OBSERVACIÓN: Este pdf lo fuimos completando a medida que desarrollamos el código para resolver el refactoring que nos asignaron, el cual nos sirvió para poder entender mejor la consigna y como funcionan las distintas partes. Nos pareció correcto entregarlo como material extra ya que tiene una explicación más detallada sobre lo que decidimos implementar a comparación del template de la presentación.

Refactoring a utilizar y sus ventajas

Refactoring: using DISTINCT instead GROUP BY

Este refactor se trata de reemplazar GROUP BY por DISTINCT siempre y cuando se pueda realizar.

Algunas de las ventajas de utilizar DISTINCT en lugar de GROUP BY son:

1. Simplicidad:
   DISTINCT es más simple y directo. Si solo necesitas obtener valores únicos de una o más columnas sin realizar agregaciones, DISTINCT es más fácil de usar y entender.
2. Claridad::
   Si el objetivo es eliminar duplicados y no realizar cálculos agregados, el uso de DISTINCT puede hacer que la consulta sea más legible y exprese claramente la intención de obtener valores únicos.
3. Rendimiento:
   En algunos casos, DISTINCT puede ser más eficiente en términos de rendimiento que GROUP BY, especialmente si estás trabajando con conjuntos de datos grandes y solo necesitas los valores únicos sin realizar agregaciones.
4. Menos Código:
   Cuando solo necesitas obtener valores únicos, el uso de DISTINCT generalmente requiere menos código que la combinación de GROUP BY y funciones de agregación.

Pre-condiciones para aplicar el refactoring

Para poder usar este refactoring tuvimos que considerar cuáles serían las pre condiciones para decidir si se podía o no aplicar el refactoring a una consulta SQLite dada.

Luego de hacer muchas pruebas y consideraciones, encontramos las siguientes pre condiciones para poder aplicar el refactoring:

1. La consulta SQLite debe ser válida y no tener errores de sintaxis.
   Válida se refiere a que el string que se ingresa como consulta, realmente sea una consulta SQLite, y que sea válida se refiere a que si es una consulta SQLite, entonces que no tenga errores de sintaxis.

2. La consulta SQLite debe contener un GROUP BY.
   Este requerimiento se debe a que si no hay un group by, no hay forma de reemplazarlo por un distinct.

3. La consulta SQLite no debe contener un DISTINCT:
   Esto es requerido ya que una consulta SQLite solo puede tener 1 distinct en el select, si ya hubiese uno en la consulta select entonces no se podría agregar otro.

4. La consulta SQLite no debe contener funciones de agregación:
   Esto se debe a que GROUP BY, cuando contiene funciones de agregación, junta los datos por algún valor específico indicado en el GROUP BY y después realiza alguna acción, como sumar todas los datos (SUM) o calcular la media de los valores agrupados (AVG), entre otros. En cambio con DISTINCT se utiliza para elegir valores únicos de una columna específica. Por lo que si se quisiera reemplazar GROUP BY por DISTINCT y que los resultados sean equivalentes es necesario que la consulta no contenga función de agregación.

5. Las columnas que se encuentran en el SELECT deben ser las mismas que se encuentran en el GROUP BY:
   Esto es necesario ya que al utilizar el DISTINCT, se aplica en el select y a todos los datos que se encuentran después de este, se filtran las columnas por los datos que se encuentran después del SELECT DISTINCT, si quisiéramos tener más datos de los que se encuentran en el group by, entonces también el DISTINCT se estaría aplicando al dato extra que no se encontraba en el group, lo que provoca que se agregue un dato más por el que se estaría aplicando el DISTINCT y daria distintos resultados por agregar un filtro más de datos.

Por ejemplo la siguiente consulta puede aplicarse el refactoring ya que cumplen con las cinco condiciones dadas anteriormente:

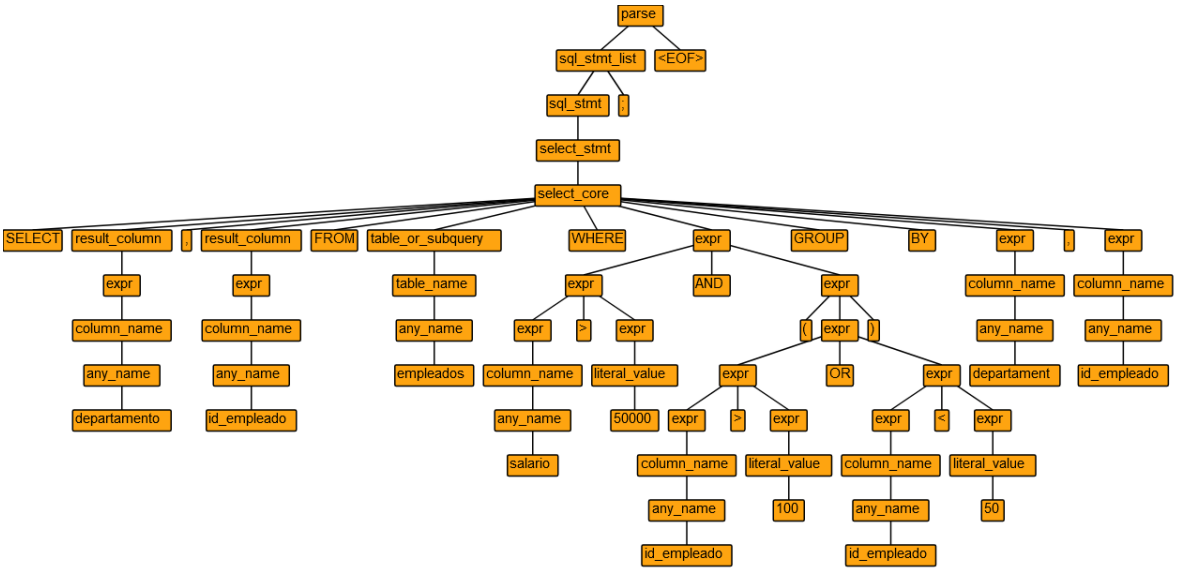
```
SELECT departamento, id_empleado
FROM empleados
WHERE salario > 50000 AND (id_empleado > 100 OR id_empleado < 50)
```

GROUP BY departamento, id_empleado;

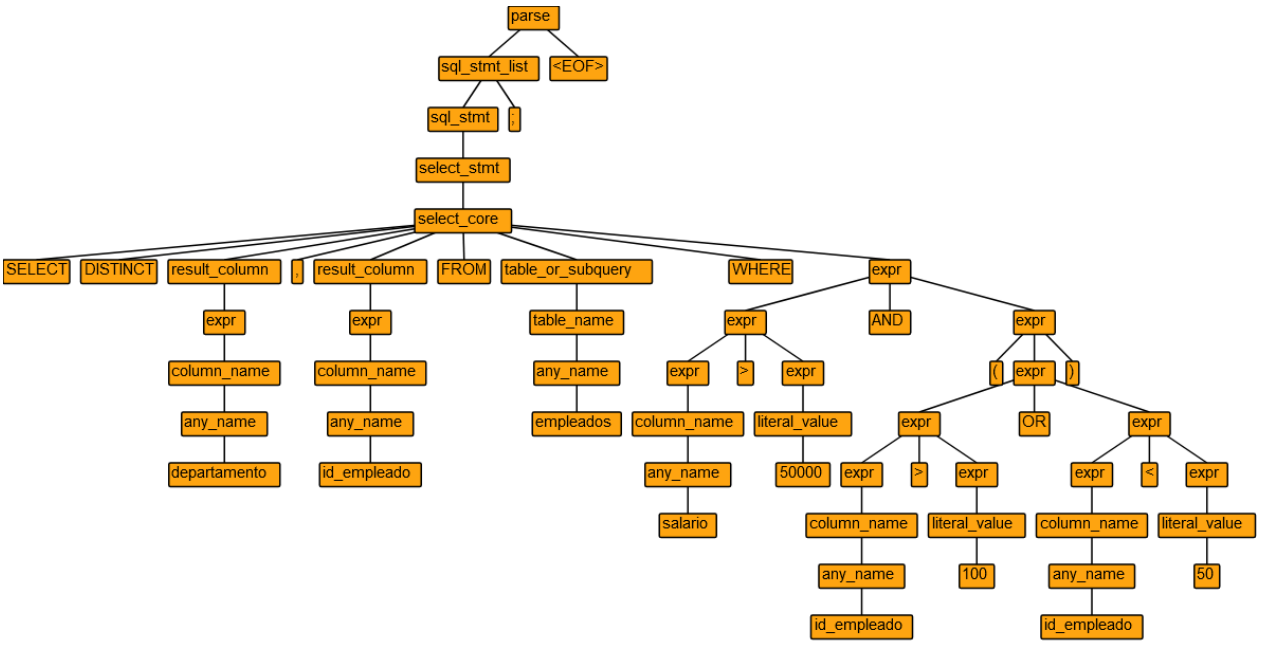
Por lo que al aplicar el refactoring el resultado de la consulta tendría que quedar de la siguiente manera:

SELECT DISTINCT departamento, id_empleado
FROM empleados
WHERE salario > 50000 AND (id_empleado > 100 OR id_empleado < 50);

Vista del árbol gramatical generado por ANTLR de la consulta inicial:



Vista del árbol gramatical generado por ANTLR como resultado de aplicar el refactoring a la consult



Por ejemplo las siguientes consultas no pueden refactorizarse:

```
SELECT category, COUNT (product) // tiene función de agregación
FROM products
GROUP BY category;
```

```
SELECT nombre, apellido, edad // en el select no debería aparecer columnas que no estén en el group by ya que agregar filtro por esas columnas
FROM personas
GROUP BY nombre;
```

Desarrollo del refactoring

Luego de las pre condiciones se procede a aplicar el refactoring a la consulta. Para este proceso decidimos utilizar el método visitSelect_core, ya que observamos que al crearse el árbol de la consulta, el nodo base que contiene una consulta SELECT completa con sus partes es el nodo select_core, entonces al acceder y modificarlo podemos trabajar con todas las partes requeridas, así como agregar DISTINCT y quitar GROUP BY.

Dentro del método visitSelect_core hacemos un recorrido de todos los nodos hijos que se encuentran dentro del subárbol que contiene la consulta SELECT, y así poder reemplazar y modificar lo necesario.

Luego nos dimos cuenta que había otras posibilidades de recorrer el subárbol. Nosotros hicimos el recorrido de los hijos en el mismo select_core, pero se podría haber hecho el recorrido de los hijos accediendo a los métodos abstractos de cada uno según corresponda, decidimos dejar el método que utilizamos en un principio ya que modificarlo requería cambiar la gran mayoría del código.

Para realizar el recorrido del árbol generado por la gramática provista por la cátedra utilizamos 2 patrones Visitor. Uno se utilizó para hacer el recorrido y analizar las consultas para las pre y post condiciones, donde no se requiere modificar el árbol. Para aplicar el refactoring utilizamos otro Visitor el cual se modifica el árbol generado.

Tanto para chequear pre o postcondiciones y aplicar el refactoring se comparan la igualdad de los tokens del nodo actual donde estoy ubicado y el nodo que necesito manipular, por ejemplo:

```
int tokenActual = terminalNode.getSymbol().getType();
if (tokenActual == SQLiteParser.SELECT_) {...}
// Si el token corresponde al nodo SELECT que es hijo del nodo select_core
```

Para saber si la consulta contiene funciones de agregación fue sencillo, ya que utilizamos el método abstracto visitFunction_name() que lo que hace es visitar los nodos que corresponden a una función de agregación, por lo que al entrar al metodo ya se estaría verificando que la consulta tiene función de agregación con el uso de una variable booleana.

También tuvimos en cuenta las consultas SQLite que tienen SELECT anidados, donde cada uno de estos SELECT podían o no contener GROUP BY, pero esto trae ciertas complicaciones.

Al tener las consultas anidadas, dentro del método que se modifica deberíamos llamar al visitor de las pre condiciones dentro del visitSelect_core para ver si la subconsulta cumple con los requisitos, pero esto estaría mal por dos razones.

Primero que los métodos de pre y post condiciones solo deberían usarse en el template method provisto por la cátedra en la clase Refactoring sin ser accedidos directamente ya que son métodos protegidos, y segundo que en el caso de que podría llamar a este visitor por cada consulta SELECT anidada, se tendrían varios visitor activos al mismo tiempo (simulando una recursión), de forma tal que mientras un visitor está activo se crea otro y dentro de este otro, y así sucesivamente la misma cantidad de veces que el número de SELECT anidados que contenga la consulta.

Para comparar si las columnas del SELECT y las columnas del GROUP BY son las mismas, creamos una lista para el SELECT y otra para GROUP BY para almacenar los nodos terminales que corresponden a sus respectivas columnas.

Para el caso del SELECT, el método ctx.result_column() que devuelve un listado de nodos “result_column” que nos permite acceder más fácilmente a las columnas del SELECT, y para el caso de GROUP BY, el método ctx.groupByExpr() que devuelve un listado de nodos “expr” correspondiente a las expresiones de GROUP BY que nos permite acceder a sus columnas.

Es importante aclarar que no almacenamos en las listas el prefijo o alias de las columnas, solo nos interesa almacenar el campo de la tabla para luego comparar.

Al momento de la comparación entre ambas listas, primero se verifica que la cantidad de columnas sean las mismas, luego se verifica que la columna de SELECT no contenga función de agregación eso implica que no se podrá refactorizar, en caso de que no tenga se comparan que los nombres de los campos sean iguales de manera ordenada.

También se verifica que la columna SELECT no contenga “*” ya que quiere decir que toma todas las columnas correspondientes de una tabla y al recibir solo un String de la consulta no podemos saber como se llaman las columnas que tiene la tabla, tambien seria un problema ya que luego el DISTINCT filtraría el resultado por todas las columnas de la tabla.

Post-condiciones para los resultados del refactoring.

Para chequear las post condiciones utilizamos la misma clase que utilizamos para las precondiciones (conditionVisitor) ya que no sería adecuado desde la vista de refactoring repetir código para hacer el recorrido del árbol, pero esta vez las condiciones cambian en comparación con las pre condiciones.

De forma resumida, para que el chequeo de postcondiciones sea válido debe cumplir lo siguiente:

1. La consulta SQLite debe ser válida y no tener errores de sintaxis.
   Al igual que en las pre-condiciones, la consulta que resulta de aplicar el refactoring debe ser una consulta SQLite válida, y si es válida, no debe contener errores de sintaxis.
2. La consulta SQLite no debe contener GROUP BY.
   Esto es requerido, ya que justamente, el refactoring aplicado elimina el GROUP BY y todos sus elementos.
3. La consulta SQLite debe contener DISTINCT, que fue agregada.
   Luego de aplicar el refactoring, además de no contener el GROUP BY, es obligatorio que contenga el DISTINCT, ya que en las pre condiciones verificamos que no contenía, pero la parte principal del refactoring es justamente agregar el DISTINCT.
4. La consulta no debe contener funciones de agregación.

Aplicación de Test (JUnit) para el refactoring.

Para la clase test, decidimos separar los métodos que devuelven resultados exitosos de los métodos que devuelven resultados fallidos.

Para los casos que cumplian condiciones y se podían refactorizar, utilizamos los assert que provee el framework JUnit, en este caso el assertEqual donde teníamos guardado en variables dentro del test la consulta SQLite válida con GROUP BY y en otra variable teníamos almacenado como quedaría esa misma consulta luego de ampliar el refactoring, con el assertEquals nos aseguramos que lo que devolvió el método transform era la consulta SQLite refactorizar correctamente.

Para los casos donde no se puede refactorizar, es decir, los casos que no cumplen las precondiciones, se utilizado assertThrows y el assertEquals, ya que dentro del material provisto por la cátedra teníamos la clase Refactoring Exception que se utiliza en la clase Refactoring, la cual envía mensajes específicos de error segun si se cumplia o no la pre y post condición.

Lo que nosotros hicimos es utilizar el método refactor con variables que contenían una consulta SQLite que no cumplía las pre condiciones, lo cual lanza una excepción del tipo correspondiente, entonces al querer aplicar el método refactor en dicho consulta se obtenía una excepción la cual comprobamos que era igual a la esperada.

En ambos métodos se testea ejecutando el método refactor de la clase Refactoring que sería el template method que realiza las precondiciones, el transform y las postcondiciones, de las cuales nosotros tuvimos que implementar cada uno de los métodos abstractos.

referencias:

[SQLite Documentation]