Comprobación de equivalencia de consultas SQL
SQL query equivalence checking

AUTORES
Daniel Muñoz García
Sergio Villanueva Agreda
Julio Martínez Sánchez
Marcos Gómez Martín

UNIVERSIDAD
COMPLUTENSE
MADRID

TRABAJO FIN DE GRADO
CURSO 2021-2022

Grado en Ingeniería Informática
Grado en Ingeniería del Software
FACULTAD DE INFORMÁTICA

Director
Enrique Martín Martín

Dedicatorias

A mi familia, por confiar en mí en mis momentos más bajos y hacerme seguir adelante, y a mis amigos por aguantarme y apoyarme hasta el final. Muchas gracias.

- Julio Martínez Sánchez

A mi familia, en especial a mi madre, la cual casi se diría que ha sufrido más que yo en los momentos más difíciles y siempre ha estado ahí para apoyarme y ayudarme a levantarme. Muchísimas gracias, de corazón.

- Sergio Villanueva Agreda

A mi novia por darme todo ese apoyo y ánimo necesario para seguir adelante, y a mi familia por apoyarme en todas las decisiones que me han hecho llegar a donde estoy ahora. Muchas gracias a todos.

- Marcos Gómez Martín

A mis padres, por su lucha y sacrificio para darme todas las oportunidades que ellos no tuvieron. A mis abuelos, por estar siempre a mi lado. A todas las personas que me han acompañado en los malos momentos. Muchas gracias.

- Daniel Muñoz García

Agradecimientos

Gracias a todas las personas que nos han acompañado durante todos estos años en la facultad. Gracias a Enrique por ayudarnos y guiarnos a lo largo de este proyecto.

Resumen

LearnSQL [1], actual juez de aprendizaje de SQL en desarrollo en nuestra facultad, ejecuta una consulta frente a una o varias bases de datos y comprueba que el resultado sea el mismo que la solución oficial proporcionada por el profesor. Esta manera de comprobar si una solución es correcta, pese a ser sencillo de implementar, presenta varios problemas, como la necesidad de realizar varias conexiones a la base de datos por cada envío recibido o, sobre todo, la limitación en los resultados, ya que dos consultas pueden devolver los mismos resultados para numerosas BD concretas pero no ser equivalentes para todas las posibles.

Este proyecto consiste en el desarrollo de una biblioteca Python que reciba dos consultas SQL y calcule su equivalencia, haciendo, en primer lugar, una transformación a su forma canónica y, en segundo lugar, comprobando que estas coincidan en su totalidad.

El trabajo consta de dos partes: la traducción de sentencias SQL a diccionarios Python, y la aplicación de las reglas del álgebra relacional [2] a estas. Todo ello tiene un fin último: la implementación de una biblioteca Python que traduzca las sentencias SQL a formato JSON [3], desarrolle las reglas mencionadas anteriormente sobre el diccionario, y verifique que la sentencia con los datos introducidos es equivalente o distinta a la sentencia a comparar. El alcance definido de nuestro trabajo será el del conjunto completo de las consultas SQL conjuntivas, ya que el problema de saber si dos consultas SQL son equivalentes es indecible, trabajando sobre el subconjunto de consultas SQL conjuntivas se pueden automatizar las comprobaciones.

El código del trabajo se encuentra alojado en un repositorio de Github y se puede acceder al mismo a través del siguiente enlace:
https://github.com/danimu03/TFG_equivalencia_SQL.git

Palabras clave

Bases de datos, juez de programación, Python, equivalencia, álgebra relacional, mo-sql-parsing, diccionario, JSON, renombramiento.

Abstract

LearnSQL, current SQL learning judge in development at our faculty, runs a query against one or more databases and checks that the result is the same as the official solution provided by the professor. This way of checking if a solution is correct, despite being simple to implement, has several problems, such as the need to make several connections to the database for each message received or, above all, the limitation in the results, since two queries can return the same results for numerous specific databases but not be equivalent for all possible ones.

This project consists in the development of a Python library that receives two SQL queries and calculates their equivalence, firstly transforming them to their canonical form and secondly checking that they coincide in their entirety.

The work consists of two parts: the translation of SQL statements to Python dictionaries and the application of relational algebra rules to them. All this has an ultimate goal: the implementation of a Python library that translates the SQL statements into JSON format, develops the previously mentioned rules about the dictionary and verifies that the statement with the entered data is equivalent or different from the statement to be compared. The defined scope of our work will be that of the complete set of conjunctive SQL queries, since the problem of knowing if two SQL queries are equivalent is inconclusive, working on the subset of conjunctive SQL queries, the checks can be automated.

The work code is hosted in a Github repository and can be accessed through the following link:
https://github.com/danimu03/TFG_equivalencia_SQL.git

Keywords

Databases, programming judge, Python, equivalence, relational algebra, mo-sql-parsing, dictionary, JSON, renaming.

Índice

Capítulo 1. Introducción

En esta sección se presenta la motivación del proyecto, los objetivos que se persiguen y el plan de trabajo que se ha seguido.

Motivación

Actualmente, los jueces automáticos que se usan en las correcciones para las consultas SQL usan principalmente la comparación de las salidas de estas. Esto permite que algunas soluciones, a pesar de que den un resultado correcto, sean erróneas para algunos casos de prueba concretos no contemplados. Como alternativa encontramos jueces automáticos que revisan las consultas en sí, sin embargo, estos algoritmos más complejos suelen ser muy costosos. Este alto coste es precisamente la razón que inspira nuestra propuesta.

En este proyecto se tomará una aproximación a estos últimos, con el espectro de las consultas conjuntivas. De esta forma el sistema que se implementará no será tan costoso al reducir el número de operaciones a realizar. Adicionalmente, se espera que tras la ejecución del programa, se sepa si las consultas a comparar son equivalentes, equivalentes salvo renombramiento, o con resultado inconcluso ya que no se puede demostrar que no lo sean.

La motivación principal de este proyecto es la existencia del juez automático LearnSQL implementado en la Facultad de Informática de la Universidad Complutense de Madrid. Este únicamente corrige los ejercicios comparando las salidas de las consultas SQL. Es por esto que queremos complementar este tipo de corrección con comparaciones entre la consulta del estudiante y la solución oficial (restringiéndose a un subconjunto manejable de consultas SQL) persiguiendo que la corrección sea aún más precisa.

Objetivos

El principal objetivo del proyecto es implementar una biblioteca Python capaz de determinar si dos sentencias SQL son equivalentes entre sí o no, aunque en algunos casos no se podrá confirmar su equivalencia. En concreto tenemos las siguientes metas:

1. Traducir las consultas SQL a expresiones del álgebra relacional en formato JSON mediante la biblioteca mo-sql-parsing [4].
2. Aplicar las reglas del álgebra relacional sobre los JSON traducidos previamente.
3. Unificar las dos partes anteriores mediante un algoritmo de selección de reglas a aplicar, en función de la consulta introducida.
4. Implementar una batería de test automáticos sobre los tres apartados anteriores.
5. Aceptar el renombramiento de tablas para que este sea irrelevante a la hora de comprobar la equivalencia entre consultas.
6. Realizar todos los aspectos anteriores en un tiempo de respuesta óptimo, de esta manera, si la aplicación tiene muchos datos de entrada a verificar, se evitarán tiempos de espera largos.

Plan de trabajo

Se elaboró un plan para el desarrollo del proyecto estructurado en cinco fases: análisis preliminar del proyecto, investigación de recursos, análisis de requisitos, implementación, testing y escritura de la memoria.

En la primera fase, análisis preliminar, llevada a cabo en el mes de septiembre, se discutió sobre la viabilidad del proyecto, así como su utilidad en un escenario real.

A continuación, durante las dos primeras semanas de octubre, se investigaron los recursos que se tendrían que usar a lo largo del proyecto. Se decidió implementar el algoritmo en Python, usando un repositorio GitHub como sistema de control de versiones. La biblioteca mo-sql-parsing para traducir las consultas SQL, el libro “Fundamentals of Database Systems” para contemplar todas las reglas del álgebra relacional a implementar, y la librería “unittest”

(https://docs.python.org/3/library/unittest.html#basic-example) que proporciona Python para la generación de test automatizados.

En las dos últimas semanas de octubre, se realizó el análisis de los requisitos a tener en cuenta en el desarrollo del proyecto. Como resultado se estableció que el primer requisito a cumplir sería realizar una documentación sobre las definiciones de SQL a JSON, y otra sobre las reglas del álgebra relacional que se iban a implementar. Ambas documentaciones deberían incluir ejemplos que servirían como guía para la implementación del algoritmo y de los tests.

A partir de noviembre se dio comienzo a la fase de implementación, se dividió al grupo en dos equipos, el primero encargado de las traducciones de las consultas SQL, y el segundo responsable de la aplicación de las reglas del álgebra relacional. Esta fase duraría hasta febrero al ser la más densa del proyecto. Una vez alcanzada la etapa final de desarrollo, daríamos paso al comienzo de la implementación de los renombramientos en marzo, que duraría hasta abril ya que este apartado resultaría ser el más exigente a nivel de dificultad.

A la vez que se implementaba el sistema, se desarrollaban test automatizados para verificar el correcto funcionamiento de las traducciones, las reglas AR y los renombramientos mediante la librería “unittest”.

Concluidos todos los aspectos anteriores, en mayo se dio comienzo a la escritura de la memoria, que contaría con los capítulos que se mencionan a continuación en el capítulo 1.4.

Organización de la memoria

Al margen del primer apartado de introducción (capítulo 1) en el que se explican la motivación y los objetivos perseguidos, así como también el plan de trabajo y la organización de la memoria aplicados, el resto de la memoria se estructura como sigue. Este primer capítulo se traduce al inglés en el capítulo 2.

En el capítulo 3 se presentan y se explican todas las tecnologías y conceptos que forman parte del TFG.

En el apartado 4 se especifican tanto la organización del sistema como las componentes de este, cómo se conectan entre sí, y qué tecnologías se utilizan para llevar a cabo el correcto funcionamiento del sistema.

En el siguiente capítulo, el 5, se explica en detalle el proceso de generación de expresiones del álgebra relacional a partir de consultas SQL, además de la función de renombramiento.

En el capítulo 6 se detalla todo el procedimiento de reducción de expresiones del álgebra relacional mediante sus respectivas reglas, y la comparación final de equivalencia con la aplicación de renombramientos.

En el capítulo 7 se exponen los métodos desarrollados para la validación del sistema, por medio de la implementación de tests de unidad y comprobación del código.

El capítulo 8 expone las conclusiones del desarrollo del proyecto, así como una argumentación de los resultados obtenidos, y el trabajo futuro que se podría llegar a realizar.

El capítulo 9 está redactado en inglés y, en él, se presentan las mismas características que el capítulo anterior.

Por último, en el capítulo 10, se describen las contribuciones de cada integrante del proyecto.

Capítulo 2. Introduction

This section presents the motivation for the project, the objectives pursued and the work plan followed.

Motivation

Currently, the automatic judges used in SQL query corrections mainly use the comparison of query outputs. This allows some solutions, although giving a correct result, to be wrong for some specific test cases not covered. Alternatively, there are automatic judges that check the queries themselves, however, these more complex algorithms are often very expensive. This high cost is precisely the reason behind our proposal.

In this project we will take an approach to the latter, with the spectrum of conjunctive queries. In this way, the system to be implemented will not be so costly by reducing the number of operations to be performed. Additionally, it is expected that after the execution of the program, it will be known whether the queries to be compared are equivalent, equivalent except renaming, or with inconclusive result since it cannot be proved that they are not.

The main motivation for this project is the existence of the LearnSQL automatic judge implemented in the Faculty of Computer Science of the Complutense University of Madrid. It only corrects the exercises by comparing the outputs of SQL queries. This is why we want to complement this type of correction with comparisons between the student's query and the official solution (restricted to a manageable subset of SQL queries) in order to make the correction even more accurate.

Objectives

The main goal of the project is to implement a Python library capable of determining whether two SQL statements are equivalent to each other or not, although in some cases it will not be possible to confirm their equivalence. Specifically, we have the following goals:

1. Translate SQL queries into relational algebra expressions in JSON format using the mo-sql-parsing library.
2. Apply the rules of the relational algebra on the previously translated JSON.
3. Unify the two previous parts by means of an algorithm for selecting the rules to be applied, depending on the query entered.
4. Implement a battery of automatic tests on the three previous sections.
5. Accept the renaming of tables so that this is irrelevant when checking the equivalence between queries.
6. Carry out all of the above aspects in an optimal response time, so that if the application has a lot of input data to verify, long waiting times can be avoided.

Work plan

A plan for the development of the project was elaborated and structured in five phases: preliminary project analysis, resource investigation, requirements analysis, implementation, testing and report writing.

In the first phase, preliminary analysis, carried out in September, the feasibility of the project was discussed, as well as its usefulness in a real scenario.

Then, during the first two weeks of October, the resources that would have to be used throughout the project were investigated. It was decided to implement the algorithm in Python, using a GitHub repository as a version control system. The mo-sql-parsing library to translate SQL queries, the book "Fundamentals of Database Systems" to contemplate all the rules of relational algebra to be implemented, and the "unittest" library (https://docs.python.org/3/library/unittest.html#basic-example) provided by Python to generate automated tests.

In the last two weeks of October, the analysis of the requirements to be taken into account in the development of the project was carried out. As a result, it was established that the first requirement to be met would be to create documentation on the SQL to JSON definitions, and another on the relational algebra rules to be implemented. Both documentations should include examples that would serve as a guide for the implementation of the algorithm and the tests.

From November onwards, the implementation phase began, the group was divided into two teams, the first in charge of translations of the SQL queries, and the second responsible for the application of the relational algebra rules. This phase lasted until February as it was the most intensive phase of the project. Once the final stage of development had been reached, we would move on to the start of the implementation of the renames in March, which would last until April as this section would prove to be the most demanding in terms of difficulty.

While the system was being implemented, automated tests were developed to verify the correct functioning of the translations, the AR rules and the renames using the "unittest" library.

Once all of the above aspects had been completed, in May the writing of the report began, which would include the chapters mentioned below in chapter 1.4.

Organization of the report

Apart from the first introductory section (Chapter 1), which explains the motivation and objectives pursued, as well as the work plan and organization of the report, the rest of the report is structured as follows. This first chapter is translated into English in chapter 2.

In chapter 3 all the technologies and concepts that are part of the dissertation are presented and explained.

Section 4 specifies both the organization of the system and its components, how they are connected to each other, and which technologies are used to carry out the correct operation of the system.

The next chapter, chapter 5, explains in detail the process of generating relational algebra expressions from SQL queries, as well as the renaming function.

Chapter 6 details the whole procedure of reducing relational algebra expressions by means of their respective rules, and the final comparison of equivalence with the application of renames.

Chapter 7 presents the methods developed for the validation of the system, by means of the implementation of unit tests and code verification.

Chapter 8 presents the conclusions of the development of the project, as well as an argumentation of the results obtained, and the future work that could be carried out.

Chapter 9 is written in English and presents the same characteristics as the previous chapter.

Finally, Chapter 10 describes the contributions of each member of the project.

Capítulo 3. Preliminares

Dedicamos este capítulo a profundizar en algunos términos y tecnologías necesarios para entender el funcionamiento de esta biblioteca.

SQL

El lenguaje estructurado de consultas (SQL, Structured Query Language) [5, 6] apoya la creación y mantenimiento de la base de datos relacional y la gestión de los datos dentro de la base de datos. Las bases de datos pueden referirse a cualquier cosa, desde una colección de nombres y apellidos, hasta un sistema complejo de almacenamiento de datos que se basa en interfaces de usuario. La base para SQL es el modelo relacional, el cual se basa principalmente en los principios de la teoría de conjunto y lógica de predicados. El núcleo del modelo relacional es la relación, que es el conjunto de columnas y filas reunidas en una estructura en forma de tabla. Esto sirve para crear entidades con datos relacionados. Una entidad puede ser cualquier cosa, una persona, lugar, evento, el cuál tiene sus atributos, nombre, ciudad, día de realización. Para toda esta cantidad de datos recopilados se utiliza SQL, lo que convierte grandes volúmenes de datos en información útil.

Consultas conjuntivas

Son consultas de la forma SELECT [7] FROM WHERE donde las condiciones WHERE son igualdades combinadas con AND.

```
SELECT nombre FROM persona
WHERE ciudad = ‘Madrid’ AND estudios = ‘S’;
```

Consultas equivalentes

Dos consultas son equivalentes cuando el resultado de ambas consultas es el mismo, es decir, las columnas resultantes y los datos mostrados son los mismos para todas las posibles bases de datos.
Otra manera de comprobar si dos consultas son equivalentes es ver si las formas canónicas en álgebra relacional de las consultas SQL son iguales.

Renombramiento

Para simplificar la utilización de varias tablas en una misma consulta se utiliza el renombrado de las mismas. Este método no modifica el nombre de las tablas en la base de datos y permite entender más fácilmente la totalidad de la consulta SQL.

```
SELECT nombre, puesto FROM Persona, Trabajo
WHERE id = persona_id;
```

En el ejemplo anterior no se entiende que el nombre corresponde a la tabla Persona y puesto a la de Trabajo. Mediante renombramiento, la consulta anterior se entiende de manera más sencilla.

```
SELECT p.nombre, t.puesto FROM Persona p, Trabajo t
WHERE p.id = t.persona_id;
```

La problemática principal con el renombramiento surge del cálculo de equivalencias de dos consultas basándose únicamente en el análisis de sus cadenas de texto. En dos consultas exactamente iguales, en las que se realiza un renombramiento

distinto se obtendría como resultado que dichas consultas no son equivalentes, cuando el resultado correcto sería una equivalencia total. Esto lo observamos en las dos siguientes consultas.

```
SELECT p.nombre, t.puesto FROM Persona p, Trabajo t
WHERE p.id = t.persona_id;
```

```
SELECT a.nombre, b.puesto FROM Persona a, Trabajo b
WHERE a.id = b.persona_id;
```

Álgebra relacional

El álgebra relacional [8] es un lenguaje de cálculo sobre relaciones, ya que se tratan de operaciones que reciben y devuelven relaciones. Consta de varias operaciones que toman como entrada una o dos relaciones y producen como resultado una nueva relación.

Las operaciones que consideramos son:

- (**σ**) Selección.
- (**Π**) Proyección.
- (**X**) Producto cartesiano.
- (⋈) Reunión.
- (**ρ**) Renombramiento.

Reglas de simplificación del álgebra relacional

Se trata de reglas de equivalencia, las cuáles indican que una expresión en una forma es equivalente a otra de distinta forma. De ésta manera, podemos sustituir una expresión por otra para generar una expresión con el mismo resultado. Utilizamos éstas reglas sobre los datos introducidos para obtener una salida más simple y que nos facilita su uso y comparación.

Un ejemplo de cómo ayuda la aplicación de las reglas a la simplificación del resultado podría ser el siguiente:

$$\Pi_{\text{nombre}} (\sigma_{\text{edad=25}} (\sigma_{\text{trabajo=“Profesor”}} (\text{Personas})))$$

Las reglas, a la vez que simplifican el resultado, nos permiten obtener variaciones equivalentes. En este caso, aplicando un método del álgebra relacional, la expresión resultante sería:

$$\Pi_{\text{nombre}} (\sigma_{\text{trabajo = “Profesor”}} (\sigma_{\text{edad = 25}} (\text{Personas})))$$

A su vez, esta expresión es equivalente a las siguientes, debido a que las igualdades son conmutativas:

$$\Pi_{\text{nombre}} (\sigma_{\text{“Profesor” = trabajo}} (\sigma_{\text{25 = edad}} (\text{Personas})))$$

$$\Pi_{\text{nombre}} (\sigma_{\text{25 = edad}} (\sigma_{\text{“Profesor” = trabajo}} (\text{Personas})))$$

LearnSQL

LearnSQL es un juez automático para el aprendizaje de las bases de datos implementado con el lenguaje de programación Python utilizando el framework de desarrollo Django. Actualmente se encuentra disponible en un servidor accesible desde https://learn.fdi.ucm.es/sql/, aunque sólo acepta el registro de estudiantes que están matriculados en la asignatura de “Base de Datos” de la Facultad de Informática y sus profesores. Al igual que con otros jueces automáticos que se usan en esta Facultad, los estudiantes pueden acceder a los problemas y realizar envíos con las soluciones de estos, mientras que los profesores tienen total acceso a la plataforma como administradores, donde pueden crear, borrar, actualizar y consultar todos los elementos del juez.

Python

Python [9, 10] es un lenguaje de programación de alto nivel, orientado a objetos y con semántica dinámica. Las estructuras de datos integradas son de alto nivel, que combinadas con la escritura dinámica y el enlace dinámico, lo hacen muy útil para el desarrollo rápido de aplicaciones, así como para su uso de conexión entre componentes existentes. Su sintaxis simple y fácil de aprender reduce el costo del programa, a su vez de ser así un lenguaje simple de aprender. Python además admite paquetes y módulos, lo que mejora la reutilización del código. El intérprete de Python y la biblioteca estándar son software libre sin cargo de uso o distribución para todas las plataformas.

Biblioteca mo-sql-parsing

Esta biblioteca Python convierte sentencias SQL en diccionarios Python, los cuales nosotros utilizamos para traducir la consulta SQL en álgebra relacional, como explicamos en apartados posteriores.

Capítulo 4. Organización del sistema

Esta biblioteca tiene dos componentes principales claramente diferenciados. La primera de ellas es la traducción a álgebra relacional a partir de las consultas SQL; la segunda, la reducción de las expresiones en álgebra relacional. Para lograr la conexión entre ambas componentes fue necesario establecer la forma con la que se representan las expresiones en álgebra relacional.

En la figura 4.1 se puede observar la estructura general de esta biblioteca. Se introducen dos consultas SQL junto con las sentencias de creación de tablas correspondientes. Cada una de las consultas se traduce a álgebra relacional, expresadas en formato JSON. Estas expresiones, mediante la aplicación de una serie de reglas del álgebra relacional, se reducen a su forma canónica. Finalmente, un comprobador determina si las consultas SQL introducidas son equivalentes.

Representación de expresiones en álgebra relacional

Para unificar ambas componentes del proyecto tuvimos que diseñar una manera de representar expresiones de álgebra relacional en Python. En este caso, decidimos que lo idóneo era utilizar el formato JSON, formato que nos permite representar fácilmente las operaciones y operandos de álgebra relacional, así como también el implementarlo, tanto para su generación como su lectura y posible modificación. Concretamente en Python los JSON se representan como diccionarios, que son con los que trabajamos a lo largo del desarrollo.

Este formato no sólo se tiene que utilizar como unión de las dos componentes que conforman el proyecto, sino que también es necesario que se mantenga tras la ejecución de todas las reglas de simplificación ejecutadas durante la segunda parte de la ejecución del programa.

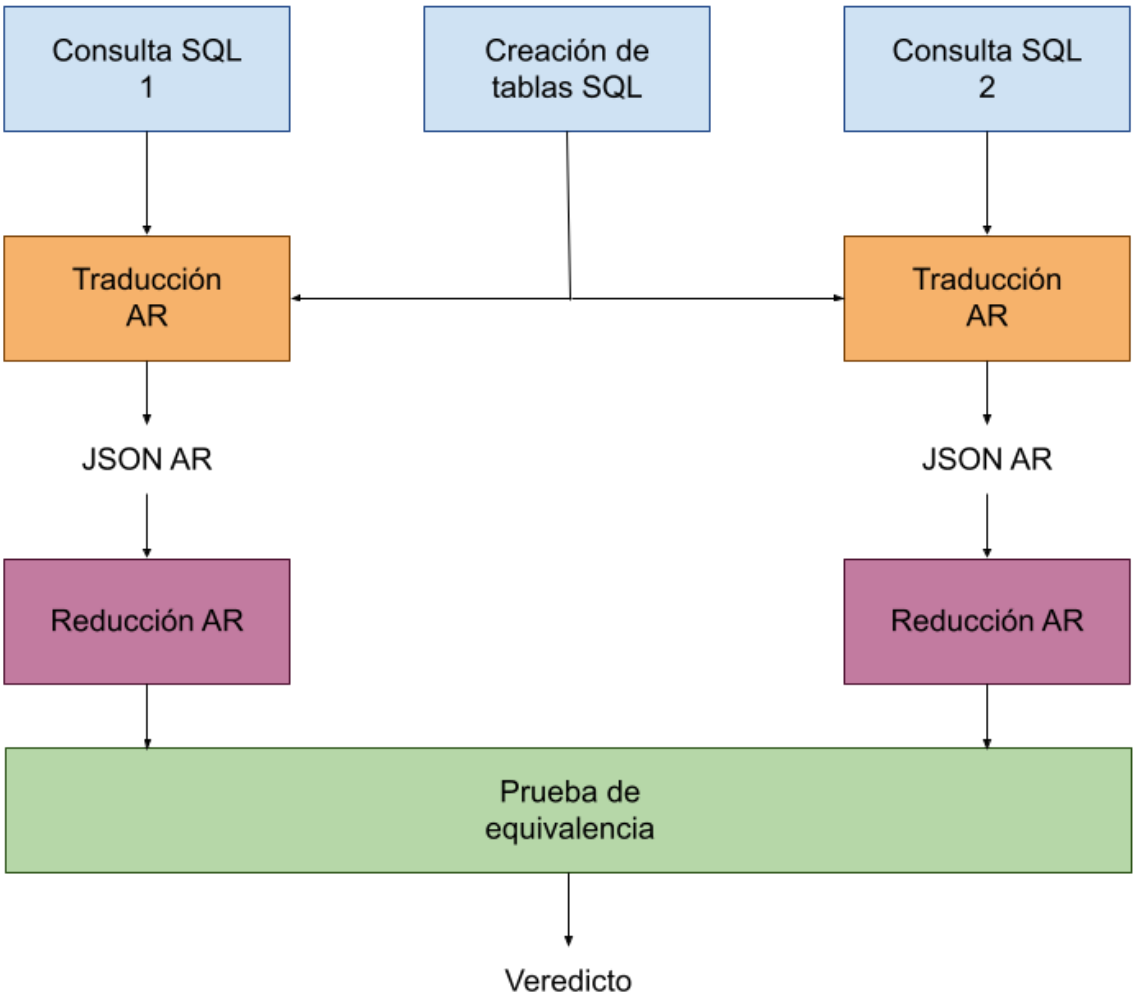


Figura 4.1. Mapa general de la biblioteca

Traducción a álgebra relacional

Esta parte se encarga de la generación de expresiones en álgebra relacional a partir de consultas en lenguaje SQL.

La idea general es que, dada una consulta SQL, así como las consultas relacionadas con la creación de tablas, se genere un JSON según el formato acordado. Este proceso se realiza siguiendo los siguientes pasos:

1. Mediante la biblioteca *mo-sql-parsing* se analiza la consulta SQL y nos devuelve un JSON.
2. Se comprueba que no haya alguna clave que no esté contemplada en la traducción que realizamos.
3. Se ejecuta el renombramiento de tablas y columnas teniendo en cuenta las consultas de creación de tablas originales.
4. Se realiza la traducción en orden:
    a. SELECT
    b. AND
    c. WHERE
    d. FROM

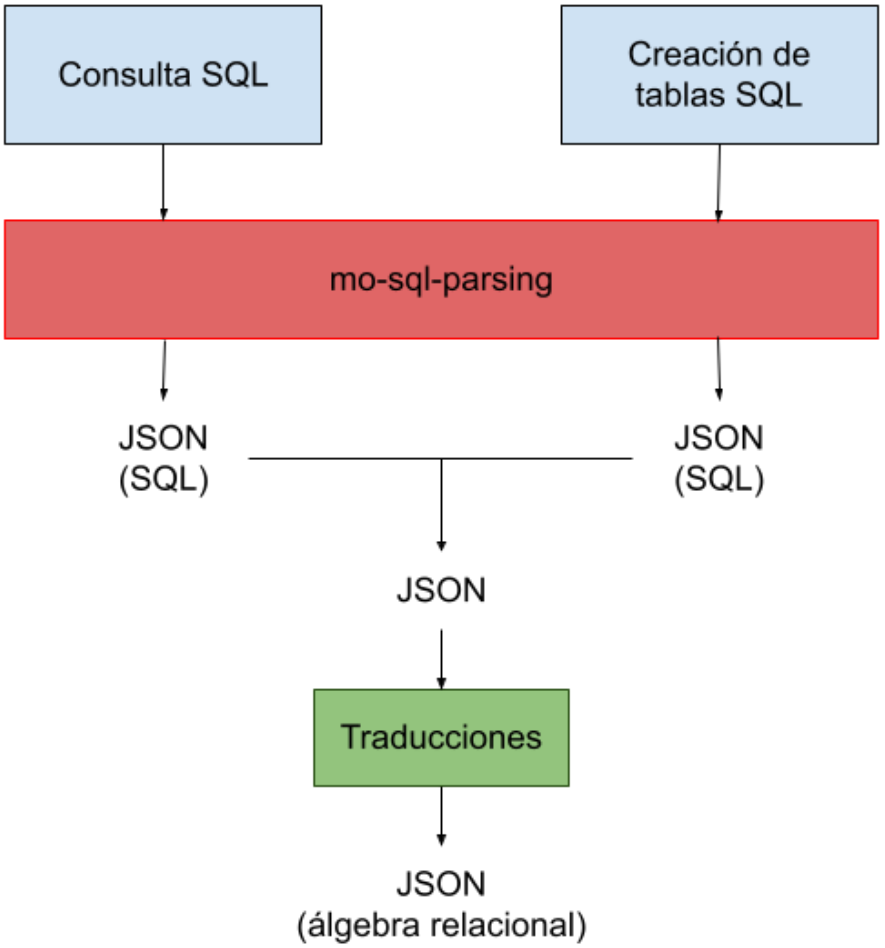


Figura 4.2. Mapa general del módulo de traducción a álgebra relacional

Reducción a partir de reglas

Esta parte consiste en, con los diccionarios obtenidos de la fase anterior, aplicar todas las posibles reglas del álgebra relacional para simplificar el JSON resultante. Una vez se han simplificado los dos diccionarios a su forma más sencilla, se hace una comprobación literal de los objetos. Si son iguales, podemos decir que las consultas son equivalentes, y de no serlo, pasaremos a la comprobación bajo renombramientos. Si tras ésta comprobación las expresiones son iguales, podemos decir que son equivalentes salvo renombramiento, y si no es el caso, el resultado será no sabemos.

Capítulo 5. Generación de expresiones AR a partir de consultas SQL

La tarea a realizar en esta parte del desarrollo de nuestra biblioteca consiste en la generación de expresiones en álgebra relacional a partir de consultas en lenguaje SQL. La idea principal es la de crear una función que reciba como parámetro un cadena de texto que contenga la consulta SQL, aplicar sobre ella las transformaciones necesarias y devolver la expresión en álgebra relacional.

En este momento surge la necesidad de establecer un formato común de las expresiones del álgebra relacional para todos los módulos de nuestra biblioteca. Por su integración con el lenguaje utilizado decidimos trabajar con formatos JSON o diccionarios, lo que nos facilita el manejo de dichas expresiones para realizar las operaciones necesarias sobre ellas.

Dado que no existe un estándar definido para las expresiones del álgebra relacional como diccionarios Python o JSON, generamos uno propio en el que tratamos las operaciones del álgebra relacional que nos vamos a encontrar en nuestras consultas, estableciendo para cada una de ellas un formato interno estandarizado.

Estándar JSON de operaciones del álgebra relacional.

Establecemos un formato genérico estandarizado interno para las operaciones de proyección, selección, renombramiento, producto cartesiano, *join*, condiciones y relaciones. Este formato debe mantenerse en todo el proceso interno de cálculo de equivalencias, tanto tras la traducción de consultas en lenguaje SQL como tras la aplicación de reglas.A continuación, mostramos los distintos formatos para cada operación definida.

- Relation

```
{   "type" : "rel",
    "table" : "name"
}
```

- Conditions

  País = “España” ^ DNI = “02311985R”

```
{ "type" : "and",
  "values" : [ {"type" : "eq",
                "values" : ["País", {"literal" : "España"}]},
               { "type" : "eq",
                 "values" : ["DNI", {"literal" : "02311985R"}]}
             ]
}
```

En las condiciones solo aparecen los operadores "eq" y "and" ya que es el alcance definido dentro de nuestro proyecto donde se delimita el cálculo de equivalencias de consultas únicamente conjuntivas.

- Selection

  $\sigma_{\text{nombre = “Marta”}}$ Persona

```
{ "type" : "sigma",
  "cond" : { "type" : "eq",
             "values" : [ "nombre",{"literal" : "nombre"}]},
             "rel" : { "type" : "rel",
                       "table" : "Persona"}
}
```

- Projection

  $\Pi_{\text{Nombre, Apellidos}}$

```
{ "type" : "pi",
  "proj" : ["Nombre", "Apellidos"],
  "rel" : { "type" : "rel",
            "table" : "Persona"}
}
```

- Renombramiento

  $\rho_{\text{Club/NombreClub}}$

```
{   "type" : "rho",
    "ren" : ["Club" , "NombreClub"],
    "rel" : { "type" : "rel",
              "table" : "Clubes"}
}
```

- Producto cartesiano

  Jugador **X** Club

```
{   "type" : "pro",
    "lrel" : {   "type" : "rel",
                 "table" : "Jugador"},
    "rrel" : {   "type" : "rel",
                 "table" : "Club"}
}
```

- Join

  Jugador $\bowtie_{\text{nombreClub = nombre}}$ Club

```
{   "type" : "join",
    "cond" : {   "type" : "eq",
                 "values" : ["nombreClub", "nombre"]
},
    "lrel" : {   "type" : "rel",
                 "table" : "Jugador"
             },
    "rrel" : {   "type" : "rel",
                 "table" : "Club"
             }
}
```

Traducción SQL a diccionario Python

Tras definir el formato para las expresiones en álgebra relacional, el siguiente paso consiste en realizar la traducción de la consulta SQL. Para ello hacemos uso de una biblioteca de Python ya existente, mo-sql-parsing.

Esta biblioteca tiene como objetivo convertir las consultas SQL en formato JSON (diccionarios Python). Originalmente estaba dirigido a MySQL, pero tras su crecimiento y nuevas versiones se han ido incluyendo nuevos motores de bases de datos, soportando en la actualidad los más utilizados como MariaDB o PostgreSQL.

Para realizar esta transformación se utiliza la función `parse` de la biblioteca mo_sql_parsing, la cual recibe un parámetro que contiene la consulta SQL y devuelve un diccionario con esta consulta en un formato JSON preestablecido.

La biblioteca soporta todo tipo de sentencias SQL, definición de datos, control de acceso y manipulación de datos. No obstante, durante nuestra implementación solo haremos uso de las sentencias DML y DDL.

Esta biblioteca nos libera del trabajo de tener que realizar un análisis completo de la cadena de texto representando las sentencias SQL a formato JSON, de esta manera trabajaremos ya con la estructura de datos donde deseamos almacenar nuestros resultados, sobre la que realizaremos distintas operaciones para transformarla y obtener el formato establecido anteriormente de expresiones del álgebra relacional en formato JSON.

El alcance de nuestra biblioteca, definido únicamente para consultas conjuntivas, hace necesario realizar un chequeo tras la obtención de la sentencia SQL en formato JSON, esto se debe a que podrían recibirse consultas no soportadas en nuestra biblioteca pero sí soportadas por la biblioteca mo-sql-parsing. El tratamiento de una sentencia no soportada supondría un error fatal dentro de nuestro proceso de análisis para obtener la expresión en álgebra relacional, por esta razón se detecta de manera prematura y se aborta el proceso en caso de ser necesario. Para ello creamos una función auxiliar encargada de comprobar si todos los operadores SQL que contiene este primer diccionario están soportadas, definiendo previamente qué operadores son:

("select", "from", "join", "on", "eq", "where", "and", "value", "cross join", "name", "literal")

- Name: en la biblioteca mo-sql-parsing hace referencia al nombre de las tablas.
- Value: en la biblioteca mo-sql-parsing hace referencia a los nombre de las columnas.
- Literal: en la biblioteca mo-sql-parsing hace referencia al tipo cadena de texto.

Ejemplo de traducción

Consulta SQL:

SELECT Nombre FROM Persona WHERE Pais = "España"

Álgebra relacional:

$$\Pi_{\text{Nombre}}\ \sigma_{\text{País = “España”}}\ (\text{Persona})$$

Resultado de la traducción:

```
{ "type" :  "pi",
  "proj" :  ["Nombre"],
  "rel" : { "type" : "sigma",
            "cond" : { "type"  :  "eq",
                       "values" : ["País", {"literal" : "España"}]}
             "rel" : { "type" : "rel",
                       "table" : "Persona"}}
}
```

Renombramiento de las consultas SQL.

La operación de renombramiento de las consultas SQL supone una de las mayores dificultades a la que nos enfrentamos en el desarrollo de este módulo. Esto se debe a que se presentan un gran número de posibilidades a la hora de renombrar las consultas, por lo que debemos tomar una decisión sobre cómo vamos a abordar de manera general todos estos casos.

Para ello tratamos todas las posibilidades que se nos presentan de la misma forma, realizar un renombramiento propio a todas las sentencias SQL a las que nos enfrentamos, tengan ellas ya un renombramiento propio o no. De esta manera obtenemos unos resultados estandarizados.

Este renombramiento propio consiste en dar un alias único a cada tabla según se va encontrando. Este alias será el propio nombre de la tabla más un número que indica la cantidad de apariciones de dicha tabla en la consulta.

```
SELECT *
FROM Club, Club

->

SELECT *
FROM Club Club1, Club Club2
```

De la misma forma tratamos todos los campos que aparecen en las condiciones del JOIN o WHERE y de la cláusula SELECT, decorándolos con el formato Alias.Campo

```
SELECT nombre
FROM Club

->

SELECT Club1.nombre
FROM Club Club1
```

Si se presenta una consulta en la cual ya existe un renombramiento previo, este se elimina y se lleva al renombramiento que hemos establecido.

A su vez se deriva la necesidad de crear una excepción propia para controlar los numerosos casos que se nos presentan en los cuales una consulta puede estar sintácticamente bien escrita, por lo que no supone ningún problema en la primera fase de transformación de la biblioteca de mo-sql-parsing, pero existe la posibilidad de, entre otros errores, referenciar a columnas inexistentes o ambiguas.

Estos fallos son detectados en el proceso de renombramiento, y para ello será necesario tener las información de las consultas de creación de todas las tablas que intervienen en la consulta. A continuación, se describen los casos particulares que se dan en el proceso de renombramiento una vez establecido nuestro formato estándar:

- Columna ambigua

  Este problema se presenta cuando hay una referencia a alguna columna cuyo nombre coincide en dos o varias tablas de la consulta y no existe un renombramiento previo que permita identificar a cual se refiere. En caso de producirse se lanzaría nuestra excepción `ErrorRenameSQL('ERROR: consulta ambigua')`.

- Columna inexistente

  Este problema se presenta cuando hay una referencia a alguna columna cuyo nombre no coincide con ninguna de las columnas de las tablas de la consulta. En caso de producirse se lanzaría nuestra excepción `ErrorRenameSQL('ERROR: no coincide la columna a proyectar con ninguna de las tablas de From')`.

- Renombramiento previo

  Este problema se presenta cuando la consulta trae un renombramiento previo y este no concuerda con ninguna tabla, esté renombrada o no. En caso de producirse se lanzaría nuestra excepción `ErrorRenameSQL('ERROR: no concuerda con el renombramiento de ninguna tabla')`.

Para encontrar estos casos, es necesario realizar una comprobación de cada columna referenciada así como de los nombres de las tablas que intervienen en la consulta SQL y tenemos almacenada en el diccionario con las sentencias de creación SQL.

Otra dificultad a la que nos enfrentamos es que en un primer momento la biblioteca mo-sql-parsing, en su formato JSON, no diferenciaba entre los nombres de columnas de tablas y las cadenas de texto, lo que nos supone una imposibilidad a la hora de renombrar dichas columnas de tablas en la cláusula WHERE si previamente no estaban renombradas. Esta complicación, sin embargo, no sucede si la igualdad se produce con un tipo distinto a cadena de texto. Esta situación se aprecia en la siguiente consulta donde se observa la situación descrita en el resultado de la transformación realizada por la biblioteca mo-sql-parsing.

SELECT *
FROM Club
WHERE nombre = “Club Deportivo Leganés”

mo-sql-parsing:

```
{ “where” : { “eq” : [“nombre”, “Club Deportivo Leganés”]}}
```

Este inconveniente se soluciona tras la actualización de versión de la biblioteca con la cual se identifica las cadenas de texto en una igualdad mediante el operador “literal”, el mismo se encuentra disponible a partir de la versión `mo-sql-parsing==8.146.22081`.

```
SELECT *
FROM Club
WHERE club.nombre = “Club Deportivo Leganés”
```

mo-sql-parsing:

```
{ “where” : { “eq” : [“club.nombre”, { “literal” : “Club Deportivo Leganés”}]}}
```

De esta forma controlamos todos los posibles errores que pueden surgir en el renombramiento, lanzando excepciones en cada caso necesario y estableciendo un formato estandarizado que se aplica siempre, independientemente de si la consulta tiene un renombramiento propio o no.

Este proceso se aplica sobre el diccionario que contiene la consulta SQL en formato JSON, ya que resulta así más fácil de aplicar que sobre la consulta ya transformada a álgebra relacional.

Funciones para la transformación de sentencias SQL a álgebra relacional

Tras la transformación de la sentencia SQL a formato JSON, el control de los operadores soportados y el renombramiento de tablas y columnas, procedemos a la creación de una biblioteca encargada de hacer la transformación de las sentencias SQL a las expresiones correspondientes de álgebra relacional en nuestro formato estandarizado.

Nuestra función `literal_to_JSON`, que recibe un diccionario con la sentencia SQL se encarga de llevar a cabo la cabo la transformación a la expresión del álgebra relacional correspondiente, haciendo uso de algunas funciones auxiliares que nos ayudan en el proceso, y devuelve un diccionario con la transformación realizada.

Durante este proceso se va transformando el JSON recorriendo cada uno de sus pares clave-valor, procesando cada uno de ellos con las funciones auxiliares creadas. Este desarrollo viene muy marcado por el formato JSON que devuelve la biblioteca mo-sql-parsing, ya que los recorridos que hacemos para generar nuestro nuevo diccionario vienen determinados por dicho formato.

Este formato plantea una serie de dificultades, algunas de ellas son:

- Es necesaria una función auxiliar para saber si una consulta tiene una proyección o join.
- Es necesaria una función auxiliar para saber si hemos tratado todos los joins en una consulta.

- Es necesaria una función auxiliar para saber si hemos tratado todos los productos cartesianos en una consulta.

La implementación general de este proceso de transformación se puede resumir en recorridos del diccionario previo y, obteniendo la información de las claves, generamos nuestro nuevo diccionario que posteriormente se devuelve.

Se realiza la transformación de todas las operaciones soportadas por nuestra biblioteca, haciendo uso de las diferentes funciones auxiliares implementadas.

Parse_Sql_To_Json

Todo el proceso definido anteriormente se controla desde la función `parse_Sql_To_Json.`

La llamada a esta función recibe como parámetros una cadena de texto que contiene la sentencia SQL de la consulta que vamos a transformar al álgebra relacional y una lista que contiene las sentencias de creación de las tablas que aparecen en la consulta previamente transformadas a formato JSON con la función anteriormente mencionada, creada para ello.

Esta función nos devuelve un diccionario que contiene la expresión en álgebra relacional correspondiente a la sentencia SQL que deseamos parsear.

El flujo de ejecución marcado es:

- Obtención de un diccionario con la sentencia SQL en formato JSON tras la llamada a la función `parse` de la biblioteca mo-slq-parsing.
- Chequeo de las operaciones soportadas mediante una función auxiliar.
- Realización del renombramiento sobre el diccionario previo mediante una función auxiliar.
- Transformación del diccionario previo renombrado con la consulta SQL a su expresión en álgebra relacional mediante una función auxiliar.

Todo este flujo de llamadas a otras funciones se encuentra bajo un bloque `try` para controlar posibles excepciones que se produzcan en las llamadas.

Capítulo 6. Reducción de expresiones AR mediante reglas y comparación de equivalencia

Es en este segundo componente del proyecto, junto a las expresiones SQL transformadas en diccionarios, donde se debe simplificar al máximo la expresión mediante la aplicación de reglas de equivalencia del álgebra relacional. Una vez llegado al punto en el que una expresión no se puede simplificar más, podemos decir que el diccionario está en su versión más simple y podemos comparar las expresiones para ver si son equivalentes.

El primer paso fue decidir qué reglas se han de aplicar y de qué forma hacerlo. Se realizó inicialmente una recolección y evaluación de las reglas [11, 12, 13], seleccionando aquellas aplicables para nuestro propósito y, a su vez, seleccionando la dirección en las que se deberían aplicar dichas reglas. Dicha tarea es necesaria para evitar que el algoritmo no se ejecute aplicando y revirtiendo la expresión usando la misma igualdad y se quede en un estado de bucle infinito.

Una vez finalizada la búsqueda de las reglas del álgebra relacional, pudimos seleccionar las siguientes 11 reglas, y les dimos un sentido de traducción, que a su vez indica la forma en la que aplicaremos las reglas:

Reglas de simplificación de álgebra relacional

Regla 1. Selecciones conjuntivas

Las operaciones de selección conjuntivas se pueden deconstruir en una única secuencia de selecciones individuales. Decidimos aplicar esta regla en la siguiente dirección para simplificar el diccionario resultante.

$$\sigma_{\theta 1}(\sigma_{\theta 2}(E)) \rightarrow \sigma_{\theta 1 \wedge \theta 2}(E)$$

Regla 2. Selecciones conmutativas

Las operaciones de selección son conmutativas. Decidimos establecer un estándar, en el cual, al aplicar esta regla, las selecciones se ordenan de manera alfabética de acuerdo a su condición. En el siguiente ejemplo de aplicación de la regla se considera $\theta 2 < \theta 1$.

$$\sigma_{\theta 1}(\sigma_{\theta 2}(E)) \rightarrow \sigma_{\theta 2}(\sigma_{\theta 1}(E))$$

Regla 3. Cascada de proyecciones

Sólo la última operación en una secuencia de proyecciones es la necesaria, por lo que las demás pueden ser eliminadas. Con esta regla simplificamos el diccionario resultante omitiendo las selecciones no necesarias.

$$\pi_{L1}\,(\pi_{L2}. \,.\,.(\pi_{Ln}\,(E)).\,.\,.) \rightarrow \pi_{L1}\,(E)$$

Regla 4. La proyección y la selección son conmutativas

Las operaciones de proyección y selección son conmutativas. Analizando los posibles resultados de aplicar esta regla en ambos sentidos, finalmente nos decantamos por organizar el diccionario resultante priorizando la selección, ya que el resultado es más simple y sencillo de analizar.

$$\pi_{A1,\,A2,\,\ldots\,,\,An}(\sigma_c\,(R)) \rightarrow \sigma_c\,(\pi_{A1,\,A2,\,\ldots\,,\,An}(R))$$

Regla 5. La selección de un producto cartesiano es igual a una reunión

En esta regla se observan dos casos, el primero es que se realice la selección de un producto cartesiano, y el otro es aquel donde se realice la selección de una reunión de tablas con condición. En ambos casos, decidimos aplicar la regla y crear una reunión con sus respectivos datos.

Regla 5A

$$\sigma_{\theta}(E_1 \times E_2) \rightarrow E_1 \bowtie_{\theta} E_2$$

Regla 5B

$$\sigma_{\theta 1}(E_1 \bowtie \sigma_{\theta 2} E_2) \rightarrow E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$$

Regla 6. Los reuniones naturales son conmutativas

Al igual que en reglas anteriores, al ser conmutativas, debemos de establecer un estándar de ordenación, y para mantener la concordancia en todo el algoritmo, decidimos ordenarlo, una vez más, de manera alfabética sobre la relación. En el siguiente ejemplo se considera $E_2 < E_1$.

$$E_1 \bowtie E_2 \rightarrow E_2 \bowtie E_1$$

Regla 7. Las reuniones son conmutativas

Como en la regla anterior, los theta join también son conmutativos, por lo que los ordenamos, a su vez, alfabéticamente. En el siguiente ejemplo se considera $E_2 < E_1$.

$$E_1 \bowtie_{\theta} E_2 \rightarrow E_2 \bowtie_{\theta} E_1$$

Regla 8. Los reuniones son asociativas

Las operaciones de las reuniones y los theta join son asociativos, por lo que en una cadena de estas operaciones, decidimos aplicar esta regla de forma que la agrupación de estos ocurra lo más a la ‘derecha’ posible, lo que en nuestros diccionarios da lugar a que sea más simple, y la parte más compleja se encuentre en el último nivel.

Reunión natural

$$(E_1 \bowtie E_2) \bowtie E_3 \rightarrow E_1 \bowtie (E_2 \bowtie E_3)$$

Theta Join

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 \rightarrow E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

Regla 9. Selección de reuniones

Ésta regla se puede aplicar de dos maneras:

Regla 9A

La forma sencilla, en la cual existe una reunión de una selección junto a otra tabla. En este caso, para simplificar el resultado, decidimos organizar el diccionario de manera que el sub-diccionario de la selección aparezca al final, y que quede de manera organizada.

$$(\sigma_{\theta 0}(E_1)) \bowtie_{\theta 0} E_2 \rightarrow \sigma_{\theta 0}(E_1 \bowtie_{\theta 0} E_2)$$

Regla 9B

Por otro lado, nos encontramos con una reunión de dos selecciones. No sólo eso, sino que para poder aplicarse esta regla, los atributos que aparecen en sendas selecciones deben de corresponder únicamente a la tabla que se hace referencia. Para la simplificación de esta regla, decidimos, al contrario que en la regla 9A, mantener de forma principal la selección, y utilizar como sub-diccionario la reunión.

$$(\sigma_{\theta 1}(E_1)) \bowtie_{\theta 0} (\sigma_{\theta 2}(E_2)) \rightarrow \sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_{\theta} E_2)$$

Regla 10. Selección de theta joins

Para la creación de esta regla existen dos variantes que cuentan a su vez con varios condicionantes necesarios para poder aplicarla.

Regla 10A

Decidimos aplicar esta regla de forma que el resultado final sea una selección de una reunión. Por lo tanto, es necesaria la ordenación de los diccionarios a la inversa y, a su vez, la condición que aparece en el join ha de pertenecer únicamente a una de las tablas.

$$(\sigma_{\theta 0}(E_1)) \bowtie_{\theta} E_2 \rightarrow \sigma_{\theta 0}(E_1 \bowtie_{\theta} E_2)$$

Regla 10B

Por otro lado, si nos encontramos con la misma disposición de los diccionarios a la regla anterior, pero esta vez existen dos condiciones en la reunión, cada una con un atributo perteneciente a una tabla distinta y sólo a esa tabla, podemos reordenar los diccionarios dejando como diccionario inicial la selección. Esta forma de aplicar la regla nos permite obtener un diccionario más sencillo y simplificado.

$$(\sigma_{\theta 1}(E_1)) \bowtie_{\theta} (\sigma_{\theta 2}(E_2)) \rightarrow \sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_{\theta} E_2)$$

Regla 11. La proyección se distribuye en la reunión

Cuando existe una reunión con dos expresiones, las cuáles ambas tienen una proyección, se puede aplicar esta regla. Al igual que en el método anterior, existen dos maneras de aplicar esta regla:

Regla 11A

Existe una reunión de dos selecciones, por lo que se pueden unir las dos proyecciones y crear una reunión más simple. Sin embargo, se deben de cumplir unas características especiales. L_1 es un atributo que sólo está en la tabla E_1 y L_2 es un atributo que sólo existe en la tabla E_2.

$$(\pi L_1(E_1)) \bowtie_{\theta} (\pi L_2(E2)) \rightarrow \pi_{L1 \wedge L2}(E_1 \bowtie_{\theta} E_2)$$

Regla 11B

Al igual que en la regla 11A existe una reunión con dos proyecciones, sin embargo estas proyecciones tienen más atributos. A su vez, existe una proyección externa que envuelve dicha reunión. La característica de éste caso es que L_1 y L_3 son atributos que sólo pertenecen a E_1, y L_2 y L_4 son atributos que sólo pertenecen a E_2. Si L_1 es distinto de L_3, y a su vez L_2 es distinto de L_4, podemos obviar los atributos L_3 y L_4. De ésta manera nos queda un diccionario más sencillo con el que trabajar.

$$\pi_{L1 \wedge L2}((\pi_{L1 \wedge L3}(E_1)) \bowtie_{\theta} (\pi_{L2 \wedge L4}(E_2))) \rightarrow \pi_{L1 \wedge L2}(E_1 \bowtie_{\theta} E_2)$$

Cabe destacar las funciones de ordenación creadas que fueron añadidas una vez se decidió implementar la posibilidad de los renombramientos. Muchas de nuestras reglas realizan ordenaciones alfabéticas de objetos, los que en un primer momento se trataban de cadenas que se podrían ordenar de forma sencilla. Al incluir los diccionarios de renombramiento al proyecto, estas ordenaciones no se podían realizar, por lo que creamos una función llamada `compare_values`, ya que la misma, recibiendo dos diccionarios, selecciona cuál debe de ir primero y cuál segundo. La ordenación de dos objetos sólo es posible si se trata de dos objetos del mismo tipo, pero los diccionarios no pueden ser ordenados. Con esta función lo que conseguimos es, según los valores incluidos en los diccionarios, poder decidir cuál va primero [14]. Si los datos son del mismo tipo, ordenamos esos datos, pero si no es así, seleccionamos el tipo de esos datos, y ordenamos los tipos, lo que da lugar a un estándar de ordenación que utilizamos en todas las reglas, independientemente de los datos internos.

Aplicación de las reglas

Una vez contamos con la implementación de todas las reglas, debemos de decidir la manera en la que vamos a aplicarlas a los diccionarios recibidos. Para ello decidimos realizar una batería de reglas, a las que, según las operaciones que aparezcan en el diccionario inicial, se deberán aplicar unos u otros métodos. Esto lleva consigo un grave problema debido a que no sólo se debe de aplicar una regla, sino que se deben de aplicar todas las posibles.

Lo expuesto deriva en la decisión de la creación de la función `applyRules`, la cual recibe un diccionario al que se le han de aplicar las reglas, las tablas necesarias que se han transformado a un diccionario en la fase anterior, y un booleano, quien marca si hemos llegado al punto final de dicho diccionario. Esta última variable es necesaria porque se trata de una función recursiva.

Inicialmente se planteó la realización de una función no recursiva a la que accediéramos en todos los niveles del diccionario. Aunque se presentaba como una buena opción inicial, no resultaba suficiente para todo lo que nuestra batería debía de realizar. El problema se encuentra en que algunas de las reglas, una vez

aplicadas, cambian la estructura del diccionario, lo que puede dar lugar a que otras reglas que antes no se debían de aplicar, ahora sí fueran necesario su uso. Por todo ello, se decidió implementar una función recursiva que empezara a aplicar las reglas únicamente si nos encontramos en el diccionario más profundo. De no ser este el caso, se realizará otra llamada a `applyRules` con los mismos parámetros iniciales, salvo el diccionario, que será el sub-diccionario del actual.

Debido a la forma en que se han de aplicar las reglas, es necesario tener una variable que marque la profundidad a la que nos encontramos en el diccionario, ya que sólo se empezará a aplicar las reglas una vez hemos llegado al punto más bajo. Desde aquí, se ha de ir ascendiendo por el diccionario, y a su vez, en los distintos niveles, también se podrán aplicar las reglas.

La decisión de qué reglas se han de aplicar depende del tipo de diccionario. Anteriormente se ha expuesto cómo se pueden aplicar las reglas, y se puede apreciar que las reglas que sirven para las operaciones de selección (‘sigma’) no sirven para las reuniones (“join”) y viceversa. Por ello, esta función es la que comprueba qué reglas se han de aplicar, y dado el supuesto de que se pueden aplicar más de una, en qué orden. El orden que se sigue para la aplicación de las reglas es el que nosotros hemos impuesto analizando las distintas salidas y buscando la manera más eficiente.

Consideremos el siguiente ejemplo, la siguiente consulta es traducida al álgebra relacional y da lugar al respectivo diccionario:

```
SELECT e.nombre
FROM Trabajadores t, Empresas e
WHERE e.empleados = 14 and t.oficio = 'doctor'
```

→

```
{'type': 'pi',
 'proj': ['Empresas1.nombre'],
 'rel': {'type' : 'sigma',
         'cond': {'type': 'and',
                  'values': [{'type': 'eq',
                              'values': ['Empresas1.empleados', 14]
                             },
                             {'type': 'eq',
                                    'values':['Trabajadores1.oficio',
                                              'doctor']
                             }
                            ]
                 },
         'rel': {'type':  'pro',
                  'lrel': {'type': 'rel',
                           'table': {'type': 'rho',
                                     'ren': ['Trabajadores',
                                             'Trabajadores1']
                                    }
                           },
                  'rrel': {'type': 'rel',
                           'table': {'type': 'rho',
                                     'ren': ['Empresas',
                                             'Empresas1']
                                    }
                           }
                 }
          }
}
```

Lo primero que hay que destacar es el cambio que se ha realizado a los renombramientos de las tablas. En la consulta podemos ver como los renombramientos son “e” y “t” de las tablas Empresas y Trabajadores respectivamente. Sin embargo, el diccionario resultante tiene como renombramientos de esas tablas “Empresas1” y “Trabajadores1”. Este paso se

realiza en el primer componente del programa, lo que facilita mucho las comprobaciones de diccionarios.

Una vez tenemos la consulta transformada en un diccionario, se ha de empezar a aplicar reglas. El orden que llevaría la aplicación de las reglas y sus respectivas salidas es el siguiente:

Regla 6: Las reuniones naturales son conmutativas →

```
{'type': 'pi',
 'proj': ['Empresas1.nombre'],
 'rel': {'type' : 'sigma',
         'cond': {'type': 'and',
                 'values': [{'type': 'eq',
                             'values': ['Empresas1.empleados', 14]
                            },
                            {'type': 'eq',
                                   'values':['Trabajadores1.oficio',
                                             'doctor']
                            }
                           ]
                 },
         'rel': {'type':  'pro',
                 'lrel': {'type': 'rel',
                          'table': {'type': 'rho',
                                    'ren': ['Empresas',
                                            'Empresas1']
                                   }
                         },
                 'rrel': {'type': 'rel',
                          'table': {'type': 'rho',
                                    'ren': ['Trabajadores',
                                            'Trabajadores1']
                                   }
                          }
                 }
          }
}
```

Regla 5A: La selección de un producto cartesiano es igual a una reunión →

```
{'type': 'pi',
 'proj': ['Empresas1.nombre'],
 'rel': {'type' : 'join',
         'cond': {'type': 'and',
                  'values': [{'type': 'eq',
                              'values': [14,'Empresas1.empleados']
                             },
                             {'type': 'eq',
                                    'values':['Trabajadores1.oficio',
                                              'doctor']
                             }
                            ]
                 },
         'lrel': {'type': 'rel',
                  'table': {'type': 'rho',
                            'ren': ['Empresas', 'Empresas1']
                           }
                 },
         'rrel': {'type': 'rel',
                  'table': {'type': 'rho',
                            'ren': ['Trabajadores', 'Trabajadores1']
                           }
                 }
         }
}
```

Ese resultaría el diccionario final, ya que no se pueden aplicar más reglas. Como se puede observar, se aplican las reglas inicialmente en los sub-diccionarios que se encuentran en la parte más ‘baja’ de la expresión. Se aplica primero la regla 6 en el sub-diccionario con ‘type’ = ‘pro’, que es el más bajo. Como no se pueden aplicar más reglas en ese diccionario, sube un nivel, y entonces se aplica la regla 5A en el sub-diccionario con ‘type’ = ‘sigma’. De esa manera se aplican las reglas en todos los diccionarios y sub-diccionarios generados.

Comprobación de equivalencia considerando renombramiento

Una de las partes más complejas de la equivalencia de consultas viene dada por el renombramiento de las tablas que se utilizan. En la fase anterior, se concretó un estándar, por el que se redefinen todos los renombramientos dados inicialmente por otros que nosotros otorgamos. Aunque esto soluciona algunos de los problemas, existen casos en que estas sustituciones no son las únicas necesarias.

Si en una sentencia aparece la misma tabla más de una vez, tendrá distintos alias. Y esto es precisamente, debido a que para saber si dos consultas son equivalentes, sus diccionarios resultantes deben de ser exactamente iguales. Un ejemplo de sentencias equivalentes salvo su renombramiento podría ser:

```
SELECT Club1.nombre, Club2.nombre
FROM Club Club1, Club Club2
```

```
SELECT Club2.nombre, Club1.nombre
FROM Club Club2, Club Club1
```

Incluyendo la comprobación de equivalencia considerando el renombramiento, este segundo componente de reducción mediante reglas quedaría con la siguiente estructura general:

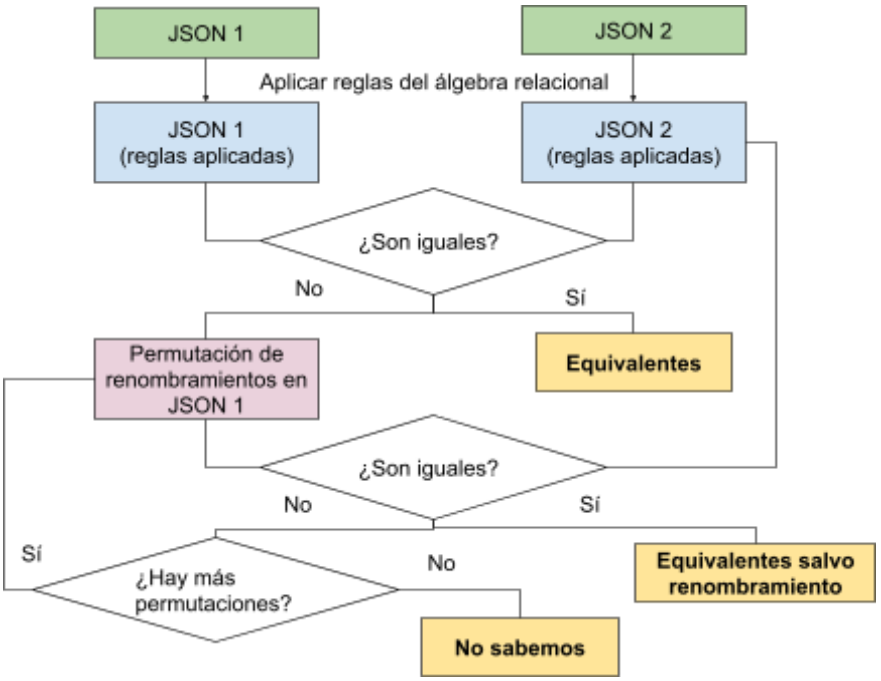


Figura 6.1. Mapa general del módulo de reducción a partir de reglas.

En este ejemplo podemos apreciar de forma sencilla cómo dos sentencias son equivalentes, sin embargo, por la forma en la que se comparan dos diccionarios mediante igualdad estricta, como respuesta obtenemos lo contrario.

Con el fin de dar solución a este problema hemos decidido permutar todos los renombramientos que aparecen en una de las consultas, e ir comprobando todas las posibles combinaciones de estos renombramientos. En el caso anterior existen únicamente esas dos permutaciones, pero todo ello varía según el número de tablas que se repiten y, a su vez, las veces que estas se repiten (si una tabla apareciera tres veces renombrada de manera distinta, las posibles combinaciones serían 6). Las funciones que utilizamos para realizar estas acciones son `permuteRenames`, `getRenames`, `groupRenames`, `getPermutations` y `recorrerDiccionario`.

En primer lugar, se ejecuta la función `permuteRenames`, a la cual se le pasa por parámetros los dos diccionarios de las consultas; uno de ellos se modificará y el otro sirve para comprobar la equivalencia de las posibles combinaciones de renombramientos.

En segundo lugar y dentro de ésta función, se hará llamada a `getRenames`, la cuál es una función que pasándole un diccionario, cuando se haya ejecutado, nos devolverá una lista con los renombramientos que se han encontrado.

```
SELECT Personas1.nombre, Personas2.nombre
FROM Personas Personas1, Personas Personas2, Club Club1, Club Club2
WHERE Personas1.club = Club1.nombre AND
      Personas2.equipo = Club2.nombre
```

Utilizando esta sentencia como ejemplo, la función `getRenames` nos devolverá una lista con los siguientes valores [Personas1, Club1, Personas2, Club2].

Posteriormente se hace llamada a la función `groupRenames`, que recibe los renombramientos obtenidos de la función anterior, y una lista de tablas, que inicialmente está vacía. Esta función, como su nombre indica, devuelve una lista de los renombramientos agrupados por la tabla a la que hacen referencia. A su vez, rellena la lista que obtuvo como parámetro con las listas a las que se les hace más de un renombramiento. Fijándonos en el ejemplo anterior, nos devolvería la lista de tablas que se renombran ([Personas, Club]) y agrupará los renombramientos según

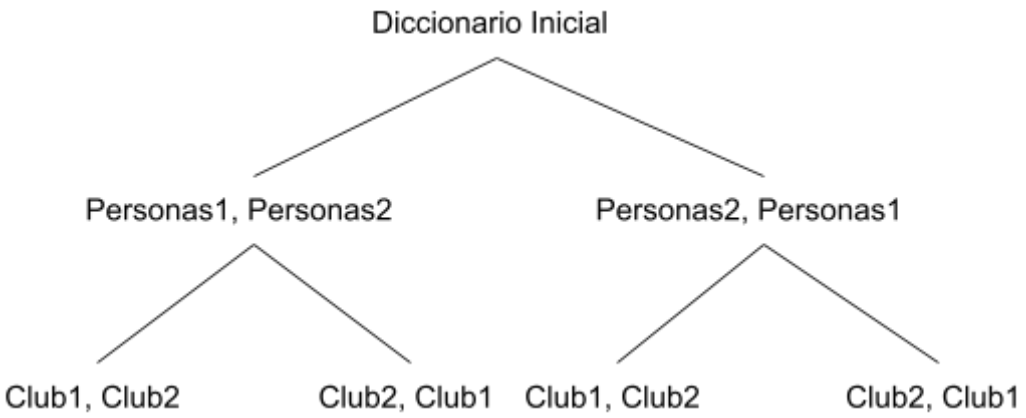
sus tablas ([[Personas1, Personas2], [Club1, Club2]]). De esta manera conseguimos los renombramientos y las tablas a las que hacen referencia en dos listas diferentes.

Una vez obtenemos todos estos datos, se ejecuta `getPermutations`, que utiliza la biblioteca Python itertools (https://docs.python.org/3/library/itertools.html), y concretamente la función `permutations`, la cual nos devuelve todas las permutaciones de una lista. Esta función se usa para todos los grupos de renombramientos, y así obtenemos todas las posibles combinaciones que pueden tener lugar. Siguiendo con el ejemplo anterior, con la lista [[Personas1, Personas2], [Club1, Club2]], nos devolvería lo siguiente:

[[(Personas1, Personas2), (Personas2, Personas1)], [(Club1, Club2), (Club2, Club1)]]

Con esto ya tenemos todos los datos necesarios para poder empezar a hacer los cambios de renombramientos. La encargada de este paso es la función `recorrerDiccionario`, la cual se encarga de, como su nombre indica, recorrer todo el diccionario en su totalidad e ir realizando los pertinentes cambios a los renombramientos. Se deben de hacer todas las posibles combinaciones de renombramiento, por lo que se trata de una función recursiva, lo que implica que no parará de renombrar hasta que se haya encontrado un diccionario equivalente, o hasta que ya no haya más posibles renombramientos.

Todas las posibles combinaciones:



Por tanto, una vez se ha añadido la parte de los renombramientos, ésta parte de simplificación mediante reglas tiene la siguiente estructura general:

Veredicto final

La ejecución del programa en su totalidad puede conducir a tres resultados distintos:

- Equivalentes: las consultas que se han insertado son equivalentes una vez se han transformado en diccionarios, y a estos se les han aplicado todas las reglas de álgebra relacional posibles.
- Equivalentes salvo renombramiento: las consultas insertadas no son equivalentes únicamente cuando se han aplicado las reglas, sino que se han debido de permutar sus renombramientos para que sean equivalentes.
- No sabemos: los diccionarios resultantes de la aplicación de las reglas y las distintas combinaciones de los renombramientos no dan lugar a dos diccionarios idénticos, por lo que no podemos asegurar si dichas consultas son equivalentes. Por ejemplo, un caso que no soportaría y que nos indicaría que no sabemos si son equivalentes sería la comprobación de “edad = 25” y “edad = 25 + edad - edad * 1”. Se puede apreciar que son equivalentes, pero nuestro programa no lo detecta.

La ejecución del programa de principio a fin sigue el siguiente orden: Primero las consultas y las tablas se transforman en diccionarios que hemos de usar. Una vez se encuentran en forma de diccionario, a ambas consultas se les aplica todas las posibles reglas del álgebra relacional. Llegado al punto donde no se puede aplicar ninguna más, comprobamos si los resultados son idénticos. Si es así, las consultas son equivalentes. Si ese no es el caso, la ejecución continúa, y pasamos a la parte de la combinación de los renombramientos. Se van haciendo todas las posibles combinaciones de renombramiento, y en el caso de que una de esas combinaciones resulte en un diccionario idéntico al otro, podemos decir que las consultas son equivalentes salvo renombramiento. En el supuesto de que se hayan probado todas las posibles combinaciones, y aún así los diccionarios no sean idénticos, el resultado de la ejecución es no sabemos, ya que no podemos identificar si dichas consultas son equivalentes.

Capítulo 7. Validación del sistema

Durante la elaboración de este proyecto se han ido evaluando las diversas componentes para comprobar, por una parte, el correcto funcionamiento de cada una de ellas por separado, y por otra, el componente final que las unifica. Esta evaluación se ha realizado mediante test de unidad [15].

Adicionalmente, se ha sometido el código implementado a un analizador para encontrar posibles problemas respecto a vulnerabilidades o bugs entre otros.

Tests de unidad

Conforme íbamos avanzando en la implementación de cada una de las partes que componen este proyecto, fuimos realizando diversos tests de unidad sobre los pasos intermedios de cada módulo para comprobar que los resultados obtenidos fueran los esperados y, en caso contrario, corregir la implementación.

Así, respecto al primero de los módulos, la traducción de consulta SQL a álgebra relacional, se realizaron pruebas desde las traducciones de consultas más básicas como un simple SELECT FROM hasta las consultas más elaboradas dentro del ámbito del proyecto.

Como se ha expuesto anteriormente, la parte de los renombramientos ocasionó bastantes problemas que, gracias a los tests unitarios realizados en esta parte, pudieron solventarse de forma relativamente sencilla.

Una vez implementados los renombramientos, y realizados los tests correspondientes, como fue lo último en llevarse a cabo dentro de la traducción, provocó que tuviéramos que actualizar los resultados esperados de los tests realizados hasta el momento, al ser el último paso dentro de la traducción.
Por otra parte, en cuanto al módulo de reducción de las expresiones en álgebra relacional, conforme se iba llevando a cabo el diseño de cada una de las reglas se iban realizando una serie de tests de unidad.

Finalmente, tras la unificación de ambos módulos, se realizó también la tanda de pruebas generales de todo el proyecto. Es decir, la traducción, reducción y comprobación de equivalencia de dos consultas SQL.

Tipo de unittest	Cantidad de Test
Traducción SQL → AR	11
Traducción SQL → AR con producto cartesiano	8
Traducción SQL → AR con condiciones	11
Traducción SQL → AR con reuniones	8
Traducción SQL → AR con selecciones	4
Traducción SQL → AR excepciones	8
Traducción SQL → AR con renombramiento	6
Creación de tablas	3
Aplicación regla 1	5
Aplicación regla 2	8
Aplicación regla 3	2
Aplicación regla 4	2
Aplicación regla 5A	3
Aplicación regla 5B	3
Aplicación regla 6	1
Aplicación regla 7	1
Aplicación regla 8B	1
Aplicación regla 9A	1
Aplicación regla 9B	2
Aplicación regla 10A	1
Aplicación regla 10B	1
Aplicación regla 11A	1

Aplicación regla 11B	1
Tests de la función `applyRules`	5
Test generales	10

Algunos de los test que se necesitaban tenían diferentes variantes, mientras que otros eran bastante simples con un funcionamiento sencillo y fácil de comprobar, de ahí la variedad de cantidad de test. En total, entre los dos componentes más los test que se hicieron de todo el proyecto general contamos con 107 test de unidad.

Analizador SAST de código estático

Durante el desarrollo de este proyecto tenemos acceso a un analizador de código estático. Este analizador de código, llamado Bugscout [16], está enfocado principalmente a detectar vulnerabilidades en el código, aunque también es capaz de detectar *bugs* y *code smells*, también conocido como malas prácticas en el desarrollo de *software*. Es una aplicación potente que lleva funcionando desde 2010 y gran número de empresas de alto nivel colaboran con ella, como Yamaha o Sabesp.
Aunque los problemas de seguridad que se pueden presentar en la creación de nuestra biblioteca, por su naturaleza, son mínimos, realizamos un análisis de nuestro código para encontrar posibles *code smells* y *bugs*.

Este analizador realiza una calificación de los problemas encontrados en función de su criticidad, siendo los niveles de mayor a menor:

- Bloqueante
- Crítico
- Mayor
- Menor
- Info

En el informe final que devuelve la aplicación tras la ejecución del análisis, se detallan distintas situaciones en las cuales la detección de una vulnerabilidad, *code smell* o *bug* no se realiza correctamente. Se puede dar el caso de un falso positivo:

la aplicación detecta una de estos tres problemas cuando no existe o se puede dar el caso de que no detecte uno de los tres fallos y si se encuentre dentro del código.
Los resultados que se obtienen al realizar el análisis de nuestro código son que no se detecta ninguna vulnerabilidad ni *bug*, pero sí varios *code smells*.
La mayoría de estos *code smells* se pueden tratar como falsos positivos o en su defecto son de una criticidad baja, algunos de ellos son:

- “String literals should not be duplicated” (CRÍTICO)
  Este aparece durante la creación de test donde se utiliza el mismo literal en varias ocasiones para realizar consultas, por lo que, pese a estar catalogado como crítico y encontrarse en un banco de pruebas, no presenta un problema real a solucionar.

- “Sections of code should not be commented out” (MAYOR)
  Recomendaciones sobre el estándar al comentar funciones. No presenta un problema real a solucionar.

- “Function names should comply with a naming convention” (MAYOR)
  Recomendación para el nombre de funciones según el estándar de recomendaciones (name_function). No presenta un problema real a solucionar.

- “Local variable and function parameter names should comply with a naming convention” (MENOR)
  Recomendación para el nombre de variables según el estándar de recomendaciones (name_variable). No presenta un problema real a solucionar.

- “"SystemExit" should be re-raised” (CRÍTICO)
  SystemExit se genera cuando se llama a sys.exit(). Se espera que esta excepción se propague hasta que la aplicación se detenga. Es necesario capturarla y relanzar de nuevo dicha excepción. Este *code smell* sí que podría convertirse en un problema, lo tratamos y solucionamos siguiendo las recomendaciones.

Capítulo 8. Conclusiones y trabajo futuro

El objetivo principal de nuestro Trabajo de Fin de Grado era implementar un algoritmo que recibiera dos sentencias SQL, verificará que estas son equivalentes a una sentencia a comparar. A la vista de los resultados obtenidos y al trabajo desarrollado, se pueden sacar varias conclusiones.

El código del trabajo se puede acceder desde el siguiente enlace:
https://github.com/danimu03/TFG_equivalencia_SQL.git

Tras el desarrollo del proyecto se puede decir que el objetivo principal se ha cumplido con éxito, ya que se ha logrado que el algoritmo funcione correctamente acorde a nuestras expectativas.

Entrando en mayor nivel de detalle, los objetivos 1, 2 y 3 propuestos en la sección 1.2 se han alcanzado, ya que en ellos reside la funcionalidad básica del proyecto:

- La traducción de consultas SQL a álgebra relacional se ha conseguido mediante la implementación del módulo Sql_To_Json, en el cual, gracias a la biblioteca *mo-sql-parsing*, hemos podido trabajar de forma sencilla con las consultas y realizar la traducción.
- La reducción de las expresiones obtenidas en el primer módulo, mediante el módulo Rules_AR, se ha conseguido gracias a la aplicación de una serie de reglas de equivalencia de expresiones de álgebra relacional en un sentido definido.
- Finalmente, la unificación de ambos módulos para obtener el veredicto final.

A su vez, el objetivo 4 también se ha cumplido al haber implementado una batería de 107 test automáticos que, dados múltiples casos de prueba, verifica el correcto funcionamiento de los distintos módulos de forma separada, así como la unión de ambos.

Por su parte, el objetivo 5 también se ha conseguido al contemplar, en las opciones del veredicto, la equivalencia de dos consultas a excepción del renombramiento.

Por último, respecto al trabajo realizado, a pesar de haber intentado mantener el algoritmo lo más óptimo posible en lo referente a tiempos de respuesta, se deja abierta la posibilidad de mejorar este aspecto del proyecto de cara al futuro. Al igual que la implementación de nuevas funcionalidades como es la comprobación de otro tipo de sentencias a parte de las sentencias SQL conjuntivas.

Trabajo futuro

Durante el desarrollo de esta biblioteca han ido surgiendo algunas ideas que, como no entraban en el objetivo principal del proyecto, así como por falta de tiempo, no han sido implementadas. Entre ellas podemos destacar las siguientes:

- Comprobación de consultas no conjuntivas: aunque este proyecto se originó bajo la idea de la comprobación de equivalencia de consultas conjuntivas, durante su desarrollo nos ha surgido interés en cómo ampliar el trabajo que íbamos realizando para la incorporación de estas consultas que no contemplamos actualmente. Esta ampliación requeriría extender el formato de los diccionarios para que puedan soportar las nueves operaciones a implementar, así como una actualización de las reglas, como la inserción de aquellas necesarias para comprobar la equivalencia de expresiones que utilizan dichas operaciones.
- Mayor testing: aunque hemos realizado una batería de *tests* bastante amplia, tal y como profundizamos en la sección 7, en algunas ocasiones nos ha dado la sensación de que podrían no ser suficientes para poder contemplar todos los posibles casos que se pueden presentar. Por esta razón, a corto plazo, vemos conveniente realizar los *tests* que pueden faltar. Más a medio y largo plazo, en unión con el punto anterior, lo ideal sería realizar *tests* para comprobar el correcto funcionamiento de las nuevas funcionalidades añadidas.
- Analizar la eficiencia del algoritmo para conocer con detalle el tiempo de ejecución, de ésta manera podríamos optimizar aquellas funciones que sean más costosas y mejorar la eficiencia del programa.

- Estudiar la posibilidad de paralelizar el algoritmo y usar distintos hilos para procesar distintas consultas SQL simultáneamente, lo que mejoraría el tiempo de respuesta del algoritmo.

Capítulo 9. Conclusions and Future Work

The main objective of our Final Degree Project was to implement an algorithm that receives two SQL sentences and verifies that they are equivalent to a sentence to be compared. In view of the results obtained and the work developed, several conclusions can be drawn.
The code of the work can be accessed from the following link:
https://github.com/danimu03/TFG_equivalencia_SQL.git

After the development of the project, it can be said that the main objective has been successfully achieved, since the algorithm has been made to work correctly according to our expectations.

Going into greater detail, the objectives 1, 2 and 3 proposed in section 1.2 have been achieved, since the basic functionality of the project lies in them:

- The translation of SQL queries to relational algebra has been achieved by implementing the Sql_To_Json module, in which, thanks to the mo-sql-parsing library, we have been able to work easily with the queries and perform the translation.
- The reduction of the expressions obtained in the first module, by means of the Rules_AR module, has been achieved thanks to the application of a series of equivalence rules of relational algebra expressions in a defined sense.
- Finally, the unification of both modules to obtain the final verdict.

In turn, objective 4 has also been met by having implemented a battery of 91 automatic tests that, given multiple test cases, verifies the correct functioning of the different modules separately, as well as the union of both.

Objective 5 has also been achieved by considering, in the verdict options, the equivalence of two queries with the exception of renaming.

Finally, with regard to the work carried out, despite having tried to keep the algorithm as optimal as possible in terms of response times, the possibility of improving this aspect of the project in the future is left open. As well as the implementation of new functionalities such as the checking of other types of sentences apart from conjunctive SQL sentences.

Future work

During the development of this library, some ideas have arisen which, as they were not part of the main objective of the project, as well as due to lack of time, have not been implemented. Among them we can highlight the following:

- Checking non-conjunctive queries: although this project originated with the idea of checking the equivalence of conjunctive queries, during its development we have become interested in how to extend the work we were doing to incorporate these queries that we do not currently contemplate. This extension would require extending the format of the dictionaries so that they can support the nine operations to be implemented, as well as an update of the rules, such as the insertion of those necessary to check the equivalence of expressions that use these operations.
- More testing: although we have carried out a fairly extensive battery of tests, as we will go into more detail in section 7, on some occasions we have had the feeling that they might not be sufficient to be able to contemplate all the possible cases that may arise. For this reason, in the short term, we believe it is appropriate to carry out the tests that may be missing. More in the medium and long term, in conjunction with the previous point, the ideal would be to carry out tests to check the correct functioning of the new functionalities added.
- Analyze the efficiency of the algorithm to know in detail the execution time, in this way we could optimize those functions that are more costly and improve the efficiency of the programme.

- Study the possibility of parallelising the algorithm and using different threads to process different SQL queries simultaneously, which would improve the algorithm's response time.

Capítulo 10. Contribuciones personales

Daniel Muñoz García

Al inicio del proyecto se realizó una primera reunión en la cual marcamos las ideas generales y la línea de trabajo que debíamos seguir. En este momento quedaron delimitadas las dos principales tareas a realizar a lo largo del proyecto, la traducción de las consultas en formato SQL a álgebra relacional y la aplicación de reglas de equivalencia para el cálculo. Esto derivó en una división de manera aleatoria de los integrantes del proyecto en dos grupos de trabajo, de esta forma pasé a trabajar sobre la tarea de traducción de consultas en formato SQL a álgebra relacional.

Una de las primeras tareas a realizar fue la de reforzar mis conocimientos de Python para poder llevar a cabo todo el desarrollo de la biblioteca sin dificultades, además fue en este momento cuando definimos que la estructura de datos que posteriormente se utilizaría sería la de JSON. Simultáneamente debíamos realizar una tarea de búsqueda de bibliotecas Python ya existentes, las cuales nos ayudarían en la traducción de cadenas de texto de consultas SQL a formato JSON, así que me centré en la biblioteca mo-sql-parsing y realicé distintas pruebas de concepto para valorar su idoneidad.

Después de encontrar una biblioteca Python que realizaba la transformación de SQL a JSON, era necesario establecer un formato JSON estándar para el álgebra relacional, del cual se haría uso por ambos equipos de trabajo del proyecto. Por ello comencé a realizar un primera versión de este formato genérico para todas las operaciones del álgebra relacional que podían aparecer dentro del rango establecido de consultas SQL de nuestro proyecto. Una vez terminado esta definición de formato fue compartida con el resto de integrantes del proyecto. Este formato propuesto quedó establecido como nuestro estándar para ambos grupos de trabajo tras la aprobación de todos los integrantes.

Establecimos el uso de un repositorio de Github para almacenar todo el trabajo realizado en el proyecto, el cual creé y compartí el acceso con el resto de integrantes del proyecto para que pudieran trabajar sobre él.

En este momento comenzamos con el desarrollo de las dos principales tareas definidas al inicio, mi trabajo comenzó a ser la implementación de varias funciones que llevaran a cabo el proceso de transformación de cadenas de texto con consultas SQL a álgebra relacional en el formato JSON establecido. Para ello fue necesario realizar numerosas pruebas con la biblioteca auxiliar de mo-sql-parsing, con el fin de entender de manera clara su funcionamiento y los resultados que producían, en particular fue necesario analizar en profundidad el formato JSON que se obtiene como salida tras la ejecución de sus funciones. La comprensión de dicho formato fue necesaria para establecer la manera en que debíamos realizar modificaciones sobre este, para llegar a nuestra expresión de álgebra relacional y desarrollar todas las funciones que llevasen a cabo este trabajo.

A lo largo del desarrollo de las funciones que realizaran el proceso de transformación SQL a AR, fueron surgiendo distintas necesidades que derivaron en la implementación de nuevos módulos auxiliares.

Para tratar el problema de los renombramientos fue necesario que desarrollara un módulo auxiliar con funciones específicas que trataran dicho problema, esta fue una de las tareas que más tiempo necesitó a lo largo de todo el proyecto, ya que como se ha descrito anteriormente, el problema del renombramiento en nuestra biblioteca trae como consecuencia numerosos casos distintos que hay que abordar para que todo funcione de manera correcta. Surgieron además, algunas dificultades a la hora de la implementación que derivaron en la necesidad de hacer otro estudio sobre nuevas versiones de la biblioteca mo-sql-parsing para poder solucionarlas.

Fue necesario también la creación de un módulo auxiliar para la traducción de sentencias de creación de tablas a formato JSON, en principio sólo eran necesarias para la tarea principal de creación y aplicación de reglas sobre expresiones de álgebra relacional, pero, debido a la solución propuesta para los problemas de los renombramientos, esta función pasó a ser necesaria para ambas partes del

proyecto. Por ello, realicé su implementación, la cual no fue tan complicada ni trajo tantos problemas como el módulo de renombramiento.

Por último, llevé a cabo un análisis de nuestro código mediante un SAST (pruebas de seguridad de aplicaciones estáticas), cuyo objetivo era la detección de vulnerabilidades, *bugs*, o *code smells*. Los resultados de este análisis fueron compartidos con todos los integrantes del proyecto para solucionar los problemas que consideramos necesarios.

Sergio Villanueva Agreda

Tras la primera reunión, en la que se marcó la idea general de este proyecto así como una especie de pasos a seguir, quedó bastante clara la división de tareas necesarias para la realización del proyecto. Una primera parte encargada de la traducción a álgebra relacional a partir de una consulta SQL; y una segunda, que se encargaba de reducir una expresión en álgebra relacional mediante la aplicación de una serie de reglas.. De manera aleatoria se decidió que yo participaría en la primera parte de las anteriormente comentadas, la traducción a álgebra relacional.

Lo primero de todo fue repasar algunos conceptos básicos relativos a SQL y álgebra relacional que estudié en los primeros años de la carrera. Además, también tuve que recuperar conocimientos de Python para poder realizar el desarrollo de la biblioteca sin muchas dificultades.

Después tuve que enfocarme en la búsqueda de alguna biblioteca que ayudase a *parsear* una consulta SQL y así facilitarnos el trabajo a realizar posteriormente. Se encontró la librería *mo-sql-parsing* y, tras realizar unas pruebas para ver si nos servía, decidimos utilizarla.

Al mismo tiempo, como habíamos acordado representar en Python las expresiones de álgebra relacional en formato JSON, estudié diversas posibles maneras de expresarlas en este formato. Finalmente, se acordó el formato que acabaríamos empleando ambas partes del proyecto.

En este momento fue necesario estudiar con mayor profundidad la librería *mo-sql-parsing* para entender cómo devuelve la consulta SQL analizada y así poder empezar a idear cómo realizar la implementación de la traducción a álgebra relacional, mediante diversas modificaciones sobre el JSON generado por la librería, y obtener el formato previamente establecido.

Después de que mi compañero implementase las primeras funciones me encargué de realizar los primeros *tests* de unidad y comprobar que la implementación realizada por él devolviera, en consultas algo más complejas los resultados

esperados. En esta ocasión, algunos de los *tests* resultaron erróneos, por lo que tuve que realizar pequeños ajustes en las funciones.

Mientras iba añadiendo más funciones, para terminar de implementar la traducción básica a álgebra relacional, fui realizando más *tests* de unidad de cada una de las traducciones que iba realizando y arreglando aquellas que en un principio parecían funcionar bien.

Cuando en una reunión se trató el tema de los renombramientos, y posible implementación de los mismos, tuve que implementar el módulo que se encargaba de los renombramientos. Mientras estaba con esta tarea me di cuenta de que hacía falta otro módulo más. Este módulo es el encargado de traducir las consultas de creación de tablas.

Por una parte, el módulo relativo a la creación de tablas no fue complicado de implementar y no llevó mucho tiempo.

En cambio, el módulo de renombramientos ha sido una de las tareas que más tiempo ha llevado en general en todo el proyecto. Además, el renombramiento, ocasionó, tras realizar unos pocos *tests* de unidad y solucionar algún fallo que encontré, el tener que actualizar todos los *tests* de unidad relacionados con la traducción realizados con anterioridad. Mientras realizaba estas actualizaciones me di cuenta de que alguna función del módulo de renombramientos no funcionaba del todo bien y procedí a su corrección.

Julio Martínez Sánchez

Partiendo de la primera reunión, se decidió la partición del proyecto en dos partes, una inicial que se encargaría de traducción de lenguaje SQL a álgebra relacional, y otra que se encargaría de la reducción de álgebra relacional mediante reglas. Cuando se hizo la repartición de tareas me convertí en integrante del desarrollo de ésta segunda parte. Lo primero que tuve que hacer es recordar e investigar sobre el álgebra relacional. Las primeras semanas se trataron de investigación, estudio y aprendizaje. No sólo lo concerniente al álgebra relacional, sino que también tuve que aprender a programar en Python, lenguaje con el que nunca antes había trabajado.

En las siguientes reuniones se decidió enfatizar nuestro trabajo en el desarrollo de la estructura que utilizamos como conexión entre las dos partes del proyecto. Se estipula utilizar diccionarios, y tuvimos que ponernos de acuerdo en su nomenclatura. Por otro lado, tuve que buscar todas las reglas del álgebra relacional que serían aplicables a nuestro proyecto.

Una vez seleccionada la base de cómo íbamos a trabajar con los diccionarios, tuve que documentar todas las reglas, el direccionamiento que le he dado a dichas reglas, y crear algún ejemplo para que quedara claro. Esta parte, aunque compleja en el estado inicial, fue fundamental para tener una base y unos referentes que utilizaré a lo largo de todo el desarrollo.

Una vez que teníamos todas las reglas que queríamos aplicar documentadas, empezó el proceso de programación. Al final recopilamos 10 reglas, algunas de ellas con sus sub-reglas particulares. Fue bastante complejo el inicio de la programación de estos métodos, no simplemente por no estar acostumbrado al entorno con el que estaba trabajando, sino que todavía la estructura decidida me resultaba difícil de procesar. Según iba avanzando el desarrollo del proyecto, iba interiorizando la programación en Python y cada vez me resultaba más sencillo el desarrollo de las reglas. Durante este periodo no sólo creé las reglas, sino que también hice tests de unidad para todas ellas que nos fueron muy útiles, ya que cualquier cambio futuro en alguna de las reglas podría suponer una salida no

esperada. Sin embargo, la creación y ejecución de estos tests nos permitía ver si algún cambio alteraba el resultado esperado.

Aún no habiendo terminado el desarrollo de todas las reglas, se decidió añadir a nuestro proyecto la operación de renombramiento en nuestras consultas. Ésto dio lugar a varios problemas. No únicamente nos alteraba la estructura de los diccionarios creados, sino que también tendríamos que tenerlo en cuenta a la hora de hacer las comprobaciones finales de equivalencia.

La sección del renombramiento ha sido la más compleja de toda ésta segunda parte. Además de estar trabajando en paralelo con la creación e implementación de las reglas, algunas de ellas muy complejas, teníamos que decidir cómo íbamos a llevar un seguimiento de los renombramientos y cómo se tendrían que tener en cuenta al final. Se decidió un estándar de nombres para los renombramientos, lo que nos simplifica en parte nuestra tarea.

Uno de los grandes problemas que tuve durante el proyecto era la ordenación de objetos. Mientras que la ordenación de objetos si son del mismo tipo es muy sencilla, cuando los tipos varían se complica bastante. Tras bastante investigación, dudas y reflexión de cómo hacerlo, al final pude crear una función, que junto a la función `sorted` de Python, permitía la ordenación de objetos. Esta función lo que hace es comparar los tipos de objetos, si son iguales los ordena alfanuméricamente. De no ser este el caso, lo que hace es la ordenación de los objetos según el orden alfabético de sus tipos.

Una vez solucionado este problema, se pudo seguir con la parte de las reglas y el renombramiento. Al final, los renombramientos se irían sustituyendo en uno de los diccionarios creados usando permutaciones. Para ello utilicé la librería `itertools`, que me devolvía permutaciones de una lista, y junto a diversas funciones recursivas y recorridos de diccionarios se consiguió obtener los renombramientos.

Una vez terminadas las reglas, decidimos hacer una batería de métodos, que con un diccionario inicial fuera aplicando todas las reglas posibles. Esta parte del proyecto también es bastante compleja, ya que hay que tener en cuenta que todas las reglas deben de ser posiblemente aplicables, considerar todas las opciones, y no sólo eso, sino que también tener en cuenta el resultado de la aplicación de otras

reglas. Algunos de estos métodos, una vez aplicados, dan lugar a un diccionario en el que se puede aplicar una regla que antes no se aplicaba. Por ello la función es recursiva, y una vez se ha acabado, se tiene una variable con la que se indica si el diccionario ha sufrido un cambio de estructura. De ser este el caso, el diccionario se vuelve a pasar por la batería de pruebas.

Teniendo todas las reglas aplicadas y teniendo los posibles renombramientos, hay que hacer todas las posibles combinaciones de estos si los diccionarios no son equivalentes en este estado con las reglas aplicadas. Para ello creé una función que recorriera todo el diccionario, y teniendo en cuenta el renombramiento que tiene originalmente, y la tabla a la que pertenece, se va sustituyendo por el renombramiento pertinente. Ésta función se ejecuta indefinidamente hasta que no existan más posibles combinaciones, o a que alguna combinación fuese equivalente al otro diccionario obtenido.

Por último hice pruebas finales de la ejecución de nuestro código, tanto pruebas para ver su correcto funcionamiento como para detectar posibles bugs. Esto dio lugar al descubrimiento de varios errores, algunos de ellos críticos, y otros que mostraban una salida no esperada, los cuales finalmente acabé por corregir.

Marcos Gómez Martín

Al inicio del proyecto, tuvo lugar una primera reunión en la que se me explicó en qué consistía este, además de las fases en las que se dividía y el reparto de tareas entre los miembros del grupo. Este reparto consistía en la partición del trabajo en dos equipos, uno encargado de las traducciones de SQL a formato JSON, y otro responsabilizado de la aplicación de las reglas del álgebra relacional.

Hasta la siguiente reunión, se me indicó familiarizarme con las herramientas y recursos que iba a utilizar, así como con los lenguajes de programación y la teoría de base de datos. Entre ellos se encontraban el sistema de control de versiones de GitHub, la biblioteca *mo-sql-parsing*, el lenguaje de programación Python, el entorno de programación PyCharm en el que se desarrollaría la aplicación, y la teoría de las consultas SQL conjuntivas.

En la siguiente reunión, una vez adquiridos los conocimientos mencionados, se especificó la línea de trabajo que seguiría el equipo para la implementación de la aplicación. Empecé con la documentación de las reglas del álgebra relacional, investigando en libros de bases de datos (enlaces) cuáles habría que implementar para el correcto desarrollo de la aplicación. Hice un documento en el que se recogían todas estas reglas, especificando el sentido en el que se aplicarían, como se puede ver en el capítulo 6, e incluyendo ejemplos que más tarde se usarían para desarrollar los tests de unidad.

Después de esto y llegada la siguiente reunión se me comentó que empezaría con la implementación de las reglas, el objetivo era realizar a lo sumo 1 o 2 reglas con sus respectivos tests, ya que al comienzo siempre aparecen dificultades y es “lo más duro”. Conseguí cumplir con el objetivo establecido y a lo largo de las próximas semanas seguí implementando el resto de reglas junto a sus test de unidad.

Llegados a cierto punto en el que las traducciones ya estaban declaradas y la mayor parte de las reglas implementadas, era momento de unir las dos componentes en las que cada equipo había estado trabajando durante los últimos meses. A priori no parece muy complicado, ya que sólo habría que hacer un estudio que en función de la entrada, se aplican ciertas reglas, y en algunos casos múltiples reglas. Sin embargo, me topé con un problema, cuando una regla modificaba un diccionario y

quería comprobar si se volvía a aplicar esa u otra regla, al ser una función recursiva, perdía el diccionario modificado. Ya que era una de mis primeras veces programando en Python, el paso de atributos con referencia no los tenía muy claro, además de que estaba acostumbrado a otros lenguajes de programación como C++, en el que esto es muy simple. Después de intentar varias soluciones, encontré una función, `deepcopy` [17], la cual construye un nuevo objeto y luego, recursivamente, inserta copias en él de los objetos encontrados en el original, lo que permite no modificar el diccionario original y, a la vez, almacenar los cambios realizados sobre este. Una vez resuelto este problema, la unión de las dos componentes de la aplicación ya funcionaba, y se dio paso al siguiente objetivo, los renombramientos.

Con el inicio de esta funcionalidad, me di cuenta de que la estructura de los diccionarios con la que había estado trabajando ya no me servía, por lo que la tuve que modificar para que fueran capaces de gestionarlos. Además, me percaté de que la ordenación de los objetos que hasta ahora se había gestionado de una forma sencilla, no estaba siendo realizada correctamente, ya que sólo ordenaba objetos del mismo tipo. Esto pasó a ser un nuevo obstáculo, que tras varios intentos, y mucha investigación, se solucionó con la función `sort` de Python, que permitía la ordenación de objetos de distintos tipos.

Una vez solucionados estos problemas, volví a centrarme en la implementación de los renombramientos, declarando un estándar de nombres para estos. Con el uso de la librería `itertools` podría realizar todas las posibles permutaciones sobre una lista que, junto al uso de la recursividad, daría lugar a una función que realizara los renombramientos de una tabla.

Terminadas todas las componentes, sólo hacía falta recorrer el diccionario y, para el renombramiento elegido, buscar por todas las posibles permutaciones hasta encontrar su equivalente o que no haya más.

Llegados a este punto con casi toda la funcionalidad implementada, me dediqué a terminar las reglas que se habían quedado sin desarrollar, a generar más tests de unidad para probar el funcionamiento de la aplicación, y a comprobar que todo se ejecutara correctamente. Se descubrieron algunos errores, pero se corrigieron.

Bibliografía

[1] Burgoa Muñoz, I., Huertas Smolis, T., Ibáñez Padial, D., & Ruiz Quintana, I. (2021). Juez para el aprendizaje de bases de datos. https://eprints.ucm.es/id/eprint/67031/

[2] *Relational Algebraic Equivalence Transformation Rules* https://www.postgresql.org/message-id/attachment/32513/EquivalenceRules.pdf

[3] *Getting Started Step-By-Step* https://json-schema.org/learn/getting-started-step-by-step

[4] *mo-sql-parsing. Parse SQL into JSON.* https://github.com/klahnakoski/mo-sql-parsing

[5] Coder House. (2021). *¿Qué es SQL y para qué sirve?* https://www.coderhouse.es/blog/que-es-sql

[6] Oppel, A., & Sheldon, R. (2010). *Fundamentos de SQL*. McGraw-Hill. pedrobeltrancanessa-biblioteca.weebly.com/uploads/1/2/4/0/12405072/fundamentos_de_sql_3edi_oppel.pdf

[7] *SELECT Oracle* https://docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/SELECT.html

[8] *Álgebra relacional*. El álgebra relacional. http://fcays.ens.uabc.mx/anterior/BD/AlgebraRelacional.pdf

[9] *Python 3.9 Tutorial*. The Python Tutorial. https://docs.python.org/3.9/tutorial/

[10] *What is Python? Executive Summary*. Python.org. https://www.python.org/doc/essays/blurb/

[11] Elmasri, R., & Navathe, S. (2016). *Fundamentals of Database Systems*. Pearson. https://iran-lms.com/images/images/Books/PDF/Fundamentals-of-Database-Systems-Pearson-2015-Ramez-Elmasri-Shamkant-B.-Navathe.pdf

[12] Hurson, A. R. *Transformation of Relational Expressions*. 14.3 Transformation of Relational Expressions. https://hurson.weebly.com/uploads/4/9/2/2/49225097/reading3.optimization.pdf

[13] *Relational Algebraic Equivalence Transformation Rules*. PostgreSQL. https://www.postgresql.org/message-id/attachment/32513/EquivalenceRules.pdf

[14] Dalke, A. *Sorting HOW TO — Python 3.10.4 documentation*. Python Docs. https://docs.python.org/3.9/howto/sorting.html#sorting-how-to

[15] *unittest — Unit testing framework — Python 3.10.4 documentation*. Python Docs. https://docs.python.org/3.9/library/unittest.html

[16] *Detección de vulnerabilidad y análisis de calidad de código.* BugScout. https://bugscout.io/es

[17] *copy — Shallow and deep copy operations* https://docs.python.org/3/library/copy.html