating employee table in database

```
EMP_Confidential=# INSERT INTO employees_confidential SELECT emp_no, birth_date, gender FROM employees;  
INSERT 0 300024
```

Figure 13: Inserting employee data into confidential table

3 Question 3

3.1 3.1 Replication Strategies

3.1.1 Full Replication

Full replication means that each of the 5 server sites stores a full copy of the employee table, including all partitions.

Pros:

- Ensures no data loss occurs, as it is replicated across multiple locations.
- Faster querying since searches can be performed locally rather than across multiple server sites.
- High availability: If one server goes down, others still have a complete copy.

Cons:

- High storage and maintenance costs due to multiple full copies of the data.
- Risk of inconsistency due to synchronization delays across multiple servers.
- Increased network traffic for update synchronization.

3.1.2 Partial Replication

Partial replication means that partitions of the table are distributed across different server sites, with some overlap for redundancy.

Pros:

- Lower storage costs compared to full replication.
- Redundant copies provide security in case of server failure.
- Frequently accessed partitions can be replicated to multiple sites, improving query speed.

Cons:

- Increased complexity for query execution, as some queries may need to access multiple sites.
- Partial redundancy means some data might become unavailable if a site fails.
- Synchronization overhead to maintain consistency across replicated partitions.

3.1.3 No Replication

No replication means each server site stores only its assigned partitions, with no duplication across sites.

Pros:

- Lowest storage costs, as no duplicate copies exist.
- No synchronization overhead, since each site manages its own data.
- Efficient updates since changes affect only one location.

Cons:

- High risk of data loss if a server goes down, as there are no backups.
- Increased query processing time, as queries may require fetching data from multiple sites.
- Reduced fault tolerance due to lack of redundancy.

3.2 3.2

The process to update a record with a specific 'emp_no' would be as follows in a fragmented system:

- **Send query to the master server:** The update request is first sent to the master server, which manages fragmentation metadata.
- **Identify the correct partition:** The system determines which partition contains the specified 'emp_no' based on the fragmentation scheme.
- **Locate the responsible server:** The server site storing the partition is identified.
- **Begin transaction and execute update:** The update is performed at the identified server.
- **Propagate changes (if replication exists):** If the partition is replicated across multiple sites, the update is sent to all relevant servers to ensure consistency.
- **Conflict resolution (if needed):** Versioning, timestamps, or two-phase commit (2PC) mechanisms may be used to maintain consistency.
- **Commit the transaction:** Once all updates are completed successfully, the transaction is committed to ensure data integrity.

4 Question 4

4.1 4.1

The code below was used to establish the foreign data wrapper, and grant access to the foreign table 'titles'.

```
\c EMP_s4321830
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');

CREATE USER MAPPING FOR "s4321830"
SERVER sharedb_server
OPTIONS (user 'sharedb', password 'Y3Y7FdqDSM9.3d47XUWg');

CREATE FOREIGN TABLE titles (
    emp_no INTEGER NOT NULL,
    title VARCHAR(40) NOT NULL,
    from_date DATE NOT NULL,
    to_date DATE NOT NULL
)
SERVER sharedb_server
OPTIONS (schema_name 'public', table_name 'titles');
```

```

EMP_s4321830=# CREATE FOREIGN TABLE titles (
    emp_no INTEGER NOT NULL,
    title VARCHAR(40) NOT NULL,
    from_date DATE NOT NULL,
    to_date DATE NOT NULL
)
SERVER sharedb_server
OPTIONS (schema_name 'public', table_name 'titles');
CREATE FOREIGN TABLE

```

Figure 14: Creating titles foreign table

4.2 4.2

The following query was used to aggregate the current average salary for each title. I put the filter on both tables, to only include their current salaries for their current titles.

```

SELECT t.title , AVG(s.salary) AS avg_salary
FROM titles t
JOIN salaries s ON t.emp_no = s.emp_no
WHERE t.to_date = '9999-01-01' AND s.to_date = '9999-01-01'
GROUP BY t.title;

```

Screenshot of output:

```

EMP_s4321830=# SELECT t.title, AVG(s.salary) AS avg_salary
FROM titles t
JOIN salaries s ON t.emp_no = s.emp_no
WHERE t.to_date = '9999-01-01' AND s.to_date = '9999-01-01'
GROUP BY t.title;

```

title	avg_salary
Assistant Engineer	57317.573578595318
Engineer	59602.737759416454
Manager	77723.666666666667
Senior Engineer	70823.437647633787
Senior Staff	80706.495879254852
Staff	67330.665204105618
Technique Leader	67506.590294483617

(7 rows)

Figure 15: Output of Average Salary Query

4.3 4.3

This code was used to create the FDW for the EMP_Confidential database and table. I had to create a user role called readonly_user which I then granted the permissions to access the confidential employee data. I then mapped the user role to my user account. I created the FDW in the same way as earlier, with creating the foreign table:

```
CREATE USER readonly_user WITH PASSWORD 'infs3200';

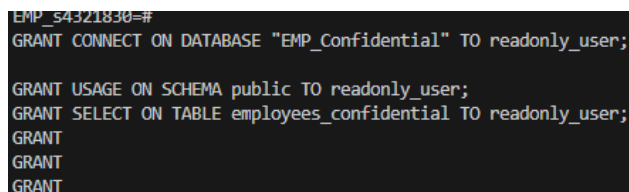
GRANT CONNECT ON DATABASE "EMP_Confidential" TO readonly_user;
\c EMP_Confidential
GRANT USAGE ON SCHEMA public TO readonly_user;
GRANT SELECT ON TABLE employees_confidential TO readonly_user;

\c EMP_s4321830
CREATE EXTENSION IF NOT EXISTS postgres_fdw;

CREATE SERVER emp_confidential_server
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', port '5432', dbname 'EMP_Confidential');

CREATE USER MAPPING FOR s4321830
SERVER emp_confidential_server
OPTIONS (user 'readonly_user', password 'infs3200');

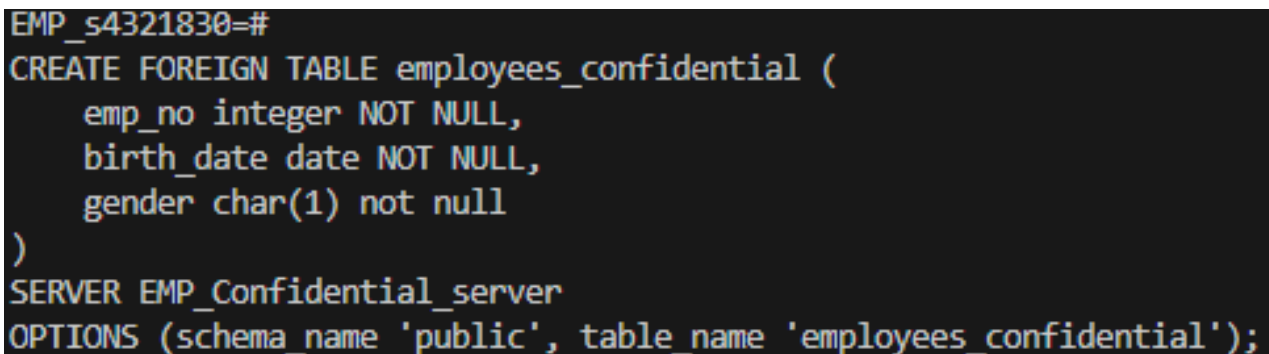
CREATE FOREIGN TABLE employees_confidential (
    emp_no INTEGER NOT NULL,
    birth_date DATE NOT NULL,
    gender CHAR(1) NOT NULL
)
SERVER emp_confidential_server
OPTIONS (schema_name 'public', table_name 'employees_confidential');
```



```
EMP_s4321830=#
GRANT CONNECT ON DATABASE "EMP_Confidential" TO readonly_user;

GRANT USAGE ON SCHEMA public TO readonly_user;
GRANT SELECT ON TABLE employees_confidential TO readonly_user;
GRANT
GRANT
GRANT
```

Figure 16: Screenshot of granting permissions to readonly user



```
EMP_s4321830=#
CREATE FOREIGN TABLE employees_confidential (
    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');
```

Figure 17: Enter Caption

Now that I could access the foreign table, I could perform a semi-join on the confidential employee data to the public data. This is done using the query below:

```

SELECT e.first_name , e.last_name
FROM employees_public e
WHERE EXISTS (
    SELECT 1 FROM employees_confidential ec
    WHERE e.emp_no = ec.emp_no
    AND ec.birth_date BETWEEN '1970-01-01' AND '1974-12-31'
);

```

```

EMP_s4321830=# SELECT e.first_name, e.last_name
FROM employees_public e
WHERE EXISTS (
    SELECT 1 FROM employees_confidential ec
    WHERE e.emp_no = ec.emp_no
    AND ec.birth_date BETWEEN '1970-01-01' AND '1974-12-31'
);
first_name | last_name
-----+-----
(0 rows)

```

Figure 18: Semi-Join Query

The way this semi-join works is that it only returns the first name and the last name, if there exists a row with the corresponding employee number in the confidential table, that has a birth date within the specified range. The select 1 is just checking if the row exists, and whether to return the data with that employees name or not.

4.4 4.4

The semi-join was shown in the previous answer, but the inner join is shown here:

```

SELECT e.first_name , e.last_name
FROM employees_public e
JOIN employees_confidential ec ON e.emp_no = ec.emp_no
WHERE ec.birth_date BETWEEN '1970-01-01' AND '1974-12-31';

```

```

EMP_s4321830=# SELECT e.first_name, e.last_name
FROM employees_public e
JOIN employees_confidential ec ON e.emp_no = ec.emp_no
WHERE ec.birth_date BETWEEN '1970-01-01' AND '1974-12-31';
first_name | last_name
-----+-----
(0 rows)

```

Figure 19: Inner-Join Query

The way the inner join works is that it joins the public and confidential tables on the employee number and then filters for birth date within the specified range.

Query plan for semi-join:

```

----- QUERY PLAN -----
Nested Loop (cost=157.49..275.36 rows=15 width=15) (actual time=10.320..10.321 rows=0 loops=1)
-> HashAggregate (cost=157.06..157.20 rows=14 width=4) (actual time=10.320..10.320 rows=0 loops=1)
    Group Key: ec.emp_no
    Batches: 1 Memory Usage: 24kB
    -> Foreign Scan on employees_confidential ec (cost=100.00..157.03 rows=15 width=4) (actual time=10.318..10.319 rows=0 loops=1)
    -> Index Scan using employees_public_pkey on employees_public e (cost=0.42..8.44 rows=1 width=19) (never executed)
        Index Cond: (emp_no = ec.emp_no)
Planning Time: 0.146 ms
Execution Time: 10.668 ms
(9 rows)

```

Figure 20: Query Plan for Semi-Join

Query plan for inner-join:

```

EMP_s4321830=# EXPLAIN ANALYZE SELECT e.first_name, e.last_name
FROM employees_public e
JOIN employees_confidential ec ON e.emp_no = ec.emp_no
WHERE ec.birth_date BETWEEN '1970-01-01' AND '1974-12-31';
----- QUERY PLAN -----
Nested Loop (cost=100.42..283.63 rows=15 width=15) (actual time=26.918..26.919 rows=0 loops=1)
-> Foreign Scan on employees_confidential ec (cost=100.00..157.03 rows=15 width=4) (actual time=26.916..26.917 rows=0 loops=1)
-> Index Scan using employees_public_pkey on employees_public e (cost=0.42..8.44 rows=1 width=19) (never executed)
    Index Cond: (emp_no = ec.emp_no)
Planning Time: 4.456 ms
Execution Time: 48.412 ms
(6 rows)

```

Figure 21: Query Plan for Inner Join

When comparing the two query plans for the semi-join and the inner join, the results indicate that the semi-join is more efficient in this scenario, particularly in terms of transmission cost.

The semi-join involves a nested loop join, which incurs a total cost of around 118. It completes a foreign scan on the employees_confidential table, which incurs a transmission cost of approx 57, and also includes an index scan on employees_public, though this index scan is never executed since no rows are returned. The semi-join only retrieves emp_no values from employees_confidential, meaning that only the necessary data is transmitted across the FDW. The filtered employee numbers are used to extract first_name and last_name from employees_public, ensuring that minimal data is transferred between databases.

The inner join also involves a nested loop join, but its cost is higher, around 183. Like the semi-join, it also performs a foreign scan on employees_confidential, which has the same 57 transmission cost. However, the inner join retrieves full rows from employees_confidential, meaning that emp_no, birth_date, and gender are transmitted to the database before filtering. This increases network overhead, as more data is transferred across the FDW. The retrieved rows are then joined with employees_public locally, contributing to the additional cost and execution time.