Práctica 2 - Optimizacion de consultas

- Quintín Mesa Romero
- Doble Grado en Ingeniería Informática y Matemáticas
- quintinmr@correo.ugr.es

1. Planteamiento y descripción de la consulta

Supongamos que tenemos dos tablas: **Producto** y **Proveedor**, con la siguiente estructura:

Tabla `Producto`

Campo	Tipo	Descripción
id_producto	int	Clave primaria que identifica de forma única a cada producto
nombre_producto	varchar	Nombre del producto
precio	decimal	Precio del producto
id_proveedor	int	Clave foránea que hace referencia a la tabla `Proveedor` y representa el proveedor del producto

Tabla `Proveedor`

Campo	Tipo	Descripción
id_proveedor	int	Clave primaria que identifica de forma única a cada proveedor

Campo	Tipo	Descripción
nombre_proveedor	varchar	Nombre del proveedor
ciudad	varchar	Ciudad donde se encuentra el proveedor

Queremos consultar los productos cuyo proveedor es de Madrid. Esta consulta, viene dada por la siguiente sentencia SQL:

```sql
SELECT id_producto, nombre_producto
FROM Producto
JOIN Proveedor ON Producto.id_proveedor = Provedor.id_proveedor
WHERE Proveedor.ciudad = 'Madrid';
```

Siguiendo el álgebra relacional, la consulta se expresa de la siguiente forma:

$$\Pi_{id\_producto,\ nombre\_producto}(\sigma_{ciudad=Madrid}(Producto\ JOIN\ Proveedor))$$

Esta consulta viene dado por el siguiente árbol de expresión algebráica:

$\Pi_{id\_producto,\ nombre\_producto}$

```
        |
```

$\sigma_{ciudad=Madrid}$

```
        |
```

$JOIN$

```
    /       \
```

$Producto \quad Proveedor$

2. Representación en memoria de la consulta

2.1 Plan Lógico

Para representar la consulta que he planteado en memoria, utilizaré un **árbol**, en concreto, un **árbol de expresión** pues considero que es lo que mejor se adapta a la hora de representar en memoria un árbol de expresión algebraica.

¿Por qué he escogido esta estructura de datos y no otra? Porque:

1. Lo primero es que un árbol de expresión es la representación natural para una expresión algebraica, donde cada uno de los nodos que lo componen representa una operación y sus hijos los operandos.
2. Lo segundo es que los árboles proporcionan un acceso eficiente a los nodos y subárboles, lo cual es ventajoso a la hora de hacer consultas.
3. Permiten la representación de consultas de manera flexible, lo cual permite que se pueda extender o modificar la consulta sin modificar la estructura que ya había (en un árbol se pueden añadir o quitar nodos sencillamente).
4. Los árboles son escalables, pueden manejar consultas de gran tamaño eficientemente, lo que los hace adecuados para representar consultas que involucren muchas tablas y operaciones complejas.

Estas son algunas de las razones que me condujeron a decidir representar el árbol de expresión algebraica como un árbol de expresión.

En cuanto a los **nodos**, he diseñado la siguiente estructura de datos:

```
class Nodo {

    private:
        // Miembros de la clase
        std::string tipo;  // Tipo de operación (proyección, selección, JOIN, etc.)
        std::string tabla;  // Para nodos hoja: tabla de la base de datos
        std::string condicion;  // Para nodos de selección: condición de selección
        std::vector<Nodo> hijos;  // Lista de nodos hijos

    public:
        // Constructor
        Nodo(std::string tipo, std::string tabla = "", std::string condicion = "");
};
```

El plan lógico se podría almacenar en un árbol de Nodo:

```
  bintree<Nodo> arbol_plan_logico;
```

Plan físico

Como se nos pide que la estructura de datos que escojamos para la representación en memoria del plan físico incluya la información del plan lógico además de las estimaciones en coste de ejecución de cada operación, creo que lo más lógico sería usar un **árbol ponerado**. Así, se representa la información del plan lógico mediante un árbol, como se ha descrito anteriormente y además se indica lo que cuesta cada nodo en número de operaciones L/E cada nodo (cada operación).

Por su parte, para representar los elementos del plan físico he diseñado la siguiente estructura de datos, que reutiliza la estructura de datos utilizada para representar en memoria los nodos del árbol de expresión algebraica del plan lógico. La diferecia es que ahora, se le añade al nodo un coste de ejecución:

```
class NodoPlan {

    private:
        Nodo nodo;   // Nodo del plan lógico
        float coste; // Coste en operaciones L/E de la operación que representa el nodo

    public:
        // Constructor
        NodoPlan(Nodo nodo, float coste);

};
```

El plan físico se podría almacenar en un árbol de NodoPlan:

```
  bintree<NodoPlan> arbol_plan_fisico;
```

3. Algoritmos para optimizar la consulta

Para optimizar la consulta que he planteado al inicio de este trabajo, se plantean dos algoritmos que aplicarán dos heurísticas de optimización sobre el plan lógico asociado a la consulta.

3.1 Algoritmo 1

> *Heurística* : **Selección lo antes posible**.

Detallamos a continuación los pasos a seguir para optimizar la consulta mediante movimiento de selección:

1. Buscar en el plan lógico un nodo que incluya una operación de **selección**.
2. Verificar qué tablas contienen el atributo de la condición de la selección.
3. Evaluar la estructura del árbol para determinar si es posible mover el nodo de selección hacia abajo en el árbol (**selección lo antes posible**), buscando la posición óptima que esté más cerca posible de la tabla.
4. Si es posible, intercambiar el nodo de selección con el nodo que está

inmediatamente por encima de la tabla. Si el nodo de seleccion ya está en la posición óptima, no podemos aplicar más esta heurística.
5. En caso de que el nodo cambie; si se realiza el intercambio, actualizar tanto el puntero al nodo padre como los punteros a los nodos hijo.

3.2 Algoritmo 2

> *Heurística*: **Proyección lo antes posible**

Detallamos a continuación los pasos a seguir para optimizar la consulta mediante movimiento de proyección:

1. Identificar un nodo en el plan lógico que realice una operación de proyección.
2. Verificar si el nodo de proyección está asociado a datos provenientes de una o más tablas.
3. Analizar la estructura del árbol para determinar si es factible descender el nodo de proyección hacia niveles inferiores, procurando situarlo lo más cerca posible de la tabla que contiene los atributos necesarios para poder hacer la proyección. En el caso de requerir atributos de múltiples tablas, la proyección se dividirá para obtener los atributos innecesarios en otras operaciones y optimizar el proceso, evitando proyectar sobre tablas que no contienen los atributos requeridos.
4. Evaluar si el nodo de proyección debe ser duplicado, bien para distribuirlo entre las tablas según los atributos necesarios o bien para mantenerlo en un nivel superior en caso de que posteriormente se realice una reunión natural o un producto cartesiano que requiera los atributos proyectados. Si no es posible descender más el nodo de proyección, no se puede seguir optimizando más mediante esta heurística.
5. Realizar el intercambio o la duplicación según lo determinado en el paso anterior. Si no es viable descender el nodo de proyección, no se puede optimizar más utilizando esta estrategia.
6. En caso de intercambio o duplicación, actualizar los punteros de ambos nodos, incluyendo tanto el puntero al nodo padre como los punteros a los nodos hijos. Esto asegura la coherencia estructural del árbol lógico.