Generación automática de tablas para probar consultas SQL
Automatic generation of tables for testing SQL queries

Trabajo de Fin de Grado
Curso 2021–2022

Autores
Maria de Lluc Bonet Seguí
Álvaro Plaza Sanz

Director
Enrique Martín Martín

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Generación automática de tablas para probar consultas SQL
Automatic generation of tables for testing SQL queries

Trabajo de Fin de Grado en Ingeniería Informática
Departamento de Sistemas Informáticos y Computación

Autores
Maria de Lluc Bonet Seguí
Álvaro Plaza Sanz

Director
Enrique Martín Martín

Convocatoria: *Junio 2022*

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Resumen

Desde hace tiempo, uno de los principales problemas de algunos jueces de aprendizaje como LearnSQL ha sido la introducción de casos de prueba de forma manual por parte de los profesores. Estos casos de prueba se utilizan de forma que se pueda verificar si las consultas introducidas por los estudiantes están bien construidas. Nuestro sistema de generación automática de tablas SQL pretende hacer frente a esta situación integrando una biblioteca al juez LearnSQL para que analice las consultas SQL y sus componentes para que de esta forma pueda generar datos que prueben estas consultas recibidas. El fin último de este trabajo es desarrollar dicha biblioteca para que posteriormente pueda ser incorporada al sistema de aprendizaje SQL.

Se puede acceder al código fuente de esta biblioteca a través del siguiente enlace al repositorio de Github:

`https://github.com/Generacion-tablas-SQL`

Palabras clave

SQL, juez de aprendizaje, biblioteca, Oracle Database, LearnSQL, Python, bases de datos

Abstract

For some time now, one of the main problems with some learning judges such as LearnSQL has been the manual introduction of test cases by teachers. This test cases are used so that it can be verified whether the queries entered by the students are well constructed. Our automatic SQL table generation system aims to address this issue by integrating a library into the LearnSQL judge to parse SQL queries and their components, so that it then generates data that tests these incoming queries. The ultimate goal of this work is to develop the mentioned library so that it can be later incorporated into the SQL learning system.

The source code of this system can be accessed through the following link to the Github repository:

`https://github.com/Generacion-tablas-SQL`

Keywords

SQL, learning judge, library, Oracle Database, LearnSQL, Python, Databases

Índice

Índice de figuras

Capítulo 1

Introducción

En este capítulo vamos a presentar la motivación para llevar acabo este proyecto, los objetivos que hemos establecido en su desarrollo y el plan de trabajo que hemos ido siguiendo.

1.1. Motivación

A día de hoy, en la Facultad de Informática de la Universidad Complutense de Madrid (UCM), es muy común el uso de jueces de aprendizaje en diferentes asignaturas de programación y algoritmia que proporcionan una retroalimentacion a los alumnos cuando realizan entregas prácticas de problemas. En concreto, se utilizan estos jueces de programación para proporcionar al alumnado una corrección de sus soluciones de manera automática en asignaturas como Fundamentos de la Programación, Fundamentos de la Algoritmia, Estructura de Datos o Bases de Datos. En esta última asignatura de Bases de Datos, se ha usado este curso 2021/2022 un juez de programación denominado LearnSQL, el cual proporciona a los alumnos una retroalimentación de las respuestas que envía en multitud de problemas prácticos, potenciando de esta forma la motivación y aprendizaje del alumnado de cara a mejorar su interés por la asignatura así como su rendimiento académico.

Sin embargo, LearnSQL tiene algunos inconvenientes en la práctica, como por ejemplo la introducción de manera manual por parte del profesorado de los casos de prueba que verifican las consultas de los alumnos. De esta idea nace nuestra motivación por desarrollar un sistema capaz de generar tablas de manera automática, que ejerciten todas las condiciones de una consulta y sus casos más extremos, tanto cuando se encuentra una sentencia con el parámetro `WHERE` como para sentencias con `JOIN ON`. Con esto se pretende conseguir un funcionamiento más eficiente de la biblioteca, que pueda integrar nuevas funcionalidades al juez de aprendizaje LearnSQL, obteniendo como resultado una corrección de mayor calidad de los problemas enviados y mayor facilidad de cara a la inserción de nuevos problemas por los profesores.

1.2. Objetivos

Como hemos mencionado con anterioridad, el objetivo de nuestro proyecto es desarrollar una biblioteca que, a partir de la definición de una o varias tablas SQL, genere las

filas necesarias para probar de la manera más exhaustiva posible una determinada consulta SQL. La intención es integrarlo posteriormente en el juez de aprendizaje LearnSQL, aportando nuevas funcionalidades que incrementen su rendimiento para poder ser utilizada en la asignatura de Bases de Datos de la UCM. Para ello, se han establecido una serie de mejoras y objetivos que se presentarán a continuación:

- Conocer la sintaxis de Oracle Database para poder hacer frente de forma correcta a la generación de tablas SQL y no incurrir en errores sintácticos.
- Ser capaces de analizar sentencias `CREATE TABLE` para identificar las restricciones que deberán cumplir los datos que se vayan a insertar, como las restricciones de unicidad, de clave primaria...
- Poder analizar consultas sobre la base de datos mediante `SELECT` de forma que se puedan tener en cuenta las condiciones que se indican en la consulta a la hora de generar los datos, como las condiciones de comparación o la unión de tablas mediante `JOIN`.
- Dada una o varias sentencias `CREATE TABLE` analizadas, ser capaces de restringir la generación aleatoria de datos a las características de la o las tablas.
- Dada una consulta `SELECT` insertada por el alumno, generar datos fronterizos a las condiciones que se requieren en una cláusula `WHERE` de la consulta.
- Otro de los objetivos es producir un código limpio y mantenible para que cualquier persona pueda proponer mejoras de cara a versiones futuras.

1.3. Plan de Trabajo

Para el desarrollo de este proyecto se ha utilizado una metodología *Open Source*, la cual se basa en un desarrollo descentralizado y colaborativo y que depende de la revisión entre compañeros y la producción de la comunidad. Esta metódica suele ser más flexible y duradera además de facilitar el acceso al código fuente por parte de los integrantes del grupo y del tutor y poder ser accesible al público para su utilización.

Durante el transcurso de este trabajo, se han ido realizando una serie de reuniones periódicas cada dos semanas donde el tutor revisaba el trabajo realizado y se concretaban comentarios de retroalimentación sobre el progreso hasta el momento. En cada sesión también se han ido marcando los objetivos de cara a la próxima reunión para poder repartir la carga de trabajo y progresar en el proyecto de manera eficaz.

El desarrollo del proyecto se ha distribuido en varios marcos temporales, donde se han ido solapando algunas funcionalidades y se han ido realizando diferentes tareas, detalladas a continuación:

- El primer marco temporal tuvo lugar en los meses de septiembre y octubre. En este primer periodo de tiempo nos centramos en el planteamiento de los objetivos a seguir durante el proyecto y en la búsqueda de bibliotecas en Python capaces de analizar consultas SQL dejándolas en un formato fácilmente usable. Además, se llevaron acabo

labores de formación y documentación en las diferentes herramientas que se iban a utilizar durante el proyecto.

- El segundo marco temporal transcurrió entre octubre y enero. En este segundo plazo de tiempo, comenzamos a llevar a cabo multitud de tareas de programación, intentando cumplir con los objetivos marcados en las reuniones. En este segundo lapso temporal se realizaron las implementaciones más básicas pero a la vez imprescindibles para tener un código bien estructurado y consistente, como la implementación de los distintos generadores de datos, el análisis de las restricciones de las columnas, el tratamiento de las diferentes condiciones de una consulta y la generación de filas para una tabla. Estas implementaciones permitirían más tarde poder desarrollar las funciones más complejas del proyecto.

- El tercer marco de tiempo tuvo lugar entre enero y marzo, cuando la dificultad del proyecto aumentó notablemente. En este tercer periodo de tiempo, realizamos las tareas más díficiles de implementar como el tratamiento de consultas con varias condiciones o la generación de filas para varias tablas. De esta forma, se fueron completando progresivamente los objetivos marcados para la finalización del proyecto.

- El último marco temporal transcurrió de abril a finales de mayo, donde llevamos acabo la finalización de las últimas funcionalidades y limpieza de código, centrándose principalmente en la redacción de la memoria y documentación necesaria.

Chapter 2

Introduction

In this chapter we will display the motivation for carrying out this project, the objectives we have set for its development and the work plan we have followed.

2.1. Motivation

Nowadays, in the Faculty of Computer Science at the Complutense University of Madrid (UCM), it is very common the use of learning judges in different programming and algorithm subjects which provide feedback to students when they perform practical problem submitions. Specifically, these programming judges are used to provide students with an automatic correction of their solutions in subjects such as Fundamentals of Programming, Fundamentals of Algorithmics, Data Structures or Databases. In the latter subject, a programming judge called LearnSQL has been used this academic year 2021/2022, which provides students with feedback on the answers they send in a multitude of practical problems, thus enhancing the motivation and learning of students in order to improve their interest in the subject as well as their academic performance.

However, LearnSQL has some drawbacks in practice, such as the manual introduction by the teacher of the test cases that verify the students' queries. From this idea was born our motivation to develop a system capable of generating tables automatically, which meet all the conditions of a query and its most extreme cases, both when a sentence with the parameter `WHERE` is found and for sentences with `JOIN ON`. This is intended to achieve a more efficient operation of the library, which can then integrate new features to the LearnSQL learning judge, resulting in a higher quality correction of the submitted problems and greater ease of insertion of new problems by teachers.

2.2. Objetives

As mentioned above, the aim of our project is to develop a library that, based on the definition of one or more SQL tables, generates the rows necessary to test a specific SQL query as exhaustively as possible. The intention is to subsequently integrate it into the LearnSQL learning judge, providing new functionalities that increase its performance so that it can be used in the UCM Databases subject. To this end, a series of improvements and objectives have been established, which will be presented below:

- Get to know the syntax of Oracle Database in order to be able to deal correctly with the generation of SQL tables without making syntactic errors.
- To be able to parse `CREATE TABLE` sentences to identify the constraints to be met by the data to be inserted, such as unique or primary key constraints.
- To be able to parse `SELECT` queries so that the conditions indicated in the query can be taken into account when generating the data, such as the comparison conditions or the joining of tables using `JOIN`.
- Given one or more parsed `CREATE TABLE` statements, be able to restrict the random generation of data to the characteristics of the table(s).
- Given a `SELECT` query inserted by the student, generate boundary data to the conditions required in a `WHERE` clause of the query.
- Produce clean and maintainable code so that anyone can propose improvements for future versions.

2.3. Workplan

For the development of this project, an *Open Source* methodology has been used, which is based on a decentralised and collaborative development and which depends on peer review and community production. This methodology tends to be more flexible and durable, as well as facilitating access to the source code by group members and the tutor and making it publicly available for use.

During the course of this project, a series of regular meetings were held every two weeks where the tutor reviewed the work done and gave feedback on the progress made so far. At each session, objectives were also set for the next meeting in order to share the workload and keep progressing effectively.

The development of the project has been distributed in several time frames, where some functionalities have been overlapping and different tasks have been carried out, as detailed below:

- The first time frame took place between September and October. In this first period of time, we focused on setting the objectives to be followed during the project and searching for Python libraries capable of parsing SQL queries in an easily usable format. In addition, training and documentation work was carried out on the different tools to be used during the project.
- The second time frame occurred from October to January. In this second time frame, we began to carry out a multitude of programming tasks, trying to meet the objectives set in the meetings. During this second time frame, the most basic but essential implementations were carried out in order to have a well structured and consistent code, such as the implementation of the different data generators, the analysis of column constraints, the treatment of different conditions in a query

and the generation of rows for a table. These implementations would later allow the development of the more complex functions of the project.

- The third time frame took place between January and March, when the difficulty of the project increased significantly. In this third time frame, we carried out the most difficult tasks to implement, such as the processing of queries with several conditions or the generation of rows for several tables. In this way, the objectives set for the completion of the project were progressively completed.
- The last time frame occurred from April to the end of May, where the we completed the last functionalities and code clean-up, focusing mainly on writing the necessary memory and documentation.

Capítulo 3

Preliminares

A lo largo de todo este trabajo, se han utilizado diferentes tecnologías que han facilitado el desarrollo del sistema de generación de tablas SQL. En este capítulo vamos a describir y profundizar en las principales tecnologías y herramientas utilizadas para la realización del proyecto.

3.1. SQL

Structured Query Language, es un lenguaje diseñado para administrar y recuperar información de sistemas de gestión de bases de datos relacionales. Es un lenguaje de consulta que permite realizar operaciones de selección, inserción, actualización y borrado de datos, así como operaciones administrativas sobre las bases de datos. Para el estudio y profundización de este lenguaje hemos utilizado los libros Oracle Database PL/SQL Language Reference [1] y Fundamentos de Sistemas de Bases de Datos [2].

3.2. Oracle

Oracle Database [3], es un sistema de gestión de bases de datos relacionales de Oracle, que permite almacenar y representar los datos de la empresa y los clientes en forma de conjuntos de datos organizados. Como software de bases de datos, Oracle Database optimiza la gestión y seguridad de los conjuntos de datos creando esquemas estructurados.

Las cantidades de datos se estructuran en columnas, tablas y filas, y los puntos de datos se relacionan con la ayuda de atributos. La gran ventaja de Oracle Database es que organiza y presenta volúmenes de datos de manera intuitiva y eficiente. Además, permite decidir al usuario si quiere usar Oracle Database en entornos locales o en la nube.

La decisión de enfocar este proyecto en Oracle siguiendo su sintaxis [4] de tipos de datos se basa en la utilización de esta herramienta en la Facultad de Informática para impartir las asignaturas de bases de datos a través del entorno de Learn SQL donde se realizan los envíos por parte del alumnado para su ejecución y corrección de manera dinámica.

3.2.1. Oracle Live SQL

Oracle Live SQL [5] es un nuevo servicio web de Oracle Database 12c que permite a los desarrolladores crear y ejecutar fácilmente scripts SQL en bases de datos Oracle y así probar las funcionalidades de esta. Es adecuado para demostraciones rápidas de código, depuración y resolución de problemas. En general, ofrece una forma sencilla de utilizar Oracle para aquellos que están aprendiendo o probando SQL, PL/SQL u otra función.

La herramienta está basada en la web, lo que permite ejecutar consultas y demostraciones en un entorno de Oracle sin necesidad de una configuración completa. Se puede acceder a esta herramienta desde cualquier lugar, no requiere ninguna configuración adicional ni concesión de permisos, y ahorra tiempo al eliminar todo el trabajo de configuración de las sesiones.

Esta herramienta web nos ha sido de gran utilidad en el proyecto puesto que nos ha facilitado el entendimiento del lenguaje SQL y nos ha permitido comprobar y verificar diferentes consultas SQL siguiendo la sintaxis de Oracle para poder construir un código consistente y que cumpla todas las reglas del lenguaje.

3.3. Learn SQL

Learn SQL [6] es un juez automático utilizado en la Facultad de Informática de la Universidad Complutense de Madrid (UCM) que permite potenciar el aprendizaje de la programación dentro del lenguaje SQL en los alumnos y favorecer la rapidez de retroalimentación entre el profesorado y los estudiantes.

Este corrector automático está desarrollado con el framework de Django y se utiliza sobre Oracle. Con esta herramienta se pretende mejorar el aprendizaje del lenguaje SQL y trata de ser extendida con nuevas funcionalidades en el futuro. Es por ello que este proyecto de generación automática de tablas SQL está orientado a su futura incorporación dentro de este entorno para facilitar y mejorar la corrección de los ejercicios.

3.4. Diagramas de ferrocarril y notación BNF

Todos los programas deben seguir normas estrictas en cuanto a su sintaxis y estructura, es por ello que para el desarrollo de un programa se debe definir de forma totalmente inequívoca todas las reglas de un determinado lenguaje.

La notación BNF (Backus-Naur Form) [7], es un metalenguaje usado para expresar gramáticas libres de contexto así como una forma matemática de definir la sintaxis sin ambigüedades. Se utiliza como notación para las gramáticas de los lenguajes de programación, de los sistemas de comando y de los protocolos de comunicación.

Por otra parte, los diagramas de ferrocarril son una forma de representación gráfica de BNF siguiendo un formato formal establecido. Los diagramas de ferrocarril son visuales y pueden ser entendidos más fácilmente por lo que resultan de utilidad de cara a comprender la notación BNF mencionada con anterioridad.

Si enfocamos estos dos conceptos de cara al lenguaje SQL, podemos encontrar multitud de diagramas BNF sobre la sintaxis SQL que nos han permitido estudiar en profundidad todas las normas y metodolgías de este lenguaje de cara a desarrollar un código que respete todas las reglas y sea completamente funcional.

En la figura 3.1 podemos ver un diagrama de ferrocarril que representan la sintaxis de una sentencia *INNER, CROSS, JOIN*.

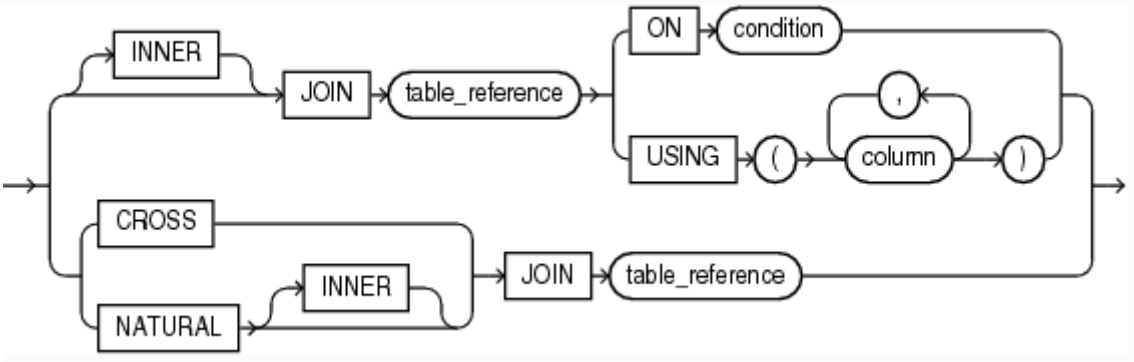

Figura 3.1: Diagrama de ferrocarril de una sentencia *INNER, CROSS, JOIN*
Fuente: https://docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/SELECT.html#GUID-CFA006CA-6FF1-4972-821E-6996142A51C6

3.5. Herramientas utilizadas

A continuación, vamos a presentar las herramientas utilizadas para llevar acabo el desarrollo de nuestro proyecto.

3.5.1. Python

Python es un lenguaje de programación de alto nivel interpretado, orientado a objetos y con semántica dinámica. Es un lenguaje de propósito general, lo que significa que puede usarse para crear una variedad de programas diferentes y no está especializado para ningún problema específico. Esta versatilidad, junto con su facilidad de uso, lo ha convertido en uno de los lenguajes de programación más utilizados en la actualidad.

La sintaxis simple y fácil de aprender de Python enfatiza la legibilidad y, por lo tanto, reduce el costo de mantenimiento del programa. Python admite módulos y paquetes, lo que fomenta la modularidad del programa y la reutilización del código.

La decisión de implementar este proyecto mediante el lenguaje de Python se basa principalmente en su polivalencia de cara al desarrollo de software puesto que proporciona un lenguaje de alto nivel que facilita su entendimiento, lo que nos ha facilitado su aprendizaje al comienzo del proyecto y es un lenguaje de paradigmas múltiples, que admite programación estructurada, funcional y orientada a objetos. Además, es un lenguaje de propóstito general que admite una amplia colección de bibliotecas y frameworks con numerosos módulos integrados, lo que nos ha permitido incorporar bibliotecas como `mo-sql-parsing` o faker para analizar las sentencias y generar datos aleatorios.

3.5.2. Mo-sql-parsing

`Mo-sql-parsing` [8] es una biblioteca de Python desarrollada por Kyle Lahnakoski cuyo objetivo principal se centra en convertir las consultas SQL en árboles de análisis aptos para JSON. Esta biblioteca es lo suficientemente útil para explorar los datos y sentencias de manera primitiva lo que permite a los programadores el estudio de datos con consultas SQL básicas.

Tras indagar en diversas bibliotecas y frameworks de Python como `sqlparse` [9], `pyparsing` [10] o `sql-metadata` [11] que resultaran de utilidad de cara a la conversión de consultas SQL en sentencias sencillas de analizar, decidimos incorporar esta bilbioteca puesto que convierte los datos a un formato muy manejable y fácil de manipular de cara a estudiar y tratar la información. Además, es una biblioteca que se ha mantenido bastante actualizada en el transcurso del proyecto por sus desarrolladores mediante periódicos reajustes y correcciones.

A continuación, se muestra un ejemplo de uso donde se analiza una sentencia `CREATE TABLE`:

```
>>> from mo_sql_parsing import parse, normal_op

>>> parse("CREATE TABLE Club(CIF NUMBER(4) PRIMARY KEY
CHECK(CIF > 0), Nombre_Club VARCHAR2(50) NOT NULL)",
calls=normal_op)
```

El resultado de la llamada a la función *parse* devolvería el siguiente objeto:

```
{‘create table’: {
    ‘name’: ‘Club’,
    ‘columns’: [{
        ‘name’: ‘cif’,
        ‘type’: {‘op’: ‘number’, ‘args’: [4]},
        ‘primary_key’: True,
        ‘check’: {‘op’: ‘gt’, ‘args’: [‘CIF’, 0]}
    },{
        ‘name’: ‘nombre_club’,
        ‘type’: {‘op’: ‘varchar2’, ‘args’: [50]},
        ‘nullable’: False
    }]
}}
```

3.5.3. Faker

`Faker` es un paquete de Python que permite generar datos falsos de manera pseudoaleatoria con el objetivo de probar o llenar bases de datos con algunos datos ficticios. De esta forma, Faker genera datos de prueba de todo tipo como nombres, direcciones, correos electrónicos, texto u oraciones.

Este paquete también permite la creación de archivos JSON de datos falsos, impresión de datos en diferentes idiomas, creación de perfiles y generación de datos falsos particulares

o con requisitos establecidos.

La incorporación de esta biblioteca en nuestro proyecto nos ha servido de gran utilidad en la fase de generación de datos aleatorios que veremos en el capítulo 4, especialmente en la generación de datos formados por caracteres que contengan ciertas restricciones en la longitud o en la propia morfología de las cadenas. A continuación se muestran una serie de funciones de Faker con sus respectivos resultados:

```
>>> from faker import Faker
>>> faker = Faker()
>>> faker.name()
    'Laura Vargas'
>>> faker.address()
    '5710 Giles Inlet Apt. 459\nStephaniefort, NM 14267'
>>> faker.text()
    'Oil soldier administration whole stock. Market baby
    sound eight.\n Chair book store body hear section head.
    Head throughout short degree admit. State also partner
    performance although remain.'
```

3.6. Aspectos a tratar en la generación de datos

En esta sección vamos a presentar algunos de los diferentes aspectos que hemos tenido que tratar de una manera específica para conseguir implementar un código capaz de generar datos que puedan probar las consultas.

3.6.1. Generación de casos de prueba que cubran una consulta

Uno de los principales aspectos que hemos tenido que tratar en el proyecto se centra en analizar las sentencias SQL recibidas para la posterior generación de datos. Para ello, se van produciendo datos de forma secuencial de manera que estos verifiquen si una consulta `SELECT` es correcta.

La metodología que hemos usado para este fin se basa en analizar sintácticamente la consulta recibida, detectar las restricciones que esta posea y generar valores teniendo en cuenta todas estas restricciones. De esta forma, hay filas de una o varias tablas que satisfacen la consulta y otras que no. Si la consulta `SELECT` está bien formulada, siempre habrá por lo menos una fila que satisfaga la consulta.

Un ejemplo donde podemos ver este procedimiento podría ser el siguiente, donde tenemos una tabla *Club* y una sentencia del tipo `SELECT` con dos condiciones de comparación:

-
```
Club(
    CIF NUMBER(9),
    Nombre VARCHAR2(30),
    Sede VARCHAR2(50),
    NumSocios NUMBER(5),
    NumAsientos(5)
);
```

- `SELECT * FROM Club WHERE NumSocios > 100 AND NumAsientos = 88;`

Los valores posibles que deberían ser generados para probar esta consulta serían los siguientes:

1. `NumSocios > 100, NumAsientos = 100`
2. `NumSocios > 100, NumAsientos != 100`
3. `NumSocios = 100, NumAsientos = 100`
4. `NumSocios = 100, NumAsientos != 100`
5. `NumSocios < 100, NumAsientos = 100`
6. `NumSocios < 100, NumAsientos != 100`

Por tanto, cuando probamos todas las posibilidades de cada consulta recibida, se utiliza la combinatoria para generar valores que analicen cada tipo de restricción encontrada. Esto permite cubrir los valores que cumplen la restricción pero también los valores próximos que no lo hacen.

Capítulo 4

Generación aleatoria de valores

SQL es un lenguaje muy extenso con varias formas de expresar una misma instrucción. Por esta razón, hemos tenido que acotar los tipos de datos soportados y las restricciones manejadas. A continuación, vamos a proceder a enumerarlas y explicar cómo se han aplicado.

4.1. Tipos de datos soportados

A la hora de generar datos para tablas SQL en la base de datos de Oracle, hemos tenido que investigar a fondo sus tipos de datos. Además de los tipos de datos integrados de Oracle, hay muchos más que esta base de datos soporta por compatibilidad.

4.1.1. Tipos de datos integrados de Oracle

En este subapartado vamos a detallar los tipo de datos integrados de Oracle.

Tipos de datos numéricos:

En la figura 4.1 se muestra un ejemplo de la sintaxis de un número.

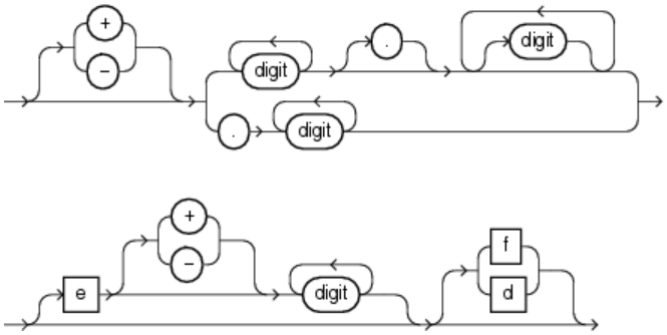

Figura 4.1: Diagrama del ferrocarril para la sintaxis de un número

Fuente: `https://docs.oracle.com/database/121/SQLRF/sql_elements003.htm#SQLRF00220`

- Números en coma fija:

- `NUMBER[(`*p* `[,`*s*`])]`: especifica un número en coma fija.

  p es la precisión numérica, es decir la cantidad total de dígitos decimales significantes, donde el dígito más significantes es aquel que está más a la izquierda, sin contar si hubiera algún cero. Acepta valores entre el 1 y el 38, ambos inclusive. Si un dato que se quiere insertar supera la precisión Oracle devuelve un error. Oracle garantiza la portabilidad de números con precisión hasta 20 dígitos de base 100, lo cual es equivalente a 39 o 40 números decimales, dependiendo de la posición del punto decimal.

  Por ejemplo, una columna definida como `NUMBER(4)` almacena números de 4 dígitos; como el 4862

  s es la escala, o el número de dígitos desde el punto decimal hasta el dígito menos significantivo. Su rango varía entre -84 y 127, ambos inclusive. Una escala **positiva** es el número de dígitos significativos desde la derecha del punto decimal hasta el dígito menos significativo. Una escala **negativa** indica el número de dígitos significativos a la izquierda del punto decimal, sin incluir el dítito menos significativo. La escala negativa sirve para redondear el dato al número especificado de lugares a la izquierda del punto decimal. Además, la escala puede ser mayor que la precisión. Esto es útil sobre todo cuando se utiliza la notación científica. Cuando esto sucede, la escala indica el número máximo de dígitos significantes a la derecha del punto decimal. Si se inserta un dato que supera la escala, Oracle lo redondea. El valor por defecto de *s* es 0.

  A continuación se muestran una serie de ejemplos:

Tipo de dato	**Número insertado**	**Número almacenado**
`NUMBER(4, 2)`	12.34	12.34
`NUMBER(4, 2)`	123.45	Error de precisión
`NUMBER(4, 2)`	12.343	12.34
`NUMBER(4, -2)`	123.4	100
`NUMBER(2, 4)`	0.34567	0.3456

- Números en coma flotante: Un número en coma flotante puede tener un punto decimal en cualquier posición o no tener ninguno.

  Oracle almacena este tipo de números usando precisión binaria (dígitos 0 y 1) y no precisión decimal (dígitos 0 a 9) como con `NUMBER`.

  - `BINARY_FLOAT`: número en coma flotante de 32 bits y precisión simple. Cada valor de este tipo necesita 5 bytes, incluyendo un byte de longitud. Por ejemplo, una columna que almacena datos de tipo `BINARY_FLOAT` podría contener valores como 25f ó +6.34F.
  - `BINARY_DOUBLE`: número en coma flotante de 64 bits y precisión doble. Cada valor de este tipo necesita 9 bytes, incluyendo un byte de longitud. Por ejemplo, una columna que almacena datos de tipo `BINARY_DOUBLE` podría contener valores como 0.5d ó -1D.

Tipos de datos de caracteres:

En la figura 4.2 se muestra un ejemplo de la sintaxis de una cadena de caracteres, donde *N* o n indica el uso del conjunto de caracteres nacional, *c* representa cualquier caracter del conjunto, *Q* o *q* indica que se utilizará un mecanismo alternativo de citación y *quote_ delimiter* es cualquier caracter excepto el espacio, el tabulador y el retorno.

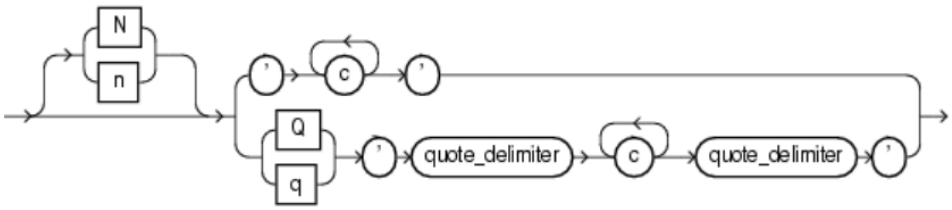

Figura 4.2: Diagrama del ferrocarril de cadenas de caracteres
Fuente: https://docs.oracle.com/database/121/SQLRF/sql_elements003.htm#SQLRF00218

- `NVARCHAR2(`*size*`)`: este tipo de dato usa el conjunto de caracteres nacional y puede almacenar cadenas de caracteres de tamaño variable. El parámetro *size* indica el tamaño máximo de bytes que puede almacenar. El tamaño máximo soportado por Oracle es de 4000 bytes.
- `VARCHAR2(`*size* `[BYTE | CHAR])`: este tipo de datos puede almacenar una longitud variable de caracteres. El parámetro *size* indica el número máximo de bytes (por defecto o usando el calificador BYTE) o de caracteres (usando el calificador CHAR) que puede almacenar la columna. El tamaño máximo soportado por Oracle es de 4000 bytes, igual que `NVARCHAR2`.
- `VARCHAR(`*size* `[BYTE | CHAR])`: este tipo de dato es actualmente sinónimo de `VARCHAR2`, sin embargo, Oracle recomienda el uso del segundo.
- `NCHAR[(`*size*`)]`: este tipo de dato usa el conjunto de caracteres nacional y almacena cadenas de caracteres de tamaño fijo. El parámetro *size* indica ese tamaño. El tamaño máximo está determinado por la definición del conjunto de caracteres nacional y Oracle soporta hasta 2000 bytes.
- `CHAR[(`*size* `[BYTE | CHAR])]`: este tipo de datos almacena cadenas de caracteres de tamaño fijo. El parámetro *size* indica ese tamaño. El tamaño máximo soportado por Oracle es de 2000 bytes o caracteres. Los calificadores BYTE y CHAR poseen las mismas semánticas que `VARCHAR2`.

Tipos de datos de fechas:

- `DATE`: almacena el día, mes y año de una fecha. Si no se especifica alguno de ellos, Oracle lo convierte al valor por defecto. Con la función `TO_DATE` se puede convertir una cadena de caracteres a una fecha, pudiendo especificar un formato para la cadena.
- `TIMESTAMP[(`*fractional_seconds_precision*`)]`: este tipo de datos es una extensión del anterior en el que también se pueden especificar horas, minutos, segundos y fracciones de segundo. El parámetro *fractional_ seconds_ precision* sirve precisamente para especificar el número de dígitos (entre 0 y 9) que queremos que Oracle almacene como parte fraccional del segundo.

Además de los mencionados, Oracle tiene otros tipos de datos integrados que hemos decidido no soportar en nuestra biblioteca. Esto es debido a dos razones: la primera es que la biblioteca que utilizamos para analizar las sentencias SQL, `mo-sql-parsing`, no soporta tipos de datos cuyo nombre tenga más de una palabra. En esta situación se encuentran cuatro tipos de datos para almacenar fechas:

- `TIMESTAMP[(`*fractional_seconds*`)] WITH TIME ZONE`
- `TIMESTAMP[(`*fractional_seconds*`)] WITH LOCAL TIME ZONE`
- `INTERVAL YEAR[(`*year_precision*`)] TO MONTH`
- `INTERVAL DAY[(`*day_precision*`)] TO SECOND[(`*fractional_seconds*`)]`

La segunda razón es que algunos de los tipos de datos que ofrece Oracle se encuentran fuera del alcance de nuestro proyecto, ya que consideramos que son tipos de datos muy poco usados en el aprendizaje del lenguaje SQL. Estos son: `RAW(size)`, `LONG RAW`, `ROWID`, `UROWID[(size)]`, `CLOB`, `NCLOB`, `BLOB` y `BFILE`.

4.1.2. Tipos de datos de ANSI, SQL/DS y DB2

Además de los tipos de datos integrados de Oracle, también soportamos algunos de los del estándar ANSI[1] y de los productos SQL/DS y DB2 de IBM que Oracle convierte internamente a sus propios tipos de datos.

Tipos de datos ANSI, SQL/DS o DB2	Conversión de Oracle
`CHARACTER(n)` `CHAR(n)`	`CHAR(n)`
`NUMERIC(p,s)` `DECIMAL(p,s)`	`NUMBER(p, s)`

`INTEGER`
`INT`
`SMALLINT`
	`NUMBER(38)`

`FLOAT` `REAL`	`NUMBER`

Hay otros tipos de datos del estándar o de las bases de datos mencionadas que Oracle convierte pero que no hemos podido soportar en nuestro proyecto, de nuevo, por incompatibilidad con la biblioteca de análisis de sentencias SQL, `mo-sql-parsing`. Estos son: `CHARACTER VARYING(n)`, `CHAR VARYING(n)`, `NATIONAL CHARACTER(n)`, `NATIONAL CHAR(n)`, `NATIONAL CHARACTER VARYING(n)`, `NATIONAL CHAR VARYING(n)`, `NCHAR VARYING(n)`, `DOUBLE PRECISION` y `LONG VARCHAR`.

4.2. Restricciones de integridad manejadas

El lenguaje SQL, además de ofrecer diferentes tipos de datos, permite también restringirlos para que cumplan las características que el cliente necesite para gestionar su base de datos relacional.

[1] ANSI: American National Standards Institute

La biblioteca que hemos desarrollado contempla las restricciones de integridad definidas a continuación:

- **Restricción NOT NULL:** prohíbe almacenar valores nulos en la columna donde se declara.
- **Restricción de unicidad:** prohíbe que varias filas tengan el mismo valor en una columna o combinación de columnas. Sí que permite almacenar varios valores nulos. Se define mediante la palabra clave `UNIQUE`.
- **Restricción de clave primaria:** es una combinación de las dos anteriores, prohíbe que varias filas tengan el mismo valor en una columna o combinación de columnas y además no pueden contener un valor nulo. Se define utilizando la palabra clave `PRIMARY KEY`.
- **Restricción de clave externa:** exige que los valores de una tabla coincidan con los valores de otra. Se declaran mediante las palabras clave `FOREIGN KEY` y `REFERENCES`.
- **Restricción de comprobación:** exige que un valor cumpla con una condición determinada. La palabra clave que las define es `CHECK`. Por ejemplo:

  ```
  CREATE TABLE Club (CIF NUMBER(4) CHECK (CIF > 0));
  ```

  obliga a que todos los valores almacenados en la columna `CIF` sean mayores que 0. Dentro de las restricciones de comprobación nos gustaría hacer hincapié en dos comparaciones usadas en cadenas de caracteres: el operador `LIKE` y la función `LENGTH`.

  - **Operador `LIKE`:** se usan para comparar un caracter o cadena de caracteres con un patrón. Los patrones pueden contener dos tipos de caracteres especiales, llamados comodines (*wildcards* en inglés). Estos son el guión bajo (_) y el símbolo %. El primero, se debe reemplazar por exactamente un caracter. El segundo, puede reemplazarse por uno, varios o ningún caracter. Por ejemplo, si Nombre_Club es 'Club Deportivo Miranda', la siguiente expresión sería verdadera:

    ```
    Nombre_Club LIKE 'Club %ivo M__anda'
    ```

  - **Función `LENGTH`**: evalúa el número de caracteres que contiene una cadena. Así, podemos comparar la longitud de una cadena de caracteres. Por ejemplo, la siguiente expresión sería verdadera si Nombre_Club contiene una cadena de caracteres de longitud mayor a 5:

    ```
    LENGTH(Nombre_Club) > 5
    ```

Para la generación de valores aleatorios, descritos en la sección 4.3, se han tenido en cuenta las restricciones NOT NULL y las de comprobación. El resto, se han tenido en cuenta a la hora de generar las diferentes filas, como explicaremos en el capítulo 5.

Oracle permite un sexto tipo de restricción de integridad: la restricción *REF*. Esta permite describir más a fondo la relación entre una columna *REF* (los valores de una columna *REF* referencian objetos en otro tipo de objeto o en una tabla relacional) y el objeto al que referencia. No soportamos esta restricción debido a que la referenciación a objetos queda fuera del alcance del proyecto.

Además, Oracle soporta dos formas de declarar las restricciones de integridad:

- **Declaración *inline*:** es parte de la definición de una columna o atributo, es decir, se declara en la misma línea que la columna. A continuación, se muestra un ejemplo de uso donde, a través de una declaración *inline*, se define una restricción de clave primaria:

```
CREATE TABLE Club(CIF NUMBER(4) PRIMARY KEY);
```

  Además, en la figura 4.3 podemos ver la definición general las restricciones mediante este tipo de declaración.

- **Declaración *out-of-line*:** es parte de la definición de una tabla, es decir, se declara en una línea al final de una sentencia `CREATE TABLE` mediante la palabra clave `CONSTRAINT` seguido de un nombre que define la restricción, la propia restricción y la columna a la que afecta. Un ejemplo de declaración *out-of-line* donde se define una restricción clave primaria sería la siguiente:

```
CREATE TABLE Club(
      CIF NUMBER(4),
      CONSTRAINT pk_cif PRIMARY KEY (CIF));
```

  En la figura 4.4 podemos ver todas las posibilidades en cuanto a la declaración *out-of-line* de una restricción.

En la versión actual de nuestro proyecto soportamos la declaración de restricciones *inline*. Como trabajo futuro proponemos implementar las restricciones *out-of-line*.

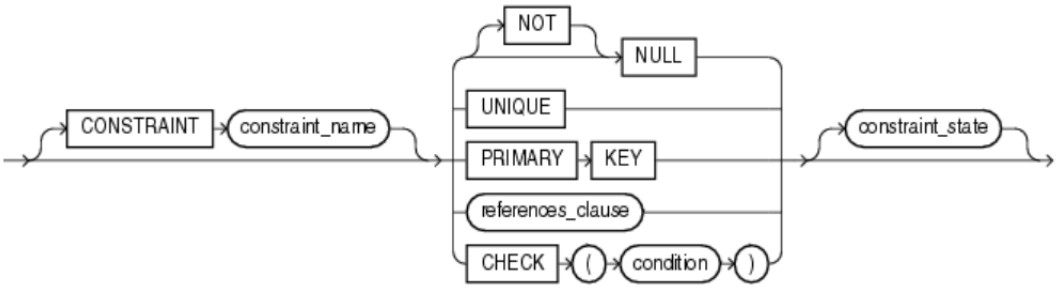

Figura 4.3: Diagrama del ferrocarril de la declaración *inline* de una restricción
Fuente: https://docs.oracle.com/cd/B19306_01/server.102/b14200/clauses002.htm#CJAGIICD

4.3. Generación aleatoria de valores

En esta sección explicaremos cómo generamos los datos, teniendo en cuenta los tipos de datos explicados en la sección 4.1 y las restricciones mencionadas en la sección 4.2.

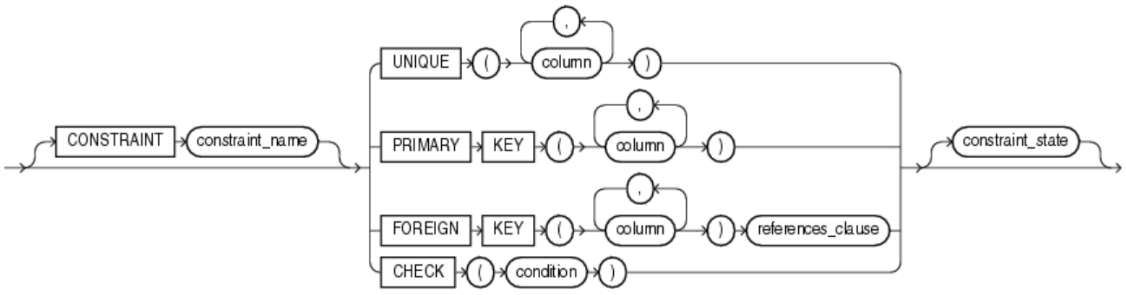

Figura 4.4: Diagrama del ferrocarril de la declaración *out-of-line* de una restricción
Fuente: `https://docs.oracle.com/cd/B19306_01/server.102/b14200/clauses002.htm#CJADJGEC`

4.3.1. Generación aleatoria de valores nulos

Una de las cuestiones que tuvimos que tratar a la hora de generar datos aleatorios se encontraba en la posibilidad de que algunos valores pudieran ser nulos. Una columna puede contener valores nulos si se especifica en su definición con la palabra clave `NULL` o, simplemente, si se omite una restricción `NOT NULL`.

A la hora de generar valores, si nos encontramos con una columna que puede contener estos valores, usamos una función auxiliar que devuelve `NULL` con un 20 % de probabilidad. Asimismo, disponemos de una función para que el usuario pueda ajustar esta probabilidad a la deseada.

4.3.2. Generación aleatoria de números

Al momento de analizar las restricciones correspondientes al tipo de dato hemos considerado lo siguiente:

- Los tipos de datos `INT, INTEGER, SMALLINT` poseen una precision de 38 dígitos y una escala igual a 0.
- Por simplicidad, los tipos de datos de coma flotante `FLOAT`, `BINARY_FLOAT`, `BINARY_DOUBLE` y `REAL` los convertimos a números de coma fija con precisión 10 y escala 4.
- La precisión y escala de los tipos de datos de numéricos de coma fija son 38 y 127, respectivamente.

Una vez cubierta la precisión y la escala del tipo de dato numérico, procedemos a determinar si existe una restricción de comprobación de la columna y, en caso afirmativo, analizar las condiciones. Las condiciones soportadas para los tipos de datos numéricos son los operadores de comparación (=, != (ó <>), >, <, >=, <=). Tras analizarlo, se guarda el número máximo y el número mínimo que se puede generar de forma que cumpla los requisitos, así como el valor al que se tiene que igualar o el valor al que debe ser diferente en caso de que se especifique en la restricción.

A continuación, generamos el valor numérico. Si la columna contiene una condición de igualdad (=), se genera el valor de la condición. Para las condiciones de desigualdad, se genera un valor numérico aleatorio con la precisión y escala determinada y que se encuentre

entre los valores máximo y mínimo establecidos. Además, si la columma contiene una condición de desigualdad de tipo !=, se vuelve a generar un número hasta que sea diferente al valor de la condición.

4.3.3. Generación aleatoria de cadenas de caracteres

El proceso de generación de cadenas de caracteres sigue un enfoque similar al del apartado anterior.

Para empezar, analizamos las restricciones correspondientes al tipo de dato, es decir, si acepta tamaños variables o no, y en caso afirmativo el tamaño máximo. Seguidamente, se examina la condición de comprobación, en caso de que la posea. Las condiciones soportadas para los tipos de datos de cadenas de caracteres son:

- Comparación de cadenas sobre patrones mediante el operador `LIKE` y sin caracteres de escape.
- Comparaciones de longitud mediante la función `LENGTH` y los operadores de comparación =, != (ó <>), >, <, >=, <=.

Después de averiguar las condiciones, se determina el tamaño máximo y mínimo de la cadena de caracteres y también el patrón de caracteres a seguir si así se indica en las condiciones de la restricción.

Tras este analisis, procedemos a generar las cadenas de caracteres. En primer lugar, creamos una instancia de Faker. Seguidamente, cuando la columna no posee una restricción de comprobación con una condición `LIKE`, se asigna un número aleatorio entre el tamaño mínimo y el tamaño máximo mediante *random* [12], una biblioteca integrada de Python. Este valor será el tamaño final de la nueva cadena de caracteres. A continuación, generamos un texto aleatorio mediante una función de Faker y ajustamos el tamaño del texto generado al tamaño determinado anteriormente.

En el caso de que la columna posea una restricción con una condición `LIKE`, el proceso de generación es algo más tedioso. Primeramente, buscamos el número máximo de caracteres que podemos agregar en caso de que nos encontremos un símbolo %. Seguidamente, iteramos sobre los caracteres de la cadena en busca de comodines. Si el caracter no es un comodín, no se cambia. En cambio, si encontramos el comodín guión bajo (_), se elige una letra del abecedario con la ayuda de la biblioteca *random* y se sustituye el comodín por esa letra. También, si encontramos el comodín %, mediante la biblioteca *random* elegimos un número entre 0 y el número máximo buscado en el primer paso para determinar la longitud del texto que va a reemplazar al comodín. Después, mediante una función de Faker generamos un texto y lo acotamos a la longitud obtenida, sustituimos el comodín por este texto y actualizamos el valor del número máximo de caracteres que se pueden agregar.

4.3.4. Generación aleatoria de fechas

Hemos simplificado la generación de fechas de forma que no se atiene a las restricciones de comprobación. Esto es debido a que lo hemos considerado fuera del alcance del proyecto para esta versión, ya que no es un aspecto prioritario en el desarrollo.

En cualquier caso, seguimos necesitando analizar el tipo de dato y sus parámetros. Empezamos identificando el tipo de dato, es decir, si es de tipo `DATE` o de tipo `TIMESTAMP`, y la precisión en segundos en el caso del tipo `TIMESTAMP`. Con esto y dadas una fecha inicial y una fecha final fijadas en '01/01/1900'y '12/12/2022', respectivamente, a través de la bibilioteca *random* se elige una fecha entre ese rango. Si se trata de un dato de tipo `TIMESTAMP`, deberemos, asimismo, limitar la fracción de segundo al número de dígitos indicado en el parámetro del dato.

Capítulo 5

Generación de filas en las tablas

Para la construcción de un sistema de generación de tablas SQL, es necesario implementar un código capaz de generar filas que cumplan con todos los requisitos para poder ser insertadas en el sistema. En este capítulo vamos a explicar la metodología que hemos seguido para generarlas.

5.1. Análisis de sentencias `CREATE TABLE` y consultas `SELECT`

Las sentencias `CREATE TABLE` permiten definir la estructura de una tabla, definiendo sus columnas y para cada una de ellas su nombre, tipo de dato y restricciones de integridad.

Por otro lado, las sentencias `SELECT` se utilizan para obtener columnas de una o varias tablas de la base de datos, pudiendo aplicar diferentes condiciones para filtrar el resultado.

Para llevar a cabo todo el proceso de creción de filas para una o varias tablas, lo primero que necesitamos es analizar los datos de entrada, es decir, las sentencias `CREATE TABLE` y las consultas con `SELECT`. Para ello, el procedimiento que hemos seguido se basa en la obtención, mediante la biblioteca `mo-sql-parsing`, de un árbol de análisis sintáctico con el que poder descomponer las sentencias y estudiar cada componente.

Una vez se obtienen las sentencias analizadas, podemos acceder a la información que contienen mediante un conjunto de instrucciones sencillas. De las sentencias `CREATE TABLE` obtenemos el nombre de las columnas, sus tipos de datos y sus restricciones de integridad. De las cláusulas `SELECT` obtenemos el nombre de las columnas que se quieren recuperar y las condiciones de filtrado.

5.2. Restricciones de unicidad y de clave primaria

Un factor a tener muy en cuenta a la hora de generar filas válidas en tablas es saber si una columna contiene una restricción de unicidad o de clave primaria (explicadas en la sección 4.2). Dada la definición de una tabla mediante una sentencia `CREATE TABLE` y que analizamos con la biblioteca `mo-sql-parsing`, debemos averiguar si alguna de sus columnas posee una de estas restricciones. En tal caso, necesitamos almacenar los valores ya creados de forma que en el proceso iterativo de generación de datos para esa columna

nos aseguremos de que no se repita ningún valor.

5.3. Geración de filas en una tabla

En este apartado vamos a presentar el proceso de generación de múltiples filas teniendo en cuenta las restricciones posibles dentro de una consulta. Para presentar el modelo de manera más clara, en este apartado explicaremos la inserción de filas en una única tabla. Luego, en el apartado 5.4, desarrollaremos el procedimiento que hemos seguido cuando disponemos de varias tablas. Para poder llevar acabo el proceso de generación de filas es necesario seguir una metodología específica y cuidadosa donde se tenga en cuenta cada componente de la consulta así como las delimitaciones existentes y su índole de cara a que las filas cumplan los requisitos.

5.3.1. Consultas con parámetro `WHERE`

Dentro del análisis de las consultas para la generación de filas, es importante tener en cuenta las sentencias con el parámetro `WHERE`, el cual se utiliza para especificar la condición de búsqueda de las filas devueltas. Para cada condición que se desea evaluar en una consulta, generamos un valor válido para ese requisito y uno o dos valores inválidos.

Dentro de este ámbito del desarrollo, debemos considerar que existen criterios de búsqueda de diferentes tipos como pueden ser las condiciones de comparación o sobre cadenas de caracteres. A continuación, veremos cómo se ha tratado cada una de estas premisas de cara a la posterior generación de los datos.

- **Condiciones de comparación:** Son aquellas que utilizan operadores de comparación para ejecutar operaciones matemáticas con dos expresiones. Podemos encontrar operadores de igualdad o de desigualdad como pueden ser $=$, $!=$ (ó $<>$), $>$, $<$, $>=$ ó $<=$ y que se utilizan para comparar datos numéricos, cadenas de caracteres y fechas.

  A la hora de tratar este tipo de condiciones, la metodología que hemos seguido se centra en la generación de valores próximos al parámetro umbral utilizado en la comparación. Para ello, se tratan todos los operadores de la misma forma de cara a la generación de los valores, con el objetivo de cubrir todas las posibilidades que puedan verificar si la consulta está bien formulada.

  A continuación veremos un ejemplo del tratamiento de este tipo de condiciones, donde se introduce una sentencia `CREATE TABLE` y una consulta `SELECT`:

```
> CREATE TABLE Jugador (
        NIF NUMBER(4) CHECK (NIF > 0 AND NIF <= 20),
        Nombre VARCHAR(30) NOT NULL,
        Altura NUMBER(3, 2) CHECK (Altura >= 1.60),
        CIF_Club NUMBER(4)
    );

> SELECT Nombre FROM Jugador WHERE NIF >= 9;
```

En este caso, tenemos una condición de comparación entre un número y una columna cuyo tipo de dato también es numérico. Se compara *NIF* con el número 9, por lo que se generarán tres filas para esa columna con los valores que cumplen la igualdad y los valores fronterizos al umbral, concretamente los valores 9, 10 y 8, en este orden. Por lo tanto, el resultado sería el siguiente:

```
> INSERT INTO jugador VALUES (9, 'Juan Meneses', 1.83,
    6704);
> INSERT INTO jugador VALUES (10, 'Marta Moreno', 1.64,
    9913);
> INSERT INTO jugador VALUES (8, 'Alex Romero', 1.83,
    2602);
```

Además de los datos de la columna *NIF*, otra columna cuyos datos se ven afectados por una condición de comparación es *Altura*. Gracias a nuestros generadores de datos, somos capaces de insertar valores que cumplen con las restricciones en las columnas que no se ven afectadas por condiciones en las cláusulas `WHERE`. En este caso, todos los valores generados para la columna *Altura* cumplen la restricción de comprobación ($>= 1.60$).

- **Condiciones sobre cadenas de caracteres:** son aquellas que usan el operador `LIKE` y la función `LENGTH`.

  Para tratar las condiciones con el operador `LIKE`, primero generamos una cadena válida, es decir, que cumpla con el patrón dado. Para ello, se almacena el patrón con el que tiene que coincidir el valor final y, de este modo, cuando se llama a la función generadora de cadenas de caracteres (explicada en la sección 4.3.3) esta puede obtenerlo. Seguidamente, se genera un valor inválido para la condición. Para ello, tomamos el valor válido generado, iteramos sobre sus caracteres hasta encontrar uno que no sea un cómodin (_ y %, definidos en la sección 4.2) y generamos un caracter aleatorio en esa posición, de esta forma deja de coincidir el valor generado y el patrón y, por tanto, ya no es válido.

  Para manejar las condiciones que limitan la longitud de una cadena de caracteres, seguimos un proceso algo más sencillo. Primeramente, sabiendo las restricciones de tamaño determinadas por la definición de la columna, comprobamos que cuando sumamos o restamos 1 al tamaño determinado por la consulta `SELECT`, este sigue siendo válido dentro de la columna. Si no es así, se ajusta el parámetro para que sí sea válido. Seguidamente, se llama al generador de cadenas de caracteres modificando en cada llamada las restricciones de tamaño de forma podemos generar cadenas con los tamaños deseados.

5.3.2. Consultas con varias condiciones

Dentro de una misma consulta nos podemos encontrar con varias condiciones de búsqueda regidas por el operador `AND`. Para tratar con este tipo de sentencias, generamos los datos para cada columna que se vea afectada por una condición siguiendo la misma metodología que en el apartado anterior. Una vez generados todos estos datos se sigue un

sistema de permutaciones que explicamos a continuación.

El objetivo de este sistema es permutar los valores de dos o más columnas que se han generado a raíz de las condiciones especificadas en una restricción `WHERE`. Como hemos indicado anteriormente, los valores válidos de una columna para cada condición son los primeros en generarse. Para realizar las permutaciones, usamos la biblioteca integrada de Python *itertools* [13], que mediante la función *product()* realiza el producto cartesiano de los valores de las columnas involucradas. Cuando finaliza, la primera tupla de la lista de permutaciones contiene los valores válidos de cada columna para cada condición. Seguidamente, para aportar más aleatoriedad al orden de las sentencias de inserción de filas, mezclamos las tuplas generadas a partir de la segunda posición con la ayuda de la función *sample()* de la biblioteca *random*. Finalmente, se filtran las tuplas que impiden que se cumplan las restricciones de clave primaria o unicidad, ya que al hacer el producto cartesiano de los valores, estos se repiten en varias tuplas. Es por esta razón que la primera tupla en la lista de permutaciones no se mezcla con el resto, porque contiene todos los valores válidos y nos interesa mantenerla.

Vamos a aclarar el proceso con un ejemplo. Consideremos que tenemos los siguientes datos de entrada:

```
> CREATE TABLE Jugador (
    NIF NUMBER(4) PRIMARY KEY NOT NULL,
    Nombre VARCHAR(30) NOT NULL,
    Altura NUMBER(3, 2) CHECK (Altura >= 1 and Altura <= 3),
    FechaNacimiento DATE,
);

> SELECT * FROM Jugador WHERE NIF = 4356 AND
    FechaNacimiento > '31/12/1999';
```

Los valores que se van a generar para la columna `NIF` son: 4356, 4355 y 4357, en este orden. Los valores que se generan para la columna `FechaNacimiento` son: '01/01/2000', '31/12/1999' y '30/12/1999', en este orden. Las permutaciones que se generan con *itertools.product()* son:

```
[(4356, '01/01/2000'), (4356, '31/12/1999'),
(4356, '30/12/1999'), (4355, '01/01/2000'),
(4355, ''31/12/1999'), (4355, '30/12/1999'),
(4357, '01/01/2000'), (4357, '31/12/1999'),
(4357, '30/12/1999')]
```

Las permutaciones después de mezclarlas quedarían, por ejemplo, de esta forma:

```
[(4356, '01/01/2000'), (4355, '30/12/1999'),
(4357, '31/12/1999'), (4355, '31/12/1999'),
(4357, '01/01/2000'), (4355, '01/01/2000'),
(4356, '31/12/1999'), (4357, '30/12/1999'),
(4356, '30/12/1999')]
```

Con este resultado podemos comprobar que la primera tupla efectivamente contiene los valores que satisfacen la consulta `SELECT`. Por último, las filas que se insertarían serían las siguientes:

```
> INSERT INTO jugador VALUES (4356, 'Juan Meneses', 1.55,
    '01/01/2000');
> INSERT INTO jugador VALUES (4355, 'Marta Moreno ', 1.82,
    '01/01/2000');
> INSERT INTO jugador VALUES (4357, 'Alex Romero', 1.64,
    '01/01/2000');
```

Observamos que únicamente se crean tres filas y no las nueve resultantes de las permutaciones. Esto se debe a que una de las columnas de la condición en la cláusula `WHERE` (*NIF*) contiene una restricción de clave primaria, con lo cual no se pueden repetir valores dentro de la columna. El resto de filas se han descartado. En el caso de que no existiera ninguna restricción de unicidad o de clave primaria, se insertarían tantas filas como el número de tuplas que se hayan calculado.

5.4. Generación de filas en varias tablas

El último hito en el desarrollo del proyecto fue añadir la posibilidad de hacer consultas sobre varias tablas mediante una condición `JOIN` en la cláusula `FROM` de una consulta `SELECT`. La función de esta condición es combinar filas de dos o más tablas. Esto lo hace comparando dos columnas, cada una de una tabla diferente. Para ejecutar un `JOIN`, el sistema de gestión bases de datos combina un par de filas, una de cada tabla para el cual la condición `JOIN` se satisface.

Hay varios tipos de consultas `JOIN` para satisfacer diferentes necesidades del usuario. En esta versión del proyecto se soportan los `INNER JOIN` con una condición. Además, una de las columnas indicadas en la condición deberá tener una restricción de clave externa, lo cual resulta de utilidad puesto que garantiza la integridad referencial, es decir, solo se puede insertar una fila en la tabla secundaria si hay una fila correspondiente en la tabla primaria. Sin embargo, esto no es una limitación puesto que es el uso más común por un alumno que utiliza la herramienta LearnSQL. Para futuras versiones de este proyecto proponemos extender esta funcionalidad para abarcar más tipos de consultas `JOIN` como `OUTER JOINS`.

Un ejemplo de consulta utilizando los `INNER JOIN` con una condición sería el siguiente:

```
> CREATE TABLE Club(
    CIF NUMBER(4) PRIMARY KEY,
    Nombre_Club VARCHAR(50) NOT NULL,
    Sede VARCHAR(100),
    NumSocios NUMBER(5),
    NumAsientos NUMBER(5)
  );
  CREATE TABLE Jugador(
    NIF NUMBER(8) PRIMARY KEY,
    Nombre VARCHAR(30) NOT NULL,
    Altura NUMBER(3, 2),
```

```
    FechaNacimiento DATE,
    CIF_Club NUMBER(4) REFERENCES Club(CIF)
  );

> SELECT * FROM Club JOIN Jugador ON CIF = CIF_Club;
```

Esta consulta devolvería las filas de las tablas Club y Jugador donde el valor de la columna `CIF` es igual al valor de la columna `CIF_Club`.

Para desarrollar esta funcionalidad se va iterando sobre las tablas creadas siguiendo su orden topológico [1], rellenándolas con datos. De este modo, cuando nos encontramos con una tabla que posee una restricción de clave externa, los valores de la columna a la que hace referencia ya están creados. Por la forma en la que está definida la sintaxis de Oracle, una columna con una restricción `REFERENCES` debe contener un valor de la columna a la que referencia. Por este motivo, todos los datos que creemos para las consultas `JOIN` van a ser válidos.

Además, también soportamos consultas `JOIN` con cláusulas `WHERE`. Para estas consultas, generamos un valor que satisface la condición de la cláusula `WHERE` y uno o dos valores que no la satisface, siguiendo la metodología explicada en la sección (5.3). Si la condición se ejecuta sobre una columna que contiene una restricción de clave externa, la columna que posea esa restricción contendrá un mismo valor repetido tantas veces como valores se hayan creado para evaluar la condición (normalmente dos o tres).

Vamos a ilustrar la metodología con un ejemplo. Partimos de las siguientes tablas y una consulta:

```
> CREATE TABLE Club(
    CIF NUMBER(4) PRIMARY KEY,
    Nombre_Club VARCHAR(50) NOT NULL,
    Sede VARCHAR(100),
    NumSocios NUMBER(5),
    NumAsientos NUMBER(5)
  );
> CREATE TABLE Jugador(
    NIF NUMBER(8) PRIMARY KEY,
    Nombre VARCHAR(30) NOT NULL,
    Altura NUMBER(3, 2),
    FechaNacimiento DATE,
    CIF_Club NUMBER(4) REFERENCES Club(CIF)
  );

> SELECT * FROM Club JOIN Jugador ON CIF = CIF_Club WHERE
  Altura > 1.65;
```

[1] "Una ordenación topológica de un grafo $G = (V, E)$ es una ordenación lineal de todos sus vértices tal que si G contiene una arista (u, v), entonces u aparece antes que v en el ordenamiento." Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.4: Topological sort search, pp.465–466.

Con estos datos de entrada se generan 3 datos para evaluar la condición de la cláusula `WHERE`, 1.66, 1.65 y 1.64. Además, la columna *Altura* pertenece a la tabla *Jugador*, que posee una columna (*CIF_ Club*) que hace referencia a otra tabla (*Club*). Las filas a insertar serían las siguientes:

```
> INSERT INTO club VALUES (-2383, 'Direc', 'Despite owner plan
  decade scientist election under spend. Language there b',
  NULL, 68455);
```

Obviamos el resto de líneas que se insertan en la tabla *Club* ya que no son relevantes para este ejemplo.

Las filas generadas para la tabla *Jugador* serían:

```
> INSERT INTO jugador VALUES (-81826849, 'Sergio Pinilla', 1.66,
'02/07/1984', -2383);
> INSERT INTO jugador VALUES (-87277641, 'Marcos Palmero', 1.65,
'22/08/1991', -2383);
> INSERT INTO jugador VALUES (-87218353, 'Carmen Martinez', 1.64,
'16/05/1985', -2383);
```

Como se puede observar, los valores de la columna *CIF_ Club* de la tabla *Jugador* son el mismo, y los valores de la columna *Altura* evalúan la condición de la cláusula `WHERE` mediante un valor que la satisface (1.66) y otros dos que no la satisfacen (1.65 y 1.64).

Capítulo 6

Características de SQL soportadas y excluidas

A la hora de desarrollar un sistema de generación de tablas SQL, es necesario especificar cuáles son las diferentes características que va a soportar dicho sistema, así como las delimitaciones que se van a excluir para poder ceñirse lo máximo posible a los objetivos establecidos y desarrollar un proyecto funcional, que posea cierta consistencia y disponibilidad de cara a ser utilizado de manera óptima.

En este capítulo, presentaremos las diferentes propiedades del lenguaje SQL que soportamos así como las características excluidas en base a las decisiones de implementación tomadas.

6.1. Propiedades de SQL soportadas

En esta sección, se presentarán las diferentes características soportadas en el proyecto distinguidas en los diferentes campos del desarrollo.

- **Tipos de sentencias:** los tipos de sentencias soportadas son `CREATE TABLE` y `SELECT`.
- **Tipos de datos:** los tipos de datos soportados son `INT`, `INTEGER`, `SMALLINT`, `NUMBER`, `FLOAT`, `BINARY_FLOAT`, `BINARY_DOUBLE`, `NUMERIC`, `DECIMAL`, `DEC`, `REAL`, `CHAR`, `CHARACTER`, `VARCHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR`, `NVARCHAR2`, `LONG`, `DATE` y `TIMESTAMP`. Todos ellos han sido explicados con detalle en la sección 4.1
- **Restricciones de integridad:** soportamos las restricciones de nulidad y no nulidad (`NULL` y `NOT NULL`), de clave primaria (`PRIMARY KEY`), de unicidad (`UNIQUE`), de comprobación (CHECK) y de clave externa (`REFERENCES`), descritas en la sección 4.2.
- **Condiciones en restricciones de comprobación:** se aceptan operadores de comparación (=, != (ó <>), >, <, >=, <=) entre números y fechas. Respecto a las cadenas de caracteres se aceptan igualdades, desigualdades de tipo != (ó <>), comparaciones de tamaño mediante la función `LENGTH` y comparación de patrones con `LIKE`.

- **Tipos de declaración de restricciones de integridad:** manejamos las declaraciones *inline* en las que no se incluya una cláusula `CONSTRAINT`.
- **Cláusulas `WHERE`:** aceptamos los operadores de comparación (=, != (ó <>), >, <, >=, <=) para números, cadenas de caracteres y fechas, así como la condición `LIKE` y la función `LENGTH`.
- **Consultas `JOIN`:** se soportan las operaciones `JOIN` de tipo `INNER JOIN` cuya condición sea explícita mediante la cláusula `ON`, que acepta condiciones de igualdad (=).

6.2. Propiedades de SQL excluidas

En esta sección se realizará una descripción de las características excluidas en nuestro proyecto diferenciadas en los distintos ámbitos de implementación. Muchas de estas limitaciones se pueden considerar para un trabajo futuro posterior a la finalización del proyecto.

- **Tipos de datos:** hay una serie tipos de datos que no manejamos por dos motivos. El primero es que algunos de ellos no están soportados por la biblioteca `mo-sql-parsing` que usamos para analizar las sentencias SQL. Estos son:
  - `TIMESTAMP[(`*fractional_seconds*`)] WITH TIME ZONE`
  - `TIMESTAMP[(`*fractional_seconds)*`] WITH LOCAL TIME ZONE`,
  - `INTERVAL YEAR[(`*year_precision*`)] TO MONTH`
  - `INTERVAL DA[(`*day_precision*`)] TO SECOND[(`*fractional_seconds)]*
  - `CHARACTER VARYING(`*n*`)`
  - `CHAR VARYING(`*n*`)`
  - `NATIONAL CHARACTER(`*n*`)`
  - `NATIONAL CHAR(`*n*`)`
  - `NATIONAL CHARACTER VARYING(`*n*`)`
  - `NATIONAL CHAR VARYING(`*n*`)`
  - `NCHAR VARYING(`*n*`)`
  - `DOUBLE PRECISION`
  - `LONG VARCHAR`

  Como se puede observar, son todos aquellos tipos de datos que se definan con más de una palabra. El segundo motivo es que algunos tipos de datos quedan fuera del alcance del proyecto porque son escasamente usados en los ejercicios sobre bases de datos. Estos son: `RAW(`*size*`)`, `LONG RAW`, `ROWID`, `UROWID[(`*size*`)]`, `CLOB`, `NCLOB`, `BLOB` y `BFILE`. Todo esto se encuentra explicado en la sección 4.1.
- **Restricciones de integridad:** la única restricción de integridad que no manejamos es la restricción *REF* (explicado en la sección 4.2), debido a que hace referencia a objetos y eso se encuentra fuera del alcance de nuestro proyecto.
- **Condiciones en restricciones de comprobación:** no soportamos operadores aritméticos dentro de una restricción de comprobación. Por ejemplo:

```
CREATE TABLE Jugador (
    NIF NUMBER(8) PRIMARY KEY,
    Nombre VARCHAR2(40) NOT NULL,
    Altura NUMBER(3, 2) CHECK(Altura > 1.10 + 0.5)
);
```

Si bien es posible sintácticamente declarar la restricción de esta forma, hemos decidido no implementarlo en nuestro código por su uso poco frecuente y por tener una sintaxis equivalente que sí soportamos (poniendo directamente el resultado de la operación 1.10 + 0.5, si seguimos el ejemplo anterior).

Además, dentro de estas restricciones los operadores relacionales sobre cadenas de caracteres no son manejados debido a su poco uso. Por ejemplo:

```
CREATE TABLE Jugador (
    NIF NUMBER(8) PRIMARY KEY,
    Nombre VARCHAR2(40) CHECK (Nombre > 'Borja'),
    Altura NUMBER(3, 2)
)
```

- **Tipos de declaración de restricciones:** no aceptamos la declaración de restricciones de tipo *out-of-line* ni las de tipo *inline* cuya definición incluya la palabra clave `CONSTRAINT`. Esto último es debido a la falta de soporte de esta característica por parte de la librería `mo-sql-parsing`.
- **Manejo de valores nulos:** el soporte de valores nulos en el proyecto se basa únicamente en la generación de estos. No se han tenido en cuenta a la hora de hacer comparaciones con ellos.
- **Cláusulas `WHERE`:** no se manejan funciones SQL, excepto la función `LENGTH` para cadenas de caracteres. Los operadores aritméticos en las cláusulas `WHERE` no están soportados.
- **Consultas `JOIN`:** no se soportan las operaciones `JOIN` de tipo `OUTER JOIN`, `CROSS JOIN` y `NATURAL JOIN`. Además, la cláusula `USING` en las operaciones de tipo `INNER JOIN` estará excluido, pudiendo usar en su lugar únicamente la cláusula `ON`.
- **Alias:** en esta versión del proyecto no se admite el renombramiento de columnas o de tablas mediante alias.

Capítulo 7

Conclusiones y Trabajo Futuro

En base a los resultados obtenidos durante el desarrollo del proyecto, se han obtenido una serie de conclusiones y propuestas de trabajo futuro, las cuales presentaremos en este capítulo.

La primera conclusión obtenida del desarrollo de este trabajo es la importancia que tiene la generación de las tablas que prueban las consultas SQL de forma automática. Esta metodología automatizada para cada sentencia evita que los profesores tengan que introducir los casos de prueba de manera manual. Esto mejora considerablemente la eficiencia del sistema LearnSQL y ahorra el proceso laborioso por parte del profesor de pensar qué filas añadir a una tabla para que pruebe correctamente una consulta enviada por un estudiante.

Otra de las conclusiones obtenidas, es la utilidad de emplear metodologías *Open Source* y plataformas como Github para el desarrollo del proyecto, puesto que facilita en gran medida la supervisión del código fuente por parte de los desarrolladores o usuarios interesados en mejorar o dar una retroalimentación del trabajo bien sea durante su desarrollo o de cara a futuras implementaciones.

En la sección 1.2 se presentaron unos objetivos. Después de la finalización del proyecto podemos afirmar que hemos alcanzado con éxito el principal objetivo que consistía en desarrollar una biblioteca que, a partir de la definición de una o varias tablas SQL, genere las filas para probar de la manera más exhaustiva posible una determinada consulta SQL. Además, también hemos cumplido con el resto de objetivos planteados, que listaremos a continuación junto con las soluciones que hemos seguido para conseguirlos:

- Conocer la sintaxis de Oracle Database para poder hacer frente de forma correcta a la generación de tablas SQL y no incurrir en errores sintácticos.
  - Para ello estudiamos la documentación de Oracle (secciones 4.1 y 4.2).
- Ser capaces de analizar sentencias `CREATE TABLE` para identificar las restricciones que deberán cumplir los datos que se vayan a insertar, como las restricciones de unicidad, de clave primaria...
  - Este objetivo lo hemos cumplido con la ayuda de la biblioteca `mo-sql-parsing` y un conjunto de comprobaciones en nuestro código (secciones 5.1 y 5.2).

- Poder analizar consultas sobre la base de datos mediante `SELECT` de forma que se puedan tener en cuenta las condiciones que se indican en la consulta a la hora de generar los datos, como las condiciones de comparación o la unión de tablas mediante `JOIN`.
    - Este objetivo lo hemos alcanzado de forma similar al anterior, mediante la biblioteca `mo-sql-parsing` y una serie de comprobaciones propias (sección 5.1).
- Dada una o varias sentencias `CREATE TABLE` analizadas, ser capaces de restringir la generación aleatoria de datos a las características de la o las tablas.
    - El objetivo se ha satisfecho mediante el desarrollo de funciones generadoras de datos que, dadas unas restricciones, devuelven un dato que las cumple (sección 4.3).
- Dada una consulta `SELECT` insertada por el alumno, generar datos fronterizos a las condiciones que se requieren en una cláusula `WHERE` de la consulta.
    - Objetivo logrado mediante un análisis exhaustivo de esas condiciones y la generación de datos fronterizos en base a ellas (sección 5.3).
- Producir un código limpio y mantenible para que cualquier persona pueda proponer mejoras de cara a versiones futuras.
    - Objetivo cumplido. Se han documentado las funciones del código y se han estructurado de manera legible e intuitiva.

A pesar de haber cumplido con los objetivos, en la versión actual de nuestro proyecto no se implementan algunas funcionalidades. Eso se debe a que el lenguaje SQL es muy extenso y no es posible abarcar todo su contenido en este trabajo de fin de grado. Por esta razón lo hemos tenido que limitar a aquello que hemos considerado más importante. Además, esto no supone un gran impacto en su aplicación a LearnSQL ya que la solución actual sigue siendo ejecutable.

7.1. Trabajo futuro

La versión actual de la biblioteca que hemos desarrollado ofrece un funcionamiento correcto usando una sintaxis básica. A continuación, se listan una serie de líneas de trabajo por las que continuar trabajando para mejorar la funcionalidad de la biblioteca.

- Adaptar el código para que permita el análisis de tablas con declaración de restricciones de integridad *out-of-line* (explicado en capítulo 4.2).
- Aprovechar las funcionalidades de la biblioteca *Faker* para producir cadenas de texto que hagan referencia al nombre de la columna (e-mail, dirección, nombre de persona...). Tal y como utilizamos la biblioteca mencionada ahora mismo, si tenemos una columna cuyo nombre es *Dirección* nuesto código generaría, por ejemplo, la cadena de texto *'Allow view page drop quite relate. Most '*. Es decir, palabras aleatorias generadas con la función text()[1] de *Faker*. Con esta propuesta, mediante la función

[1]Documentación en `https://faker.readthedocs.io/en/master/locales/la.html#faker.providers.lorem.la.Provider.text`

address()[2], la cadena generada podría ser, por ejemplo, *'Callejón Virginia Collado 21. Lugo, 37687'*.

- Manejar las funciones de agregación en las consultas `SELECT` como `AVG`, `COUNT`, `MAX`, `MIN`... También la función `ORDER BY`. Para ello se el código debería ser capaz de identificarlas y generar datos que cumplan con la definición de cada función.

- Desarrollar una función que, dado un operador no soportado, itere un número finito de veces generando en cada iteración un dato totalmente aleatorio, o que cumpla las restricciones y condiciones que sí se han podido analizar. Después de generar el dato debería comprobarse mediante una función como eval()[3], que la condición del `SELECT` se cumple. Por ejemplo, teniendo la consulta:

```
SELECT * FROM Club WHERE NumAsientos > 100 * 2;
```

  En este ejemplo, donde la operación 100 * 2 no está soportada, se ejecutaría la función propuesta generando datos aleatorios y mediante eval() podríamos comprobar si se satisface la condición o no de este modo:

```
NumAsientos = 98  # Dato generado aleatoriamente
eval("NumAsientos > 100 * 2")
```

  Si llegara al número máximo de iteraciones sin generar un número que satisfaga la condición, se lanzaría una excepción.

- Para poder personalizar más una consulta, a veces es necesario el uso de subconsultas o consultas anidadas del estilo:

```
SELECT * FROM Jugador WHERE GolesMarcados > (SELECT
    GolesMarcados FROM Jugador WHERE Nombre = "Juana");
```

  Proponemos que en futuras versiones del proyecto se tengan en cuenta estas subconsultas para aumentar la funcionalidad de la biblioteca.

- Es muy común el uso de alias que hacen referencia a tablas y que facilitan la lectura o que permiten diferenciar entre columnas de diferentes, como por ejemplo:

```
SELECT * FROM Club c JOIN Jugador j ON c.CIF = j.CIF
    ORDER BY j.Nombre;
```

  Planteamos que en próximas versiones del proyecto se detecten estos alias y se tengan en cuenta a la hora de localizar las tablas.

- Por último, proponemos que se integre algún sistema automático de *testing* para probar el código. Recomendamos las pruebas basadas en propiedades más que los tests unitarios debido al factor aleatorio que caracteriza a nuestra biblioteca. Para este fin sugerimos el uso biblioteca de Python *Hypothesis*[4]. Por ejemplo, si tenemos la siguiente tabla:

```
CREATE TABLE Club(NumAsientos NUMBER(4) CHECK (
        NumAsientos > 0));
```

[2]Documentación en `https://faker.readthedocs.io/en/master/locales/es_ES.html#faker-providers-address`

[3]Documentación en `https://docs.python.org/3/library/functions.html#eval`

[4]Documentación en https://hypothesis.readthedocs.io/en/latest/

Mediante el sistema automático de *testing*, al generar multitud de valores se probaría que todos ellos fueran mayores que 0.

Chapter 8

Conclusions and Future Work

Based on the results obtained during the development of the project, a series of conclusions and proposals for future work have been obtained, which we will detail in this chapter.

The first conclusion obtained from the development of this work is the importance of the generation of tables that test SQL queries automatically. This automated methodology for each statement avoids teachers having to insert test cases manually. This greatly improves the efficiency of the LearnSQL system and saves the teachers the laborious process of thinking about which rows to add to a table to correctly test a query submitted by a student.

Another of the conclusions obtained is the usefulness of Open Source methodologies and platforms such as Github for the development of the project, since it greatly facilitates the supervision of the source code by developers or users interested in improving or giving feedback on the work either during its development or for future implementations.

In Section 1.2 goals were presented. After the completion of the project we can state that we have successfully achieved the main objective which was to develop a library that, given the definition of one or more SQL tables, generates rows to test as exhaustively as possible a given SQL query. In addition, we have also fulfilled the rest of the objectives, which we will list below together with the solutions we have followed to achieve them:

- Get to know the Oracle Database syntax in order to be able to correctly deal with the generation of SQL tables without making any syntactic errors.
    - For this purpose we studied the Oracle Database documentation (Sections 4.1 and 4.2).
- To be able to parse `CREATE TABLE` sentences to identify the constraints to be met by the data to be inserted, such as unique or primary key constraints.
    - We have accomplished this goal with the help of the `mo-sql-parsing` library and a set of evaluations in our code. (Sections 5.1 and 5.2).
- To be able to parse `SELECT` queries so that the conditions indicated in the query can be taken into account when generating the data, such as the comparison conditions or the joining of tables using `JOIN`.

    - This objective has been achieved in a similar way to the previous one, by means of the `mo-sql-parsing` library and a series of validations (Section 5.1).

- Given one or more parsed `CREATE TABLE` statements, to be able to restrict the random generation of data to the characteristics of the table(s).

    - The objective has been satisfied by developing data generator functions that, given a set of constraints, return data that satisfies them (Section 4.3).

- Given a `SELECT` query inserted by the student, generate boundary data to the conditions required in a `WHERE` clause of the query.

    - Objective achieved by a thorough analysis of those conditions and the generation of boundary data based on them (Section 5.3).

- Produce clean and maintainable code so that anyone can propose improvements for future versions.

    - Objective achieved. The functions of the code have been documented and structured in a readable and intuitive way.

Despite having met the objectives, some functionalities are not implemented in the current version of our project. This is due to the fact that the SQL language is very extensive and it is not possible to cover all of its content in this final degree project. For this reason, we have had to limit it to what we have considered to be the most important things. Moreover, this does not have a great impact on its application to LearnSQL, as the current solution is still executable.

8.1. Future work

The current version of the library we have developed offers correct operation using a basic syntax. Listed below are a number of lines of work along which one can continue to work to improve the functionality of the library.

- Adapt the code to allow the analysis of tables with *out-of-line* definition of integrity constraints (explained in Chapter 4.2).

- Take advantage of the functionalities of the *Faker* library to produce text strings that refer to the name of the column (e-mail, address, person's name...). As we use the aforementioned library right now, if we have a column whose name is *Address* our code would generate, for example, the text string *'Allow view page drop quite relate. Most '*. That is, random words generated with the *Faker* text()[1] function. With this proposal, using the function address()[2], the generated string could be, for example, *'Callejón Virginia Collado 21, 37687'*.

- Handle aggregate functions in `SELECT` queries such as `AVG`, `COUNT`, `MAX`, `MIN`... Also the function `ORDER BY`. To do this, the code should be able to identify them and generate data that complies with the definition of each function.

[1] Documentation in `https://faker.readthedocs.io/en/master/locales/la.html#faker.providers.lorem.la.Provider.text`

[2] Documentation in `https://faker.readthedocs.io/en/master/locales/es_ES.html#faker-providers-address`

- Develop a function that, given an unsupported operator, iterates a finite number of times generating in each iteration a totally random value, or one that fulfils the constraints and conditions that have been previously analysed. After generating the data, a function such as eval()[3] should be used to check that the conditions in the `SELECT` query is satisfied. For example, having the query:

  ```
  SELECT * FROM Club WHERE NumSeats > 100 * 2;
  ```

  In this example, where the 100 * 2 operation is not supported, the function would be executed generating random data and with eval() we could check if the condition is satisfied or not this way:

  ```
  NumSeats = 98  # Randomly generated value
  eval("NumSeats > 100 * 2")
  ```

  If it were to reach the maximum number of iterations without generating a number that satisfies the condition, an exception would be thrown.

- In order to further customise a query, it is sometimes necessary to use sub-queries or nested queries like:

  ```
  SELECT * FROM Player WHERE ScoredGoals > (SELECT
      ScoredGoals FROM Player WHERE Name = "Juana");
  ```

  We propose that future versions of the project take these sub-queries into account in order to increase the functionality of the library.

- It is very common to use aliases that refer to tables and that make it easier to read or to differentiate between different columns, for example:

  ```
  SELECT * FROM Club c JOIN Player p ON c.CIF = p.CIF
      ORDER BY p.Name;
  ```

  We suggest that in future versions of the project these aliases are detected and taken into account when localising the tables.

- Lastly, we suggest that some automatic testing system is integrated to test the code. We recommend property-based testing rather than unit tests due to the random factor that characterises our library. For this purpose we suggest the use of Python's library *Hypothesis*[4]. Using this type of testing, if we have, for example, the following table:

  ```
  CREATE TABLE Club(NumSeats NUMBER(4) CHECK (
           NumSeats > 0));
  ```

  By means of the automatic testing system, when generating a multitude of values, all of them would be tested to be greater than 0.

[3]Documentation in `https://docs.python.org/3/library/functions.html#eval`
[4]Documentation in https://hypothesis.readthedocs.io/en/latest/

Capítulo 9

Contribuciones Personales

En este capítulo describiremos las contribuciones personales al proyecto. Al ser un equipo de dos participantes, la mayoría de tareas se han realizado conjuntamente. A continuación, se detallan en profundidad estas aportaciones.

9.1. Maria de Lluc Bonet Seguí

Investigación

A finales de septiembre empezamos con la primera parte del trabajo que consistía en formarnos en las diferentes herramientas que íbamos a utilizar durante el transcurso del proyecto. Para poder realizarlo de manera correcta y eficaz debíamos partir de unos conocimientos básicos. Empecé buscando información y documentándome sobre la base de datos de Oracle, su sintaxis y sus tipos de datos. Para lograr un mayor entendimiento tuve que, previamente, estudiar la notación BNF con sus correspondientes diagramas del ferrocarril ya que la documentación de Oracle incluye multitud de estos.

Como nunca antes había utilizado el lenguaje de programación Python, antes de empezar a desarrollar el código para crear la biblioteca tuve que aprender su sintaxis. Para ello, estudié su documentación y realicé algunos cursos online gratuitos que me ayudaron a afianzar los conceptos.

El útimo paso antes de empezar con la implementación fue realizar una labor de investigación para encontrar una biblioteca Python que analizara sentencias SQL. Tras una búsqueda exhaustiva en la página web de PyPI (Índice de Paquetes de Python), contribuí a la hora de hacer pruebas con algunas de las bibliotecas que más nos llamaron la atención como `mo-sql-parsing`[8], `sql-parse`[9] y `sql-metadata`[11] para así poder elegir finalmente cuál nos iba a ser más útil.

Implementación

Desde finales de octubre hasta principios de mayo estuvimos desarrollando nuestra biblioteca. Al ser un proyecto con únicamente dos integrantes, las tareas se han ido realizando conjuntamente, ayudándonos mutuamente. A continuación, procederé a explicar las tareas que he ido desarrollando junto con mi compañero.

La curva de aprendizaje al inicio era bastante elevada debido a que el código partía de cero, sin muchas referencias y además nunca había programado en Python. La primera tarea consistió en desarrollar generadores de datos para números, cadenas de caracteres y fechas (explicados en la sección 4.3). Yo me encargué de los dos primeros. Para ello fui de menos a más, primero generando valores numéricos sin ninguna restricción y luego creando funciones que clasificaran los atributos de una tabla como el tipo de dato de las columnas y sus restricciones asociadas para así poder generar valores a partir de ellos y conseguir un generador funcional. Tras desarrollar el generador de números, adapté ese código para que funcionara también con cadenas de caracteres, incluyendo más funcionalidades en el clasificador previo a la generación de valores como el manejo de la función `LENGTH` o el operador `LIKE`.

Tras implementar generadores de datos funcionales, la siguiente tarea consistió en generar filas para una tabla a partir de una consulta `SELECT` sencilla. Para ello, tuve que adaptar el código para que aceptara dichas consultas y las analizara y, de este modo, poder tenerlas en cuenta a la hora de generar nuevos datos. Más adelante, amplié el funcionamiento a consultas más complejas que aceptaran una o varias condiciones. Esto resultó ser una de las tareas más complicadas con las que me encontré ya que tuve que pensar cuál era la mejor forma de generar datos frontera al valor de una condición, así como diseñar e implementar un sistema de permutaciones que combinara los resultados de varias condiciones.

Después de obtener los resultados deseados al insertar filas para una única tabla, la última tarea consistió en adaptar todo el código para que siguiera funcionando cuando se introdujeran dos o más tablas. Para este fin, ajusté las funciones que clasifican los datos de entrada para que analizaran más palabras clave, propias de la conjunción de tablas en las consultas, como `JOIN`.

Finalmente, dediqué la última semana a revisar el código, eliminando comentarios o código innecesario y comprobando que no hubiera ningún error en la implementación. De esta forma completaría los objetivos propuestos.

Memoria

Para la escritura de la memoria decidimos dividirnos el trabajo en las partes comunes. De este modo, me he encargado de redactar el capítulo 4 explicando los tipos de datos soportados y describiendo cómo generamos aleatoriamente los datos. También me he encargado de escribir el apartado de trabajo futuro en el capítulo 7, donde se proponen una serie de mejoras de cara a futuras versiones de nuestra biblioteca. Además, he contribuido en la escritura del capítulo 5 donde explicamos la generación de filas y el capítulo 6 donde dejamos claro cuáles son las características de nuestra biblioteca. Asimismo, he cooperado en la redacción de los objetivos y conclusiones y de la bibliografía.

Por último me he encargado de traducir al inglés los capítulos de introducción y conclusiones y trabajo futuro así como el resumen y además he revisado la memoria de forma global para detectar y corregir errores ortográficos, gramaticales y de comprensión.

9.2. Álvaro Plaza Sanz

Investigación

Una vez comenzado este proyecto en el mes de septiembre, ambos integrantes del grupo tuvimos una primera reunión con nuestro tutor para asignar las tareas iniciales relacionadas con esta primera fase de investigación y toma de contacto con las herramientas y conocimientos en los que profundizar.

En esta primera fase del proyecto, mi labor principal fue la búsqueda de diferentes bibliotecas de Python y su respectiva documentación que sirvieran de utilidad de cara a analizar sentencias SQL y obtener un formato legible y fácil de tratar. Para ello, realicé una busqueda exhaustiva de las diferentes bibliotecas de utilidad y realicé diferentes pruebas hasta encontrar finalmente aquellas que serían apropiadas para el proyecto. Entre las bibliotecas de Python encontradas se encontraban `sql-parse` [9], `pyparsing` [10], `sql-metadata` [11] o `mo-sql-parsing` [8], bilbioteca utilizada finalmente en el proyecto.

En este primer marco temporal del proyecto, tuve que realizar una labor de aprendizaje y afianzamiento con las herramientas y programas que se iban a utilizar y con la sintaxis de SQL y el lenguaje de Python. Para ello, fue necesario que me documentara sobre la base de datos de Oracle y los diagramas en notación BNF que definen su sintaxis. Al principio, la curva de aprendizaje fue bastante elevada puesto que nunca habia trabajado con las herramientas y entornos de programación propuestos, pero de manera progresiva fui comprendiendo el funcionamiento del sistema a desarrollar y gran parte de la documentación y pude consolidar los conocimientos sobre el lenguaje de programación de Pyhton realizando cursos formativos y ejercicios prácticos.

Además, en esta primera fase del proyecto, mi labor también fue la recopilación de un listado de los tipos de datos soportados por Oracle, agrupándolos en bloques según su tipología. De igual manera, creé un repositorio de GitHub para ir subiendo el código fuente y las diferentes implementaciones durante el transcurso del proyecto.

Implementación

Después de completar la fase previa de investigación y documentación, se comenzaron a llevar acabo los primeros pasos en el desarrollo del código para implementar nuestra biblioteca. Al tratarse de un proyecto conformado únicamente por dos desarrolladores, la mayoría de las funcionalidades han sido realizadas de manera conjunta y retroalimentadas de manera continua durante todo el desarrollo por ambos integrantes. Seguidamente procederé a detallar las funcionalidades que se han ido desarrollando junto a mi compañera.

Mi labor comenzó realizando implementaciones para los generadores de datos, más específicamente mi labor fue desarrollar los generadores de datos de tipo numérico, distinguiendo entre números en coma fija y números en coma flotante y más tarde los generadores de fechas, distinguiendo entre fechas en formato `DATE` y en formato `TIMESTAMP`; ambos tipos de datos están detallados en la sección 4.1. Una vez desarrollados los generadores para los datos numéricos y fechas, implementé una mejora en los generadores para poder clasificarlos según su tipología y generar datos con determinadas restricciones.

Después de conseguir poner en marcha los distintos tipos de generadores de datos, la siguiente tarea consistía en ser capaces de generar diferentes filas para una única tabla y con una única consulta sencilla. Por tanto, para poder analizar las sentencias entrantes, tuve que crear un código bien estructurado que fuera capaz de acceder a los diferentes parámetros y restricciones de las sentencias `CREATE TABLE` y consultas `SELECT`. Para ello, mi labor fue la creación de diferentes funciones que, teniendo en cuenta las diferentes restricciones de cada tipo de dato soportado (explicadas en el capítulo 5), fueran capaces de generar filas para una tabla con una única condición.

Tras conseguir desarrollar un código capaz de generar filas de datos para una consulta con una única condición, mi siguiente tarea fue extender esta funcionalidad para poder generar filas en consultas más complejas con una o varias condiciones. Para ampliar esta funcionalidad, mi desempeñó consisitió en crear funciones que tuvieran en cuenta las comparaciones entre los tipos de datos y las restricciones de cada consulta.

Una vez conseguimos tener un modelo funcional de la biblioteca, capaz de generar filas para una tabla con una o varias condiciones, la última tarea consisitió en extender esta funcionalidad a varias tablas. Para ello, mi tarea en este ámbito del proyecto se basó en la reestructuración de las funciones clasificadoras para poder incluir consultas de tipo `JOIN`. Para poder realizar este trabajo, mi labor se centró en conseguir que las funciones clasificadoras sean capaces de comparar dos columnas de dos o más tablas.

Una vez finalizadas todas las funcionalidades propuestas como objetivo, llevé acabo una limpieza del código fuente, comentando las diferentes funciones y detallando la función de sus parámetros de entrada y de salida.

Memoria

En el último marco temporal de este proyecto, se pasó a la realización de la memoria y toda la documentación respectiva al proyecto, ambos integrantes del grupo decidimos dividir equitativamente los diferentes capítulos a redactar.

En este último tramo del proyecto, mi contribución ha sido la elaboración del capítulo 3, donde se explican todos los conceptos necesarios para comprender este trabajo así como las tecnologías y herramientas utilizadas, el capítulo 1, donde se detalla de manera introductoria nuestas motivaciones y objetivos así como nuestro plan de trabajo seguido durante todo el desarrollo del proyecto y el capítulo 5, donde se explica todo el proceso metódico de generación de filas.

También he contribuido a la redacción del capítulo 6, donde explicamos las propiedades del lenguaje SQL que soportamos así como las características excluidas y a la redacción del capítulo 7 de conclusiones y trabajo futuro.

Bibliografía

[1] Sheila Moor, Eric Belden. Oracle Database PL/SQL Language Reference, 11g Release 2 (11.2). E25519-13. Diciembre 2014. Disponible en https://docs.oracle.com/cd/E11882_01/appdev.112/e25519/title.htm

[2] Ramez Elmasri, Shamkant Navathe. Fundamentos de Sistemas de Bases de Datos (5a Ed). Addison-Wesley, 2007. Mayo 2007.

[3] Oracle Database. https://www.oracle.com/es/database/ Accedido: 2022-05-27.

[4] Mary Beth Roeser. Database SQL Language Reference. https://docs.oracle.com/database/121/SQLRF/toc.htm Accedido: 2022-05-27.

[5] Oracle Live SQL. https://livesql.oracle.com/. Accedido: 2022-05-27.

[6] Jesús Correas, Enrique Martín, Manuel Montenegro, Adrián Riesco, Rubén Rubio. LearnSQL: un juez para el aprendizaje de las bases de datos. Luis Hernández Yáñez, Jornada Aprendizaje Eficaz con TIC en la UCM, Enero 2022. Ediciones Complutense, 978-84-669-3754-2. Disponible en: https://eprints.ucm.es/id/eprint/69766/.

[7] BNF and syntax diagrams. http://theteacher.info/index.php/computing-principles-01/software-development/1-2-2-applications-generation/3683-bnf-and-syntax-diagrams. Accedido: 2022-05-27.

[8] Kyle Lahnakoksi. Mo-sql-parsing. https://github.com/klahnakoski/mo-sql-parsing. Accedido: 2022-05-27.

[9] Christian Clauss, Andi Albrecht. Python-sqlparse. https://github.com/andialbrecht/sqlparse. Accedido: 2022-05-27.

[10] Paul McGuire. PyParsing – A Python Parsing Module. https://pypi.org/project/pyparsing/. Accedido: 2022-05-27.

[11] Maciej Brencz. Sql-metadata. https://github.com/macbre/sql-metadata. Accedido: 2022-05-27.

[12] Python Software Foundation. Random — Generate pseudo-random numbers. https://docs.python.org/3/library/random.html. Accedido: 2022-05-27.

[13] Python Software Foundation. Itertools — Functions creating iterators for efficient looping. https://docs.python.org/3/library/itertools.html. Accedido: 2022-05-27.