Semantic analysis of SQL queries in DES

Análisis semántico de consultas SQL en DES

TRABAJO DE FIN DE GRADO

Doble Grado en Ingeniería Informática y Matemáticas

Autor:
Javier Amado Lázaro

Director:
Fernando Sáenz Pérez

Nota asignada: **8.6**

Universidad Complutense de Madrid

25 de mayo de 2025

Agradecimientos

A mi familia, en especial a mis padres, por educarme con valores, por su apoyo constante, por enseñarme a no rendirme nunca, por su cariño y por hacer posible que nunca me faltara de nada.

A Lucía, por motivarme, por enseñarme, por ser una gran fuente de inspiración y un ejemplo a seguir, por su apoyo incondicional y por creer en mí siempre.

A mis amigos de Leganés y del tenis, por confiar en mí y animarme en cada paso.

A los amigos que he conocido en esta maravillosa etapa, en especial a los *escape roomers*, por todas las experiencias y momentos que hemos vivido juntos, tanto buenos como malos.

A Fernando Sáenz Pérez, por dedicarme su tiempo, por su atención en todo momento, por todo lo que he aprendido y porque sin él este trabajo no habría sido posible.

Resumen

DES es un sistema de bases de datos deductivas que incorpora SQL en particular como lenguaje de consulta. Una de sus características distintivas es la capacidad de realizar análisis semántico sobre consultas SQL con el fin de detectar sentencias que, aunque sintácticamente son correctas, presentan posibles errores desde el punto de vista semántico.

Este trabajo amplía las funciones del sistema DES para mejorar la detección de errores semánticos en consultas SQL. Para ello, se ha trabajado con el lenguaje de programación Prolog, incorporando en el módulo de análisis de consultas nuevos algoritmos que permiten detectar casos de errores semánticos no contemplados hasta el momento.

La motivación principal de este proyecto surge de la necesidad de mejorar la experiencia de aprendizaje de los estudiantes que usan la herramienta DES en asignaturas relacionadas con bases de datos. En este contexto, contar con herramientas que no sólo advierten de errores sintácticos, sino también de problemas semánticos que son más difíciles de detectar y corregir, puede contribuir de manera relevante a la formación académica de los alumnos.

Como punto de partida, se llevó a cabo un estudio detallado tanto del funcionamiento interno del análisis semántico que realiza la herramienta DES, como de la naturaleza de los errores semánticos más comunes. La selección de los errores a implementar se basó en estudios previos y datos estadísticos que analizan las dificultades que presentan los estudiantes al redactar consultas SQL, y respaldan la necesidad de mejorar esta funcionalidad para favorecer el aprendizaje autónomo y la correcta comprensión del lenguaje SQL.

Como resultado, se ha mejorado el sistema DES mediante la incorporación de nuevos algoritmos capaces de identificar errores semánticos sutiles en cláusulas SQL como `GROUP BY`, `ORDER BY`, `DISTINCT`, `UNION` o `JOIN`. Es importante destacar que todos estos algoritmos se han desarrollado en el contexto de consultas que utilizan la cláusula `SELECT`.

Palabras clave: DES, consulta, cláusula, error semántico, SQL.

Abstract

DES is a deductive database system that incorporates SQL in particular as a query language. One of its distinguishing features is the ability to perform semantic analysis on SQL queries in order to detect statements that, although syntactically correct, have possible errors from a semantic point of view.

This work extends the functions of the DES system to improve the detection of semantic errors in SQL queries. To this end, we have worked with the Prolog programming language, incorporating new algorithms in the query analysis module that allow us to detect cases of semantic errors that have not been considered until now.

The main motivation for this project arises from the need to improve the learning experience of students who use the DES tool in subjects related to databases. In this context, having tools that not only warn of syntactic errors, but also of semantic problems which are more difficult to detect and correct, can make a relevant contribution to students' academic training.

As a starting point, a detailed study of both the inner workings of the semantic analysis performed by the DES tool and the nature of the most common semantic errors was carried out. The selection of the errors to be implemented was based on previous studies and statistical data that analyse the difficulties that students have when writing SQL queries, and support the need to improve this functionality in order to favour autonomous learning and the correct understanding of the SQL language.

As a result, the DES system has been improved by incorporating new algorithms capable of identifying subtle semantic errors in SQL clauses such as `GROUP BY`, `ORDER BY`, `DISTINCT`, `UNION` or `JOIN`. It is important to note that all these algorithms have been developed in the context of queries using the `SELECT` clause.

Keywords: DES, query, clause, semantic error, SQL.

Índice general

Capítulo 1

Introducción

El presente Trabajo de Fin de Grado tiene como objetivo principal la mejora de la herramienta educativa DES [23], una plataforma utilizada por la Facultad de Informática de la Universidad Complutense de Madrid para la enseñanza de asignaturas relacionadas con bases de datos. A lo largo de los años, DES se ha consolidado como un recurso fundamental en la docencia práctica de materias relacionadas con el diseño y la manipulación de bases de datos relacionales, así como el aprendizaje del lenguaje SQL.

Dentro del ámbito del aprendizaje de SQL, uno de los desafíos más frecuentes para el alumnado es la aparición de errores en las consultas. Entre ellos, los errores semánticos son especialmente relevantes: se trata de consultas que, aunque están correctamente escritas desde el punto de vista sintáctico, no devuelven el resultado esperado debido a un mal planteamiento lógico o una comprensión incorrecta del problema.

Este trabajo se centrará en el estudio y clasificación de este tipo de errores semánticos en consultas SQL, con el objetivo de diseñar e implementar algoritmos que permitan detectarlos de forma automática. Estos algoritmos se integrarán en la herramienta DES para ampliar las funcionalidades actuales del sistema, que hasta ahora solo reconoce un número limitado de errores semánticos. De esta manera, se busca enriquecer la utilidad de DES como herramienta docente y mejorar la experiencia de aprendizaje de los estudiantes.

El análisis de estos errores se basará en el artículo [2], que ofrece una clasificación estructurada según las cláusulas del lenguaje SQL. Se seguirá dicha clasificación como marco de referencia, manteniendo también la numeración de los errores propuesta por el autor, ya que coincide con la utilizada por los desarrolladores originales del sistema DES. Esto facilitará una integración ordenada y consistente en la herramienta.

Antes de abordar estos desarrollos, se comenzará con una breve revisión del lenguaje SQL (Capítulo 2) [25], necesario para entender el tipo de errores que se analizarán. A continuación, se presentará más en detalle la herramienta educativa DES (Capítulo 3) y se explicará el uso de Prolog (Capítulo 4) [26], el lenguaje en el que está implementado DES y sobre el que se desarrollarán los nuevos algoritmos, así como una breve introducción a Datalog [10], una derivación de Prolog que también usa DES.

Finalmente, se describirán tanto los errores semánticos ya implementados en DES (Capítulo 5) como los que han sido incorporados en el marco de este proyecto (Capítulo 6), para concluir con un apartado dedicado a las conclusiones y posibles líneas de trabajo futuro (Capítulo 7). En el Apéndice I aparece una guía de usuario para probar la herramienta DES, mientras que el Apéndice II se trata de una muestra de las mejoras que ha experimentado DES gracias al desarrollo de este proyecto.

1.1. Objetivos

Como se ha comentado con anterioridad, la herramienta cuenta con funcionalidades para detectar errores sintácticos, pero su capacidad para identificar errores semánticos es limitada. Estos errores, que a menudo pasan desapercibidos, dificultan la comprensión del diseño y funcionamiento correcto de las bases de datos.

Con el fin de abordar esta carencia, el objetivo principal de este trabajo es ampliar la capacidad de DES para detectar errores semánticos, mediante la implementación de nuevos algoritmos específicamente diseñados para tal fin.

Los objetivos específicos del proyecto son los siguientes:

- Analizar las limitaciones actuales de DES en cuanto a la detección de errores semánticos.
- Investigar y categorizar los errores semánticos más comunes cometidos por los estudiantes en consultas SQL.
- Diseñar e implementar los algoritmos que permitan detectar dichos errores y añadirlos a la funcionalidad de la herramienta.
- Desarrollar un conjunto de pruebas con distintos casos para validar la efectividad de los algoritmos implementados.

Con este trabajo se pretende no solo mejorar el rendimiento y utilidad de DES, sino también contribuir al proceso de aprendizaje de los estudiantes, proporcionándoles un entorno más didáctico e inteligente que facilite la detección autónoma de errores semánticos y refuerce su comprensión del lenguaje SQL y del diseño de bases de datos.

1.2. Plan de trabajo

En primer lugar, ante la necesidad de adquirir y profundizar en los conocimientos necesarios para el desarrollo del proyecto, durante el mes de septiembre de 2024 se inició el estudio del lenguaje de programación Prolog, mediante la lectura de obras especializadas [26]. Tras realizar diversos ejercicios prácticos que aseguraban la correcta comprensión del lenguaje, se procedió al análisis del código ya implementado en el sistema DES.

De forma paralela, se comenzó un proceso de documentación teórica en torno a los errores semánticos en consultas SQL, prestando especial atención a su diferenciación respecto a los errores sintácticos. Esta fase incluyó la revisión de artículos científicos y material docente [1, 2, 16-18, 23] que permitió sentar las bases conceptuales necesarias para el posterior diseño e implementación de los algoritmos de detección de errores semánticos.

El proceso de estudio del código y fundamentación teórica se prolongó hasta el mes de febrero de 2025. Durante este período, se mantuvieron varias reuniones con uno de los desarrolladores principales de la herramienta, Fernando Sáenz Pérez, lo cual facilitó una comprensión más profunda del sistema.

Finalizados todos los procesos anteriores, en el mes de marzo se comenzó la implementación de nuevo código en DES. El primer paso fue revisar y corregir un error semántico previamente identificado, relacionado con el uso innecesario de la cláusula `DISTINCT`. Una vez solucionado este caso, se continuó con la incorporación de errores vinculados a la cláusula `GROUP BY`, que no habían sido tratados en la herramienta DES hasta ese momento. La mayoría de estos errores quedaron implementados a mediados de abril, a excepción del error 21 (siguiendo la numeración del artículo de Brass y Goldberg [2]), cuya complejidad requirió extender el desarrollo del mismo hasta principios de mayo.

A su vez, se comenzaron a desarrollar algoritmos para detectar errores semánticos en las cláusulas `ORDER BY` y `UNION`, además de corregir el falso positivo previamente identificado como Error 8.

Además, durante todo este proceso se fueron creando ficheros de prueba para validar el correcto funcionamiento de los algoritmos. Cabe destacar que, a principios de abril, también se inició la redacción de la memoria del proyecto, en paralelo con el desarrollo técnico, con el fin de documentar adecuadamente el trabajo realizado y facilitar su presentación final.

Introduction

The main objective of this Final Degree Project is the improvement of the DES [23] educational tool, a platform used by the Faculty of Computer Science of the Complutense University of Madrid for the teaching of database-related subjects. Over the years, DES has established itself as a fundamental resource in the practical teaching of subjects related to the design and manipulation of relational databases, as well as the learning of the SQL language.

Within the scope of SQL learning, one of the most frequent challenges for students is the appearance of errors in queries. Among them, semantic errors are particularly relevant: these are queries that, although correctly written from a syntactic point of view, do not return the expected result due to a bad logical approach or an incorrect understanding of the problem.

This work will focus on the study and classification of this type of semantic errors in SQL queries, with the aim of designing and implementing algorithms that allow them to be detected automatically. These algorithms will be integrated into the DES tool to extend the current functionalities of the system, which so far only recognises a limited number of semantic errors. In this way, the aim is to enrich the usefulness of DES as a teaching tool and improve the learning experience of students.

The analysis of these errors will be based on the article [2], which provides a structured classification according to SQL language clauses. This classification will be followed as a frame of reference, keeping also the numbering of the errors proposed by the author, as it coincides with the one used by the original developers of the DES application. This will facilitate an orderly and consistent integration into the tool.

Before addressing these developments, we will start with a brief review of the SQL language (Chapter 2) [25], which is necessary to understand the type of errors that will be analysed. This will be followed by a more detailed presentation of the DES educational tool (Chapter 3) and an explanation of the use of Prolog (Chapter 4) [26], the language in which DES is implemented and on which the new algorithms will be developed, as well as a brief introduction to Datalog [10], a derivation of Prolog that also uses DES.

Finally, both the semantic errors already implemented in DES (Chapter 5) and those that have been incorporated in the framework of this project (Chapter 6) will be described, to conclude with a section devoted to conclusions and possible lines of future work (Chapter 7). Appendix I contains a user guide for testing the DES tool, while Appendix II is a sample of the improvements that DES has undergone as a result of the development of this project.

Objectives

As mentioned above, the tool has functionality to detect syntactic errors, but its ability to identify semantic errors is limited. These errors, which often go unnoticed, make it difficult to understand the correct design and operation of databases.

In order to address this shortcoming, the main objective of this work is to extend the ability of DES to detect semantic errors by implementing new algorithms specifically designed for this

purpose.

The specific objectives of the project are as follows:

- To analyse the current limitations of DES in terms of semantic error detection.
- To investigate and categorise the most common semantic errors made by students in SQL queries.
- Design and implement algorithms to detect these errors and add them to the functionality of the tool.
- Develop a set of tests with different cases to validate the effectiveness of the implemented algorithms.

The aim of this work is not only to improve the performance and usability of DES, but also to contribute to the learning process of students by providing them with a more didactic and intelligent environment that facilitates the autonomous detection of semantic errors and reinforces their understanding of SQL language and database design.

Work plan

Firstly, given the need to acquire and deepen the knowledge necessary for the development of the project, during the month of September 2024, the study of the Prolog programming language began, through the reading of specialised works [26]. After carrying out various practical exercises to ensure correct understanding of the language, the code already implemented in the DES application was analysed.

At the same time, a process of theoretical documentation on semantic errors in SQL queries was started, paying special attention to their differentiation from syntactic errors. This phase included the review of scientific articles and teaching material [1, 2, 16-18, 23] that allowed us to lay the conceptual foundations necessary for the subsequent design and implementation of the semantic error detection algorithms.

The process of studying the code and theoretical foundations lasted until February 2025. During this period, several meetings were held with one of the main developers of the application, Fernando Sáenz Pérez, which facilitated a deeper understanding of the system.

Once all of the above processes were completed, the implementation of new code in DES began in March. The first step was to review and correct a semantic error previously identified, related to the unnecessary use of the clause `DISTINCT`. Once this case was solved, we continued with the incorporation of errors linked to the `GROUP BY` clause, which had not been dealt with in the DES tool until then. Most of these errors were implemented in mid-April, with the exception of error 21 (following the numbering of Brass and Goldberg's article [2]), whose complexity required extending its development until the beginning of May.

At the same time, algorithms began to be developed to detect semantic errors in the clauses `ORDER BY` and `UNION`, and the false positive previously identified as Error 8 was corrected.

In addition, test files were created throughout this process to validate the correct functioning of the algorithms. It should be noted that, at the beginning of April, the drafting of the project report also began, in parallel with the technical development, in order to adequately document the work carried out and facilitate its final presentation.

Capítulo 2

SQL y bases de datos relacionales

En este capítulo se repasarán conceptos fundamentales del lenguaje de programación SQL con el objetivo de refrescar los conocimientos al lector y facilitar la comprensión de los apartados posteriores del proyecto.

2.1. Introducción

El lenguaje de consulta estructurado, conocido como SQL (*Structured Query Language*) [12, 25], es un lenguaje de programación diseñado específicamente para gestionar y manipular datos almacenados en bases de datos relacionales. Este tipo de bases de datos organiza la información en forma de tabla, donde las filas representan registros de datos y las columnas definen atributos o campos.

SQL permite realizar una amplia variedad de operaciones sobre los datos: insertar, actualizar, eliminar, buscar o recuperar información. Además, proporciona herramientas para el mantenimiento de la base de datos y la optimización de su rendimiento. Por ello, SQL se ha convertido en una pieza clave tanto en el desarrollo de software como en la administración de datos a gran escala.

2.2. Contexto histórico

El origen de SQL se remonta a la década de 1970, en el contexto del proyecto System R desarrollado por IBM. Este proyecto tenía como objetivo demostrar la viabilidad práctica del modelo de bases de datos relacionales propuesto por Edgar F. Codd [4, 5]. Para interactuar con los datos almacenados en System R, IBM diseñó un nuevo lenguaje llamado SEQUEL (*Structured English Query Language*), que posteriormente fue renombrado como SQL debido a conflictos de marca registrada.

En 1979, la empresa Relational Software Inc., que más tarde se convertiría en Oracle, reconoció el enorme potencial comercial de este nuevo modelo relacional y del lenguaje SQL y lanzó Oracle V2, la primera implementación comercial de una base de datos que utilizaba SQL. Este lanzamiento marcó un antes y un después en la historia del software empresarial.

Desde entonces, SQL ha evolucionado continuamente, convirtiéndose en el estándar universal para la gestión y consulta de bases de datos relacionales. Ha sido estandarizado y certificado por organismos internacionales como ANSI (*American National Standards Institute*) e ISO (*International Organization for Standarization*) [15], lo que ha reforzado su posición como el lenguaje dominante en el ámbito de las bases de datos a nivel global.

En la actualidad, SQL no solo es compatible con bases de datos centralizadas, sino también con sistemas distribuidos, permitiendo el manejo de información a través de múltiples redes y

servidores. Su uso abarca desde grandes corporaciones hasta proyectos educativos, como es el caso de la herramienta DES, que se tratará más adelante en este documento.

Además, el desarrollo de versiones de código abierto como MySQL, PostgreSQL, SQLite o Firebird ha democratizado el acceso a esta tecnología, facilitando su adopción por estudiantes, desarrolladores independientes y pequeñas empresas.

Según el índice de popularidad de lenguajes de programación TIOBE [28], que clasifica los lenguajes de programación más utilizados a nivel mundial y se actualiza mensualmente, SQL continúa figurando entre los diez lenguajes más utilizados en el mundo, lo que demuestra su vigencia y relevancia en el ecosistema tecnológico actual.

2.3. Elementos del lenguaje SQL

El lenguaje SQL está compuesto por un conjunto de elementos que permiten interactuar de forma precisa y potente con bases de datos relacionales. Estos componentes constituyen la base de sentencias SQL, que permiten realizar una amplia variedad de operaciones como consultas, actualizaciones, definiciones de estructuras...

Los comandos SQL pueden ejecutarse a través de una interfaz de línea de comandos específica o desde entornos gráficos. Además, pueden ser enviados desde aplicaciones clientes a servidores de bases de datos, los cuales procesan la solicitud y devuelven resultados.

Los principales elementos que componen el lenguaje SQL son:

- **Cláusulas:** Son los componentes estructurales de una sentencia SQL. Cada cláusula cumple una función específica dentro de una sentencia y permite refinar la operación que se desea ejecutar sobre los datos. Las cláusulas pueden estar presentes en diversas operaciones como en consultas, actualizaciones o creaciones de tablas.
- **Expresiones:** Una expresión es una combinación de valores, operadores y funciones que produce un resultado. Estas expresiones se utilizan en múltiples contextos, como en listas de selección (`SELECT`) o en condiciones de filtrado (`WHERE`). Las expresiones pueden producir valores escalares (como números o cadenas) o tablas completas, que consisten en filas y columnas de datos.
- **Predicados:** Los predicados son condiciones lógicas que se utilizan dentro de cláusulas, principalmente en `WHERE` y `HAVING`, para filtrar registros según determinados criterios. Son fundamentales para seleccionar solo los datos que cumplen determinadas condiciones.
- **Consultas:** Una consulta es una instrucción SQL diseñada para recuperar datos de una o más tablas, en función de los criterios definidos por el usuario. La consulta más común es el comando `SELECT`, que permite especificar qué columnas mostrar en los resultados.
- **Instrucciones DDL (*Data Definition Language)* y DML (*Data Manipulation Language*):** Estas instrucciones van desde crear una tabla (`CREATE`), modificarla (`ALTER`), insertar algún dato (`INSERT`)... Es importante señalar que las consultas forman parte del lenguaje de manipulación de datos, aunque algunas veces se hace referencia a ellas como parte del lenguaje de consultas (DQL - *Data Query Language*).
- **Comandos:** Se trata de comandos específicos que permiten controlar la ejecución, gestionar archivos y depurar errores. Por ejemplo, en DES [23], comandos como `/process` para cargar sentencias SQL desde un archivo, o `/trace_sql` y `/debug_sql` para inspeccionar y depurar consultas.

El lenguaje SQL se puede dividir en varias partes, como consultas, control de acceso y control de transacciones. Este trabajo se centrará en la parte de consultas, específicamente en el uso de

SELECT, que es una de las instrucciones más importantes para consultar y recuperar datos de las bases de datos.

2.4. Consultas SQL

Las consultas SQL son las operaciones más comunes y esenciales del lenguaje SQL. A través de una consulta SQL, se puede recuperar información específica de una o varias tablas dentro de una base de datos. Estas operaciones se realizan principalmente mediante el comando SELECT, que es muy importante para extraer datos de las tablas.

Se puede ver que las principales cláusulas que componen una consulta SQL son:

- **SELECT:** Especifica las columnas que se desean extraer. Se puede utilizar el asterisco * para seleccionar todas las columnas de la relación.
- **FROM:** Indica las tablas o relaciones de las que se van a extraer los datos. En una consulta se debe incluir esta cláusula, ya que define la fuente de los datos.
- **WHERE:** Establece las condiciones para filtrar las filas de la tabla. Solo las filas que cumplen estas condiciones serán incluidas en los resultados. Términos como AND y OR permiten combinar múltiples condiciones.
- **GROUP BY:** Agrupa las filas que tienen valores idénticos en una o más columnas. Esta cláusula se utiliza en combinación con funciones de agregación como COUNT(), SUM(), AVG(), MIN(), MAX() para calcular valores sobre cada grupo.
- **HAVING:** Actúa como un filtro posterior al GROUP BY, permitiendo establecer condiciones sobre los grupos creados. Es similar a WHERE, pero se aplica en resultados agregados.
- **ORDER BY:** Ordena los resultados de la consulta de forma ascendente (ASC) o descendente (DESC), según una o más columnas. Si no se utiliza, el orden de los resultados no está garantizado.

A continuación, se examina el siguiente ejemplo:

```
SELECT WORKDEPT, COUNT(*) AS TOTALEMPLOYEES
FROM EMPLOYEE
WHERE SEX = 'F'
GROUP BY WORKDEPT
HAVING COUNT(*) > 5
ORDER BY TOTALEMPLOYEES DESC;
```

Mediante FROM se extraen los datos sobre la tabla EMPLOYEE, que contiene la información de los empleados. Se filtran los empleados que tienen sexo femenino mediante la cláusula WHERE SEX = 'F'. A continuación, los resultados se agrupan por departamento gracias a GROUP BY WORKDEPT, de modo que las empleadas de cada departamento se agrupan juntas y se extraen solo aquellos departamentos que tengan más de 5 empleadas (HAVING COUNT(*) > 5). Este filtro se aplica después de agrupar los datos. Por último, se ordenan los departamentos con más empleadas de mayor a menor mediante ORDER BY TOTALEMPLOYEES DESC. Por tanto, se puede asegurar que mediante la cláusula SELECT, el resultado de la consulta muestra el código de cada departamento (WORKDEPT) junto con el número total de empleadas (TOTALEMPLOYEES) de cada departamento, ordenado de mayor a menor número de empleadas.

2.4.1. Combinación de múltiples tablas

En muchas bases de datos relacionales, la información está distribuida entre varias tablas relacionadas entre sí. Para poder consultar datos que estén repartidos en diferentes tablas, se utilizan los `JOIN` [22].

Un `JOIN` permite combinar filas de dos o más tablas basándose en una condición de relación, normalmente una clave común. Se especifica mediante la cláusula `JOIN` junto con `ON` para definir cómo se relacionan las tablas.

Existen diferentes tipos de `JOIN`:

- **INNER JOIN:** Devuelve solo las filas que tienen coincidencias en ambas tablas.
- **LEFT JOIN:** Devuelve todas las filas de la tabla izquierda y las coincidencias de la tabla derecha. Si no hay coincidencia, se rellena con `NULL`.
- **RIGHT JOIN:** Al contrario que `LEFT JOIN`, devuelve todas las filas de la tabla derecha, y las coincidencias de la tabla izquierda. Si no hay coincidencia, se usa `NULL`.
- **FULL JOIN:** Devuelve todas las filas cuando hay coincidencia en alguna de las tablas. Las filas sin coincidencia muestran `NULL` en los campos de la otra tabla.
- **CROSS JOIN:** Devuelve el producto cartesiano entre dos tablas, es decir, todas las combinaciones posibles entre sus filas. No es muy habitual.

Se presentan dos ejemplos para aclarar los conceptos:

```
SELECT E.EMPNO, E.FIRSTNME, E.LASTNAME, D.
    DEPTNAME
FROM EMPLOYEE E
INNER JOIN DEPARTMENT D ON E.WORKDEPT = D.DEPTNO;
```

Esta consulta muestra el número, nombre y apellido de cada empleado, junto con el nombre del departamento en el que trabaja. Se utiliza `INNER JOIN` para que solo se muestren los empleados que tienen asignado un departamento, es decir, que el atributo `WORKDEPT` no sea `NULL` y que exista ese código en el atributo `DEPTNO` de la tabla departamento. Si algún empleado no tiene departamento, no aparecerá en los resultados.

```
SELECT E.EMPNO, E.FIRSTNME, E.LASTNAME, D.
    DEPTNAME
FROM EMPLOYEE E
LEFT JOIN DEPARTMENT D ON E.WORKDEPT = D.DEPTNO;
```

En este caso, la única diferencia con la consulta anterior es el uso de `LEFT JOIN` en vez de `INNER JOIN`. De esta manera, se muestran todos los empleados, incluso los que no tienen departamento asignado. Si un empleado no tiene coincidencia en la tabla `DEPARTMENT`, el valor de `DEPTNAME` aparecerá como `NULL`.

2.4.2. Claves primarias, candidatas y foráneas

Una clave primaria (*primary key*) en SQL es un atributo o conjunto de atributos que identifica de manera única cada fila de una tabla. No puede contener valores nulos ni duplicados y debe ser única. Existen también las claves candidatas (*candidate key*), que son todos aquellos conjuntos

de atributos que podrían cumplir el rol de clave primaria. Se elige una clave de ellas como principal y el resto se consideran alternativas. Cabe destacar que la cláusula `CANDIDATE KEY` no forma parte del estándar de SQL, sino que es una propuesta del sistema DES. En SQL estándar, las claves candidatas se representan mediante las restricciones `UNIQUE` y `NOT NULL`.

Por otro lado, también se tienen las claves foráneas (*foreign key*). Se trata de un atributo de una tabla que hace referencia a la clave primaria de otra tabla, estableciendo una relación entre ambas. Cobran especial relevancia en el uso de reuniones `JOIN`.

Estas claves se definen al crear la tabla mediante las cláusulas `PRIMARY KEY`, `CANDIDATE KEY` (en el caso de DES) y `FOREIGN KEY`.

2.5. Dependencias funcionales

Las dependencias funcionales [13] son un concepto clave en el diseño de bases de datos, y resultan especialmente relevantes para comprender ciertos errores descritos en el Capítulo 6. Por este motivo, se incluye a continuación una explicación básica que facilite su entendimiento, incluso para quienes no estén familiarizados con el tema.

Una dependencia funcional es una relación entre columnas de una tabla que indica que el valor de una de ellas permite determinar unívocamente el valor de otra.

En términos simples, se dice que existe una dependencia funcional cuando conocer el valor de una columna (o conjunto de columnas) permite conocer de forma unívoca el valor de otra columna. Por este motivo, se afirma que una columna determina a otra.

Las dependencias funcionales se expresan mediante una flecha

$$\texttt{X} \rightarrow \texttt{Y}$$

Esto se lee como `X` determina a `Y` o `Y` depende funcionalmente de `X`, siendo `X` e `Y` atributos de una relación o tabla.

DNI	Nombre	Apellido	Departamento
12345678A	Marta	López	Ventas
87654321B	Pedro	García	Finanzas

Tabla 2.1: Ejemplo de tabla de empleados

En la Tabla 2.1, el campo `DNI` es único para cada empleado. Por tanto, conociendo el valor del `DNI`, es posible conocer también de forma precisa el nombre, el apellido y el departamento del empleado. Esto se expresa de la siguiente manera:

$$\texttt{DNI} \rightarrow \texttt{Nombre}, \texttt{Apellido}, \texttt{Departamento}$$

Es decir, el `DNI` determina funcionalmente a los demás atributos.

Por el contrario, no se puede decir que el `Apellido` determine al `Departamento`, ya que podrían existir varios empleados con el mismo apellido trabajando en distintos departamentos. En este caso, no se cumple la condición de unicidad, y por tanto no hay una dependencia funcional.

Una de las propiedades fundamentales de las dependencias funcionales es la transitividad. Esta propiedad establece que si un atributo `A` determina a un atributo `B`, y `B` determina a un atributo `C`, entonces `A` también determina a `C`.

Como ejemplo, se considera la Tabla 2.2. En este caso, se puede asegurar que el `DNI` determina el `DepartamentoID`, ya que cada empleado pertenece a un único departamento, y a su vez, el

DNI	DepartamentoID	Departamento
12345678A	D001	Ventas

Tabla 2.2: Ejemplo de transitividad

`DepartamentoID` determina el nombre del `Departamento`. Por tanto, aplicando la transitividad, también se cumple que:

$$\texttt{DNI} \rightarrow \texttt{Departamento}$$

Una vez explicado el concepto de transitividad, resulta fundamental introducir el cierre transitivo, que está directamente relacionado con las dependencias funcionales y su transitividad. Este concepto se refiere al conjunto de atributos que pueden estar determinados, de forma directa o indirecta, a partir de un conjunto dado de atributos mediante la aplicación sucesiva de dependencias funcionales.

En otras palabras, dado un conjunto de atributos, su cierre transitivo incluye todos los atributos que pueden deducirse a partir de ese conjunto inicial utilizando las dependencias funcionales conocidas.

Este concepto resulta útil porque permite conocer todo lo que se puede determinar a partir de un conjunto de atributos, lo cual es fundamental a la hora de detectar redundancias en las cláusulas `GROUP BY` y `ORDER BY`. Como se verá en detalle en el Capítulo 6, si un atributo ya se encuentra dentro del cierre de otro que ya está presente en la consulta, puede que su inclusión sea innecesaria.

Para finalizar con esta introducción acerca de las dependencias funcionales, solo queda explicar las diferentes formas de deducir o detectar dependencias funcionales a partir de las estructuras y condiciones que aparecen en una consulta SQL:

- **Dependencias provenientes de tablas:** Algunas tablas contienen anotaciones o metadatos que indican explícitamente relaciones funcionales entre atributos. Por ejemplo, si en una descripción de tabla se indica que `'A determined by B'`, esto se traduce en la dependencia funcional

  $$\texttt{B} \rightarrow \texttt{A}$$

- **Dependencias derivadas de condiciones en la cláusula `WHERE`:** En ciertas consultas SQL, es posible encontrar condiciones de igualdad entre atributos, como `A = B`. Cuando esto ocurre, se puede interpretar que existe una dependencia funcional en ambos sentidos:

  $$\texttt{A} \rightarrow \texttt{B}, \texttt{B} \rightarrow \texttt{A}$$

  Estas relaciones de igualdad implican que ambos atributos tendrán el mismo valor dentro del conjunto de resultados, por lo que el conocimiento de uno permite conocer el otro sin ambigüedad.

- **Dependencias derivadas de claves primarias o candidatas:** La mayoría de las relaciones contienen algún atributo que actúa como clave primaria o candidata. Por definición, una clave primaria (o candidata) determina funcionalmente a todos los demás atributos de su tabla. Es decir, si se conoce el valor de la clave primaria, se puede determinar de forma única el valor de cualquier otro atributo de esa fila. Por tanto, siendo `A` la clave primaria o candidata de una tabla y `B` cada uno del resto de atributos de dicha tabla, se cumple la siguiente dependencia funcional para cualquier atributo `B`:

  $$\texttt{A} \rightarrow \texttt{B}$$

Estos dos tipos de dependencias son las que se tendrán en cuenta dentro de los algoritmos descritos en el Capítulo 6 de este trabajo, con el objetivo de detectar redundancias y simplificar consultas SQL.

Capítulo 3

El sistema DES

En este capítulo se introduce la herramienta DES, describiendo sus principales funcionalidades y características, así como el manejo de errores en consultas SQL.

Según se expone en ciertos artículos de investigación [23], DES (*Datalog Educational System*) es una herramienta diseñada para facilitar el aprendizaje del lenguaje SQL en el contexto de bases de datos. Como se ha mencionado con anterioridad, fue desarrollado en la Universidad Complutense de Madrid con la intención de proporcionar un entorno de trabajo que no solo ejecute consultas SQL, sino que también ofrezca retroalimentación avanzada sobre los errores cometidos por los usuarios.

Es un sistema interactivo, multiplataforma, de código abierto y gratuito, implementado sobre el lenguaje de programación Prolog, y que permite trabajar con lenguajes de consulta Datalog y SQL simultáneamente. Esta dualidad permite que estudiantes e investigadores exploren el funcionamiento de las bases de datos tanto deductivas como relacionales desde un entorno unificado. DES proporciona además una interfaz gráfica mediante ACIDE (un entorno de desarrollo integrado en Java) [20] y también tiene soporte para Emacs.

Una de las características distintivas de DES es que no requiere instalación, puede ejecutarse desde cualquier carpeta, y está disponible para los principales sistemas operativos (Windows, Linux, Mac Os X). A su vez, presenta una interfaz web, llamada DESweb [17]. Además, es compatible con varios intérpretes de Prolog modernos como Ciao [3], GNU Prolog [8], SWI-Prolog [27] y SICStus Prolog [24]. En este proyecto se ha optado por SWI-Prolog por ser la única opción gratuita soportada actualmente por el sistema, lo que lo hace especialmente adecuado para un entorno educativo y de desarrollo accesible.

Desde el punto de vista técnico, DES implementa una infraestructura interna basada en *tabling*, una técnica que memoriza resultados de consultas previas para evitar cómputos redundantes. Este mecanismo es fundamental en su motor deductivo, ya que permite gestionar consultas recursivas y evitar computación innecesaria. La memoria intermedia utilizada para almacenar estos resultados se denomina tabla de extensión.

DES soporta características avanzadas como negación estratificada, agregados, reuniones externas (*outer joins*), duplicados... A su vez, cuenta con un potente sistema de restricciones, que permite definir claves primarias, foráneas, tipos y condiciones complejas.

Además, cuenta con un amplio repertorio de comandos, muy parecido al de PostgreSQL, que permiten al usuario, entre otras funcionalidades, facilitar la carga y gestión de programas en Datalog y SQL. Por ejemplo, se pueden cargar archivos con reglas Datalog usando el comando `/consult`, procesar sentencias SQL con `/proccess`, o insertar dinámicamente una regla o hecho Datalog mediante `/assert`.

También ofrece opciones para controlar el comportamiento, como activar o desactivar el ma-

nejo de duplicados mediante `/duplicates on` y `duplicates off`, o simplificar automáticamente programas con `/simplify on`.

Otra de las funciones más destacadas es el trazado y la depuración. Comandos como `/trace_datalog` o `/debug_sql` permiten analizar el flujo de ejecución y detectar errores lógicos. Además, se pueden ejecutar consultas directamente en ambos lenguajes, SQL y Datalog, de forma integrada en el mismo entorno. Cabe destacar que los comandos en DES son muy similares a los utilizados en PostgreSQL.

Tras presentar la herramienta, se pondrá el foco en su principal aportación diferenciadora, objetivo de este proyecto: un análisis exhaustivo de errores en las consultas SQL que va mucho más allá de lo que ofrecen otros gestores de bases de datos.

A diferencia de los gestores de bases de datos tradicionales (como MySQL, Oracle o PostgreSQL), que suelen dar mensajes de error muy básicos y sin mucho detalle, DES analiza en profundidad tanto los errores sintácticos como los semánticos. De esta forma, ayuda a los estudiantes y programadores a comprender mejor sus equivocaciones y aprender de ellas.

Además, no se limita a detectar errores básicos, sino que evalúa propiedades lógicas de las consultas, verifica restricciones y alerta sobre inconsistencias que incluso pasan desapercibidas en muchos sistemas comerciales. La herramienta utiliza técnicas de programación lógica con restricciones (CLP - *Constraint Logic Programming* [11]) para realizar el análisis semántico, lo que le permite analizar condiciones siempre verdaderas, redundantes, inconsistentes, etc.

A continuación se presentan dos ejemplos ilustrativos que muestran cómo DES gestiona errores tanto sintácticos como semánticos. Es importante destacar que, en el caso de los errores semánticos, el sistema no los trata como fallos que impidan la ejecución, sino que los señala mediante advertencias. Estas consultas son sintácticamente válidas, pero DES informa al usuario de que podrían no producir los resultados esperados.

En el primer ejemplo, se muestra una consulta con un error sintáctico: la cláusula `FROM` está mal escrita.

```
SELECT LASTNAME
FRO EMPLOYEE;
```

Ante esta instrucción, DES detecta el problema de forma inmediata y muestra el siguiente mensaje de error:

```
Error: (SQL) Expected FROM clause after 'SELECT LASTNAME FRO '.
```

En el segundo ejemplo, la consulta está correctamente escrita desde el punto de vista sintáctico, pero contiene una condición lógicamente inconsistente en la cláusula `WHERE`:

```
SELECT LASTNAME
FROM EMPLOYEE
WHERE WORKDEPT = 'D01' AND WORKDEPT = 'D02';
```

Aquí, DES identifica que la condición no puede cumplirse nunca y emite la siguiente advertencia semántica:

```
Warning: [Sem] Inconsistent condition.
```

Este tipo de análisis preventivo convierte a DES en una herramienta especialmente útil en contextos educativos, ya que no solo permite ejecutar consultas, sino también comprender cuándo una instrucción puede ser sintácticamente correcta pero conceptualmente errónea.

Capítulo 4

Prolog y Datalog

En este capítulo se ofrece una breve introducción a los lenguajes de programación Prolog y Datalog, abordando sus orígenes, terminologías y las principales diferencias entre ambos.

4.1. Prolog

Dado que la herramienta DES está implementada en el lenguaje de programación Prolog [19, 26], resulta fundamental ofrecer una breve introducción a dicho lenguaje para facilitar la comprensión de los algoritmos desarrollados en este trabajo, así como del funcionamiento interno de la propia plataforma.

4.1.1. Contexto histórico

Prolog (*Programming in Logic*) fue ideado a principios de la década de 1970 por Alain Colmerauer y Philippe Roussel [6], basado en la interpretación procedimental de las cláusulas de Horn [9] desarrollada por Robert Kowalski. Su creación surgió con el propósito de integrar la lógica como un lenguaje declarativo de representación del conocimiento.

Durante las primeras décadas, Prolog se convirtió en el lenguaje preferido por los investigadores europeos de inteligencia artificial, mientras que en Estados Unidos predominaba el uso de Lisp [14], lo cual generó un prolongado debate entre ambas comunidades científicas sobre la idoneidad de uno u otro lenguaje para aplicaciones en IA, una discusión que, a día de hoy persiste.

Actualmente, Prolog continúa siendo un lenguaje vigente y relevante, figurando de forma recurrente en rankings como el índice de TIOBE [28], donde suele situarse entre los 20 lenguajes de programación más utilizados.

4.1.2. Introducción al lenguaje

Prolog es un lenguaje de programación declarativo especialmente adecuado para modelar problemas en los que se deben representar relaciones entre objetos o entidades. A diferencia de los lenguajes imperativos (como Python o Java), donde se detallan las instrucciones paso a paso, en Prolog el programador define hechos, reglas y luego formula consultas que el sistema intenta resolver aplicando un mecanismo de inferencia lógica.

Su fundamento teórico se basa en la lógica de primer orden (también llamada lógica de predicados), lo cual le permite centrarse en la representación del conocimiento en lugar de en los procesos algorítmicos.

El funcionamiento de Prolog se basa en tres pilares fundamentales:

- **Unificación:** Es el proceso de comparar estructuras lógicas (como términos o predicados) para determinar si pueden coincidir, asignando valores a variables si es necesario. Este mecanismo es esencial para hacer coincidir consultas con hechos o reglas.
- **Representación estructurada:** Los términos en Prolog incluyen a los hechos y cláusulas, pero también a los datos, ya sean estructurados o no. Se representan internamente como estructuras jerárquicas en forma de árbol, facilitando la organización de relaciones entre entidades.
- **Backtracking:** Cuando Prolog busca una solución y encuentra un camino que no lleva al resultado deseado, automáticamente retrocede para explorar otras alternativas hasta que encuentra una solución o determina que no existe.

4.1.3. Sintaxis

Términos

En Prolog, los términos constituyen la unidad básica del lenguaje. Un término se construye a partir de su nombre, seguido de cero o más argumentos, los cuales se colocan entre paréntesis y separados por comas. Este nombre se conoce como funtor e identifica un término compuesto o predicado. Actúa como el etiquetado de la relación o estructura, seguido por un número fijo de argumentos (su aridad). Por ejemplo, en `padre(juan, maria)`, `padre` es el funtor y tiene una aridad de 2.

Los términos pueden clasificarse en tres tipos principales:

- **Constantes:** Pueden ser números (enteros o reales) o átomos (también llamados funtores cuando actúan como identificadores de relaciones o propiedades). Siempre comienzan con una letra minúscula. Por ejemplo, `luis`.
- **Variables:** Se utilizan para representar valores desconocidos o genéricos. En Prolog, una variable comienza con una letra mayúscula o un guión bajo, y puede incluir letras, números y guiones bajos. Existe también una variable anónima, escrita simplemente como _, que se utiliza cuando no se necesita referirse al valor que tome esa variable.
- **Términos compuestos:** Son construcciones formadas por un funtor seguido de una lista de argumentos entre paréntesis, que pueden ser constantes, variables u otros términos compuestos. Cuando estas estructuras representan relaciones lógicas definidas mediante hechos o reglas, se denominan predicados. Por ejemplo, `padre('Raúl', 'Marina')` es un término compuesto que también es un predicado con aridad 2 (es decir, con dos argumentos), lo que indica que Raúl es el padre de Marina.

Programas

Un programa en Prolog se basa en la declaración de conocimiento mediante hechos y reglas, con el objetivo de responder consultas lógicas sobre dicho conocimiento.

En general, su estructura se compone de los siguientes elementos:

- **Dominio:** Define el conjunto de valores posibles (constantes, estructuras) que pueden aparecer como argumentos de los predicados.
- **Predicados:** Representan relaciones o propiedades entre los elementos del dominio. Se identifican por su nombre y su aridad (número de argumentos). Pueden ser parte de una cláusula, ya sea un hecho o una regla. Por ejemplo, `padre(juan, maria)`.

- **Cláusulas:** Son las unidades fundamentales de conocimiento y se dividen en:
  - **Hechos:** Sentencias que declaran relaciones o propiedades que se consideran verdaderas. Hay dos tipos: monádicos(un argumento), como por ejemplo, `mamifero(perro)`, o poliádicos (más de un argumento), como por ejemplo `hermano(juan, pedro)`.
  - **Reglas:** Establecen relaciones condicionales. Indican que un predicado es verdadero si se cumplen ciertas condiciones. Tras conocer esta definición de reglas, se puede decir que los hechos son reglas con cuerpo siempre cierto (true). Su sintaxis general es `cabeza :- cuerpo`. Un ejemplo puede ser:

    ```
    abuelo(X, Y) :- progenitor(X, Z), progenitor(Z, Y).
    ```

    También se pueden definir reglas recursivas:

    ```
    predecesor(X, Y) :- progenitor(X, Y).

    predecesor(X, Y) :- progenitor(X, Z), predecesor(Z, Y).
    ```

- **Consultas:** Permiten interrogar la base de conocimiento usando los predicados definidos. A través de ellas, se pueden verificar hechos o realizar averiguaciones específicas. Las consultas comenzarán con `?-`, que no será necesario escribir ya que forma parte del prompt de Prolog, y finalizarán con un punto (`'.'`).

  Un ejemplo sería: `progenitor(X, jaime).` En esta consulta se devuelven los valores de `X` que cumplen la condición especificada.

4.1.4. SWI-Prolog

En este proyecto se utilizará SWI-Prolog, una de las implementaciones más populares y completas del lenguaje de programación Prolog. Desarrollada por Jan Wielemaker [30] desde 1987, SWI-Prolog es una herramienta de código abierto que ha sido mantenida y mejorada de forma continua durante décadas.

Se caracteriza por ofrecer un entorno de desarrollo integrado, una amplia colección de bibliotecas, soporte para programación lógica avanzada, interfaces gráficas, conectividad con base de datos y capacidades para desarrollo web. Además, dispone de una extensa documentación [27] y una comunidad activa que facilita su aprendizaje y uso.

4.2. Datalog

Datalog [7] es un lenguaje de programación declarativa basado en lógica de primer orden, diseñado originalmente para trabajar con bases de datos deductivas. Es un lenguaje derivado de Prolog, con restricciones que lo hacen adecuado para consultas con conjuntos de datos finitos, como los almacenados en bases de datos relacionales.

En este trabajo se emplea como lenguaje lógico intermediario entre el lenguaje SQL y el sistema de razonamiento basado en restricciones utilizado por la herramienta DES [21]. Para comprender su papel, es necesario introducir sus fundamentos en primer lugar.

4.2.1. Contexto histórico

Datalog fue desarrollado en la década de 1980 por investigadores del ámbito de bases de datos, como Jeffrey Ullman y David Maier, con el fin de cubrir la necesidad de crear un lenguaje lógico con semántica bien definida y aplicable al contexto relacional. Datalog se orientó desde el principio al modelado de reglas y relaciones en bases de datos.

4.2.2. Terminología

Datalog es un lenguaje declarativo que presenta una terminología muy similar a la descrita anteriormente en Prolog, cuyos elementos principales son hechos, reglas y consultas. El sistema deduce todos los hechos posibles a partir de otros hechos y reglas conocidas. Además, cabe destacar que la adición de nuevos hechos nunca elimina derivaciones anteriores y las reglas deben ser finitas en su evaluación, es decir, sin variables libres no ligadas. A su vez, una de las principales fortalezas de Datalog, al igual que Prolog, es la gran potencia que posee en la expresión de consultas recursivas. Otra de sus principales características es la ausencia de funciones y términos compuestos.

Pero Datalog no es solo una variante sintáctica de Prolog, sino que presenta diferencias en cuanto a su semántica, modelo de ejecución y modo de razonamiento. En Prolog, la evaluación de las consultas sigue una estrategia *top-down* (de arriba hacia abajo), en la que el sistema intenta resolver la consulta buscando reglas cuya cabeza coincida con ella y aplicándolas en el orden en el que fueron escritas. Este proceso es secuencial y se apoya en el uso de *backtracking* para explorar distintas alternativas si la primera no conduce a solución. Como resultado, Prolog devuelve una solución cada vez, y requiere intervención del usuario, habitualmente introduciendo punto y coma (';'), para obtener más respuestas. Este enfoque es sensible al orden de las reglas y puede entrar en bucles infinitos en algunos casos.

Por otro lado, Datalog emplea una estrategia *bottom-up* (de abajo hacia arriba) [29], partiendo de los hechos base y aplicando las reglas del programa para derivar nuevas conclusiones, generando todas las consecuencias lógicas posibles hasta alcanzar un punto fijo, es decir, un estado en el que no se puede derivar nada más. A diferencia de Prolog, Datalog devuelve el conjunto completo de soluciones de forma automática, sin necesidad de interacción del usuario ni dependencia del orden de las reglas.

4.2.3. Semánticas en Datalog

Según el artículo [10], la semántica de Datalog puede analizarse desde distintos puntos de vista:

1. **Semántica de punto fijo (*Fixpoint semantics*):** Es la más utilizada en implementación y se basa en la idea de construir progresivamente el significado del programa aplicando sus reglas hasta que no se puedan derivar más hechos. El proceso comienza con los hechos conocidos y a partir de ellos se van aplicando las reglas del programa para generar nuevos hechos. Esto se repite hasta alcanzar un estado en el que no se derivan hechos adicionales, lo que se conoce como punto fijo. El conjunto de hechos obtenido en ese punto representa el modelo mínimo del programa, que es el menor conjunto de hechos que satisface todas las reglas definidas.

2. **Semántica de teoría de modelos (*Model-theoretic semantics*):** Interpreta cada regla Datalog como una cláusula lógica de Horn (fórmula de la lógica de primer orden con al menos un antecedente y un consecuente). Un modelo de un programa Datalog es un conjunto de hechos que satisface todas las reglas lógicamente. El objetivo es encontrar el conjunto más pequeño de hechos que hace verdadero el programa (modelo mínimo). Este modelo siempre existe, es único y representa la salida del programa.

3. **Semántica basada en pruebas (*Proof-theoretic semantics*):** Considera a Datalog como un sistema deductivo. Una consulta es verdadera si se puede derivar desde los hechos mediante una secuencia válida de aplicación repetida de reglas y sustitución de variables. Este enfoque convierte a Datalog en un proceso de cálculo lógico autónomo, en el que todo el conocimiento se deriva exclusivamente a partir de hechos y reglas explícitas, sin necesidad de axiomas externos.

4.2.4. Datalog en DES

Tal y como muestra el artículo [21], Datalog desempeña un papel clave en el sistema DES como lenguaje intermediario para el análisis semántico entre SQL y un sistema de razonamiento lógico basado en restricciones (CLP, *Contraint Logic Programming*). Con esta arquitectura, DES es capaz de detectar errores semánticos en consultas SQL sin necesidad de ejecutar dichas consultas sobre bases de datos reales, trabajando únicamente a partir de la lógica de la consulta y los metadatos del esquema. DES adopta una estrategia *bottom-up*, pero con una particularidad importante: se guía mediante una evaluación *top-down*, lo que restringe el espacio de búsqueda con el objetivo de evitar generar hechos innecesarios.

Datalog está presente en las siguientes fases del funcionamiento de DES:

1. **Traducción de SQL a Datalog**: El primer paso consiste en transformar cada consulta SQL en un conjunto de reglas de Datalog que mantienen su significado semántico. Este proceso incluye soporte para subconsultas con términos como `IN` o `EXISTS`, cláusulas `JOIN`, `GROUP BY` o modificadores como `DISTINCT`.

   La consulta SQL:

   ```
   SELECT EMP.NAME, DEPT.NAME
   FROM EMP JOIN DEPT
   ON EMP.DEPT = DEPT.ID;
   ```

   se traduce a Datalog como:

   ```
   respuesta(NAMEMP, NAMEDEPT) :-
                                emp(NAMEMP, DEPTIDE),
                                dept(DEPTIDD, NAMEDEPT),
                                DEPTIDE = DEPTIDD.
   ```

   Se puede observar que la consulta retorna pares de nombres de empleados y departamentos, siempre que el identificador del departamento coincida con el del identificador del departamento del empleado.

   En primer lugar, se obtiene la cabeza del predicado (`respuesta(NAMEMP, NAMEDEPT)`), que es equivalente al `SELECT` en SQL. Representa el resultado, que son los pares de nombres de empleados y departamentos. En el cuerpo de la regla de Datalog, se encuentran `emp(NAMEMP, DEPTIDE)` y `dept(DEPTIDD, NAMEDEPT)`, que representan las relaciones departamento y empleado, con sus respectivos atributos. Por último, `DEPTIDE = DEPTIDD` representa la condición de la cláusula `WHERE`, expresada como una restricción de igualdad entre las variables que representan los ID de los departamentos en ambas tablas.

2. **Interpretación de Datalog en CLP:** Una vez generado el programa Datalog, se convierte en un programa CLP, donde las condiciones como expresiones lógicas o aritméticas se transforman en restricciones sobre variables.

   ```
   SELECT SALARY
   FROM EMPLOYEE
   WHERE SALARY BETWEEN 5000 AND 2000;
   ```

   En esta consulta SQL, la condición de la cláusula `WHERE` se convierte en un conjunto de restricciones CLP

   $$\{\texttt{SALARY} \geq 5000,\ \texttt{SALARY} \leq 2000\}$$

   que, al ser evaluadas, revelan que el intervalo es inválido, ya que el mínimo es mayor que el máximo, así que la condición será inconsistente.

Este proceso también se aplica a las restricciones `CHECK` en definiciones de tablas o expresiones complejas dentro de subconsultas, sin necesidad de instanciar los datos reales de la base de datos.

En esta etapa, se aplican resolutores (*solvers*) especializados para distintos dominios como enteros, racionales, cadenas... que permiten razonar sobre las condiciones impuestas en la consulta. Además, se utiliza la cooperación de dominios para mejorar los resultados de la resolución, coordinando diferentes resolutores que manejan distintos niveles de propagación de restricciones.

Un resolutor es un componente del sistema capaz de analizar restricciones lógicas o matemáticas. Por ejemplo, puede determinar si una condición es inconsistente, o deducir que `X = 45` y `Y = 35` resolviendo el sistema de ecuaciones `X - Y = 10` y `X + Y = 80`. Gracias a estos resolutores y a la combinación de los mismos, DES mejora la capacidad de detectar errores lógicos complejos.

Capítulo 5

Estado del arte: Errores implementados en DES

Este capítulo se enfoca en la presentación de los tipos de errores en consultas SQL: sintácticos y semánticos, con un análisis detallado de los casos semánticos contemplados por la herramienta DES.

5.1. Errores comunes en consultas SQL

Al escribir consultas SQL, es habitual cometer errores que impiden su correcta ejecución o que generan resultados incorrectos. Estos errores pueden clasificarse en dos grandes grupos: errores sintácticos y errores semánticos [17]. Conocer sus diferencias es fundamental para detectar y corregir problemas de manera eficiente.

5.1.1. Errores sintácticos

Los errores sintácticos ocurren cuando la consulta no cumple las reglas gramaticales de SQL. Es decir, hay una estructura incorrecta en la sentencia, que impide que el motor de la base de datos pueda interpretarla. Estos errores son detectados inmediatamente por el sistema, el cual genera un mensaje de error claro y directo indicando cuál ha sido la causa del mismo. DES realiza esta última tarea, a diferencia de otros sistemas que simplemente indican que ha habido un error, sin concretarlo.

Entre las causas más frecuentes se encuentran la omisión de palabras clave obligatorias, como olvidar incluir la cláusula `FROM` en una consulta, el uso incorrecto de comas, paréntesis o signos de puntuación, así como errores tipográficos en los nombres de columnas o tablas. También es habitual alterar el orden correcto de las cláusulas (por ejemplo, colocar `FROM` antes de `SELECT`) o no encerrar los valores literales de tipo texto entre comillas simples.

```
SELECT DEPTNAME
FROM DEPARTMENT
WHERE (DEPTNAME = 'Contabilidad';
```

Este código produce el siguiente error en DES:

`Error: (DL) Right bracket ')' not found after 'SELECT DEPTNAME FROM DEPARTMENT WHERE ('.`

Se debe a que se ha olvidado cerrar un paréntesis en la condición de la cláusula `WHERE`.

5.1.2. Errores semánticos

Los errores semánticos, en cambio, no impiden que la consulta se ejecute, pero hacen que el resultado sea incorrecto o no tenga sentido lógico. Ocurren cuando la consulta está bien estructurada en términos de sintaxis, pero el significado de lo que se pide no se corresponde con la lógica del modelo de datos o con la intención del usuario.

Este tipo de errores suele ser más difícil de detectar, ya que no siempre generan mensajes de error. En su lugar, devuelven resultados vacíos, inconsistentes o inesperados.

DES analiza la semántica de la consulta en base a propiedades como claves primarias, claves foráneas, restricciones de integridad, condiciones siempre falsas o redundantes, ausencia de condiciones de unión, entre otros aspectos.

Entre las causas más comunes se encuentran el uso inapropiado de condiciones de la cláusula `WHERE`, como filtrar por una columna que no tiene relación directa con lo que se quiere obtener; errores de reuniones (`JOIN`), que pueden generar un número de filas incorrecto debido a reuniones mal definidas; agrupaciones incorrectas mediante `GROUP BY`...

Un ejemplo de error semántico se corresponde con el segundo ejemplo que se ha dado en el Capítulo 3. En la próxima sección se verán un gran número de ejemplos de este tipo de error.

5.2. Errores semánticos implementados en DES

A continuación, se describen los errores semánticos que han sido implementados en el sistema DES. El análisis semántico de consultas SQL con cláusula `SELECT` se lleva a cabo principalmente mediante el predicado `check_sql_select_semantic_error/3`, el cual recibe como parámetros el árbol sintáctico de la consulta SQL y las reglas de Datalog resultado de la transpilación. Esta estructura de entrada es necesaria ya que los distintos predicados utilizados en el cuerpo pueden requerir uno, otro o incluso ambos parámetros.

El esquema general de este predicado principal se muestra a continuación. Su cuerpo está compuesto por llamadas a otros predicados específicos, cada uno encargado de verificar un tipo concreto de error semántico:

```
check_sql_select_semantic_error(SQLst, RNVss, ARs) :-
  ...,
  check_sql_inconsistent_condition(Rs, ARs),        % Error 1
  check_sql_distinct(Hs, Bss, (Hs, Bss)),           % Error 2
  check_sql_constant_column(SQLst, Rs),             % Error 3
  check_sql_duplicated_column_values(SQLst, Rs),    % Error 4
  check_sql_unused_tuple(Bss, NVss),                % Error 5
  check_sql_unnecessary_join(Bss, NVss),            % Error 6
  ...
```

Antes de comenzar con el análisis, se presentan las relaciones utilizadas en los ejemplos con el fin de facilitar la comprensión de cada uno de los errores que se van a explicar.

```
CREATE TABLE DEPARTMENT
  (DEPTNO   CHAR(3)          NOT NULL,
   DEPTNAME VARCHAR(36)      NOT NULL,
   LOCATION CHAR(16),
   PRIMARY KEY (DEPTNO))

CREATE TABLE EMPLOYEE
  (EMPNO      CHAR(6)        NOT NULL,
   FIRSTNME   VARCHAR(12)    NOT NULL,
   LASTNAME   VARCHAR(15)    NOT NULL,
   WORKDEPT   CHAR(3),
   JOB        CHAR(8),
   SEX        CHAR(1),
   SALARY     DECIMAL(9,2),
   BONUS      DECIMAL(9,2),
   PRIMARY KEY (EMPNO))
```

Los siguientes subapartados presentan una catalogación de errores semánticos en consultas SQL basada en el artículo de Brass y Goldberg [2], que constituye una referencia fundamental en este ámbito.

Esta clasificación no solo se organiza en función de las cláusulas del lenguaje SQL (`SELECT`, `FROM`, `GROUP BY`, etc.), sino que también agrupa los errores según categorías conceptuales como formulaciones ineficientes, consultas innecesarias o redundantes, violación de patrones estándar...

Cabe recordar que en este proyecto se seguirá la misma categorización y numeración de errores que en el artículo, ya que coincide con la empleada por los desarrolladores de DES, facilitando así su integración.

5.2.1. Consultas innecesarias

Error 1: Condición inconsistente

Este error hace alusión a consultas `SQL` que siempre devuelven un resultado vacío, sin importar el contenido de la base de datos. Esto se debe a condiciones contradictorias en la cláusula `WHERE`.

```
SELECT SALARY
FROM EMPLOYEE
WHERE WORKDEPT = 'DEPT1' AND WORDEPT = 'DEPT2';
```

En este ejemplo, se filtran los empleados que tienen como código de departamento `DEPT1` y `DEPT2`, algo que es imposible que se cumpla simultáneamente. Por ello, esta consulta siempre retornará un valor vacío y en DES se imprimirá el siguiente mensaje de advertencia:

```
Warning: [Sem] Inconsistent condition.
```

5.2.2. Cláusula `SELECT`

Error 2: `DISTINCT` innecesario

En algunas ocasiones se puede probar que una consulta no puede devolver valores duplicados. En esos casos, `DISTINCT` no debe ser usado, ya que resta rapidez a la ejecución de la consulta.

```
SELECT DISTINCT DEPTNO
FROM DEPARTMENT;
```

Por ejemplo, en esta consulta, `DISTINCT` es innecesario ya que el atributo `DEPTNO` es una clave primaria de la relación `DEPARTMENT`, por lo que los valores que tome el atributo `DEPTNO` serán únicos, sin necesidad de aplicar el `DISTINCT`. La herramienta DES, ante esta consulta, mostrará el siguiente mensaje de aviso:

```
Warning: [Sem] Unnecessary DISTINCT because of primary key in [DEPARTMENT].
```

Error 3: Resultado con columna constante

Una columna de salida es innecesaria si contiene un único valor que es constante y que viene derivado de la consulta sin ninguna relación con el estado de la base de datos.

Si una constante es escrita como término de la lista de atributos de `SELECT`, es obviamente a propósito y no se debe lanzar ninguna advertencia. A su vez, si `SELECT` viene acompañado del símbolo * y produce un resultado con columna constante, aún así se pasará por alto ya que es mucho más corto y eficaz escribir esta sentencia, que escribir las columnas restantes (no constantes) explícitamente, así que de nuevo, no aparecerá ninguna advertencia en este caso.

Sin embargo, a continuación se tiene el siguiente ejemplo:

```
SELECT SALARY
FROM EMPLOYEE
WHERE SALARY = 1000;
```

En la forma normal conjuntiva de la cláusula `WHERE` aparece una condición que iguala el atributo `SALARY` a una constante. A su vez, aparece `SALARY` también como término de la lista `SELECT`, entonces este caso sí que debe ser capturado. Además, si apareciese otro atributo igualado a `SALARY`, el valor de ese atributo será constante también, y su uso en la lista `SELECT` también generará la siguiente advertencia :

```
Warning: [Sem] Constant output column 'SALARY' with value '1000'.
```

Error 4: Columnas de salida duplicadas

Una columna de salida es innecesaria cuando es siempre idéntica a otra de las columnas que aparece en la lista `SELECT`.

```
SELECT SALARY, BONUS
FROM EMPLOYEE
WHERE SALARY = BONUS;
```

En este caso, ambas columnas de salida `SALARY` y `BONUS` tienen los mismos valores, ya que se les ha impuesto en la cláusula `WHERE` que sus valores deben coincidir. Por tanto, estas columnas ofrecen una salida duplicada y DES ofrecerá el siguiente aviso:

```
Warning: [Sem] Duplicated output column values in SELECT list.
```

5.2.3. Cláusula `FROM`

Error 5: Tabla no utilizada

Si alguna de las tablas declaradas en la cláusula `FROM` no ha sido accedida en ningún lugar de la consulta.

```
SELECT 1
FROM EMPLOYEE
```

Esta consulta no utiliza en ningún momento la relación `EMPLOYEE`, ya que simplemente devolverá un 1 por cada fila de la tabla `EMPLOYEE`, generando la siguiente advertencia:

```
Warning: [Sem] Columns of relation 'EMPLOYEE' are not used.
```

Error 6: `JOIN` innecesario

Este error ocurre cuando se hace un `JOIN` con una tabla que no aporta información adicional. Por ejemplo, si sólo se accede a su clave primaria y esa clave resulta que es una clave foránea de otra tabla. También sucede cuando se usan dos alias de la misma tabla innecesariamente. Estos joins pueden ralentizar las consultas y hacerlas más difíciles de leer.

```
SELECT E.EMPNO
FROM EMPLOYEE E JOIN DEPARTMENT D
ON E.WORKDEPT = D.DEPTNO;
```

En este caso, la relación `DEPARTMENT` no aporta ningún valor adicional y DES imprimirá por pantalla el siguiente mensaje de aviso:

```
Warning: [Sem] Unnecessary join with 'DEPARTMENT'. There is a foreign key
relating this with 'EMPLOYEE', and non-key attributes are not accessed.
```

Se podrían obtener los mismos resultados simplemente con el uso de la relación `EMPLOYEE`, tal y como muestra el siguiente ejemplo simplificado:

```
SELECT EMPNO
FROM EMPLOYEE;
```

Error 7: Tablas siempre idénticas

Se produce cuando se usan alias de la misma tabla (`X`, `Y`) y se les impone una condición que iguala sus claves primarias (`X.id = Y.id`). Esto hace que ambos alias se refieran siempre a la misma fila, volviendo uno de ellos innecesario. Es un caso especial de `JOIN` innecesario (Error 6), y muchas veces conduce a una consulta vacía, como en el Error 1. Sin embargo, como ocurre con frecuencia, se considera que merece una advertencia específica.

```
SELECT E1.FIRSTNME, E2.LASTNAME
FROM EMPLOYEE E1, EMPLOYEE E2
WHERE E1.EMPNO = E2.EMPNO;
```

En este caso, la relación `E1` y `E2` de `EMPLOYEE` contendrán los mismos datos, ya que se ha filtrado por la condición `E1.EMPNO = E2.EMPNO`. Una de las dos relaciones es innecesaria. Para este caso, DES ofrece la siguiente salida:

```
Warning: [Sem] Tuple variables are always identical for the different
occurrences of 'EMPLOYEE'.
```

5.2.4. Cláusula WHERE

Error 8: Condición implícita, tautológica o inconsistente

Este error ocurre cuando en la cláusula WHERE las condiciones son siempre verdaderas, redundantes dentro de conjunciones o disyunciones...

Por ejemplo:

```
SELECT FIRSTNME
FROM EMPLOYEE
WHERE (SEX = 'M' OR SEX = 'F');
```

En este ejemplo, el atributo SEX sólo puede tomar los valores 'M' y 'F', por lo que la condición del WHERE siempre va a ser verdadera y, por consiguiente, no aporta nada a la consulta. Se lanza la siguiente advertencia por pantalla:

```
Warning: [Sem] Tautological condition: (SEX = 'M' OR SEX = 'F').
```

Error 9: Comparación con NULL

Condiciones como = NULL provocan este error. Un ejemplo de mal uso puede ser el siguiente:

```
SELECT SALARY
FROM EMPLOYEE
WHERE WORKDEPT = NULL;
```

Ante este caso, aparece la siguiente advertencia:

```
Warning: [Sem] Null comparison in: 'WORKDEPT = NULL'. Consider using IS NULL
instead.
```

El sistema sugiere que se usen términos como IS NULL o IS NOT NULL, ya que cualquier comparación con un valor NULL es siempre falsa. El ejemplo corregido sería el siguiente:

```
SELECT SALARY
FROM EMPLOYEE
WHERE WORKDEPT IS NULL;
```

Error 11: Operador de comparación excesivamente genérico

Se produce cuando se usan operadores más generales de lo necesario, como >=, <=, IN..., cuando un operador más específico expresaría lo mismo de forma más clara y precisa. Aunque el resultado de la consulta no cambie, se reduce la legibilidad del código y puede llevar a alguna confusión.

```
SELECT EMPNO
FROM EMPLOYEE
WHERE WORKDEPT LIKE '%';
```

En esta ocasión, el sistema ofrecerá la siguiente salida:

```
Warning: [Sem] Condition 'WORKDEPT LIKE '%'' can be better rewritten as
'WORKDEPT IS NOT NULL'.
```

Error 12: `LIKE` sin comodines

Se trata de un error cuando se usa `LIKE` sin el símbolo `_` y `%`, que son esenciales para que este operador cumpla su función principal: realizar búsquedas por patrones dentro de cadenas de texto.

```
SELECT LASTNAME
FROM EMPLOYEE
WHERE LASTNAME LIKE 'A';
```

Aunque el resultado normalmente es el mismo, se debe usar el operador `=`, mejorando la claridad y el rendimiento, además de evitar interpretaciones erróneas. El mensaje de advertencia que ofrece DES en este caso es el siguiente:

```
Warning: [Sem] LIKE pattern 'A' without wildcards (either % or _).
```

Error 13: `SELECT` innecesariamente complicado en subconsultas `EXISTS`

En las subconsultas `EXISTS`, la lista de atributos que aparece junto a `SELECT` no importa demasiado. Debería ser algo simple como `*`, una constante o un único atributo. Incluso el uso de `DISTINCT` sería extraño.

```
SELECT E1.LASTNAME
FROM EMPLOYEE E1
WHERE EXISTS (SELECT E2.SALARY, E2.SEX FROM
    EMPLOYEE E2 WHERE E2.SEX = 'M');
```

Por ejemplo, en este caso, si se quiere saber si existen empleados que sean de sexo masculino, no hace falta que en la subconsulta se extraigan las columnas `E2.SALARY` y `E2.SEX`, ya que no tienen ninguna utilidad en la consulta. DES ofrece la siguiente salida:

```
Warning: [Sem] Unnecessarily complicated SELECT in EXISTS-subquery.
Consider using '*' instead of 'E2.SALARY, E2.SEX'.
```

Una posible modificación de la anterior consulta podría ser la siguiente:

```
SELECT E1.LASTNAME
FROM EMPLOYEE E1
WHERE EXISTS (SELECT 1 FROM EMPLOYEE E2
              WHERE E2.SEX = 'M');
```

Con saber si hay una fila que cumpla las condiciones de la subconsulta es suficiente, por lo que solo se necesita extraer `SELECT 1`.

5.2.5. Funciones de agregación

Error 16: `DISTINCT` innecesario en funciones de agregación

Funciones de agregación como `MIN` o `MAX` nunca necesitan `DISTINCT`. A su vez, cuando el `DISTINCT` es usado en otras funciones de agregación, podría no ser necesario.

```
SELECT WORKDEPT, COUNT(DISTINCT EMPNO)
FROM EMPLOYEE
WHERE SALARY >= 2000
GROUP BY WORKDEPT;
```

En este ejemplo, al ser el atributo `EMPNO` una clave primaria de la relación `EMPLOYEE`, no es necesario añadir el `DISTINCT` a la función de agregación `COUNT`, ya que los valores del atributo `EMPNO` serán únicos. Para este caso, se ofrece el siguiente mensaje de advertencia:

```
Warning: [Sem] Unnecessary DISTINCT in COUNT because it applies to a key.
```

Si el atributo al que se le aplica el `DISTINCT` hubiese sido una clave candidata, el tratamiento habría sido el mismo.

```
SELECT MAX(DISTINCT SALARY)
FROM EMPLOYEE;
```

Como se ha dicho con anterioridad, el `DISTINCT` con las funciones de agregación `MAX` y `MIN` no tiene ningún efecto, ya que siempre se va a tomar el mayor o menor valor, sin importar el número de veces que se repita. DES ofrece la siguiente advertencia:

```
Warning: [Sem] DISTINCT should not be applied to the argument of MAX.
```

Error 17: Argumento innecesario en `COUNT`

Hay dos versiones de la función de agregación `COUNT`. Una con un argumento y otra sin argumento (simplemente con `COUNT(*)`). Cuando no hay `DISTINCT` y el argumento no puede ser nulo, se aconsejará la versión sin argumento.

```
SELECT COUNT(EMPNO)
FROM EMPLOYEE
WHERE SALARY > 2500
GROUP BY EMPNO, SALARY;
```

Al ejecutar esta consulta en DES, salta el siguiente mensaje:

```
Warning: [Sem] Unnecessary argument of COUNT because it cannot be null due
to a primary key.
```

En este caso, se tiene el atributo `EMPNO`, que es clave primaria de la relación `EMPLOYEE`, por lo que nunca puede ser nulo. Por tanto, `COUNT(EMPNO)` actuará exactamente igual que `COUNT(*)`, así que se puede omitir `EMPNO` ya que no aporta nada y reemplazarlo por `*`.

5.2.6. Cláusula `HAVING`

Dado que estos errores coinciden con los de la cláusula `WHERE` que han sido vistos con anterioridad, se considerarán como ya revisados.

5.2.7. Formulaciones ineficientes

Error 25: Ineficiencia en el uso de `HAVING`

Si la condición del `HAVING` involucra únicamente atributos de `GROUP BY` sin funciones de agregación, esa condición puede escribirse tanto en `WHERE` como en `HAVING`. Sin embargo, es mucho más eficiente verificarla en la cláusula `WHERE`.

```
SELECT WORKDEPT
FROM EMPLOYEE
GROUP BY WORKDEPT
HAVING WORKDEPT = 'A00';
```

En este ejemplo, la condición del `HAVING` no usa funciones de agregación, por lo que esta condición podría evaluarse antes del agrupamiento, y por tanto, debería estar en la cláusula `WHERE`, reduciendo filas antes de agrupar, mejorando el rendimiento del programa. El sistema DES lanza la siguiente advertencia:

`Warning: [Sem] Conditions over grouped columns [EMPLOYEE.WORKDEPT] occur in the HAVING clause without an aggregate. Consider moving them to the WHERE clause.`

El ejemplo corregido sería:

```
SELECT WORKDEPT
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
GROUP BY WORKDEPT;
```

5.2.8. Violaciones de patrones estándar

Error 27: Falta de condición de `JOIN`

Este error ocurre cuando una consulta que involucra varias tablas no especifica correctamente las condiciones de unión. Esto puede generar resultados incorrectos.

```
SELECT E.FIRSTNME, E.LASTNAME, D.DEPTNAME
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.SALARY > 30000;
```

Se puede ver cómo se están combinando las variables relacionales `EMPLOYEE` y `DEPARTMENT` sin una condición de `JOIN`, lo que genera un producto cartesiano entre ambas relaciones. DES avisa de este aspecto mediante la siguiente advertencia:

`Warning: [Sem] Missing join condition for [EMPLOYEE,DEPARTMENT].`

Solo se filtra por los empleados con salarios mayores de un valor, pero falta una condición necesaria que relacione empleados con departamentos, como puede ser el siguiente resultado:

```
SELECT E.FIRSTNME, E.LASTNAME, D.DEPTNAME
FROM EMPLOYEE E JOIN DEPARTMENT D
ON E.WORKDEPT = D.DEPTNO
WHERE E.SALARY > 30000;
```

Error 32: `HAVING` innecesario

El uso de `HAVING` sin `GROUP BY` es extraño. Como `HAVING` sirve para filtrar grupos, y en este caso no hay agrupación, consecuentemente la consulta solo evalúa un 'grupo'. En la mayoría de los casos, se debe usar `WHERE` en su lugar.

```
SELECT SUM(SALARY)
FROM EMPLOYEE
HAVING AVG(SALARY) > 5000.0;
```

Esta consulta produce el siguiente aviso:

`Warning: [Sem] Found a HAVING clause with condition 'AVG(SALARY) > 5000.0' without a GROUP BY clause.`

Tal y como se ha explicado, en este caso, al no haber un `GROUP BY`, la relación `EMPLOYEE` es un único grupo, por lo que está aplicando la condición del `HAVING` a toda la tabla. Se debe reemplazar `HAVING` por `WHERE`, aunque como `WHERE` no puede usar funciones de agregación, la consulta correcta quedará reestructurada de la siguiente manera:

```
SELECT SUM(SALARY)
FROM EMPLOYEE
WHERE (SELECT AVG(SALARY)
       FROM EMPLOYEE) > 5000;
```

Error 33: `DISTINCT` **en** `SUM` **y** `AVG`

El uso de `DISTINCT` dentro de `SUM` o `AVG` es algo sospechoso que puede llevar a resultados no esperados o incorrectos, ya que eliminan duplicados que probablemente deberían contarse.

```
SELECT AVG(DISTINCT SALARY)
FROM EMPLOYEE;
```

En este caso, se estarían eliminando salarios duplicados, algo que puede alterar la media de salarios de los empleados. Por tanto, se avisa al usuario para que sea consciente de lo que está pasando mediante la siguiente advertencia:

`Warning: [Sem] Using SUM with DISTINCT might not be appropriate.`

Capítulo 6

Propuesta y resolución

Se analizan las principales causas de los errores semánticos en consultas SQL, destacando aquellos que ocurren con mayor frecuencia. Además, se detallan las mejoras introducidas en el análisis semántico del sistema DES como parte del desarrollo de este proyecto.

6.1. Motivación

Antes de abordar la ampliación de la capacidad del sistema DES para detectar errores semánticos en consultas SQL, se consideró necesario realizar un estudio preliminar que permitiera identificar las causas más habituales de este tipo de errores y en qué partes de las consultas suelen producirse con mayor frecuencia. Para ello, se revisaron trabajos previos y estudios estadísticos [16, 18] centrados en el comportamiento de los estudiantes a la hora de escribir consultas SQL.

En el primero de ellos, se realizó un experimento con 104 estudiantes a los que se propusieron seis ejercicios de SQL, recogiendo un total de 8.829 intentos de respuesta. Los resultados fueron contundentes: en torno al 57 % de los intentos contenían errores semánticos. Por otro lado, el segundo estudio se realizó a 286 estudiantes y se identificó que entre el 22 % y el 45 % de las consultas también presentaban errores de este tipo. Ambos estudios coinciden en que los errores semánticos no son solo frecuentes, sino más difíciles de superar que los sintácticos.

Además, el segundo artículo destaca que en el 58 % de los casos en los que el estudiante abandonó un ejercicio, su último intento fue una consulta semánticamente incorrecta, aunque sintácticamente válida. Esto pone de manifiesto lo difícil y frustrante que resulta para los estudiantes identificar y corregir estos errores lógicos en sus consultas.

En ambos artículos se analizan las estrategias que los estudiantes adoptan frente a estos errores. Muestran que los errores semánticos suelen derivarse de una gestión deficiente de la complejidad: uso excesivo de subconsultas y paréntesis, estructuras innecesariamente complejas y poca claridad general en la construcción de la consulta. Además, en lugar de simplificar la consulta tras un error, muchos estudiantes optan por añadir más elementos, lo que aumenta la complejidad de la consulta sin llegar a resolver el problema original. De hecho, en el 36.8 % de los casos analizados, el intento posterior a un error fue más complejo que el anterior. También se observó que los estudiantes en muy pocas ocasiones reformulan de cero las consultas. La mayoría de los cambios entre intentos eran mínimos y el 49.5 % implicaba una modificación muy leve.

Se coincide en que los errores más recurrentes están relacionados con un mal uso de las subconsultas y una notable dificultad con la cláusula `GROUP BY`. Como recomendación, proponen fomentar estrategias de desarrollo paso a paso y animar a los estudiantes a reestructurar sus soluciones desde cero cuando detectan que su enfoque inicial no funciona.

Con esta base, se decidió profundizar en un análisis más cuantitativo sobre los errores semánti-

Concepto	Aciertos	Error sintáctico	Error semántico
`GROUP BY` con `HAVING`	4 %	58 %	37 %
`SELF-JOIN`	2 %	37 %	61 %
`GROUP BY`	7 %	63 %	30 %
`NATURAL JOIN`	4 %	64 %	32 %
`Subconsultas`	5 %	61 %	34 %
`Consultas con una tabla`	11 %	48 %	41 %
`Subconsultas correlacionadas`	6 %	52 %	42 %
`Total`	6 %	54 %	40 %

Tabla 6.1: Principales errores. Tabla recogida en el artículo [1].

Cláusula	Error	Concepto
`WHERE (46%)`	Omisión o error en la condición	`SELF-JOIN`, subconsulta correlacionada, `JOIN`
`FROM (26%)`	`SELF-JOIN` sin usar	`SELF-JOIN`
`HAVING (13%)`	Omisión de la cláusula `GROUP BY` o `HAVING`, uso de columna equivocada	`HAVING`
`GROUP BY (5%)`	Omisión de la cláusula `GROUP BY`, uso de columna incorrecta	`GROUP BY` con y sin `HAVING`
`ORDER BY (5%)`	Omisión de la cláusula `ORDER BY`, uso de columna incorrecta o incompleta	`GROUP BY` y consultas con una única tabla
`SELECT (5%)`	Omisión de columna, columna innecesaria	`GROUP BY` y consultas con una única tabla

Tabla 6.2: Porcentaje de error de cada tipo de consulta. Tabla recogida en el artículo [1].

cos más comunes. Para ello, se recurrió a un estudio [1] que recoge datos de más de 161.000 intentos de consulta SQL por parte de 2.300 estudiantes, recopilados a lo largo de nueve años mediante una herramienta llamada AsseSQL. En este estudio, se clasifican los errores por tipo de consulta, diferenciando entre aciertos, errores sintácticos y errores semánticos.

En la Tabla 6.1 se puede ver que el `SELF-JOIN` aparece como el tipo de consulta con mayor tasa de error semántico (61 %), seguido de cerca por las subconsultas correlacionadas, las consultas con una única tabla y el `GROUP BY` con `HAVING`. Además, en el artículo se menciona que el número de intentos finales con errores semánticos es tres veces superior al número de intentos finales con errores sintácticos.

Por otro lado, la Tabla 6.2 muestra los errores semánticos exclusivamente en los intentos finales de los estudiantes. Se observa que la mayoría de estos errores se deben a la omisión de cláusulas clave (`GROUP BY`, `HAVING`, `JOIN`...), el uso incorrecto de condiciones (`WHERE`) o la selección de columnas innecesarias en `SELECT`.

Profundizando en el análisis de los datos, se observa que aunque las consultas simples sobre una única tabla presentan la mayor tasa de aciertos, no están exentas de errores.

En consultas que requieren el uso de `GROUP BY`, muchos estudiantes directamente omiten la cláusula o la utilizan de forma incorrecta, ya sea incluyendo columnas que no deberían agruparse o escribiendo condiciones en la cláusula `WHERE` que deberían estar en `HAVING`. En ejercicios que requerían subconsultas, más del 80 % de los alumnos no las utilizó y en el caso de subconsultas correlacionadas, aproximadamente el 70 % no logró identificar correctamente el patrón esperado.

Respecto al uso de `JOIN`, se detectó una omisión sistemática de las condiciones de unión, lo que provoca una especial confusión con el uso de `NATURAL JOIN`. El caso más problemático es el

de los SELF-JOIN, donde el 75 % de los estudiantes no comprendió que era necesario replicar la tabla para resolver la consulta correctamente.

6.2. Nuevos errores implementados

Tras el estudio previo, se concluyó que este trabajo se centraría, en primer lugar, en corregir ciertos errores para mejorar su funcionalidad pero, sobre todo, en desarrollar nuevos algoritmos principalmente enfocados en la cláusula GROUP BY ya que como se observó en la sección anterior, es una fuente frecuente de errores semánticos por parte de los usuarios y el sistema DES aún no está suficientemente desarrollado en este aspecto. Además, se implementarán algoritmos para detectar errores relacionados con las cláusulas ORDER BY y UNION. El código implementado se puede consultar en el archivo des_sql_semantic.pl. Para más información acerca de la ejecución del programa, consúltese el Apéndice I.

6.2.1. Error 2: DISTINCT innecesario

Al estudiar este error en profundidad, se descubrió que había varios casos que no estaban recogidos en el algoritmo implementado en ese momento. Por ello, surgió la idea de ampliar el alcance del sistema DES en el reconocimiento de este error. Cabe recordar que este error se daba cuando en algunas ocasiones, se incluía el término DISTINCT, a pesar de que la consulta no podía devolver valores duplicados. Por tanto, en estos casos, DISTINCT debería ser eliminado.

```
SELECT DISTINCT EMPNO
FROM EMPLOYEE
```

En esta consulta, como el atributo EMPNO es una clave primaria, no es necesario añadir DISTINCT ya que este atributo tendrá valores únicos por definición.

El algoritmo implementado hasta ese momento solo recogía los casos relacionados con claves primarias y candidatas. La mejora introducida en este proyecto es principalmente la incorporación del análisis del caso GROUP BY, el cual no se contemplaba previamente. Por ejemplo, se considera la siguiente consulta:

```
SELECT DISTINCT SALARY, BONUS
FROM EMPLOYEE
GROUP BY SALARY, BONUS;
```

En esta consulta, la cláusula GROUP BY agrupa las filas de la tabla EMPLOYEE por cada combinación única de valores SALARY y BONUS. Como resultado, el conjunto de salida contendrá una única fila por cada grupo distinto de estos atributos. Por tanto, la cláusula DISTINCT es redundante y no tiene ningún efecto adicional ya que la unicidad de resultados ya está garantizada mediante el GROUP BY.

Para ampliar la alcanzabilidad de este error, se decide seguir el siguiente algoritmo diseñado en [2]. Este algoritmo permite detectar cuándo el uso de DISTINCT es innecesario ya que la consulta, por su estructura y restricciones, no puede devolver filas duplicadas. Para ello, analiza las condiciones de la cláusula WHERE, los atributos que aparecen en GROUP BY, identifica qué atributos determinan de forma unívoca cada fila del resultado y concluye que DISTINCT es redundante si esos atributos garantizan unicidad.

1. Se expresan las condiciones de la cláusula `WHERE` en forma normal conjuntiva.
2. Sea K el conjunto de t_i que son referencias a atributos (es decir, no son términos compuestos ni constantes). Inicialmente, K es el conjunto de atributos que aparece como columnas de salida en la cláusula `SELECT`. La idea es que K contenga aquellos atributos que tienen un único valor para cada fila del resultado.
3. Se añaden a K todos los atributos A tales que $A = c$ o $c = A$, siendo c una constante que aparece en alguna condición de la cláusula `WHERE`.
4. Mientras K cambie con respecto a la iteración anterior, se repite el siguiente bucle:
   - Se añaden a K aquellos atributos A tales que $A = B$ o $B = A$ aparecen en alguna condición de la cláusula `WHERE` y $B \in K$.
   - Si K contiene una clave (primaria o candidata) de alguna de las tablas X_i del `FROM`, entonces se añaden a K el resto de atributos pertenecientes a esa relación X_i.
5. Si K contiene una clave (primaria o candidata) de todas las relaciones X_i, entonces la cláusula `DISTINCT` es superflua. Además, si la consulta contiene una cláusula `GROUP BY`, se comprueba que todas las columnas del `GROUP BY` están incluidas en K.

Este algoritmo se implementa mediante el predicado `check_sql_unnec_distinct(SQLst)`, que recibe como único parámetro de entrada el árbol sintáctico de la consulta SQL. Dependiendo del motivo por el cual la cláusula `DISTINCT` resulta redundante, DES genera uno de los siguientes mensajes de advertencia:

- Cuando K incluye una clave de todas las relaciones X_i involucradas en la consulta:

  `Warning: [Sem] Unnecessary DISTINCT because of primary key in [`X_i`].`
- Cuando el uso de `DISTINCT` es innecesario debido a la presencia de una cláusula `GROUP BY` que ya garantiza unicidad:

  `Warning: [Sem] Using unnecessary DISTINCT because of GROUP BY.`

6.2.2. Error 8: Condición implícita, tautológica o inconsistente

Se trata de arreglar el siguiente caso que resultaba en un falso positivo del Error 8.

```
SELECT D.DEPTNO, D.DEPTNAME
FROM DEPARTMENT D
LEFT JOIN EMPLOYEE E
    ON E.WORKDEPT = D.DEPTNO
WHERE E.LASTNAME IS NULL;
```

En la implementación original del algoritmo que detecta el Error 8, se genera una advertencia siempre que se utilice la condición `IS NULL` sobre una columna que ha sido declarada como `NOT NULL` en el momento de definir la tabla. En ese caso, se lanza el siguiente mensaje:

`Warning: [Sem] Inconsistent condition: IS NULL is applied to a column with a NOT NULL constraint ('E'.'LASTNAME').`

Esto ocurre, por ejemplo, con la columna `LASTNAME`, que está definida como no nula en la tabla `EMPLOYEE`.

Sin embargo, este tipo de comportamiento no es del todo correcto en consultas que incluyen cláusulas del tipo `LEFT JOIN` ya que, en ese contexto, las columnas pertenecientes a la tabla

del lado derecho sí que pueden contener valores nulos cuando no existe correspondencia con las filas de la tabla del lado izquierdo. Ocurre lo mismo en el caso `RIGHT JOIN`, donde las columnas potencialmente nulas provienen de la tabla de la izquierda y en `FULL JOIN`, donde ambas tablas pueden aportar valores nulos.

Para solucionar este problema y evitar que la advertencia se lance de forma incorrecta en estos casos, se ha ajustado el algoritmo. Ahora, al procesar un `JOIN`, se identifica cuál es la tabla que puede introducir valores nulos (dependiendo del `JOIN`) y se guarda su nombre. A partir de ahí, si la condición `IS NULL` se aplica a una columna de dicha tabla no se considera un error semántico ya que se reconoce que es válida la posibilidad de nulidad en ese contexto. El código se recoge en el predicado `check_sql_null_tautological_inconsistent_condition(SQLst, Rs)`, recibiendo como parámetros de entrada el árbol sintáctico de SQL y las reglas de Datalog.

6.2.3. Error 18: `GROUP BY` innecesario en subconsulta `EXISTS`

Este error consiste en identificar como innecesaria la cláusula `GROUP BY` cuando aparece dentro de una subconsulta que utiliza `EXISTS`. En este tipo de subconsultas, la función de `EXISTS` es simplemente verificar la existencia de filas que cumplan una condición, por lo que una agrupación mediante `GROUP BY` no tiene ningún efecto si no está acompañada de una cláusula `HAVING`.

Es decir, si la subconsulta solo necesita confirmar si existen filas, no se justifica agrupar los datos. En el caso de que el uso de `GROUP BY` fuese estrictamente necesario debido a la presencia de funciones de agregación en la cláusula `SELECT`, entonces se tendría una instancia del Error 13, ya contemplado anteriormente.

Un ejemplo de este error sería:

```
SELECT DEPTNO, DEPTNAME
FROM DEPARTMENT D
WHERE EXISTS (
    SELECT WORKDEPT
    FROM EMPLOYEE E
    WHERE E.WORKDEPT = D.DEPTNO
    GROUP BY WORKDEPT
    );
```

En este caso, `GROUP BY` dentro de la subconsulta no aporta ningún valor ya que `EXISTS` únicamente verifica si hay al menos una fila que cumpla la condición `E.WORKDEPT = D.DEPTNO`. Como no hay ninguna cláusula `HAVING`, la agrupación es redundante. Por tanto, esta situación es detectada como un error semántico.

Una posible solución a este error podría ser:

```
SELECT DEPTNO, DEPTNAME
FROM DEPARTMENT D
WHERE EXISTS (
    SELECT 1
    FROM EMPLOYEE E
    WHERE E.WORKDEPT = D.DEPTNO
    );
```

Esta consulta obtiene los mismos resultados comprobando únicamente que hay alguna tupla que cumple con las condiciones de la subconsulta, sin necesidad de agrupamiento.

El algoritmo que se ha implementado para reconocer este error se recoge en el predicado `check_group_by_in_exists_subqry(SQLst)`, recibiendo como único parámetro de entrada el

árbol sintáctico de la consulta SQL. Tras comprobar que la subconsulta `EXISTS` tiene cláusula `GROUP BY` y no tiene cláusula `HAVING`, se imprimirá el siguiente mensaje de advertencia por pantalla:

```
Warning: [Sem] Using unnecessary GROUP BY in EXISTS subquery.
```

6.2.4. Error 19: `GROUP BY` de grupos unitarios

Se reconoce este error semántico cuando la agrupación mediante la cláusula `GROUP BY` tiene como resultado que cada grupo consta solamente de una única fila.

```
SELECT EMPNO
FROM EMPLOYEE
GROUP BY EMPNO;
```

Se recuerda que el atributo `EMPNO` es una clave primaria de la tabla `EMPLOYEE`. Esto significa que el atributo `EMPNO` siempre tomará valores únicos, no habrá duplicados. Por tanto, si el criterio de agrupación es mediante este atributo, cada grupo estará formado únicamente por una fila, que coincidirá con uno de los valores únicos que se recogen en `EMPNO`. Se trataría del mismo problema, si en vez de ser una clave primaria fuese una clave candidata.

```
SELECT WORKDEPT, COUNT(*)
FROM EMPLOYEE
GROUP WORKDEPT;
```

En este caso no se produce el error ya que el criterio de agrupación ya no es en base a un atributo que sea clave, sino al atributo `WORKDEPT`, que no es ni clave primaria ni candidata, por lo que no toma valores únicos y por tanto el criterio de agrupación general no produce que haya agrupaciones que consten de una sola fila.

A su vez, se ha tenido en cuenta que, aunque todos los atributos que aparecen en el criterio de agrupación no sean claves primarias o candidatas, se debe comprobar también que en caso de existir una clave de alguno de estos tipos, el resto de atributos que acompañan la cláusula `GROUP BY` no dependen funcionalmente de esta clave ya que en este caso, sí que se volverían a obtener agrupaciones con una única fila.

El algoritmo que se ha implementado para reconocer este error se puede expresar en los siguientes pasos:

1. Se comprueba que la cláusula `GROUP BY` no está vacía.

2. Sea K_G el conjunto de atributos t_i que aparecen en la cláusula `GROUP BY`.

3. Sea K'_G el subconjunto de atributos de la cláusula `GROUP BY` que, a su vez, son claves primarias (*primary key*) o claves candidatas (*candidate key*) de las tablas X_i asociadas a los atributos de K_G. Es decir, $K'_G \subseteq K_G$.

4. Se comprueba si $K_G = K'_G$. En caso afirmativo, se muestra el siguiente mensaje de advertencia por pantalla:

   ```
   Warning: [Sem] GROUP BY with singleton groups.
   ```

   En caso contrario, se continúa en el paso 5.

5. Sea DF el cierre transitivo del conjunto de dependencias funcionales que existen en las relaciones X_i. También se consideran como dependencias funcionales aquellas comparaciones en la cláusula `WHERE` entre atributos tales que $A = B$, lo que implica $A \rightarrow B$ y $B \rightarrow A$, tal y como se ha explicado en el Capítulo 2.

6. Sea K'' el conjunto de atributos que, en base a las dependencias funcionales del conjunto DF, dependen funcionalmente de algún atributo de K'_G.

7. Sea $K''' = K'_G \cup K''$. En caso de que $K''' = K_G$, se lanzará el mismo mensaje que en el paso 4. En caso contrario, se puede asegurar que la consulta no contiene este error semántico.

La detección de este error en el algoritmo implementado se lleva a cabo mediante el predicado `check_group_by_with_singleton_groups(SQLst,Closure)`, el cual toma como entrada el árbol sintáctico de la consulta SQL. Dentro de este predicado se calculan las dependencias funcionales y su correspondiente cierre transitivo. Dado que este cálculo es necesario para otros errores (concretamente el Error 21 y el Error 24), se ha definido el parámetro de salida `Closure` con el objetivo de reutilizar el resultado en los predicados posteriores. Esta estrategia mejora la eficiencia del código ya que evita realizar el mismo cálculo múltiples veces.

6.2.5. Error 20: `GROUP BY` de un solo grupo

Si hay siempre un único grupo, la cláusula `GROUP BY` es innecesaria, excepto cuando el atributo que se encuentra en `GROUP BY` forma parte de la lista de proyección `SELECT` (sin ser argumento de una función de agregación). Se tiene el siguiente ejemplo:

```
SELECT COUNT(SALARY)
FROM EMPLOYEE
WHERE SALARY = 2000
GROUP BY SALARY;
```

Aquí se puede ver que al filtrar en la cláusula `WHERE` por los empleados que tienen un salario de una determinada cantidad, todas las filas tendrán el mismo valor para `SALARY`, lo que hace que `GROUP BY` solo genere un único grupo. Por tanto, como `SALARY` ya es constante y no aparece en la lista de los atributos que se proyectan en `SELECT`, se puede omitir el `GROUP BY`. La consulta corregida sería la siguiente:

```
SELECT COUNT(SALARY)
FROM EMPLOYEE
WHERE SALARY = 2000;
```

Este error se ha implementado mediante el predicado `check_group_by_only_with_one_group(SQLst)`, recibiendo como único parámetro de entrada el árbol sintáctico de la consulta SQL. En primer lugar, se extraen los atributos que vienen en la cláusula `SELECT` y se comprueba que efectivamente no hay coincidencias con los atributos del `GROUP BY` ya que en caso contrario, no se daría el error. Una vez comprobado lo anterior, se extraen los atributos que están igualados a constantes, subconsultas que devuelven un único valor o subconsultas con el término `ALL`. Una vez extraídos todos los atributos que cumplen los casos anteriormente mencionados, se comprueba que los atributos de la cláusula `GROUP BY` coinciden con alguno de estos atributos. En caso de que todo lo anterior se cumpla, se emitirá el mensaje:

`Warning: [Sem] GROUP BY with only a single group.`

6.2.6. Error 21: Término innecesario en `GROUP BY`

Se produce cuando el atributo de agrupación está funcionalmente determinado por otros atributos del `GROUP BY` y si no aparece en `SELECT` o `HAVING` fuera de las posibles agregaciones, puede suprimirse de la cláusula `GROUP BY`.

```
SELECT SALARY
FROM EMPLOYEE
WHERE SALARY = BONUS
GROUP BY SALARY, BONUS;
```

En este caso, se puede ver que en la cláusula `WHERE` se igualan los atributos `SALARY` y `BONUS`. Esto se traduce como si hubiese una dependencia funcional recíproca entre ellos, es decir,

$$\texttt{SALARY} \rightarrow \texttt{BONUS}$$

$$\texttt{BONUS} \rightarrow \texttt{SALARY}$$

Por tanto, como `SALARY` aparece en la cláusula `SELECT`, el atributo redundante que se puede eliminar de la cláusula `GROUP BY` será `BONUS`.

Esta consulta podría ser simplificada de la siguiente manera:

```
SELECT SALARY
FROM EMPLOYEE
WHERE SALARY = BONUS
GROUP BY SALARY;
```

La agrupación por salario es suficiente para obtener el mismo resultado que en la consulta errónea.

El algoritmo que se ha desarrollado para reconocer este tipo de error semántico sigue el siguiente esquema:

1. Se comprueba que la cláusula `GROUP BY` no está vacía.
2. Sea K_S el conjunto de atributos t_i que aparecen en la cláusula `SELECT` y K_H el conjunto de atributos t_i que aparecen en la cláusula `HAVING`.
3. Sea K el conjunto resultante de la unión de los conjuntos K_S y K_H, es decir, $K = K_S \cup K_H$.
4. Sea K_G el conjunto de atributos t_i que se encuentran en la cláusula `GROUP BY`.
5. Sea DF el cierre transitivo del conjunto de dependencias funcionales que existen en las relaciones X_i. También se consideran como dependencias funcionales aquellas comparaciones en la cláusula `WHERE` entre atributos tales que $A = B$, lo que implica $A \rightarrow B$ y $B \rightarrow A$.
6. Se itera sobre cada atributo $t_i \in K_G$ y, para cada uno, se comprueba si, en base a las dependencias funcionales recogidas en DF, dicho atributo depende funcionalmente de algún otro atributo del conjunto K_G y este mismo atributo no se encuentra en el conjunto K. En caso de que se cumplan todas estas premisas, se muestra el siguiente mensaje por pantalla:

   `Warning: [Sem] Unnecessary GROUP BY term '`t_i`'.`

Este error se recoge en el predicado `check_if_attr_grp_by_is_unnec(SQLst,Closure)`. Este predicado recibe como parámetros de entrada el árbol sintáctico de la consulta SQL y el cierre transitivo (calculado en el Error 19), necesario para el desarrollo de este algoritmo tal y como se ha visto.

6.2.7. Error 22: `GROUP BY` puede ser reemplazado por `DISTINCT`

Si los atributos que acompañan a la cláusula `SELECT` aparecen exactamente a su vez en la cláusula `GROUP BY` y no se están usando funciones de agregación, la cláusula `GROUP BY` puede ser reemplazada por `SELECT DISTINCT`, que es más clara y concisa. Por ejemplo:

```
SELECT WORKDEPT, JOB
FROM EMPLOYEE
GROUP BY WORKDEPT, JOB;
```

En este caso, se puede ver que no se usa ninguna función de agregación (`AVG`, `SUM`...) y las columnas de la cláusula `SELECT` coinciden con las de la cláusula `GROUP BY`, por lo que lo único que se está haciendo es eliminar duplicados, así que el `GROUP BY` se podrá sustituir por el `DISTINCT`. La consulta mejorada sería así:

```
SELECT DISTINCT WORKDEPT, JOB
FROM EMPLOYEE;
```

En el algoritmo implementado en el sistema DES, este error se comprueba mediante el predicado `check_if_distinct_instead_of_group_by(SQLst)`, que recibe como único parámetro de entrada el árbol sintáctico de la consulta SQL. En primer lugar, se comprueba que la cláusula `GROUP BY` está incluida en la consulta, además de comprobar que la cláusula `HAVING` no lo está y no hay funciones de agregación en `SELECT`. Una vez realizadas estas comprobaciones, se extraen los atributos de la cláusula `SELECT` y de la cláusula `GROUP BY` y se comprueba que en efecto, estos conjuntos son iguales. Si todo esto se cumple, se imprimirá el siguiente mensaje:

```
Warning: [Sem] GROUP BY can be replaced by DISTINCT.
```

6.2.8. Error 23: `UNION` puede ser reemplazado por `OR`

Este error se produce en el caso en el que se emplea el término `UNION` o `UNION ALL` para combinar dos consultas y ambas consultas usan las mismas tablas, devuelven las mismas columnas y tienen condiciones disjuntas en la cláusula `WHERE`. Se presenta el siguiente ejemplo:

```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
UNION
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE WORDEPT = 'A01';
```

Se puede comprobar fácilmente que ambas consultas usan la misma relación en la cláusula `FROM`, devuelven las mismas columnas en el `SELECT` y las condiciones del `WHERE` son excluyentes. Por tanto, el `UNION` es innecesario y se pueden modificar ambas consultas por solo una consulta que agrupe ambas condiciones con un `OR`:

```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE WORKDEPT = 'A00' OR WORDEPT = 'A01';
```

Este error es detectado por el predicado `check_if_union_by_or(SQLst,Bss)`. Este predicado recibe como parámetros de entrada tanto el árbol sintáctico de la consulta SQL como las reglas de Datalog. En primer lugar, se verifica que ambas consultas tienen en su lista de proyección de `SELECT` los mismos atributos. A continuación, se comprueba que las relaciones en la cláusula `FROM` coinciden en ambas partes. Posteriormente, se genera una consulta simplificada que combina las condiciones de ambas cláusulas `WHERE` mediante un `AND`. Se evalúa la nueva consulta utilizando el predicado ya existente `check_sql_null_tautological_inconsistent_condition/2`, pasando el árbol sintáctico de SQL y las reglas de Datalog como parámetros. El objetivo es verificar si la conjunción de condiciones resulta inconsistente.

Si se satisfacen todas las premisas previamente explicadas, se emitirá el siguiente mensaje de aviso:

```
Warning: [Sem] UNION can be replaced by OR.
```

Cabe destacar que para crear esta nueva consulta simplificada, ha sido necesario utilizar Datalog, ya que la lógica del predicado `check_sql_null_tautological_inconsistent_condition/2` está desarrollada principalmente en dicho lenguaje.

6.2.9. Error 24: Término innecesario en `ORDER BY`

Este error se produce cuando en la cláusula `ORDER BY` se incluye una columna que está determinada funcionalmente por otra columna que aparece en la cláusula `ORDER BY`. En estos casos, este atributo no influye en el orden real y es innecesario.

```
SELECT SALARY
FROM EMPLOYEE
WHERE SALARY = BONUS
ORDER BY SALARY, BONUS;
```

En este caso, se puede ver que en la cláusula `WHERE` se igualan los atributos `SALARY` y `BONUS`. Esto se traduce como si hubiese una dependencia funcional recíproca entre ellos, es decir,

$$\texttt{SALARY} \rightarrow \texttt{BONUS}$$

$$\texttt{BONUS} \rightarrow \texttt{SALARY}$$

Por tanto, uno de los atributos será redundante en la cláusula `ORDER BY` ya que con mantener uno de ellos, prevalecerá el orden establecido.

Esta consulta podría ser simplificada de la siguiente manera:

```
SELECT SALARY
FROM EMPLOYEE
WHERE SALARY = BONUS
ORDER BY SALARY;
```

Con ordenar por salario es suficiente para obtener el mismo resultado que en la consulta errónea. El algoritmo implementado para este caso sigue el siguiente esquema:

1. Se comprueba que la cláusula `ORDER BY` no está vacía.
2. Sea K_O el conjunto de atributos t_i que se encuentran en la cláusula `ORDER BY`.
3. Sea DF el cierre transitivo del conjunto de dependencias funcionales que existen en las relaciones X_i. También se consideran como dependencias funcionales aquellas comparaciones en la cláusula `WHERE` entre atributos tales que $A = B$, lo que implica $A \rightarrow B$ y $B \rightarrow A$.

4. Se itera sobre cada atributo $t_i \in K_O$ y, para cada uno, se comprueba si, en base a las dependencias funcionales recogidas en DF, dicho atributo depende funcionalmente de algún otro atributo del conjunto K_O. En caso afirmativo, se muestra el siguiente mensaje por pantalla:

   `Warning: [Sem] Unnecessary ORDER BY term '`t_i`'.`

La implementación de este error se recoge en el predicado `check_if_ord_by_is_unnec(SQLst, Closure)`, recibiendo como parámetros de entrada el árbol sintáctico de la consulta SQL y el cierre transitivo calculado en el algoritmo del Error 19. Se procede de una forma similar al caso de `GROUP BY` innecesario, pero de una manera más simplificada. Esta vez no hay necesidad de comprobar si los atributos de la cláusula `ORDER BY` se encuentran en las cláusulas `SELECT` o `HAVING`.

6.2.10. Validación mediante archivos de prueba

La corrección de estos algoritmos se ha verificado mediante la creación de diversos archivos de prueba, que contienen consultas SQL diseñadas para cubrir distintas casuísticas de cada tipo de error. En la mayoría de casos, las consultas estaban formadas por una única tabla, aunque también se han probado casos con múltiples tablas. Estos archivos se encuentran en la carpeta del proyecto bajo el nombre de `des_sql_semantic_error_X.sql` donde `X` representa el número correspondiente al error semántico evaluado. Cabe destacar que no se ha creado ningún archivo de prueba para el Error 23, ya que actualmente DES no implementa de forma completa la verificación de errores asociados a la cláusula `UNION`. No obstante, está previsto que en el futuro se desarrollen las funcionalidades necesarias para permitir la detección y validación de este tipo de errores.

Capítulo 7

Conclusiones y trabajo futuro

El principal objetivo de este Trabajo de Fin de Grado ha sido ampliar las capacidades de la herramienta DES en la detección de errores semánticos en consultas SQL, una necesidad previamente identificada. Aunque el sistema cuenta con un módulo muy completo para el análisis sintáctico, presentaba limitaciones en lo que se refiere al análisis semántico. Esta mejora se ha enfocado especialmente en el contexto educativo con la intención de facilitar el proceso de aprendizaje de los estudiantes de bases de datos.

En cuanto a los objetivos planteados al inicio del proyecto, se puede afirmar que se han cumplido en su totalidad. Se realizó un análisis exhaustivo de las limitaciones existentes en la herramienta, así como una revisión empírica de los errores semánticos más recurrentes que cometen los estudiantes al redactar consultas SQL, basándose en estudios previos y datos estadísticos extraídos de artículos previamente mencionados. A partir de este estudio, se diseñaron e implementaron siete nuevos algoritmos destinados a la detección de errores semánticos que no estaban contemplados por el sistema DES anteriormente. Cinco de ellos han abarcado la cláusula `GROUP BY`, mientras que los otros dos tienen relación con las cláusulas `UNION` y `ORDER BY`. Además, se han corregido dos algoritmos ya existentes relacionados con errores semánticos de las cláusulas `DISTINCT` (perfeccionando su comportamiento y ampliando su cobertura) y `OUTER JOIN` (evitando que se produjesen falsos positivos en sus tres versiones `LEFT`, `RIGHT` y `FULL`).

Un aspecto fundamental del desarrollo de este proyecto ha sido la creación de conjuntos de pruebas específicas, diseñadas para comprobar la corrección de los nuevos algoritmos. Estas pruebas permiten validar la detección de un mismo tipo de error en distintos escenarios, lo que aporta gran robustez a la solución implementada.

Además, a lo largo del desarrollo de este proyecto se ha profundizado en el aprendizaje del lenguaje de programación Prolog. Pese a ser menos habitual en entornos de desarrollo actuales, resulta especialmente adecuado para tareas relacionadas con el análisis lógico y la representación declarativa del conocimiento. Asimismo, Prolog ha demostrado ser una herramienta eficaz para el prototipado rápido de algoritmos basados en reglas y patrones, lo cual ha facilitado notablemente la implementación del análisis semántico de consultas SQL. Aunque previamente solo se había tenido una toma de contacto inicial con este lenguaje en asignaturas como Programación Declarativa o Inteligencia Artificial, este trabajo ha facilitado una comprensión más profunda de sus fundamentos, así como de su uso práctico en un contexto real.

En términos de relevancia, este trabajo representa un avance notable en la capacidad pedagógica del sistema DES. Al mejorar la detección de errores semánticos, se proporciona a los estudiantes una retroalimentación más precisa y útil que contribuirá al entendimiento de las causas que producen este tipo de errores.

Respecto al impacto del trabajo, su efecto se medirá en el medio y largo plazo. En los próximos cursos académicos, será posible analizar si el uso de la herramienta mejorada genera algún tipo

de impacto en la reducción de los porcentajes de errores semánticos entre los estudiantes. Una mejora en ese sentido indicaría que la herramienta no solo ha incrementado su funcionalidad técnica, sino que a su vez ha cumplido con su propósito didáctico de forma efectiva.

7.1. Trabajo futuro

En primer lugar, tal y como se ha introducido en la sección anterior, resulta fundamental evaluar el impacto real de las mejoras introducidas mediante estudios en contextos reales docentes. De esta manera, se podrá comparar el rendimiento y evolución del aprendizaje de los estudiantes antes y después de la incorporación de los nuevos algoritmos, proporcionando así una validación objetiva sobre la eficacia del sistema.

A su vez, aunque los objetivos marcados al inicio del proyecto se han alcanzado de forma satisfactoria, el proyecto deja abiertas diversas líneas de mejora. Una de las más relevantes sería continuar ampliando la capacidad de la herramienta para reconocer errores semánticos, abordando casos aún más complejos y sutiles que aún no han sido contemplados. Un ejemplo podría ser el siguiente:

```
SELECT E1.LASTNAME
FROM EMPLOYEE E1
WHERE E1.EMPNO NOT IN (
    SELECT E2.EMPNO
    FROM EMPLOYEE E2
    WHERE E2.JOB = 'MANAGER'
);
```

Esta consulta devuelve los empleados cuyo puesto no sea `'MANAGER'`, utilizando una subconsulta que recupera los identificadores `EMPNO` de todos los empleados con dicho puesto. Sin embargo, dicha subconsulta resulta innecesaria, ya que la misma lógica puede expresarse de forma más simple y clara mediante la condición `JOB <> 'MANAGER'`. Por tanto, cuando la subconsulta no depende de otras tablas y el criterio de inclusión o exclusión se basa únicamente en un atributo específico, es preferible utilizar condiciones directas en lugar de subconsultas con `IN` o `EXISTS`, que no solo añaden complejidad, sino que también disminuyen la legibilidad de la consulta. Este tipo de casos se corresponden con el Error 15 descrito por Brass y Goldberg [2].

Por otro lado, otra línea interesante de mejora sería la incorporación de un sistema de sugerencias automatizadas que no solo identifique errores semánticos, sino que también ofrezca posibles correcciones o alternativas estructurales. Esta funcionalidad incrementaría notablemente el valor didáctico de la herramienta, ya que proporcionaría una retroalimentación más completa y orientativa, similar a la que podría ofrecer un tutor personalizado, capaz de explicar el motivo de cualquier error y guiar al estudiante hacia una solución adecuada.

Conclusions and future work

The main objective of this Final Degree Project has been to extend the capabilities of the DES tool in the detection of semantic errors in SQL queries, a previously identified need. Although the system has a very complete module for syntactic analysis, it had limitations in terms of semantic analysis. This improvement has been especially focused on the educational context with the intention of facilitating the learning process of database students.

With regard to the objectives set at the beginning of the project, it can be stated that they have been fully met. An exhaustive analysis of the existing limitations of the tool was carried out, as well as an empirical review of the most recurrent semantic errors made by students when writing SQL queries, based on previous studies and statistical data extracted from previously mentioned articles. From this study, seven new algorithms were designed and implemented to detect semantic errors that were not previously covered by the DES system. Five of them have covered the `GROUP BY` clause, while the other two are related to the `UNION` and `ORDER BY` clauses. In addition, two existing algorithms related to semantic errors of the clauses `DISTINCT` (refining their behaviour and extending their coverage) and `OUTER JOIN` (avoiding false positives in their three versions `LEFT`, `RIGHT` and `FULL`) have been corrected.

A key aspect of the development of this project has been the creation of specific test suites, designed to test the correctness of the new algorithms. These tests make it possible to validate the detection of the same type of error in different scenarios, which makes the implemented solution very robust.

In addition, throughout the development of this project, the Prolog programming language has been learnt in depth. Despite being less common in current development environments, it is particularly suitable for tasks related to logical analysis and declarative knowledge representation. Prolog has also proven to be an effective tool for rapid prototyping of algorithms based on rules and patterns, which has greatly facilitated the implementation of semantic analysis of SQL queries. Although previously there had only been an initial contact with this language in subjects such as Declarative Programming or Artificial Intelligence, this work has facilitated a deeper understanding of its fundamentals, as well as its practical use in a real context.

In terms of relevance, this work represents a remarkable advance in the pedagogical capacity of the DES system. By improving the detection of semantic errors, students are provided with more accurate and useful feedback that will contribute to the understanding of the causes of semantic errors.

Regarding the impact of the work, its effect will be measured in the medium and long term. In the next academic years, it will be possible to analyse whether the use of the improved tool generates any impact on the reduction of semantic error rates among students. An improvement in that sense would indicate that the tool has not only increased its technical functionality, but has also fulfilled its didactic purpose effectively.

Future work

Firstly, as introduced in the previous section, it is essential to evaluate the real impact of the improvements introduced through studies in real teaching contexts. In this way, it will be possible to compare the performance and evolution of student learning before and after the incorporation of the new algorithms, thus providing an objective validation of the effectiveness of the system.

At the same time, although the objectives set at the beginning of the project have been satisfactorily achieved, the project leaves several lines of improvement open. One of the most relevant would be to continue expanding the tool's capacity to recognise semantic errors, addressing even more complex and subtle cases that have not yet been contemplated. An example could be the following:

```
SELECT E1.LASTNAME
FROM EMPLOYEE E1
WHERE E1.EMPNO NOT IN (
SELECT E2.EMPNO
FROM EMPLOYEE E2
WHERE E2.JOB = 'MANAGER'
);
```

This query returns employees whose position is not `'MANAGER'`, using a subquery that retrieves the identifiers `EMPNO` for all employees with that position. However, such a sub-query is unnecessary, as the same logic can be expressed more simply and clearly by the condition `JOB <> 'MANAGER'`. Therefore, when the sub-query does not depend on other tables and the inclusion or exclusion criterion is based only on a specific attribute, it is preferable to use direct conditions instead of sub-queries with `IN` or `EXISTS`, which not only add complexity, but also decrease the readability of the query. This type of case corresponds to Error 15 described by Brass and Goldberg [2].

On the other hand, another interesting line of improvement would be the incorporation of an automated suggestion system that not only identifies semantic errors, but also offers possible corrections or structural alternatives. This functionality would significantly increase the didactic value of the tool, since it would provide a more complete and orientative feedback, similar to that which a personal tutor could offer, capable of explaining the reason for any error and guiding the student towards an adequate solution.

Apéndice I

Este apéndice proporciona una guía básica para instalar y ejecutar la herramienta DES.

1. **Requisitos previos:** Antes de comenzar, es necesario disponer de una versión compatible de SWI-Prolog. Se puede descargar en el siguiente enlace:

   https://www.swi-prolog.org/download/stable?show=all.

2. **Descarga de la herramienta DES:** La versión de SWI-Prolog de DES puede descargarse mediante el siguiente enlace:

   https://github.com/javial02/TFG-Informatica.

   Una vez descargado el archivo `.zip`, debe descomprimirse en cualquier ubicación del sistema. Esto generará una carpeta llamada `des`, que contiene todos los archivos necesarios para ejecutar la aplicación.

3. **Ubicación del archivo modificado:** Dentro de la carpeta `des`, se encuentra el archivo `des_sql_semantic.pl`, que contiene una de las partes principales del análisis semántico que realiza DES. Todo el código desarrollado en este proyecto puede encontrarse a partir de la línea 1.468, tras el comentario `'Javier Amado Lázaro starts from here'`. A partir de este punto, se encuentra la implementación de los nuevos algoritmos incorporados para la detección de errores semánticos en consultas SQL. A su vez, el Error 8 viene modificado en la línea 958.

4. **Ejecución de la herramienta:** Para ejecutar la herramienta DES, se deben seguir los siguientes pasos:

   - Abrir un terminal o línea de comandos.
   - Posicionarse en la carpeta `des` descomprimida previamente.
   - Ejecutar el siguiente comando: `swipl -l des.pl`. Esto iniciará la herramienta DES dentro del entorno SWI-Prolog.

5. **Uso básico:** Una vez arrancada la aplicación, el usuario podrá introducir consultas SQL o comandos de definición de tablas, recibiendo como salida el resultado junto con el análisis correspondiente, incluyendo posibles errores sintácticos o advertencias semánticas si se detectan.

Apéndice II

Para evidenciar las mejoras introducidas en el sistema DES, se ha utilizado un depurador basado en un comparador de archivos, concretamente en la herramienta *WinDiff*. Esta herramienta permite visualizar de forma clara las diferencias entre las salidas de dos versiones distintas de DES. En este caso, se compara la versión inicial de DES y la versión tras introducir los nuevos algoritmos implementados durante el desarrollo de este proyecto. Se podrá ejecutar la depuración del sistema DES siguiendo las instrucciones redactadas en el archivo *README.txt* que se encuentra en *DES.zip*.

DES dispone de una batería de casos de prueba diseñada específicamente para verificar la corrección de los algoritmos implementados en el análisis semántico de consultas SQL. Esta batería, al ejecutarse, genera automáticamente un fichero con la salida del sistema.

Con el objetivo de comprobar la corrección de los nuevos algoritmos implementados en este proyecto, se ha ampliado dicha batería incluyendo nuevos casos de prueba, contenidos en los archivos `des_sql_semantic_error_X.sql` mencionados anteriormente. Todos estos archivos han sido integrados en el archivo de configuración `des_semantic_5.ini`, lo que facilita su ejecución conjunta con el resto de casos de prueba ya existentes.

A continuación, se ha ejecutado la simulación de la batería de pruebas utilizando la versión original del análisis semántico de DES (disponible al inicio del proyecto) y se ha guardado su salida como referencia. Posteriormente, se ha ejecutado la misma batería de casos de prueba pero esta vez utilizando la versión modificada que incorpora los algoritmos desarrollados en este trabajo, correspondiente al archivo `des_sql_semantic.pl`.

Las salidas de ambas ejecuciones han sido comparadas mediante *WinDiff*, donde se han podido observar respuestas adicionales generadas por DES frente a consultas que anteriormente no eran detectadas como erróneas desde el punto de vista semántico. Gracias a las mejoras introducidas, el sistema identifica correctamente estos errores y muestra mensajes de salida apropiados según el tipo de fallo, tal y como se explicó en el Capítulo 6.

A continuación, se muestran algunos ejemplos ilustrativos de estas diferencias. En todas las figuras, las líneas añadidas se muestran en amarillo, mientras que aquellas modificadas o eliminadas respecto a la versión anterior aparecen en rojo.

En la Figura 7.1 se muestra cómo ahora se detecta correctamente un caso de uso innecesario de `DISTINCT` cuando se utiliza `GROUP BY`, algo que la versión anterior del sistema no era capaz de reconocer.

La Figura 7.2 ilustra la corrección de un falso positivo relacionado con el Error 8. En versiones anteriores, ciertas consultas con atributos no nulos eran marcadas erróneamente como semánticamente incorrectas, algo que ya no ocurre tras la mejora.

La Figura 7.3 representa un caso en el que DES reconoce el Error 18, asociado al uso innecesario de `GROUP BY` dentro de subconsultas con `EXISTS`.

En la Figura 7.4 se observa cómo el sistema identifica situaciones donde `GROUP BY` genera agrupaciones de una sola fila, lo que se corresponde con el Error 19.

En la Figura 7.5 se puede ver cómo ahora se detectan consultas en las que `GROUP BY` genera siempre un único grupo, haciendo innecesaria la cláusula. Esto se vincula con el Error 20.

La Figura 7.6 muestra la detección de casos en los que algunos atributos del `GROUP BY` son redundantes por estar funcionalmente determinados por otros (Error 21).

En la Figura 7.7 se presenta un ejemplo en el que el sistema concluye que el uso de `GROUP BY` puede ser reemplazado directamente por `DISTINCT`, simplificando la consulta y reflejando el Error 22.

Por último, la Figura 7.8 evidencia cómo DES identifica correctamente los términos redundantes de la cláusula `ORDER BY`, como se ha visto en el Error 24.

```
3096        DES> %
3097        DES> create or replace table t(a int, b int);
3098        DES> select distinct a,b from t group by a,b;
        !>  Warning: [Sem] Using unnecessary DISTINCT because of GROUP BY.
3099        answer(t.a:int,t.b:int) ->
3100        {
3101        }
3102        Info: 0 tuples computed.
```

Figura 7.1: Error 2: `DISTINCT` innecesario.

```
3113        DES> create table department (deptno char(3) primary key, deptname varchar(36) not null);
3114        DES> create table employee (empno char(6) primary key, firstname varchar(12), lastname varchar(15) not null, workdept char(3), f
3115        DES> %
3116        DES> select deptno, deptname from department left join employee on workdept = deptno where lastname is null order by deptno;
3117  <!    Warning: [Sem] Inconsistent condition: IS NULL is applied to a column with a NOT NULL constraint ("employee"."lastname")
3118        answer(deptno:char(3),deptname:varchar(36)) ->
3119        {
3120        }
3121        Info: 0 tuples computed.
3122        DES> select deptno, deptname from department right join employee on workdept = deptno where deptname is null order by deptno;
3123  <!    Warning: [Sem] Inconsistent condition: IS NULL is applied to a column with a NOT NULL constraint ("department"."deptname")
3124        answer(deptno:char(3),deptname:varchar(36)) ->
3125        {
3126        }
3127        Info: 0 tuples computed.
3128        DES> select deptno, deptname from department full join employee on workdept = deptno where deptname is null order by deptno;
3129  <!    Warning: [Sem] Inconsistent condition: IS NULL is applied to a column with a NOT NULL constraint ("department"."deptname")
3130        answer(deptno:char(3),deptname:varchar(36)) ->
3131        {
3132        }
3133        Info: 0 tuples computed.
```

Figura 7.2: Error 8: Condición implícita, tautológica o inconsistente.

```
3135        DES> % Error 18: Unnecessary GROUP BY in EXISTS subquery
3136        DES> create or replace table t (a int, b int, c int);
3137        DES> create or replace table s (a int, b int, c int);
3138        DES> create or replace table u (a int, b int);
3139        DES> %
3140        DES> select * from t where exists (select 1 from s group by b);
        !>  Warning: [Sem] Using unnecessary GROUP BY in EXISTS subquery.
3141        answer(t.a:int,t.b:int,t.c:int) ->
3142        {
3143        }
3144        Info: 0 tuples computed.
3145        DES> select * from t where exists (select 1 from s where s.a = t.a group by s.b);
        !>  Warning: [Sem] Using unnecessary GROUP BY in EXISTS subquery.
3146        answer(t.a:int,t.b:int,t.c:int) ->
3147        {
3148        }
3149        Info: 0 tuples computed.
```

Figura 7.3: Error 18: `GROUP BY` innecesario en subconsultas `EXISTS`.

```
3178        DES> create or replace table s(a int, b int, primary key (a, b));
3179        DES> create or replace table t(a int primary key, b string determined by a);
3180        DES> %
3181        DES> select a from t group by a;
        !>  Warning: [Sem] GROUP BY can be replaced by DISTINCT.
        !>  Warning: [Sem] GROUP BY with singleton groups.
3182        answer(t.a:int) ->
3183        {
3184        }
3185        Info: 0 tuples computed.
3186        DES> select a, b from s group by a, b;
        !>  Warning: [Sem] GROUP BY can be replaced by DISTINCT.
        !>  Warning: [Sem] GROUP BY with singleton groups.
3187        answer(s.a:int,s.b:int) ->
3188        {
3189        }
3190        Info: 0 tuples computed.
```

Figura 7.4: Error 19: `GROUP BY` de grupos unitarios.

```
3228        DES> create or replace table t(a int, b int);
3229        DES> %
3230        DES> create or replace table t(a int, b int);
3231        DES> %
3232        DES> select count(a) from t where a = 3 group by a;
3233        Warning: [Sem] Constant argument found for COUNT(3).
        !>  Warning: [Sem] GROUP BY with only a single group.
3234        answer($a3:int) ->
3235        {
3236        }
3237        Info: 0 tuples computed.
3238        DES> select 1 from t where a = 3 group by a;
        !>  Warning: [Sem] GROUP BY with only a single group.
3239        answer($a3:int) ->
3240        {
3241        }
3242        Info: 0 tuples computed.
```

Figura 7.5: Error 20: `GROUP BY` de un solo grupo.

```
3256        DES> create table j(a int, b int);
3257        DES> select a from j where a = b group by a, b;
        !>  Warning: [Sem] Unnecessary GROUP BY term "b".
3258        answer(j.a:int) ->
3259        {
3260        }
3261        Info: 0 tuples computed.
```

Figura 7.6: Error 21: Término innecesario en `GROUP BY`.

```
3304        DES> create or replace table t(a int, b int);
3305        DES> create or replace table s(a int, b int);
3306        DES> %
3307        DES> select a, b from t group by a, b;
        !>  Warning: [Sem] GROUP BY can be replaced by DISTINCT.
3308        answer(t.a:int,t.b:int) ->
3309        {
3310        }
3311        Info: 0 tuples computed.
```

Figura 7.7: Error 22: `GROUP BY` puede ser reemplazado por `DISTINCT`.

```
3361        DES> %
3362        DES> create or replace table t(a int primary key, b int determined by a, c int determined by b);
3363        DES> select a from t order by a, b;
    !>  Warning: [Sem] Unnecessary ORDER BY term "b".
3364        answer(t.a:int) ->
3365        {
3366        }
3367        Info: 0 tuples computed.
```

Figura 7.8: Error 24: Término innecesario en `ORDER BY`.

Bibliografía

[1] Alireza Ahadi et al. «Students' semantic mistakes in writing seven different types of SQL queries». En: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 2016, págs. 272-277.

[2] Stefan Brass y Christian Goldberg. «Semantic errors in SQL queries: A quite complete list». En: *Journal of Systems and Software* 79.5 (2006), págs. 630-644.

[3] Francisco Bueno et al. «The Ciao prolog system». En: *Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM)* 95 (1997), pág. 96.

[4] Edgar F. Codd. «A Relational Model of Data for Large Shared Data Banks». En: *Communications of the ACM* 13.6 (1970), págs. 377-387.

[5] Edgar F. Codd. «Data Models in Database Management». En: *ACM SIGMOD Record* 11.2 (1972), págs. 112-114.

[6] Alain Colmerauer y Philippe Roussel. «The birth of Prolog». En: *History of programming languages—II*. 1996, págs. 331-367.

[7] *Datalog - Wikipedia — en.wikipedia.org*. `https://en.wikipedia.org/wiki/Datalog`. [Accessed 25-05-2025].

[8] Daniel Diaz, Philippe Codognet et al. «Design and implementation of the gnu prolog system». En: *Journal of Functional and Logic Programming* 6.2001 (2001), pág. 542.

[9] Ronald Fagin. «Horn clauses and database dependencies». En: *Journal of the ACM (JACM)* 29.4 (1982), págs. 952-985.

[10] Yuri Gurevich. «Datalog: A perspective and the potential». En: *International Datalog 2.0 Workshop*. Springer. 2012, págs. 9-20.

[11] Joxan Jaffar y J-L Lassez. «Constraint logic programming». En: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1987, págs. 111-119.

[12] *Lenguaje SQL, historia y conceptos básicos — universidadviu.com*. `https://www.universidadviu.com/es/actualidad/nuestros-expertos/lenguaje-sql-historia-y-conceptos-basicos`. [Accessed 25-05-2025].

[13] Victor Matos y Becky Grasser. «Sql-based discovery of exact and approximate functional dependencies». En: *ACM SIGCSE Bulletin* 36.4 (2004), págs. 58-63.

[14] John McCarthy. «History of LISP». En: *History of programming languages*. 1978, págs. 173-185.

[15] Jim Melton. «Sql language summary». En: *Acm Computing Surveys (CSUR)* 28.1 (1996), págs. 141-143.

[16] Daphne Miedema, George Fletcher y Efthimia Aivaloglou. «So many brackets! An analysis of how SQL learners (mis) manage complexity during query formulation». En: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 2022, págs. 122-132.

[17] Fernando Sáenz Pérez. «DESweb: una herramienta para el aprendizaje de SQL». En: *Actas de las JENUI* (2019), págs. 95-102.

[18] Seth Poulsen et al. «Insights from student solutions to sql homework problems». En: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 2020, págs. 404-410.

[19] *Prolog - Wikipedia, la enciclopedia libre — es.wikipedia.org*. `https://es.wikipedia.org/wiki/Prolog`. [Accessed 25-05-2025].

[20] Fernando Sáenz-Pérez. «ACIDE: an integrated development environment configurable for LaTeX». En: *The PracTeX Journal* 3.1 (2007), págs. 1-17.

[21] Fernando Sáenz-Pérez. «Applying constraint logic programming to SQL semantic analysis». En: *Theory and Practice of Logic Programming* 19.5-6 (2019), págs. 808-825.

[22] Fernando Sáenz-Pérez. «Outer joins in a deductive database system». En: *Electronic Notes in Theoretical Computer Science* 282 (2012), págs. 73-88.

[23] Fernando Sáenz-Pérez, Rafael Caballero y Yolanda García-Ruiz. «A deductive database with datalog and sql query languages». En: *Programming Languages and Systems: 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings 9*. Springer. 2011, págs. 66-73.

[24] *SICStus Prolog Homepage — sicstus.sics.se*. `https://sicstus.sics.se/`. [Accessed 25-05-2025].

[25] Abraham Silberschatz et al. *Fundamentos de bases de datos*. Vol. 11. McGraw-Hill Ciudad de México, México, 2002.

[26] Leon Sterling y Ehud Y Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[27] *swi-prolog.org*. `https://www.swi-prolog.org/`. [Accessed 20-05-2025].

[28] *TIOBE Index - TIOBE — tiobe.com*. `https://www.tiobe.com/tiobe-index/`. [Accessed 20-05-2025].

[29] Jeffrey D Ullman. «Bottom-up beats top-down for datalog». En: *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1989, págs. 140-149.

[30] Jan Wielemaker. «SWI-Prolog: history and focus for the future». En: *ALP Issue* 152 (2012).