GETSQL – Translation of QEL to SQL

Pär Sjöberg
Abstract
Edutella is a Peer-to-Peer (P2P) system for exchanging information about resources on the Internet. Originally it was designed to be used by different educational institutions. It is built on the framework of JXTA for P2P systems and Semantic web technology with Resource Description Framework, RDF. Edutella has its own query language, QEL, which is designed for exchanging queries and is not per se executable. This thesis investigates the possibilities of translating QEL to SQL with the focus on three things

- Completeness, translating as much of the QEL language as possible.
- Efficiency, making the execution of the SQL efficient.
- Independence of database layout.

This is of interest since none of the existing translations live up to these requirements. In this report we present an algorithm for translating a subset of QEL to SQL queries. We also discuss different ways of increasing the efficiency of the execution of the SQL query. Last we present our design and implementation of a translator, named the Generic Edutella query language Translator to SQL, or abbreviated GETSQL.

Referat
GETSQL: Översättning av QEL till SQL
Edutella är ett Peer-to-Peer (P2P) system för att utbyta information om resurser på Internet. Det var från början tänkt att användas av olika utbildningsinstitutioner. Det bygger på JXTAs ramverk för P2P system och Semantic web-teknologi med Resource Description Framework, RDF. Edutella har sitt eget frågespråk, QEL, som är inriktat på överföring av frågor och inte exekverbart. Den här rapporten utredar förutsättningarna för att översätta QEL till SQL med fokus på tre saker

- Att vara komplett, översätta så mycket som möjligt av QEL.
- Effektivitet, att göra SQL-frågorna så effektiva som möjligt.
- Oberoende av databaslayout.

Detta är av intresse eftersom inga av de befintliga översättarna lever upp till dessa krav. I rapporten presenteras först en algoritm för att översätta en delmängd QEL till SQL-frågor. Sedan går vi igenom hur algoritmen kan modifieras för att effektivisera exekveringen av SQL frågan. Sist presenterar vi vår design och implementation av en översättare, kallad the Generic Edutella query language Translator to SQL, eller förkortat GETSQL.
Contents

1 Introduction ....................................................... 1
   1.1 Edutella queries .............................................. 1
   1.2 Edutella providers ........................................... 1
   1.3 Problem Definition .......................................... 2
       1.3.1 Problem ................................................ 2
       1.3.2 Structure of this report .............................. 3
       1.3.3 Administrative ........................................ 3

2 Background .......................................................... 4
   2.1 RDF and the relational model two approaches ............... 4
       2.1.1 Schema based .......................................... 4
       2.1.2 Statement based ....................................... 4
       2.1.3 Our design ............................................. 5
   2.2 Datalog and QEL ............................................... 5
       2.2.1 Datalog syntax ......................................... 5
       2.2.2 QEL .................................................... 7
       2.2.3 Different levels of QEL ............................... 7
       2.2.4 Built-in predicates in QEL ........................... 8
       2.2.5 Outer join ............................................. 9
   2.3 Relational Algebra and SQL .................................. 9
       2.3.1 Relational algebra ..................................... 9
       2.3.2 The SELECT operation ................................. 10
       2.3.3 The PROJECT operation ............................... 10
       2.3.4 The JOIN operations ................................... 11
       2.3.5 The SET operations ..................................... 11
       2.3.6 SQL .................................................... 12
       2.3.7 SELECT clause ........................................ 12
       2.3.8 FROM clause ........................................... 12
       2.3.9 WHERE clause ......................................... 12
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>Constraining variables</td>
<td>35</td>
</tr>
<tr>
<td>6.6</td>
<td>Generating query results</td>
<td>35</td>
</tr>
<tr>
<td>6.7</td>
<td>Rule references</td>
<td>37</td>
</tr>
<tr>
<td>6.8</td>
<td>Outer join</td>
<td>37</td>
</tr>
<tr>
<td>6.9</td>
<td>SQL translation example</td>
<td>38</td>
</tr>
</tbody>
</table>

7 Conclusions and future work 40

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Conclusions</td>
<td>40</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Theoretical translation</td>
<td>40</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Efficiency</td>
<td>41</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Database layout independence</td>
<td>41</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Results</td>
<td>41</td>
</tr>
<tr>
<td>7.2</td>
<td>Future work</td>
<td>42</td>
</tr>
</tbody>
</table>

Bibliography 44
Chapter 1

Introduction

The most commonly used P2P systems today are used for filesharing on the Internet. Systems like Kazaa and Gnutella have gathered millions of users that share, for example, music and movie files. They supply an interface where the users can search the network for the desired files by specifying queries on metadata about the files. The metadata present is not very complex.

1.1 Edutella queries

The Edutella P2P network was originally designed for sharing information about educational resources on the Internet[1]. The information of these resources needs a much richer form of metadata[2] to be described than the file sharing system mentioned above. Edutella’s query language, RDF Query Exchange Language, abbreviated to QEL, is designed to query metadata defined with the W3C recommendation for standardized metadata on the web, the Resource Description Framework, RDF[3].

RDF has many similarities to a deductive database and the QEL syntax is based on Datalog, a deductive database language. QEL has its own built-in predicates that are designed specifically for querying the RDF data model. QEL also introduces a feature that is not defined in Datalog, the outer join. This is taken from the world of relational databases.

1.2 Edutella providers

The Edutella network is consists of peers. The peers are divided into two groups, consumer and provider peers. The provider peers are responsible for receiving queries from consumer peers, over the Edutella network, somehow executing
them against an RDF data source and transmitting the reply back to the consumer. The data source can be of various kinds, for example, an RDF/XML file or a relational database.

For a provider to be useful it needs to implement most of the functionality of the QEL language and make use of it in order to answer complex queries in an efficient way.

The providers implemented so far have all been able to answer queries that are not very complex and with RDF models that are not very large. This is not good enough for the Edutella network to be useful on a larger scale.

1.3 Problem Definition

The QEL language is focused on exchange and not directly executable. Therefore each provider in the network has to support an executable query language and also implement a translation engine from QEL to that query language.

A common approach for storing RDF is as tables in a relational database. This report will focus on translating QEL to Structured Query Language, SQL, which is a standardised way to execute queries against a relational database.

1.3.1 Problem

The original problem for this thesis can be divided into three parts

- Translate QEL to SQL on a theoretical level.
- Look at the efficiency issues of a translation.
- See what can be done with different database layouts and database vendors.

First we need to investigate how much of QEL that can be translated to SQL. Datalog is based on relational calculus while SQL is based on relational algebra[4]. Relational calculus and relational algebra are incompatible[5], which means that everything expressed in Datalog cannot be expressed in relational algebra and vice versa, everything expressed in relational algebra cannot be expressed in Datalog. However, work has been done to translate parts of datalog into SQL. This is the starting point from where we try to translate QEL, which largely builds on Datalog. An exception to this is one of the most recent improvements, the support for outer join. Outer join is taken from the world of relational databases and therefore it is built in to most database implementations and supported in SQL.
The second part of the problem is investigating how to make the translation efficient. A provider needs to be able to handle queries fast. We will investigate what changes can be done to the translation algorithm in order to improve the efficiency of the resulting SQL.

Lastly we will look at how we can make an implementation independent of what database engine that is used and which database layout that is used to store the RDF model. There are several ongoing projects developing frameworks for RDF and some of them support storage of the RDF model in a relational database. Typically, they have different approaches for the design of the database layout. There are also some differences in the SQL language that is implemented by different relational database vendors.

1.3.2 Structure of this report

This report is structured as follows:
Chapter 2 gives a short introduction to RDF and relational models. It also introduces the query languages and the theory behind them. In chapter 3 we describe the algorithm used to make the translation. Chapter 4 discusses how to improve efficiency of the translation. In chapter 5 and 6 we talk about the design and implementation of GETSQL, the Generic Edutella query language Translator to SQL.

1.3.3 Administrative

This Master’s thesis work (20 credits) has been carried out at the Centre for User Oriented IT Design (CID), at the Royal Institute of Technology (KTH), Stockholm. The supervisor has been Ambjörn Naeve, CID, KTH and the examiner has been Yngve Sundblad, CID, KTH.
Chapter 2

Background

In this chapter we will start with a short introduction to how the RDF Model can
be represented in a relational database. Then we describe the query languages
and the theories they are based on.

2.1 RDF and the relational model - two approaches

There are mainly two different approaches to store RDF in a relational database.
We have called them Schema based and Statement based.

2.1.1 Schema based

One approach of storing RDF in a relational database is to focus on the predi-
cates(properties) of the statements. This approach was used by the RDF open
source project Sesame[6] in their first test setup with a relational database
repository. In this approach the database layout is dynamic, every predicate is
represented by a table in the database[7]. We call this a Schema based approach,
the database schema is based on the RDF model. This is not the common way
of using a relational database and they found that there was efficiency problems
when they had to create many new tables.

2.1.2 Statement based

Another approach of storing RDF in a relational database is to focus on the
Statements of the RDF model. Basically all statements are stored in one table
which has the columns subject, predicate and object. Jena[8] is another open
source project developing an RDF framework. This framework is the base for
the Java API for handling RDF in Edutella. Jena uses a static database layout
for storing RDF.
2.1.3 Our design

Sesame has in its latest releases, due to lacking efficiency, started to move towards a statement based database layout, similar to the one Jena has. For this reason we made our design and implementation with a statement based database layout in mind. However, we think that it should be possible, without too many changes, to also have support for a schema based database layout.

2.2 Datalog and QEL

Datalog is a language used to implement deductive databases and deductive databases can be seen as an extension of relational databases based on predicate logic. Let’s go through a short listing of the ingredients of Datalog. A Datalog database $\mathfrak{D}$ consists of a finite set of predicates where each is either

- a ground fact $q$, which is similar to a relation in a relational database or
- a range-restricted rule, $q \supset p_1, \ldots, p_n$ with $n > 0$

with the condition that the predicates are partitioned into two sets:

1. Extensional predicates (EDB), which appear in facts and bodies of rules and queries, but not in heads of rules.

2. Intensional predicates (IDB), which appear in heads and bodies of rules and queries, but not in the facts.

The single RDF statement can be looked on as a fact (EDB) and several statements put together can be described with a rule (IDB). This is why QEL is based on the deductive database language, Datalog.

2.2.1 Datalog syntax

Here we will give a short introduction to the syntax of Datalog with some sample queries.

To define the data, the facts base, for a Datalog database we use the EDB predicates. Predicates consist of a name that suggests what they relate to and a fixed number of arguments. Arguments in the facts are always constants.

In figure (2.1) we define some facts about employees at a company and the departments they work in. We have two predicates, employee and department. The employee predicate contains name, SSN (Social Security Number), department number and salary of an employee. The department predicate contains department name, department number and SSN of the manager of the department.
employee('John', 'Smith', '123456789', 5, 30000).
employee('Alicia', 'Zelaya', '999887777', 4, 35000).
employee('James', 'Borg', '888665555', 1, 55000).
department('Research', 5, '33345555').
department('Administration', 4, '987654321').
department('Headquarters', 1, '888665555').

Figure 2.1: A small Datalog facts base.

From the facts in figure (2.1) we want to retrieve the first name, surname, department number and salary of the employee with SSN '123456789'. To form this query we use the predicate employee and replaces the constants we look for with variables. A predicate that contains variables is called a query literal. The query will be formed like:


Figure 2.2: A simple Datalog query using only an EDB predicate.

The only match we will get is information about the employee John Smith. The previous query uses a query literal that represents an EDB predicate, directly stored in the facts base. The next example shows how the IDB predicates are defined to make a more complex query.

manager(DNAME, FNAME, LNAME) : –department(DNAME, DNO, SSN), employee(FNAME, LNAME, SSN, DNO, SALARY).


Figure 2.3: A slightly more complex Datalog query using both EDB and IDB predicates.

The manager rule is used to match department name, and the departments managers first and last name. In the query we use the rule to match the information for the department called 'Headquarters'. With the given facts we will get the name James Borg as result.

In Datalog there is no way to express or in a rule body. For example if we want information about employees from two different departments we cannot do it with one rule. We need to use disjunction. The next example shows how to create a disjunctive query which is a query that has more than one rule for an IDB predicate.
\[ d(DNO) : \neg \text{department('Research', DNO, X)}. \]
\[ d(DNO) : \neg \text{department('Headquarters', DNO, X)}. \]
\[ ?(FNAME, LNAME) \Rightarrow \text{employee(FNAME, LNAME, X, DNO, Y), d(DNO)}. \]

Figure 2.4: A Datalog query using a disjunctive rule.

The query matches the first and last names of employees on the departments called 'Research' and 'Headquarters'. The predicate \( d \) has two different rules which both are used to get the result of the query. With the facts given in figure (2.1) we will get the names John Smith and James Borg. Note that, in this query we specified which variables we are interested in. We will only get values back for the FNAME and LNAME variables.

2.2.2 QEL

Edutellas query language is RDF Query Exchange Language, QEL. It is designed to query RDF data models and based on Datalog. In the next sections we will present the language features, for a more extensive description see [4].

2.2.3 Different levels of QEL

There are five basic complexity levels defined for QEL. These are

- Rule-less query
  A query that is written without using rules. This means that only the matching predicates qel:s and qel:member are used to retrieve data.

- Conjunctive query
  A query that has at most one rule for each predicate.

- Disjunctive query
  A query that allows several rules for each predicate. However, a rule body may not contain a reference to itself, that is recursion.

- Linear recursive query
  A query that can contain linear recursive rules.

- General recursive query
  A query that can contain non-linear recursive rules.
2.2.4 Built-in predicates in QEL

QEL does not use the built-in predicates from Datalog. Instead it defines several built-in predicates of its own. These are divided into two groups, matching predicates and constraining predicates.

2.2.4.1 QEL namespace

Since new predicates can be defined in the query we need a way to separate the built-in predicates from them. To do that the QEL namespace has been created. The QEL namespace is always, http://www.edatella.org/qel#. This means that whenever a QEL built-in predicate is used it is prefixed with qel. For example the QEL built-predicate like will be written qel:like (X, 'Edatella').

2.2.4.2 Matching predicates

The matching predicates in QEL are used to match data from the RDF repository. These can be looked on as the EDB predicates in Datalog. These predicates are qels and qelmember.

- qels
  
  qels takes three arguments and is used to match triples from the RDF statements.

- qelmember
  
  qelmember takes two arguments and is used to check if resources are members of RDF containers.

2.2.4.3 Constraining predicates

The constraining predicates can be compared to the built-in predicates of Datalog. They are used to constrain the data that has been matched and are specifically designed for data within the RDF Model. The following constraining predicates have been defined in QEL.

- qel:equals
  
  is used to check if two RDF literals or RDF resources are the same.

- qel:nodeType
  
  is used to constrain the node type of an RDF node.

- qel:stringValue
  
  is used to specify a certain string value of an RDF literal.

- qel:language
  
  is used to specify the language of an RDF literal.
• qel:dataType
  is used to specify the data type of an RDF literal.

• qel:like
  is used to specify a substring that should be contained by the RDF resource
  or the RDF literal.

• qel:greaterThan
  is used to specify that an RDF literal has to be greater than a certain
  number.

• qel:lessThan
  is used to specify that an RDF literal has to be less than a certain number.

Since the constraining predicates not match any data they also support the
negation operation. This is specified with the operator qel:not.

2.2.5 Outer join

QEL has added functionality that is not defined in Datalog, outer join. This
means that a predicate is always true even if there is no matches for a variable.
In that case the value of the variable is bound to null. This is taken from the
field of relational databases. We will denote the outer in QEL by adding an ‘*’
to the predicates that are to be outer join predicates. For example,

\[ qel: s^*(X, dc: title, Y) \]

2.3 Relational Algebra and SQL

The SQL language used to query relational databases for data is based on Rel-
ational Algebra. In the coming sections we will describe how they relate.

2.3.1 Relational algebra

Relational algebra is a language used to manipulate relations[9]. All operations
defined in relational algebra use relations as operands and result in new rela-
tions. There are two kinds of operations in relational algebra, the set operations
from the mathematical set theory and some operations defined specifically for
relational databases. Relational algebra is a procedural language which means
the result is reached in a step by step way. To look at the different operators
we define a simple realtional model with the following two relations, the first
called EMPLOYEE and the second DEPARTMENT.
2.3.2 The select operation

The select operation is used to retrieve a subset of tuples from a relation. The general notation for the select operation is:

$$\sigma_{<\text{selection condition}>}(<\text{Relation name}>)$$

The relation created with the select operation has the same attributes as the operand relation. The Boolean expression $<$selection condition$>$ is made up of a number of clauses where attributes are compared to each other or to constants. The comparison operators can vary depending on what domain the attributes come from. For example, if we want to retrieve information about the employee with SSN '123456789' we apply the select operation like this on relation (2.5):

$$R \leftarrow \sigma_{\text{SSN}='123456789'}(\text{EMPLOYEE})$$

2.3.3 The project operation

The project operation is defined to project a subset of attributes from a relation to a new relation. The general notation for the project operation is:

$$\pi_{<\text{attribute list}>}(<\text{Relation name}>)$$

The $<$attribute list$>$ can only contain attributes from the operand relation. For example, if we want to project the FNAME, LNAME and SALARY attributes from relation (2.5) we do it like this:

$$R \leftarrow \pi_{\text{FNAME, LNAME, SALARY}}(\text{EMPLOYEE})$$
The project operation also deletes duplicate tuples from the new relation so that the result always is a relation.

2.3.4 The JOIN operations

The JOIN operations are used to combine tuples from two relations into a single tuple. This operation is very important for any relational database with more than one single relation. The JOIN operations are denoted by

\ [<Relation 1 >\joindrelation,<join condition>\ <Relation 2 >

The <join condition> is a Boolean expression that specifies which attributes to join on. There are two kinds of joins defined in relational algebra

- natural join and
- outer join

The natural join means that only tuples that has matching attributes given in the <join condition> are included in the result relation.

Outer join are used to get result even if the <join condition> does not match. There are three kinds of outer join specified, left, right and full outer. Left outer join means that all tuples for Relation 1, the one to the left in the expression, is included in the result relation. For all tuples in Relation 2 where attributes does not match the <join condition> the value NULL is filled in. Right outer join works in the same way except that it is the tuples in Relation 2 that is kept and Relation 1 is filled with NULL values. In Full outer join missing values in both relations are filled with NULL values. We will denote outer join in relational algebra by adding an '*' to the side of the join operator which is to be the outer join. For example, \( R1 \bowtie_{R1.X=R2.Y}^* R2 \), means relation R1 left outer join R2 with the condition that column X in R1 is equal to column X in R2.

2.3.5 The Set operations

A relation is defined as a set of tuples with a fixed set of attributes. Hence, the mathematical set operations, union, intersection, difference and cartesian product are defined in relational algebra. Since the tuples in a relation need to have a fixed set of attributes the relations used as operands in, union, intersection and difference must be, so called, union compatible. This means that the two relations have the same number of attributes and that each pair of attributes have the same domain.
2.3.6 SQL

SQL is the executable language that all relational database vendors support. There has been three different standard versions released so far. SQL is a very rich language and to describe it completely is out of scope for this report. We will describe the parts we have used in the translation implementation and its connection to the relational algebra described in the previous section. This means we will only describe the parts used to retrieve data and create result relations. This is done by using the SQL SELECT statement.

The SELECT statement syntax is divided in three different clauses. The next three sections describes what they are used for and their connections to relational algebra.

2.3.7 SELECT clause

In the select clause we specify what attributes to be included in the result relation of the query. This is similar to the projection operation of the relational algebra. The difference is that SQL does not automatically delete duplicates. To delete duplicates from the result relation we have to use the keyword DISTINCT before the list of attributes. In the select clause it is possible to rename attributes in the result relation.

2.3.8 FROM clause

In the from clause we specify which relations we want to retrieve data from and how they should be joined together. This is where the join operations from the relational algebra is applied. The relations we want to use can be stored in the database or created by using another SELECT statement, a subquery. Some database implementations do not support the use of subsqueries.

2.3.9 WHERE clause

The where clause is where the selection operation from relational algebra is applied. In the where clause we specify what conditions need to be met by the tuples to be included in the result relation. The conditions are Boolean operators defined for the domain of the attribute. For example, an attribute in the domain of integers can be conditioned with > 5. The conditions are then put together with logical AND or OR operators.
Chapter 3

Theoretical translation

This chapter describes the algorithm used to make the translation. But first we give a short introduction on how Datalog queries can be expressed using relational algebra, the base for SQL. Then we describe an outline for translating non recursive QEL queries to relational expressions. Last, we give two examples on QEL queries, one very simple and then a more complex one and show how the translation algorithm can be applied to them.

3.1 Expressing Datalog with relational algebra

All datalog queries except full recursion (linear recursion works though) are possible to translate to relational algebra, see for example [5]. However, since few database implementations support linear recursion (as defined in SQL99) we have restricted our focus of attention to datalog queries without recursion. Furthermore, QEL goes further than datalog by including Outer Join [10] in the style of relational algebra. Even if the semantics of Outer Join is borrowed from relational algebra, converting the extended datalog is uncharted territory which have to be done carefully.

Now when we know that a translation is theoretically possible lets investigate how it can be done in practise.

3.2 Translation of nonrecursive QEL

The base for the algorithm used to translate QEL to relational expressions is the outline for an inference algorithm used to translate nonrecursive datalog queries to relational expressions [9]. When creating the relational expressions we assume that the relational model representing the RDF model has a relation S, with the attributes subject, predicate and object, in which all the statements
are stored, see section 2.1.3. Now we need to show that there is a way to derive a relation representing the result of the query.

The translation algorithm is recursive and the recursive step is described in section 3.2.4 and the terminating requirement is described in section 3.2.3. First we will give some useful terminology and then go through how we deal with the steps.

3.2.1 Matching data with built-in predicates, qels and qel:member

There are numerous built-in predicates defined in QEL. However, so far only two have been defined to match data from the RDF model, namely qel: and qel:member. They are called matching predicates. Each of these predicates matches a subset of the statements stored in the RDF model. For example

\[ gel : s(X, dc : title, Y) \]

Matches all statements that have a predicate, dc:title. In relational algebra (and using the relation \( S \) for the statements) this would be written

\[ \pi_{\text{subject,object}}(\sigma_{\text{predicate='dc:title'}}(S)) \]

We introduce the term limited variable for a variable occurring in a non outer join matching predicate. We use this definition in the next section where we explain how to treat outer join predicates.

3.2.2 How to treat outer join predicates

To make the description of the algorithm we will define two terms that we will use in the description of the different steps.

- common variable
- grouping

A common variable is a variable that is contained by two or more query literals in a rule body. It connects the query literals together. Note that the same variable name can appear in different rules and it does not necessarily mean that it should be matched to the same value.

A grouping is defined as a set of query literals in an outer join part of a rule body that are connected through common variables. In more detail, among all query literals in the outer join part of a rule body, constructed by one or several
groupings, starting with a single query literal including those that have common
variables with them and so on.

There are two kinds of groupings defined. Left outer join groupings and full
outer join groupings. We will only consider the left outer join groupings in this
thesis since it is not clear how the full outer join should be executed yet.

In short, a left outer join grouping is a grouping where all common variables
are limited. All left outer join query literals are executed after the non outer
join query literals, that is, after all limited variables have been bound to a value.

3.2.3 Termination requirement for translation algorithm

Since we are only considering nonrecursive QEL queries the bottom terminat-
ing requirement for our translation algorithm is that there are rules that only
contain QEL builtin-predicates. The top-level query body can be treated the
same way as a rule body.

To derive relations for such rules we first derive relations for the matching
predicates qel$s$ and qel:member that this rule contains, as shown in the previous
section. After that we create the new relation by using the NATURAL JOIN
operation on the derived relations using the common variables as the join condi-
tion. When we join relations together with the NATURAL JOIN operation
the final result relation is not dependent on which order we do the join. The
order will affect the efficiency though and we will discuss this in the next chap-
ter. Lastly we apply the constraining predicates as constraints on the created
relation.

3.2.4 Recursive step

The recursion in the translation algorithm is reflected in the fact that a rule
can reference other rules, but not itself or other rules that directly or indirectly
reference it, since the query cannot be recursive. In order to derive relations
for rules that contain both references to other rules as well as the QEL defined
built-in predicates we only need to derive relations for the built-in matching
predicates and the rules. According to what we previously discussed, we can
derive a relation from a rule containing only built-in predicates, and we know
that there has to be such a rule in the bottom of our query structure, because
this is what makes the algorithm terminate. Then we just join the relations
together and apply the constraints in the same way as for the previous case.
3.2.5 Disjunctive rules

If we have disjunctive rules, i.e. several rules defined with the same head, we derive relations for each rule and then create the new relation by using the UNION operator. The UNION operator takes two relations with the same column definition and merges them.

3.2.6 Outer join

Outer join is the most recently added feature to the QEL language and it has not been completely defined. The definition we used is made in [10].

The basic step to create a relation for a left outer join grouping are the same as for the non outer join literals. The only difference is that instead of joining the relations created within the grouping with NATURAL JOIN we use the LEFT OUTER JOIN operator. The grouping is then joined with a LEFT OUTER JOIN to the relation created with the non outer join literals.

3.3 Two examples

To show how the translation works in practise we give two examples. The example queries are first presented in plain english, then written with QEL and finally translated to relational algebra by using the algorithm defined earlier. In the examples we use the algorithm exactly as it is described, in the next chapter we will look at what we can do to increase efficiency of the algorithm. The first is a very simple query and the second a more typical Edutella query.

3.3.1 A simple example

To give a very simple introduction on how the translation is to be done, we can take the query:

Give me all resources of type exampleType:Book and their titles, requiring that the title contains the word 'Edutella'.

In QEL the query would have been written with just one of the built-in matching predicates, qels, like this:

\[
\text{books}(X, T) : \neg \text{qel : } s(X, \text{rdf : type, exampletype : Book}), \\
\text{qel : } s(X, \text{dc : title, T}), \text{qel : like(T, Edutella')}, \\
?\text{(X, T)} \leftarrow \text{books}(X, T).
\]

This query has only one conjuctive rule \text{Books}(X, T) which only contain built-in predicates. Hence, the relation derived from the rule is also the relation
representing the query results. We use the algorithm defined earlier. First we derive relations for the two query literals that have matching predicates. We call the relations $S_1$ and $S_2$. They are derived as follows:

\[
S_1(X) \leftarrow \pi_{\text{subject}}(\sigma_{\text{predicate}=\text{rdf}\text{/type} \land \text{object}=\text{exampleType:Book}}(S))
\]

\[
S_2(X, T) \leftarrow \pi_{\text{subject}, \text{object}}(\sigma_{\text{predicate}=\text{dc}\text{/title}}(S))
\]

The second step is to join the two relations together, using the NATURAL JOIN operator, to create a new relation, $S_3$. It looks like this expressed in relational algebra:

\[
S_3(X, T) \leftarrow \pi_{S_1, S_2}(S_1 \bowtie_{S_1.X=S_2.X} S_2)
\]

The last step will be to create the result relation, R. This is how it would be written in relational algebra:

\[
R(X, T) \leftarrow \sigma_{S_3.T \text{LIKE} \%\text{Edutella} \%} S_3
\]

Here we have introduced the LIKE condition to our relational algebra. It is used to check if a string contains a substring. Now we have created the relation representing the result of the query.

### 3.3.2 A more typical and complex Edutella query

A typical Edutella query could be formulated like this in plain english:

Give me all the resources of the type exampleType:Book.
For all those resources, return the title and subject if they are available.
If the resources have a creator that is typed as exampleType:Entity, then return its full name (vcard:FN) and its organisational belonging (vcard:ORG) if they are available.

When this query is formulated in a more formal way with QEL/Datalog syntax it looks like this:

\[
\text{creator}(X, F, O) : - \text{g}_e \text{l} : s(X, dc : \text{creator}, C), \text{g}_e : s(C, \text{rdf} : \text{type}, \text{exampleType} : \text{Entity}), \text{g}_e : s^\ast(C, \text{vcard} : \text{FN}, F), \text{g}_e : s^\ast(C, \text{vcard} : \text{ORG}, O).
\]

\[
?(X, T, S, F, O) \rightarrow \text{g}_e \text{l} : s(X, \text{rdf} : \text{type}, \text{exampleType} : \text{Book}), \text{g}_e : s^\ast(X, dc : \text{title}, T), \text{g}_e : s^\ast(X, dc : \text{subject}, S), \text{creator}^\ast(X, F, O).
\]

This query is more complex than the first example. It contains a rule and also some outer join literals. Once again we apply the steps from the algorithm. The first step is to derive relations for all query literals that are non outer join.
We have only one and it is a matching predicate, q.e.s. We form the relation $S_1$, with the attribute $X$:

$$S_1(X) \leftarrow \pi_{\text{subject}}(\sigma_{\text{predicate}=\text{rdf:type}} \land \text{object} = \text{exampleType:Book} (S))$$

Now we look at the outer join literals. We can see that they form a grouping with the common variable $X$ and since $X$ was bound in the non outer join step this will be a LEFT OUTER JOIN. First we get the relations, $S_2$, with the attributes $X$ and $T$, and $S_3$, with the attributes $X$ and $S$, from the two matching predicates:

$$S_2(X, T) \leftarrow \pi_{\text{subject}, \text{object}}(\sigma_{\text{predicate}=\text{dc:title}} (S))$$

$$S_3(X, S) \leftarrow \pi_{\text{subject}, \text{object}}(\sigma_{\text{predicate}=\text{dc:subject}} (S))$$

Now we derive a relation $R$ with the attributes $X$, $F$ and $O$, from the rule creator($X$, $F$, $O$). Like with the query we start by looking at the non outer join literals that the rule contains, two matching predicates. We form the relations $S_4$, with attribute $X$ and $C$, and $S_5$, with attribute $C$:

$$S_4(X, C) \leftarrow \pi_{\text{subject}, \text{object}}(\sigma_{\text{predicate}=\text{dc:creator}}(S))$$

$$S_5(C) \leftarrow \pi_{\text{subject}}(\sigma_{\text{predicate}=\text{rdf:type}} \land \text{object} = \text{exampleType:Entity} (S))$$

We form the relation $R_1$ by joining $S_4$ and $S_5$ together with the NATURAL JOIN operator:

$$R_1(X, C) \leftarrow \pi_{X, S_4, C}(S_4 \bowtie S_4.X=S_5.X \ S_5)$$

Now we form the relations $S_6$, with the attributes $C$ and $F$, and $S_7$, with the attributes $C$ and $O$:

$$S_6(C, F) \leftarrow \pi_{\text{subject}, \text{object}}(\sigma_{\text{predicate}=\text{vcard:FN}} (S))$$

$$S_7(C, O) \leftarrow \pi_{\text{subject}, \text{object}}(\sigma_{\text{predicate}=\text{vcard:ORG}} (S))$$

Since the grouping formed has the common variable $C$ which was bound in the non outer join step we now can form the relation $R$ by joining $R_1$ with relations $S_6$ and $S_7$ using the left outer join operator:

$$R(X, F, O) \leftarrow \pi_{R_1.X, S_6.F, S_7.O}(R_1 \bowtie_{R_1.X=S_6.X} S_6 \bowtie_{R_1.X=S_7.X} S_7)$$
The last step to create the query result relation $Q$, with the attributes $X$, $T$, $S$, $F$, $O$ will be to join the four relations derived:

$$Q(X,T,S,F,O) \leftarrow \pi_{S_1.X.S_2.T.S_3.R.F.R.O}(S_1 \bowtie S_1.X=S_2.X S_2 \bowtie S_1.X=S_3.X S_3 \bowtie S_1.X=R.X R)$$
Chapter 4

Increasing efficiency

In this chapter we discuss ways to increase the efficiency of the algorithm described in chapter 3.

4.1 Efficiency

One important issue to take in consideration when translating QEL queries to an executable language is efficiency. Efficiency in this context means that the provider can reply as fast as possible. Efficiency requirements increase with the number of users on the network. Users does not want to wait too long for replies on their queries and providers can even be jammed if the traffic is too intense and the queries cannot be handled fast enough.

Relational databases have built-in query planners responsible for optimizing the execution plan of the query. When we tested the implementation we discovered some problems with the query-planner for complex queries. Therefore we tried to find ways to help the query planner.

To increase the efficiency of the algorithm discussed in chapter 3 we have found two approaches. We have only considered the non outer join parts of the query. They are presented in the next two sections.

4.2 Constraining literals

In the algorithm described in chapter 3 we apply the constraining predicates last when creating a relation from a rule. This means that the join built in the previous step can contain many tuples that will not be part of the result relation. This will still give the correct result but it may lead to inefficient handling in the database. We cannot assume that end users will write efficient queries. The QEL language specification does not take into account efficiency
issues of implementation. Let’s consider a transform of the query. The transform
is done by rewriting the query and moving the constraining literals into the rule
which bodies containing the matching literals that are to be constrained. This
will not affect the final result of the query but it will create smaller intermediate
relations.

Consider the query:

$$\text{booktitle}(X, T) : \neg qel : s(X, \text{rdf} : \text{type}, \text{qeltypes} : \text{Book}), qel : s(X, \text{dc} : \text{title}, T), \neg (X, T) \rightarrow \text{booktitle}(X, T), qel : \text{like}(T, \text{'Edutella'}).$$

It is a badly written variant of the query in section 3.4.1. Executed as
it stands it would result in a relation containing all resources typed as qel-
types:Book and their titles. We are only interested in the books that have a
title containing the word 'Edutella' and therefore we applied qel:like on the rule.
The problem is that the constraining predicate is added too late. To make it
more efficient we transform the query to:

$$\text{booktitle}(X, T) : \neg qel : s(X, \text{rdf} : \text{type}, \text{qeltypes} : \text{Book}),$$

$$qel : s(X, \text{dc} : \text{title}, T), qel : \text{like}(T, \text{'Edutella'}).$$

$$?(X, T) \rightarrow \text{booktitle}(X, T).$$

We also make a change in how we derive the relations by applying the con-
straining predicate directly on the affected matching predicate like this:

$$\pi_{\text{subject, object}}(\sigma_{\text{predicate='dc:title'}} \land \text{object LIKE 'Edutella'}}(S))$$

Now, when we join, all tuples with a title not containing the word 'Edutella'
will be excluded.

4.3 Order of joins

Another thing that we did not discuss is in which order we join the relations
derived in a rule. Since we only consider natural joins, the order does not affect
the resulting relation. However, it affects the efficiency of the query. Consider
the following rule:

$$r(H, K) : \neg a(H, I), b(I, J), c(J, K).$$

We need to derive relations for the rules $a, b$ and $c$ and then we join them
together to create the relation for the rule $r$. If we first join the relation derived
from rule \(a\) with the relation derived from rule \(c\) we will get a cartesian product since they do not have any common variables. A better approach would be to first join relations with common variables. In this example we would start with joining relation \(a\) with \(b\) and then join relation \(ab\) with \(c\). This chaining process is similar to Jenas Fastpath\[8\] algorithm used by their RDQL translator.

When prioritizing to join relations with common variables first we still do not know where to start, in the example above we could equally well have started to join \(c\) with \(b\) and the \(bc\) with \(a\). To choose the order of execution it would be good to know in advance which relation that would generate the smallest resultset and to start with that one. The knowledge is unfortunately specific to what kind of data that is queried and would require statistical optimization for each installation. This is outside the scope of this thesis. However, it is not unreasonable that the relations with many constraints would generate small resultsets. Hence, when we choose which literal we should use first in the join we start with the one that has the most constraints. Literals for matching predicates are always considered to have more constraints than rule referencing literals.

When we have found the starting literal we go through all remaining literals and look for common variables with the previously joined literals. We are done when all literals, that are matching predicates or referencing rules have been added.

### 4.4 Summary

We found two ways of increasing efficiency for the execution of the translated query. They are independent of each other.

First is to transform the query by moving the constraining predicates to the rule bodies where they constrain the matches. This will decrease the number of tuples retrieved from the database earlier in the execution process.

Second, we create chains of literals with common variables starting with the literal that has the most constraints. This is to reduce the risk of getting cartesian products of relations during the execution. The final result will not be affected by the order.
Chapter 5

GETSQL – Design

In this chapter we describe the design of our solution and how it integrates the theory of translating and increasing efficiency described in chapter 3 and 4. The SQL generated in the translation will always be dependent on what kind of database layout used. This means that to change database layout the translator has to be rebuilt to some extent.

5.1 Introducing GETSQL

When we designed our translator we wanted it to be as simple as possible to adopt it to new database layouts. We call our solution Generic Edutella query language Translator to SQL, GETSQL.

To separate the parts of the translation independent of the database layout from the parts that are dependent we have divided GETSQL in two layers. They are

- the Logical layer, and
- the Data access layer

In the logical layer we have a representation of the QEL query and an API for it used by the Data access layer. When we create the representation of the query we apply the features described in chapter 4. First we transform the constraining predicates according to what we described in section 4.2. Then we create a join table and calculate the join order we have described in section 4.3.

In the Data access layer we generate the SQL according to the theoretical algorithm described in chapter 3. The structure of the SQL statement generated is independent of the database layer and therefore a major part of the Data
access layer is also independent of the database layout. The parts that are
dependent are the SQL needed for each matching and constraining predicate.

In the next sections we will look closer at the layers.

5.2 Logical layer

The logical layer is used to implement the parts of the translation algorithm
that is not dependent on the database layout used. It is used to keep track on
which order the literals of the QEL query should be handled. It is also used to
transform the query so that the constraining predicates can be directly applied
on the matching predicates. The logical layer is used to produce an abstract
version of the query, that is realised in the data access layer. The logical layer
does not need any changes when a new database layout is to be supported.
However to implement the data access layer we have defined some methods to
make the logical layer readable by the data access layer.

In the logical layer the query is transformed into a tree structure that directly
corresponds to the algorithm described in chapter 3. However, it is first in the
data access layer that the actual algorithm is executed.

5.3 Logical layer API

We have defined an API for accessing the information needed by the data ac-
cess layer to generate correct SQL for this QEL query. This API contains two
interfaces, IRelationRuleWrapper and IJoinEntry.

5.3.1 IRelationRuleWrapper

This Interface contains methods for the data access layer to use when its gen-
erating the translation of the query. It contains the following methods:

- List getParameterVariables()
  Method that returns a list with the parameter variables that form the
  result of this rule/top-level query.

- QueryLiteral
  getSelectedMatchingLiteralForResource(variable)
  Method that returns a query literal that contains the given variable. It is
  used to determine from which relation the values of the variable should be
taken.
• **List getJoinList()**
  Method that returns a list with information about how to join the literals for this rule/top-level query. The list contains instances of the interface IJoinEntry described below.

• **boolean hasUnion()**
  Method to determine if a rule reference is disjunctive and therefore is a union.

• **List getUnionParts()**
  Method that returns a list with instances of IRelationRuleWrappers that are represented by this instance.

• **List getConstrainingLiteralForResource(Resource variable)**
  Method that returns a list with the constraining predicates used to constrain the given variable.

• **boolean isVariableLimited(Resource variable)**
  Method to determine whether a variable is limited or not.

• **boolean canVariableValueBeRDFLiteral(Resource variable)**
  Method to determine if a variables' value can be bound to an RDF literal. This is true if the variable only appears in object position in statements and not the QEL built-in predicate qelnodetype binding it to an RDF resource.

• **boolean canVariableValueBeRDFResource(Resource variable)**
  Method to determine if a variables' value can be bound to an RDF resource. This is true if the variable not have any QEL built-in predicates that require it to be an RDF literal.

### 5.3.2 IJoinEntry

This interface contains methods to make the information stored in the joinlist of the logical layer available to the data access layer. It contains the following methods:

• **QueryLiteral getLiteralToJoin()**
  Method that returns the query literal to add to join.

• **int getJoinType()**
  Method that get an integer representing the type of join to be used. The different types are natural join and left outer join.
• List getJoinVariables()
  Method that returns a list with the variables used to join this join entry to the previously added.

• QueryLiteral getLiteralToJoinToForVariable(Resource variable)
  Method that returns the query literal joined that contains the given variable.

5.4 Data access layer

The data access layer is used to generate SQL for a specific database layout. It needs to be able to read and interpret the information in the logical layer and then translate it to SQL that is applicable for the underlying database layout. When a new database layout is used this layer has to be re-implemented. How this is done is described in the next section.

5.5 Implementing new data access layers

The data access layer consists of two parts. The SQL generator and the result converter. In the next two sections we will describe what they are used for and how to implement them to support new database layouts.

5.5.1 Generating the SQL

This part of the data access layer is responsible for translating the QEL query to a SQL statement that is adequate for the underlying database layout. In GETSQL this is done by reading the stored query from the logical layer using the provided API. The algorithm that needs to be implemented is similar to the theoretical translation algorithm described in chapter 3. How this should be done in general can be described in the following steps:

1. Get the parameter variables from the logical layer and add them to this SQL statements SELECT clause.

2. Go through the join list.
   a) Generate SQL for the non outer join matching predicates and rule references. For the rule references generate a subselect SQL statement for them by using this algorithm from step 1. If there are more than one rule matching a reference use the SQL operator UNION to put them together. Join them together by using the join condition available by the IJoinEntry interface.
b) Generate SQL for the outer join matching predicates and rule references. For each outer join matching predicates a subselect SQL statement must be generated containing SQL for the matching predicate and also SQL for the constraining predicates that constrains its variables. The rule references are handled the same way as in a. Join each SQL statement by using the join condition available by the IJoinEntry interface.

3. Constrain the variables by generating SQL for the non outer join constraining predicates contained by this query/rule.

To implement the algorithm above we have created an abstract class containing methods that are independent of the database layout and some abstract methods that need to be implemented in a subclass for each new database layout supported. We have also designed some interfaces to make the generator easy to implement. In the next sections we will describe the interfaces and the abstract methods.

5.5.2 ISQLGenerator

This interface represents the SQL statement generated to the part of the provider peer responsible for executing the query. It also defines a method to specify a model ID for the RDF model to be queried. The interface looks like this:

- **String getSql (IRelationRuleWrapper relationRuleWrapper)**
  Method that takes an object that implements the IRelationRuleWrapper and returns the SQL statement as a String.

- **public void setProperty(String propertyName, String propertyValue)**
  Method that sets different properties needed in the Data access layer. For example, if the database contains multiple models a model ID can be set with this method.

5.5.3 ISQLParts

This interface represents the FROM and WHERE clauses of the SQL statement. It is used in AbstractSQLGenerator, described below, to generate the SQL statement. The interface looks like this:

- **List getFromParts()**
  Method that return a list with objects implementing the interface ISQLFromPart. This list is then used to generate the FROM clause of the SQL statement.
- **List getWhereParts()**
  Method that return a list with objects implementing the interface ISQLWherePart. This list is then used to generate the WHERE clause of the SQL statement.

### 5.5.4 ISQLFromPart

This interface represents a table instance or a subselect statement in the SQL statements FROM clause. The interface looks like this:

- **int getJoinType()**
  Method that returns an integer that represents what type of join to be used on this object.

- **String getTableSql()**
  Method that returns a string with the SQL that defines this table instance or subselect in the FROM clause.

- **String getJoinCondition()**
  Method that returns the join condition for this object in the SQL statement.

### 5.5.5 ISQLWherePart

This interface represents a condition in the SQL statements WHERE clause. The interface looks like this:

- **String getCondition()**
  Method that returns the condition to be added in the WHERE clause of the SQL statement.

### 5.5.6 AbstractSQLGenerator

This abstract class contains some methods that are independent of the database layout. It also implements the ISQLGenerator interface to expose the SQL statement to the object responsible for generating the result. A subclass of this class needs to be implemented for each new database layout supported. There are some abstract methods to be implemented by the subclass. They are:

- **protected abstract String getSelect(List parameterVariables)**
  Method returns the SELECT clause of the SQL statement for the toplevel query. It takes a list with the result variables of the QEL query.
• protected abstract String getSubSelect()
  Method returns the SELECT clause for the subselect SQL statement for a rule in this query.

• protected ISQLParts getMatchingLiteralSql(IJoinEntry joinEntry)
  Method returns an object implementing the ISQLParts interface described above for a QEL matching literal. It takes an object implementing the IJoinEntry interface.

• protected ISQLParts getVariableConstraintSql(Resource variable, List literalsToConstrain, List constraintingLiterals)
  Method that returns an object implementing the ISQLParts interface described above for the QEL constraining literals. It takes the variable to be constrained and a list containing the QEL matching predicates to be constrained and also a list with the QEL constraining literals.

• protected String getOuterJoinSelect(QueryLiteral q1)
  Method that returns a string with the subselect SQL statement used for an outer join QEL query literal.

• protected String getJoinConditionSql(String q1ToJoin, String q1ToJoinWith, String variableName)
  Method that returns a string with the join condition. It takes the name of the predicate to join and the name predicate already joined and also the name of the common variable.

5.5.7 Converting the result

The data access layer also need methods for generating a QEL resultset from the result generated by the SQL query and then send it back to the Edutella network. The transmission of the QEL resultset is independent of the underlying database layout. This is taken care of by the GETSQLProviderConnection.

However, the SQL resultset is not independent of the underlying database layout. Therefore, a method to translate the SQL resultset to a QEL resultset has to be implemented for each new database layout supported. We have created an interface called IResultConverter. The interface looks like this:

• ResultSet convertResult(java.sql.ResultSet rs, Query qelQuery, GETSQLQueryFormat) throws SQLException
  Method that takes the SQL resultset and returns a QEL resultset.

A class that implement this interface needs to be implemented for each new database layout.
In the next chapter we will look at a specific database layout and the SQL we need to generate for it.
Chapter 6

GETSQL – implementation

In this chapter we describe the database layout that we have made the implementation for and the SQL generated for a QEL query.

6.1 Jena 1.6 – database layout

We chose to implement our GETSQL translator with the database layout used in Jena 1.6. In this section we will describe how the RDF is stored with that database layout. In the next section we will show what the generated SQL looks like for a simple query.

The database layout contains some administration tables which are not interesting to our translation since we only want to get the relevant metadata. Therefore we will only describe the parts that is used directly to store the RDF data.

The Jena 1.6 database layout is statement centric, which means it has a table for storing all RDF statements as triples consisting of subject, predicate and object. This table is called RDF_STATEMENTS and is shown in figure (6.1). However, the RDF_STATEMENTS does not contain the values of the RDF resources and RDF literals. It only contains references to the resources and literals and the values are stored in three other tables. Objects in an RDF statement can be either an RDF resource or an RDF literal. Therefore the RDF_STATEMENTS table contains a column which specifies if the object is a resource or a literal. Furthermore this database layout supports storing several RDF models in it. Therefore, there is a field containing the ID of the model containing the statement.
<table>
<thead>
<tr>
<th>RDF_STATEMENTS</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATEMENT_ID</td>
<td>integer</td>
<td>Unique ID for the statement.</td>
</tr>
<tr>
<td>SUBJECT</td>
<td>integer</td>
<td>ID of the subject resource.</td>
</tr>
<tr>
<td>PREDICATE</td>
<td>integer</td>
<td>ID of the predicate resource.</td>
</tr>
<tr>
<td>OBJECT</td>
<td>integer</td>
<td>ID of the object resource.</td>
</tr>
<tr>
<td>OBJECT_ISLITERAL</td>
<td>integer</td>
<td>Bit to determine if object is literal or resource.</td>
</tr>
<tr>
<td>MODEL</td>
<td>integer</td>
<td>ID of the model containing this statement.</td>
</tr>
<tr>
<td>REIFICATION</td>
<td>integer</td>
<td>Not used.</td>
</tr>
<tr>
<td>IS_REIFIED</td>
<td>integer</td>
<td>Not used.</td>
</tr>
</tbody>
</table>

Figure 6.1: Description of the RDF_STATEMENTS table.

The RDF resources are described in two tables, RDF_RESOURCES and RDF_NAMESPACES, figure (6.2). The RDF_RESOURCE table contains the local name of the RDF resources and a reference to the RDF_NAMESPACES table. In the RDF_NAMESPACES table the URI for the different RDF resource namespaces are stored. These two tables are used to reduce the number of duplicate resources stored in the database.

<table>
<thead>
<tr>
<th>RDF_RESOURCES</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>integer</td>
<td>Unique ID for the resource.</td>
</tr>
<tr>
<td>NAMESPACE</td>
<td>integer</td>
<td>ID of the namespace for this resource.</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>integer</td>
<td>Not used.</td>
</tr>
<tr>
<td>LOCALNAME</td>
<td>string</td>
<td>String containing the localname of the resource.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RDF_NAMESPACES</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>integer</td>
<td>Unique ID for URI.</td>
</tr>
<tr>
<td>URI</td>
<td>string</td>
<td>String containing the URI for this namespace.</td>
</tr>
</tbody>
</table>

Figure 6.2: Description of the RDF_RESOURCES and RDF_NAMESPACES tables.

The values of the RDF literals are stored in a table called RDF_LITERALS, figure (6.3). The RDF_LITERALS table also contain information about language for the RDF literal. Note that this layout does not support the use of typed RDF literals which therefore is ignored in this translation.

<table>
<thead>
<tr>
<th>RDF_LITERALS</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>integer</td>
<td>Unique ID for this RDF literal.</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>string</td>
<td>Language for this RDF literal.</td>
</tr>
<tr>
<td>LITERAL_IDX</td>
<td>string</td>
<td>Part of the RDF literal that can be indexed by the database.</td>
</tr>
<tr>
<td>LITERAL</td>
<td>string</td>
<td>Value of RDF literal if longer than the indexe allow.</td>
</tr>
<tr>
<td>INT_OK</td>
<td>int</td>
<td>Bit to determine if RDF literal is integer.</td>
</tr>
<tr>
<td>INT_LITERAL</td>
<td>integer</td>
<td>Integer value of the RDF literal. If it exists.</td>
</tr>
<tr>
<td>WELLFORMED</td>
<td>integer</td>
<td>Not used.</td>
</tr>
</tbody>
</table>

Figure 6.3: Description of the RDF_LITERALS table.
6.2 The SQL for Jena 1.6 database layout

In the next sections we will describe how the different parts of the SQL is generated by our implementation of the data access layer for a Jena 1.6 database layout with a Postgresql database.

6.3 Renaming

When we generate SQL we use different instances of the same tables. To do that we need to use aliases. To create an alias in Postgresql SQL the keyword AS is used. We also rename the attributes in the SELECT clause that represent the variables of the query. Some of the aliases used are compounds of different parts. The parts are put together with the _ character.

6.4 Matching predicates

Each matching predicate qel:s or qel:member are derived to relations according to previous chapters of this report. For each use of a matching predicate we need to use one instance of the RDF_STATEMENTS table and also instances of the tables containing the values of the RDF resources and RDF Literals. If we have the matching predicate:

\[ qel: s(X, dc:title, Y) \]

The instance of the RDF_STATEMENTS table in the FROM clause is formed like this:

\[
\text{(SELECT}
\text{ }
\text{SUBJECT AS \text{var0},}
\text{ }
\text{\text{0 AS \text{var0}}_\text{ISLITERAL},}
\text{ }
\text{PREDICATE,}
\text{ }
\text{OBJECT AS \text{var1},}
\text{ }
\text{OBJECT_\text{ISLITERAL AS \text{var1}}_\text{ISLITERAL},}
\text{ }
\text{MODEL -}
\text{FROM}
\text{RDF\_STATEMENTS) AS literal0}
\]

We form it as a subselect, note the parenthesis around the select statement, to make it work with the construction we use for the constraining predicates described below. We need the model attribute if the database contains several different RDF models and we want to specify which we want to query. When we name the instance of the RDF_STATEMENTS table we call it literal followed by an index number. So if there was another matching predicate its
RDF_STATEMENTS instance would have been named literal1. We also rename the attributes that represents variables in the SELECT clause. They are named var followed by an index number. If a variable only occurs in the object position of matching predicates and its nodetype is not explicitly constrained. We use the attribute OBJECT_ISLITERAL and rename it with the variables alias and add the suffix ISLITERAL.

In this matching literal the subject and object are variables but the predicate is a constant. This means that we want to match all statements containing the ID of the resource dctitle in the predicate position of RDF_STATEMENTS table. To do this we first need to get the ID by looking it up in the RDF_RESOURCES and RDF_NAMESPACES tables. When we name these table instances the alias is formed three parts put together:

- What the name is of the RDF_STATEMENTS table instance.
- Which of the tables it is. RDF_RESOURCES abbreviated to RES, RDF_NAMESPACES to NS and RDF_LITERALS to LIT
- What position of the statement table it represent, SUBJECT, PREDICATE or OBJECT

We add the following statements to the FROM clause:

```
RDF_NAMESPACES AS literal0_NS_PREDICATE CROSS JOIN
RDF_RESOURCES AS literal0_RES_PREDICATE
```

The CROSS JOIN is a Postgresql specific command to make sure the execution of the query is done in the specified order. Typically one would use a ',' between the tables in the FROM clause. But then the built-in query planner will be used on the query and we want to avoid that.

In the WHERE clause we join the literal0 tables PREDICATE with the literal0_RES_PREDICATE tables attribute ID. Then specify which localname and namespace we are interested in and we join the two tables together on the namespace ID. In our example we have the localname, title and the namespace is, http://purl.org/dc/elements/1.1/. The SQL will look like this:

```
literal0_PREDICATE = literal0_RES_PREDICATE.ID AND
literal0_RES_PREDICATE.NAMESPACE = literal0_NS_PREDICATE.ID AND
literal0_RES_PREDICATE.LOCALNAME = 'title' AND
literal0_NS_PREDICATE.URI = 'http://purl.org/dc/elements/1.1/'
```
6.5 Constraining variables

Since the values of RDF Literals and RDF Resources are stored in separate tables we cannot directly constrain the variables in the WHERE clause of the SQL statement. Instead we have to constrain against the resources and literals in their respective tables according to their IDs. We also need information about if the value is a resource or a literal. To do this we create a subselect statement that selects the IDs of the wanted resources and/or literals and a 1 if the value is a literal and 0 if the value is a resource. This subselect is added in the FROM clause of the SQL statement and then it is joined with the relevant matching predicates in the WHERE clause.

For example, consider the last query executed with a constraint on the object:

\[ ? = \text{qel} : s(X, dc : \text{title}, Y), \text{qel} : \text{stringValue}(Y, 'Edutella'). \]

The qel:stringValue will result in a subselect that looks like this:

\[
\begin{align*}
\text{(SELECT} & \\
& \text{LIT}_\text{var1}.ID \text{ AS } \text{var1}, \\
& \text{1 AS } \text{var1}_\text{ISLITERAL} \\
\text{FROM} & \\
& \text{RDF LITERALS LIT}_\text{var1} \\
\text{WHERE} & \\
& \text{LIT}_\text{var1}.LITERAL_\text{IDX} = 'Edutella') \text{ AS CONSTRAINTS}_\text{var1} \\
\end{align*}
\]

The subselect is named CONSTRAINTS_var1 and joined with the matching predicate in the WHERE clause like:

\[
\text{literal0.var1 = CONSTRAINTS_var1.var1 AND} \\
\text{literal0.var1}_\text{ISLITERAL} = \text{CONSTRAINTS_var1.var1}_\text{ISLITERAL}
\]

If a variable is constrained by several constraining predicates we add the constraints to the subselect.

6.6 Generating query results

The matching predicates and the rule references get the IDs of the RDF resources and RDF literals stored in the RDF_STATEMENTS table. We are interested in the actual values and therefore we need to get them from the tables containing the values, RDF_RESOURCES, RDF_NAMESPACES and RDF_LITERALS.

Consider the very simple query:

In this query we have two variables X and Y. Since X is in the subject position it can only be of type RDF resource. Y, on the other hand can be of type RDF resource or RDF literal. In the SQL query generated in our implementation X will be named var0 and Y will be named var1, qel:s predicate will be named literal0. For var0 we need to add an instance each of the tables RDF_RESOURCES and RDF_NAMESPACES. For var1 we need to add an instance of the table RDF_LITERALS and an instance of each RDF_RESOURCES and RDF_NAMESPACES.

\[
\begin{align*}
&\text{LEFT OUTER JOIN RDF LITERALS AS LIT\_{var1} ON literal0.vari = LIT\_{var1}.ID} \\
&\text{LEFT OUTER JOIN RDF RESOURCES AS RES\_{var0} ON literal0.vari = RES\_{var0}.ID} \\
&\text{LEFT OUTER JOIN RDF RESOURCES AS RES\_{var1} ON literal0.vari = RES\_{var1}.ID} \\
&\text{LEFT OUTER JOIN RDF NAMESPACES AS NS\_{var0} ON RES\_{var0}.NAMESPACE = NS\_{var0}.ID} \\
&\text{LEFT OUTER JOIN RDF NAMESPACES AS NS\_{var1} ON RES\_{var1}.NAMESPACE = NS\_{var1}.ID}
\end{align*}
\]

Figure 6.4: Example of the variable join.

We add the instances, see figure (6.4) we need at the end of the FROM clause. To be sure to get the correct values we use the LEFT OUTER JOIN statement to add the instances. If we use a natural join we risk losing tuples that should be in the result. In principle we do not need to use the LEFT OUTER JOIN construction for var0 in figure (6.4) since it will always exist.

In figure (6.5) we show how the values are retrieved in the SELECT clause.

\[
\begin{align*}
&\text{NS\_{var0}.URI|RES\_{var0}.LOCALNAME AS var0_{RESOURCE}} \\
&\text{NS\_{var1}.URI|RES\_{var1}.LOCALNAME AS var1_{RESOURCE}} \\
&\text{LIT\_{var1}.LITERAL_ID AS var1_{LITERAL}} \\
&\text{LIT\_{var1}.LANGUAGE AS var1_{LANGUAGE}} \\
&\text{literal0.vari\_ISLITERAL AS var1\_ISLITERAL}
\end{align*}
\]

Figure 6.5: Example on how to get the values.

To get the resource values we need we concatenate the namespace URI with the resource localname. The concatenation of strings in Postgresql is done with \texttt{||}. The literal value is taken from the RDF LITERALS instance. We name the variable values with the variable alias followed by the RESOURCE suffix for resources and LITERAL suffix for literals. For the literal we also get the language field. It is named with the variable alias followed by the suffix, LANGUAGE. To know if the value for var1 should be an RDF resource or RDF literal we add the var1\_ISLITERAL attribute. If the value is 1 then its a literal and if its 0 the value is a resource.
6.7 Rule references

The difference with a rule reference and the top-level query is that we do not need to get the actual values from the database yet, if they are visible at the top level query they will be retrieved then. First we only need to get the IDs of the RDF resources and RDF literals that the rule matches. Therefore there will be a difference in the SELECT clause of the subselect generated for a rule reference and the variable join of the top-level query is not needed. Consider the query:

\[ \text{title}(X,Y) : -\text{qel} : s(X, \text{dc:} \text{title}, Y). \]
\[ ? - \text{qel} : s(X, \text{rdf:} \text{type, examplet} \text{ype : Book), title}(X,Y). \]

First the \text{qel:s} predicate generates the SQL we described earlier. The rule reference generates a subselect statement where the SELECT clause will look like this:

\begin{verbatim}
SELECT
  literal0.var0 AS var0,
  0 AS var0_1SLITERAL,
  literal0.var1 AS var1,
  literal0.var1_1SLITERAL AS var1_1SLITERAL
\end{verbatim}

The rest of the subselect statement follows the same structure as described above. The whole subselect statement is named in the same way as the \text{qel:s} predicate, in this query it will be named literal1 and it will be joined with literal0 in the WHERE clause.

6.8 Outer join

To generate SQL for outer join matching predicates we need to think differently. Consider the query:

\[ ? - \text{qel} : s(X, \text{rdf:} \text{type, examplet} \text{ype : Book), qel} : s^*(X, \text{dc:} \text{title}, Y). \]

In this query we want all resources of type examplet\text{ype:Book}. We also want their titles if they exist. To do this we need to use the \text{LEFT OUTER JOIN} operator for the second matching predicate. As we discussed earlier we need an instance of the \text{RDF\_STATEMENTS} table for each matching predicate and also instances of the \text{RDF\_RESOURCES}, \text{RDF\_NAMESPACE\_S} and \text{RDF\_LITERALS}.

Not to lose any tuples in the result we generate a subselect statement for the outer join matching predicate containing the join between the
RDF STATEMENTS table instance and the tables needed to constrain it. Within this subselect statement the SQL is generated as described earlier. The SELECT clause is generated in the same way as for the rule references. The subselect statement is then joined together with SQL generated for the non outer join matching predicate using the LEFT OUTER JOIN operator.

6.9 SQL translation example

We will give an example of a whole translation of a simple QEL query. We use the query that we used in a previous section:

\[ ? - qel : s(X, dc : title,Y), qel : stringValue(Y,'Edutella'). \]

This query will be translated into the following SQL:

```
SELECT
    NS_var0.URI||RES_var0.LOCALNAME AS var0_RESOURCE,
    NS_var1.URI||RES_var1.LOCALNAME AS var1RESOURCE,
    LIT_var1.LITERAL_INDEX AS var1_LITERAL,
    LIT_var1.LANGUAGE AS var1_LANGUAGE,
    literal0.var1_1LITERAL AS var1_1LITERAL
FROM
    RDF_NAMESPACES AS literal0_NS_PREDICATE CROSS JOIN
    RDF_RESOURCES AS literal0_RES_PREDICATE CROSS JOIN
    (SELECT
        SUBJECT AS var0,
        0 AS var0_1LITERAL,
        PREDICATE,
        OBJECT AS var1,
        OBJECT_1LITERAL AS var1_1LITERAL,
        MODEL
    FROM
        RDF_STATEMENTS) AS literal0 CROSS JOIN
    (SELECT
        LIT_var1.ID AS var1,
        1 AS var1_1LITERAL
    FROM
        RDF_LITERALS AS LIT_var1
    WHERE
        LIT_var1.LITERAL_INDEX = 'Edutella'
    ) AS CONSTRAINTS_var1
LEFT OUTER JOIN RDF_LITERALS AS LIT_var1 ON literal0.var0 = LIT_var1.ID
LEFT OUTER JOIN RDF_RESOURCES AS RES_var0 ON literal0.var0 = RES_var0.ID
LEFT OUTER JOIN RDF_RESOURCES AS RES_var1 ON literal0.var1 = RES_var1.ID
LEFT OUTER JOIN RDF_NAMESPACES AS NS_var0 ON RES_var0.NAMESPACE = NS_var0.ID
```

38
LEFT OUTER JOIN RDF_NAMESPACES AS NS_var1 ON RES_var1.NAMESPACE = NS_var1.ID
WHERE
    literal0.PREDICATE = literal0_RES_PREDICATE.ID AND
    literal0_RES_PREDICATE.NAMESPACE = literal0_NS_PREDICATE.ID AND
    literal0_RES_PREDICATE.LOCALNAME = 'title' AND
    literal0_NS_PREDICATE.URI = 'http://purl.org/dc/elements/1.1/' AND
    literal0.MODEL = 1 AND
    literal0_var1 = CONSTRAINTS_var1.var1 AND
    literal0_var1__ISLITERAL = CONSTRAINTS_var1.var1__ISLITERAL AND
    literal0_var0__ISLITERAL = 0 AND
    literal0_var1__ISLITERAL = 1
Chapter 7

Conclusions and future work

In this chapter we will summarize the conclusions we draw from this thesis. We also outline some of the future work that can be done to further investigate how to use QEL with RDF stored in relational databases.

7.1 Conclusions

We divided this thesis work in three problems

- translation of QEL to SQL, on a theoretical level,
- provider efficiency,
- database layout dependence.

In the next sections we will go through the results for each part of the problem.

7.1.1 Theoretical translation

We found work done to translate Datalog questions to relational algebra. We applied it to QEL that is a subset of Datalog and we were able to translate all of the built in predicates of QEL. We were also able to handle both conjunctive and disjunctive rules. The outer join feature defined in QEL is not part of Datalog, however since it is a part of SQL this was also something we were able to translate. The limitation we found was recursive rules. To be able to support recursion the translator has to support the latest version of SQL, SQL3 (SQL99). However, not many relational database implementations has support for SQL3 so we left this unsolved.
7.1.2 Efficiency

We have also tried to find what can be done in making the translator give an efficient translation of the QEL query. Efficiency means that the consumer peer will get a reply on the query fast enough. We found two ways to make the query more efficient. First we use the limiting predicates earlier in the query structure. For example if we have rule that matches all resources of the type exampleType:Book and another rule that uses this rule with the limitation added to a word in the title. In this example we transform the query and move the limitation to the rule that gets all the books. We have also seen that the join order of the relations is important and we have given a suggestion on how to determine what order that gives the most efficient query execution. However, there is work to be done to improve how the order is determined.

7.1.3 Database layout independence

The last issue we have looked at is building a translator that is not so dependent on the database layout used. We have suggested a way to divide the translation into a logical and data access layer and created the GETSQL package. The logical layer contains a representation of the QEL query rewritten with the two suggestions we made to make the query more efficient. We also defined an API to make the representation of the query readable from the data access layer. The data access layer generates the SQL query to get the RDF from the underlying relational database and converts the result set to a QEL result set to be sent back to the consumer peer. These two layers helps in implementing new GETSQL peers for different database layouts.

7.1.4 Results

To try the theories we implemented GETSQL to query an RDF model stored in a relational database, using the PostgreSQL database engine and the database layout of Jena 1.6 as relational model for our SQL. Our implementation supports the use of all built in predicates in QEL, it supports the use of the outer join feature and conjunctive and disjunctive rules. We tested the implementations efficiency and were able to see a difference by using our suggestion to make the queries more efficient. The two layer structure we use made it possible to implement GETSQL for the relational model used by Jena 1.6 and we are confident it will make support for other database layouts possible.
7.2 Future work

The translation of QEL to SQL is an important issue for the Edutella project and this thesis does not complete the work. Further investigation is needed concerning efficiency. It would also be of interest to make implementations with other database layouts to test if the GETSQL really is good enough to support any kind of database layout for storing RDF models. The implementation we have done is with the statement centric database layout of Jena 1.6. It would be of interest to try to make an implementation with the Sesame schema based database layout.

There is also an interest in investigating how to transform QEL queries according to the metadata model used. Different metadata models can use different vocabularies that overlap. Hence some mapping should take place so that you can get matches in the vocabulary when searching the other.

There is also structural differences, for example simple and qualified Dublin Core. Dublin Core, DC, is a set of properties to model metadata. Simple and qualified use the same properties but for qualified DC the structure is deeper. One example is the property title. In simple DC the title is pointing directly to an RDF Literal containing the title of the resource while the same property in qualified DC could point to a structure containing several alternative titles. Still there is large overlapping semantics that should not be ignored.

To create this mapping some kind of translation of the query could be performed on the provider side but there are some issues that need to be considered. For example how much of the mapping should be visible. If a query contains a constant RDF resource that is among the resources that need to be mapped and the same resource is requested back. Then the peer sending the query may not accept the result as correct.

Another issue that needs to be explored is the RDF Schemes, RDFS. In RDFS metadata can be modelled in an object oriented way. A property can be a subclass of a more general property. This means that a property, for example car, can have sub properties, for example Volvo or SAAB. The problem is how to handle queries that wants all resources which has a property that is a super property. If resources that has the sub property of the specified property also should be matched.

In this report we have handled the QEL queries. Lately an extension called Retrieval been discussed. Retrieval should be used to retrieve information for a given resource, retrieval of all properties and their values for that resource.
For example Retrieval could be used to get the anonymous closure for a resource, i.e.
get the whole graph around a resource that ends with an non anonymous
RDF resource or an RDF Literal.
Bibliography


