POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Biblioteca de representación de SQL en Erlang y Elixir

Autor: Miguel Emilio Ruiz Nieto

Director: Ángel Herranz Nieva

MADRID, JULIO DE 2018

No se vive celebrando victorias, sino superando derrotas.

Ernesto "Che" Guevara

A papá, que me enseñó a no bajar los brazos
A mamá, que me enseñó a confiar en mi intuición
A Josemi, por aparecer en los momentos oportunos
A mis amigos Nacho, Álvaro y Fernando, que hemos hecho juntos este camino
A mi amigo Nacho, que hemos estado juntos de principio a fin
A Cebra, Tom, Luna, Gigante, Atila y Bucky: mis amigos más fieles
Y en especial, a mi amigo y mentor Ángel Herranz.

Índice general

Resumen V

Abstract VII

1. INTRODUCCIÓN 1

1.1. Erlang: introducción y características del lenguaje . . . 3

1.1.1. Tipos básicos . . . 3

1.1.2. Pattern Matching . . . 5

1.1.3. Estructuras de datos: record . . . 6

1.1.4. Dialyzer . . . 6

1.2. EPGSQL: breve introducción y uso . . . 7

1.2.1. Conexión . . . 7

1.2.2. Simple Query y Extended Query . . . 8

2. DESARROLLO 9

2.1. Motivación . . . 9

2.2. Abstract Syntax Tree . . . 11

2.3. Serialización de las consultas . . . 11

2.3.1. Transacciones . . . 13

2.3.2. Operaciones CRUD . . . 14

2.3.3. Sentencias exclusivas de PostgreSQL 18

2.3.4. Otros símbolos representados 20

2.4. Funcionalidades sin añadir 25

2.4.1. UPDATE FROM 25

2.5. Dificultades encontradas y soluciones propuestas 26

2.5.1. Los tipos `name()`, `id()` e `identifier_chain()` 26

2.6. Diseño 26

2.7. Trabajos futuros 27

3. CONCLUSIONES **29**

Bibliografía **31**

Índice de figuras

1.1. Arquitectura Cliente/Servidor 1

2.1. Ejemplos de `JOIN` 24

Resumen

Hoy en día, las organizaciones están adaptando sus infraestructuras para ofrecer sus servicios a través de Internet. A pesar de que Erlang no es un lenguaje popular como los lenguajes orientados a objetos como Java o C++, proporciona tolerancia a fallos y escalabilidad para sistemas en tiempo real. Erlang Embedded SQL (EESQL) es una biblioteca para escribir consultas SQL por medio de estructuras de datos de Erlang.

En la primera parte del documento se hará una breve introducción al lenguaje Erlang, donde se verán los aspectos más relevantes para poder entender posteriormente el funcionamiento de la biblioteca.

Finalmente, en la segunda parte se presentará EESQL y la manera de representar cada una de las consultas SQL disponibles en esta versión de la biblioteca, y se discutirán algunos problemas encontrados durante el desarrollo y sus soluciones.

Abstract

Nowadays, the organizations are adapting their infrastructure in order to provide their services through the Internet. Despite of the fact Erlang is not a popular language like Object-Oriented languages such as Java or C++, it supplies fault tolerance and scalability for real-time systems. Erlang Embedded SQL (EESQL) is a library for writing SQL queries using Erlang data structures.

In the first part of the document we will introduce the Erlang language, particularly the most relevant aspects in order to understand the functioning of the library afterwards.

Finally, in the second part, we will introduce EESQL and the way how to represent each one of the available SQL queries in this version of the library, and discuss some found problems during the development and their solutions.

1

INTRODUCCIÓN

Actualmente, los procesos tradicionalmente hechos en papel se están quedando obsoletos debido a la transformación digital. Ello requiere que las organizaciones cuenten con la infraestructura necesaria para que pueda darse dicho cambio. Si bien las aplicaciones de escritorio han sido las principales protagonistas en los últimos años, hoy en día están en auge las aplicaciones web, ya que son independientes del sistema operativo en las que se ejecuten y no requieren que el usuario necesite instalar algún tipo de software en su máquina para después poder utilizarlas, entre otras razones.

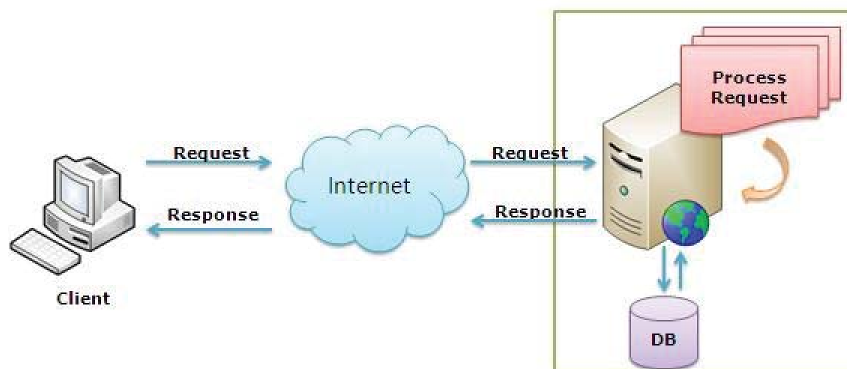


Figura 1.1: Arquitectura Cliente/Servidor

1 INTRODUCCIÓN

El esquema de una aplicación web no difiere mucho según el servicio que se desee cubrir: se requiere de uno o varios servidores web que atiendan las peticiones de los posibles clientes (dispositivos móviles, ordenadores u otros servidores) que lleguen y una base de datos de la cual realizar operaciones de lectura y escritura sobre los datos que se manejen. También cabe la posibilidad de que el servidor sea cliente de otros servidores para poder cubrir las funcionalidades que ofrece.

Haciendo uso del protocolo HTTP, las peticiones serán enviadas al servidor, y este será el encargado de dar una respuesta en base a cada petición, en el formato que se haya indicado o de la manera en la que se maneje la información. Si está en su mano, el servidor realizará una consulta a la base de datos en el tipo de lenguaje que la base de datos maneje (SQL o NoSQL). Con la información que le devuelva dicha query, el servidor le dará forma de manera que sea información útil para el usuario, bien sea en un fichero JSON o generando un fichero PDF.

```
$ curl http://api.myapi.com/v1/stats?sort=name&modifiedBefore=2018-04-12
```

```
SELECT * FROM stats WHERE sort = name AND '2018-04-12' < modified
```

Listado 1.1: Llamada a API y query que ejecuta

Este trabajo está ubicado en cómo construir las peticiones de la base de datos que posteriormente se enviarán a la misma en forma de queries SQL, y forma parte de una de las bibliotecas privadas de la startup inglesa Coowry Ltd[1].

Los sistemas de Coowry están desarrollados con lenguajes de programación funcional, tales como Erlang o Elixir, y esta biblioteca, llamada EESQL (Erlang Embedded SQL), nace de la necesidad de representar de una manera atractiva y cómoda queries SQL para dar servicio a una API implementada en Erlang. El uso de esta biblioteca está complementado con la utilización de la biblioteca `epgsql`[2], que sirve para crear un driver con una base de datos PostgreSQL y lanzar queries a la misma.

Este capítulo tiene la intención de dar al lector una noción sobre el lenguaje Erlang desde el punto de vista de la programación funcional secuencial, sin entrar en la potencia de la programación concurrente dado que no es necesario para poder entender más adelante el funcionamiento de EESQL.

[1] https://www.coowry.com
[2] https://github.com/epgsql/epgsql

1.1 Erlang: introducción y características del lenguaje

Erlang es un lenguaje funcional que fue creado a mediados de los años 80 en los laboratorios Ericsson por Joe Armstrong y Robert Virding, fruto de la necesidad de encontrar un lenguaje que fuera adecuado para los dispositivos de telecomunicación que aparecieron en esa época. Aunque sus primeras versiones estaban escritas en el lenguaje Prolog, está inspirado en lenguajes funcionales como ML, lenguajes concurrentes como ADA y del propio lenguaje Prolog [1].

A diferencia de otros lenguajes funcionales como OCaml o Haskell, cuya característica más llamativa es que son fuertemente tipados, Erlang carece de un sistema de tipos estático. No obstante, la ligereza de sus procesos y su modo de comunicación entre los mismos mediante paso de mensajes lo convierte en un lenguaje atractivo para aplicaciones sistemas de tiempo real, distribuidos, con alta concurrencia y con tolerancia a fallos.

Esta sección pretende dar una visión de los aspectos más importantes del lenguaje desde el punto de vista de programación funcional para poder entender el modo de funcionamiento de la biblioteca EPGSQL y, en el próximo capítulo, el de EESQL.

1.1.1 Tipos básicos

Aunque Erlang no tenga un sistema de tipos que se pueda verificar en tiempo de compilación, es importante conocer la sintaxis del lenguaje para entender con ejemplos sencillos la capacidad del mismo y cómo el intérprete ejecuta las líneas de código[3].

Átomos

Los átomos son un tipo de dato que por su propio nombre ya tienen suficiente expresividad, y se representan por una letra minúscula seguido de cualquier carácter, o por cualquier carácter siempre que esté envuelto entre comillas simples, tal y como se muestra en los ejemplos [2]:

```
1> ok.
2> some_Atom.
3> 'Miguel␣Emilio'.
```

[3]Nótese que los ejemplos descritos son como si estuvieran ejecutados en un intérprete de Erlang y cada línea de código escrita termina en punto (.), salvo que se indique lo contrario.

Listado 1.2: Ejemplos de átomos

Erlang no tiene tipos booleanos como tal: los átomos `true` y `false` sirven como booleanos en los valores de retorno de las funciones de comparación como los operadores binarios.

Tuplas

Las tuplas son una estructura de datos que se utilizan cuando se sabe con certeza el número de elementos que se van a manejar. Las tuplas son de la forma `{E1, E2,...,EN}` donde `EX` puede ser de cualquier tipo, incluso una tupla. Ejemplos de tuplas serían:

```
1> {abc, def}.
2> {123, 456}.
3> {dog, cat, fish}.
4> {person, "Miguel", "Ruiz"}.
```

Listado 1.3: Ejemplos de tuplas

Una tupla cuyo primer elemento es un átomo se denomina *tupla etiquetada* (“tagged tuple”). Podría darse el caso de declarar una tupla que contuviera un solo elemento, sin embargo, no se considera una buena praxis.

Listas

Las listas son un elemento muy potente de los lenguajes funcionales y con ellas se pueden realizar buena parte de las funciones sin necesidad de recurrir a estructuras de datos más complejas.

Una particularidad de Erlang con respecto a otros lenguajes funcionales es que al no tener tipos estáticos, las listas pueden contener cualquier elemento, tal y como se muestra en el ejemplo:

```
1> L = [1, dos, <<"3">>, {number, "4"}].
```

Listado 1.4: Ejemplo de lista

Toda lista se compone de dos elementos: la cabeza o inicio de la lista (*head*) y la cola (*tail*), que es el resto de la lista. La representación de una lista de números del 1 al 5 podría darse de cualquiera de las siguientes formas:

```
2> Numbers = [1, 2, 3, 4, 5].
[1 | [2, 3, 4, 5]]
[1 | [2 | [3, 4, 5]]]
[1 | [2 | [3 | [4, 5]]]]
[1 | [2 | [3 | [4 | [5]]]]]
[1 | [2 | [3 | [4 | [5 | []]]]]]
```

Listado 1.5: Varias formas de declarar una lista

El carácter | es el delimitador entre la cabeza de la lista y su cola. La cola de una lista con un solo elemento es la lista vacía (`[]`).

Una característica de Erlang con respecto a las cadenas de caracteres (Strings de Java) y a los caracteres es que internamente son listas de números y números, respectivamente. Si escribimos los códigos ASCII de las letras "A", "B" y "C" en el shell saldrá por pantalla la cadena "ABC":

```
3> [65, 66, 67].
"ABC"
```

Listado 1.6: Declaración de una cadena de caracteres

Erlang permite representar estas listas de valores numéricos como valores ASCII envueltos por comillas dobles [1][3].

1.1.2 Pattern Matching

El pattern matching es la manera en los lenguajes declarativos (lógicos y funcionales) de representar la asignación de variables, además de el flujo de funcionamiento de las funciones y de extraer valores de los datos [1].

El esquema que sigue es el de `Pattern = Expression`, donde `Pattern` es el nombre de la variable o la estructura de datos que se quiere "encajar", y `Expression` son las estructuras de datos, variables ligadas o llamadas a funciones que se desean ligar. Existen dos tipos de variables: las ligadas (*bound*), que ya poseen un valor concreto e inmutable, y las no ligadas (*unbound*), que aun no tienen un valor fijado.

Ejemplos del uso del pattern matching sería el siguiente:

```
1> Integer = 1.
2> List = [1,2,3,4].
3> [Head | Tail] = List.
4> {ok, {person, Name, Surname}} = module:person_data(1).
```

Listado 1.7: Ejemplos de uso del pattern matching

Con respecto al uso del pattern matching en las funciones, el mecanismo es el siguiente: Se va analizando cada rama de la función verificando si el input "encaja" con alguna de ellas. En caso de hacerlo, se ejecuta la función por la rama por la cual haya encajado. Esto también puede ir acompañado de otras cláusulas como son las guardas y la sentencia **`case .. of`**, pero no serán objeto de estudio en este trabajo.

1.1.3 Estructuras de datos: record

Como ya se ha mencionado, las tuplas son estructuras de datos que se utilizan cuando se conoce el número de datos que se quiere utilizar. Cuando dicho número se considera grande, lo mejor es usar `record`, que no es más que una tupla con nombre. La forma de crear un record es de la siguiente manera [4].

```
-record(Name, {
  % Los campos key1 y key2 tienen
  % valores por defecto
  key1 = Default1,
  key2 = Default2,
  % key3 tiene undefined como valor
  % por defecto
  key3
}).
```

Listado 1.8: Declaración de un record

Los campos del record pueden tener valores por omisión si se desea que los tenga, y si no contendrán el átomo **`undefined`** si no se opera con dicho campo.

1.1.4 Dialyzer

Dialyzer es una herramienta incluida en el estándar de Erlang/OTP que sirve para realizar un análisis estático de tipos. Con ella, el código es menos propenso a dar errores en tipo de ejecución. Dialyzer permite declarar tipos propios que el desarrollador quiera definir, con el siguiente esquema:

```
-type TypeName() :: TypeDefinition.
```

Listado 1.9: Declaración de un type

También permite declarar la especificación de las funciones con el esquema

```
-spec FunctionName(ArgumentTypes) -> ReturnTypes.
```

Listado 1.10: Declaración de un spec

Es importante destacar que Erlang ofrece tipos preconstruidos de los tipos básicos que maneja el propio lenguaje y que el desarrollador tiene al alcance [3].

1.2 EPGSQL: breve introducción y uso

Para poder entender la utilidad de EESQL, previamente se explicará muy brevemente algunas de las funcionalidades que EPGSQL ofrece.[4]

1.2.1 Conexión

A la hora de ejecutar una query, es necesario establecer previamente una conexión con la base de datos que sea persistente. EPGSQL ofrece una función para establecer una conexión con la base de datos deseada:

```
{ok, Connection} = epgsql:connect("localhost", "coowry",
"password", [
  {database, "my_database"},
  {timeout, 6000}
]).
```

Listado 1.11: Establecimiento de conexión con una Base de datos

La función consta de cuatro parámetros: la dirección IP de la máquina donde se encuentre la base de datos, el usuario y contraseña con el que se pretende establecer la conexión y, por último, una lista de parámetros adicionales, tales como el nombre de la base de datos o un timeout por si la conexión quedara inutilizada. Devuelve una tupla con la conexión o, en caso de fallo, una tupla con el error que se ha producido.

[4] Los ejemplos descritos en esta sección están inspirados en los que aparecen en el repositorio de Github `https://github.com/epgsql/epgsql`.

1.2.2 Simple Query y Extended Query

EPGSQL ofrece diferentes funciones para ejecutar las queries. La más sencilla es la función `squery`:

```
epgsql:squery(Connection,
"SELECT name FROM users WHERE id = 1").
```

Listado 1.12: Ejecución de una query con la función `squery`

Esta función recibe como argumentos la conexión que se ha obtenido anteriormente y la query que se desea ejecutar. El inconveniente está en que no permite hacer un enlazado de los parámetros que pueda recibir la query. Por otro lado, la función `equery` sí que permite hacer un enlazado de parámetros, ya que tiene un argumento más que `squery` que es una lista de los parámetros que se desean enlazar:

```
epgsql:equery(Connection,
"SELECT name FROM users WHERE id = $1", [1]).
```

Listado 1.13: Ejecución de una query con la función `equery`

Esta función es fundamental para entender EESQL, ya que la función principal de esta devuelve una tupla del tipo `{Query, [Parameters]}`.

2

DESARROLLO

En el Capítulo 1 se ha presentado el lenguaje Erlang y se ha visto cómo EPGSQL realiza consultas a la base de datos correspondiente y lo necesario para ello. Este capítulo abordará el funcionamiento de EESQL y la motivación que hay detrás de esta biblioteca.

2.1 Motivación

Como ya se ha visto, cuando se produce una llamada a un servicio web y ésta requiere de hacer una consulta a la base de datos, se deberá pasar los parámetros para dicha consulta, en el caso de los hubiera. Partiendo de la consulta del ejemplo de la introducción:

```
-- Si no está definido from:
SELECT * FROM stats WHERE sort = name;

-- Si está definido from:
SELECT * FROM stats WHERE sort = name AND from < modified;
```

Listado 2.1: Ejemplo de query SQL

Podría pensarse que la mejor forma de construir la consulta a ejecutar sea de la siguiente forma:

```
Query = "SELECT * FROM stats " ++
        "WHERE sort " ++ "=" ++ Name ++
        (if From /= undefined -> " AND " ++ From ++ " < modified";
            true -> "" end).
```

Listado 2.2: Construcción de una query

Habría que convertir los parámetros a listas de caracteres y concatenarlas. Sin embargo, esto puede ocasionar errores por parte del desarrollador. Además, el operador `++` es computacionalmente costoso, y las variables `Name` y `From` pueden recibir inyecciones de código SQL, que hagan que ejecutar esa query sea un ataque muy comprometido para la compañía.

En la documentación de la especificación de los tipos de Erlang [3] hay un tipo denominado `iodata()` que está representado de la siguiente forma:

```
-spec iodata() :: iolist() | binary().
-spec iolist() :: maybe_improper_list(byte() | binary() | iolist(),
                                      binary() | []).
```

Listado 2.3: Especificación del tipo `iodata()`

El tipo `iodata()` es precisamente el que se utiliza en EPGSQL para representar la query que se desea ejecutar. Este tipo permite devolver una lista que contenga los elementos que se detallan en su especificación, de manera que es computacionalmente menos costoso que el operador `++`, más fácil de componer para el desarrollador, y potencialmente se estaría evitando un ataque. Por tanto, un primer requisito para EESQL es que devuelva un valor de tipo `iodata()` para representar la query. Por otro lado, se ha visto que para enlazar los parámetros de cada consulta hay que pasarlos en una lista, y ésta puede contener cualquier tipo de dato, luego EESQL también debe devolver una lista con los parámetros que se deban enlazar en la consulta.

Así pues, una llamada a EESQL ha de devolver una tupla `{Query, Parameters}`, donde `Query` es de tipo `iodata()` y `Parameters` es una lista de `literal()`, es decir, de los parámetros de la `Query`, que podría estar vacía en caso de que no los hubiera. El tipo de dato que contiene dicha lista se verá con más detenimiento en su sección correspondiente.

A continuación se muestra un fragmento de código que hace uso de EESQL. Puede parecer poco intuitivo para el lector que no está muy familiarizado con el lenguaje, pero se puede apreciar que, a diferencia del ejemplo del listado 2.2, se hace uso del pattern matching para saber si se ha pasado un segundo argumento en la llamada a la API y, a continuación, se invoca a la función `to_sql` del módulo eesql.

```
...
```

```
WhereClause = case From of
                  undefined -> {sort, '=', Name};
                  Date -> {'and', [{sort, '=', Name},
                                   {Date, '<', modified}]}
              end,
{Query, Parameters} = eesql:to_sql(#select{from = [stats], where = WhereClause}),
  ...
```

Listado 2.4: Llamada a la función principal de EESQL

2.2 Abstract Syntax Tree

Los compiladores son programas que analizan otros programas escritos en el mismo lenguaje que el compilador o en otro diferente, y generan un código intermedio (como el caso de la máquina virtual de Erlang) o un código máquina (como los lenguajes ensamblador) que la máquina que lo va a ejecutar pueda comprender. Esta sección no pretende dar una explicación exhaustiva de los compiladores, pero es necesario tener una ligera noción para hablar del **Abstract Syntax Tree**.

El Abstract Syntax Tree (AST) o Árbol Sintáctico Abstracto, es una estructura de datos generada por el analizador sintáctico del compilador para verificar que los tokens identificados en la fase de análisis léxico son correctos con respecto a la gramática del lenguaje. Esta estructura es fundamental para entender cómo funciona EESQL, ya que se utiliza la gramática de SQL para poder representar las consultas.

2.3 Serialización de las consultas

El propósito de esta sección es el de explicar de manera detallada el funcionamiento de EESQL, dado que se entiende que el lector ahora tiene (si no la tenía ya a priori) una noción básica de Erlang y de la programación funcional.

La función principal de EESQL, cuyo nombre es `to_sql`, recibe un argumento del tipo `sql_stmt()` y devuelve una tupla `{Query, Parameters}` que, como se ha mencionado anteriormente, son de tipo `iodata()` y una lista de tipo `literal()`, respectivamente. El tipo `sql_stmt()` es un tipo que se ha creado mediante Dialyzer, y engloba otros tipos que sirven para representar las queries que se desean ejecutar. La representación se logra gracias a la gramática de contexto libre [5] de la que se hace uso para asegurar que la sintaxis de la query es correcta.

```
-type sql_stmt() ::
```

```
      commit_stmt()
    | start_trans_stmt()
    | rollback_stmt()
    | query_spec()
    | insert_stmt()
    | update_stmt()
    | delete_stmt()
    | truncate_stmt()
    | pg_refresh_stmt()
    | union_stmt()
    | pg_with_as().
```

Listado 2.5: Definición del tipo `sql_stmt()`

Dado que los subtipos de los que está compuesto `sql_stmt()` son variados, la función `to_sql/1` se apoya en una función auxiliar llamada `to_sql/2`, que aplica el pattern matching sobre los subtipos para saber cuál es la query que tiene que representar. Es importante destacar que los símbolos no terminales de la gramática son tipos declarados con Dialyzer, y que los símbolos terminales se han representado con los tipos básicos que se han visto en la sección 1.1 del Capítulo 1, además de los que están definidos en el estándar de Erlang/OTP, tal y como se define en la especificación de los tipos de Erlang [3].

Cabe señalar que la función `to_sql/2` se apoya en otras funciones auxiliares, que actúan como funciones de casting de tipos para representar los parámetros o las condiciones booleanas que se puedan encontrar en la query.

En lo que respecta a la función `to_sql/2`, el primer parámetro que recibe es un número entero positivo, que va incrementando en las sucesivas llamadas recursivas que se pueda hacer a esta función, e indica el nivel que ocupa en la representación del AST para aquella parte de la query que se está representando. El segundo parámetro que recibe es una tupla etiquetada con la propia sentencia SQL que se desea representar por parte de la función principal.

```
-spec to_sql(Pos,
               {sql_stmt, sql_stmt()}
             | {where_clause, undefined | predicate()}
             | {predicate, predicate()}
             | {value_expr, value_expr()}
             | {value_expr_list, [value_expr()]}
             | {table_ref, table_ref()}
             | {table_primary, table_primary()}
             | {literal, literal()}
             | {offset, undefined | {pos_integer(), pos_integer()}}
             | {on_conflict_update_target,
```

```
     undefined |[column_reference()], [column_reference()]})
-> {Pos, {Equery, Params}}
when Pos :: pos_integer(),
     Equery :: iodata(),
     Params :: [literal()].
```

Listado 2.6: Especificación de la función `to_sql/2`

A continuación se va a explicar cómo se serializan cada una de las queries que en la actual versión de la biblioteca se pueden representar, además de ciertos símbolos no terminales de la gramática que aportan valor a algunas de las queries, así como la utilidad que tienen las mismas.

2.3.1 Transacciones

Las transacciones son bloques de código SQL que se utilizan para realizar operaciones que potencialmente pueden dar un error en tiempo de ejecución, de manera que permiten volver a un estado en el cual se sabe que es consistente antes de haber ejecutado dichas operaciones.

Begin Transaction

Para ejecutar una transacción, lo primero que hay que hacer es expresar qué sentencias componen dicha transacción. La sentencia encargada de ello se conoce como `BEGIN TRANSACTION`, y en EESQL se representa con el tipo `start_trans_stmt()`, que no es más que el átomo `start_transaction`.

Commit Statement

La sentencia `COMMIT` finaliza la transacción de datos en una base de datos relacional, haciendo que los cambios producidos sean visibles para el resto de los usuarios y garantizando que no se van a perder. En EESQL hay tres formas de declarar esta sentencia:

```
-type commit_stmt() :: commit | commit_and_chain
                              | commit_and_no_chain.
```

Listado 2.7: Especificación del tipo `commit_stmt()`

`CHAIN` significa que una nueva transacción comienza con el mismo nivel de la que se acaba de cerrar.

Rollback Statement

`ROLLBACK` devuelve a la base de datos a un estado anterior y todos los cambios hechos durante una transacción quedan descartados. El tipo que se ha definido para representar esta sentencia en EESQL es `rollback_stmt()`, cuya definición es el átomo `rollback`.

2.3.2 Operaciones CRUD

Las operaciones CRUD (Create, Read, Update, Delete) corresponden a las sentencias `INSERT`, `SELECT`, `UPDATE` y `DELETE`, respectivamente. Estas cuatro sentencias se han definido en EESQL mediante sus propios tipos, partiendo de la gramática de referencia [5], y cada una de ellas corresponde a un record de Erlang que se detallarán en los siguientes apartados.

Query Specification

Tal y como se define en la gramática, este símbolo no terminal especifica una tabla como resultado de una operación sobre una tabla o conjunto de tablas y bajo unas condiciones dadas. Por supuesto, este símbolo define la sentencia `SELECT`. En la gramática, el símbolo no terminal `<query specification>` está expresado de la siguiente forma:

```
<query specification> ::= SELECT [<set quantifier>] <select list> <table expression>
<select list> ::= <asterisk> | <select sublist> [{<comma> <select sublist>}...]
<select sublist> ::= <derived column> | <qualified asterisk>
<qualified asterisk> ::= <asterisked identifier chain> <period> <asterisk>
                       | <all fields reference>
<asterisked identifier chain> ::= <asterisked identifier>
                                  [{<period> <asterisked identifier>}...]
<asterisked identifier> ::= <identifier>
<derived column> ::=  <value expression> [<as clause>]
<as clause> ::= [AS] <column name>
<all fields reference> ::= <value expression primary> <period> <asterisk>
                        [AS <left paren> <all fields column name list> <right paren>]
<all fields column name list> ::= <column name list>
```

Listado 2.8: Expresión del símbolo `<query specification>`

El tipo definido para esta sentencia es `query_spec()`, y corresponde con el siguiente record:

```
-record(
 select,
 {
   quantifier = all :: eesql:set_quant(),
   columns = [] :: list(eesql:derived_column()),
   %% The rest of columns represents <table expression>
   from :: eesql:from_clause(),
   where :: eesql:predicate() | undefined,
   group_by = [] :: list(eesql:column_reference()),
   order_by = [] :: list(eesql:sort_spec()),
   offset :: {non_neg_integer(), non_neg_integer()}
          | undefined,
   for_update = false :: boolean()
 }
).
```

Listado 2.9: Definición del record para la sentencia `SELECT`

El símbolo `<set quantifier>` corresponde con el campo `quantifier` del record, y tiene por valor por defecto el átomo `all`. El campo `columns` corresponde con el símbolo no terminal `<select list>`, y por valor por defecto tiene la lista vacía, que se corresponde con el símbolo terminal *. Si se fija bien en la gramática, el símbolo no terminal `<table expression>` corresponde con los campos `from`, `where` y `group_by`, que a su vez son los símbolos no terminales `from_clause`, `where_clause` y `group_by_clause`, respectivamente; estas dos últimas son símbolos opcionales en la gramática. Los campos `order_by`, `offset` y `for_update` están documentados en la web de PostgreSQL[1].

Insert Statement

La sentencia `INSERT` se utiliza para crear nuevas filas en una tabla de la base de datos. En la gramática, se corresponde con el símbolo no terminal `<insert statement>`, que se expresa de la siguiente manera:

```
<insert statement> ::= INSERT INTO <insertion target> <insert columns and source>
<insertion target> ::= <table name>
<insert columns and source> ::= <from subquery>
                              | <from constructor>
                              | <from default>
```

[1] https://www.postgresql.org/docs/current/static/sql-select.html

```
<from subquery> ::= [<left paren> <insert column list> <right paren>]
                    [<override clause>] <query expression>
<from constructor> ::= [<left paren> <insert column list> <right paren>]
                       [<override clause>] <contextually typed table value constructor>
<override clause> ::= OVERRIDING USER VALUE | OVERRIDING SYSTEM VALUE
<from default> ::= DEFAULT VALUES
<insert column list> ::= <column name list>
```

Listado 2.10: Expresión del símbolo `<insert statement>`

Este símbolo está representado en EESQL con el tipo `insert_stmt()`, y su definición es el siguiente record:

```
-record(
 insert,
 {
   table :: eesql:table_name(),
   columns :: nonempty_list(eesql:column_name()),
   values :: nonempty_list(eesql:row_value_expr()), %% values to insert
   %% UPDATE when the columns conflicts
   on_conflict_update_target :: undefined | [eesql:column_name()]
 }
).
```

Listado 2.11: Definición del record para la sentencia `INSERT`

El campo `table` del record se asocia al símbolo no terminal `<table name>` de la gramática, y se representa mediante el tipo `table_name()`, que indica la tabla donde se van a insertar la o las filas correspondientes. El campo `columns` corresponde con el símbolo no terminal `<from constructor>`, y el campo `values` se asocia con el símbolo `<insert column list>`. El campo `on_conflict_update_target` corresponde a un parámetro opcional de PostgreSQL, que indica qué acciones tomar en caso de que haya que resolver algún conflicto[2].

Update Statement

`UPDATE` sirve para actualizar los campos de una fila o varias de filas de una tabla, y en la gramática se encuentra expresado de la siguiente forma:

```
<update statement: searched> ::= UPDATE <target table> SET <set clause list>
                                 [WHERE <search condition>]
```

[2] https://www.postgresql.org/docs/10/static/sql-insert.html

Listado 2.12: Expresión del símbolo `<update statement: searched>`

La representación que se le ha dado en EESQL ha sido mediante el tipo `update_stmt()`, y su definición es la que se detalla a continuación:

```
-record(
update,
{
  table :: eesql:table_name(), %% UPDATE target table
  %% (SET) columns to update
  set :: nonempty_list(eesql:set_clause()),
  %% (WHERE) search condition
  where :: eesql:predicate() | undefined
}
).
```

Listado 2.13: Definición del record para la sentencia `UPDATE`

Al igual que en el record `insert`, el campo `table` referencia a la tabla a la cual se van a actualizar una o varias filas. El campo `set` corresponde al símbolo no terminal `<set clause list>`, para indicar las columnas que se van a actualizar junto con sus correspondientes nuevos valores. Por último, el símbolo no terminal opcional `<search condition>` está asociado al campo `where` que se utiliza para establecer una condición para la actualización.

Delete Statement

La sentencia `DELETE` borra filas de la tabla que se especifique. Su expresión en la gramática es la que sigue a continuación:

```
<delete statement: searched> ::= DELETE FROM <target table> [WHERE <search condition>]
```

Listado 2.14: Expresión del símbolo `<delete statement: searched>`

El tipo asignado a esta sentencia es `delete_stmt()`, que está representado con el siguiente record:

```
-record(
 delete,
 {
   %% (DELETE FROM) target table
   table :: eesql:table_name(),
```

```
    %% (WHERE) search_condition
    where :: eesql:predicate() | undefined
  }
 ).
```

Listado 2.15: Definición del record para la sentencia `DELETE`

Al igual que pasa con los records de las sentencias `INSERT` y `UPDATE`, el campo `table` se utiliza para identificar la tabla donde se va a producir el borrado de filas, mientras que el campo `where` se utiliza para establecer una condición por la cual se produce el borrado de filas.

2.3.3 Sentencias exclusivas de PostgreSQL

Esta sección aborda algunas sentencias que no se encuentran especificadas en la gramática de referencia, pero que forman parte de la sintaxis que ofrece PostgreSQL.

Refresh Materialized View

Antes de explicar esta sentencia, es necesario conocer lo que es una vista materializada. Las vistas materializadas (*materialized views*) son tablas creadas a partir de la ejecución de una query. No obstante, los datos de estas tablas no son almacenados de la misma manera que las tablas convencionales, de manera que hay que refrescar sus datos mediante la sentencia `REFRESH MATERIALIZED VIEW` [3]. Esta sentencia recibe como parámetro el nombre de la tabla, entre otros parámetros que no están contemplados en esta versión de la biblioteca.

Esta sentencia está tipada mediante `pg_refresh_stmt()`[4], que contiene el record que contiene a continuación:

```
-record(
  pg_refresh,
  {
    materialized_view :: eesql:table_name()
  }).
```

Listado 2.16: Definición del record para la sentencia `REFRESH MATERIALIZED VIEW`

[3] https://www.postgresql.org/docs/10/static/sql-refreshmaterializedview.html
[4] Los tipos que empiezan por *pg* indican que son exclusivos de PostgreSQL.

Truncate

Tal y como se especifica en la documentación de PostgreSQL, la sentencia `TRUNCATE` borra todas las filas de un conjunto de tablas; tiene el mismo efecto de un `DELETE`, pero actúa de manera más rápida y efectiva [5].

En EESQL esta sentencia está incluida mediante el tipo `truncate_stmt()`, y define el siguiente record:

```
-record(
truncate,
{
  table :: eesql:table_name(),
  cascade = false :: boolean()
}).
```

Listado 2.17: Definición del record para la sentencia `TRUNCATE`

El campo `cascade` hace referencia al parámetro `CASCADE` de la sentencia, que por defecto está **`false`**, pero si está activado, borra ("trunca") las tablas que contengan claves foráneas referentes a la tabla a la que se aplica `TRUNCATE`.

Cláusula opcional WITH

Con la cláusula opcional `WITH`, se puede establecer una relación entre un alias y una query, para después usarla en la cláusula `FROM` de una sentencia `SELECT`, por ejemplo[6]. Cabe mencionar que esta versión de la biblioteca solo permite establecer una relación entre un identificador que sirva como alias y un `SELECT`. Así pues, el record utilizado para representar este es el siguiente:

```
-record(
 pg_with,
 {
   definitions :: [{eesql:id(), eesql:query_spec()}],
   select :: eesql:query_spec()
 }).
```

Listado 2.18: Definición del record `pg_with`

[5] `https://www.postgresql.org/docs/current/static/sql-truncate.html`

[6] Aunque la cláusula `WITH` se puede expresar en la gramática, se ha seguido la documentación de PostgreSQL para su desarrollo.

El campo `definitions` permite declarar varios alias junto con su correspondiente `SELECT`. La documentación de PostgreSQL especifica que se puede asignar las sentencias `INSERT`, `DELETE` y `UPDATE` a los alias. Este cláusula sirve para referenciar en la `SELECT` principal a las subqueries por nombre.

2.3.4 Otros símbolos representados

En esta sección se van a tratar algunos símbolos no terminales de la gramática que tienen su propia representación en EESQL. La función auxiliar principal en la que se apoya la biblioteca (`to_sql/2`) trata a estos símbolos de la gramática, de manera que posean su propio nivel en la representación del AST y sea más fácil representar la query correspondiente.

Value Expression

El símbolo `<value expression>` es uno de los símbolos con más expresividad de la gramática [5]. Sin embargo, en la representación de EESQL está limitada a los siguientes tipos:

```
-type value_expr() ::
      identifier_chain() %% <column reference>
    | {routine_invocation(), [value_expr()]} %% Represents function calls
    | [value_expr()] % Array (maybe nested)
    | literal().
```

Listado 2.19: Definición del tipo `value_expr()`

Con el tipo `value_expr()` se pueden representar cadenas de identificadores separadas por punto, llamadas a rutinas con sus correspondientes parámetros, arrays y el tipo `literal()`. Cabe mencionar que los arrays son estructuras exclusivas de PostgreSQL, que no debieran ser usadas en otras bases de datos como por ejemplo MySQL.

Predicate

El tipo `predicate()` sirve para representar las expresiones booleanas que se deseen y que se utilizan para establecer algún tipo de filtro a la hora de ejecutar una query. Este tipo está definido de la siguiente manera:

```
-type predicate() ::
      boolean()
    | {'not', predicate()}
```

```
| {'and', [predicate()]}
| {'or', [predicate()]}
| {value_expr(), binop(), value_expr()}
| {is_null, column_reference()}
| {exists, query_spec()}
| {between, value_expr(), value_expr(), value_expr()}
| {in, value_expr(), query_spec()}.
```

Listado 2.20: Definición del tipo `predicate()`

Where Clause

El símbolo `<where clause>` se trata con el tipo `predicate()`. En caso de haber un predicado que haya que tratar, se serializa haciendo una llamada a la propia función auxiliar principal, añadiendo la palabra reservada `WHERE` al principio, de lo contrario, se devolverá un vacío (`""`).

Table Primary

El símbolo `<table primary>` se encuentra expresado en la gramática como un símbolo no terminal al cual se accede mediante el símbolo `<table reference>`, que explicaremos más adelante. EESQL define el tipo `table_primary()` de la siguiente manera:

```
-type table_primary() :: id()
                      | {id(), id()} %% AS
                      | {query_spec(), id()}
                      | pg_call()
                      | {pg_call(), id}.
```

Listado 2.21: Definición del tipo `table_primary()`

Con este tipo se pueden serializar identificadores como las columnas de una tabla, ya sea por su propio nombre o especificando un alias, darle a una subsentencia `SELECT` un alias, o una llamada a rutina, ya sea por ella misma o dándole un alias. Con respecto a las llamadas a rutina, es necesario mencionar que son propias de PostgreSQL y que no debieran utilizarse en otras bases de datos.

Table Reference

En la gramática, el símbolo `<table reference>` expresa una referencia a una tabla, que bien puede ser una `<table primary>` o una `<joined table>`. Dado que ya se ha explicado el primer símbolo en el apartado anterior, en esta se va a explicar cómo se representa la cláusula `JOIN`.

La cláusula `JOIN` sirve para combinar dos o más filas o tablas basándose en una columna en común para establecer la relación.

```
<joined table> ::= <cross join>
                 | <qualified join>
                 | <natural join>
                 | <union join>
<cross join> ::= <table reference> CROSS JOIN <table primary>
<qualified join> ::= <table reference> [<join type>] JOIN
                    <table reference> <join specification>
<natural join> ::= <table reference> NATURAL [<join type>] JOIN <table primary>
<union join> ::= <table reference> UNION JOIN <table primary>
<join specification> ::= <join condition> | <named columns join>
<join condition> ::= ON <search condition>
<named columns join> ::= USING <left paren> <join column list> <right paren>
<join type> ::= INNER | <outer join type> [OUTER]
<outer join type> ::= LEFT | RIGHT | FULL
<join column list> ::= <column name list>
```

Listado 2.22: Expresión del símbolo `<joined table>`

En EESQL se ha definido el tipo `joined_table()` que está definido por los tipos correspondientes a los símbolos de la gramática que expresa. Se pueden representar los diferentes tipos de `JOIN` mediante los records creados para cada uno de ellos, tal y como se muestra en los siguientes listados:

```
-record(
 cross_join,
 {
   left :: eesql:table_ref(),
   right :: eesql:table_primary()
 }
).
```

Listado 2.23: Definición del record `cross_join`

```
-record(
 natural_join,
 {
   left :: eesql:table_ref(),
   right :: eesql:table_primary()
 }
).
```

Listado 2.24: Definición del record `natural_join`

```
-record(
 union_join,
 {
   left :: eesql:table_ref(),
   right :: eesql:table_primary()
 }
).
```

Listado 2.25: Definición del record `union_join`

```
-record(
 qualified_join,
 {
   type = inner :: eesql:join_type(),
   left :: eesql:table_ref(),
   right :: eesql:table_ref(),
   on :: eesql:predicate()
 }
).
```

Listado 2.26: Definición del record `union_join`

A continuación se muestran algunos de los tipos de `JOIN` que se pueden representar con EESQL:

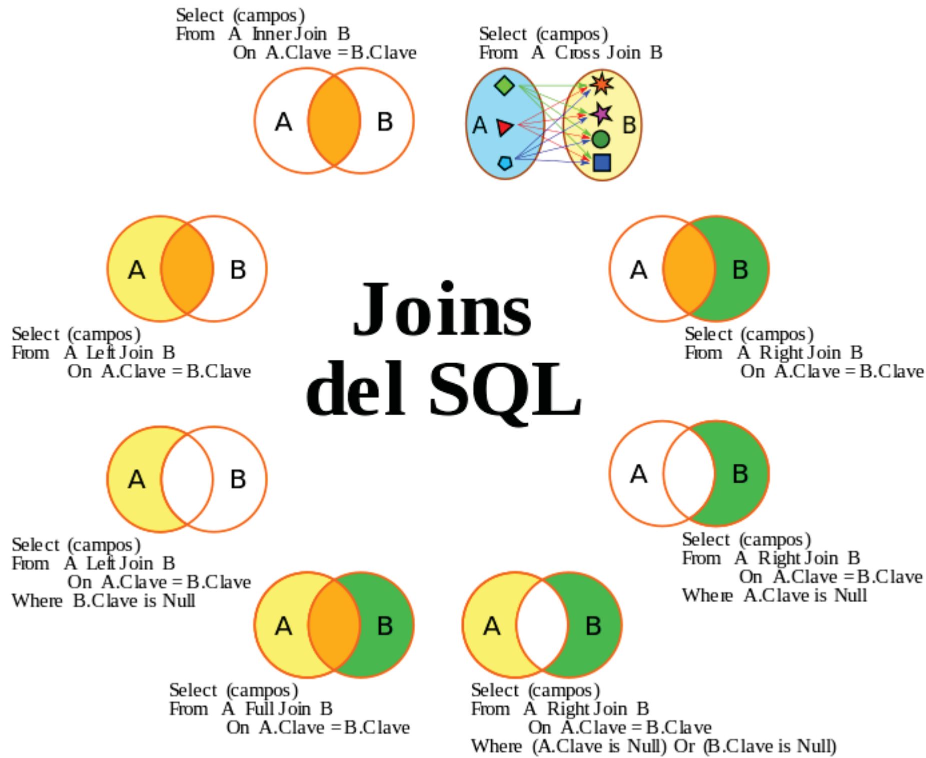


Figura 2.1: Ejemplos de `JOIN`

Literal

El tipo `literal()` está basado en el símbolo no terminal `<literal>` definido en la gramática. Este tipo se utiliza para componer la lista de parámetros que devuelve la función `to_sql/1` como segundo parámetro de la tupla. La definición del tipo es la que se muestra a continuación:

```
-type literal() ::
      null
    | boolean()
    | binary()
    | integer()
    | float().
```

Listado 2.27: Definición del tipo `literal()`

La función auxiliar `to_sql/2` serializa este tipo para que los parámetros correspondientes tengan su propio nivel de representación en el AST, en forma de parámetros con la forma

"$Num", donde "Num" es un número entero positivo. La principal razón por la cual se usa esta técnica es la de evitar posibles ataques por inyecciones de código SQL.

2.4 Funcionalidades sin añadir

Esta sección está dedicada a funcionalidades que están implementadas pero que no se pueden utilizar todavía, al menos en esta versión de la biblioteca. Para ser más preciso, se va a hablar sobre el enriquecimiento de la sentencia `UPDATE`.

2.4.1 UPDATE FROM

La sentencia `UPDATE` permite actualizar una campo de una fila de la tabla con datos de otra tabla; esto se conoce por *correlated update*. Para poder representar este tipo de query, es necesario rediseñar el record `update`, de tal manera de que mantenga la funcionalidad que ha prestado hasta el momento y que los cambios no sean muy profundos para el usuario de la biblioteca. Así pues, la nueva implementación del record `update` es la que sigue a continuación:

```
-record(
 update,
 {
   table :: eesql:table_name(), %% UPDATE target table
   %% (SET) columns to update
   set :: nonempty_list(eesql:set_clause()) |
   {[eesql:column_name()], eesql:query_spec()},
   %% (FROM) Source of data (<from clause>)
   from = [] :: [eesql:table_ref()],
   %% (WHERE) search condition
   where :: eesql:predicate() | undefined
 }
).
```

Listado 2.28: Redefinición del record `update`

Si se compara este record con el del listado 2.13, se aprecia que el campo `set` está enriquecido, de manera que ahora se pueden enlazar varias columnas a una subquery `SELECT`, mediante una tupla, tal y como se muestra en el anterior listado. Además, se ha añadido la cláusula `FROM` para indicar de qué tablas se sacan las columnas, en caso de que se quiera establecer una igualdad entre las columnas de dos o mas tablas. Para garantizar el correcto

funcionamiento de esta nueva funcionalidad, se realizaron pruebas unitarias con EUnit, un framework para realizar tests unitarios que se explicará más adelante.

2.5 Dificultades encontradas y soluciones propuestas

Esta sección trata de recapitular uno de los problemas que se han encontrado durante el desarrollo de EESQL y la solución que se llevó a cabo. Este problema trata sobre una refactorización de tipos en EESQL y de una de las funciones de casting.

2.5.1 Los tipos `name()`, `id()` e `identifier_chain()`

Existía un tipo en EESQL que se utilizaba para definir otros tipos como `column_name()`, `column_reference()` o `table_name()`. Este tipo se llamaba `name()` y estaba definido como un átomo. Si bien es cierto que este tipo representaba bien a los tipos mencionados anteriormente, el problema era que no representaba a ningún símbolo de la gramática. Por tanto, se tomó la decisión de sustituir este tipo por dos tipos que sí que tenían su representación como símbolos de la gramática. Estos tipos son `id()`, que representa el símbolo `<identifier>`, e `identifier_chain()`, que representa el símbolo `<identifier chain>`, este último es una composición de `<identifier>` separados por punto. Estos dos tipos son ambos átomos al igual que lo era el tipo `name()`, pero la diferencia está en que estos ahora sí que referencian bien a sus correspondientes símbolos tal y como se define en la gramática. Además, se han definido dos funciones de casting para cada tipo, que implementan la misma función de conversión, pero ayudan a entender mejor el código a los futuros desarrolladores de la biblioteca.

2.6 Diseño

En esta sección se van a tratar las herramientas utilizadas que forman parte del desarrollo y mantenimiento de esta biblioteca:

- **EUnit**. EUnit es un framework de testing para realizar pruebas unitarias en Erlang. Las características más importantes del testing unitario son que reduce el riesgo de hacer cambios en el código fuente; da ayuda al desarrollador en el proceso de desarrollo del código; hace que la integración de componentes sea más sencilla y las propias pruebas sirven como documentación del código desarrollado [6].

- **Dialyzer**[7]. Como ya se ha explicado en la sección 1.1 el capítulo 1, Dialyzer es una herramienta del estándar de Erlang/OTP que sirve para realizar un análisis estático de tipos, así como definir los tipos que el desarrollador considere oportuno y especificar los argumentos y los valores devueltos por las funciones del módulo en cuestión.

- **Rebar3**[8]. Rebar3 es una herramienta que facilita crear y desarrollar bibliotecas y aplicaciones en Erlang. Entre las funcionalidades que ofrece están las de gestionar las dependencias del proyecto; compilar el código; ejecutar Dialyzer y realizar pruebas sobre el proyecto, además de documentar el proyecto que se esté desarrollando. EESQL es una biblioteca que está bajo Rebar3 y en la que se da uso a todas estas funcionalidades.

2.7 Trabajos futuros

Esta sección está dedicada a hablar sobre los hitos que aun están por cubrir en este trabajo.

Si bien es cierto que este trabajo estaba dedicado al desarrollo de la biblioteca tanto en Erlang como Elixir, no hay hasta este momento ningún avance con el desarrollo de una nueva biblioteca para este último lenguaje. Sin embargo, cabe pensar que el funcionamiento de esta nueva biblioteca será similar al de EESQL: una función principal a la que se le pase como input una estructura de datos de Elixir y devuelva una representación de la query junto con sus correspondientes parámetros en una tupla. Además, dado que Elixir también es un lenguaje con tipado dinámico, habría que aplicar la misma técnica de la declaración de tipos explícitos con la herramienta Dialyzer para este lenguaje (Dialyxir). Por otra parte, las sentencias que hemos visto en este capítulo mantendrían su representación (uso de átomos para las queries de transacciones, records para las sentencias de operaciones CRUD, etc.).

Por otra parte, no se ha hablado sobre la motivación que existe el haber declarado dos tipos como son `id()` e `identifier_chain()`, que están definidos por el mismo tipo. Existe la necesidad de validar por medio de expresiones regulares estos dos tipos, de manera que se notifique al desarrollador, pero esta funcionalidad no ha sido implementada todavía para no perder la integración con los sistemas de Coowry.

Por último, EESQL tiene potencia como para poder representar la gramática de SQL en su totalidad, de modo que enriquecer las funcionalidades que ofrece es un hito constante.

[7] `http://erlang.org/doc/man/dialyzer.html`

[8] `https://www.rebar3.org`

2 DESARROLLO

3

CONCLUSIONES

Ya se mencionó en el Capítulo 1 que EESQL es una biblioteca privada que forma parte del tooling de Coowry, pero se desea que esta adquiera suficientes funcionalidades como para poder liberar la biblioteca en alguna plataforma de desarrollo como Github o Gitlab y que pueda resultar de utilidad para los desarrolladores de la comunidad Erlang. Poder liberar la biblioteca supondría conocer la opinión de otros desarrolladores ajenos a la compañía que podrían dar un enfoque diferente e incluso colaborar en añadir nuevas funcionalidades de otras bases de datos como MySQL o MariaDB.

Desarrollar esta biblioteca ha sido todo un reto desde el punto de vista tanto profesional como académico, que ha puesto a prueba mis conocimientos sobre Erlang y SQL y ha hecho darme cuenta de mis fortalezas y de mis debilidades. Espero que esta biblioteca sea de referencia a medio plazo dentro de la comunidad Erlang.

3 CONCLUSIONES

Bibliografía

[1] F. Cesarini y S. Thompson, *Erlang Programming*. O'Reilly, 2009.

[2] F. Hébert, *Learn You Some Erlang for Great Good!* No Starch Press, 2013.

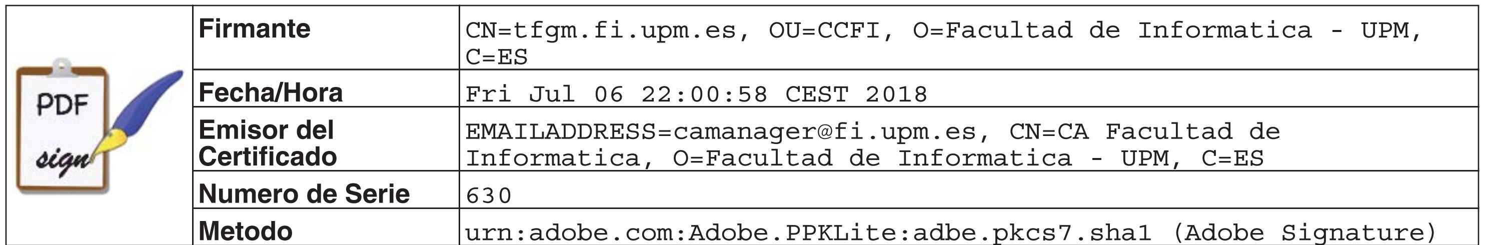
[3] ERICSSON AB. (2018). Types and functions specifications, dirección: `http://erlang.org/doc/reference_manual/typespec.html` (visitado 19-06-2018).

[4] J. Armstrong, *Programming Erlang*. The Pragmatic Bookshelf, 2007.

[5] R. Savage. (2017). BNF grammar for ISO/IEC 9075-2:2003 - database language SQL (SQL-2003) sql/foundation, dirección: `https://ronsavage.github.io/SQL/sql-2003-2.bnf.html` (visitado 25-06-2018).

[6] ERICSSON AB. (2018). EUnit - a lightweight unit testing framework for erlang, dirección: `http://erlang.org/doc/apps/eunit/chapter.html` (visitado 30-06-2018).

Este documento esta firmado por



PDF sign	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Jul 06 22:00:58 CEST 2018
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)