

Optimización de Consultas Distribuidas en PostgreSQL

Laboratorio extra



Integrantes:

Huarino Anchillo, Noemi (100 %)
Tovar Tolentino, Mariel Carolina (100 %)

Docente: Heider Sanchez
Curso: Base de datos 2
Grupo de Laboratorio: 5

Índice

1. P0. Creación de Tablas Fragmentadas	1
1.1. Fragmentación de tablas	2
1.1.1. Fragmentación de la Tabla Medico	3
1.1.2. Fragmentación de la Tabla Diagnostico	3
1.2. Población de tablas	4
1.2.1. Generación de Datos para la Tabla Medico	4
1.2.2. Inserción de Datos para la Tabla Medico	6
1.2.3. Generación de Datos para la Tabla Diagnóstico	6
1.2.4. Inserción de Datos para la Tabla Medico	7
1.3. Distribución de Registros en los Fragmentos	7
1.3.1. Distribución de Registros en la Tabla Medico	8
1.3.2. Distribución de Registros en la Tabla Diagnostico	8
2. P1. Algoritmos distribuidos localmente	9
2.1. Consulta 1	9
2.1.1. Sentencia SQL de la implementación de la consulta optimizada	9
2.1.2. Output	10
2.1.3. Gráfico del plan de ejecución resultante	11
2.2. Consulta 2	11
2.2.1. Sentencia SQL de la implementación de la consulta optimizada	11
2.2.2. Output	13
2.2.3. Gráfico del plan de ejecución resultante	14
2.3. Consulta 3	14
2.3.1. Sentencia SQL de la implementación de la consulta optimizada	14
2.3.2. Output	16
2.3.3. Gráfico del plan de ejecución resultante	17
2.4. Consulta 4	18
2.4.1. Sentencia SQL de la implementación de la consulta optimizada	18
2.4.2. Output	19
2.4.3. Gráfico del plan de ejecución resultante	20
3. P2. Algoritmos distribuidos en al menos tres servidores	21
3.1. Creación de los servidores	21
3.2. Master	22
3.3. Worker 1	23
3.4. Worker 2	24
3.5. Consulta 1	24
3.6. Consulta 2	26
3.7. Consulta 3	27
3.8. Consulta 4	29

1. P0. Creación de Tablas Fragmentadas

En este caso se pide crear 2 tablas principales:

- **Médico**(DNI, Email, Nombre, Apellidos, Especialidad, NumColegiado, Centro Salud, Ciudad)
- **Diagnóstico**(Id, Dni_Paciente, Dni_Medico, Ciudad, Diagnóstico, Peso, Edad, Sexo)

Estas tablas se van a fragmentar para posteriormente aplicar algoritmos de consultas distribuidas eficientes. Las consultas a realizar son las siguientes:

Consulta 1

La consulta devuelve todas las filas de la tabla Diagnostico, pero las ordena según la columna Ciudad, de forma ascendente.

```
1 SELECT * FROM Diagnostico ORDER BY Ciudad;
```

Consulta 2

La consulta devuelve una lista de los valores únicos de la columna DNI_Paciente de la tabla Diagnostico, es decir, elimina los registros duplicados, y solo muestra una vez cada DNI_Paciente.

```
1 SELECT DISTINCT DNI_Paciente FROM Diagnostico;
```

Consulta 3

Esta consulta devuelve una lista de las edades de los pacientes junto con la cantidad de pacientes que tienen cada edad.

```
1 SELECT Edad, COUNT(*) FROM Diagnostico GROUP BY Edad;
```

Consulta 4

La consulta devuelve una lista de especialidades de los médicos junto con la cantidad de diagnósticos asociados a cada especialidad.

```
1 SELECT Especialidad, COUNT(*)  
2 FROM Medico M  
3 JOIN Diagnostico D ON M.Dni = D.Dni_medico;
```

El código que usamos para la creación de tablas es:

```
1 --create tables:
2 CREATE SCHEMA IF NOT EXISTS labsito;
3 SET search_path TO labsito;
4
5 --correr lo siguiente solo una vez:
6 SET enable_partition_pruning = on;
7 --PO
8 CREATE TABLE IF NOT EXISTS Medico (
9     Dni BIGINT NOT NULL,
10    Nombre VARCHAR(60) NOT NULL,
11    Apellidos VARCHAR(60) NOT NULL,
12    Especialidad VARCHAR(60) NOT NULL,
13    NumColegiado VARCHAR(15) NOT NULL,
14    CentroSalud VARCHAR(60) NOT NULL,
15    Ciudad VARCHAR(60) NOT NULL,
16    PRIMARY KEY (Dni, Ciudad) --clave primaria incluye Ciudad
17 ) PARTITION BY HASH (Ciudad);
18
19
20 CREATE TABLE IF NOT EXISTS Diagnostico (
21     id SERIAL NOT NULL,
22     Dni_paciente BIGINT NOT NULL,
23     Dni_medico BIGINT NOT NULL,
24     Ciudad VARCHAR(40) NOT NULL,
25     Diagnostico TEXT NOT NULL,
26     Peso_kg NUMERIC(5, 2) NOT NULL,
27     Edad INT NOT NULL,
28     Sexo CHAR(1) CHECK (Sexo IN ('M', 'F')),
29     PRIMARY KEY (id, Ciudad), -- Incluye Ciudad en la clave primaria
30     FOREIGN KEY (Dni_medico, Ciudad) REFERENCES Medico(Dni, Ciudad) --
31     Clave externa incluye Ciudad
32 ) PARTITION BY HASH (Ciudad);
```

A continuación se da una breve explicación de las tablas:

- **Medico:** Tabla que contiene datos de médicos, particionada por Ciudad. La clave primaria es la combinación de Dni y Ciudad.
- **Diagnostico:** Tabla que contiene registros de diagnósticos médicos, particionada también por Ciudad. La clave primaria es la combinación de id y Ciudad. Además, tiene una clave externa que referencia la tabla Medico a través de las columnas Dni_medico y Ciudad.

1.1. Fragmentación de tablas

La fragmentación de tablas es una técnica utilizada para dividir una tabla en varias partes (fragmentos) para mejorar el rendimiento, la disponibilidad, y la escalabilidad de las bases de

datos. En este caso, se utilizaron dos tipos de fragmentación: la **fragmentación por hash** para la tabla Medico, y la **fragmentación horizontal derivada** para la tabla Diagnostico.

1.1.1. Fragmentación de la Tabla Medico

La tabla Medico se fragmentó utilizando la técnica de **fragmentación por hash** en el atributo Ciudad. La fragmentación por hash divide los datos en fragmentos de manera uniforme, asignando a cada fila un fragmento en función de un valor calculado a partir de una función de hash aplicada al atributo especificado, en este caso, la Ciudad.

Para la fragmentación de la tabla Medico, se decidió crear **4 fragmentos** debido a la cantidad de ciudades, utilizando un valor de MODULUS 4 y asignando los fragmentos con REMAINDER valores de 0, 1, 2 y 3. Esta decisión se tomó para distribuir uniformemente los médicos en 4 particiones, asegurando que las consultas de la base de datos puedan distribuirse entre múltiples fragmentos, mejorando el rendimiento en términos de acceso y procesamiento.

El código para la creación de los fragmentos de la tabla Medico es el siguiente:

```
1 --Partitions:
2 --Medico
3 CREATE TABLE medico_1 PARTITION OF Medico FOR VALUES WITH (MODULUS 4,
  REMAINDER 0);
4 CREATE TABLE medico_2 PARTITION OF Medico FOR VALUES WITH (MODULUS 4,
  REMAINDER 1);
5 CREATE TABLE medico_3 PARTITION OF Medico FOR VALUES WITH (MODULUS 4,
  REMAINDER 2);
6 CREATE TABLE medico_4 PARTITION OF Medico FOR VALUES WITH (MODULUS 4,
  REMAINDER 3);
```

1.1.2. Fragmentación de la Tabla Diagnostico

La tabla Diagnostico fue particionada utilizando la técnica de **fragmentación horizontal derivada**. Este tipo de fragmentación significa que los fragmentos de la tabla Diagnostico se derivan de la fragmentación de la tabla Medico. En otras palabras, la distribución de los registros en Diagnostico sigue la misma distribución que se aplicó a la tabla Medico.

La fragmentación de la tabla Diagnostico fue realizada utilizando los mismos valores de MODULUS 4 y REMAINDER que en la tabla Medico, asegurando que los registros de diagnóstico se asignen a las mismas particiones que los registros de médicos correspondientes.

El código para la creación de los fragmentos de la tabla `Diagnostico` es el siguiente:

```
1 --Diagnostico
2 -- Particiones derivadas de Medico
3 CREATE TABLE diagnostico_1 PARTITION OF Diagnostico
4 FOR VALUES WITH (MODULUS 4, REMAINDER 0);
5
6 CREATE TABLE diagnostico_2 PARTITION OF Diagnostico
7 FOR VALUES WITH (MODULUS 4, REMAINDER 1);
8
9 CREATE TABLE diagnostico_3 PARTITION OF Diagnostico
10 FOR VALUES WITH (MODULUS 4, REMAINDER 2);
11
12 CREATE TABLE diagnostico_4 PARTITION OF Diagnostico
13 FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

1.2. Población de tablas

La población de datos se generó de manera aleatoria utilizando la biblioteca **Faker** en un script de **Python** para simular la información de médicos y diagnósticos médicos, lo que permitió obtener una base de datos de prueba para las tablas `Medico` y `Diagnóstico`. A continuación, se describe el proceso de generación de los datos.

1.2.1. Generación de Datos para la Tabla `Medico`

Se creó un archivo CSV con registros simulados de médicos. El proceso de generación de datos sigue los siguientes pasos:

- **Ciudades:** Se utilizaron las ciudades de Perú, como Huanuco, Iquitos, Piura, Tumbes, Loreto, Lima, Tacna y Ayacucho. Para distribuir los médicos entre estas ciudades, se asignó un número de médicos por ciudad proporcional al total de registros requeridos.
- **Campos:**
 - **Dni:** Se generaron números únicos de 8 dígitos para cada médico.
 - **Nombre y Apellidos:** Utilizando **Faker**, se generaron nombres y apellidos aleatorios.
 - **Especialidad:** Se asignaron especialidades médicas aleatorias como Cardiología, Pediatría, Neurología, Ginecología y Dermatología.
 - **Número de Colegiado:** Se generaron números de colegiado únicos de 6 dígitos.
 - **Centro de Salud:** Se utilizaron nombres aleatorios de empresas, generados por **Faker**.

- Ciudad: Los médicos se distribuyeron aleatoriamente entre las ciudades previamente mencionadas.

El código en Python para la generación de los datos para la tabla Medico es el siguiente:

```

1 import csv
2 from faker import Faker
3 import random
4
5 # Configuración de Faker
6 fake = Faker('es_ES')
7 especialidades = ['Cardiología', 'Pediatría', 'Neurología', '
    Ginecología', 'Dermatología']
8 ciudades = ['Huanuco', 'Iquitos', 'Piura', 'Tumbes', 'Loreto', 'Lima', '
    Tacna', 'Ayacucho']
9
10 # Generar registros para la tabla Medico
11 def generar_medico_csv(nombre_archivo, n):
12     # Crear una lista de ciudades repetidas equitativamente
13     ciudades_repartidas = []
14     repeticiones_por_ciudad = n // len(ciudades) # Cantidad de médicos
    por ciudad
15     for ciudad in ciudades:
16         ciudades_repartidas.extend([ciudad] * repeticiones_por_ciudad)
17     random.shuffle(ciudades_repartidas) # Barajar las ciudades para
    distribuirlas aleatoriamente
18
19     with open(nombre_archivo, mode='w', newline='', encoding='utf-8') as
    file:
20         writer = csv.writer(file)
21         # Escribir encabezados
22         writer.writerow(['Dni', 'Nombre', 'Apellidos', 'Especialidad', '
    NumColegiado', 'CentroSalud', 'Ciudad'])
23
24         for i in range(n):
25             dni = fake.unique.random_number(digits=8)
26             nombre = fake.first_name()
27             apellidos = fake.last_name()
28             especialidad = random.choice(especialidades)
29             num_colegiado = fake.unique.random_number(digits=6)
30             centro_salud = fake.company()
31             ciudad = ciudades_repartidas[i]
32             # Escribir registro
33             writer.writerow([dni, nombre, apellidos, especialidad,
    num_colegiado, centro_salud, ciudad])
34         print(f"Archivo CSV generado: {nombre_archivo}")
35
36 # Generar archivo para 1000 registros
37 generar_medico_csv("medico.csv", 1000)

```

Se generaron **1000 registros** para la tabla **Medico**.

1.2.2. Inserción de Datos para la Tabla Medico

Insertamos los datos de `medico.csv` en PGAdmin:

```
1 COPY Medico(Dni, Nombre, Apellidos, Especialidad, NumColegiado,
  CentroSalud, Ciudad)
2 FROM 'C:/Users/Public/bd2/medico.csv'
3 DELIMITER ','
4 CSV HEADER;
```

1.2.3. Generación de Datos para la Tabla Diagnóstico

El archivo CSV de diagnósticos fue generado considerando la relación entre médicos y pacientes. Los pasos para la generación de los datos son los siguientes:

- **Médicos:** Se extrajeron los DNIs y las ciudades de los médicos del archivo `medico.csv`.
- **Diagnósticos:** Se asignaron diagnósticos médicos de una lista predeterminada que incluye Diabetes, Obesidad, Hipertensión y Cardiopatía.
- **Campos:**
 - **id:** Se asignó un identificador único para cada diagnóstico.
 - **Dni_paciente:** Se generaron números de DNI aleatorios para los pacientes.
 - **Dni_medico:** Se seleccionaron aleatoriamente médicos de la tabla **Medico**.
 - **Ciudad:** Se asignó la ciudad del médico correspondiente.
 - **Diagnóstico:** Se eligió aleatoriamente un diagnóstico de la lista.
 - **Peso (kg):** Se generó un peso aleatorio entre 50 y 120 kilogramos.
 - **Edad:** Se asignó una edad aleatoria entre 20 y 90 años.
 - **Sexo:** Se asignó un sexo aleatorio (M o F).

El código en Python para la generación de los datos para la tabla **Diagnóstico** es el siguiente:

```
1 def generar_diagnostico_csv(nombre_archivo, archivo_medico, n):
2     # Leer los DNIs y ciudades de los médicos
3     medicos = []
4     with open(archivo_medico, mode='r', encoding='utf-8') as file:
5         reader = csv.DictReader(file)
6         for row in reader:
```



```

7         medicos.append((row['Dni'], row['Ciudad']))
8
9     diagnosticos = ['Diabetes', 'Obesidad', 'Hipertensi n', 'Cardiopat a
10    ']
11
12     with open(nombre_archivo, mode='w', newline='', encoding='utf-8') as
13     file:
14         writer = csv.writer(file)
15         # Escribir encabezados
16         writer.writerow(['id', 'Dni_paciente', 'Dni_medico', 'Ciudad', '
17         Diagnostico', 'Peso_kg', 'Edad', 'Sexo'])
18
19         for i in range(1, n + 1):
20             dni_paciente = fake.unique.random_number(digits=8)
21             dni_medico, ciudad = random.choice(medicos)
22             diagnostico = random.choice(diagnosticos)
23             peso = round(random.uniform(50, 120), 2)
24             edad = random.randint(20, 90)
25             sexo = random.choice(['F', 'M'])
26             # Escribir registro
27             writer.writerow([i, dni_paciente, dni_medico, ciudad,
28             diagnostico, peso, edad, sexo])
29         print(f"Archivo CSV generado: {nombre_archivo}")
30
31 # Generar archivo para 1000 registros de Diagnostico
32 generar_diagnostico_csv("diagnostico.csv", "medico.csv", 1000)

```

Se generaron **1000 registros** para la tabla Diagnóstico.

1.2.4. Inserción de Datos para la Tabla Medico

Insertamos los datos de diagnostico.csv en PGAdmin:

```

1 COPY Diagnostico(id, Dni_paciente, Dni_medico, Ciudad, Diagnostico,
2     Peso_kg, Edad, Sexo)
3 FROM 'C:/Users/Public/bd2/diagnostico.csv'
4 DELIMITER ','
5 CSV HEADER;

```

1.3. Distribución de Registros en los Fragmentos

Para analizar la distribución usamos el código:

```

1 SELECT * FROM Medico;
2 SELECT ciudad, count(*) from medico group by ciudad
3 --SELECT hashtext('Huanuco');--0 es el resultado de aplicar mod 8

```

```

4 --SELECT hashtext('Lima');--5
5 --SELECT hashtext('Loreto');--4
6 --SELECT hashtext('Tumbes');--3
7 --SELECT hashtext('Piura');--2
8 --SELECT hashtext('Ayacucho');--7
9 --SELECT hashtext('Tacna');--6
10 --SELECT hashtext('Iquitos');--1
11
12 --Esto fue lo m s equitativo posible que pudimos obtener
13 --despues de muchos intentos para que ningun fragmento se quedara vacio:
14 SELECT * FROM medico_1;--250
15 SELECT * FROM medico_2;--250
16 SELECT * FROM medico_3;--375
17 SELECT * FROM medico_4;--125
18
19 SELECT * FROM Diagnostico;
20 SELECT * FROM diagnostico_1;--267
21 SELECT * FROM diagnostico_2;--247
22 SELECT * FROM diagnostico_3;--356
23 SELECT * FROM diagnostico_4;--130

```

1.3.1. Distribución de Registros en la Tabla Medico

- medico_1: 250 registros
- medico_2: 250 registros
- medico_3: 375 registros
- medico_4: 125 registros

Este comportamiento es común en fragmentaciones por hash, donde la distribución depende de cómo los valores del atributo **Ciudad** se mapean a los valores de **REMAINDER**. En este caso, el fragmento **medico_3** contiene más registros que los otros, lo que puede ser resultado de una concentración de valores de **Ciudad** que caen en el **REMAINDER 2**. Mientras tanto, **medico_4** tiene el menor número de registros, lo que indica que pocos valores de **Ciudad** mapean a este fragmento.

1.3.2. Distribución de Registros en la Tabla Diagnostico

- diagnostico_1: 267 registros
- diagnostico_2: 247 registros
- diagnostico_3: 356 registros
- diagnostico_4: 130 registros

Como los registros de `Diagnostico` están relacionados con los registros de `Medico` por el atributo `Ciudad`, los mismos valores de `REMAINDER` se aplican a los fragmentos correspondientes de `Diagnostico`. Así, el fragmento `diagnostico_3` contiene la mayor cantidad de registros, con 356 registros, y el fragmento `diagnostico_4` contiene solo 130 registros.

2. P1. Algoritmos distribuidos localmente

Nos piden diseñar el algoritmo distribuido optimizado para cada consulta, entonces:

- Primero diseñamos las sentencias SQL que implementa el algoritmo distribuido optimizado, también encerramos nuestro bloque de transacción (`begin`, `commit`).
- También usamos el comando `CREATE TEMPORARY TABLE` para la creación de particiones intermedias y las mismas son eliminadas una vez entregado el resultado.

2.1. Consulta 1

2.1.1. Sentencia SQL de la implementación de la consulta optimizada

```
1 -- Consulta a: Select * From Diagnostico Order By Ciudad.
2 BEGIN;
3
4 -- Creamos una tabla temporal para consolidar los resultados de las
   particiones
5 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
6 CREATE TEMPORARY TABLE temp_diagnostico AS
7 SELECT * FROM diagnostico_1
8 UNION ALL
9 SELECT * FROM diagnostico_2
10 UNION ALL
11 SELECT * FROM diagnostico_3
12 UNION ALL
13 SELECT * FROM diagnostico_4;
14
15 -- Realizamos la consulta ordenada por Ciudad sobre la tabla temporal.
16 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
17 SELECT * FROM temp_diagnostico ORDER BY Ciudad;
18
19 -- Eliminamos la tabla temporal
20 DROP TABLE temp_diagnostico;
21
22 COMMIT;
```

2.1.2. Output

Notifications

Data Output

</

Figura 1: Output de la consulta 1

2.1.3. Gráfico del plan de ejecución resultante

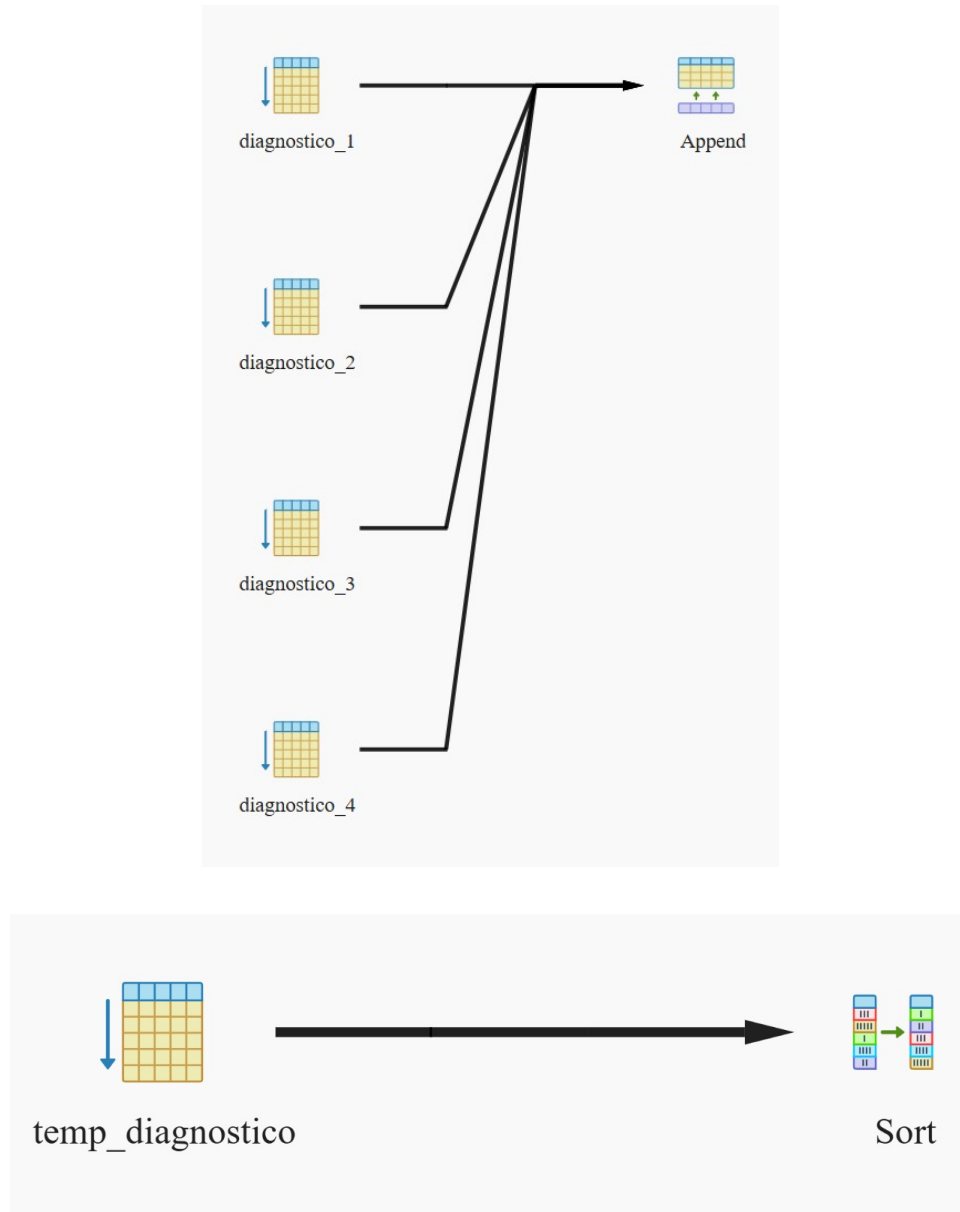


Figura 2: Gráfico del plan de ejecución de la Consulta 1

2.2. Consulta 2

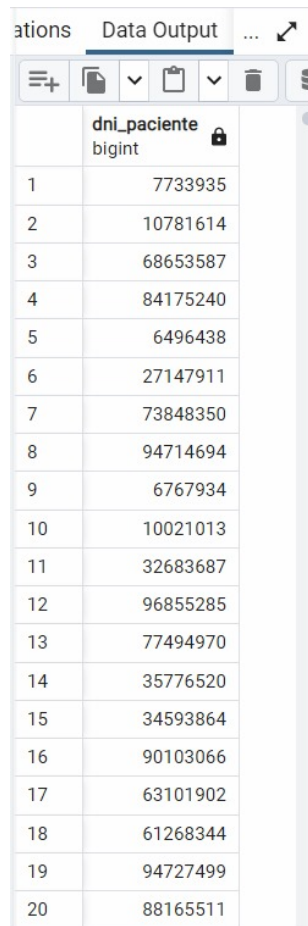
2.2.1. Sentencia SQL de la implementación de la consulta optimizada

```

1 --Consulta b: Select distinct DNI_Paciente From Diagnostico.
2 BEGIN;
3
4 -- Consolidar resultados de las particiones en una tabla temporal
5 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
6 CREATE TEMPORARY TABLE temp_dni_paciente AS
7 SELECT DNI_Paciente FROM diagnostico_1
8 UNION ALL
9 SELECT DNI_Paciente FROM diagnostico_2
10 UNION ALL
11 SELECT DNI_Paciente FROM diagnostico_3
12 UNION ALL
13 SELECT DNI_Paciente FROM diagnostico_4;
14
15 -- Obtener los valores nicos
16 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
17 SELECT DISTINCT DNI_Paciente FROM temp_dni_paciente;
18
19 -- Eliminar la tabla temporal
20 DROP TABLE temp_dni_paciente;
21
22 COMMIT;

```

2.2.2. Output



The image shows a screenshot of a database query output window. The window has a title bar with 'ations' and 'Data Output' tabs. Below the title bar is a toolbar with icons for adding, deleting, and other operations. The main area displays a table with two columns: an index from 1 to 20 and a column labeled 'dni_paciente' with a 'bigint' data type and a lock icon. The table contains 20 rows of patient DNI numbers.

	dni_paciente bigint
1	7733935
2	10781614
3	68653587
4	84175240
5	6496438
6	27147911
7	73848350
8	94714694
9	6767934
10	10021013
11	32683687
12	96855285
13	77494970
14	35776520
15	34593864
16	90103066
17	63101902
18	61268344
19	94727499
20	88165511

Figura 3: Output de la consulta 2

2.2.3. Gráfico del plan de ejecución resultante

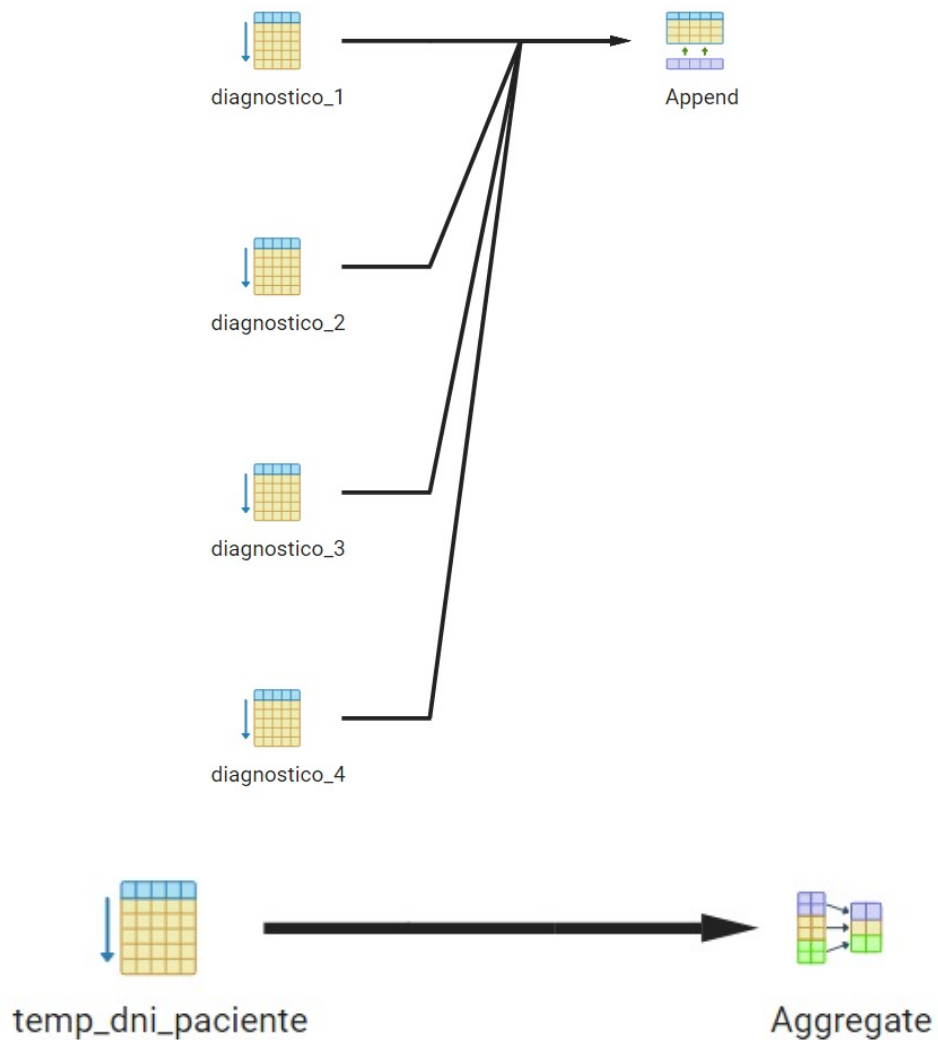


Figura 4: Gráfico del plan de ejecución de la Consulta 2

2.3. Consulta 3

2.3.1. Sentencia SQL de la implementación de la consulta optimizada

```
1 --Consulta c: Select Edad, Count(*) From Diagnostico Group By Edad
2 BEGIN;
3
4 -- Consolidar resultados de las particiones en una tabla temporal
```

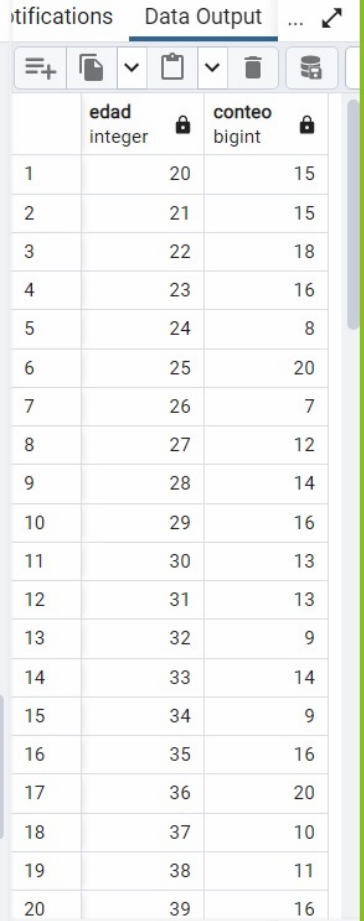


```

5 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
6 CREATE TEMPORARY TABLE temp_group_edad AS
7 SELECT Edad FROM diagnostico_1
8 UNION ALL
9 SELECT Edad FROM diagnostico_2
10 UNION ALL
11 SELECT Edad FROM diagnostico_3
12 UNION ALL
13 SELECT Edad FROM diagnostico_4;
14
15 -- Realizar la agrupaci n y contar registros por edad
16 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
17 SELECT Edad, COUNT(*) AS conteo FROM temp_group_edad
18 GROUP BY Edad
19 ORDER BY Edad;--lo quisimos ordenar asi
20
21 -- Eliminar la tabla temporal
22 DROP TABLE temp_group_edad;
23
24 COMMIT;

```

2.3.2. Output



The screenshot shows a database query output window with a tab labeled "Data Output". The window displays a table with 20 rows and 3 columns. The first column is an index from 1 to 20. The second column is labeled "edad" with a data type of "integer". The third column is labeled "conteo" with a data type of "bigint". The table contains the following data:

	edad integer	conteo bigint
1	20	15
2	21	15
3	22	18
4	23	16
5	24	8
6	25	20
7	26	7
8	27	12
9	28	14
10	29	16
11	30	13
12	31	13
13	32	9
14	33	14
15	34	9
16	35	16
17	36	20
18	37	10
19	38	11
20	39	16

Figura 5: Output de la consulta 3

2.3.3. Gráfico del plan de ejecución resultante

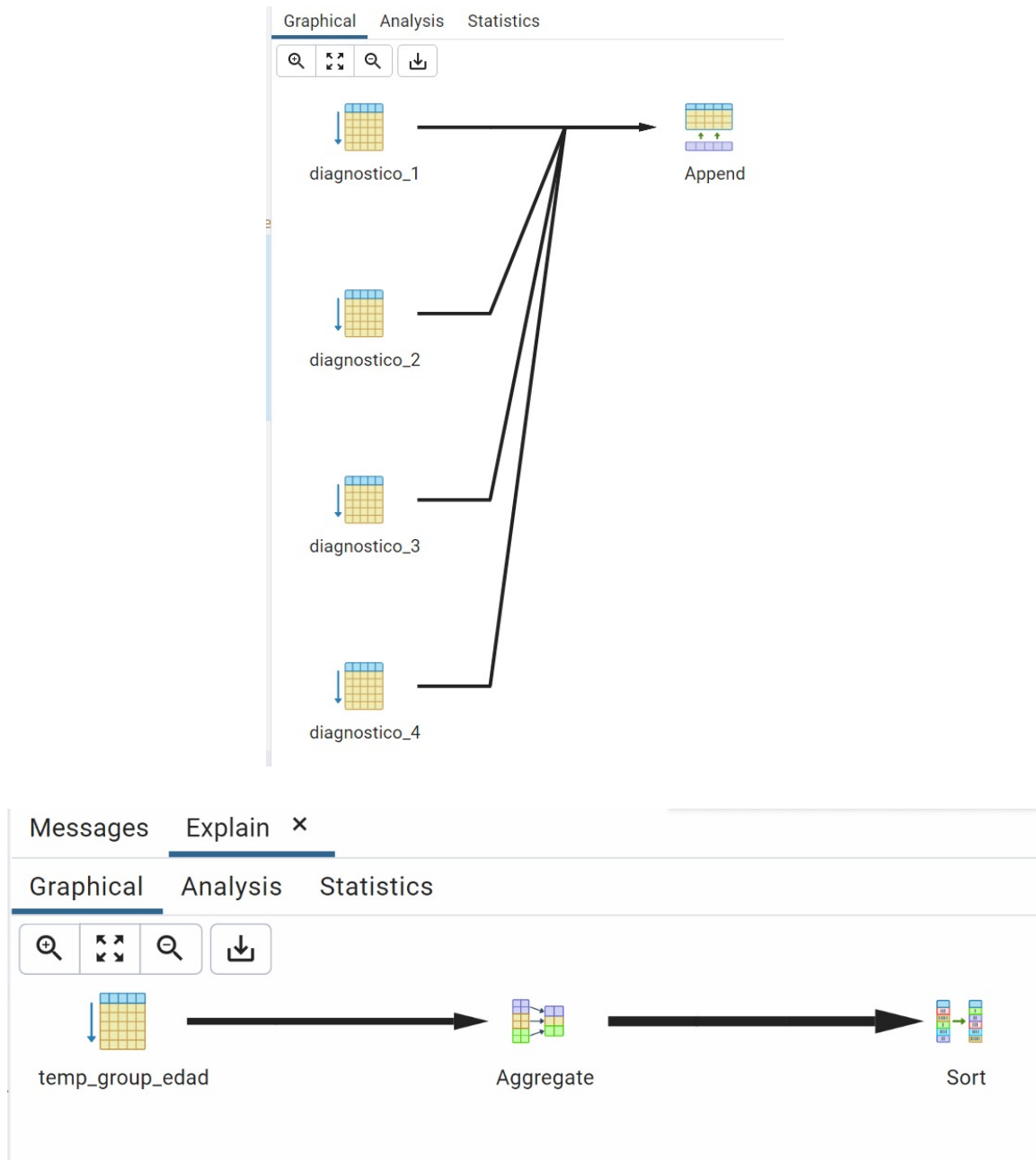


Figura 6: Gráfico del plan de ejecución de la Consulta 3

2.4. Consulta 4

2.4.1. Sentencia SQL de la implementación de la consulta optimizada

```
1  --Consulta d: Select Especialidad, Count(*) From Medico M Join Diagnostico
   D on M.DNI = D.DNI_Medico
2  BEGIN;
3
4  -- Consolidar particiones de Medico
5  --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
6  CREATE TEMPORARY TABLE temp_medico AS
7  SELECT * FROM medico_1
8  UNION ALL
9  SELECT * FROM medico_2
10 UNION ALL
11 SELECT * FROM medico_3
12 UNION ALL
13 SELECT * FROM medico_4;
14
15 -- Consolidar particiones de Diagnostico
16 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
17 CREATE TEMPORARY TABLE temp_diagnostico AS
18 SELECT * FROM diagnostico_1
19 UNION ALL
20 SELECT * FROM diagnostico_2
21 UNION ALL
22 SELECT * FROM diagnostico_3
23 UNION ALL
24 SELECT * FROM diagnostico_4;
25
26 -- Realizar el JOIN y la agrupaci n
27 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
28 SELECT M.Especialidad, COUNT(*) AS conteo
29 FROM temp_medico M
30 JOIN temp_diagnostico D ON M.Dni = D.Dni_medico
31 GROUP BY M.Especialidad
32 ORDER BY conteo;
33
34 -- Eliminar tablas temporales
35 DROP TABLE temp_medico;
36 DROP TABLE temp_diagnostico;
37
38 COMMIT;
```

2.4.2. Output









Notifications		Data Output	
			
			
	especialidad character varying	conteo bigint	
1	Cardiología	176	
2	Dermatología	181	
3	Pediatría	204	
4	Neurología	213	
5	Ginecología	226	

Figura 7: Output de la consulta 4

2.4.3. Gráfico del plan de ejecución resultante

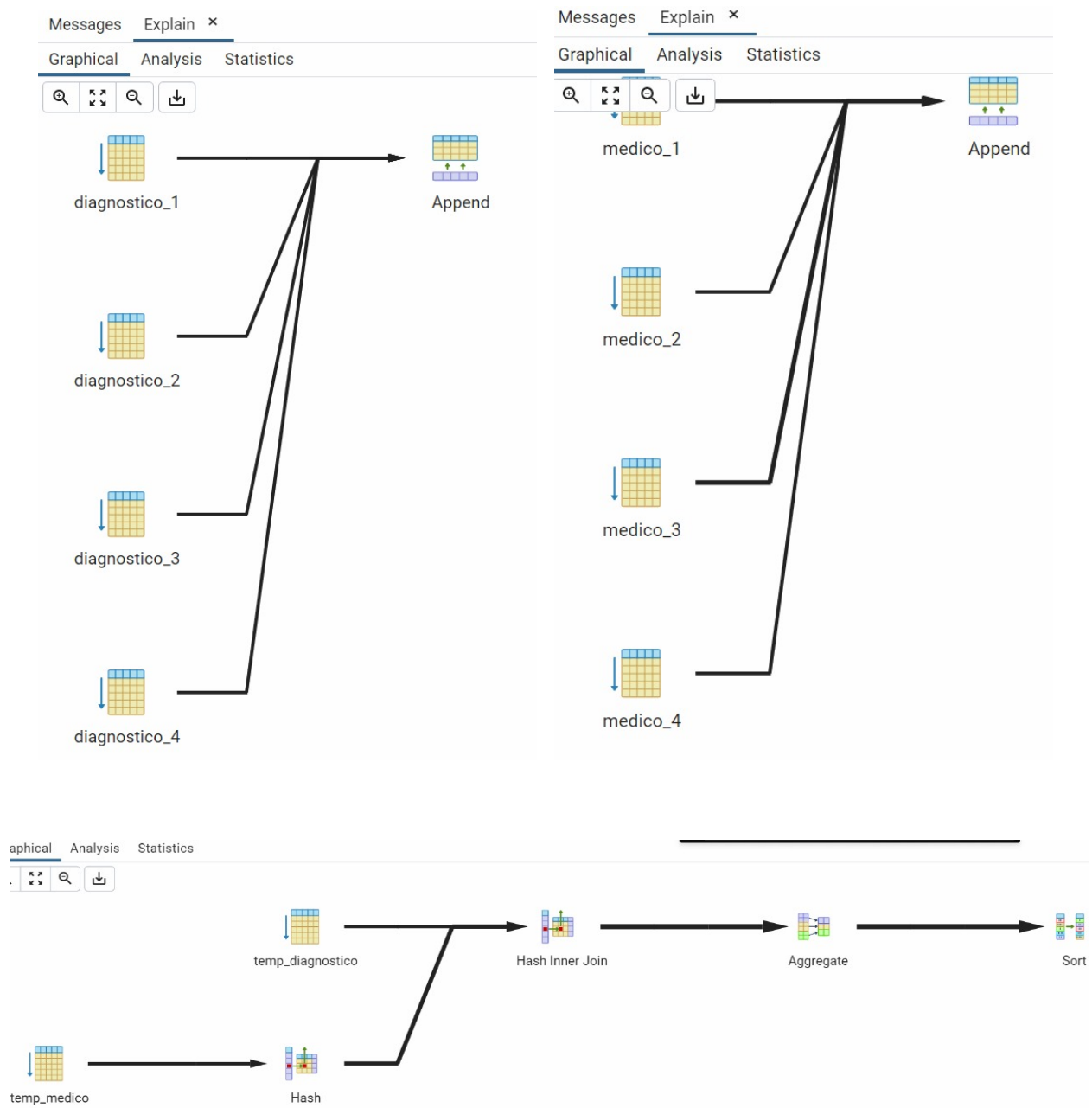


Figura 8: Gráfico del plan de ejecución de la Consulta 4

3. P2. Algoritmos distribuidos en al menos tres servidores

3.1. Creación de los servidores

Para esta parte, elaboramos un archivo [docker-compose.yml](#). Este archivo define un entorno con **3 servicios** en Docker Compose para una configuración de bases de datos PostgreSQL. A continuación, se detalla cada componente:

■ Servidor Master:

- Actúa como el nodo principal donde se centralizan las operaciones.
- Usa la imagen `postgres:latest`.
- Configurado con:
 - Usuario: `user1`.
 - Contraseña: `123456`.
 - Base de datos: `db1`.
- Expone el puerto **5501** en la máquina host.

■ Servidores Worker:

- Se incluyen dos nodos secundarios que distribuyen la carga de trabajo, cada uno con configuraciones independientes:
 - **Worker1:**
 - ◇ Usuario: `user2`.
 - ◇ Contraseña: `123456`.
 - ◇ Base de datos: `db2`.
 - ◇ Puerto expuesto: **5502**.
 - **Worker2:**
 - ◇ Usuario: `user3`.
 - ◇ Contraseña: `123456`.
 - ◇ Base de datos: `db3`.
 - ◇ Puerto expuesto: **5503**.

■ Red de comunicación:

- Los servicios están conectados mediante una red Docker llamada `postgres-network`.
- Se utiliza el driver `bridge` para establecer una comunicación privada entre los contenedores.

Posteriormente, procedemos a ejecutar el comando `docker-compose up -d` para subir los servidores en Docker Desktop y se pueden ver que han sido creados en la imagen a continuación:

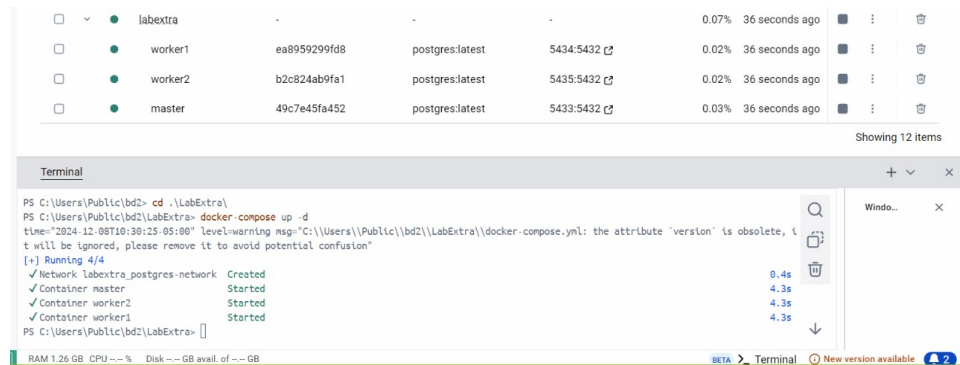


Figura 9: Gráfico los servidores creados en Docker

Luego registramos los servidores en PGAdmin como se muestra en la imagen:

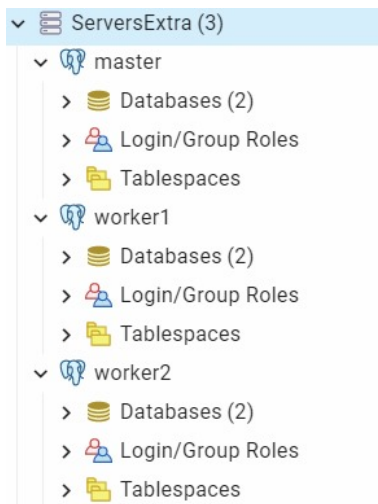


Figura 10: Gráfico los servidores registrados en PGAdmin

3.2. Master

Nuestro código [master_p2.sql](#) configura un entorno en el servidor **Master** para interactuar con los servidores **Worker1** y **Worker2** utilizando **postgres_fdw** (PostgreSQL Foreign Data Wrapper). Esta configuración permite importar datos de bases de datos remotas y realizar consultas distribuidas. A continuación, se describen los elementos principales:

- **Extensión `postgres_fdw`:**

- Se activa la extensión `postgres_fdw`, necesaria para interactuar con bases de datos externas.

- **Creación de servidores remotos:**

- Se definen los servidores `worker1` y `worker2`, especificando sus direcciones (host y puerto) y la base de datos asociada (`db2` para `worker1` y `db3` para `worker2`).

- **Mapeos de usuario:**

- Se crean mapeos de usuario que permiten al usuario `user1` del **Master** acceder a las bases de datos remotas como `user2` en `worker1` y `user3` en `worker2`.

- **Importación de esquemas:**

- Se importan esquemas específicos (`medico_1`, `medico_2`, `diagnostico_1`, etc.) desde los servidores remotos al esquema público del **Master**.

3.3. Worker 1

En nuestro código [worker1_p2.sql](#) se crean las primeras dos particiones de las tablas `Medico` y `Diagnostico`, organizadas por ciudad. Las tablas `medico_1` y `medico_2` almacenan información sobre médicos, mientras que `diagnostico_1` y `diagnostico_2` contienen diagnósticos médicos asociados.

- **Estructura:**

- Claves primarias incluyen `Dni` y `Ciudad` (para médicos) o `id` y `Ciudad` (para diagnósticos).
- Restricciones garantizan integridad referencial y valores válidos para `Sexo` ('M' o 'F').

- **Datos insertados:**

- `medico_1` y `medico_2` incluyen médicos en Trujillo, Huancayo, Cusco, y Chiclayo.
- `diagnostico_1` y `diagnostico_2` registran diagnósticos vinculados a estos médicos.

3.4. Worker 2

En nuestro código [worker2_p2.sql](#) se crean las otras 2 particiones para las tablas `Medico` y `Diagnostico`, también organizadas por ciudad. Las tablas `medico_3` y `medico_4` contienen información de médicos, mientras que `diagnostico_3` y `diagnostico_4` registran los diagnósticos médicos.

■ Estructura:

- Claves primarias incluyen `Dni` y `Ciudad` (para médicos) o `id` y `Ciudad` (para diagnósticos).
- Restricciones garantizan integridad referencial y valores válidos para `Sexo` ('M' o 'F').

■ Datos insertados:

- `medico_3` y `medico_4` incluyen médicos en Tacna, Piura, Lima, y Arequipa.
- `diagnostico_3` y `diagnostico_4` contienen diagnósticos vinculados a estos médicos.

3.5. Consulta 1

```
1 -- Consulta a): Select * From Diagnostico Order By Ciudad.
2
3 BEGIN;
4
5 -- Crear tabla temporal para consolidar particiones
6 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
7 CREATE TEMPORARY TABLE temp_diagnostico AS
8 SELECT * FROM diagnostico_1
9 UNION ALL
10 SELECT * FROM diagnostico_2
11 UNION ALL
12 SELECT * FROM diagnostico_3
13 UNION ALL
14 SELECT * FROM diagnostico_4;
15
16 -- Realizar la consulta ordenada
17 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
18 SELECT * FROM temp_diagnostico ORDER BY Ciudad;
19
20 -- Eliminar la tabla temporal
21 DROP TABLE temp_diagnostico;
22
23 COMMIT;
```

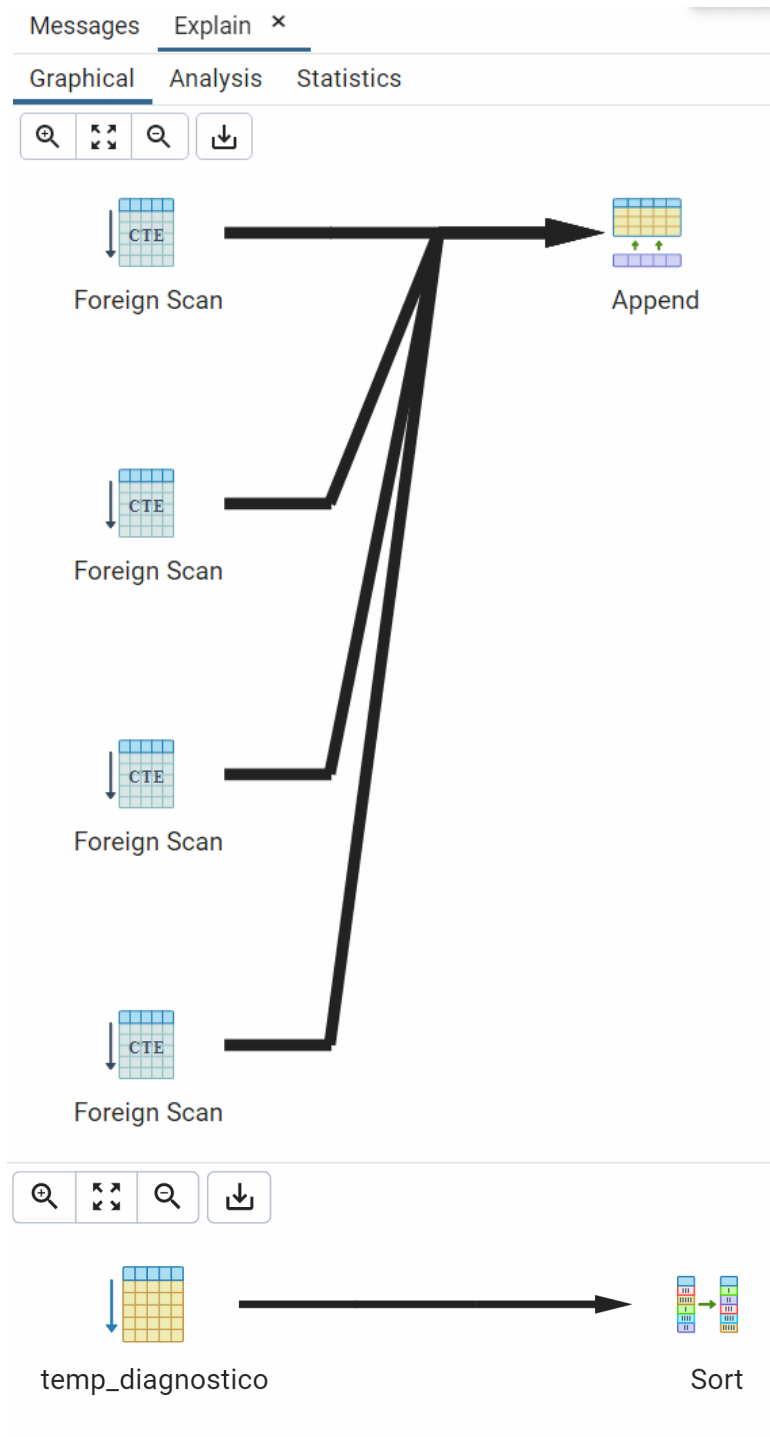


Figura 11: Gráfico del plan de ejecucion de la Consulta 1

3.6. Consulta 2

```
1 --Consulta b: Select distinct DNI_Paciente From Diagnostico.
2
3 BEGIN;
4
5 -- Crear tabla temporal para consolidar DNI_Paciente
6 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
7 CREATE TEMPORARY TABLE temp_dni_paciente AS
8 SELECT DNI_Paciente FROM diagnostico_1
9 UNION ALL
10 SELECT DNI_Paciente FROM diagnostico_2
11 UNION ALL
12 SELECT DNI_Paciente FROM diagnostico_3
13 UNION ALL
14 SELECT DNI_Paciente FROM diagnostico_4;
15
16 -- Realizar la consulta DISTINCT
17 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
18 SELECT DISTINCT DNI_Paciente FROM temp_dni_paciente;
19
20 -- Eliminar la tabla temporal
21 DROP TABLE temp_dni_paciente;
22
23 COMMIT;
```

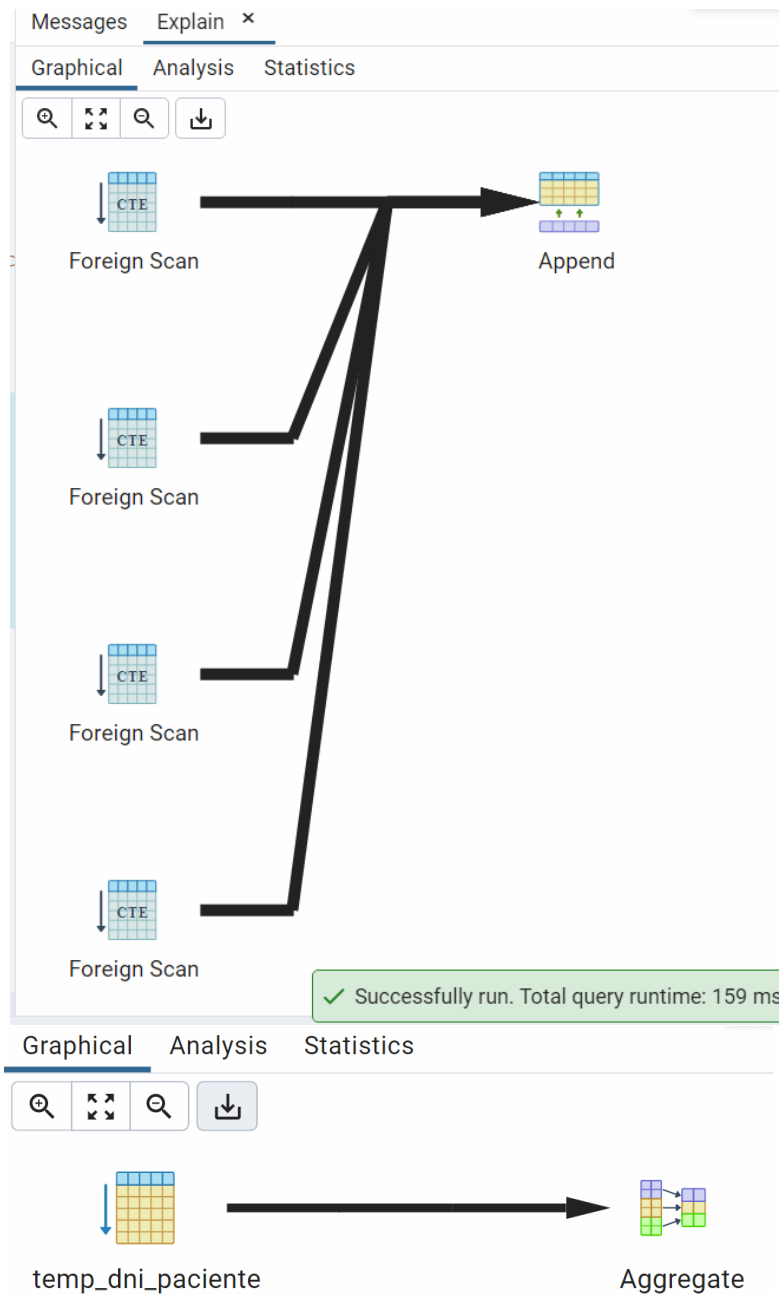


Figura 12: Gráfico del plan de ejecucion de la Consulta 2

3.7. Consulta 3

```

1 --Consulta c: Select Edad, Count(*) From Diagnostico Group By Edad
2 BEGIN;
3
4 -- Crear tabla temporal para consolidar Edad
5 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
6 CREATE TEMPORARY TABLE temp_group_edad AS
7 SELECT Edad FROM diagnostico_1
8 UNION ALL
9 SELECT Edad FROM diagnostico_2
10 UNION ALL
11 SELECT Edad FROM diagnostico_3
12 UNION ALL
13 SELECT Edad FROM diagnostico_4;
14
15 -- Realizar la agrupaci n y contar registros
16 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
17 SELECT Edad, COUNT(*) AS conteo FROM temp_group_edad
18 GROUP BY Edad
19 ORDER BY Edad;
20
21 -- Eliminar la tabla temporal
22 DROP TABLE temp_group_edad;
23
24 COMMIT;

```

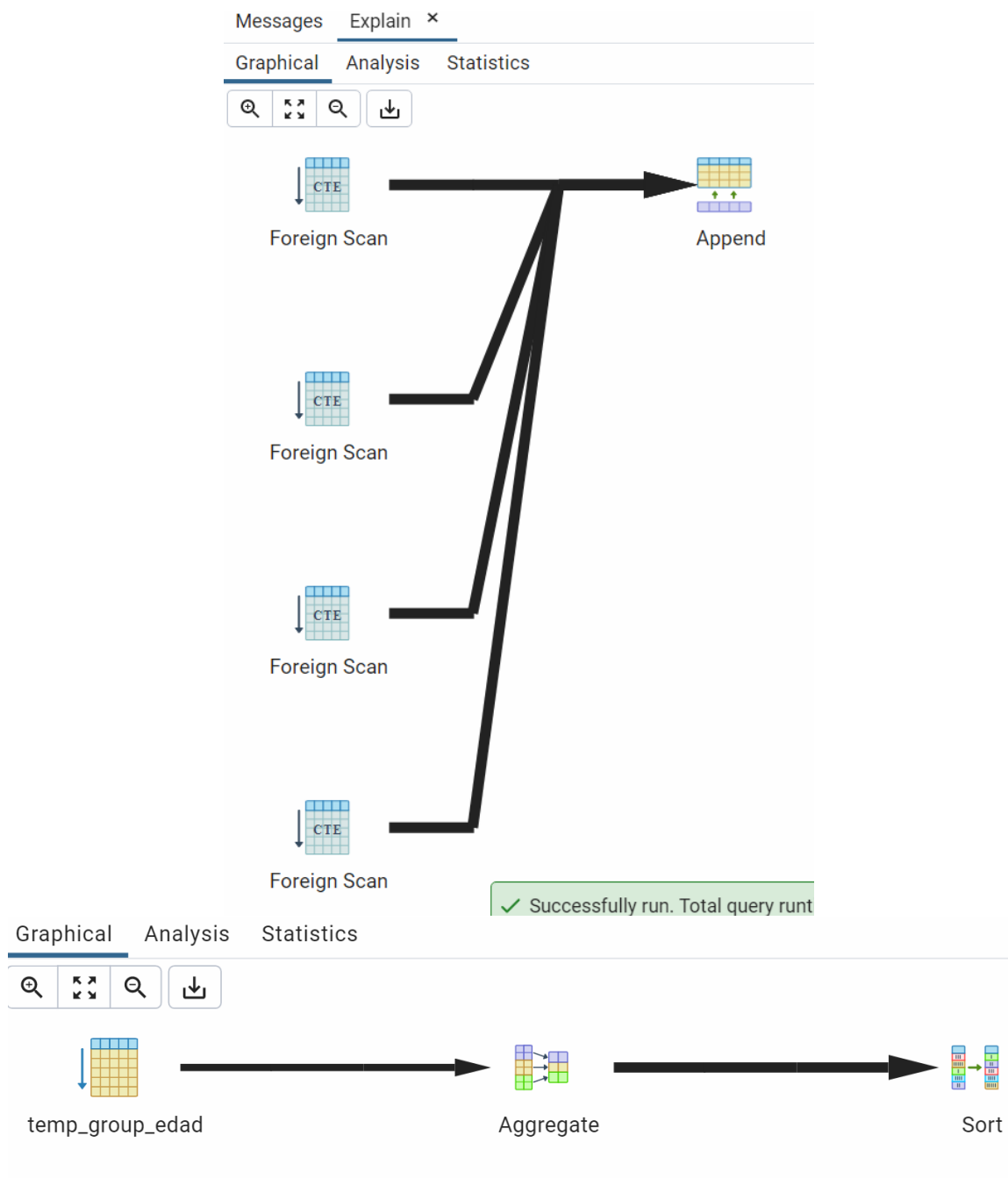


Figura 13: Gráfico del plan de ejecucion de la Consulta 3

3.8. Consulta 4

```

1  --Consulta d: Select Especialidad, Count(*) From Medico M Join Diagnostico
   D on M.DNI = D.DNI_Medico
2  BEGIN;
3
4  -- Consolidar las particiones de Medico y Diagnostico
5  --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
6  CREATE TEMPORARY TABLE temp_medico AS
7  SELECT * FROM medico_1
8  UNION ALL
9  SELECT * FROM medico_2
10 UNION ALL
11 SELECT * FROM medico_3
12 UNION ALL
13 SELECT * FROM medico_4;
14
15 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
16 CREATE TEMPORARY TABLE temp_diagnostico AS
17 SELECT * FROM diagnostico_1
18 UNION ALL
19 SELECT * FROM diagnostico_2
20 UNION ALL
21 SELECT * FROM diagnostico_3
22 UNION ALL
23 SELECT * FROM diagnostico_4;
24
25 -- Realizar JOIN y agrupaci n
26 --EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
27 SELECT M.Especialidad, COUNT(*) AS conteo
28 FROM temp_medico M
29 JOIN temp_diagnostico D ON M.Dni = D.Dni_medico
30 GROUP BY M.Especialidad
31 ORDER BY conteo;
32
33 -- Eliminar tablas temporales
34 DROP TABLE temp_medico;
35 DROP TABLE temp_diagnostico;
36
37 COMMIT;

```

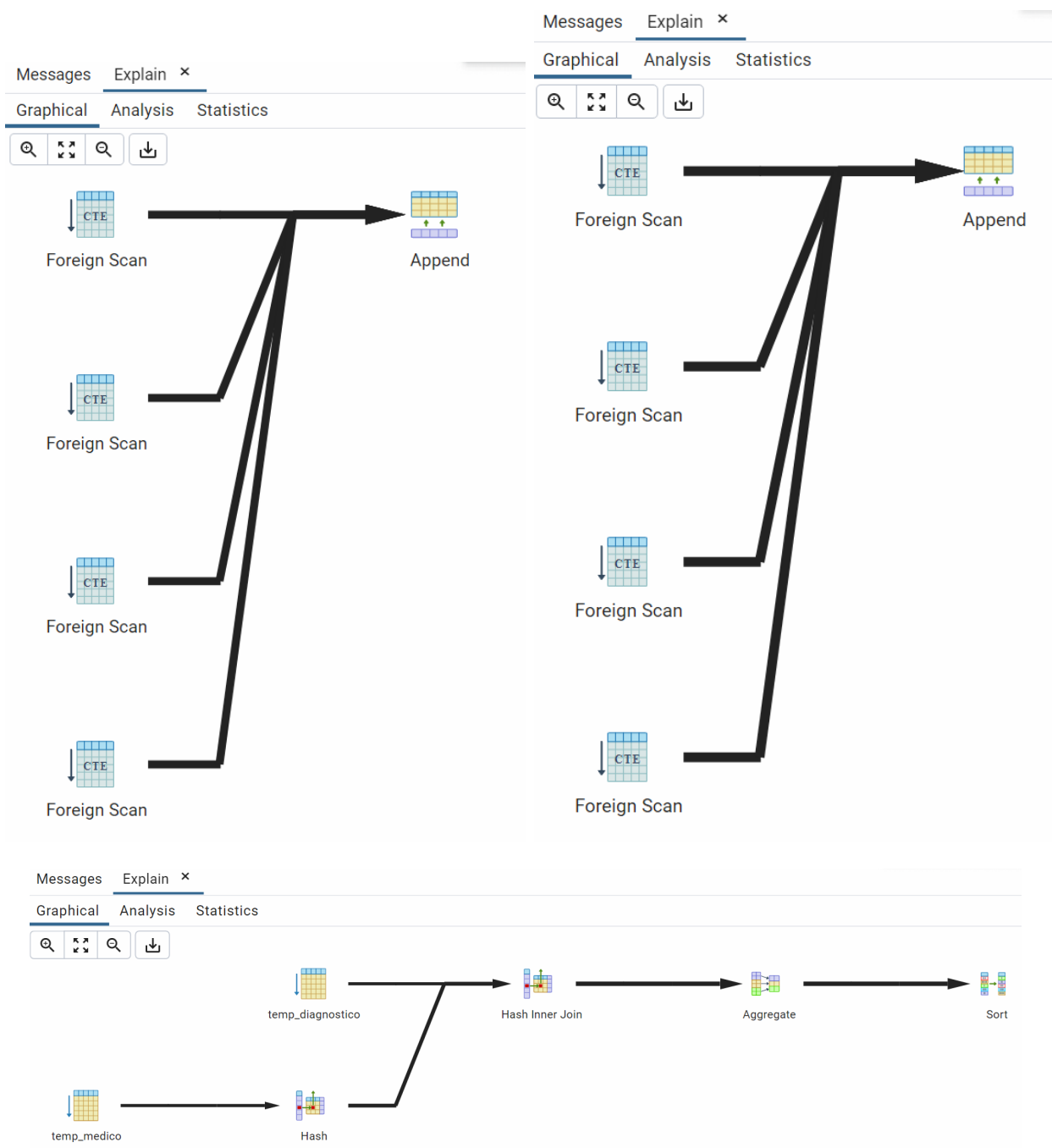



Figura 14: Gráfico del plan de ejecución de la Consulta 4