LENGUAJES DE PROGRAMACIÓN

ANALIZADOR DE UN LENGUAJE DE CONSULTA PERSONALIZADO

Anderson Barrera
Santiago García
Daniel García
Manuel Castiblanco
Paula Carvajal

Universidad Nacional de Colombia
Marzo 2025

¿QUÉ SE IMPLEMENTÓ?

Analizador de lenguaje de consulta SQL-like

Objetivo:
Diseño e implementación de un analizador SQL-like en Haskell utilizando la biblioteca Parsec.

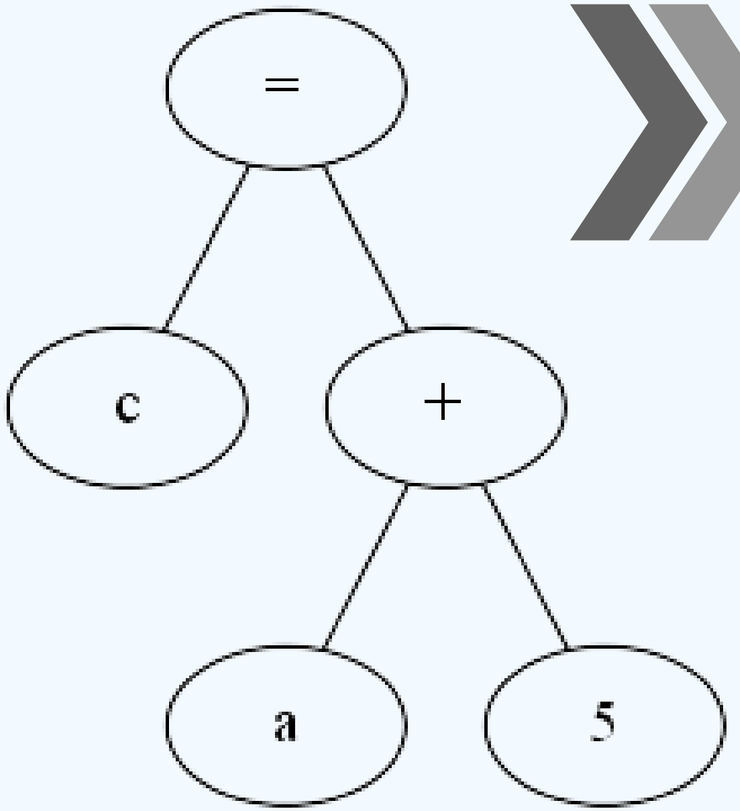
Funcionalidades:

- Reconocimiento de cláusulas SELECT, FROM, y WHERE.
- Soporte para operadores de comparación y expresiones lógicas (AND, OR).
- Exportación de consultas en formato DOT para visualización con Graphviz.

INTRODUCCIÓN

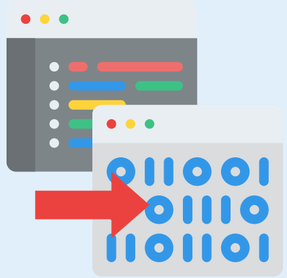
Haskell, AST y Parsec

Este proyecto desarrolla un analizador de un lenguaje de consulta personalizado utilizando **Haskell** y la biblioteca **Parsec**. Su objetivo es procesar consultas SQL-like y estructurarlas en un **Árbol de Sintaxis Abstracta (AST)**. Se detallan aspectos teóricos sobre compiladores y parsing, así como el diseño e implementación del analizador.



FUNDAMENTOS TEÓRICOS

Herramientas necesarias



Compiladores y Parsing

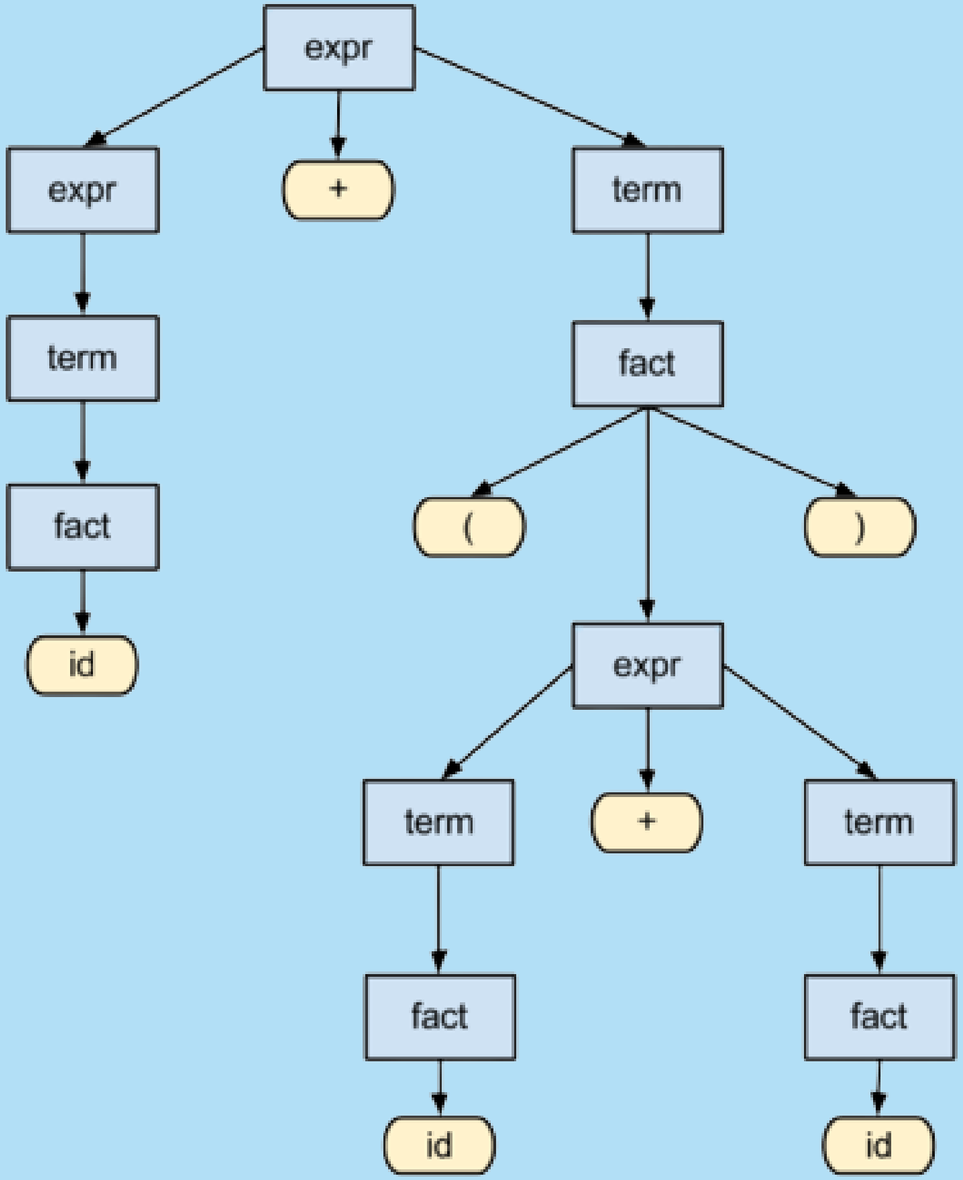
Programa que traduce código fuente de un lenguaje de alto nivel a un formato ejecutable en el computador. Dentro de este, el análisis sintáctico es un elemento fundamental.

Programación Funcional y Haskell

Haskell es un lenguaje de programación funcional que enfatiza la inmutabilidad y el uso de las funciones como elemento de primera clase (Bird. 2014)

Biblioteca Parsing

Parsec es una biblioteca de parsing basada en combinadores, lo que permite definir analizadores sintácticos de manera modular y declarativa



DISEÑO DEL ANALIZADOR

Árbol de sintaxis abstracta

Este diseño se basa en el código desarrollado, considerando la representación estructurada de consultas SQL-like mediante Haskell y Parsec. Se ha implementado un Árbol de Sintaxis Abstracta (AST) para representar las consultas y un conjunto de parsers para identificar los distintos componentes de una instrucción SQL.

DISEÑO DEL ANALIZADOR

El analizador desarrollado consta de los siguientes módulos:

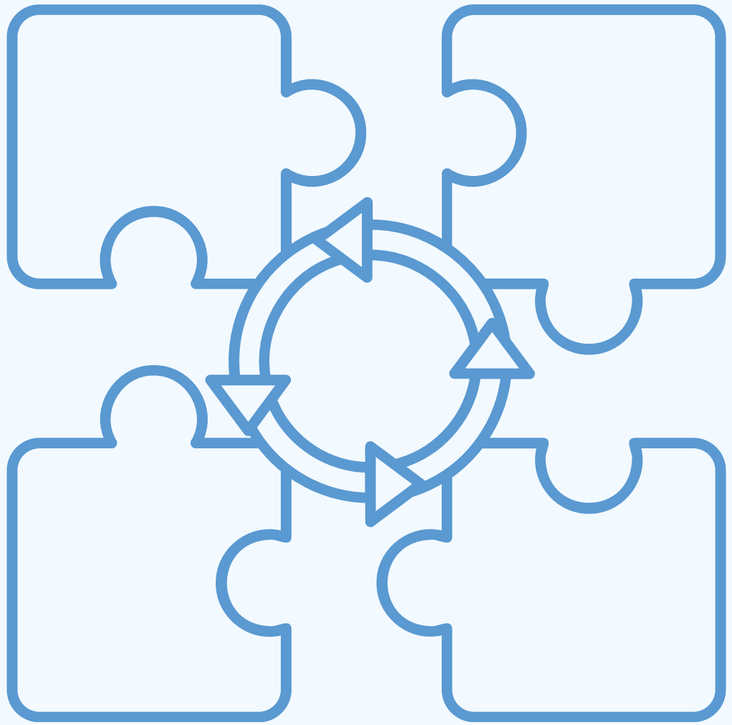
1. **Definición del AST**: Estructura de datos en Haskell para representar consultas.
2. **Parser de operadores**: Maneja operadores como **=**, **¿**, **¡**, **¿=**, **¡=**, y **!=**
3. **Parser de condiciones**: Se analiza la condición en la cláusula **WHERE**.
4. **Parser de expresiones**: Soporta operadores lógicos **AND** y **OR**.
5. **Parser de consultas**: Se analiza la estructura general de una consulta **SELECT**, **FROM** y **WHERE**.

Cada componente del analizador utiliza combinadores de Parsec, lo que permite modularidad y reutilización del código.

IMPLEMENTACIÓN DEL PARSER

Uso de combinadores de parsers

La implementación del parser se basa en el **uso de combinadores de parsers** proporcionados por la biblioteca Parsec en Haskell. Se han desarrollado diferentes funciones para **reconocer** y **procesar** los elementos de una consulta SQL-like.



IMPLEMENTACIÓN DEL PARSER

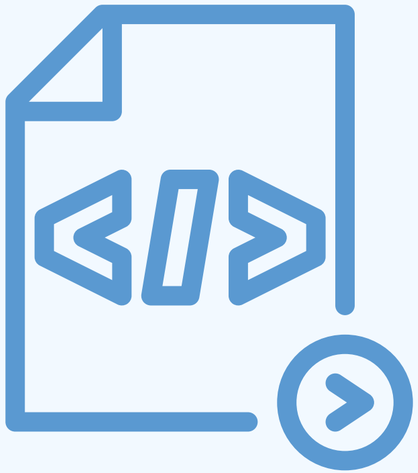
Los principales parsers implementados incluyen:

Parser de palabras clave:

Detecta y maneja palabras clave como SELECT, FROM y WHERE.

Parser de columnas:

Identifica y extrae los nombres de las columnas en una instrucción SELECT.



Parser de tablas:

Analiza el nombre de la tabla especificada en la cláusula FROM.

Parser de condiciones:

Procesa las condiciones presentes en WHERE, incluyendo operadores de comparación y expresiones lógicas.

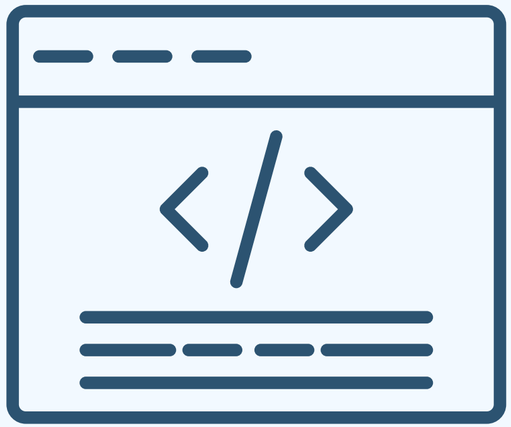
PRUEBAS Y EVALUACIÓN

Funcionalidad y precisón del analizador

Consultas simples sin WHERE.

Consultas con una sola condición en WHERE.

Consultas con múltiples condiciones usando AND y OR.



Uso de operadores de comparación como =, >, <, >=, <= y !=.

Consultas con nombres de tablas y columnas que contienen guiones bajos y números.

PRUEBAS Y EVALUACIÓN

```
Main> main
===== Prueba 1: Consulta sin WHERE =====
Consulta: SELECT nombre, edad FROM usuarios;
Esperado: Query {selectFields = ["nombre","edad"], fromTable = "usuarios", whereClause = Nothing}
---------------------------
===== Prueba 2: Consulta con WHERE =====
Consulta: SELECT nombre FROM usuarios WHERE edad > 30;
Esperado: Query {selectFields = ["nombre"], fromTable = "usuarios", whereClause = Just (Cond Condition {column = "edad", operator = Gt, value = "30"})}
---------------------------
===== Prueba 3: WHERE con AND y OR =====
Consulta: SELECT nombre FROM usuarios WHERE edad > 30 AND ciudad = 'Madrid' OR salario >= 5000;
Esperado: Query {selectFields = ["nombre"], fromTable = "usuarios", whereClause = Just (Or (And (Cond Condition {column = "edad", operator = Gt, value = "30"}) (Cond Condition {column = "ciudad", operator = Eq, value
= "Madrid"})) (Cond Condition {column = "salario", operator = Ge, value = "5000"}))}
---------------------------
===== Prueba 4.1: WHERE con operador = =====
Consulta: SELECT nombre FROM usuarios WHERE edad = 30;
Esperado: Query {selectFields = ["nombre"], fromTable = "usuarios", whereClause = Just (Cond Condition {column = "edad", operator = Eq, value = "30"})}
---------------------------
===== Prueba 4.2: WHERE con operador >= =====
Consulta: SELECT nombre FROM usuarios WHERE edad >= 30;
Esperado: Query {selectFields = ["nombre"], fromTable = "usuarios", whereClause = Just (Cond Condition {column = "edad", operator = Ge, value = "30"})}
---------------------------
===== Prueba 4.3: WHERE con operador <= =====
Consulta: SELECT nombre FROM usuarios WHERE edad <= 30;
Esperado: Query {selectFields = ["nombre"], fromTable = "usuarios", whereClause = Just (Cond Condition {column = "edad", operator = Le, value = "30"})}
---------------------------
===== Prueba 4.4: WHERE con operador != =====
Consulta: SELECT nombre FROM usuarios WHERE edad != 30;
Esperado: Query {selectFields = ["nombre"], fromTable = "usuarios", whereClause = Just (Cond Condition {column = "edad", operator = Ne, value = "30"})}
---------------------------
===== Prueba 4.5: WHERE con operador < =====
Consulta: SELECT nombre FROM usuarios WHERE edad < 30;
Esperado: Query {selectFields = ["nombre"], fromTable = "usuarios", whereClause = Just (Cond Condition {column = "edad", operator = Lt, value = "30"})}
---------------------------
===== Prueba 4.6: WHERE con operador > =====
Consulta: SELECT nombre FROM usuarios WHERE edad > 30;
Esperado: Query {selectFields = ["nombre"], fromTable = "usuarios", whereClause = Just (Cond Condition {column = "edad", operator = Gt, value = "30"})}
---------------------------
===== Prueba 6: WHERE con combinaciÃ³n de AND y OR =====
Consulta: SELECT nombre FROM empleados WHERE edad > 25 AND departamento = 'Ventas' OR salario > 5000 AND experiencia >= 5;
Esperado: Query {selectFields = ["nombre"], fromTable = "empleados", whereClause = Just (Or (And (Cond Condition {column = "edad", operator = Gt, value = "25"}) (Cond Condition {column = "departamento", operator =
Eq, value = "Ventas"})) (And (Cond Condition {column = "salario", operator = Gt, value = "5000"}) (Cond Condition {column = "experiencia", operator = Ge, value = "5"})))}
```

RESULTADOS OBTENIDOS

Los resultados muestran que el analizador es **capaz de interpretar correctamente las consultas SQL-like y generar el AST esperado**.

Además, se han realizado pruebas con entradas malformadas para evaluar la robustez del manejo de errores.

Optimización y Mejoras

A partir de los resultados de las pruebas, se identificaron algunos errores recurrentes que fueron corregidos para mejorar la precisión y eficiencia del parser.

01

Manejo de espacios en blanco

Error: Algunas consultas fallaban debido a espacios adicionales o faltantes entre los tokens clave.

02

Conexión segura

Error: El parser evaluaba AND y OR en el mismo nivel de precedencia, causando errores en consultas con lógica combinada.

03

Conexión fiable

Error: Los mensajes de error eran poco informativos al fallar la validación de una consulta.

04

Identificadores con caracteres especiales:

Error:No se reconocían correctamente identificadores que contenían guiones bajos o números.

Mejoras

01

Manejo de espacios en blanco

Solución: Se agregaron combinadores spaces de Parsec en los lugares adecuados para garantizar un reconocimiento flexible de espacios en blanco.

02

Conexión segura

Solución: Se implementó un nuevo parser que diferencia la precedencia de AND y OR, asegurando que AND se evalúe primero.

03

Conexión fiable

Solución: Se implementó try en los parsers más susceptibles de fallar y se mejoró el manejo de errores con mensajes detallados usando <?> en Parsec.

04

Identificadores con caracteres especiales:

Solución: Se modificó el parser de identificadores para aceptar letter <|> digit <|> char '_', permitiendo nombres de tablas y columnas más flexibles.

LIMITACIONES DEL PROGRAMA

El presente trabajo desarrolla un parser en Haskell utilizando Parsec para interpretar y transformar consultas SQL-like en estructuras de datos formales. Sin embargo, el modelo actual presenta diversas **limitaciones** que restringen su aplicabilidad en escenarios de análisis más complejos. A continuación, se detallan las principales deficiencias identificadas en la implementación actual:

LIMITACIONES

01

Restricción en los nombres de columnas y tablas: El parser solo permite nombres conformados por caracteres alfabéticos,

02

Ausencia de soporte para valores de texto entre comillas: Actualmente, el parser no admite valores entre comillas en la cláusula WHER

03

Falta de operadores avanzados: La implementación solo reconoce operadores básicos de comparación (=, >, <, >=, <=, !=).

04

Evaluación incorrecta de la precedencia entre AND y OR: El parser no respeta la jerarquía de operadores lógicos en WHERE.

05

Falta de reconocimiento de alias en SELECT: El parser no soporta el uso de alias mediante la cláusula AS

06

Ausencia de soporte para ORDER BY, GROUP BY y LIMIT: No se han implementado reglas para procesar estas cláusulas esenciales en SQL

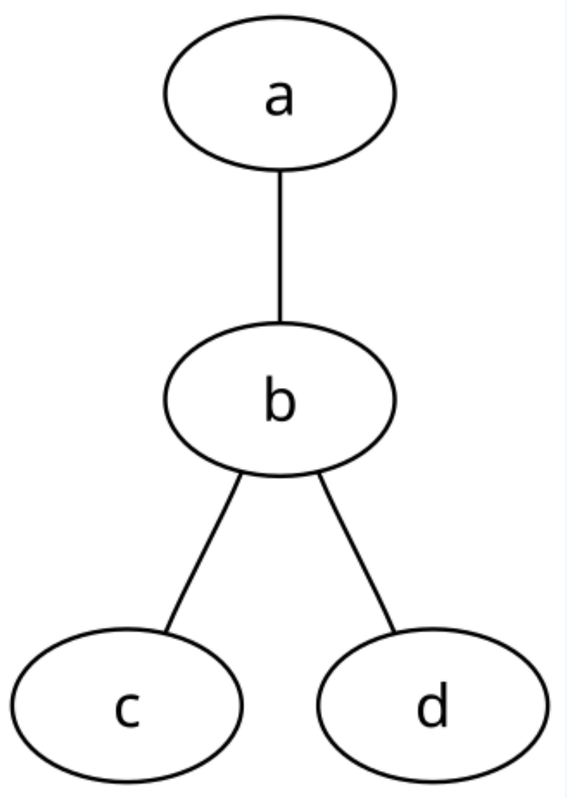
Si bien la implementación desarrollada proporciona una **base funcional** para el análisis estructural de consultas SQL-like, estas limitaciones **restringen** su aplicabilidad en escenarios más avanzados. Mejoras como el soporte para identificadores más flexibles, valores textuales, operadores avanzados y estructuras de consulta adicionales **aumentarían** la precisión y alineación con el estándar SQL

CONVERSIÓN DE CONSULTAS SQL A FORMATO DOT

Representación gráfica de la estructura de una consulta

La **conversión** de consultas SQL al formato **DOT** permite representar gráficamente la estructura de una consulta mediante nodos y relaciones dirigidas. Este proceso **facilita** la visualización de la estructura lógica de una consulta, especialmente en términos de las relaciones entre **SELECT**, **FROM** y las condiciones establecidas en **WHERE**.

```
 // el nombre del grafo y el punto y coma al
final de la línea son opcionales
 graph nombre_del_grafo {
     a -- b -- c;
     b -- d;
 }
```



Estructura base del grafo

La representación en formato DOT comienza con la **definición** de un grafo dirigido (digraph). Se establece un nodo principal Query, que representa la consulta en general. A este nodo se le **conecta** un nodo Table, que representa la tabla especificada en la cláusula **FROM**. La estructura base del grafo se genera mediante las siguientes líneas de código:

```
digraph Query {
  node [shape=box];
  Query -> Table [label="FROM"];
  Table [label="nombre_de_tabla"];
}
```

Esto permite establecer el punto de inicio de la representación gráfica de la consulta

Representación de la Cláusula SELECT

Cada columna seleccionada en la consulta SQL se conecta al nodo **Query**, indicando los campos que forman parte del resultado de la consulta. Esta **conexión** se construye **iterando** sobre los campos especificados en la consulta:

```
Query -> campo1 [label="SELECT"];
Query -> campo2 [label="SELECT"];
```

Cada campo se **representa** como un nodo individual conectado a **Query**.

Conversión de la Cláusula WHERE

Si la consulta incluye una cláusula **WHERE**, se agrega un nodo **Where** conectado a **Query**. Este nodo representa las condiciones de filtrado de la consulta. La conversión se maneja con la siguiente estructura:

```
Query -> Where [label="WHERE"];
Where -> Condicion1;
Where -> Condicion2;
```

De esta manera, si **WHERE** está presente, se añade un nodo específico para indicar su existencia.

Representación de Condiciones (AND, OR)

La **conversión** de las condiciones lógicas dentro de WHERE se maneja mediante una **estructura recursiva** que genera nodos para cada condición. Si la condición es **simple**, se representa como un nodo con el nombre de la columna, el operador de comparación y el valor correspondiente:

```
Where -> Cond_campo [label="campo operador valor"];
```

Si la condición involucra operadores lógicos AND o OR, se genera un nodo intermedio de tipo elíptico que representa la conexión entre las condiciones:

```
Where -> Where_AND [label="AND"];
Where_AND [shape=ellipse];
Where_AND -> Cond1;
Where_AND -> Cond2;
```

De manera similar, OR sigue la misma estructura con un nodo Where_OR.

GENERACIÓN DEL ARCHIVO DOT

Finalmente, la estructura generada se **concatena** en una **única cadena de texto** y se puede guardar en un archivo .dot para su posterior visualización con herramientas como **Graphviz**. Esta representación permite **interpretar** rápidamente la estructura de la consulta SQL en una forma gráfica **clara y organizada**. En términos generales, el proceso de conversión de **SQL** a **DOT** sigue una estructura bien definida:

- Se crea un nodo Query conectado a la tabla de FROM.
- Cada campo de SELECT se enlaza a Query.
- Si existe WHERE, se genera un nodo Where conectado a Query.
- Las condiciones AND y OR se manejan con nodos intermedios en forma de elipse.
- El resultado final se exporta a DOT para su visualización gráfica

Esta conversión **facilita el análisis y optimización de consultas**, permitiendo comprender de manera visual su estructura y relaciones internas.

CONCLUSIONES

Desarrollo de un analizador para un lenguaje de consulta SQL-like utilizando Haskell y la biblioteca Parsec

01

Implementación de un parser modular

Flexible y utilizando combinadores de Parsec, facilitando su mantenimiento y ampliación.

02

Definición de un Árbol de Sintaxis Abstracta (AST)

Definición de un Árbol de Sintaxis Abstracta (AST) que representa de manera clara y organizada las consultas SQL-like.

03

Optimización del manejo de espacios en blanco

Asegura la correcta interpretación de consultas con distintos formatos.

04

Mejora en la gestión de errores

Mediante mensajes descriptivos que simplifican la depuración y corrección de consultas mal formadas.

TRABAJOS FUTUROS

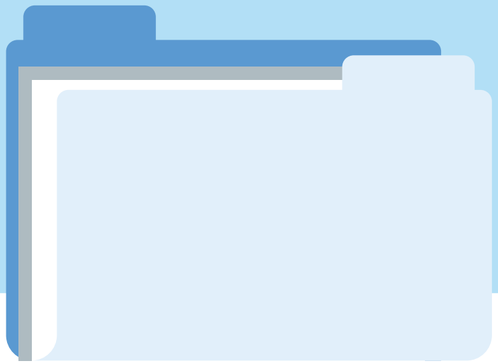
Oportunidades de mejora y expansión

El analizador desarrollado, aunque funcional, ofrece oportunidades de mejora y expansión, como: **extensión de operadores** (soporte para `LIKE`, `IN`, `BETWEEN`), **soporte para funciones agregadas** (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`), **optimización del rendimiento** para reducir la complejidad computacional, i**ntegración con una interfaz gráfica o API REST** para facilitar su uso, e **implementación de pruebas automatizadas** para garantizar estabilidad.

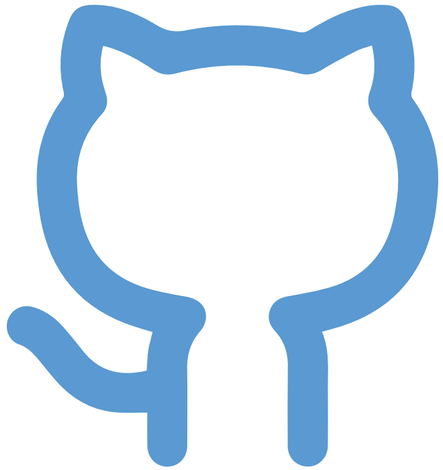
Este trabajo sienta las bases para sistemas más complejos de procesamiento de consultas, permitiendo adaptaciones según las necesidades del usuario o del sistema.

BIBLIOGRAFÍA

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools. Addison-Wesley.
- Appel, A. W. (1998). Modern Compiler Implementation in ML. Cambridge University Press. 5
- Bird, R. (2014). Thinking Functionally with Haskell. Cambridge University Press.
- Leijen, D., & Meijer, E. (2001). Parsec: Direct style monadic parser combinators for the real world. Utrecht University



REPOSITORIO



Link de acceso

Muchas gracias

por su atención