Marco de Testing Metamórfico para Consultas SQL con Valores Nulos
Metamorphic Testing Framework for SQL Queries with Null Values

Trabajo de Fin de Máster
Curso 2019–2020

Autor
Gonzalo Machado Salazar
Director
Mercedes García Merayo
Jesús Correas Fernández

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Marco de Testing Metamórfico para Consultas SQL con Valores Nulos
Metamorphic Testing Framework for SQL Queries with Null Values

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

Autor
Gonzalo Machado Salazar

Director
Mercedes García Merayo
Jesús Correas Fernández

Convocatoria: *Septiembre 2020*
Calificación: *9.0*

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

29 de septiembre de 2020

Autorización de difusión

El abajo firmante, matriculado en el Máster en Ingeniería en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Marco de Testing Metamórfico para Consultas SQL con Valores Nulos", realizado durante el curso académico 2019-2020 bajo la dirección de Mercedes García Merayo y Jesús Correas Fernández en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Gonzalo Machado Salazar

29 de septiembre de 2020

Dedicatoria

Dedicado al pilar fundamental de mi vida, mi mamá. Por sus enseñanzas, su dedicación, su esfuerzo, su enorme sacrificio para poder brindarme lo mejor y su apoyo incondicional a lo largo de los años. Gracias a ti he logrado alcanzar una meta más. Te amo.

Agradecimientos

Quiero agradecerle a todas las personas que me acompañaron durante estos dos años de Máster, que creyeron y me apoyaron durante esta etapa.

A mis tutores Jesús Correas Fernández y Mercedes García Merayo por todo el apoyo brindado durante el desarrollo de este Trabajo de Fin de Master a lo largo de estos meses.

A mis amigos Andrés Gerstl, Malena Rojas y Anibal Lozano que a pesar de estar separados geográficamente, estuvieron presentes durante el desarrollo de este trabajo y fueron de gran ayuda para afianzar mis conocimientos como lo han hecho desde el Grado.

A Santiago Bermúdez y Gabriel Sellès compañeros de la UCM que me acompañaron a lo largo de este nuevo reto que fue el Máster y que hoy los considero parte de mis amigos más cercanos.

A mis mejores amigos Francia Carrillo y Rubén Márquez por estar disponibles para mi las 24 horas del día y por ser ese apoyo que me impulsó en los momentos que más lo necesité.

Resumen

La falta de información dentro de las bases de datos relacionales expresada mediante valores nulos presenta un problema a la hora de garantizar la calidad de los datos y de las consultas SQL sobre esos datos. Esto ocurre debido a que existen múltiples interpretaciones de los valores nulos y en muchos casos las consultas no consideran que se puedan producir valores nulos como resultado de su evaluación, o bien no reflejan en el código la interpretación correcta de dichos valores. Los valores nulos han estado presentes en las bases de datos desde prácticamente las primeras implementaciones de sistemas de gestión de bases de datos relacionales, pero la implementación del estándar SQL plantea múltiples problemas. Al realizar consultas sobre una base de datos que maneja valores nulos, los resultados pueden no ser los esperados, ya sea por omisión de resultados (falsos negativos) o por resultados incorrectos (falsos positivos).

Por esta razón, en este trabajo se propone una herramienta que analice diferentes consultas SQL y permita al desarrollador detectar posibles errores en aquellas consultas que tengan valores nulos utilizando un marco de testing metamórfico. Después de estudiar la bibliografía relacionada sobre pruebas de bases de datos, esta parece ser la primera propuesta que aplica relaciones metamórficas a consultas SQL con valores nulos.

Palabras clave

Pruebas metamórficas, Pruebas de Bases de Datos, Consultas SQL, valores nulos.

Abstract

The lack of information within relational databases expressed by null values poses important problems when trying to ensure the quality of the data and of the SQL queries evaluated on that data. This occurs because there are multiple interpretations of null values and in many cases the queries either do not consider that null values can occur as a result of their evaluation, or they do not reflect the correct interpretation of these values in the code. Null values have been present in databases since the first implementations of relational database management systems, but the implementation of the SQL standard can generate multiple problems. When querying a database that handles null values, the results may not be produced as expected, either due to the omission of results (false negatives) or incorrect results (false positives).

For this reason, this work proposes a tool that analyzes different SQL queries and allows the developer to detect possible errors in those queries that have null values using a metamorphic testing framework. After studying the related literature on database testing, this appears to be the first proposal of the application of metamorphic relationships to SQL queries on null values.

Keywords

Metamorphic Testing, Database Testing, SQL Queries, null values.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Resumen: Este capítulo introduce las motivaciones fundamentales para la realización de este trabajo, así como los objetivos del mismo. Además, incluye de forma esquemática el plan de trabajo seguido para la realización de este Trabajo de Fin de Máster y la estructura general del resto de este documento.

1.1. Motivación

Desde la introducción del Modelo Relacional de Codd en 1970, las bases de datos relacionales basadas en SQL se han convertido en la tecnología más relevante para la representación, almacenamiento de datos y recuperación de información en la industria. Las sucesivas ampliaciones del modelo han permitido resolver complejos problemas técnicos relacionados con diversos aspectos como concurrencia de transacciones, distribución, etc. Uno de los mecanismos que se ha utilizado desde casi el principio ha sido la posibilidad de introducir información incompleta: determinados datos de la base de datos que, o bien son desconocidos o no disponibles en el momento en el que se introduce la información en la base de datos, o no aplicables en el contexto particular correspondiente. La forma de representar esta información incompleta ha sido mediante valores nulos. Sin embargo, a pesar de que las bases de datos relacionales tienen una gran capacidad expresiva para representar la estructura de los datos y realizar consultas complejas sobre su contenido de forma compacta

y robusta, esta forma de representar la información incompleta puede afectar a la calidad de dichos datos y los resultados de las consultas. Aunque la completitud de los datos es un aspecto esencial de la calidad de los mismos, ya que en muchos escenarios es crucial garantizar la completitud de las respuestas de las consultas, es natural no contar siempre con toda la información necesaria. Por lo general, se supone que una base de datos está completa, y en general los desarrolladores de aplicaciones asumen este hecho. Por ejemplo, en una base de datos relacional se considera que cada relación contiene todas las tuplas que necesitan aparecer en la relación. Sin embargo, hay situaciones en las que la información puede ser parcial, es decir, pueden faltar algunas tuplas y se dice que la base de datos está incompleta. Los datos pueden estar incompletos de dos maneras: los registros pueden faltar como un todo o los valores de atributo de un registro pueden estar ausentes, indicados por un valor NULL. Este es utilizado para representar un valor desconocido o la falta del mismo. Es importante destacar que NULL es diferente de un valor 0. Si la base de datos es parcial o incompleta, es necesario reconsiderar el significado del resultado de las consultas sobre la base de datos y analizar si se garantiza que una respuesta a una consulta determinada sea completa incluso en presencia de valores nulos. (Levy, 1996; Nutt et al., 2012)

Este trabajo se centra en la definición de un marco de testing metamórfico con la finalidad de identificar posibles errores en los resultados obtenidos de la ejecución de consultas SQL que pueden tratar con valores NULL. Esta técnica de testing se utiliza en aquellas situaciones en las que no se dispone de un oráculo que proporcione los resultados esperados de la ejecución de los tests y se basa en relaciones definidas entre los tests y los resultados obtenidos de la aplicación de los mismos. Hasta la fecha, existen trabajos enfocados en la realización de testing para el lenguaje SQL y las bases de datos, sin embargo no tenemos constancia de ninguna implementación de un marco de testing metamórfico aplicado a bases de datos con valores nulos.

1.2. Objetivos

El objetivo fundamental del presente trabajo es desarrollar e implementar el prototipo de una herramienta web que permita detectar posibles errores en consultas SQL con valores nulos utilizando relaciones metamórficas y de este modo poder realizar pruebas y verificar los resultados automáticamente. Las relaciones metamórficas se basaran en transformaciones de fragmentos de código de las consultas originales para la comparación de los resultados de la ejecución de diferentes sentencias SELECT de modo que ayuden a determinar de que manera los posibles valores NULL afectan a dichos resultados.

Los objetivos específicos de este trabajo son los siguientes:

- Estudiar la bibliografía relacionada con el testing de bases de datos y su relación con bases de datos incompletas.
- Analizar las distintas interpretaciones de valores nulos y los problemas que producen las bases de datos incompletas en el diseño de consultas complejas.
- Evaluar la aplicabilidad de la técnica de testing metamórfico para plantear posibles transformaciones en las que el desarrollador pueda identificar errores en consultas SQL relacionadas con valores nulos.
- Plantear esquemas de relaciones metamórficas que permitan automatizar el testing metamórfico de consultas SQL.
- Construir un prototipo que nos permita evaluar la viabilidad de esta técnica.
- Evaluar los resultados obtenidos con el prototipo desarrollado.

1.3. Plan de trabajo

Para la elaboración de este trabajo se identificaron inicialmente las tareas a realizar y estas fueron divididas en las siguientes fases.

- Fase 0: Evaluación del problema.

- Fase 1: Investigación y revisión de trabajos previos.
- Fase 2: Análisis de relaciones metamórficas.
- Fase 3: Evaluación de la aplicabilidad de testing metamórfico sobre bases de datos incompletas.
- Fase 4: Planteamiento de relaciones metamórficas.
- Fase 5: Definición de plantillas de relaciones metamórficas.
- Fase 6: Diseño e implementación de los módulos que componen el prototipo.
- Fase 7: Evaluación del prototipo.

Adicionalmente, para la supervisión y gestión de este trabajo se plantea la realización de reuniones semanales para mantener un orden en el proyecto y estar al corriente de lo que está ocurriendo. Estas reuniones también servirán para hacer revisiones continuas y así poder identificar problemas durante su desarrollo, establecer si se han alcanzado los objetivos de la fase y planificar pautas para continuar con el desarrollo.

1.4. Estructura del Documento

El esquema del presente trabajo se encuentra organizado en 7 capítulos, cuyo contenido se describe a continuación:

- Capítulo 1 - Introducción: Expone la motivación del trabajo realizado junto a los objetivos planteados al inicio del mismo.
- Capítulo 2 - Preliminares: Expone los fundamentos teóricos que dan lugar al sistema propuesto. Incluye los conceptos básicos utilizados para el entendimiento de este trabajo.
- Capítulo 3 - Marco de Testing Metamórfico: Es el capítulo central, en el que se presenta la propuesta desarrollada.

- Capítulo 4 - Implementación: Describe de forma detallada el proceso de desarrollo de la herramienta, las distintas operaciones que realiza y sus principales usos.

- Capítulo 5 - Evaluación: Se presenta un caso de estudio para la evaluación de la herramienta.

- Capítulo 6 - Manual de Usuario: Detalla el proceso de instalación y uso de la herramienta.

- Capítulo 7 - Conclusiones y Trabajo Futuro: Expone las conclusiones que se pueden extraer sobre el trabajo realizado una vez se ha implementado la solución propuesta. Adicionalmente, se indican posibles modificaciones sobre la herramienta que podrían mejorar su funcionamiento.

Capítulo 2

Preliminares

Resumen: En este capítulo se describen los fundamentos teóricos y conceptos básicos utilizados para el entendimiento de este trabajo.

2.1. Testing de Software

El *testing* de software se define como cualquier actividad destinada a evaluar un atributo o capacidad de un programa o sistema con la intención de encontrar errores y así poder determinar que cumple con los resultados requeridos o esperados (Hetzel, 1988; Myers et al., 2004).

El testing es una parte fundamental en todo proceso de desarrollo de software (Abran y Moore, 2004) que generalmente se realiza con alguno de los siguientes propósitos (Pan, 1999):

- **Para mejorar la calidad:** Dado que las computadoras y el software se utilizan en aplicaciones críticas, el resultado de un error puede ser grave y estos pueden causar grandes pérdidas.

- **Para la verificación y validación** Las pruebas pueden servir como métricas. Se pueden hacer afirmaciones basadas en la interpretación de los resultados de las pruebas, ya sea que el producto funcione en determinadas situaciones o no.

- **Para estimar la fiabilidad:** Las pruebas pueden servir como un método de muestreo estadístico para obtener datos de fallos para la estimación de fiabilidad.

El testing de software es un área muy consolidada dentro de la ingeniería del software y existe un gran número de técnicas de testing. En este trabajo vamos a utilizar la técnica denominada testing metamórfico.

2.2. Testing Metamórfico

Este tipo de testing se basa en relaciones metamórficas. Dada una función f, una relación metamórfica establece una relación entre una serie de entradas , $x_1, x_2, \ldots, x_n$ y las salidas correspondientes $f(x_1), f(x_2), \ldots, f(x_n)$ con $n > 1$. Si la función es implementada como un programa, estas relaciones representan relaciones entre las diferentes ejecuciones del programa. Por ejemplo, consideremos una función *log* que calcula el logaritmo con base 2 de un número . La identidad del producto logarítmico establece que $\log_2 x \cdot y = \log_2 x + \log_2 y$. Luego, podemos establecer una relación metamórfica para este programa basándonos en esta propiedad.

$$R = \{(x_1, x_2, x_3, log(x_1), log(x_2), log(x_3)) \mid x_1 = x_2 \cdot x_3 \rightarrow log(x_1) = log(x_2) + \log(x_3)\}$$

Una relación metamórfica define como dado un *source test* case (x_1), se pueden generar *follow-up test* cases (x_2, x_3). Es importante señalar que las relaciones metamórficas se pueden restringir a source test cases que cumplan algunas condiciones. La idea detrás del testing metamórfico es que podemos utilizar las relaciones metamórficas para detectar posibles fallos en los programas. Si se viola una relación entonces el programa es defectuoso. Las pruebas metamórficas no verifican las salidas de los casos de pruebas individuales, sino la relación entre los resultados de los diferentes casos de prueba. Esto significa que no es necesario un oráculo. Por lo tanto esta estrategia puede aliviar el problema de la falta de oráculo en el caso de la aplicación de técnicas de testing. No obstante, debemos tener en cuenta que

incluso cuando se satisfacen todas las relaciones metamórficas consideradas, no podemos decir que el programa es correcto. La figura 2.1 ilustra las fases básicas del proceso de testing metamórfico. El primer paso corresponde a la identificación de las propiedades asociadas al sistema que está bajo prueba y su representación como relación metamórfica. Utilizando los source test cases se generarán a partir de las relaciones metamórficas los follow-up test cases. Finalmente, ambos serán ejecutados y se verificará que se cumplen las relaciones metamórficas.

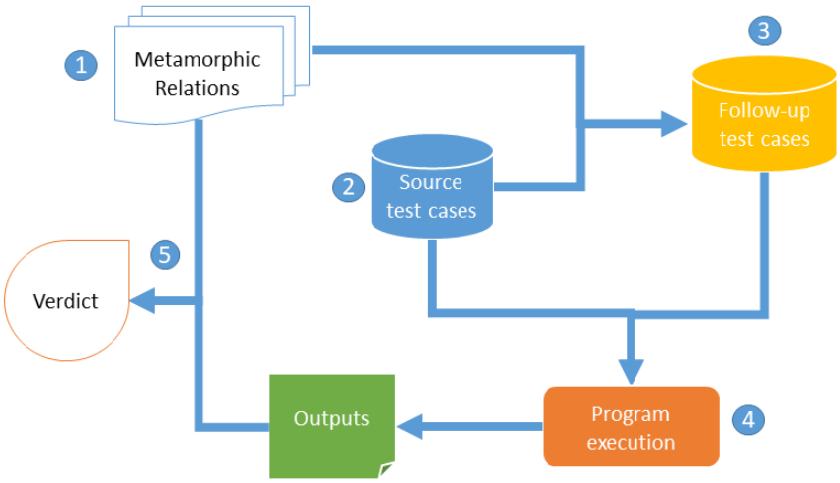

Figura 2.1: Proceso del Testing Metamórfico

2.3. SQL

Lenguaje de consulta estructurado, Structured Query Language por sus siglas en inglés o mejor conocido como SQL, es un lenguaje estándar utilizado para almacenar, manipular y recuperar datos en bases de datos.

Fue creado por la empresa IBM en los años 70 y posteriormente, SQL se convirtió en un estándar ANSI en el año 1986 y un estándar ISO en el año 1987. Hoy en día se utiliza en una gran cantidad de sistemas de gestión de bases de datos.

2.3.1. Sentencia SELECT

Este trabajo está enfocado a las operaciones de recuperación de datos sobre bases de datos SQL. Estas operaciones se realizan mediante la sentencia SELECT. Esta

sentencia tiene una gran capacidad expresiva, pues implementa el álgebra relacional, que es el lenguaje de consulta formal desarrollado para el modelo relacional desarrollado por Codd (Codd, 1970). La sentencia SELECT permite combinar las condiciones de búsqueda sobre una o varias tablas y realizar operaciones muy complejas sobre los datos. Por este motivo, la sintaxis y las opciones de esta sentencia son también enormemente complejas. En este trabajo utilizaremos una versión simplificada de esta consulta para desarrollar el prototipo de testing metamórfico. En la Figura 2.2 se presenta la versión simplificada de la sintaxis de la sentencia SELECT que se considerará a lo largo de este trabajo.

```
SELECT [DISTINCT] [expression [[AS] output_name ] [, ...]]
FROM from_clause
[WHERE condition]
[ORDER BY [expression [ASC | DESC| USING operator] [, ...]]
```

Figura 2.2: Sintaxis general de la sentencia SELECT.

La sentencia SELECT tiene varias cláusulas, algunas de ellas opcionales. A continuación se detallan cada una de ellas:

- SELECT: Especifica las columnas a consultar.
- DISTINCT: Filtra los valores repetidos.
- AS: Proporciona un alias que se puede utilizar para cambiar temporalmente el nombre de tablas o columnas.
- FROM: Especifica las tablas a consultar.
- WHERE: Filtra los registros, solo aquellos que cumplen con una condición específica son recuperados.
- ORDER BY: Ordena el conjunto de resultados en orden ascendente o descendente.

En el resto del trabajo denotaremos con $\mathcal{Q}$ a el conjunto de consultas que pueden ser creadas con la sintaxis de la Figura 2.2.

Este trabajo no va a tener en cuenta el contenido de la cláusula FROM, y en particular no se van a analizar las subconsultas que aparezcan en esta cláusula ni las condiciones de reuniones de tablas que se utilicen en expresiones JOIN ON. Esta herramienta se centrará en la condición asociada a la cláusula WHERE que es usada para filtrar las filas devueltas por la sentencia SELECT. Una condición es una expresión booleana o la combinación de varias expresiones booleanas que utilizan los operadores AND, OR y NOT. Una expresión booleana puede incluir diferentes operadores. La Tabla 2.1 lista los operadores considerados en nuestra herramienta.

Operador	Sintaxis	Ejemplo
Lógicos	expresión **AND** expresión | expresión **OR** expresión | **NOT** expresión	age=25 OR city = 'London'
Comparativos	expresión {< | = | > | <= | >= | <>} expresión	age<=18
Valores NULL	expresión **IS [NOT] NULL**	MidName IS NULL
Patrón	expresión **[NOT] LIKE** expresiónWildCard where WildCard='%' or '_'	name LIKE '%th'
Rango	expresión **[NOT] BETWEEN** expresión **AND** expresión	age BETWEEN 18 AND 35
Conjunto	expresión **[NOT] IN** (expresión [, expresión]*)	city in ('London', Óxford')

Tabla 2.1: Operadores para expresiones en la cláusula WHERE.

Las condiciones presentan diferentes formas, desde la más simple como las que incluyen operadores de conjunto, de patrones (LIKE), rangos y operaciones de com-

paración con valores nulos, hasta las más complejas que incluyen subconsultas. Una subconsulta es una sentencia SELECT anidada dentro de otra sentencia SQL. En el caso de la sentencia SELECT el resultado de la subconsulta es usado como condición en la consulta externa. Las subconsultas pueden ser clasificadas en diferentes tipos dependiendo de los resultados que retornan.

- Subconsulta escalar: Retorna un solo valor.
- Subconsulta de fila: Retorna una sola fila pero con múltiples columnas.
- Subconsultas tabulares: Retornan múltiples filas y múltiples columnas por fila.

A continuación, en la Tabla 2.2 se listan los operadores considerados en los que participa una subconsulta y posteriormente el detalle del funcionamiento de cada uno de ellos.

expresión **[NOT] IN** Subconsulta	population IN (SELECT population FROM Cities)
[NOT] EXISTS Subconsulta	EXISTS (SELECT population FROM Cities)
[NOT] expresión {< | = | > | <= | >= | <>} **ALL** Subconsulta	population >= ALL (SELECT population FROM Cities)
[NOT] expresión {< | = | > | <= | >= | <>} **ANY** Subconsulta	population=ANY (SELECT population FROM Cities)
expresión {< | = | > | <= | >= | <>} Subconsulta	population = (SELECT population FROM Cities WHERE name='Caracas')

Tabla 2.2: Condiciones con Subconsultas

- **[NOT] IN:** Verifica que la expresión del primer operando esté entre los resultados de la subconsulta.
- **[NOT] EXISTS:** Verifica si existen o no filas como resultado de la ejecución de la subconsulta.
- **ALL:** Verifica si todos los valores de la subconsulta cumplen con la condición dada.
- **ANY:** Verifica si alguno de los valores de la subconsulta cumplen con la condición dada.
- Por último, las comparaciones directas con una subconsulta necesitan que sea una subconsulta de fila.

Los resultados producidos por una subconsulta indican el alcance en el que pueden ser usadas. En el contexto de las condiciones incluidas en la cláusula WHERE, las subconsultas escalares solo pueden ser usadas como un valor constante. Por otra parte, para los operadores ALL, ANY e IN se utilizan subconsultas tabulares. En nuestro sistema solo consideraremos aquellas subconsultas que retornan exactamente una fila. En el caso del operador [NOT] EXISTS cualquier tipo de subconsulta puede ser usada.

Adicionalmente, podemos distinguir entre subconsultas correlacionadas y no correlacionadas. Las subconsultas correlacionadas hacen referencia a columnas de las tablas de la consulta exterior. Esto significa que la clausula WHERE de la subconsulta correlacionada usa la información de la consulta externa. Por ejemplo, en la Figura 2.3 se puede observar el uso de la columna departamento de la consulta exterior dentro de la subconsulta.

```
SELECT dni, salario, departamento FROM empleado emp1
WHERE salario IN (SELECT MAX (emp2.salario) FROM empleado emp2
                  WHERE emp1.departamento = emp2.departamento)
```

Figura 2.3: Ejemplo de consulta correlacionada.

La diferencia principal es que una subconsulta correlacionada se ejecuta una vez por cada fila seleccionada de la consulta exterior y una subconsulta no correlacionada se ejecuta independientemente de la consulta exterior.

2.3.2. Manejo de Valores NULL en SQL

SQL tiene varias reglas para tratar con valores de este tipo. El valor NULL es utilizado para representar un valor que falta y que generalmente puede tener tres interpretaciones diferentes (Elmasri y Navathe, 2015).

- **Valor UNKNOWN:** El valor existe pero no se conoce, o no se sabe si el valor existe o no.
- **Valor No Disponible:** El valor existe pero es omitido intencionalmente.
- **Valor No Aplicable:** El valor no aplica o no está definido para un caso particular.

En muchos casos no es posible determinar cuál de estos significados es el correcto, por lo que SQL no es capaz de distinguir entre los diferentes significados que puede tomar el valor NULL. Intuitivamente, se puede decir que NULL se refiere a 'cualquier valor', pero en general, cada valor NULL es considerado distinto a otro valor NULL entre los diferentes registros de la base de datos. Cuando un atributo que acepta valores NULL está involucrado en una operación de comparación, el resultado se considera UNKNOWN. Esto puede ocurrir en comparaciones como $NULL = NULL$ o $NULL > 0$, en las cuales no es posible determinar si una condición es verdadera o falsa porque esta dependería del valor NULL. Por esta razón, SQL usa un fragmento de la lógica trivaluada de Kleene (Kleene, 1938) con los valores *verdadero*, *falso* y *UNKNOWN* en lugar de utilizar la lógica booleana estándar *verdadero* o *falso*. Para esto, es necesario definir los resultados o valores de verdad de las expresiones de la lógica trivaluada cuando se utilizan los conectores lógicos AND, OR o NOT. Los valores resultantes se muestran en la Tabla 2.4.

En el caso de operaciones que incluyen operadores lógicos, el resultado será UNKNOWN cuando cualquiera de las expresiones involucradas sea UNKNOWN y la eva-

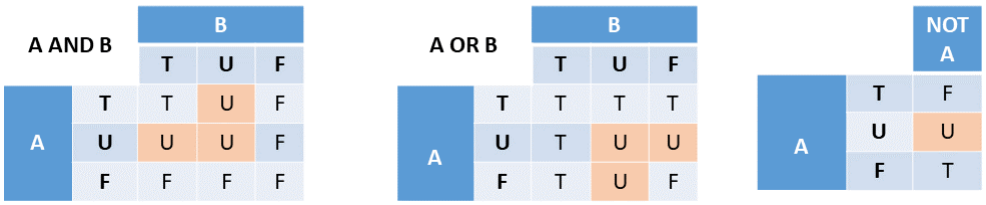

Figura 2.4: Tablas de verdad de la lógica trivaluada.

luación dependa de dicha expresión. Por ejemplo, consideremos la condición NULL > 0 OR $10 < 11$. La evaluación de la expresión izquierda NULL > 0 es UNKNOWN; sin embargo no afecta al resultado general porque la expresión derecha $10 < 11$ es verdadera. No importa si el valor NULL corresponde a un valor que hace que la expresión NULL > 0 sea verdadera o falsa, el resultado no depende del valor NULL. Otro ejemplo NULL > 0 AND $10 < 11$, muestra como el valor UNKNOWN puede influenciar el resultado global. Dependiendo de la evaluación de NULL > 0, la evaluación de la condición puede ser verdadera o falsa, por lo tanto el valor UNKNOWN conduce a un resultado incierto. Un razonamiento similar se aplica para el operador NOT.

Esta forma de interpretar los valores nulos plantea en muchos casos problemas e interpretaciones peculiares de consultas SQL. De hecho, existe una amplia bibliografía sobre este aspecto en la que se proponen diversas semánticas para el tratamiento de valores nulos, como se ve a continuación.

2.3.3. Valores NULL en SQL

Hay muchos trabajos relacionados con bases de datos incompletas, así como varias semánticas para los valores nulos. Los enfoques teóricos proporcionan las interpretaciones más completas sobre bases de datos incompletas al precio de la eficiencia. Una de las primeras semánticas es la llamada semántica de nulos de Codd (Codd, 1979; Guagliardo y Libkin, 2019). Esta semántica se reduce a considerar los valores nulos marcados como no repetidos en las tablas de la base de datos. La semántica de nulos de SQL se basa fundamentalmente en esta semántica. Por otra parte, la semántica de nulos etiquetados (Imielinski y Lipski, 1984) considera las bases de

T1	
ColA	ColB
b	2
NULL	1

T2		
ColA	ColB	ColC
a	2	y
NULL	NULL	x

Figura 2.5: Ejemplo de una base de datos sencilla.

datos donde los valores NULL pueden mantener el mismo valor nulo en diferentes ocurrencias. Esta semántica es solo de interés teórico, y permite una evaluación naif, considerando los valores nulos como nuevas constantes. Las tablas condicionales (Imielinski y Lipski, 1984) permiten tuplas con diferentes valores en tablas sujetas a condiciones de disyunción. La semántica de SQL con respecto a los valores nulos es eficiente, con un alto costo de poder expresivo. Esta considera los valores, de forma similar a la semántica de Codd. Se maneja mediante una lógica trivaluada que trae resultados no lógicos como se puede ver en el siguiente ejemplo: supongamos una base de datos con una tabla T con una columna A y una sola fila NULL. el resultado de la consulta `SELECT A FROM T WHERE A = A` no devuelve filas, a pesar de la tautología en la cláusula WHERE. Resultados como este, producidos por bases de datos SQL, son claramente contrarios a la intuición o, en algunos casos, simplemente incorrectos.

Cuando tratamos con consultas en SQL debemos tomar en cuenta que el resultado incluye solo aquellas filas que hacen que la evaluación de la expresión en las cláusulas WHERE o HAVING sea verdadera. En caso contrario, la consulta no retornará la fila. En caso de que la evaluación tome un valor UNKNOWN, este es tratado como falso. Podemos encontrar consecuencias inesperadas de la ocurrencia de valores nulos en las condiciones incluidas en estas cláusulas. Por ejemplo, consideremos la base de datos en la Figura 2.5 que contiene la tabla T1 con dos columnas `colA` y `colB` y dos filas (`NULL`,1) y (`NULL`,2).

La consulta

```
select colA
from T1
where colA=colA OR colA<>colA
```

devolverá un conjunto vacío, porque las filas guardadas en la tabla son rechazadas debido a que la evaluación de la condición en la cláusula WHERE en ambos casos es UNKNOWN. Otro ejemplo que retorna un resultado sorprendente involucra el operador NOT IN.

```
select colB
from T1
where colB NOT IN (NULL,1)
```

A pesar de que existe una fila en la tabla que almacena el valor 2 para la columna colB y no aparece en los valores considerados en la lista (NULL, 1) no se retorna el valor. En este caso la evaluación de la condición es UNKNOWN porque dependiendo de los valores que le asignemos a NULL el resultado puede ser verdadero o falso. Si consideramos el valor 0 entonces la expresión para la fila considerada será verdadero, mientras que si usamos el valor 2 entonces será evaluado a falso. por lo tanto la evaluación final sera UNKNOWN. Este comportamiento afecta también a las subconsultas. Consideremos la siguiente consulta.

```
select colA from T1
where colA  NOT IN (select colA from T2);
```

El resultado de esta consulta es un conjunto vacío debido a que el resultado de la subconsulta contiene el valor NULL. Este es un ejemplo en el que podemos perder respuestas. En el mismo modo podemos obtener tuplas que no deberían ser incluidas en los resultados. Este es el resultado de la siguiente consulta.

```
select colA from T2 as A
where NOT EXISTS (select * from T2 as B
                  where colC='x' AND B.colB > A.colB);
```

El resultado de esta consulta, incluye todos los valores de la `colA` porque la subconsulta regresa un conjunto vacío por cada fila en la tabla 2 . Por lo tanto la condición de `NOT EXISTS` se evalúa a verdadero y la respuesta final incluye todas las filas de la tabla T2 en vez de ninguna de ellas.

Además, como se afirma en (Guagliardo y Libkin, 2017), entre otros trabajos, la evaluación de SQL en bases de datos incompletas no solo pierde ciertas respuestas (falsos negativos), como en el ejemplo mostrado anteriormente, pero también puede producir falsos positivos, es decir, puede devolver algunas tuplas que no pertenecen a determinadas respuestas.

Capítulo 3

Marco de Testing Metamórfico

Resumen: En este capítulo se presenta el marco de testing que se propone para la detección de posibles errores en los resultados obtenidos por consultas SQL que tratan con columnas que pueden contener valores nulos.

El objetivo principal de nuestro marco de testing es proporcionar a los testers una herramienta que los ayude a determinar si los resultados producidos por una sentencia SELECT que trata con valores NULL corresponde con los resultados esperados. Es importante resaltar que estos resultados dependerán del *significado* de los valores NULL para las diferentes columnas incluidas en la consulta de interés.

3.1. Relaciones metamórficas

Nuestra propuesta se basa en el concepto de testing metamórfico, por lo cual definiremos una colección de relaciones metamórficas entre sentencias SELECT y el resultado obtenido de la ejecución de las mismas. Para ello, usaremos una *plantilla metamórfica*, que corresponde a una abstracción de todas las relaciones metamórficas consideradas en nuestro marco de testing.

A continuación, proporcionamos una definición de relación metamórfica y las nociones de *source* y *follow-up inputs*. Las relaciones metamórficas que proponemos consideran dos entradas y dos salidas y por tanto nuestra definición se restringe a esta condición.

Definición 1 *Sea RS un SGBD y $\mathcal{D}$ una instancia de una base de datos DB. Dada una consulta de entrada q sobre DB, se denota por $RS(q, \mathcal{D})$ al conjunto de los resultados de la ejecución de q por RS sobre DB.*

Una relación metamórfica *MR es una relación entre dos valores de entrada, q_1 y q_2, y las salidas correspondientes, $RS(q_1, \mathcal{D})$ y $RS(q_2, \mathcal{D})$, que denotaremos como $MR(q_1, q_2, RS(q_1, \mathcal{D}), RS(q_2, \mathcal{D})$. Diremos que q_1 es un* source input *y que q_2 es un* follow-up input*.*

Los source inputs generalmente son proporcionados por los testers, mientras que los follow-up inputs deben generarse automáticamente a partir de los source inputs y la definición de la relación metamórfica.

En nuestro marco, los follow-up inputs se generarán en base a la coincidencia de un fragmento de código del source input con un *patrón*. Un patrón corresponde a un fragmento de código SQL que se puede encontrar en una consulta SQL. Las Tablas 3.1 y 3.2 presentan los patrones considerados en esta propuesta. Los patrones incluidos en la Tabla 3.1 (A1-A6) se centran en las subconsultas anidadas que pueden aparecer en la sentencia SELECT, mientras que los patrones definidos en la Tabla 3.2 están relacionados con la ocurrencia de columnas que acepten valores NULL en la lista de expresiones de la cláusula SELECT (B1-B2), así como algunos operadores que pueden aparecer en la cláusula WHERE (B3-B6). Estas tablas también incluyen, para cada patrón, la *transformación* propuesta para generar los follow-up inputs.

Definición 2 *Sea $\mathcal{P}$ un conjunto de patrones. La función de matching $Match$: $\mathcal{Q} \times \mathcal{P} \rightarrow Boolean$ se define $\forall q \in \mathcal{Q}, p \in \mathcal{P}$:*

$$Match(q,p) = \begin{cases} true & \text{si } q \textit{ contiene un fragmento de código que coincida con } p \\ false & \textit{en cualquier otro caso.} \end{cases}$$

Dada una consulta q tal que se cumple $Match(q, p)$, se denota $Trf(q, p)$, a la consulta obtenida por el reemplazo del código SQL de la consulta q que coincide con p al aplicar la transformación asociada.

A continuación, se presentan las relaciones metamórficas propuestas para detectar posibles errores relacionados con los resultados de las consultas SQL que incluyan

columnas que acepten valores NULL. Para su definición se utiliza una *plantilla metamórfica* que representa el conjunto de todas las relaciones metamórficas que se pueden obtener si se consideran los patrones y transformaciones incluidos en las Tablas 3.1 y 3.2.

El ámbito de aplicación de la relación metamórfica es una instancia específica de la base de datos con la que está relacionada la consulta. La relación entre las salidas viene dada por una expresión de igualdad.

Definición 3 *Sea RS un SGBD,* $\mathcal{D}$ *una instancia de una base de datos DB y* $\mathcal{P} = \{A.i \mid 1 \leq i \leq 7\} \cup \{B.i \mid 1 \leq i \leq 6\}$ *el conjunto de patrones considerados. Dado* $q \in \mathcal{Q}$ *sobre DB,* $\forall p \in \mathcal{P}$ *tal que* $Match(q, p)$

$$\texttt{EqResults}_p(q,\, \mathit{Trf}(q,p), RS(q,\mathcal{D}),\, RS(\mathit{Trf}(q,p),\mathcal{D}))$$
$$\Updownarrow$$
$$RS(q,\mathcal{D}) \equiv RS(\mathit{Trf}(q,p),\mathcal{D})$$

Básicamente, el fragmento de código en la sentencia SELECT que coincida con cualquiera de los patrones considerados, será reemplazado por el código de la transformación correspondiente. La nueva sentencia SELECT, que es generada automáticamente, será el follow-up input. Algunas consultas pueden encajar con más de un patrón. Por ejemplo, cuando la consulta se ajusta al patrón A.5. En este caso, el patrón A.6 y A.7 también pueden ser utilizados para transformar la consulta. También se podrán considerar dos relaciones metamórficas si el código de la consulta presenta una subconsulta que corresponde al patrón A.6 ya que en este caso también se ajustará al patrón A.7.

Una vez transformada la consulta, ambas (source y follow-up inputs) son ejecutadas utilizando el SGBD considerado y la instancia actual de la base de datos asociada a la consulta, verificándose si las salidas satisfacen las condiciones impuestas por la relación metamórfica. Si alguna de las relaciones metamórficas no se cumple, entonces se ha detectado un posible fallo. En caso contrario, se puede iterar el proceso anterior para construir nuevos follow-up inputs a partir de la consulta transformada hasta que no se puedan aplicar más transformaciones.

Patrón		Transformación	
A.1	X *comp* ALL (SELECT col FROM T WHERE C)	AT.1	NOT EXISTS (SELECT col FROM T WHERE C AND X $\neg$*comp* col)
A.2	NOT X *comp* ANY (SELECT col FROM T WHERE C)	AT.2	NOT EXISTS (SELECT col FROM T WHERE C AND X *comp* col)
A.3	X NOT IN (SELECT col FROM T WHERE C)	AT.3	X <> ALL (SELECT col FROM T WHERE C)
A.4	X *comp* (SELECT col FROM T WHERE C)	AT.4	X IS NULL OR (SELECT col FROM T WHERE C) IS NULL OR X *comp* (SELECT col FROM T WHERE C)
A.5	NOT EXISTS (SELECT col FROM T WHERE E.X = col AND C)	AT.5	X NOT IN (SELECT col FROM T WHERE C)
A.6	NOT EXISTS (SELECT col FROM T WHERE E.X comp col AND C)	AT.6	X $\neg$*comp* ALL (SELECT col FROM T WHERE C)
A.7	NOT EXISTS (SELECT col FROM T WHERE E.X comp col AND C)	AT.7	NOT X *comp* ANY (SELECT col FROM T WHERE C)
where *comp* $\in \{<, <=, >, >=, =, <>\}$			

Tabla 3.1: Relaciones metamórficas para sentencias SELECT con subconsultas.

3.2. Árbol Sintáctico Abstracto de Consultas SQL

La siguiente gramática representa el subconjunto del lenguaje SQL que se considera en este trabajo. Esta gramática nos permite construir representaciones estruc-

B.1	SELECT X *arithOper* Y	BT.1	SELECT COALESCE(X,N) *arithOper* COALESCE(Y,N) where $arithOper \in \{+, -, *, /\}$ and N is the identity element of the operation.
B.2	SELECT X *stringOper* Y	BT.2	SELECT COALESCE(X,“ ”) *stringOper* COALESCE(Y,“ ”)
B.3	WHERE X *comp* Y	BT.3	WHERE X *comp* Y OR X is NULL OR Y IS NULL
B.4	WHERE X BETWEEN Y AND Z	BT.4	WHERE X BETWEEN Y AND Z OR X IS NULL OR Y IS NULL OR Z IS NULL
B.5	WHERE X IN (A,. . ., Z)	BT.5	WHERE X IN (A,. . ., Z) OR A IS NULL OR . . . OR Z IS NULL
B.6	WHERE X LIKE Y	BT.6	WHERE X LIKE Y OR X IS NULL OR Y IS NULL

Tabla 3.2: Relaciones metamórficas para las cláusulas SELECT y WHERE.

turadas del Árbol Sintáctico Abstracto (AST) de las consultas SQL. Los símbolos no terminales están representados con identificadores en mayúscula.

$$
\begin{array}{lcl}
QUERY & := & SELECT\ \ FROM\ \ WHERE \\
SELECT & := & \texttt{select}\ \ L \\
L & := & EXP \mid EXP,\ \ L \\
FROM & := & \texttt{from}\ \ TABLELIST \\
TLIST & := & T \mid T, TLIST \\
T & := & tblName \mid tblName\ \ \texttt{as}\ \ alias \\
WHERE & := & \texttt{where}\ \ COND \\
COND & := & EXP\ \ \texttt{between}\ \ EXP\ \ \texttt{and}\ \ EXP \mid STREXP\ \ \texttt{like}\ \ STREXP \mid \\
 & & EXP\ \ relOp\ \ EXP \mid EXP\ \ \texttt{is null} \mid EXP\ \ \texttt{is not null} \mid \texttt{not}\ \ COND \mid \\
 & & COND\ \ \texttt{or}\ \ COND \mid COND\ \ \texttt{and}\ \ COND \mid SUBQUERY \\
SUBQUERY & := & \texttt{exists}\ \ (QUERY) \mid \texttt{not exists}\ \ (QUERY) \mid EXP\ \ \texttt{in}\ \ (QUERY) \mid \\
 & & EXP\ \ \texttt{not in}\ \ (QUERY) \mid EXP\ \ relOp\,\texttt{any}\ \ (QUERY) \mid \\
 & & EXP\ \ relOp\,\texttt{all}\ \ (QUERY) \mid EXP\ \ relOp\ \ (QUERY) \\
EXP & := & STREXP \mid ARITHEXP \\
STREXP & := & STREXP\ \ \texttt{||}\ \ STREXP \mid strConstant \mid colName \mid \texttt{null} \\
ARITHEXP & := & ARITHEXP\ \ arithOp\ \ ARITHEXP \mid numConstant \mid colName \mid \texttt{null}
\end{array}
$$

Donde $relOp \in \{<, >, =, \neq, \geq, \leq\}$, $arithOp \in \{+, -, *, /\}$, $tblName$ es un nombre de tabla, $colName$ es un nombre de columna válido de una tabla utilizada en la consulta, y *strConstant* y *numConstant* son constantes string y numéricas respec-

tivamente.

Esta gramática nos permite generar un árbol sintáctico abstracto, donde las hojas corresponden a los nodos de tabla, nombres de columna y constantes y los nodos intermedios corresponden a los símbolos no terminales.

Usaremos el AST asociado a una consulta para determinar si la consulta tiene que lidiar potencialmente con valores NULL y, por lo tanto, podría conducir a *resultados incorrectos*. Esas son las situaciones en las que las relaciones metamórficas podrían ayudar a los testers a decidir si la consulta devuelve la respuesta *esperada*. Se utiliza una estructura de la forma $type(case, nullable, subtrees)$ para representar un AST. El nombre de la estructura *type* es el tipo de nodo (el símbolo no terminal en la gramática para los nodos intermedios o el tipo de símbolo terminal para las hojas) y tiene como primer argumento el caso particular aplicado al símbolo no terminal, por ejemplo, `between-and` si es el primer caso de un nodo *COND*, el valor del símbolo terminal o ϵ si no hay casos aplicables, como puede ser el caso *SELECT*. Los otros dos argumentos corresponden a la etiqueta con el estado *nullable* del nodo en el AST, y a la lista de *sub-árboles* cuyos nodos raíz son los hijos de este árbol, o ϵ si éste árbol es una hoja y no tiene hijos.

Para poder determinar las consultas que pueden producir un resultado diferente con una consulta equivalente, se recorre el árbol y se etiquetan los nodos con el estado *nullable* pudiendo este tomar el valor de *verdadero* o *falso* dependiendo del caso. Para ello se tomarán los siguientes valores como referencia:

- Los nombres de columnas que permiten valores NULL serán etiquetados como *nullable*. Esta información es extraída de la *metadata* de la base de datos.
- Los valores NULL serán etiquetados como *nullable*.
- Cualquier constante diferente de NULL es etiquetado como *no nullable*.
- Cualquier nombre de tabla es etiquetado como *no nullable*.

Esta información es propagada al resto de los nodos del árbol utilizando el algoritmo recursivo 1. Se puede observar que este algoritmo procesa el árbol en postorden, pues necesita la información de nulabilidad de los subárboles de los que está

1: **function** propagateNullable($\langle type, case, subtrees \rangle$)
2: $results$: $array[subtrees.length]$ of $\langle nullableResult : boolean, nullableContents : boolean \rangle$
3: $i = 0$
4: **for each** $(tree \in subtrees)$ **do**
5: $results[i]$ = propagateNullable($tree$)
6: $i = i + 1$
7: $nullableResult$ = getNullableResult($type, case, results$)
8: $nullableContents$ = getNullableContents($type, case, results$)
9: **return** $\langle nullableResult, nullableContents \rangle$

Algorithm 1: Algoritmo de Propagación de NULL. Parte Recursiva

compuesto para poder calcular la nulabilidad del árbol completo. Por cada nodo del árbol, este algoritmo obtiene dos valores distintos sobre la nulabilidad del nodo, calculados en los algoritmos 2 y 3. Ambos reciben como parámetros el tipo y caso del símbolo no terminal a evaluar, y la lista de nulabilidad de los subárboles que están por debajo del nodo evaluado. El primero de estos dos algoritmos devuelve cierto si el subárbol correspondiente devuelve un valor que puede ser nulo, y falso en caso contrario. Por ejemplo, en una subconsulta puede ocurrir que el número de filas devueltas sea correcto, pero algunas filas pueden contener valores nulos. Esta información es relevante para evaluar el resultado de expresiones en las que intervienen subconsultas, como puede ser el caso del operador IN. El algoritmo 3 devuelve cierto si el contenido del subárbol puede verse afectado por expresiones en las que el valor nulo modifique su comportamiento, por ejemplo, subconsultas que pueden devolver un número de filas diferente dependiendo de la nulabilidad de las expresiones dentro de la subconsulta.

```
1: function getNullableResult(type, case, nulls)
2:   nullable : boolean
3:   switch (type)
4:     QUERY :
5:       nullable := nulls[0].nullableResult  // Only SELECT clause propagates
         nullable results
6:     SELECT :
7:       nullable := nulls[0].nullableResult
8:     L :
9:       nullable := nulls[0].nullableResult ∨ nulls[1].nullableResult
10:    WHERE :
11:      nullable := false
12:    COND :
13:      nullable := false
14:    EXP :
15:      nullable := nulls[0].nullableResult
16:    STREXP :
17:      if (case = ||) then
18:        nullable := nulls[0].nullableResult ∨ nulls[1].nullableResult
19:      else
20:        nullable := nulls[0].nullableResult
21:    ARITHEXP :
22:      nullable := nulls[0].nullableContents
23:  return nullable
```

Algorithm 2: Algoritmo de propagación del valor NULL. Evaluación cláusula SELECT

```
1: function getNullableContents(type, case, nulls)
2:   nullable : boolean
3:   switch (type)
4:     QUERY :
5:       nullable := nulls[2].nullableContents // Only WHERE clause propagates nullable contents
6:     SELECT :
7:       nullable := false
8:     L :
9:       nullable := nulls[0].nullableContents ∨ nulls[1].nullableContents
10:    WHERE :
11:      nullable := nulls[0].nullableContents
12:    COND :
13:      switch (case)
14:        between-and :
15:          nullable := nulls[0].nullableContents ∨ nulls[1].nullableContents ∨ nulls[2].nullableContents
16:        {like, <, =, ≠, ≥, ≤} :
17:          nullable := nulls[0].nullableContents ∨ nulls[1].nullableContents
18:        {is-null, is-not-null} :
19:          nullable := false
20:        SUBQUERY :
21:          nullable := nulls[0].nullableContents
22:        not :
23:          nullable := nulls[0].nullableContents
24:        {or, and} :
25:          nullable := nulls[0].nullableContents ∨ nulls[1].nullableContents
26:    SUBQUERY :
27:      switch (case)
28:        {exists, not-exists} :
29:          nullable := nulls[0].nullableContents
30:        {in, {<, =, ≠, ≥, ≤}-any} :
31:          nullable := nulls[0].nullableContents ∨ nulls[1].nullableContents   // (1)
32:        {not-in, {<, =, ≠, ≥, ≤}-all, {<, =, ≠, ≥, ≤}-all} :
33:          nullable := nulls[0].nullableContents ∨ nulls[1].nullableContents ∨
                nulls[1].nullableResult   // (2)
34:        {<, =, ≠, ≥, ≤} :
35:          nullable := nulls[0].nullableContents ∨ nulls[1].nullableResults   // (3)
36:    EXP :
37:      nullable := nulls[0].nullableContents
38:    STREXP :
39:      if (case = ||) then
40:        nullable := nulls[0].nullableContents ∨ nulls[1].nullableContents
41:      else
42:        nullable := nulls[0].nullableContents
43:    ARITHEXP :
44:      nullable := nulls[0].nullableContents
45:  return nullable
```

Algorithm 3: Algoritmo de propagación de NULL. Evaluación cláusula WHERE.

Capítulo 4

Implementación

Resumen: En este capítulo se describe la implementación de la propuesta para su automatización y se describen de las diferentes herramientas tecnológicas utilizadas para el desarrollo dela misma.

4.1. Solución Propuesta

Con el objetivo de aplicar el maco propuesto en el capitulo anterior, se propone como solución el desarrollo de una aplicación web que disponga de los siguientes módulos y funcionalidades:

- Módulo de gestión de usuarios: Para la creación de usuarios y control de acceso al sistema.
- Módulo de gestión de instancias: Para el almacenamiento de los datos de conexión a las diferentes instancias de bases de datos.
- Módulo de gestión de consultas: Para la creación de consultas SQL a ser analizadas.
- Módulo de aplicación de relaciones metamórficas: Para el análisis de las consultas aplicando relaciones metamórficas.

4.1.1. Módulo de gestión de usuarios

Este módulo contará con las siguientes funcionalidades:

- Registro de los usuarios en el sistema almacenando su nombre, apellido, correo electrónico y contraseña.
- Inicio de sesión de los usuarios en el sitio web con su correo electrónico y contraseña.
- El módulo permitirá la recuperación de la contraseña de los usuarios en caso de ser olvidada.
- El módulo permitirá la modificación de cierta información personal de los usuarios.

4.1.2. Módulo de gestión de instancias

Este módulo contará con la siguiente funcionalidad:

- Creación, consulta, modificación o eliminación de los datos de conexión a una instancia de bases de datos, tales como: nombre de la base de datos, usuario, contraseña, host y puerto de conexión.

4.1.3. Módulo de gestión de consultas

Este módulo contará con la siguiente funcionalidad:

- Creación, consulta, modificación o eliminación de consultas SQL asociadas a una instancia previamente almacenada, para su posterior análisis aplicando relaciones metamórficas.

4.1.4. Módulo de aplicación de relaciones metamórficas

Este módulo contará con la siguiente funcionalidad:

- Procesamiento y análisis de una consulta SQL, mostrando las posibles transformaciones realizadas una vez se hayan aplicando las relaciones metamórficas.

4.2. Historias de Usuario

Para poder cubrir las necesidades de cada uno de los módulos planteados, se definieron las historias de usuario asociadas a la aplicación. A continuación se detalla cada una de ellas en la Tabla 4.1.

Código	**Historia de Usuario**
HU-01	El usuario podrá registrarse en la aplicación
HU-02	El usuario podrá iniciar sesión en la aplicación
HU-03	El usuario podrá recuperar su contraseña vía correo electrónico
HU-04	El usuario podrá modificar su información personal
HU-05	El usuario podrá consultar las instancias de bases de datos
HU-06	El usuario podrá crear nuevas instancias de bases de datos
HU-07	El usuario podrá modificar las instancias de base de datos
HU-08	El usuario podrá eliminar las instancias de bases de datos
HU-09	El usuario podrá consultar las consultas SQL
HU-10	El usuario podrá crear nuevas consultas SQL
HU-11	El usuario podrá modificar las consultas SQL
HU-12	El usuario podrá eliminar las consultas SQL
HU-13	El usuario podrá consultar el análisis de la aplicación de relaciones metamórficas

Tabla 4.1: Historias de Usuario de la Aplicación

4.3. Modelo lógico de la base de datos.

Tomando en cuenta las funcionalidades básicas del sistema y las historias de usuario, se diseñó el modelo lógico de base de datos que se puede observar en la Figura 4.1 . Este modelo es el utilizado por la aplicación para guardar la información correspondiente con los usuarios, los datos de conexión de instancias de bases de datos y consultas SQL a ser analizadas. No debe ser confundido con las instancias sobre las cuales se realizarán la ejecución de las consultas SQL aquí almacenadas.

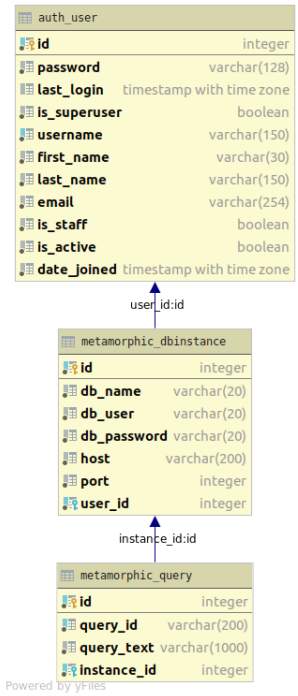

Figura 4.1: Modelo Lógico de la Base de Datos de la Aplicación.

4.4. Limitaciones

La implementación de la solución propuesta es un prototipo que permite ilustrar el concepto de pruebas metamórficas y la viabilidad de la aplicación de la técnica de testing metamórfico sobre consultas SQL. Dada la complejidad del lenguaje SQL estándar, no es el objetivo de este trabajo considerar todas las cláusulas y posibilidades de la sentencia SELECT. A continuación se listan las limitaciones que tendrá la herramienta respecto al tipo de sentencias SELECT que puede tratar:

- Se analizarán únicamente las sentencias SELECT.
- No se contempla el uso de funciones de agregación ni cláusulas HAVING, GROUP BY y ORDER BY.
- No se contempla el uso de subconsultas en la cláusula FROM y no se consideran las condiciones de reunión de tablas establecidas en expresiones de tipo JOIN ON.
- Solo se analizan las subconsultas que aparezcan en la cláusula WHERE.

- No se contempla el uso de select * en las subconsultas.
- Solo se permite una condición que afecte a columnas de tablas externas en una subconsulta correlacionada.
- Se identifican como referencias a columnas de una tabla externa a una subconsulta aquellos identificadores que no son nombres de columna de una tabla interna de la subconsulta.
- No se contempla las llamadas a funciones que no tengan argumentos, por ejemplo: SYSDATE.
- Los operadores IN, NOT IN, ANY, ALL y operadores de comparación sobre subconsultas solo permiten que la subconsulta devuelva una única columna.

Es importante resaltar que esta herramienta se ha diseñado para ser ejecutada localmente. Por lo tanto, no se contempla la totalidad de las medidas de seguridad que debe tener una aplicación web. Por ejemplo, la encriptación para el almacenamiento y envío de información sobre las credenciales de las conexiones a las diferentes instancias de bases de datos. Se recomienda usar bases de datos de prueba o de ser necesario un usuario con privilegios que se limiten a la consultas básicas de tablas o vistas. Además el archivo de configuración del servidor Django puede contener información sensible, por lo que si se desea alojar la aplicación en un servidor se deberán aplicar medidas de seguridad sobre este archivo.

4.5. Arquitectura de la Solución

Tal y como se indica en los capítulos anteriores, con esta herramienta se busca aplicar distintas relaciones metamórficas de manera automática para revelar posibles errores en la ejecución de consultas que traten con valores NULL. Para el desarrollo e implementación de la herramienta propuesta se utilizaron principalmente las siguientes tecnologías:

4.5.1. PostgreSQL

PostgreSQL (Group, 2020) es un sistema gestor de base de datos objeto-relacional de alta capacidad y de código libre. Este gestor puede ser ejecutado en la mayoría de los sistemas operativos comunes como Windows, MacOs y diferentes distribuciones Linux. Incluye la mayoría de los tipos de datos manejados de forma estándar e incluye también manejo y almacenamiento de objetos binarios. PostgreSql es compatible con los estándares ANSI / ISO SQL. Tiene soporte completo para transacciones confiables, así como soporte a base de datos orientada a objetos. Es de fácil integración y posee una comunidad extensa de apoyo con innumerable documentación. Adicionalmente, posee soporte completo para subconsultas incluyendo soporte para subconsultas en la clausula "FROM". Tiene un catálogo de sistema completamente relacional y soporta varios esquemas por base de datos definido por el estándar SQL. Maneja integridad de datos con la inclusión de funcionalidades para claves primarias y foráneas con restricciones y actualización/eliminación en cascada, restricción de chequeos, restricción de único y restricción de nulos. PostgreSQL maneja gran cantidad de extensiones para exponer sus características avanzadas. Entre ellas, maneja secuencias auto-incrementables, LIMIT/OFFSET para resultados parciales, herencias simples o múltiples de tablas, reglas de sistema y manejo de errores.

La principal razón para esta elección es que PostgreSQL es de código abierto y después de un período largo de desarrollo es considerada robusta y de confianza. PostgreSQL versión 12 tiene al menos 160 de las 179 funcionalidades obligatorias de SQL:2016. En particular, todas las características consideradas en este marco corresponden a la ISO/IEC 9075-1 (SQL/Framework) y la ISO/IEC 9075-2 (SQL/-foundation) a la que PostgreSQL pertenece.

4.5.2. Python

"Python es un lenguaje de programación de alto nivel interpretado, orientado a objetos con semántica dinámica. Su alto nivel construido en las estructuras de datos, combinado con la tipificación dinámica y vinculante, lo hacen muy atractivo para el desarrollo rápido de aplicaciones" (Python Software Foundation, 2015). Se

eligió Python por ser un lenguaje robusto y completo, además su sintaxis es sencilla y de fácil legibilidad. Por otra parte, posee un soporte extensivo de herramientas de todo tipo incluidas por defecto en su librería estándar, y por último posee una gran cantidad de paquetes de terceros disponibles a través del índice de paquetes Python (PyPI).

4.5.3. Django

Es un marco web de código abierto y escrito en Python que impulsa el desarrollo rápido de sistemas, cuenta con un conjunto de componentes que se encargan de realizar muchas de las configuraciones y funcionalidades básicas para el desarrollo de una aplicación web. Además, cuenta con su propio servidor web que es de fácil uso e incluye todo tipo de personalización.

La arquitectura de Django sigue el patrón *model-view-controller* (MVC) tan al pie de la letra que puede ser llamado un marco MVC.

- Modelo: La porción de acceso a la base de datos.
- Vista: La porción que selecciona qué datos mostrar y cómo mostrarlos.
- Controlador: La porción que delega a la vista.

Pero debido a que el controlador es manejado directamente por el mismo marco web y la parte más importante se produce en los modelos, las plantillas y las vistas, Django es mejor conocido como un marco *model-template-view* MTV. (George, 2015)

- Model (Modelo): La capa de acceso a la base de datos. Esta capa contiene toda la información sobre los datos: cómo acceder a estos, cómo validarlos, cuál es el comportamiento que tiene, y las relaciones entre los datos.
- Template (Plantilla): La capa de presentación. Esta capa contiene las decisiones relacionadas a la presentación: como algunas cosas son mostradas sobre una página web u otro tipo de documento.
- View (Vista): Esta capa contiene la lógica que accede al modelo y la delega a la plantilla apropiada: es un puente entre el modelos y las plantillas. Ya que las

vistas procesan toda la información referente a las respuestas de solicitudes, esta puede contener la lógica de negocios tanto en el archivo `view.py` o hacer uso de otros archivos.

Como todas las arquitecturas cliente-servidor, Django usa objetos de solicitud y respuesta para la comunicación entre el cliente y el servidor. Al ser Django un marco web, estamos hablando de objetos de tipo HTTP. En la Figura 4.2 se ilustra la estructura mencionada.

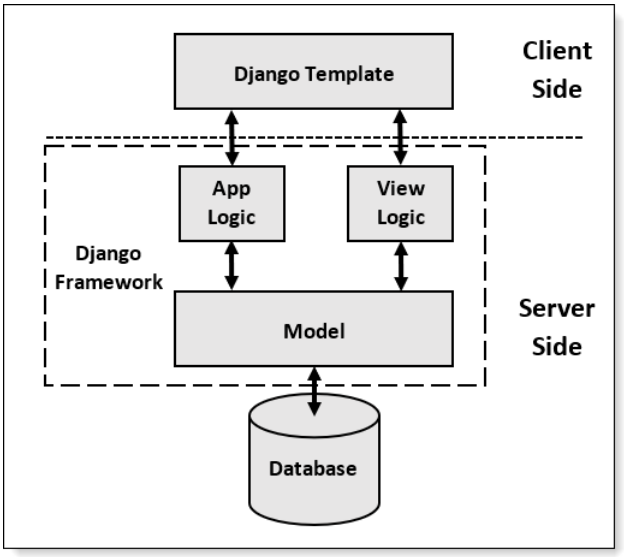

Figura 4.2: Patrón de diseño MTV de Django.
Fuente: (George, 2015)

4.6. Funcionamiento de la Aplicación

4.6.1. Front-end

Django facilita la representación y el renderizado del código HTML utilizando su motor interno de plantillas. Sin embargo, el estilo predeterminado en las páginas HTML generalmente necesita hojas de estilo en cascada (CSS) para que el diseño sea personalizable y agradable a la vista del usuario. Por esta razón, para el desarrollo de la interfaz gráfica se utilizó una plantilla HTML gratuita y el marco Bootstrap, que consiste en archivos CSS y JavaScript predefinidos los cuales se pueden vincular

a las plantillas HTML para mejorar su diseño gráfico. Para realizar las peticiones al servidor para el procesamiento de las consultas se hacen llamadas asíncronas utilizando `AJAX`, esto con la finalidad de no tener que recargar la página.

4.6.2. Back-end

Las aplicaciones web de Django administran los datos a través de objetos Python denominados modelos. Los modelos definen la estructura de los datos almacenados, los tipos de campo y comportamientos esenciales. La definición de los modelos es independiente del gestor de base de datos que se vaya a utilizar. Una vez definidos los modelos no es necesario utilizar el lenguaje SQL para la comunicación con la base de datos, Django proporciona automáticamente una API que permite crear, consultar, actualizar y eliminar objetos. Para efectos de nuestra aplicación se crearon los modelos correspondientes a las tablas mostradas en la Figura 4.1.

Luego de crear los modelos, se definen las distintas URL que los usuarios usarán para solicitar una página de nuestra aplicación. Cada URL está asociada a una vista y esta a su vez con una plantilla HTML dependiendo de la información que se quiera mostrar.

Una vista es una función de Python que acepta una petición web y devuelve una respuesta. Esta respuesta puede ser el contenido HTML de una página web, un documento XML, una imagen, etc. La vista en sí contiene cualquier lógica necesaria para devolver dicha respuesta.

El diagrama de la Figura 4.3 describe el flujo de datos principal y los componentes necesarios para manejar solicitudes y respuestas HTTP.

Una vez definidos los diferentes componentes, se ha desarrollado toda la lógica referente al procesamiento de las consultas SQL. En la Figura 4.4 se ilustra de manera genérica el flujo del núcleo de la herramienta.

4.6.3. psqlparse

Adicionalmente a las tecnologías mencionadas anteriormente, es importante destacar psqlparse (Culquicondor, 2017). Es una librería o módulo de Python que per-

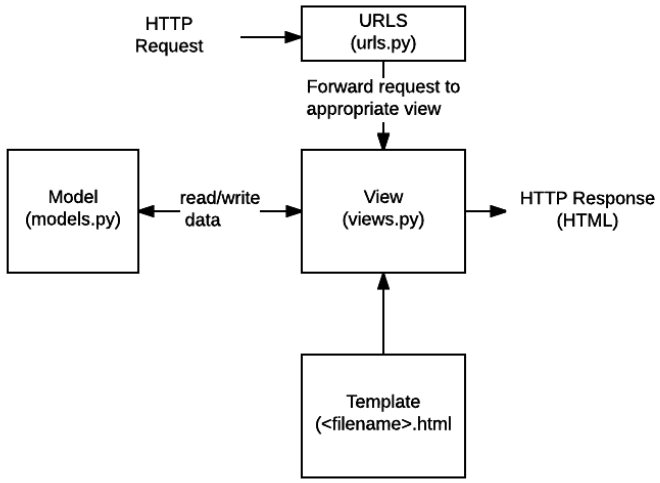

Figura 4.3: Flujo del manejo de solicitudes de Django. (Network, 2017)

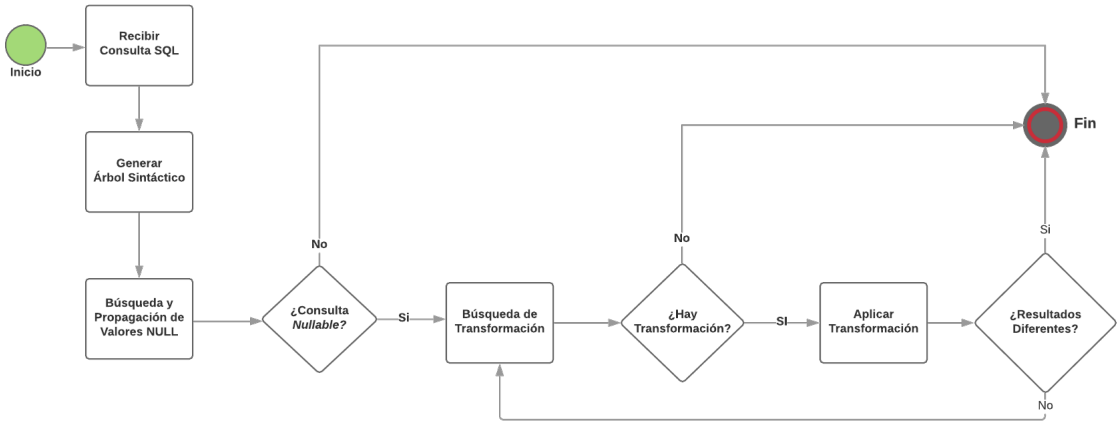

Figura 4.4: Flujo de la aplicación.

mite accesar al analizador de PostgreSQL y utilizarlo fuera de su servidor. La librería usa directamente la fuente real del servidor de PostgreSQL para procesar consultas SQL y así devolver un árbol sintáctico que represente la estructura de dicha consulta.

Un árbol es una estructura de datos que simula gráficamente la forma de un árbol jerárquico y se puede definir simplemente como un conjunto de nodos conectados entre sí. Adicionalmente, cada nodo es una estructura que puede contener un valor, una condición u otra estructura de datos. En un árbol genérico los nodos se identifican con los siguientes nombres:

- **Raíz:** Es el nodo superior del árbol.

- **Hijo:** Nodos conectados que están directamente debajo de otro nodo.
- **Padre:** Nodo conectado que está directamente arriba de otro nodo.
- **Hoja:** Nodos sin hijos.

Además es importante resaltar las siguientes características: existe un único nodo marcado como raíz, cada nodo que no sea la raíz está asociado con un nodo padre y cada nodo puede tener un número arbitrario de nodos hijo.

Para construir el árbol en Python, la librería simplemente sigue el concepto descrito anteriormente. Inicialmente se define una clase llamada *'Node'*, esta servirá como clase padre para definir los diferentes tipos de nodo que representarán la sintaxis del lenguaje SQL. En la Figura 4.5. se muestra el diagrama de clases con los diferentes tipos de nodo que serán utilizados para el análisis del árbol.

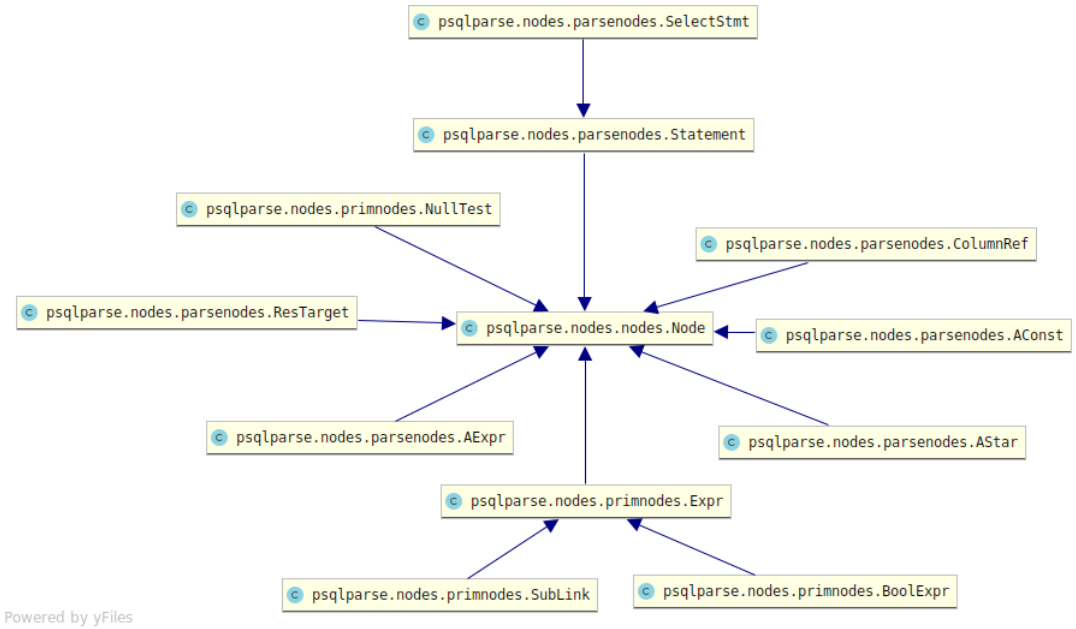

Figura 4.5: Diagrama de Clases de la librería *psqlparse*.

A pesar de que la librería incluye otros nodos que permiten representar en su totalidad los diferentes valores que pueden aparecer en cualquier consulta SQL, para los efectos de este trabajo se muestran únicamente los posibles nodos involucrados en una sentencia SELECT con la sintaxis simplificada vista en la sección 2.3.1. A continuación, en la Tabla 4.2 se muestra el detalle de cada uno de ellos.

Nodo	**Representación**
SelectStmt	Sentencia SELECT
ResTarget	Resultado de la cláusula SELECT
ColumnRef	Referencia a una columna de tabla
AStar	Ocurrencia de un '*' en la cláusula SELECT
AConst	Constante
AExpr	Expresión de tipo X *Oper* Y
BoolExpr	Expresión booleana (AND, OR, NOT)
NullTest	Ocurrencia de NULL
SubLink	Subconsulta

Tabla 4.2: Nodos a utilizar definidos en *psqlparse*.

Nos enfocaremos particularmente en el nodo SelectStmt el cual representa la ocurrencia de una sentencia SELECT. En la Figura 4.6 podemos ver su estructura y a continuación se detallan sus atributos.

- target_list: Atributo de tipo lista, representa los nodos que aparecen en la cláusula SELECT.
- from_clause: Atributo de tipo lista, representa los nodos que aparecen en la cláusula FROM.
- where_clause: Atributo de tipo nodo, representa las condiciones que aprecen en la cláusula WHERE.
- nullable_contents: Atributo de tipo boolean, representa la etiqueta para la propagación de la nulidad del contenido de la consulta en su cláusula WHERE.
- nullable_results: Atributo de tipo boolean, representa la etiqueta para la propagación de la nulidad del resultado de la consulta en su cláusula SELECT.
- nullable: Atributo de tipo boolean, representa la etiqueta de nulidad de la consulta.

Es importante mencionar que este nodo posee otros atributos, pero para efectos de este trabajo se utilizan únicamente los aquí mostrados.

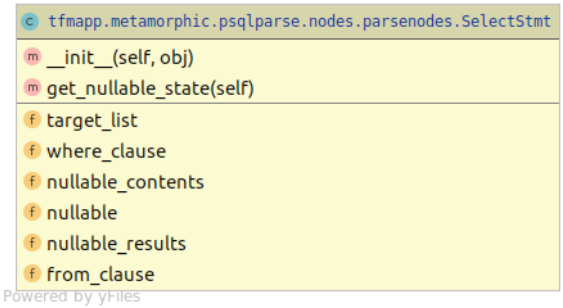

Figura 4.6: Nodo de una sentencia SELECT en Python.

El método principal de esta librería es el método `.parse()`, que recibe como parámetro una consulta SQL en formato string o varias consultas separadas por comas (‘,’) y retorna el árbol sintáctico como una jerarquía de diccionarios de Python.

Consideremos una base de datos con la tabla empleado como en la Figura 4.7. Las columnas DNI, Nombre, Primer Apellido son obligatorias (NOT NULL) y las columnas Segundo Apellido, Edad son opcionales (NULL). La consulta a analizar es la siguiente:

```
SELECT nombre, primer_apellido
FROM empleado
WHERE edad > 30;
```

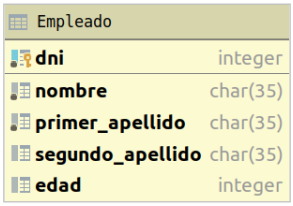

Figura 4.7: Ejemplo.

Se puede obtener la representación del árbol sintáctico haciendo uso del método `.parse()` de la siguiente forma:

```
import psqlparse
psqlparse.parse('SELECT nombre, primer_apellido FROM
    empleado WHERE edad > 30;')
```

El árbol sintáctico resultante se muestra en la Figura 4.8.

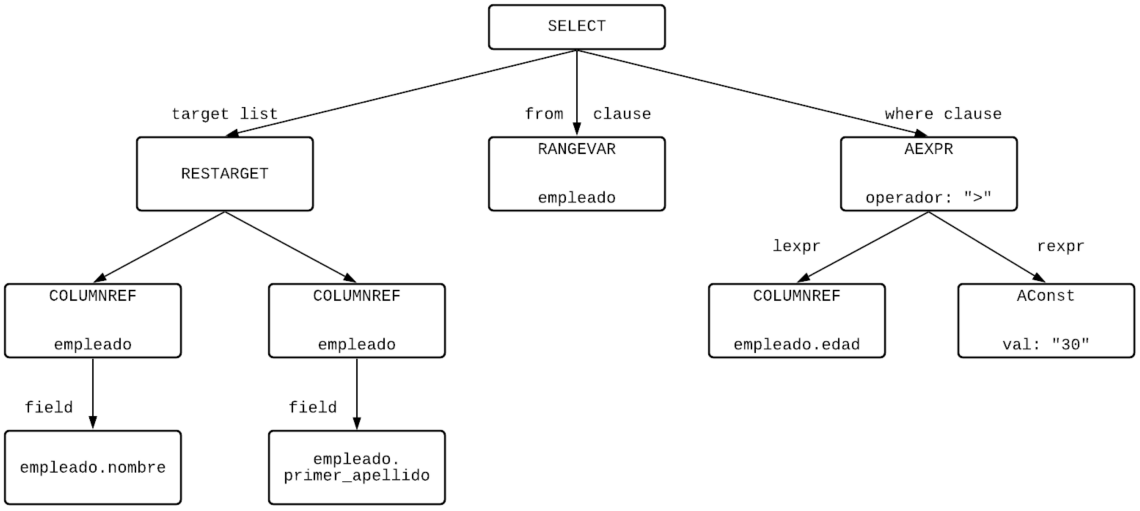

Figura 4.8: Árbol Sintáctico

4.6.4. Etiquetado del Árbol

Como se menciona anteriormente, los diferentes tipos de nodo heredan de la clase padre *Node*, en la que se define la función `.get_nullable_state()`. Luego, esta función es sobrescrita por cada uno de los nodos hijo para extraer la información necesaria sobre su nulidad, tomando en cuenta su estructura y comportamiento particular. La Figura 4.9. muestra una representación del uso del concepto de herencia y sobrescritura de funciones de la programación orientada a objetos.

Para este proceso de etiquetado se realiza una recorrido en profundidad donde se itera sobre cada atributo. Al ser estos, a su vez, de tipo nodo o lista de nodos, se hace la llamada a la función `.get_nullable_state()` por cada uno de ellos, para así ir localizando los nodos hijos que puedan estar en su interior y lograr expandir la estructura completa. Una vez que se ha llegado al nivel de profundidad máximo o a las hojas del árbol, se propaga el valor de nulidad nuevamente hasta la raíz. En el siguiente fragmento de código se muestra la implementación de la función utilizada

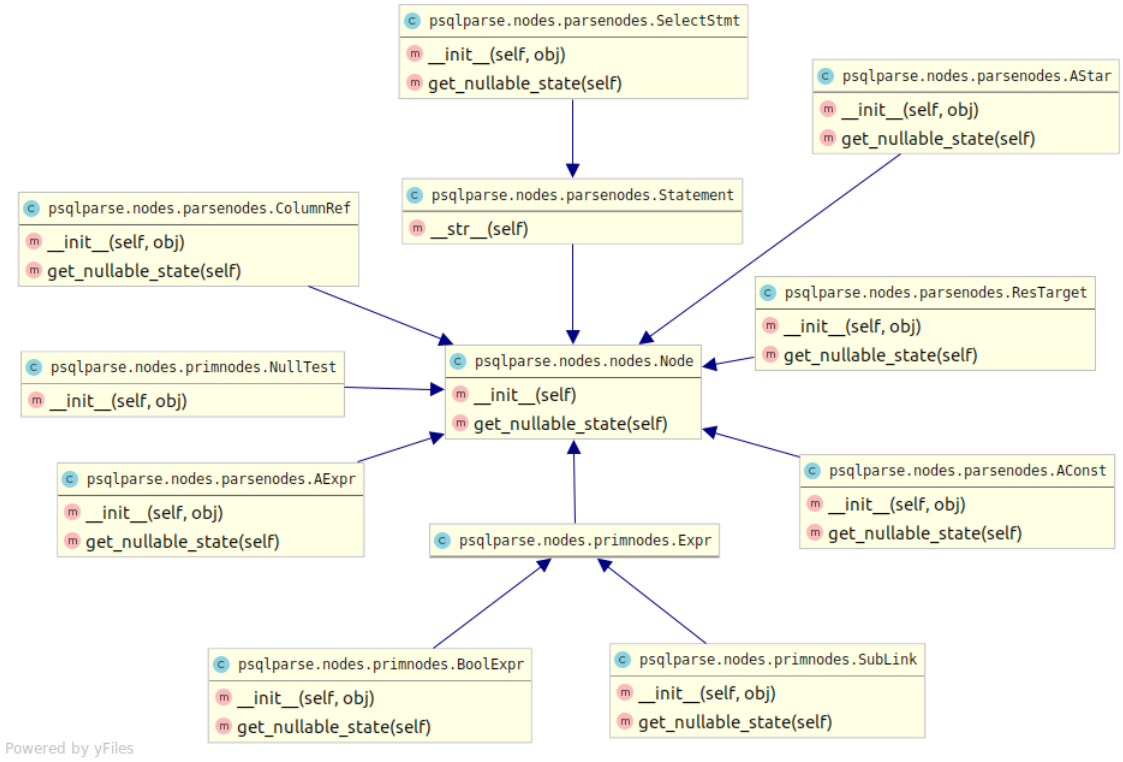

Figura 4.9: Nodos que utilizan la función `.get_nullable_state()`

para recorrer cada nodo del árbol:

```
def get_nullable_state(self):
    _nullables = list()
    for attr in six.itervalues(self.__dict__):
        if isinstance(attr, list):
            for item in attr:
                if isinstance(item, Node): _nullables.
                    append(item.get_nullable_state())
        elif isinstance(attr, Node):
            _nullables.append(attr.get_nullable_state())
    return _nullables
```

Una vez recorrido el árbol del ejemplo anterior utilizando esta función, quedaría etiquetado como se muestra en la Figura 4.10.

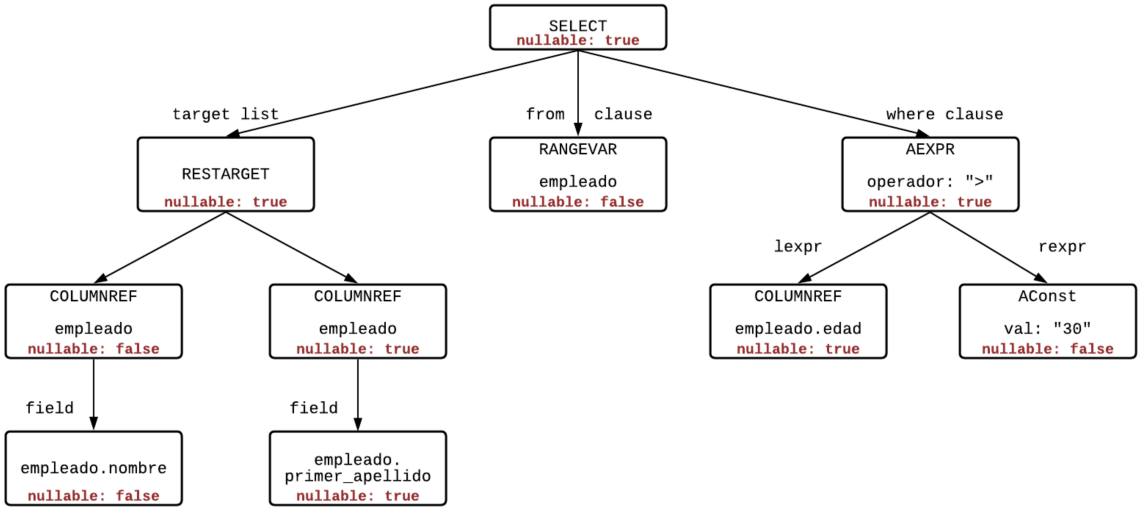

Figura 4.10: Árbol etiquetado con los valores nullables

4.6.5. Búsqueda de transformaciones

Una vez que el árbol ha sido etiquetado, se procede a buscar los posibles casos de transformación para así generar las consultas equivalentes que correspondan a las relaciones metamórficas anteriormente mencionadas, estas transformaciones solo se aplicarán a los nodos afectados por un valor NULL. La búsqueda se realiza con un recorrido similar al de etiquetado del árbol, pero esta vez siguiendo la estructura de un nodo de tipo *selectstmt*, que contiene como nodos hijos los correspondientes a las cláusulas SELECT, FROM Y WHERE. Como se menciona en las limitaciones de este trabajo no se toma en cuenta la cláusula FROM por lo que se procede a verificar la ocurrencia de los casos primero en el SELECT y luego en el WHERE. En ambos casos, se compara la estructura de los nodos que puedan estar presentes en estas cláusulas con la estructura de cada caso definido y si hay coincidencia se hace una llamada a la función de transformación correspondiente.

Siguiendo con el ejemplo propuesto, se verifica que la claúsula SELECT es *nullable*, pero al no existir una ocurrencia de un *arithOper* o *stringOper* en el nodo, no se aplica ninguna transformación. Por otra parte, en la cláusula WHERE que también es *nullable*, se consigue un nodo de tipo AExpr con la forma X *comp* Y, el cual coincide con el caso B.3

4.6.6. Aplicación de transformaciones

Por cada relación metamórfica existe una función encargada de aplicar las transformaciones previamente definidas. La función recibe el nodo original, luego se va contruyendo su cambio equivalente utilizando aquellos atributos del nodo necesarios y convirtiéndolos en string con ayuda del `printer`, finalmente se guarda la transformación realizada en el atributo *equivalent* del nodo.

Aplicando la transformación del caso B.3 al ejemplo dado, se generaría la siguiente consulta equivalente:

```
SELECT nombre , primer_apellido
FROM empleado
WHERE (edad > 30 OR edad IS NULL)
```

4.6.7. Análisis y Resultado

Una vez aplicada la transformación se ejecuta la nueva sentencia generada en la base de datos para su posterior análisis aplicando el siguiente criterio.

Primero se compara el número de filas devueltas por ambas consultas, en caso de ser diferentes se detiene la búsqueda pues se puede decir que se ha encontrado un fallo, en caso contrario, se procede a realizar la unión de ambas consultas utilizando la teoría de conjuntos, para verificar que independientemente de que el número de filas sea el mismo, se devuelve exactamente la misma información. De no ser así, tendremos diferentes resultados para cada consulta, indicando que los resultados obtenidos en cada caso son diferentes y por tanto el diseño de la consulta puede no ser el deseado.

Capítulo 5

Evaluación

Resumen: En este capítulo se presentan una serie de consultas SQL diseñadas para evaluar la aplicabilidad de nuestra propuesta mediante la herramienta desarrollada.

Después de desarrollar la herramienta, la misma se ha evaluado con un conjunto de consultas que permiten ilustrar el uso de algunas plantillas de las relaciones metamórficas presentadas en el capitulo 3.

5.1. Base de Datos de Prueba

El modelo que se muestra en la Figura 5.1 representa una base de datos sencilla de un restaurante en la que se almacenan diferentes platos de comida y los empleados encargados de prepararlos.

Para ello se definen las siguientes tablas:

- `Dish`: Esta tabla contiene los distintos platos que se almacenaran en la base de datos. Puede ocurrir que algunos platos no se ajusten a las categorías existentes por lo que la información de la columna `category` no seria aplicable. Adicionalmente, se registra en la columna `idemp` el empleado encargado de la preparación del plato, en el caso de que cualquier empleado pueda realizarlo esta información sería omitida.

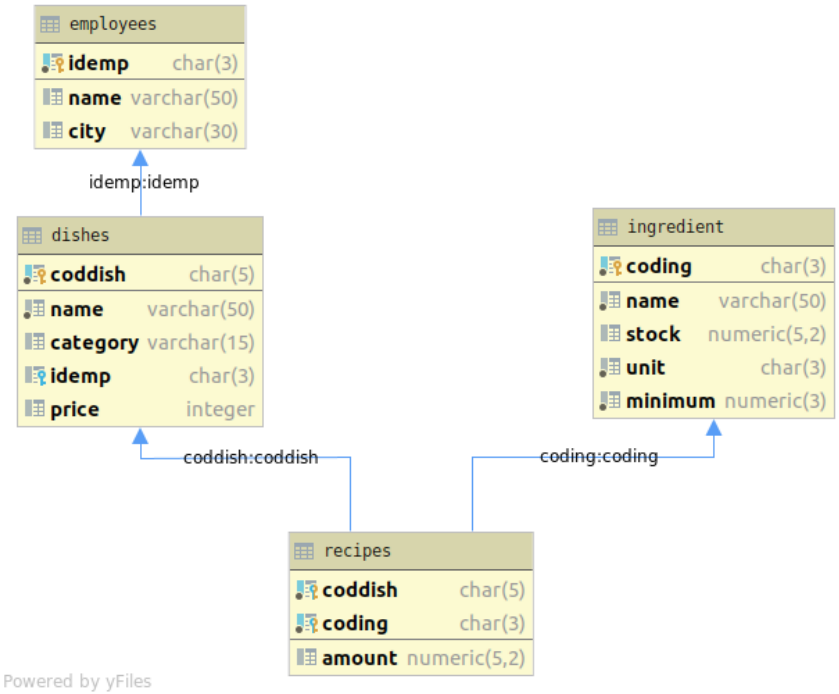

Figura 5.1: Modelo Lógico de la Base de Datos de Ejemplo.

- `Recipe`: Esta tabla contiene la información acerca de los ingredientes necesarios para la elaboración de un plato en particular. Puede ocurrir que alguno de los ingredientes sea opcional para la receta por lo que la información de la columna `amount` puede no estar disponible.

- `Ingredient`: Esta tabla contiene los ingredientes almacenados en la base de datos. Puede ocurrir que no se sepa la cantidad existente, por lo que la información de la columna `stock` es desconocida para ese caso.

- `Employee`: Empleados encargados de la preparación de los platos.

Se puede observar que algunas de las columnas se ajustan a las categorías de interpretación de valores NULL mencionadas anteriormente. Cuando el valor existe pero no se conoce, cuando el valor es omitido intencionalmente o cuando el valor no aplica.

5.2. Consulta Simple (SELECT)

En este apartado vamos a evaluar la aplicación del caso B.2. Para ello se utilizará una consulta simple sin condiciones en la cláusula WHERE y con una sola ocurrencia en la cláusula SELECT. En la Figura 5.2 se muestran los valores almacenados para las columnas `name` y `category` de la tabla `dish`, en ella se puede apreciar que existen valores NULL para la categoría de algunos platos.

	name	category
1	Chips	Side
2	Omelette	[null]
3	Fish soup	Starter
4	Fish and Chips	Main
5	Angus Steak	Main
6	Chicken Bao	[null]
7	Carrot Cake	Dessert
8	Salad	Side
9	Pudding	Dessert
10	Hamburguer	Main
11	MonkFish	Main

Figura 5.2: Columnas `Name` y `Category`.

Se busca concatenar la información de estas dos columnas utilizando la consulta:

```
SELECT d.name || ' ' || d.category
FROM dishes d;
```

Al ejecutar la consulta, esta devuelve los resultados que se muestran en la Figura 5.3 en los que se puede observar que hay filas con valor NULL en el resultado. Al intentar hacer la evaluación con el operador de concatenación "||" cuando participa un valor NULL (por ejemplo el caso de `Omelette || null`) este retorna NULL. Un compartamiento inesperado pues a pesar de que este plato no tenga categoría puede interesarnos obtener al menos el nombre.

Al utilizar el prototipo de la herramienta, este detecta el caso B.3 y automáticamente aplica la transformación correspondiente, generando la consulta:

```
SELECT COALESCE(d.name,' ') || ' ' || COALESCE(d.category,' ')
FROM dishes d;
```

Y como resultado de la ejecución de la consulta transformada, ahora podemos ver en la Figura 5.4 que los dos filas que devolvían el valor NULL anteriormente

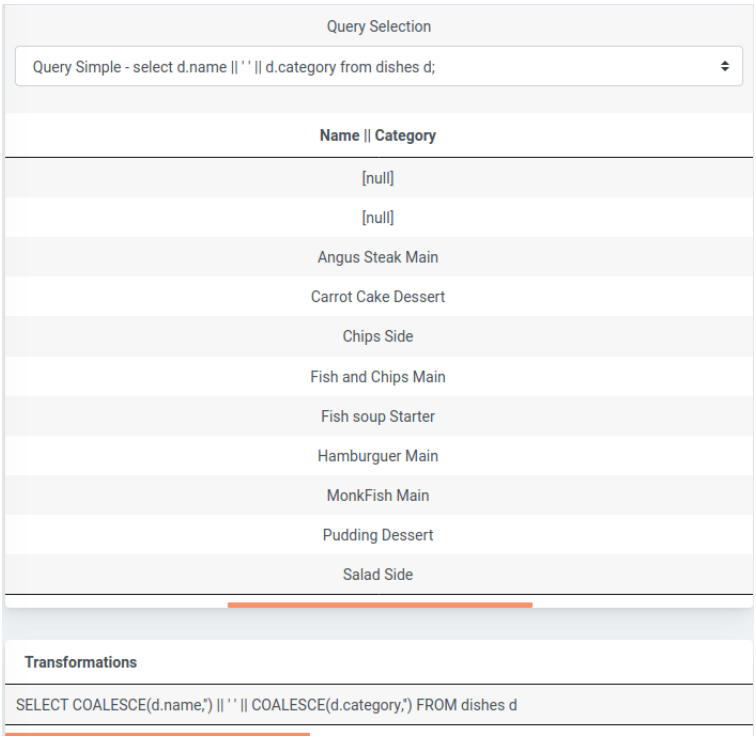

Figura 5.3: Resultado de la ejecución de la consulta.

ahora han sido completadas con la información de “Omelette” y “Chicken Bao”.

COALESCE(Name) || COALESCE(Category)
Omelette
Chicken Bao
Angus Steak Main
Carrot Cake Dessert
Chips Side
Fish and Chips Main
Fish soup Starter
Hamburguer Main
MonkFish Main
Pudding Dessert
Salad Side

Figura 5.4: Resultado de la ejecución de la consulta transformada.

5.3. Consulta Simple (WHERE)

En este apartado vamos a evaluar la aplicación del caso B.5. Para ello se utilizará una consulta simple con una ocurrencia en la cláusula WHERE.

Se desea consultar aquellos platos que no pertenecen a la categoría `main` ni a la categoría `starter` utilizando la consulta

```
SELECT coddish, name, category
FROM dishes
WHERE category NOT IN ('Main','Starter');
```

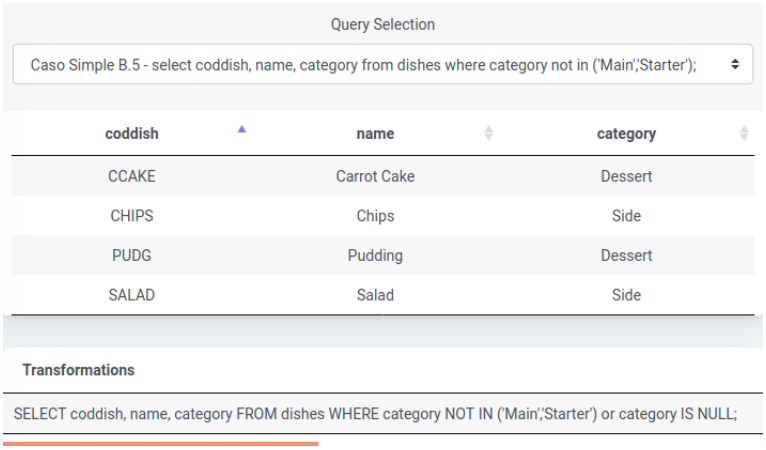

Figura 5.5: Resultado de la ejecución de la consulta.

El resultado de esta consulta omite resultados puesto que no muestra los platos que no tienen categoría, estos platos tampoco pertenecen a las categorías antes mencionadas por lo que deberían aparecer. La herramienta detecta el patrón B.2 y automáticamente aplica la transformación correspondiente, generando la consulta:

```
SELECT coddish, name, category
FROM dishes
WHERE category NOT IN ('Main','Starter') or category IS NULL;
```

Ahora se puede observar en la Figura 5.6 como el resultado devuelve los dos platos que no tienen una categoría asignada.

coddish	name	category
CCAKE	Carrot Cake	Dessert
CHIPS	Chips	Side
CKBAO	Chicken Bao	[null]
OMLT	Omelette	[null]
PUDG	Pudding	Dessert
SALAD	Salad	Side

Figura 5.6: Resultado de la ejecución de la consulta transformada B.2.

5.4. Subconsulta con el operador NOT IN

Se evalúa la aplicación del caso A.3 con una consulta que contenga una subconsulta en la cláusula WHERE.

Se desea consultar aquellos empleados que no preparan ningún plato utilizando la consulta:

```
SELECT name
FROM employees e
WHERE e.idemp NOT IN (SELECT distinct(d.idemp) FROM dishes d);
```

Esta consulta devuelve el conjunto vacío porque la condición de la subconsulta es evaluada como UNKNOWN. Al aplicar la transformación del caso A.3, se genera la consulta:

```
SELECT name
FROM employees e
WHERE NOT EXISTS (SELECT DISTINCT(idemp)
                  FROM dishes d
                  WHERE d.idemp = e.idemp);
```

Ahora se puede observar en la Figura 5.7 como el resultado cambia al utilizar la consulta transformada y que si existen empleados que no preparan ningún plato.

5.5. Subconsulta con operador de comparación

Se evalúa la aplicación del caso A.7 con una consulta que contenga una subconsulta en la cláusula WHERE. Se desea consultar aquellos ingredientes cuyo stock sea

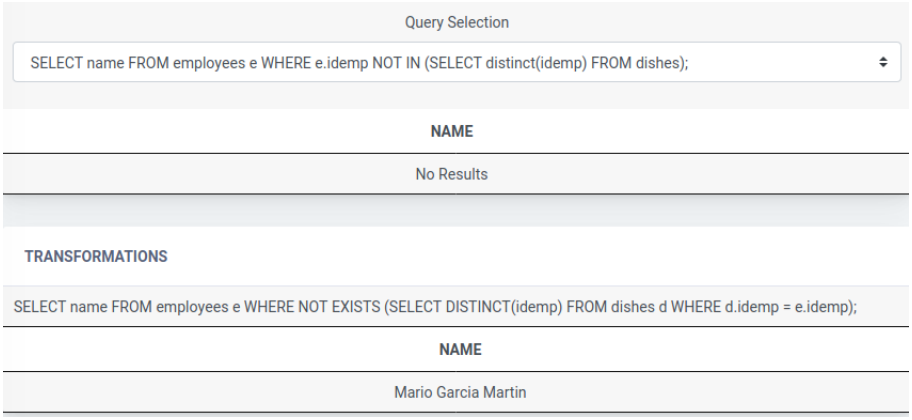

Figura 5.7: Resultado de la ejecución de la consulta original y la consulta transformada.

menor que el requerido para la elaboración de algún plato, utilizando la consulta:

```
SELECT  i.name, i.stock, r1.amount
FROM  ingredient AS i, recipes AS r1
WHERE r1.coding=i.coding and
stock < (SELECT MIN(amount)
         FROM recipes r2
         WHERE r2.coding =r1.coding)
```

Al aceptar y contener valones NULL las columnas `stock` y `amount` hace que la consulta omita la información de los ingredientes de los cuales no se sabe exactamente la cantidad y de los ingredientes que pueden ser opcionales en una receta.

Al aplicar la transformación del caso A.7, se genera la consulta:

```
SELECT  i.name, i.stock, r1.amount
FROM ingredient AS i, recipes AS r1
WHERE r1.coding=i.coding AND
   (stock IS NULL OR (stock < (SELECT MIN(amount)
                               FROM recipes r2 where r2.coding =
                                  r1.coding)) OR (SELECT MIN(
                                  amount)
                               FROM recipes r2 WHERE r2.coding =
                                  r1.coding) IS NULL)
```

Se puede observar en la Figura 5.8 la diferencia de los resultados. Una vez se ha

hecho la transformación, ahora se muestran aquellos ingredientes de los cuales no se sabe su cantidad en stock y aquellos que son opcionales.

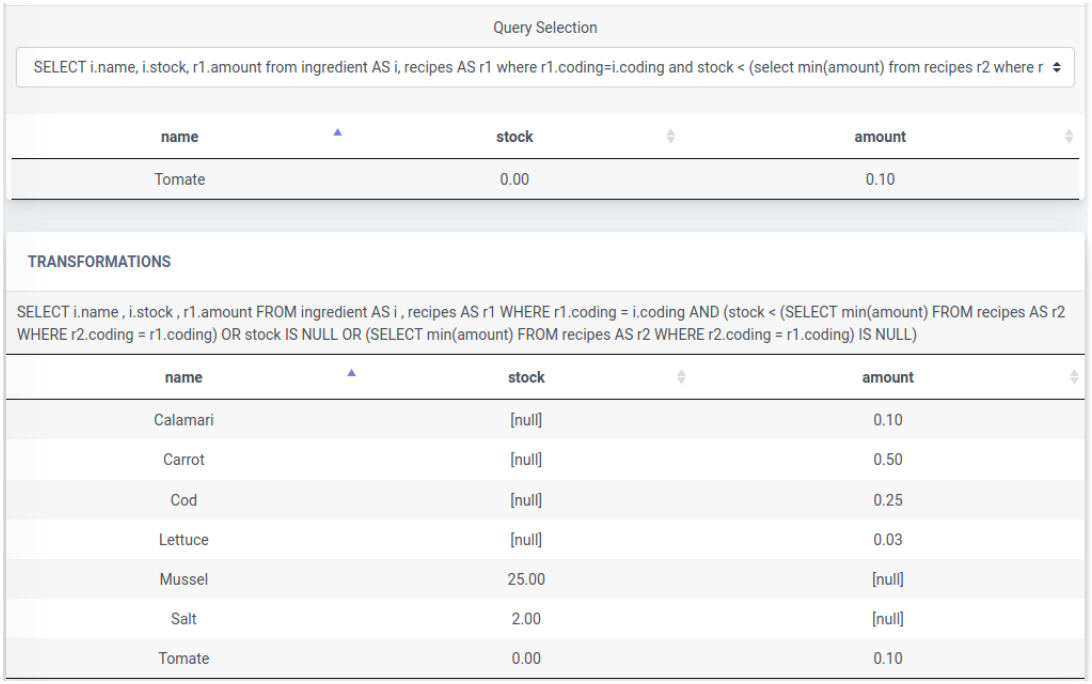

Figura 5.8: Resultado de la ejecución de la consulta original y la consulta transformada A.7.

Capítulo 6

Manual de Usuario

Resumen: En este capítulo se detalla el proceso de instalación y uso de la herramienta propuesta. Es importante resaltar que el sistema se ha probado en un entorno Linux. Por lo tanto, este manual especifica la instalación de la herramienta en dicho entorno.

6.1. Instalación

El prototipo de la herramienta que se describe en este trabajo se encuentra en la plataforma de control de versiones GitHub, donde se puede acceder al repositorio a través del siguiente enlace `https://github.com/GonzaloMachado/TFM-Metamorphic.git`. Al no estár alojada en un servidor, se deben realizar las siguientes configuraciones para su uso local.

La herramienta está desarrollada en Python 3.6 sobre el marco de desarrollo web Django 3.0 y PostgreSQL 12.3, por lo que se deben tener instaladas estas plataformas antes de iniciar. A continuación se indican las instrucciones necesarias para instalar esta herramienta.

La instalación de Python incluye la herramienta PIP, el gestor e instalador de paquetes de Python, que permite añadir diferentes paquetes desde PyPi (Python Package Index) un repositorio de software para el lenguaje de programación Python que usaremos mas adelante. Aunque es opcional, se recomienda utilizar *venv* para

generar un entorno aislado para la instalación de paquetes. Para crear un entorno nuevo, se elige el directorio deseado y se ejecuta el comando:

```
$ python3 -m venv my-env
```

Una vez creado el ambiente, se debe activar el mismo:

```
$ source my-env/bin/activate
(my-env)$
```

A continuación se descarga el repositorio (`git clone`) en el directorio `/tfmapp` y se instalan las dependencias o requerimientos del proyecto utilizando:

```
(my-env)$ pip3 install -r requirements.txt
```

El fichero `requirements.txt` contiene la información acerca de los paquetes necesarios y su versión correspondiente para hacer uso de la herramienta. Los principales paquetes utilizados se describen a continuación:

- **Cython v0.28.5**: Cython es un lenguaje que permite desarrollar extensiones del lenguaje C para Python, admite la llamada a funciones en C y la declaración de tipos de variables C en atributos de clase. Esto hace que Cython sea el lenguaje ideal para empaquetar librerías externas C que aceleran la ejecución del código Python.
- **Django Settings Export v1.2.1**: Es un módulo que permite acceder a las configuraciones internas de la aplicación Django (backend) en las plantillas HTML (frontend).
- **Django Widget Tweaks v1.4.8**: Es un módulo que permite personalizar la presentación de los campos de un formulario utilizando CSS o JavaScript sin tener que modificar el código en Python.
- **Psycopg2 v2.8.5**: Psycopg2 es un módulo adaptador de PostgreSQL para Python que permite acceder a la base de datos utilizando un objeto para la creación de conexiones.

Este fichero incluye la instalación del marco web Django descrito en el capítulo anterior. A continuación se debe configurar la base de datos que se utilizará la aplicación para los usuarios, instancias y consultas. Para ello se modifica el archivo settings.py en el apartado DATABASE

Luego, se debe sincronizar el estado de la base de datos con el conjunto actual de modelos y migraciones utilizando los siguientes comandos:

```
(my-env)$ python3 manage.py makemigrations
(my-env)$ python3 manage.py migrate
```

(Opcional) Se puede crear una cuenta de administrador para manejar información relacionada con la aplicación (usuarios, instancias, consultas) utilizando el comando

```
(my-env)$ python3 manage.py createsuperuser
```

Para finalizar, se debe iniciar el servidor con el archivo manage.py, también ubicado en el directorio /tfmapp. Para ello se utiliza el comando:

```
(my-env)$ python3 manage.py runserver
```

Una vez iniciado el servidor se accede a la herramienta con el enlace
`http://127.0.0.1:8000/home/`

6.2. Uso de la Herramienta

A continuación se explica cada uno de los módulos desarrollados y su funcionamiento con detalle.

6.2.1. Registro e Inicio de Sesión

Al acceder al enlace anterior, nos llevará a la página de inicio de sesión donde debemos registrarnos en la aplicación a través del formulario correspondiente rellenando la información básica para la creación de un nuevo usuario como se muestra en la Figura 6.1

- Nombre.
- Apellido.
- Usuario.
- Correo Electrónico.
- Contraseña.

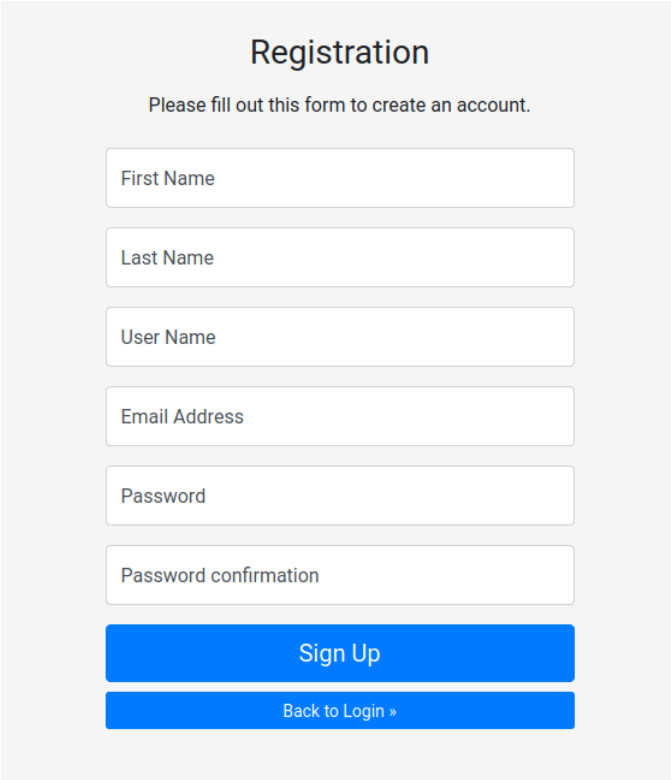

Figura 6.1: Registro de Usuario

En caso de olvidar la contraseña se podrá recuperar en cualquier momento haciendo click sobre la opción *I forgot my password*. Se deberá ingresar el correo electrónico proporcionado en el registro y nos llegará un correo con los pasos a seguir para restablecerla tal y como se muestra en las Figuras 6.2, 6.3 y 6.4

Una vez registrados, podremos iniciar sesión con nuestras credenciales a través del formulario de la Figura 6.5 y posteriormente se mostrará el menú principal con las diferentes opciones como se muestra en la Figura 6.6

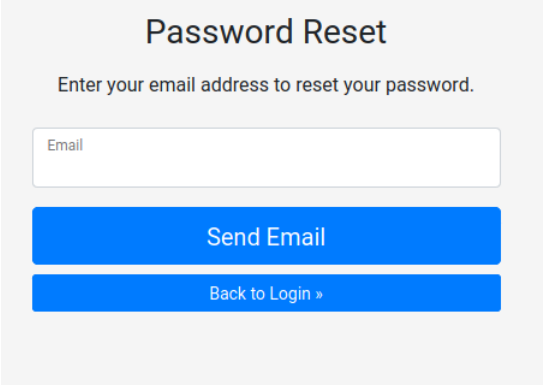

Figura 6.2: Restablecer contraseña

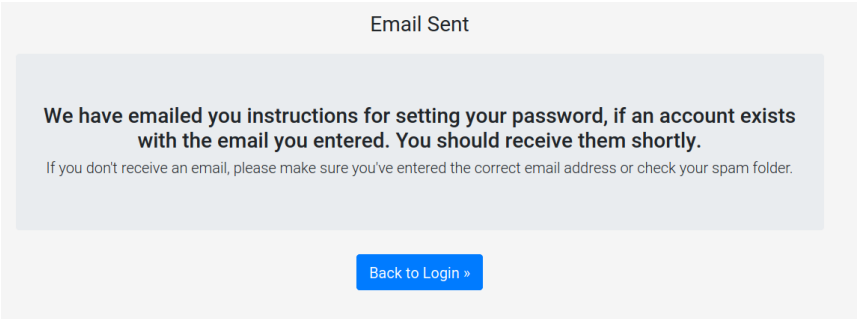

Figura 6.3: Correo Enviado

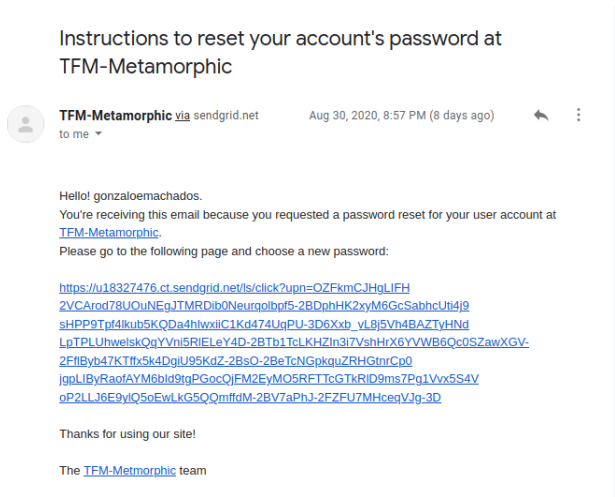

Figura 6.4: Correo para restablecer contraseña

6.2.2. *Database Instances*

Haciendo click en el apartado de instancias de base de datos, se podrá consultar y editar las instancias previamente guardadas o añadir una nueva pulsando el botón

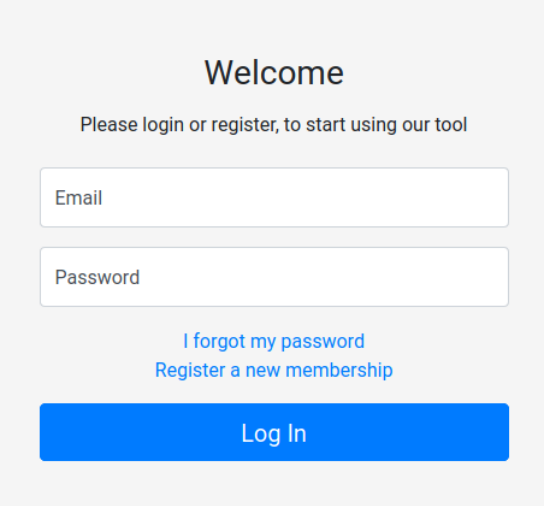

Figura 6.5: Inicio de Sesión

con el símbolo '+'. Para ello se deberá proporcionar la siguiente información.

- Nombre: El nombre de la base de datos a la que nos queremos conectar.
- Usuario: Nombre del usuario para la autenticación.
- Contraseña: Contraseña para la autenticación.
- Host: Dirección de host donde está alojada la base de datos. (P/ej: localhost)
- Puerto: Número del puerto de conexión a la base de datos. (P/ej: 5432 para PostgreSQL)

Las instancias serán utilizadas para establecer una conexión con una base de datos en particular donde se desea realizar el testing metamórfico. Es importante destacar que para poder almacenar una nueva instancia, la base de datos debe existir y el servidor debe estar ejecutándose ya que el sistema intentará realizar una conexión para validar su uso.

6.2.3. *Queries*

En esta sección se permite consultar editar o agregar nuevas consultas SQL. Al igual que las instancias, la sentencia debe ser sintácticamente correcta y ejecutable para poder ser almacenada. En la figura 6.8 se muestra esta sección.

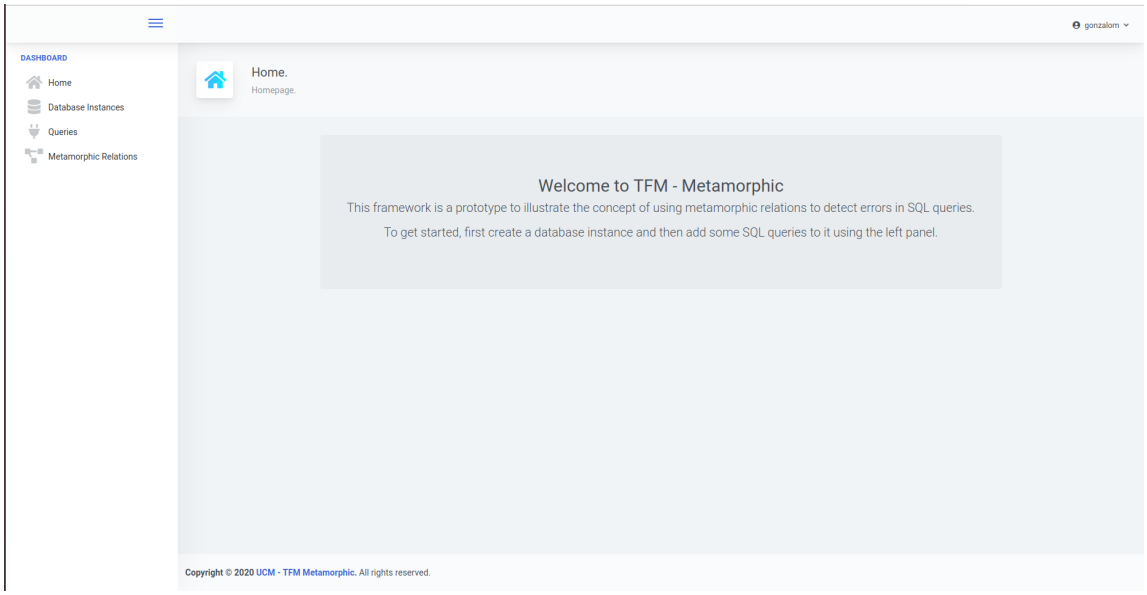

Figura 6.6: Página Inicial

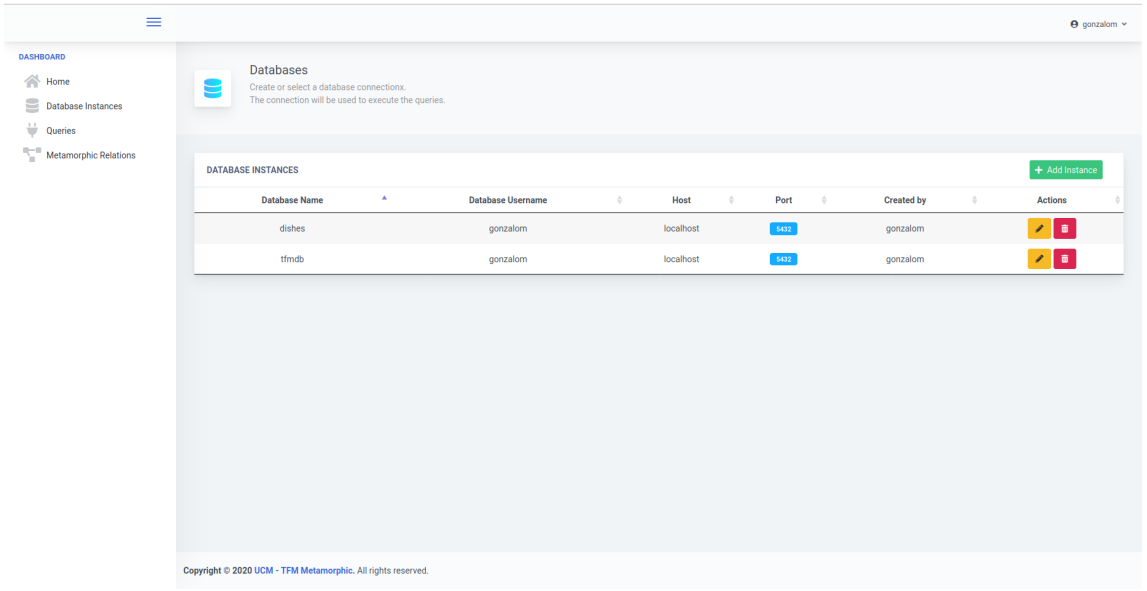

Figura 6.7: Instancias de Bases de Datos

El sistema pedirá un string con la consulta SQL y se deberá seleccionar a que instancia de base de datos pertenece dicha consulta entre la lista de instancias previamente almacenadas.

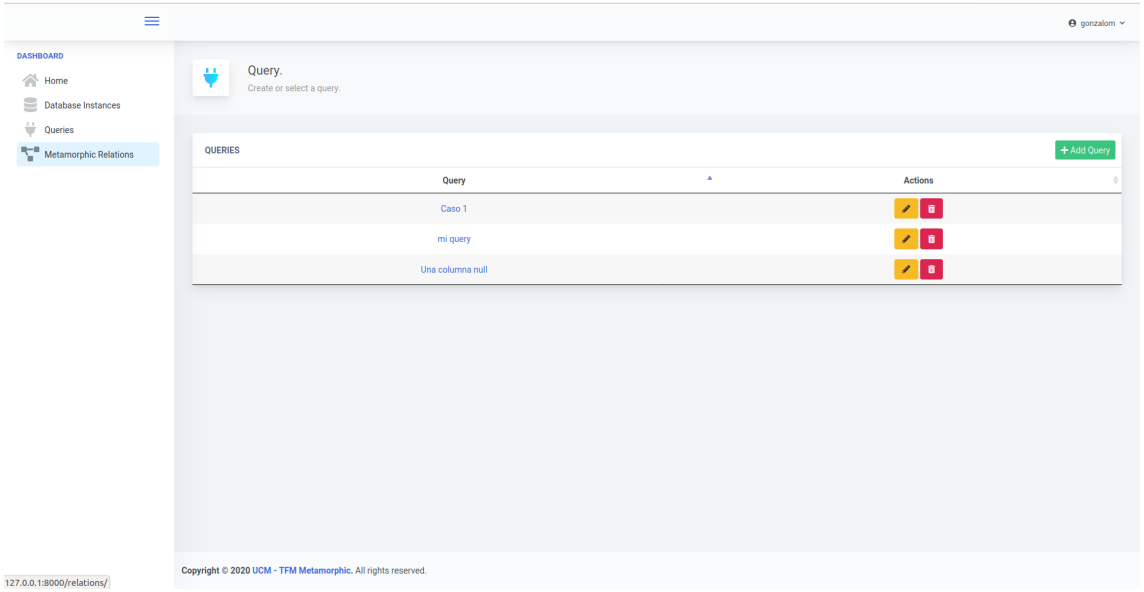

Figura 6.8: Consultas SQL

6.2.4. *Metamorphic Relations*

Como se muestra en la Figura 6.9 en esta sección se podrán elegir entre las diferentes consultas almacenadas para así iniciar el proceso de búsqueda y aplicación de relaciones metamórficas.

Al seleccionar una consulta, se mostrará el resultado de ejecutar la sentencia sobre la instancia a la cual pertenece. Adicionalmente, si alguna de las transformaciones es aplicable, se se mostrará dicha transformación y el resultado de ejecutar la sentencia transformada.

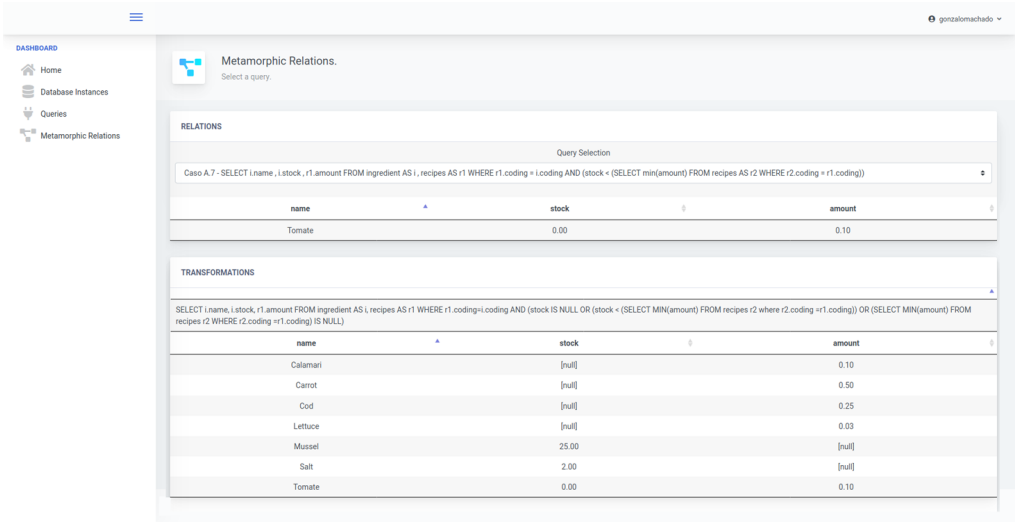

Figura 6.9: Metamorphic

Capítulo 7

Conclusiones y Trabajo Futuro

Resumen: En este capítulo se presentan las conclusiones obtenidas una vez desarrollado el proyecto así como las recomendaciones y posibles mejoras que se podrían implementar para extender la funcionalidad del sistema y darle continuidad a este trabajo.

7.1. Conclusiones

Una vez finalizado el diseño, desarrollo e implementación de la herramienta propuesta en este trabajo y habiendo cumplido con todas las historias de usuario detalladas en la Tabla 4.1, se ha alcanzado el objetivo general del presente trabajo garantizando que el sistema funciona correctamente y obteniendo, como producto final, el prototipo de una aplicación web que genera de manera automática una serie de transformaciones a las consultas SQL utilizando plantillas metamórficas que ayudan a los usuarios o testers detectar posibles errores en la ejecución de dichas consultas en donde intervengan valores NULL.

A continuación se presentan los resultados según los objetivos específicos expuestos en el Capítulo 1:

- Estudiar la bibliografía: Se investigó la literatura relacionada con el testing de bases de datos y su relación con las bases de datos incompletas. A pesar de

conseguir información sobre otros tipos de testing sobre bases de datos, no se encontró ningún estudio acerca del uso de testing metamórfico sobre bases de datos con valores nulos.

- Análisis de valores NULL: Una vez investigado las diferentes interpretaciones que pueden tomar los valores nulos, se realizó un análisis haciendo uso de ellos en diferentes consultas SQL, observando en detalle que los resultados dependen en gran medida del significado que se dé a los valores nulos.

- Aplicabilidad de la técnica de testing metamórfico: se han planteado varios esquemas de relaciones metamórficas que permitan automatizar el testing metamórfico de consultas SQL.

- Se ha construido un prototipo que nos permite evaluar la viabilidad de esta técnica: Una vez terminados cada uno de los módulos antes descritos, se procedió a la integración de los mismos, se comprobó que se siguiera el flujo deseado de la aplicación y la interoperabilidad entre ellos, garantizando su correcto funcionamiento.

- Se han evaluado los resultados obtenidos con el prototipo desarrollado: Se ha diseñado una base de datos de prueba en la que participaban columnas sensibles a ocurrencias del valor NULL dándoles una interpretación distinta a cada una de ellas y posteriormente se han utilizado algunas consultas SQL para probar y demostrar el funcionamiento la herramienta.

Este trabajo es la versión inicial de la herramienta que hemos denominado "TFM - Metamorphic" en la que se han desarrollado las operaciones básicas para la utilización del sistema por parte de los usuarios, incluyendo el registro e inicio de sesión de los mismos.

7.2. Trabajo Futuro

Este trabajo sirve como base para seguir realizando nuevas funcionalidades sobre la herramienta e ir mejorando los procesos. Se debe evaluar la infraestructura

y todos sus componentes en búsqueda de fallas o posibles mejoras. Aunque la herramienta desarrollada cumple con los objetivos fijados al inicio de este trabajo, al ser el lenguaje SQL tan extenso y el ámbito del testing metamórfico sobre bases de datos con valores NULL tan poco explorado se exponen algunas posibles mejoras que se podrían realizar.

- Interfaz de Usuario: Al ser esta herramienta un prototipo inicial, se utilizó una plantilla HTML sencilla que cubriera las necesidades básicas del sistema, sin embargo, se podría mejorar ciertos aspectos gráficos y así hacerla más agradable e intuitiva para los usuarios.

- Este prototipo está diseñado para ser utilizado de forma local por un desarrollador. Se puede extender para instalarlo en un servidor y que lo pueda utilizar un equipo de programación compartiendo las bases de datos de pruebas y las consultas introducidas y transformaciones generadas.

- Sentencias SQL: Como se menciona anteriormente, el lenguaje SQL es un lenguaje muy potente y extenso, esta herramienta solo contempla parte de la sintaxis de la sentencia SELECT, pero en un futuro se pudiera extender al uso de toda la sintaxis del SELECT así como también otras sentencias.

  En particular, aunque requiere un estudio en profundidad, es especialmente interesante estudiar la aplicabilidad de esta técnica a las consultas agregadas que utilizan cláusulas GROUP BY y especialmente condiciones sobre grupos en cláusulas HAVING.

- De forma análoga, esta técnica se puede aplicar a sentencias de modificación de datos: INSERT, DELETE y UPDATE. En todas ellas se pueden establecer condiciones complejas en una cláusula WHERE. De especial interés resulta la aplicación de esta técnica sobre subconsultas dentro de la cláusula SET de sentencias UPDATE, pues permiten expresar subconsultas SELECT en el punto preciso en el que se realiza la evaluación de los valores a modificar en las filas afectadas por la sentencia.

- Otro área de aplicación relevante en bases de datos complejas se encuentra en los procedimientos almacenados que utilizan cursores para recorrer estas consultas. En este caso, el lenguaje procedural puede incluir evaluaciones sobre valores nulos, lo que que la aplicación de la técnica sea más compleja. De la misma forma, se puede estudiar la aplicabilidad a disparadores de bases de datos *activas*, que ejecutan código procedural como resultado de eventos de modificación de datos sobre tablas.

- Sistemas Gestores de Bases de Datos: La herramienta solo contempla el uso de PostgreSQL, pero se puede extender la compatibilidad de la herramienta con otros SGBD comerciales.

Chapter 8

Introduction

Resumen: This chapter introduces the fundamental motivations for the development of this work, as well as its objectives. It also includes schematically the work plan followed to develop this Master's Thesis and jcf the general structure of the rest of this document .

8.1. Motivation

Since the introduction of Codd's Relational Model in 1970, SQL-based relational databases have become the most relevant technology for data representation, storage and information retrieval in the industry. The successive extensions of the model have made it possible to solve complex technical problems related to various aspects such as concurrency of transactions, distribution, etc. One of the mechanisms that has been used from almost the beginning has been the possibility of entering incomplete information: certain data from the database that are either unknown or not available at the time the information is entered into the database or not applicable in the corresponding particular context. A way to represent this incomplete information has been through null values. However, despite the fact that relational databases have a great expressive capacity to represent the structure of the data and build complex queries on its contents in a robust way, this way of representing incomplete information can affect the quality of both data and query results. Al-

though the completeness of the data is an essential aspect of the quality of the same, it is crucial to guarantee the completeness of the answers to the queries, in many scenarios. It is natural not to have all the necessary information. It is generally assumed that a database is complete, and application developers generally assume this fact. For example, in a relational database each relation is considered to contain all the tuples that need to appear in the relation. However, there are situations where the information may be partial, for example some tuples may be missing and the database is said to be incomplete. Data can be incomplete in two ways: records can be missing as a whole, or a record's attribute values can be absent, indicated by a NULL value. This is used to represent an *UNKNOWN* value or the absence of it. Importantly, NULL is different from a 0 value. If the database is partial or incomplete, it is necessary to reconsider the meaning of the query result against the database and analyze whether a response to a particular query is guaranteed to be complete even in the presence of null values. (Levy, 1996; Nutt et al., 2012)

This work focuses on the definition of a metamorphic testing framework in order to identify possible errors in the results obtained from the execution of SQL queries that can deal with NULL values. This testing technique is used in those situations in which there is no oracle that provides the expected results of the execution of the tests and is based on defined relationships between the tests and the results obtained from the application of those tests. To this date, there are works focused on testing the SQL language and databases, however we have no evidence of any implementation of a metamorphic testing framework applied to databases with null values.

8.2. Objectives

The main objective of this work is to develop and implement the prototype of a web tool that allows detecting possible errors in SQL queries with null values using metamorphic relationships and thus being able to perform tests and verify the results automatically. The metamorphic relationships will be based on transformations of code fragments of the original queries to compare the results of the execution of

different SELECT statements in order to help determine how the possible NULL values affect those results.

The specific objectives for this work are the following:

- Study of the literature related to database testing and its relation with incomplete databases.
- Analyze the different interpretation of NULL values and the problems that incomplete databases produce when designing complex queries.
- Evaluate the applicability of the metamorphic testing technique to propose possible transformations in which the developer can identify errors related to NULL values in SQL queries.
- Propose metamorphic relationship schemas that allow automating metamorphic testing of SQL queries.
- Build a prototype that allows us to evaluate the viability of this technique.
- Evaluate the results obtained with the developed prototype.

8.3. Work Plan

For the preparation of this work the main tasks to develop were identified and then divided into the following phases.

- Phase 0: Evaluation of the problem.
- Phase 1: Research and review of previous works.
- Phase 2: Analysis of metamorphic relationships.
- Phase 3: Evaluation of the applicability of metamorphic testing on incomplete databases.
- Phase 4: Approach to metamorphic relationships.
- Phase 5: Definition of metamorphic relationship templates.

- Phase 6: Design and implementation of the modules that make up the prototype.

- Phase 7: Prototype evaluation.

Additionally, for the supervision and monitoring of this work, weekly meetings are proposed to maintain order in the project and keep up to day with any news. These meetings will also serve to make continuous reviews and thus be able to identify problems during their development, establish if the objectives of the phase have been achieved and plan guidelines to continue with the development.

8.4. Document Structure

The scheme of this work is organized in 7 chapters and its content is described below:

- Chapter 1 - Introduction: Exposes the motivation of the work done and the objectives set at the beginning of it.

- Chapter 2 - Preliminaries: Exposes the theoretical foundations which results in the proposed system. It includes the basic concepts used to understand this work.

- Chapter 3 - Metamorphic Testing Framework: It is the central Chapter, in which the developed proposal is presented.

- Chapter 4 - Implementation: Describes in detail the development process of the tool, the different operations it performs and its main uses.

- Chapter 5 - Evaluation: A case study is presented for the evaluation of the tool.

- Chapter 6 - User Manual: Details the installation and use of the tool.

- Chapter 7 - Conclusions and Future Work: Exposes the conclusions that can be drawn about the work done once the proposed solution has been implemented. Additionally, possible modifications to the tool that could improve its operation are indicated.

Chapter 9

Conclusions and Future Work

Resumen: This chapter presents the conclusions obtained once the project has been completed, as well as the recommendations and possible improvements that could be implemented to extend the functionality of the system and give continuity to this work.

9.1. Conclusions

Once the design, development and implementation of the tool proposed in this work was completed and having complied with all the user stories detailed in Table 4.1, the general objective of this work was achieved, guaranteeing that the system works correctly and obtaining as a final product, the prototype of a web application that automatically generates a series of transformations to SQL queries using metamorphic templates that help users or testers detect possible errors in the execution of these queries where NULL values might be involved.

The results are presented below according to the objectives set in Chapter 8.

- Study of the bibliography: A search of the literature related to database testing and its relationship with incomplete databases was performed. Despite getting information on other types of testing on databases, we did not find any studies on the use of metamorphic testing on databases with null values.

- Analysis of null values: Once the different interpretations that null values can take had been investigated, an analysis was carried out using them in different SQL queries, observing in detail that the results depend to a great extent on the meaning given to null values.
- Applicability of the metamorphic testing technique: several schemas of metamorphic relationship schemes have been proposed to automate the metamorphic testing of SQL queries.
- A prototype has been built that allows us to evaluate the viability of this technique: Once each of the modules described above had been completed, they were integrated, it was verified that the desired flow of the application was followed and the interoperability between them, ensuring their correct operation.
- The results obtained with the developed prototype have been evaluated: A test database has been designed in which several columns sensitive to occurrences of the NULL value have been added, giving a different interpretation to each of them. Then, some SQL queries have been used to test and demonstrate the operation of the tool.

This project is the initial version of the tool that we have called "TFM - Metamorphic" in which the basic operations for the use of the system by users were developed, including their registration and login.

9.2. Future Work

This project serves as the basis for continuing to carry out new functionalities for the tool and to improve the processes. The infrastructure and all its components must be evaluated to detect faults or possible improvements. Although the developed tool meets the objectives set at the beginning of this project, since the SQL language is so extensive and the metamorphic testing field on databases with NULL values so little explored, some possible improvements that could be made are exposed.

- User Interface: As this project was an initial prototype, a simple HTML template was used to cover the basic needs of the system, however, certain graphic aspects could be improved and thus make it more pleasant and intuitive for users.

- SQL Statements: As mentioned above, the SQL language is a very powerful and extensive language, this tool only considers part of the syntax of the SELECT statement, but in the future it could be extended to the entire SELECT syntax as well as other statements.

  In particular, although it requires an in-depth study, it is especially interesting to study the applicability of this technique to aggregate queries that use GROUP BY clauses and especially conditions on groups in HAVING clauses.

- Database Management Systems: The tool only contemplates the use of PostgreSQL, but since there are other DBMS that may be preferred by the user, the compatibility of the tool with some of them could be extended.

- Similarly, this technique can be applied to data modification statements: INSERT, DELETE and UPDATE. In all of them, complex conditions can be established in a WHERE clause. Of special interest is the application of this technique on subqueries within the SET clause of UPDATE statements, since they allow to express SELECT subqueries at the precise point at which the evaluation of the NULL values to be modified in the rows affected by the statement is performed.

- Another relevant application area in complex databases is stored procedures that use cursors to traverse these queries. In this case, the procedural language can include evaluations on null values, which makes the application of the technique more complex. In the same way, the applicability to *active* database triggers can be studied, which execute procedural code as a result of data modification events on tables.

Bibliografía

Abran, A. y Moore, J. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

Codd, E. F. A relational model of data for large shared data banks. *Commun. ACM*, vol. 13(6), páginas 377–387, 1970.

Codd, E. F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, vol. 4(4), página 397–434, 1979. ISSN 0362-5915.

Culquicondor, A. psqlparse. `https://github.com/alculquicondor/psqlparse`, 2017.

Elmasri, R. y Navathe, S. B. *Fundamentals of Database Systems*. Pearson, 7th edición, 2015. ISBN 0133970779.

Fielding, R. Hypertext transfer protocol – http/1.1. 1999. Disponible en `http://www.ietf.org/rfc/rfc2616.txt` (último acceso, Agosto, 2020).

Fumàs, E. ¿qué es ajax? ¿para qué sirve? 2013. Disponible en `http://www.ibrugor.com/blog/que-es-ajax-para-que-sirve/` (último acceso, Agosto, 2020).

George, N. *Mastering Django*. GNW Independent Publishing, 2015.

Group, T. P. G. D. About postgresql. 2020.

Guagliardo, P. y Libkin, L. Correctness of sql queries on databases with nulls. *SIGMOD Rec.*, vol. 46(3), página 5–16, 2017. ISSN 0163-5808.

Guagliardo, P. y Libkin, L. On the codd semantics of SQL nulls. *Inf. Syst.*, vol. 86, páginas 46–60, 2019.

Gutiérrez, J. ¿qué es un framework web? 2007. Disponible en `http://www.lsi.us.es/~javierj/investigacion_ficheros/Framework.pdf` (último acceso, Agosto, 2020).

Hetzel, W. *The Complete Guide to Software Testing, Second Edition*. John Wiley & Sons, 1988.

IBM. Ibm knowledge center. 2020. Disponible en `https://www.ibm.com/support/knowledgecenter/es/SSEPGG_11.5.0/com.ibm.db2.luw.glossary.doc/doc/glossary.html` (último acceso, Agosto, 2020).

Imielinski, T. y Lipski, W. Incomplete information in relational databases. *J. ACM*, vol. 31(4), página 761–791, 1984. ISSN 0004-5411.

Kleene, S. C. On Notation for Ordinal Numbers. *The Journal of Symbolic Logic*, vol. 3(4), páginas 150–155, 1938. ISSN 0022-4812.

Levy, A. Y. Obtaining complete answers from incomplete databases. En *VLDB*, vol. 96, páginas 402–412. Citeseer, 1996.

Myers, G., Sandler, C., Badgett, T. y Thomas, T. *The Art of Software Testing*. Business Data Processing: A Wiley Series. Wiley, 2004.

Network, M. D. Django tutorial part 5: Creating our home page. 2017. Disponible en `https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Home_page` (último acceso, Agosto, 2020).

Network, M. D. Css. 2020a. Disponible en `https://developer.mozilla.org/es/docs/Web/CSS` (último acceso, Agosto, 2020).

Network, M. D. Html. 2020b. Disponible en `https://developer.mozilla.org/es/docs/Web/HTML` (último acceso, Agosto, 2020).

Nutt, W., Razniewski, S. y Vegliach, G. Incomplete databases: Missing records and missing values. vol. 7240, páginas 298–310. 2012.

Pan, J. Software testing [scholarly project]. 1999. Disponible en `http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/index.html` (último acceso, Agosto, 2020).

PowerData. ¿qué es el sistema manejador de base de datos? 2015. Disponible en `http://blog.powerdata.es/el-valor-de-la-gestion-de-datos/bid/406549/Qu-es-el-sistema-manejador-de-bases-de-datos.` (último acceso, Agosto, 2020).

Rodríguez, J. Definición de javascript. 2005. Disponible en `https://www.gestiopolis.com/definicion-javascript/` (último acceso, Agosto, 2020).

SendGrid, T. Why sendgrid. 2020. Disponible en `https://sendgrid.com/why-sendgrid/` (último acceso, Agosto, 2020).

Apéndice A

Glosario

- **HTML:** HTML, siglas en inglés de HyperText Markup Language o Lenguaje de marcas de hipertexto, es la pieza más básica para la construcción de la web y se usa para definir el sentido y estructura del contenido en una página web. Dicho lenguaje es interpretado por un navegador para su posterior visualización por parte del usuario. (Network, 2020b)

- **CSS:** CSS, siglas en inglés de Cascade Style Sheet u Hoja de Estilo en Cascada, es un lenguaje que describe cómo se va a mostrar un documento en la pantalla y cómo se renderizan los diferentes elementos. En otras palabras, permite personalizar o dar formato de diseño a los diferentes objetos presentes en este caso en una página web realizada en HTML. (Network, 2020a)

- **Bootstrap:** Es un framework para el desarrollo y creación de interfaces web que está basado en HTML, CSS y JavaScript y adapta su visualización a cualquier dispositivo.

- **JavaScript:** "JavaScript es un lenguaje de "scripting" (programación ligera) interpretado por casi todos los navegadores, que permite añadir a las páginas web efectos y funciones adicionales a los contemplados en el estándar HTML." (Rodríguez, 2005)

- **AJAX:** "Ajax es el acrónimo de Asynchronous Javascript and XML, es decir, Javascript y XML Asíncrono. El principal objetivo del AJAX, es intercam-

biar información entre el servidor y el cliente (navegadores) sin la necesidad de recargar la página. De esta forma, ganamos en usabilidad, experiencia y productividad del usuario final." (Fumàs, 2013)

- **Framework.** "En general, con el término framework, nos estamos refiriendo a una estructura software compuesta de componentes personalizables e intercambiables para el desarrollo de una aplicación. En otras palabras, un framework se puede considerar como una aplicación genérica incompleta y configurable a la que podemos añadirle las últimas piezas para construir una aplicación concreta."(Gutiérrez, 2007)

- **HTTP:** HTTP o protocolo de la transferencia de hipertexto, es el protocolo de aplicación de petición y respuesta de los modelos clientes servidor. (Fielding, 1999)

- **Aplicación Web** Es un tipo de aplicación basada en la arquitectura cliente-servidor que utiliza el protocolo HTTP, dicha aplicación se ejecuta directamente en un navegador web y puede ser codificada en diferentes lenguajes de programación soportados por el navegador, por lo cual no depende del sistema operativo o de la instalación adicional de otro software.

- **Cliente:** Se llama cliente a cualquier dispositivo que se conecte a un servidor para consumir los servicios que este suministra.

- **Servidor:** El servidor es un programa que siempre está a la espera de solicitudes prevenientes de los clientes que se conecten a él utilizando el protocolo de comunicación HTTP.

- **Bases de Datos Relacionales:** Una base de datos relacional es una base de datos que se trata como un conjunto de tablas y se manipula de acuerdo con el modelo de datos a través de relaciones entre ellas. Contiene un conjunto de objetos que se utilizan para almacenar y gestionar los datos, así como para acceder a los mismos. Las tablas, vistas, índices, funciones, activadores y paquetes son ejemplos de estos objetos. (IBM, 2020)

- **Sistema Gestor de Bases de Datos (SGBD):** Un sistema gestor de bases de datos o DataBase Management System (DBMS) es una colección de software muy específico, cuya función es servir de interfaz entre la base de datos, el usuario y las distintas aplicaciones utilizadas. Como su propio nombre indica, el objetivo de los sistemas gestores de base de datos es precisamente el de manejar un conjunto de datos para convertirlos en información relevante para la organización, ya sea a nivel operativo o estratégico. Su uso permite realizar un mejor control a los administradores de sistemas y, por otro lado, también obtener mejores resultados a la hora de realizar consultas que ayuden a la gestión empresarial mediante la generación de la tan perseguida ventaja competitiva. (PowerData, 2015)

- **SendGrid:** SendGrid es una herramienta para gestión de correos transaccionales fundada en el año 2009. (SendGrid, 2020). Está basado en tecnologías de nube para servicios de correo transccionales. Un correo transaccional es una funcionalidad que envía uno o más correos por cada operación realizada en un sistema. Sendgrid provee un API con el cual es posible interactuar y realizar operaciones de envío de correos masivamente o individuales dado un estado del sistema. Además del API, esta herramienta provee con un sistema para crear plantillas de correo predeterminadas que pueden contener elementos sustituibles para personalizar los correos enviados a la comunidad suscrita.