UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE INTERFAZ PARA CONSULTAS SQL SOBRE ALGORITMOS RECURSIVOS PARA GRAFOS EN SISTEMAS DE BASES DE DATOS RELACIONALES

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

LUIS PATRICIO BUSTOS CARRASCO

PROFESOR GUÍA:
CLAUDIO GUTIÉRREZ GALLARDO

MIEMBROS DE LA COMISIÓN:
GONZALO NAVARRO BADINO
DIONISIO GONZÁLEZ GONZÁLEZ

SANTIAGO DE CHILE
2024

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL
EN COMPUTACIÓN
POR: LUIS PATRICIO BUSTOS CARRASCO
FECHA: 2024
PROF. GUÍA: CLAUDIO GUTIÉRREZ

IMPLEMENTACIÓN DE INTERFAZ PARA CONSULTAS SQL SOBRE ALGORITMOS RECURSIVOS PARA GRAFOS EN SISTEMAS DE BASES DE DATOS RELACIONALES

SQL es un lenguaje de consulta basada en el uso de álgebra relacional. Si bien el álgebra relacional no permite la expresión de consultas recursivas, existen distintas funcionalidades implementadas en este lenguaje para poder realizarlas.

La codificación de este tipo de consultas no siempre resulta del todo intuitiva para un programador que desea implementarlas, debido a que la lógica de la recursión en SQL hace que la tabla temporal generada por la consulta recursiva crezca de manera iterativa e incremental, hasta que su tamaño no varía, complejizando la codificación de problemas recursivos en SQL.

Para abordar este problema se propone una codificación genérica para consultas recursivas en bases de datos relacionales que facilite la implementación de estas consultas para un desarrollador. Para probar esta codificación genérica se implementarán consultas recursivas en SQL para los siguientes problemas clásicos de grafos que corren en tiempo polinomial: Topological Sorting, Connected Components, Eulerian Circuit, Minimum Spanning Tree, Eulerian Circuit y Planarity Testing.

Se evaluará la factibilidad de la implementación de la propuesta de recursión generalizada para la simplificación de consultas recursivas para el caso de un motor de bases de datos SQL, mediante la entrega de un orden que permita a una consulta recursiva llegar a un punto fijo.

En este trabajo se realizó la implementación una codificación genérica para consultas recursivas. Junto con esta codificación se implementó una interfaz gráfica que permite la creación de datasets que representan grafos y la codificación de las consultas recursivas sobre los datos generados. Esta interfaz permite la codificación de estas consultas siguiendo el formato de la consulta genérica implementada. También se implementaron consultas recursivas para los 6 problemas propuestos anteriormente sobre la base de la codificación genérica propuesta.

Se obtuvo como resultado que es posible implementar una codificación genérica para consultas recursivas en SQL para bases de datos relacionales. Por otro lado, en el contexto de la evaluación sobre el motor de bases de datos relacionales PostgreSQL, se obtiene el resultado de que es posible entregar un orden dentro de la sub-consulta recursiva, la que se reduce a entregar la profundidad máxima que puede tener la recursión. Además, se concluye que implementación actual de recursión en el contexto de bases de datos relacionales no es idónea para resolver de manera eficiente los problemas propuestos, debiendo requerir de recursos externos para desarrollar ciertos algoritmos y de estar bajo diversas limitaciones respecto a las instrucciones que se pueden entregar en la sub consulta recursiva.

Para mi familia.

Agradecimientos

Quiero agradecer a todos quienes han estado presentes y han dado su apoyo incondicional para poder llegar hasta acá.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Índice de Códigos

Capítulo 1

Introducción

SQL es el lenguaje de bases de datos relacionales más utilizado en el mundo. Este lenguaje de consulta se basa en el uso de álgebra relacional. A pesar de que el álgebra relacional no es capaz de expresar consultas recursivas, debido a la necesidad de su uso, existen diferentes funcionalidades añadidas a SQL que permiten su realización.

Un ejemplo de una consulta recursiva en SQL para la suma de los números enteros entre 1 y 100 se presenta en el código 1.1:

Código 1.1: Consulta recursiva para la suma de los números ente 1 y 100

```
1  WITH RECURSIVE t(n) AS (
2      VALUES (1)
3    UNION
4      SELECT n+1 FROM t WHERE n < 100
5  )
6  SELECT sum(n) FROM t;
```

Si bien la recursión en este tipo de lenguaje de consulta es un tema que ha sido extensamente estudiado tanto teórica como prácticamente [1], la codificación de este tipo de consultas en SQL no resulta del todo intuitiva para el programador. Esto se debe a que la lógica de la recursión en SQL implícitamente hace uso de los resultados intermedios obtenidos como tablas temporales. Esto hace que esta tabla temporal siga creciendo hasta un punto donde el resultado de la tabla temporal obtenida no varía. Esta última tabla es la que SQL entrega como resultado de la consulta recursiva. Esto hace más compleja la codificación de problemas recursivos en SQL [2].

Para resolver este problema, en la tesis de Valentina Urzúa [2] se propone una versión generalizada para la una consulta recursiva en un lenguaje de bases de datos relacional, la cual facilita que un programador pueda realizar este tipo de consultas mediante la especificación de tres sub-consultas: Una consulta base, una consulta recursiva y una tercera consulta que determina el orden al que se ajustará la recursión (que en la versión tradicional es por defecto la inclusión de las diferentes versiones de la tabla temporal).

En esta memoria se mostrará que es posible realizar la implementación de la propuesta del capítulo 3.2.2 de la tesis antes mencionada, para permitir la simplificación de la codificación

de consultas recursivas en SQL.

Para ello, se implementará una interfaz que permita una codificación general de una consulta recursiva en SQL. Mostraremos su aplicación en la implementación de estas consultas para el caso de problemas conocidos de grafos que corren en tiempo polinomial.

Esta interfaz será evaluada mediante su capacidad para responder de manera genérica a diferentes problemas de grafos del tipo indicado. Para esto se desarrollará un mecanismo que permite codificar grafos como tablas SQL.

Con las tablas (grafos) guardadas, se procederá a realizar la codificación de consultas recursivas SQL para diferentes problemas de grafos, en pos de determinar la forma en que se deberá implementar la codificación genérica para estas consultas.

Este documento se organiza de la siguiente manera:

1. En el capítulo 2 se resume el estado del arte en cuanto a la recursión en bases de datos relacionales SQL, sus aplicaciones y la aplicación de consultas recursivas en bases de datos de grafos.

2. En el capítulo 3 se especifican los objetivos que tiene esta memoria y como estos serán evaluados.

3. El capítulo 4 especifica los pasos y consideraciones tomadas para la implementación de la solución propuesta.

4. Posteriormente, en el capítulo 5 se realiza de la evaluación de la implementación de la interfaz realizada para la consulta genérica de consultas recursivas en SQL y de la factibilidad de implementar la codificación genérica propuesta.

5. Finalmente, en el capítulo 6 se presenta la conclusión a partir de los resultados obtenidos en el capítulo anterior y se listan las contribuciones entregadas por este trabajo.

Capítulo 2

Estado del Arte

A continuación se muestra el estado del arte en el que se enmarca este trabajo. Este capítulo se dividirá en tres secciones. En primer lugar, se hará una muestra de como funciona la recursión en SQL para sistemas de bases de datos relacionales. Posteriormente, se hará una revisión de como la recursión ha sido implementada en distintos sistemas de bases de datos relacionales. En segundo lugar, se hará una revisión bibliográfica de la dificultad de la implementación de este tipo de consultas en motores de bases de datos relacionales, tomando como referencia el trabajo realizado en la tesis de Valentina Urzúa [2]. Finalmente, se mostrará la definición y algoritmos conocidos para problemas de grafos polinomiales.

2.1. Recursión en sistemas de bases de datos relacionales

La recursión fue incluida en el estándar SQL-99. [1] bajo la necesidad de expresar consultas mediante una clausura transitiva. Esta se realiza mediante la utilización de la keyword 'RECURSIVE'. Este tipo de consulta se basa en el uso de tablas temporales que se construyen por la acumulación de filas producidas por la consulta recursiva.

Consideremos el siguiente ejemplo de una jerarquía en la figura 2.1. Esta puede ser descrita como una tabla SQL con la siguiente forma:

Tabla 2.1: Tabla jerarquia

padre	**hijo**
A	B
B	D
B	E
E	G
E	H
A	C
C	F

Ahora nuestro problema es encontrar todos los ancestros de un nodo dentro de la jerarquía.

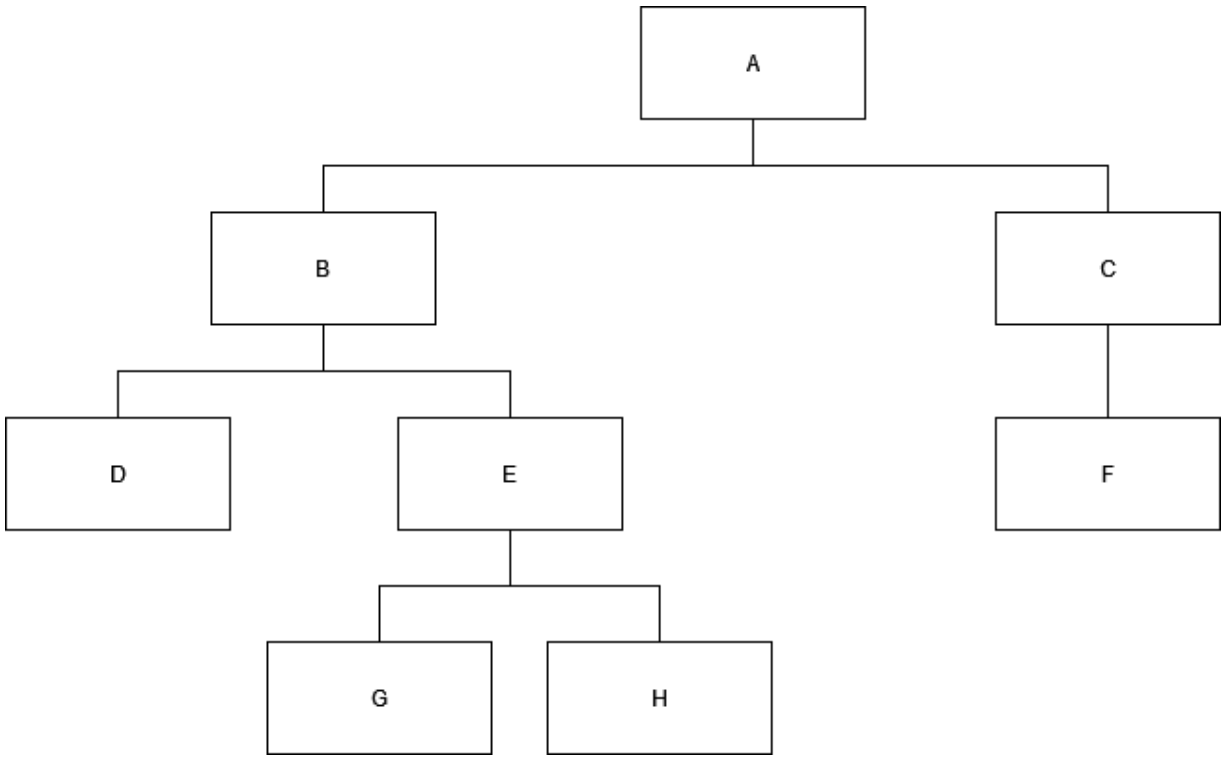

Figura 2.1: Ejemplo jerarquía

Esto puede resolverse con una consulta recursiva de la siguiente forma:

Código 2.1: Consulta recursiva para ancestro en una jerarquia

```
1  WITH RECURSIVE Ancestro(ancestro, descendiente) AS (
2    SELECT
3      padre, hijo
4    FROM
5      Jerarquia
6    UNION
7    SELECT
8      Ancestro.ancestro, Jerarquia.hijo
9    FROM
10     Jerarquia
11   INNER JOIN ON
12     Ancestro.descendiente = Jerarquia.padre
13 );
```

Esto se logra mediante el uso de una consulta base y una consulta recursiva, la cual toma el caso base y comienza a construir una tabla acumulada con los resultados de los pasos anteriores. Para que esta consulta no se ejecute de manera indefinida, se define el concepto de Punto Fijo [3] de la siguiente manera:

> Un punto fijo de una función f es un valor v tal que la función aplicada al valor retorna el mismo valor, es decir, $f(v) = v$

En otras palabras, una consulta recursiva en SQL termina cuando los intentos por añadir más filas a la tabla acumulada no generan modificaciones en esta. Esto se basa en asumir

que la sección recursiva (i.e. la tabla acumulada) de la consulta es monótonamente creciente, es decir, que aumenta su tamaño o se mantiene igual en cada iteración.

Una forma visual de ver como crece la tabla acumulada para la consulta 2.1 aplicado a los datos mostrados en la tabla 2.1 puede encontrarse en la figura 2.2.

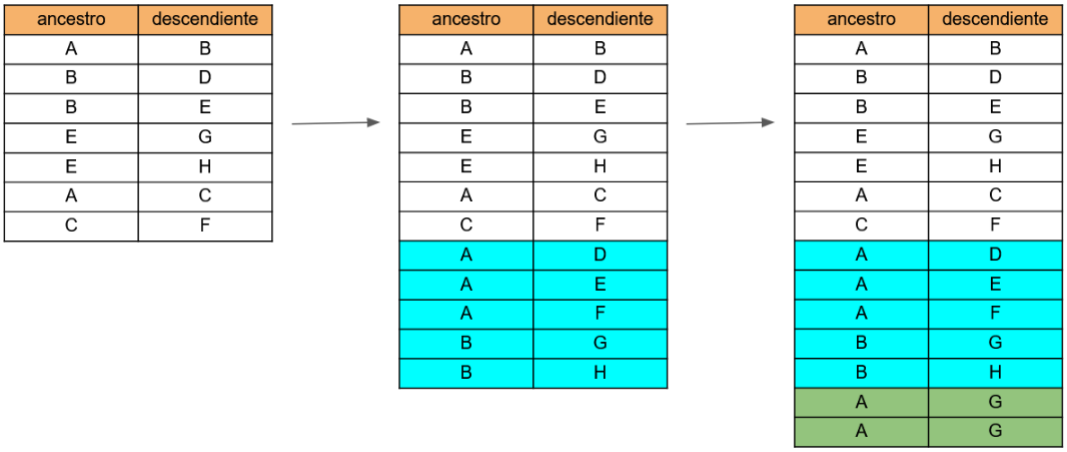

Figura 2.2: Ejemplo de crecimiento de la tabla acumulada para la consulta recursiva del código 2.1

Esto genera una complicación al momento de escribir consultas recursivas en SQL, ya que nos limita a aquellos problemas en donde aquellos resultados intermedios no son necesariamente crecientes. Para el caso de los algoritmos de grafos esto representa un problema particular, ya que muchos de estos problemas utilizan negación y esto puede producir problemas para encontrar el Punto Fijo. [4]. Existen propuestas de extensiones a SQL para soportar algunos los casos que SQL clásico no soporta [5].

Un ejemplo del uso de este tipo de consultas SQL puede verse en el trabajo de Maximilian Schüle[6], en donde se hace la implementación de algoritmos clásicos de Data Mining como consultas recursivas en SQL con el fin de mejorar la performance de estas.

2.1.1. Implementación de la recursión en sistemas de bases de datos relacionales

La implementación de la recursión varía según el sistema elegido. Se hará un recuento de como la recursión está implementada en las 4 bases de datos relacionales más utilizadas en el mercado: Oracle, PostgreSQL, SQL Server y MySQL [7].

En los cuatro sistemas, la recursión funciona a partir de la implementación de Common Tables Expressions (CTEs). En el caso de Oracle, las consultas recursivas están pensadas para trabajar con data jerárquica mediante la expresión CONNECT BY [8] y la utilización de una consulta base y una consulta recursiva que se construye según cómo se desea recorrer los datos a partir de la consulta base.

La recursión en PostgreSQL es soportada desde la versión 8.4 [9], mediante el keyword WITH RECURSIVE. La forma de escribir consultas recursivas en este motor de bases de datos sigue la misma estructura que la provista por Oracle, debiendo definir una consulta base y una consulta recursiva que iterativamente irá construyendo la tabla de trabajo a partir la relación que se establezca con la consulta base. Este motor de bases de datos provee el soporte para la operación UNION ALL entre la consulta base y la consulta recursiva, como una forma de protección contra los ciclos, como por ejemplo cuando este se produce como un resultado de filas duplicadas [10]. Finalmente, la implementación de recursión en SQL Server y en MySQL a partir de su versión 8, también están basadas en el uso de CTEs[11, 12], siguiendo la misma lógica de los otros dos motores ya mencionados.

La detección de ciclos es un paso fundamental para la implementación de consultas recursivas, ya que una consulta recursiva podría quedar ejecutándose de manera indefinida si es que esta no es capaz de detectar que sus nuevas iteraciones ya no están produciendo nuevos datos. En PostgreSQL, existe la keyword 'CYCLE' para simplificar la detección de ciclos, a la cual primero especifica la lista de columnas a las cuales hacer la detección de ciclos, otra columna para revisar si es que se ha detectado uno y finalmente una columna para llevar el camino recorrido por la recursión.

Respecto a la implementación de algoritmos recursivos en SQL para grafos, Ordoñez [13] propone una optimización de consultas recursivas lineales en SQL. Esta optimización se aplica para los algoritmos de Transitive Closure y Power Matrix of Adjacency matrix bajo distintas topologías, asumiendo una tabla en donde cada fila contiene la información de un vértice del grafo.

2.2. Propuesta de recursión generalizada

Teniendo en cuenta lo expuesto en la subsección anterior respecto a la recursión en bases de datos relacionales, haremos revisión sobre lo propuesto en la literatura para abordar las dificultades de la implementación de consultas recursivas en este tipo de sistemas.

En la tesis mencionada en la introducción [2], se estudia sobre la extensión del lenguaje de consulta de bases de datos de grafos G-CORE para que este pueda soportar la recursión. Dentro de este estudio, se hace una propuesta para un operador general recursivo para SQL.

Una consulta recursiva está compuesta de una consulta base q_0 y una consulta recursiva q_r. Este proceso sobre se aplica sobre un conjunto de datos (una tabla para el caso de SQL) D. En primer lugar, se obtiene el resultado de $q_0(D)$. Este resultado D_1 después es procesado por la consulta q_r para producir D_2 y así sucesivamente. Normalmente, esto ocurre cuando el tamaño de los resultados producidos (i.e los resultados parciales) $D_1, ..., D_n$ llegan a un Punto Fijo.

A partir de esto, se propone separar el orden de la recursión del conjunto de resultados parciales $D_1, ..., D_n$ mediante la siguiente modularización:

1. Una consulta base q_0

2. Una consulta recursiva q_r

3. Un orden parcial $(P, \leq)$ y una relación H desde los resultados parciales a A

En SQL el paso (3) no debe ser definido explícitamente dado que P serían los resultados parciales de (1) y (2) y el orden la inclusión $\subseteq$. Aquí la autora explica que si bien omitir este paso puede ser visto como una ayuda para un desarrollador, finalmente se generan dificultades para aquellos casos donde una consulta no debe seguir el orden $((D_1, ..., D_n), \subseteq)$

Para lograr esto, lo que se propone es incorporar la posibilidad de definir (3) independiente del orden $((D_1, ..., D_n), \subseteq)$, mediante la abstracción de este orden implícito hacia una forma explícita, con la definición de otro orden $(A, \leq)$ y una función no decreciente F sobre este orden parcial.

Bajo este caso, un desarrollador debe seguir definiendo q_0 y q_r pero con la salvedad de que en vez de preocuparse de llegar a un resultado monótono con los resultados parciales de la consulta recursiva, ahora se puede confiar en el orden $(A, \leq)$. Esto trae el costo de tener que codificar explícitamente el orden $(A, \leq)$ y también la relación H entre $(A, \leq)$ y $((D_1, ..., D_n), \subseteq)$.

El ejemplo mostrado en la sección 2.1 podría no llegar a un punto fijo si es que dentro del grafo existe un ciclo y se utilizara UNION ALL en vez de UNION, ya que la primera instrucción añade las nuevas filas generadas por la subconsulta recursiva independiente si es que estas ya se calcularon en una iteración anterior. Con estas dos condiciones la consulta recursiva iteraría indefinidamente dentro del ciclo al ir añadiendo en cada iteración nuevas filas a la tabla temporal.

En el caso de PosrgreSQL, tenemos el ejemplo de la consulta recursiva para obtener la suma de los primeros 100 números naturales del código 1.1. Aquí en la subconsulta recursiva se especifica mediante la instrucción WHERE $n < 100$ el orden sobre los resultados obtenidos en cada aplicación de la consulta recursiva. Con esto se obtiene que a partir de la iteración 101 se llegue a un punto fijo respecto al tamaño de la tabla temporal producida por la consulta, haciendo retornar la tabla obtenida. Si es que no estuviese definida esta instrucción dentro de la subconsulta recursiva, esta solo se detendría al momento de producir un overflow para el valor que puede tener un número entero en PostgreSQL.

2.3. Problemas de grafos polinomiales

En la siguiente sección se mostrará la definición formal de seis problemas conocidos de grafos que corren en tiempo polinomial junto con la descripción de algunos de los algoritmos más conocidos para resolverlos con esta restricción de tiempo. Estos problemas serán los que se utilizarán en las secciones siguientes para el desarrollo de esta memoria, los cuales fueron elegidos desde el capítulo 6 del libro Combinatorial Algorithms de Ludek Kucera sobre algoritmos de grafos polinomiales [14].

Topological Sorting

Def 1 (Topological Sorting) Sea G un grafo dirigido sin ciclos. Un orden topológico (Topological Sorting) de G es un orden lineal ($\leq$) sobre sus nodos tales que si el arco (u, v) pertenece al grafo, entonces el nodo $u \leq v$ en el orden.

Dentro de los algoritmos más conocidos para resolver este problema se encuentra el algoritmo de Kahn [15], el cual va iterativamente seleccionando los nodos sin arcos de entrada, removiéndolos del grafo y actualizando la lista de arcos entrantes para los nodos restantes. Este algoritmo continúa hasta que todos los nodos han sido visitados.

Por otro lado, también puede ser utilizado el algoritmo de Depth-First Search (DFS). Este algoritmo es modificado para visitar los vértices en un orden en específico, asegurándose que un nodo es añadido a este orden solo después de que todos sus nodos adyacentes hayan sido visitados [16].

Connected Components

Def 2 (Connected Components) Una componente conexa de un grafo G es un subgrafo conexo de G que no es parte de otra componente conexa más grande.

Dentro de los algoritmos conocidos para encontrar componentes conexas dentro de un grafo se consideran los algoritmos de Depth-First Search (DFS) o Breadth-First Search (BFS)[17] con los que es posible desde un nodo encontrar el resto de nodos que son alcanzables por este, identificando así las componentes conexas del grafo.

Eulerian Circuit

Def 3 (Eulerian Circuit) Un camino (o ciclo) euleriano de un grafo G es un camino que contiene todos los nodos y arcos de G sin la repetición de nodos.

Para el problema de Eulerian Circuit tenemos el algoritmo de Hierholzer[18] que se basa en recorer cada arco exactamente una vez asegurándose de que el circuito recorrido visite todos los arcos y nodos. Por otro lado, tenemos el algoritmo de Fleury [19] el cual tiene un acercamiento similar al algoritmo de Hierholzer, con la diferencia de que en este algortmo se seleccionan aquellos vértices que no desconectan el grafo.

Minimum Spanning Tree

Def 4 (Spanning Tree) Un Spanning Tree de un grafo G es un subgrafo de G que contiene todos sus nodos y es un árbol (i.e, no tiene ciclos)

Def 5 (Minimum Spanning Tree) Un Minimum Spanning Tree de un grafo G es un Spaning Tree sobre G cuyo peso es el mínimo posible.

Entre los algoritmos conocidos para resolver este problema tenemos en primer lugar el algoritmo de Kruskal [20], el cual es un algoritmo avaro que construye el Minimum Spanning Tree mediante añadir arcos al árbol en orden creciente según sus pesos siempre y cuando esta adición no cree un ciclo en el árbol. En segundo lugar, tenemos el algoritmo de Prim [21], que también es un algoritmo avaro, que comienza con un nodo arbitrario y va armando el minimum spanning tree añadiendo un nodo a la vez bajo el criterio de seleccionar arco tenga el peso mínimo que conecte este nodo perteneciente al árbol con uno nuevo.

Maximum Matching

Def 6 (Matching) Dado un grafo $G = (V, E)$, un matching M en G es un conjunto de arcos no adyacentes entre sí.

Def 7 (Maximum Matching) Un Maximum Matching es un Matching que contiene la mayor cantidad de arcos.

En el caso de Maximum Matching, para un grafo arbitrario, tenemos el algoritmo de Edmonds (Blossom Algorithm)[22], el cual opera buscando de manera iterativa caminos de aumento, consistentes en un camino entre nodos sin matching donde el número de caminos es par. Cuando uno de estos caminos es encontrado el algoritmo contrae este camino a un único vértice, permitiendo simplificar el grafo sin perder potenciales matches.

Para el caso de grafos bipartitos existen algoritmos más eficientes como el algoritmo de Hopcroft-Karp [17] o el algoritmo de Ford-Fulkerson [23].

Planarity Testing

Def 8 (Planarity Testing) Un grafo G es planar si es que este puede ser dibujado en el plano sin que ningún arco se cruce.

Dentro de los criterios conocidos para determinar si es que un grafo G es planar se encuentra el teorema de Kuratowski, el cual indica que un grafo es planar si es que este no contiene un subgrafo que sea una subdivisión de K_5 (el grafo completo de 5 nodos) o de $K_{3,3}$ (grafo bipartito completo de 6 nodos).

Capítulo 3

El problema y la solución

En este capítulo se detallará cómo se pretende resolver el problema de la implementación de una codificación genérica para consultas recursivas en SQL para problemas de grafos que corren en tiempo polinomial. El capítulo se dividirá en dos secciones: La primera definirá los objetivos que plantea esta memoria. La segunda sección tomará los objetivos propuestos y mostrará cuál es la solución propuesta para estos.

3.1. Objetivos

En esta sección se presentan los objetivos que tiene esta memoria. Primeramente, se planteará el objetivo general de este trabajo, para seguir con un desglose de este objetivo general en objetivos específicos, los cuales guiarán el proceso de implementación de la codificación genérica para consultas recursivas en SQL para problemas de grafos que corren en tiempo polinomial.

3.1.1. Objetivo General

El objetivo de esta memoria consiste en estudiar el comportamiento de la recursión en el contexto de lenguajes para bases de datos relacionales SQL. Se implementará un modelo más genérico que permite codificar de manera más simple algoritmos de grafos clásicos cuya complejidad computacional es de tiempo polinomial.

En otras palabras, el objetivo de esta memoria consiste en estudiar la factibilidad de implementar la propuesta de recursión generalizada mostrada en el capítulo 2.2 para la simplificación de consultas recursivas para el caso de un motor de bases de datos relacional SQL mediante separar el orden de la recursión.

3.1.2. Objetivos Específicos

1. Entender el comportamiento de la recursión en un sistema de bases de datos relacional SQL.
2. Diseñar y desarrollar una interfaz que permita la realización de consultas recursivas mediante una codificación genérica para SQL.
3. A partir de la codificación genérica del ítem anterior, implementar la solución para los siguientes problemas

a) Topological Sorting
b) Connected Components
c) Eulerian Circuit
d) Minimum Spanning Tree
e) Maximum Matching
f) Planarity Testing

4. Probar las codificaciones sobre fuentes de datos que representen grafos como tablas SQL.

3.2. Solución propuesta

Para poder resolver los objetivos antes propuestos se propone la siguiente solución:

El primer paso consiste en obtener los conjuntos de datos necesarios para posteriormente implementar las consultas recursivas a estudiar. Estos conjuntos de datos deben representar grafos de diferentes topologías. Para esto, se hará uso de subconjuntos de algunos de los datasets propuestos por el Linked Data Benchmark Council (LBDC) [24] los cuales proveen archivos que representan los nodos y vértices de una topología. Los datasets provistos cuentan con topologías entre aproximadamente 630.000 hasta 555.000.000 nodos. Como en este trabajo no se hará una revisión de la performance de las consultas a realizar con distintos tamaños de topologías, se implementará un script utilizando el lenguaje de programación Python en su versión 3.10.6 [25].

Este script a implementar recibirá como entrada la cantidad de nodos que se quieren para una topología de prueba. A partir de esto, leerá los archivos de nodos y vértices del dataset para determinar de manera aleatoria cuál de los nodos de la topología completa se utilizará para la salida de este script. Esta salida debe ser con algún formato que permita la carga del grafo resultante como una tabla de SQL. Para la carga de estos datos y las posteriores consultas SQL, se hará el uso del motor PostgreSQL en su versión 14.4.

En segundo lugar, se implementará un backend utilizando la misma versión utilizada para el script de generación del grafo, en conjunto al framework Django [26]. Esta backend servirá de puente para comunicar la base de datos PostgreSQL junto a un frontend que se implementará utilizando Javascript y el framework React.js [27]. Este frontend contendrá la interfaz desde la cual un usuario podrá ingresar de manera más sencilla una consulta recursiva para SQL mediante una caja de texto donde un usuario pueda colocar su consulta a conveniencia.

En la Figura 3.1 se adjunta un diagrama mostrando la arquitectura propuesta como solución para este problema.

En tercer lugar, con la interfaz conectada al backend y este a su vez conectado con la base de datos, se procederá a implementar las distintas consultas recurisvas para los siguientes problemas de grafos de tiempo polinomial:

1. Topological Sorting
2. Connected Components

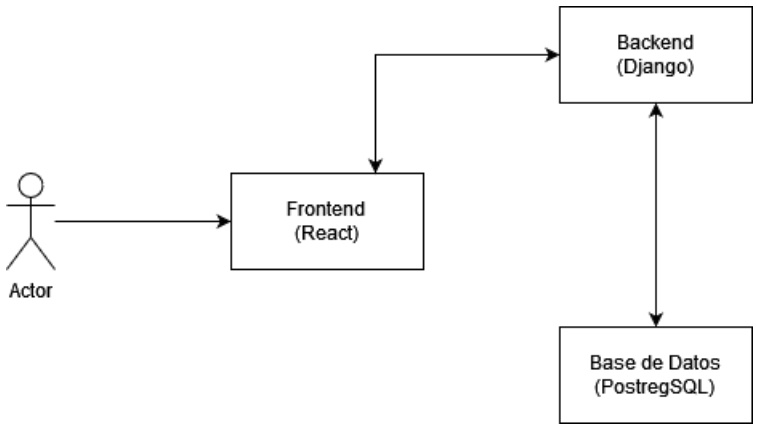

Figura 3.1: Arquitectura propuesta para la solución

3. Eulerian Circuit
4. Minimum Spanning Tree
5. Maximum Matching
6. Planarity Testing

La codificación a implementar para cada uno de estos problemas utilizará la siguiente consulta SQL como base:

Código 3.1: Consulta recursiva base para SQL

```
1  WITH RECURSIVE Q(n) AS (
2      Q0
3    UNION ALL
4      Qr
5  )
6  SELECT Agg;
```

Para esta consulta consideraremos las siguientes componentes:

- $Q(n)$: Q Representa la función recursiva y n a los valores que tendrá la tabla de salida de la consulta.
- Q_0: Corresponde al caso base de la consulta recursiva
- Q_r: Corresponde a la parte recursiva de la consulta.
- Agg: Finalmente, aquí se codifica consulta que se hará sobre el resultado de la tabla resultante de la consulta recursiva.

Para el cuarto paso, teniendo estas consultas ya definidas mediante el uso de la consulta base antes mencionada, se procederá a implementar una codificación genérica que permita resolver estos problemas. Esta codificación será ingresada desde la interfaz antes desarrollada y debe permitir al usuario ingresar $Q(n)$, Q_0, Q_r y Agg y entregar el resultado de la consulta.

Se asumirá que el usuario conoce de antemano las tablas que se utilizarán para satisfacer las consultas posteriores. En caso de que el usuario codifique una consulta para la cual no existen datos o bien la codificación presente algún problema de sintaxis, la interfaz deberá ser capaz de responder al usuario con un mensaje de error adecuado a la entrada provista.

Finalmente, se hará una evaluación de la codificación genérica que permite la resolución de los problemas antes propuestos. Esta evaluación consistirá en determinar qué tan general puede hacerse esta codificación para todos los problemas provistos.

3.2.1. Selección de datos de prueba y sistema de carga de datos

Como se menciona en la sección de solución propuesta, se utilizarán datasets provistos por el Linked Data Benchmark Council (LDBC). Los datasets provistos por LDBC corresponden a archivos comprimidos de la forma *<nombre-dataset>.tar.zst* los cuales al ser descomprimidos entregan una serie de archivos correspondientes al dataset, en donde nos fijaremos en particular en el archivo *<nombre-dataset>.v*, que corresponde a la lista de aristas del grafo que tiene el dataset.

Cada entrada en el archivo de aristas se compone de 3 valores: nodo de entrada, nodo de salida y peso de la arista. De estos valores solo utilizaremos los de nodo de entrada y nodo de salida, ya que no consideraremos aquellos casos en que el grafo tiene pesos en sus aristas.

Se implementó un script de Python que toma este archivo de vértices e inserta los valores mencionados en el párrafo anterior a una tabla dentro de una base de datos PostgreSQL. La inserción de los valores leídos se realiza en bloques para agilizar este proceso, ya que al hacerlo de manera individual disminuye la performance del script.

Como la cantidad de aristas de los datasets provistos LDBC son bastante grandes (incluso aquellos más pequeños), se realizó también la implementación de otro script que, a partir de una cantidad de nodos dada, genera un grafo aleatorio. Este script genera un número de nodos igual al de la entrada, los recorre uno por uno y de manera aleatoria se crea un vértice entre ellos.

Capítulo 4

Implementación de la solución propuesta

En este capítulo se mostrará como se realizó la implementación de la solución propuesta en el capítulo anterior. Para esto, esta sección se dividirá en dos subsecciones: En primer lugar, se mostrará la implementación de la generación de los datos para las consultas y la interfaz gráfica para que un usuario pueda implementar sus consultas mediante la codificación genérica propuesta. En segundo lugar, se mostrará la implementación de cada una de las consultas SQL recursivas para los 6 problemas propuestos.

4.1. Implementación de la generación de datos e interfaz

Para que un usuario tenga la posibilidad de crear un grafo como tabla en una base de datos relacional, se ha implementado la siguiente interfaz web mediante el uso del framework de Javascript, React.js. Esta interfaz web está conectada a un backend que utiliza el framework Django en conjunto a la librería *django-rest-api*.

Este backend implementa los scripts mencionados en la subsección anterior para la selección y carga de un grafo en la base de datos. En particular, se dispone de dos endpoints para la comunicación entre la interfaz web:

- /api/parser/create_graph
- /api/parser/execute_query

Estos endpoints son utilizados respectivamente para la creación de un grafo como tabla dentro de la base de datos y para la ejecución de consultas sobre la base de datos. Las vistas asociadas a estos endpoints establecen la conexión con la base de datos PostgreSQL para la ejecución de ambas instrucciones.

Se espera que cuando esté definida la consulta genérica esta pueda ser ingresada en esta interfaz y muestre los resultados de esta al igual que como se realiza con la respectiva consulta SQL y que además se pueda visualizar la consulta SQL resultante de la digitación de esta mediante la consulta genérica implementada.

El código fuente de la implementación de la interfaz (tanto del frontend como del backend) puede encontrarse en Github [28].

4.2. Implementación de consulta genérica

Teniendo como base la consulta recursiva base para SQL propuesta en el código 3.1 y la implementación de las consultas SQL específicas de los problemas mencionados utilizando esta base, se observa lo siguiente:

1. En $Q(n)$ y Agg es necesario escribir el nombre de la función recursiva que tendrá la consulta recursiva a realizar sobre la tabla que representa el grafo.

2. Dependiendo el tipo de problema, hay que variar entre ejecutar la orden de UNION o de UNION ALL entre la Q_0 y Q_r. La elección radicará en la necesidad de no mantener filas repetidas en la tabla temporal de trabajo (UNION) o bien tener que repetir filas en cada iteración (UNION ALL).

3. Asumiendo la estructura antes mencionada para la tabla (representación de las aristas mediante dos columnas) a la que se realizarán las consultas, tanto en Q_0 y Q_r hay que hacer mención a la tabla y a sus dos columnas.

4. Existe una extensión que hacer a las consultas cuando en Q_0 o Q_r se necesita especificar un nodo inicial o una condición específica con la cual hacer una comparación en la sub consulta recursiva.

5. También es necesario definir cómo se hará la consulta Agg sobre los resultados de la consulta recursiva, de manera similar a lo que se especifica en el ítem 3 para Q_0 o Q_r.

A partir de estas observaciones, se establece de manera preliminar la forma que de la consulta generalizada:

$$f(T_n, ALL, A_0, A_r, A_{Agg})$$

En donde:

- f corresponde al nombre de la función
- T_n corresponde al nombre de las columnas que tendrá la tabla resultante de la consulta recursiva.
- El conjunto T_n es utilizado en Q_0 y Q_r
- ALL es un indicador booleano para definir si entre Q_0 y Q_r debe realizarse la operación de UNION o UNION ALL.
- A_0 es la especificación de Q_0 para el filtro que esta pueda tener y la variable inicial con la que comenzará la recursión.
- A_r es la especificación de Q_r para el filtro que esta pueda tener y la operación que se hará en cada recursión. Es en esta especificación donde un programador debe indicar cuál será el orden que seguirá la consulta recursiva para llegar a un Punto Fijo.

- A_{Agg} es el detalle de la relación de orden Agg que se espera para la salida de la consulta recursiva.

Esta consulta generalizada podría ser reducida si es que no se desea especificar el nombre de las columnas (i.e. T_n) de la tabla resultante y dejar en manos de la interfaz el obtenerlos e insertarlos en la consulta final, dejándola de la siguiente manera:

$$f(T, ALL, A_0, A_r, A_{Agg})$$

De esta propuesta preliminar para la consulta generalizada, se puede asumir que los valores de f, T (T_n en la versión larga) y ALL pueden ser entregados como una cadena de texto indicando los nombres respectivos para cada uno de estos valores.

Para ejemplificar, la consulta del código 1.1 para la suma de los primeros 100 números naturales podría escribirse de la siguiente manera: t(n, ALL, VALUES (1), SELECT n+1 FROM t WHERE n < 100, SELECT sum(n) FROM t).

4.3. Implementación de consultas SQL recursivas

A continuación se muestran las implementaciones de consultas SQL recursivas para los problemas propuestos. Estas consultas fueron implementadas siguiendo la estructura base para las consultas recursivas en SQL definida en el código 3.1. Para las consultas se asume que la tabla a la cual se realizarán estas consultas se llama *edges* y esta contiene la lista de aristas del grafo representadas con las columnas *source_vertex* y *target_vertex* que corresponden al vértice de inicio y al vértice de salida. Además, se considera la columna extra *weight* para indicar el peso del arco entre los nodos del grafo para aquellos problemas que lo requieren. La implementación de estas consultas también puede ser encontrada en [28].

Para cada uno de los problemas implementados se mostrará el código de las consultas SQL y la explicación de la consulta base (A_0), la consulta recursiva (A_r) y la consulta final sobre la CTE (A_{Agg}), finalizando con un resumen de la consulta completa y su salida.

Topological Sorting

La implementación de la consulta para Topological Sorting puede encontrarse en el Código 4.1. Para su la implementación de esta consulta se realizó una variante del algoritmo DFS para grafos. El código se divide de la siguiente manera:

- En la sub-consulta base, definida entre las líneas 2 y 3, se eligen aquellos nodos que no tienen aristas de entrada y se fija su nivel en 0.
- En la consulta recursiva definida entre las líneas 5 y 7 se hace la selección de aquellas aristas donde el nodo de entrada es equivalente al nodo de salida para las filas seleccionadas anteriormente y se les suma 1 a su nivel.
- Finalmente, en la consulta de agregación, entre las líneas 9 y 11, se seleccionan los distintos nodos de entrada y su nivel máximo, ordenándolos por este nivel de manera decreciente.

Código 4.1: Consulta SQL propuesta para el problema de Topological Sorting

```
1  WITH RECURSIVE TopologicalOrder(vertex, depth) AS (
2    SELECT edges.source_vertex, 0 FROM edges
3     WHERE source_vertex NOT IN (SELECT target_vertex FROM edges)
4    UNION
5     SELECT edges.target_vertex AS id, TopologicalOrder.depth + 1 FROM edges
6     JOIN TopologicalOrder
7     ON edges.source_vertex = TopologicalOrder.vertex
8    )
9    SELECT vertex, depth FROM TopologicalOrder
10   GROUP BY vertex, depth
11   ORDER BY MAX(depth);
12
```

A modo de resumen, la consulta comienza con vértices sin nodos de entrada y recursivamente añade vértices conectados a los seleccionados anteriormente hasta que todos los vértices están cubiertos. El resultado final entrega el orden topológico basado en el nivel máximo de cada vértice.

Connected Components

La implementación de la consulta para Connected Components puede encontrarse en el Código 4.2. Para la implementación de esta consulta también se utilizó una variante del algoritmo de DFS para encontrar recursivamente las componentes conexas del grafo. Este código se divide de la siguiente manera:

- En la sub-consulta base, entre las líneas 2 y 3, se seleccionan todas las aristas, considerando cada una de ellas como una componente separada y asigna el vértice de entrada como el identificador de la componente.
- En la consulta recursiva, definida entre las líneas 7 y 9, se conectan los nodos que comparten la misma componente conexa, basándose en si el nodo de entrada o de salida de la componente se encuentran en ella.
- Finalmente, en la consulta de agregación, definida entre las líneas 11 y 14, se hace la selección de las distintas componentes conexas obtenidas por la consulta recursiva y agrega los vértices de cada componente en un arreglo.

Código 4.2: Consulta SQL propuesta para el problema de Connected Components

```
1  WITH RECURSIVE ConnectedComponents(source_vertex, target_vertex,
   ↪ component) AS (
2    SELECT source_vertex, target_vertex, source_vertex AS component
3    FROM edges
4
5    UNION
6
7    SELECT e.source_vertex, e.target_vertex, cc.component
```

```
8           FROM edges e
9           JOIN ConnectedComponents cc ON e.source_vertex = cc.target_vertex OR e.
    ↪ target_vertex = cc.source_vertex
10        )
11        SELECT DISTINCT component, array_agg(DISTINCT source_vertex) AS nodes
12        FROM ConnectedComponents
13        GROUP BY component
14        ORDER BY component;
15
```

En términos generales, la consulta comienza con cada vértice como su propia componente y recursivamente conecta a aquellos vértices que comparten una arista hasta identificar las componentes conexas en el grafo. El resultado final entrega las componentes conexas junto con los vértices que componen cada componente.

Eulerian Circuit

La implementación de la consulta para Eulerian Circuit puede encontrarse en el Código 4.3. La implementación se basa en una aplicación del algoritmo de DFS para recorrer todo el grafo y encontrar un circuito euleriano. Este código se divide de la siguiente manera:

- En la sub-consulta base, definida entre las líneas 2 y 8, se selecciona el nodo de valor mínimo como punto de partida. Se inicializa la tabla de trabajo con el vértice inicial, el vértice actual, un arreglo de los vértices visitados y una profundidad de 1.
- En la consulta recursiva, definida entre las líneas 10 y 20, se selecciona el siguiente vértice dentro del circuito euleriano mediante el JOIN entre el vértice actual con la tabla de vértices. Actualiza el vértice actual, añade el nuevo vértice al arreglo de vértices visitados y aumenta la profundidad en 1. La recursión se detiene con la condición impuesta en la línea 19 en que la profundidad del camino no puede ser mayor a la cantidad del número de aristas del grafo.
- Finalmente, en la consulta de agregación, entre las líneas 22 y 27, se seleccionan los vértices visitados por la consulta y se ordenan por la profundidad de la recursión alcanzada.

Código 4.3: Consulta SQL propuesta para el problema de Eulerian Circuit

```
1     WITH RECURSIVE EulerianCircuit(start_vertex, current_vertex, visited_vertices,
    ↪ depth) AS (
2       SELECT
3         start_vertex,
4         start_vertex AS current_vertex,
5         ARRAY[start_vertex] AS visited_vertices,
6         1 AS depth
7       FROM
8        (SELECT MIN(source_vertex) as start_vertex FROM edges) AS initial
9       UNION
10      SELECT
11        ec.start_vertex,
12        edges.target_vertex AS current_vertex,
```

```
13            ec.visited_vertices || edges.target_vertex,
14            ec.depth + 1
15          FROM
16            EulerianCircuit ec
17            JOIN edges ON ec.current_vertex = edges.source_vertex
18          WHERE
19            ec.depth < (SELECT COUNT(*) from edges)
20            AND edges.target_vertex <> ALL(ec.visited_vertices)
21        )
22        SELECT
23          *
24        FROM
25          EulerianCircuit
26        WHERE depth < (SELECT COUNT(*) from edges)
27        AND current_vertex = start_vertex;
28
```

Recapitulando, la consulta comienza con el vértice mínimo e iterativamente añade vértices al circuito hasta que todos los nodos son visitados. El resultado final muestra el progreso del circuito con el vértice inicial, el vértice actual, el arreglo de los vértices visitados y el número de iteraciones realizadas por la recursión.

Minimum Spanning Tree

La implementación de la consulta para Minimum Spaning Tree puede encontrarse en el Código 4.4. Esta consulta también basa su implementación en realizar una variante del algoritmo de DFS, similar a lo propuesto por el algoritmo de Prim. El código se divide de la siguiente manera:

- En la sub-consulta base, definida entre las líneas 3 y 11, se hace la selección de se fija el nivel base 1 para el nodo inicial que es fijado en la línea 11.
- En la consulta recursiva, definida entre las líneas 16 y 26, se selecciona recursivamente con el estado actual del MST. En cada ciclo se van añadiendo los nodos con el camino de peso mínimo al árbol existente y se previenen los ciclos al revisar que la profundidad de la recursión sea menor a la cantidad de nodos que tiene el grafo menos uno, puesto qué esta es la profundidad máxima que puede tener un nodo desde la raíz del árbol. Este chequeo se hace en la línea 26 fijando que el nivel del MST sea menor a la cantidad de vértices menos 1.
- Finalmente, en la consulta de agregación, definida entre las líneas 29 y 36, se seleccionan los pares de nodos resultantes del MST junto con su nivel.

Código 4.4: Consulta SQL propuesta para el problema de Minimum Spaning Tree

```
1         WITH RECURSIVE MST AS (
2           SELECT
3               e.source_vertex AS source_vertex,
4               e.target_vertex AS target_vertex,
```

```
5               e.weight,
6               1 AS level,
7               ARRAY[e.source_vertex, e.target_vertex] AS connected_vertices
8           FROM
9               edges e
10          WHERE
11              e.source_vertex = (SELECT MIN(source_vertex) FROM edges)
12
13          UNION
14
15          SELECT
16              e.source_vertex AS source_vertex,
17              e.target_vertex AS target_vertex,
18              e.weight,
19              MST.level + 1,
20              MST.connected_vertices || e.target_vertex
21          FROM
22              edges e
23          JOIN
24              MST ON (e.source_vertex = MST.target_vertex OR e.target_vertex = MST.
    ↪ target_vertex)
25          WHERE
26              MST.level < (SELECT COUNT(DISTINCT source_vertex) FROM edges) - 1
27              AND NOT (e.source_vertex = MST.source_vertex AND e.target_vertex =
    ↪ MST.target_vertex)
28              AND NOT (e.target_vertex = ANY(MST.connected_vertices))
29              AND e.weight = (
30                    SELECT MIN(weight)
31                    FROM edges
32                    WHERE (source_vertex = e.source_vertex AND target_vertex = e.
    ↪ target_vertex)
33                    OR (source_vertex = e.target_vertex AND target_vertex = e.
    ↪ source_vertex)
34                )
35
36      SELECT
37          source_vertex,
38          target_vertex,
39          MIN(level) as level,
40          MIN(weight) as weight
41      FROM
42          MST
43      WHERE
44          NOT EXISTS (
45              SELECT
46                  (SELECT MIN(source_vertex) FROM edges)
47              FROM
48                  MST AS M
49              WHERE
50                  MST.source_vertex = M.target_vertex
51                  AND MST.target_vertex = M.source_vertex
52          )
```

```
53        GROUP BY
54          source_vertex, target_vertex
55        ORDER BY
56            source_vertex, target_vertex;
57
```

Cabe destacar que para esta consulta en particular se hace uso de la columna *weight* para obtener el valor del peso de cada arista. En resumen, la consulta comienza con un nodo inicial dado y recursivamente añade nodos al árbol, asegurándose de que el grafo resultante se mantenga acíclico. Finalmente, se obtiene como resultado el conjunto de las aristas que forman el MST.

Maximum Matching

La implementación de la consulta para Maximum Matching puede encontrarse en el Código 4.5. Esta implementación es una implementación avara para encontrar un matching dentro de un grafo. :

- En la sub-consulta base, entre las líneas 3 y 10, se fija un nodo inicial en la línea 10 y se le asigna un nivel inicial de 1.
- En la consulta recursiva, entre las líneas 15 y 23, se hace un JOIN recursivo con el resultado previo mediante la selección de aquellos nodos que cumplan la condición de que el nodo de entrada del nodo actual haga match con el vértice de salida del nodo anterior y aumenta su nivel en 1. La condición impuesta en la línea 23 se asegura de que solo los vértices sin match sean seleccionados para así poder encontrar los matches restantes.
- Finalmente, en la consulta de agregación, entre las líneas 26 a 32, se seleccionan aquellos nodos donde el nivel sea par para obtener aquellos nodos que tienen un match.

Código 4.5: Consulta SQL propuesta para el problema de Maximum Matching

```
1         WITH RECURSIVE matching AS (
2          SELECT e.target_vertex AS node, e.source_vertex AS matched
3          FROM edges e
4          WHERE NOT EXISTS (
5           SELECT 1
6           FROM edges e2
7           WHERE e.target_vertex = e2.source_vertex
8          )
9          UNION ALL
10         SELECT m.node, e.source_vertex AS matched
11         FROM matching m
12         JOIN edges e ON m.matched = e.target_vertex
13         WHERE NOT EXISTS (
14          SELECT 1
15          FROM edges e2
16          WHERE e.source_vertex = e2.source_vertex
```

```
17            AND e.source_vertex <> e2.target_vertex
18          )
19        )
20        SELECT node, matched
21        FROM matching
22        WHERE matched IS NOT NULL;
23
```

Cabe mencionar que no fue posible implementar otro tipo de algoritmos mediante la propuesta de recursión generalizada, puesto que este tipo de consultas no sirven para implementar algoritmos óptimos conocidos (como lo serían el algoritmo de Edmonds o Ford-Fullkerson para este problema) y se requiere el empleo de otro tipo de mecanismos provistos por PostgreSQL, como lo son librerías para generar funciones mediante un lenguaje procedural.

A modo de resumen, la consulta recursiva recorre el grafo desde un nodo dado y selecciona aquellos nodos que forman un maximum matching. Esta selección se realiza mediante el filtrado de aquellos niveles impares (i.e. nodos sin match).

Planarity Testing

Para la implementación de la consulta para resolver el problema de Planarity testing se utilizará el Teorema de Kuratowski. Para esto, en el Código 4.6, se muestra una consulta recursiva que busca la existencia del sub grafo K_5 utilizando una tabla de vértices como entrada. Este código se divide de la siguiente manera:

- En la sub-consulta base, definida entre las líneas 3 y 13, se seleccionan aquellos grupos de 5 nodos que están conectados entre sí.
- En la consulta recursiva, definida entre las líneas 18 y 21, se aplica el JOIN para comprobar que los nodos identificados en la consulta base estén completamente conectados.
- Finalmente, en la consulta de agregación en la línea 24 y 25, se entrega en cada fila el conjunto de 5 nodos que conforman un sub grafo K_5 dentro de la tabla de vértices.

Código 4.6: Consulta SQL propuesta para el problema de Planarity Testing

```
1     WITH RECURSIVE k5_check AS (
2       -- Anchor member
3       SELECT e1.target_vertex AS node1, e2.target_vertex AS node2, e3.target_vertex
    ↪  AS node3,
4              e4.target_vertex AS node4, e5.target_vertex AS node5
5       FROM edges e1
6       JOIN edges e2 ON e1.source_vertex = e2.source_vertex AND e1.target_vertex <
    ↪ e2.target_vertex
7       JOIN edges e3 ON e2.source_vertex = e3.source_vertex AND e2.target_vertex <
    ↪ e3.target_vertex
8       JOIN edges e4 ON e3.source_vertex = e4.source_vertex AND e3.target_vertex <
    ↪ e4.target_vertex
9       JOIN edges e5 ON e4.source_vertex = e5.source_vertex AND e4.target_vertex <
    ↪ e5.target_vertex
```

```
10          WHERE NOT (e1.target_vertex = e2.target_vertex OR e1.target_vertex = e3.
    ↪ target_vertex OR e1.target_vertex = e4.target_vertex OR e1.target_vertex = e5.
    ↪ target_vertex
11                  OR e2.target_vertex = e3.target_vertex OR e2.target_vertex = e4.
    ↪ target_vertex OR e2.target_vertex = e5.target_vertex
12                  OR e3.target_vertex = e4.target_vertex OR e3.target_vertex = e5.
    ↪ target_vertex
13                  OR e4.target_vertex = e5.target_vertex)
14
15          UNION ALL
16
17          -- Recursive member
18          SELECT e.target_vertex AS node1, k.node2 AS node2, k.node3 AS node3, k.
    ↪ node4 AS node4, k.node5 AS node5
19          FROM k5_check k
20          JOIN edges e ON (e.source_vertex = k.node1 AND e.target_vertex > k.node5)
21          WHERE NOT (e.target_vertex = k.node2 OR e.target_vertex = k.node3 OR e.
    ↪ target_vertex = k.node4 OR e.target_vertex = k.node5)
22        )
23        -- Final Select
24        SELECT DISTINCT node1, node2, node3, node4, node5
25        FROM k5_check;
26
```

En definitiva, la consulta explora recursivamente el recorrido entre los 5 nodos encontrados por la consulta base para determinar si es que estos 5 nodos están completamente conectados, añadiéndolo a la tabla de trabajo.

Capítulo 5

Evaluación

La evaluación de la solución propuesta se dividirá en dos partes: La primera corresponde a la evaluación de la interfaz creada para la ejecución de consultas recursivas en PostgreSQL, donde se hará la revisión sobre la interfaz web implementada, la carga de los datos y factibilidad de implementar de forma genérica la implementación de consultas recursivas sobre el motor de bases de datos relacional PostgreSQL.

En segundo lugar, se hará la evaluación de las ejecuciones de las consultas recursivas que se codificaron para los problemas propuestos para probar la interfaz, donde se evaluará la factibilidad de la implementación de la propuesta de recursión generalizada para la simplificación de consultas recursivas para el caso de un motor de bases de datos SQL mediante separar el orden de la recursión.

5.1. Evaluación de la carga de datos e interfaz

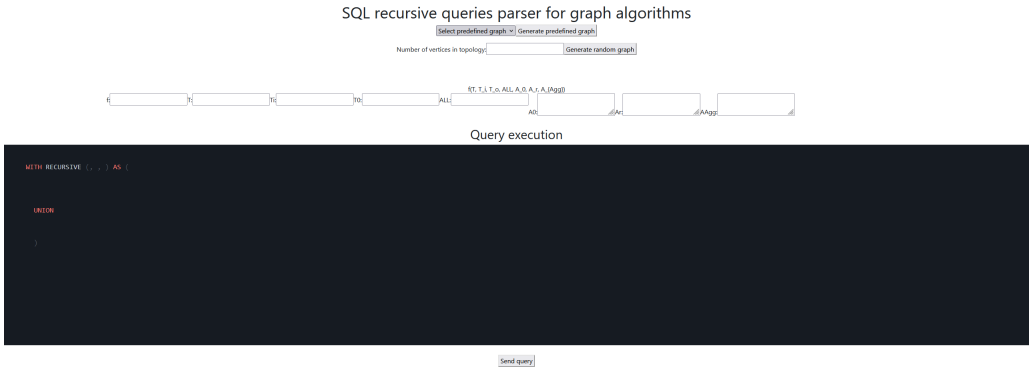

Figura 5.1: Interfaz inicial para un usuario

Actualmente, en esta interfaz es posible crear una tabla y realizar consultas sobre algoritmos recursivos para grafos sobre esta tabla creada.

La creación de la tabla se puede realizar de dos formas, las cuales utilizan el endpoint /api/parser/create_graph:

1. A partir del selector superior y el botón 'Generate predefined graph'.

2. A partir de un cuadro de input donde se recibe el número de vértices deseado para una topología y al presionar el botón 'Generate random graph' se generarán de manera aleatoria aristas para cada uno de los vértices ingresados.

Al momento de realizar la implementación y hacer pruebas con las consultas implementadas, se vio que el desempeño de estas es bastante difícil de medir con los datasets provistos por LDBC, puesto que el dataset con el menor número de aristas para un grafo es del orden de 5 millones, lo cual, considerando la complejidad computacional de los algoritmos implementados, no son posibles de ejecutar en un tiempo razonable para esa cantidad de datos. Es por esto que al generar un grafo mediante el selector 'Generate predefined graph' se generará una tabla a partir desde alguno de los datasets provistos en [28] dentro la carpeta */back/memoria_back/data* en la cual se añaden datasets con casos ideales para las consultas propuestas y también la opción de generar grafos aleatorios con un número mucho más reducido de vértices y aristas para que las consultas realizadas puedan terminar en un corto tiempo.

Cuando la tabla es generada, esta se mostrará como una tabla en la interfaz web para que el usuario pueda ver la topología final del grafo. Cabe mencionar que solo se mostrarán las primeras 100 filas de la tabla para no tener una mala visualización al momento de tener una topología con muchos vértices. Un ejemplo de esto puede verse en la Figura 5.2.

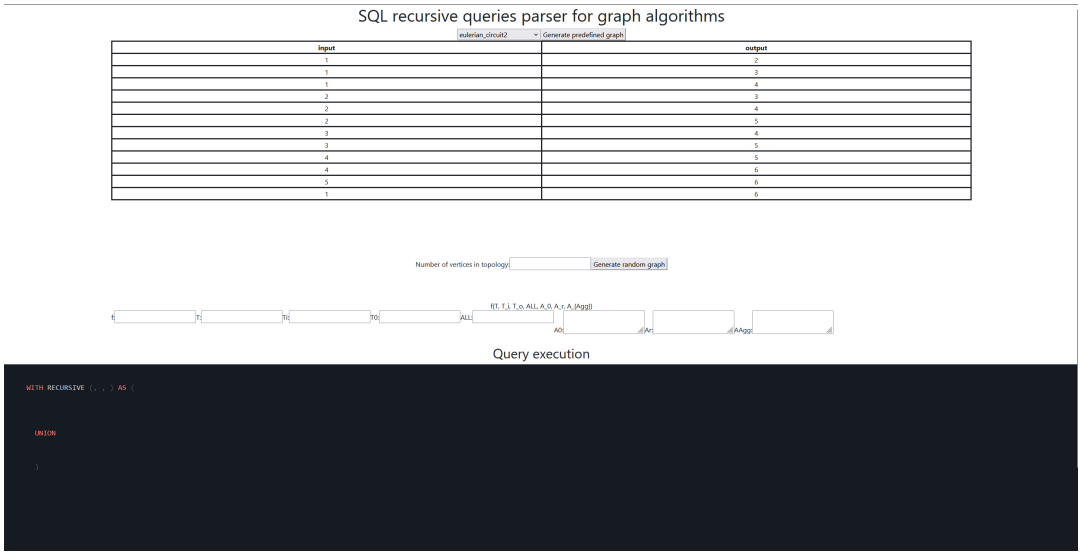

Figura 5.2: Resultado de la creación aleatorio de un grafo con 15 nodos

Por otro lado, se pueden hacer consultas a la tabla creada mediante un cuadro de texto ubicado al inferior de los botones para su creación.

Este cuadro de texto tiene la capacidad de resaltar la sintaxis SQL para así facilitar la escritura de las consultas que se deseen realizar a la tabla antes creada.

Después de escrita la consulta, se puede presionar el botón 'Send query' para que la consulta sea enviada al backend para que este la ejecute en la base de datos mediante el uso del endpoint /api/parser/execute_query. Un ejemplo de esto puede verse en la Figura 5.3 donde se realiza la consulta SQL para el problema de Topological Sort y su resultado es desplegado debajo del botón de envío de la consulta.

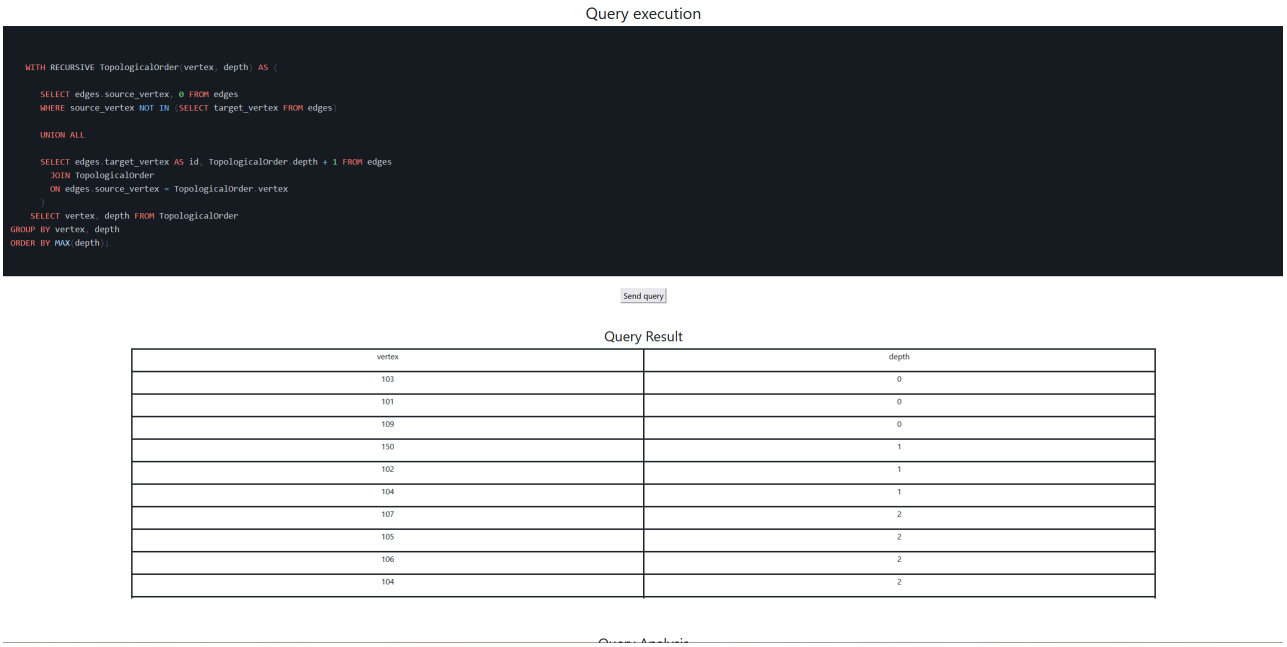

Figura 5.3: Captura de la interfaz con la ejecución de la consulta SQL recursiva para el problema de Topological Sort y su resultado, utilizando el grafo generado en la Figura 5.2

La lógica detrás de la interfaz implementada puede ser exportada a cualquier otro lenguaje o framework. En el código 5.1 se incluye un ejemplo en el lenguaje de programación Python donde la función *parse_recursive_query* recibe como parámetros necesarios para la consulta recursiva y posteriormente los inserta dentro de una sentencia SQL en la variable *query* para finalizar realizando la consulta hacia la base de datos con el método *execute_query*(). En el código 5.2 se muestra como se realizaría la llamada para el código de Python propuesto para el problema de Connected Components.

Código 5.1: Ejemplo de función de Python para la consulta generalizada

```
def parse_recursive_query(f, t_n, all, a_0, a_r, a_agg):
    query = f"WITH RECURSIVE {f}({t_n}) AS (
              {a_0}
            UNION {all}
              {a_r}
          )
          {a_agg};"
    execute_query(query)

```

Código 5.2: Ejemplo de llamado a la función de consulta generalizada propuesta en Python para el problema de Connected Components

```
parse_recursive_query(
    "ConnectedComponents",
```

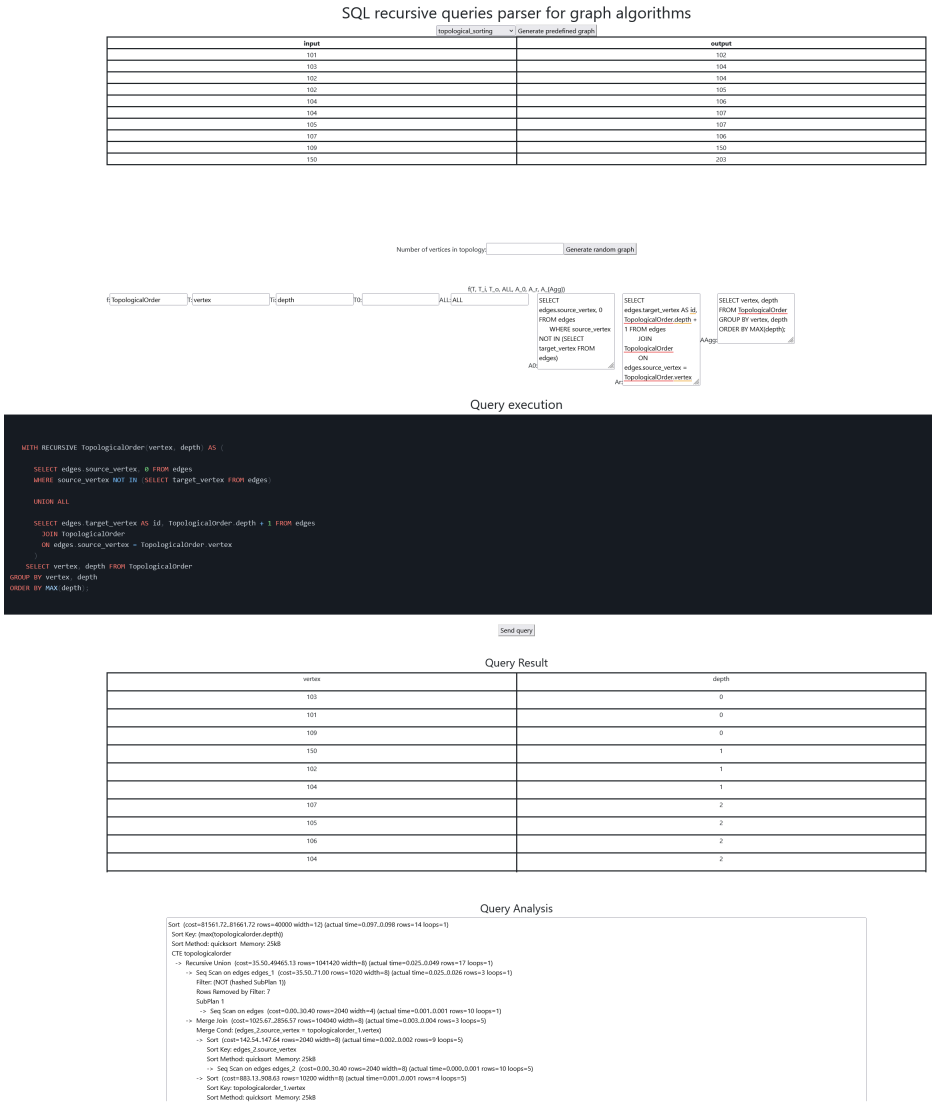

Figura 5.4: Muestra de la interfaz al realizar la consulta para el problema de Topological Sorting

```
        "source_vertex, target_vertex, component",
        "SELECT source_vertex, target_vertex, source_vertex AS component FROM edges",
        "",
        "SELECT e.source_vertex, e.target_vertex, cc.component
        FROM edges e
        JOIN ConnectedComponents cc ON e.source_vertex = cc.target_vertex OR e.target_vertex = cc.source_vertex",
        "SELECT DISTINCT component, array_agg(DISTINCT source_vertex) AS nodes
        FROM ConnectedComponents
        GROUP BY component
        ORDER BY component"
    )

```

5.2. Evaluación de la ejecución de consultas recursivas propuestas en PostgreSQL

Finalizada la evaluación de la interfaz para realizar las consultas, se hará la evaluación de las consultas recursivas implementadas para los problemas propuestos. Esta evaluación se dividirá en dos partes: En primer lugar, se hará la revisión a cada una de las consultas implementadas para cada problema. En segundo lugar, se realizará la evaluación de PostgreSQL para la resolución de problemas de grafos utilizando la recursión provista por este sistema de bases de datos relacionales.

5.2.1. Evaluación de las consultas implementadas

La evaluación de las consultas implementadas utilizarán un dataset predefinido para el problema en particular, en pos de evaluar si fue factible la implementación de la consulta recursiva que los resuelve, basándose en la propuesta de recursión generalizada. Los datasets predefinidos se encuentran dentro del repositorio de la interfaz implementada [28] dentro de la carpeta */back/memoria_back/data*, en donde para cada problema existe un archivo del llamado $< nombre\_problema > .e$ que define las aristas del grafo.

Topological Sorting

Para la consulta de Topological Sorting se asumió que el grafo de entrada no contendría ciclos, permitiendo que esta consulta pueda terminar al visitar a todos los nodos. Esto permite que la consulta no tenga problemas al momento de ejecutar de manera iterativa la sub consulta recursiva, ya que con esta condición base eventualmente se llegará a un punto en donde todos los nodos hayan sido visitados y añadidos a la tabla temporal de trabajo, llegando a un punto fijo.

Si bien este problema está pensado para grafos acíclicos dirigidos, si es que el grafo de entrada tuviese ciclos, esta consulta, tal como está implementada, podría no llegar a un punto fijo, puesto que con las condiciones de la sub consulta recursiva se irían añadiendo en cada iteración los nodos que conforman el ciclo con un nivel extra, haciendo que las filas sean distintas, haciendo que el keyword UNION no distinga que se está añadiendo el mismo arco, dado el aumento en la columna de nivel en cada iteración. Para evitar esto, el programador puede limitar la profundidad de la recursión en la sub consulta recursiva, especificando en el JOIN que el nivel no puede ser mayor a la cantidad de nodos, tal como se hizo para el problema de Minimum Spanning Tree. Esto no evitará que se repitan aquellas aristas que generan ciclos en la tabla resultante de la consulta recursiva, pero sí permitirá que esta llegue a un punto fijo.

Connected Components

En el caso del problema de Connected Components tenemos una implementación similar al caso de Topological Sorting, con la salvedad de que este problema en particular no busca

establecer un orden dentro de los componentes del grafo, sino que solo hacer un recorrido sobre este para encontrar la componente conexa para cada nodo. Aquí no es necesario aplicar un orden sobre la consulta recursiva independiente de si el grafo contiene ciclos o no, puesto que como no se está añadiendo una nueva columna incremental como lo era el nivel en Topological Sorting, solo estamos añadiendo los nodos a cada componente y al utilizar el keyword UNION en vez de UNION ALL evitamos que en la tabla temporal de trabajo se añadan nodos repetidos a cada componente, logrando así llegar a un punto fijo cuando se recorren todos los nodos de cada componente encontrada.

Aquí podemos observar que para cierto tipo de problemas no es necesario especificar un orden si es la consulta lo amerita, dejando la responsabilidad de encontrar el punto fijo a la unión entre la consulta base y la consulta recursiva.

Eulerian Circuit

Para el problema de Eulerian Circuit ya se hace necesaria la especificación de un orden que nos permita detener la recursión, puesto que si no está presente la recursión puede correr de manera indefinida al generarse ciclos dentro de los caminos que recorre el algoritmo implementado en la consulta para este problema.

En este problema en particular, sabemos que un circuito euleriano no puede tener un largo mayor al número de vértices que componen el grafo. Con esto, se especifica en la línea 19 de la consulta propuesta para este problema 4.3, indicándole a la consulta que la profundidad no debe ser mayor a la cantidad de vértices que contiene el grafo.

Es mediante este mecanismo que un orden puede ser especificado en el contexto de una consulta recursiva en PostgreSQL, siendo para el caso de este problema añadir como condición dentro de la sub consulta recursiva de que la profundidad calculada no superase el número de aristas que componen el grafo. Esto permite que al momento de hacer la evaluación iterativa interna, esta no produzca más resultados, llegando así a un punto fijo.

Minimum Spanning Tree

Para la consulta de Minimum Spanning Tree también se tiene una condición similar a la que se presentó en el problema de Eulerian Circuit, puesto que para este problema sabemos que la profundidad del árbol no puede ser mayor a la cantidad de nodos que tiene el árbol. Esto es tomando en consideración el caso de borde en el que el grafo representara una lista enlazada de elementos. Con esto en cuenta esto, la condición de no añadir arcos que generen ciclos y de elegir siempre el arco con peso mínimo, se establece la subconsulta recursiva, siendo esta la medida para permitir a la consulta llegar a un punto fijo.

Al momento de hacer la implementación de la consulta se consideró la posibilidad de mantener una tabla externa para mantener la lista de pesos mínimos encontrados y poder accederla y actualizarla durante el paso recursivo, pero PostgreSQL no permite realizar este tipo de operaciones en el contexto de una consulta recursiva, por lo que se mantuvo la propuesta entregada, ya que esta entrega un árbol minimal dentro del grafo dada la condición de siempre elegir el arco con peso mínimo.

Maximum Matching

Como se mencionó en la sección de implementación para este problema, no es posible definir algunos de los algoritmos conocidos para encontrar un Maximum Matching dentro del marco de la propuesta de recursión generalizada. Para poder definir estos algoritmos y poder utilizarlos dentro del contexto de una consulta recursiva, es posible implementar funciones auxiliares mediante el uso del lenguaje procedural provisto por PostgreSQL para que puedan ser llamadas dentro de la sub consulta recursiva.

Planarity Testing

Finalmente, en el caso de la consulta para el problema de Planarity Testing, tenemos una consulta basada en el teorema de Kuratowski para encontrar instancias del sub grafo K_5 dentro de la tabla de vértices de entrada. Para poder completar los dos criterios que tiene este teorema será necesario implementar una consulta similar a la propuesta, pero con la finalidad de encontrar las instancias del sub grafo $K_{3,3}$. Con esto podemos apreciar que si bien existen problemas que pueden resolverse mediante la implementación para consultas recursivas que provee PostgreSQL, estos pueden necesitar más de una consulta para poder resolverlos completamente.

5.2.2. Evaluación general de las implementaciones de consultas recursivas para los problemas propuestos

A partir de la evaluación individual de las consultas implementadas para los problemas propuestos, se procede a realizar una evaluación general de la factibilidad de aplicar la propuesta de recursión generalizada en el contexto del motor de bases de datos relacionales PostgreSQL.

Desde la evaluación de las consultas implementadas se puede apreciar que tal como se especifica en la documentación de PostgreSQL [9], que a pesar de existir una manera de expresar una consulta como recursiva, esta se evalúa internamente por el sistema de PostgreSQL de manera iterativa e incremental. Esto se aprecia al realizar las consultas propuestas junto al comando EXPLAIN ANALYZE, en donde se realiza constantemente un escaneo secuencial sobre la tabla de vértices al momento de realizar un nuevo paso dentro de la recursión. Se adjunta un ejemplo en la figura 5.5 para el caso de la ejecución del problema de Connected Components en un grafo aleatorio.

Respecto al objetivo de simplificar el ingreso del orden (la medida) que permite determinar cuándo se detendrá la consulta recursiva, podemos ver que en todos los casos donde se aplica esta se refiere o bien a la cantidad máxima de nodos u aristas que puede contener. Esto se define en las restricciones impuestas en el JOIN entre la tabla recursiva. También se observó que el motor PostgreSQL no permite la realización de ciertas operaciones dentro de la consulta recursiva, como lo son la incapacidad del motor de aceptar agregaciones sobre los campos de la tabla de trabajo, la de hacer llamados a la tabla recursiva dentro de la consulta recursiva y llamar a operaciones como INSERT para ir almacenando datos en una tabla auxiliar.

Por otro lado, como se explicó en las implementaciones particulares de los problemas (co-

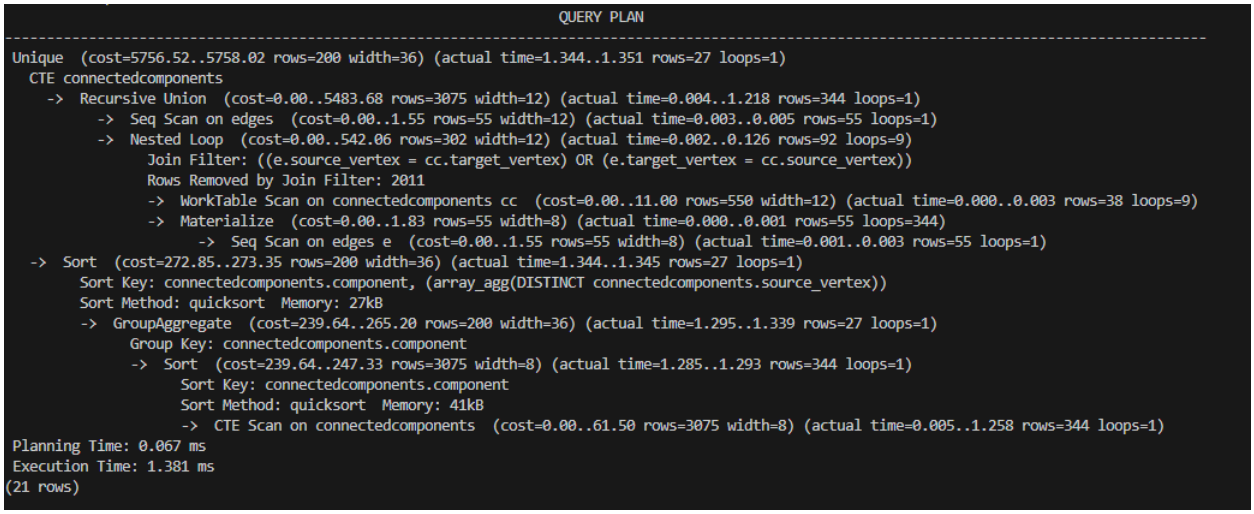

Figura 5.5: Resultado del EXPLAIN ANALYZE sobre la consulta para el problema de Connected Components para un grafo aleatorio de 30 nodos

mo en Maximum Matching), no es posible definir algunos de los algoritmos conocidos para los problemas presentados mediante solo una consulta recursiva, como está hoy en día definida en SQL. Para su implementación sería necesario definir funciones adicionales mediante una función procedural utilizando alguna librería compatible con el motor para que estas puedan ser llamadas dentro de la llamada recursiva.

Capítulo 6

Conclusión

A partir de los resultados obtenidos y su posterior evaluación se puede concluir lo siguiente:

En primer lugar, se concluye que es posible implementar una interfaz generalizada para consultas recursivas en sistemas de bases de datos relacionales. Se implementó una interfaz funcional que permite la generación de datasets que representan un grafo mediante la carga de una tabla en un sistema de bases de datos relacional. Esta generalización puede ser exportada a diferentes sistemas o plataformas para simplificar la implementación de este tipo de consultas. Esto resuelve el objetivo específico de diseñar y desarrollar una interfaz que permita la realización de consultas recursivas mediante una codificación genérica para SQL.

En segundo lugar, para el caso de PostgreSQL vemos que la recursión se realiza mediante la evaluación de la consulta base, su inserción en una tabla temporal de trabajo y la evaluación de la consulta recursiva que toma los datos de esta tabla temporal hasta que no entregue nuevos resultados. Además, como se mostró en las implementaciones y evaluaciones de los diferentes problemas a resolver, podemos concluir que es posible entregar un orden dentro de la sub-consulta recursiva, esta se reduce solamente a entregar la profundidad máxima que puede tener la recursión. Para el caso de los problemas de grafos vistos en este trabajo, esta profundidad límite dependerá del problema y puede estar dada por la cantidad máxima de nodos u aristas que deba tener la solución. Por otro lado, también se concluye que la actual implementación de recursión para bases de datos relacionales no es idóneo para resolver de manera eficiente los problemas propuestos, debiendo requerir de recursos externos para desarrollar ciertos algoritmos y de estar bajo diversas limitaciones respecto a las instrucciones que se pueden entregar en la sub consulta recursiva.

Tomando en consideración las conclusiones expuestas en los párrafos anteriores se determina que el objetivo general de estudiar el comportamiento de la recursión en el contexto de lenguajes para bases de datos relacionales SQL y de implementar un modelo más genérico para la codificación de este tipo de consultas está teóricamente resuelto, puesto que es posible escribir una consulta recursiva de manera generalizada, pero esta no es suficiente para implementar diferentes tipos de algoritmos dada la implementación actual de la recursión en los motores de bases de datos relacionales.

Finalmente, la contribución realizada por el trabajo realizado se resume en:

- La entrega de una interfaz que permite una codificación genérica para consultas recur-

sivas en SQL. Esta codificación puede ser implementada en cualquier servicio e independiente del lenguaje de programación utilizado por este para generalizar este tipo de consultas.

- La implementación -a modo de casos de prueba– de las consultas recursivas para 6 problemas conocidos de grafos que corren en tiempo polinomial, junto a la evaluación de la ejecución de cada una de estas ante un caso aleatorio y uno ideal.
- La evaluación del uso de un motor de bases de datos relacionales (PostgreSQL) para la ejecución de consultas recursivas para este tipo de problemas y la factibilidad de la factibilidad de la implementación de la propuesta de recursión generalizada para la simplificación de consultas recursivas para el caso de un motor de bases de datos SQL.

A modo de conclusión final, se muestra que si bien es posible desarrollar una interfaz generalizada para abordar problemas de grafos que corren en tiempo polinomial en un motor de bases de datos relacionales (utilizando como ejemplo el motor PostgreSQL), pero como esta memoria muestra, la recursión en los sistemas de bases de datos relacionales está lejos aún de ser práctica para este tipo de problemas.

Bibliografía

[1] Melton, J. y Simon, A. R., SQL: 1999: understanding relational language components. Elsevier, 2001.

[2] Urzúa, V., "A general approach to recursion in g-core," Master's thesis, Universidad de Chile, 2022.

[3] Ramakrishnan, R., Gehrke, J., y Gehrke, J., Database management systems, vol. 3. McGraw-Hill New York, 2003.

[4] Abiteboul, S., Hull, R., y Vianu, V., Foundations of databases, vol. 8. Addison-Wesley Reading, 1995.

[5] Aranda, G., Nieva, S., Sáenz-Pérez, F., y Sánchez-Hernández, J., "Formalizing a broader recursion coverage in sql," en Practical Aspects of Declarative Languages: 15th International Symposium, PADL 2013, Rome, Italy, January 21-22, 2013. Proceedings 15, pp. 93–108, Springer, 2013.

[6] Schüle, M. E., Kemper, A., y Neumann, T., "Recursive sql for data mining," en Proceedings of the 34th International Conference on Scientific and Statistical Database Management, pp. 1–4, 2022.

[7] "Db engines ranking." https://db-engines.com/en/ranking. Acceso: 2023-07-02.

[8] "Oracle, sql language reference." https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/SELECT.html#GUID-CFA006CA-6FF1-4972-821E-6996142A51C6. Acceso: 2023-07-02.

[9] "Postgresql." https://www.postgresql.org. Acceso: 2023-01-22.

[10] Przymus, P., Boniewicz, A., Burzańska, M., y Stencel, K., "Recursive query facilities in relational databases: a survey," en Database Theory and Application, Bio-Science and Bio-Technology: International Conferences, DTA and BSBT 2010, Held as Part of the Future Generation Information Technology Conference, FGIT 2010, Jeju Island, Korea, December 13-15, 2010. Proceedings, pp. 89–99, Springer, 2010.

[11] "Sql server, with common table expression transact sql." https://learn.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-ver15#recursive-cte-example. Acceso: 2023-07-02.

[12] "Mysql, with (common table expressions)." https://dev.mysql.com/doc/refman/8.0/en/with.html. Acceso: 2023-07-02.

[13] Ordonez, C., "Optimization of linear recursive queries in sql," IEEE Transactions on knowledge and Data Engineering, vol. 22, no. 2, pp. 264–277, 2009.

[14] Kucera, L., "Combinatorial algorithms—adam hilger," SNTL, Prague, 1990.

[15] Kahn, A. B., "Topological sorting of large networks," Communications of the ACM, vol. 5, no. 11, pp. 558–562, 1962.

[16] Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Stein, C., "Section 22.4: topological sort," Introduction to Algorithms (2nd ed.), MIT Press and McGraw-Hill, pp. 549–552, 2001.

[17] Hopcroft, J. y Tarjan, R., "Algorithm 447: efficient algorithms for graph manipulation," Communications of the ACM, vol. 16, no. 6, pp. 372–378, 1973.

[18] Hierholzer, C. y Wiener, C., "Über die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren," Mathematische Annalen, vol. 6, no. 1, pp. 30–32, 1873.

[19] Fleury, J., "Notice sur un problème de géométrie de situation," Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences, vol. 97, pp. 958–961, 1883.

[20] Kruskal, J. B., "On the shortest spanning subtree of a graph and the traveling salesman problem," Proceedings of the American Mathematical society, vol. 7, no. 1, pp. 48–50, 1956.

[21] Prim, R. C., "Shortest connection networks and some generalizations," The Bell System Technical Journal, vol. 36, no. 6, pp. 1389–1401, 1957.

[22] Edmonds, J., "Paths, trees, and flowers," Canadian Journal of mathematics, vol. 17, pp. 449–467, 1965.

[23] Ford, L. R. y Fulkerson, D. R., "Maximal flow through a network," Canadian journal of Mathematics, vol. 8, pp. 399–404, 1956.

[24] "Ldbc graphalytics benchmark (ldbc graphalytics)." https://ldbcouncil.org/benchmarks/graphalytics/. Acceso: 2023-01-22.

[25] "Python 3.11 docs." https://docs.python.org/3.11/. Acceso: 2023-01-22.

[26] "The django project." https://www.djangoproject.com/. Acceso: 2023-01-22.

[27] "React." https://reactjs.org/. Acceso: 2023-01-22.

[28] "Repositorio interfaz." https://github.com/Lxopato/memoria. Acceso: 2023-07-02.