An Effective SPARQL Support over Relational Databases

Jing Lu[1,2], Feng Cao[2], Li Ma[2], Yong Yu[1], and Yue Pan[2]

[1] Shanghai Jiao Tong University, Shanghai 200030, China
{robert_lu,yyu}@cs.sjtu.edu.cn
[2] IBM China Research Laboratory, Beijing 100094, China
{caofeng,malli,panyue}@cn.ibm.com

Abstract. Supporting SPARQL queries over relational databases becomes an active topic recently. However, it has not received enough consideration when SPARQL queries include restrictions on values (i.e filter expressions), whereas such a scenario is very common in real life applications. Challenges to solve this problem come from two aspects, (1) databases aspect. In order to fully utilize the well-developed SQL optimization engine, the generated SQL query is desired to be a single statement. (2) SPARQL query aspect. A practical SPARQL query often embeds several filters, which require comparisons between RDF results of different types. The type of RDF resources needs to be dynamically determined in the translation. In this paper, we propose an effective approach to support SPARQL queries over relational databases, with the above challenges in mind. To ensure the seamless translation, a novel facet-based scheme is designed to handle filter expressions. Optimization strategies are proposed to reduce the complexity of the generated SQL query. Experimental results confirm the effectiveness of our proposed techniques.

1 Introduction

The Resource Description Framework (RDF) data is often persisted in relational DBMSs by triple stores. The RDF data is represented as a collection of triples <subject, predicate, object>. SPARQL [1] is W3C's recommendation for RDF query. Based on matching graph patterns, it provides lots of facilities to extract RDF subgraphs. Given a data source D, a SPARQL query consists of a pattern which is matched against D and the values obtained from this matching.

As one of the building blocks of SPARQL query, a SPARQL filter expression restricts the graph pattern matching solutions. It provides the following functionalities: 1) The ability to restrict the value of literals and arithmetic expressions; 2) The ability to preprocess the RDF data by built-in functions. For example, `isIRI(term)` returns true if *term* is an IRI. Therefore, in real applications, users often submit various SPARQL queries including filter expressions to express their specific requirements on results. For example, a campus analyst may issue the following query in Figure 1, which requests IRI and optional

```
SELECT ?person ?tel WHERE {
  ?person rdf:type bm:GraduateStudent
  {
    { ?person bm:like ?interest } UNION
    { ?person bm:love ?interest }
  } .
  OPTIONAL { ?person bm:telephone ?tel } .
  ?person bm:age ?age .
  FILTER ( ?age < 25 &&
           REGEX(STR(?interest), "Ball$") )
}
```

Fig. 1. An example of SPARQL query

telephone number of all graduated students with age less than 25 and have an interest of ball sports, in a UOBM [2] ontology (The bm:age is an extended property of the UOBM).

Previous methods on supporting SPARQL over relational databases [3,4,5] mainly supported basic SPARQL query patterns. They either ignored filter expressions due to complexity or adopted memory-based method for evaluation. For example, in Sesame [6,3] or Jena2 [7], a SPARQL query with a global filter expression is first translated into a SQL without filter expressions, and then the results are further filtered in Java program. If there are nested filter expressions in an optional query pattern, only part of the query pattern is evaluated, then the results are filtered in the program and further SQLs are issued one by one. Therefore, the evaluation of filter expressions is a time consuming operation due to the expensive I/O cost. How to efficiently and scalably process SPARQL query with filter expressions is still an open question [4,5].

A single SQL query translated from a SPARQL query can fully utilize the well-developed SQL optimization engine. This is because existing DBMS optimizers usually make a query plan based on one SQL statement. In addition, the generated single SQL query can be directly embedded into other SQL queries as a sub-query, which can be optimized by DBMSs as a whole. A seamless integration of SPARQL queries with SQL queries is attractive to real applications. However, the requirement of a single SQL statement is a strong constraint to the translation.

Mutually-interwaved filter expressions make the translation of SPARQL difficult. Filter expressions often consist of multiple functions and operators, and a filter operator may have different behaviors on different operands. When the operands are variables or complex expressions, it is usually hard to determine their actual type. Furthermore, variables may be bound to different kinds of literals in different results. That is, the type of RDF resources needs to be dynamically determined in the translation. As Figure 2, a user wants to query the students whose interest and major are the same. However, the classification information in the RDF triple data is string in some cases and the code of integer in other cases. The comparison function over string and integer literal are different

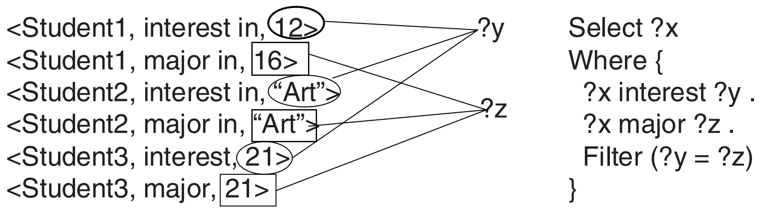

Fig. 2. An example of dynamic data type of RDF resource

in the SPARQL standard. Therefore, the generated SQL should be able to adopt different comparison methods for different types of operands.

To address the above problem, we present a complete translation process from a SPARQL query to a SQL query. The main contributions of the paper are as follows.

– We propose an effective method to translate a complete SPARQL query into a single SQL, so that the generated SQL can be directly embedded as a sub-query into other SQL queries. By that way, SPARQL queries can be seamlessly integrated with SQL queries.
– We present a novel idea of facet-based scheme to translate filter expressions into SQL statements and support most of SPARQL features, such as nested filters in optional patterns.
– We propose two optimization strategies for SQL generation, and perform experiments on benchmark data. The experimental results show the effectiveness of our method.

1.1 Preliminaries: SPARQL Pattern Tree

The generated SQL may be different for different database schemas. To simplify the discussion, we assume that all the triples are stored in a triple table, in which internal IDs are used instead of IRIs or literal strings. The IRI and literal strings are stored in two separate tables. Most known RDF stores adopt such a schema.

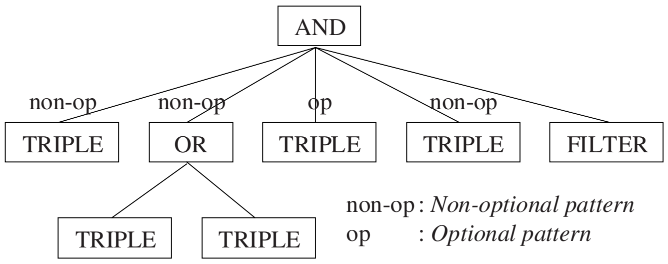

Fig. 3. An example SPARQL pattern tree

Generally, the graph pattern part in a SPARQL query can be expressed as a SPARQL pattern tree. The tree shows the backbone of the SPARQL query and is used in the translation. Similar pattern trees are used in [4]. Figure 3 shows the SPARQL pattern tree of the query in Figure 1. There are four possible types of nodes in the pattern tree:

(1) **AND node**, which corresponds to conjunction of graph patterns in a SPARQL query. In our approach, an AND node can have multiple child nodes. That is, consecutive nested AND patterns are included in one AND node as child nodes. Each child node has a flag indicating whether the node is optional or not. In this way, the optional patterns in SPARQL queries are also covered by AND nodes.

(2) **OR node**, which corresponds to a UNION pattern in a SPARQL query. Similar to the AND pattern, nested UNION patterns are represented by one OR node.

(3) **TRIPLE node**, which represents a graph pattern with a single triple. The subject, predicate and object in the triple could be constants or variables.

(4) **FILTER node**, which represents a filter expression in a SPARQL query. A FILTER node is always connected to an AND node as a special child node. In bottom-up semantics of SPARQL, only variables appearing in corresponding AND pattern can be used in the filter.

AND node, OR node and TRIPLE node represent graph patterns in the RDF graph. We call them pattern nodes.

The remainder of this paper is organized as follows. Section 2 describes the problem of filter expression translation. Section 3 presents our translation method. Section 4 proposes two optimization strategies. Section 5 presents an example of our translation. Section 6 provides experimental analysis. In section 7, we briefly survey the related work. Section 8 concludes this paper.

2 Problem Statement

A SPARQL filter can not be translated into a SQL expression straightforwardly. The result of a filter expression or a sub-expression is an RDF object, which could be a literal, an IRI reference or an error (e.g. when applying functions on unbound values). However, the result of a SQL expression is always a primitive value, such as a string, a double or an integer. Taking literals as an example, we know that a literal could have a lexical form, an optional language tag and an optional datatype. It is always hard to describe a literal object by a primitive value.

Fortunately, we found that usually only a primitive part of an RDF object is used in a function or operator. Thus, we define "facets" of RDF objects which facilitates the translation from a SPARQL filter to a SQL expression.

With the concept of facet, the problem of filter translation changes from "What is the SQL for a filter expression" to "What is the SQL for the Boolean Facet of a filter expression".

2.1 Facet of an RDF Object

A facet of an RDF object is somewhat like a view, or a data field of an object. The value of each facet is always a primitive value of a specific type. We define the following facets:

- *IRI Facet*, which is the full IRI string of an IRI reference. The primitive datatype is string. For literals, this facet is not available.
- *Lexical Facet*, which is the lexical form of a literal. The primitive datatype is string. This facet is only available for literals.
- *Language Facet*, which is the language tag of a literal. The primitive datatype is string. This facet is only available for literals. If the literal is a typed literal or a plain literal without a language tag, the value is an empty string. This is consistent with the definition of the built-in SPARQL function "`Lang`" .
- *Datatype Facet*, which is the full IRI of the datatype of a typed literal. The primitive datatype is string. This facet is only available for literals. For plain literals, the value is a `NULL`.
- *Numeric Facet*, which is the numeric value of a numeric literal. The primitive datatype is double. This facet is only available for typed literals with datatype xsd:float, xsd:double, xsd:decimal or a sub-type of xsd: decimal.
- *Boolean Facet*, which is the boolean value of a boolean literal. Boolean Facet is translated into SQL predicates, such as "`a=b`" or "`a IS NULL`". This facet is only available for typed literals with datatype xsd:boolean.
- *Date time Facet*, which is the date time value of a date time literal. In implementation, we map a date time value into a 64-bit integer, so that the nature time order keeps in this mapping. Thus, the primitive datatype is 64-bit integer. This facet is only available for typed literals with datatype xsd:dateTime.
- *ID Facet*, which is the internal ID of the RDF object. The primitive datatype is integer. As a constant or a calculated result is not required to have a corresponding internal ID, this facet is only available for variables.

IRI, Lexical, Language and Datatype Facets express natural attributes of RDF objects. Numeric, Boolean and Date time Facets express typed literals in corresponding native data types of databases, which facilitates the translation of filter expressions. ID Facet is used to check whether a variable is bound or not.

3 SPARQL to SQL Translation

In this section, we discuss in detail our facet-based SPARQL to SQL translation approach. We refer to it as the FSparql2Sql. The FSparql2Sql consists of two parts: translation of pattern nodes and translation of filter expressions.

3.1 Translation of Pattern Nodes

Similar to the approach in [4], each pattern node is translated into a SQL sub-query. For each variable in the pattern, there is a corresponding column in the query result, which contains the ID of the candidate result. If a variable is unbound, a `NULL` is returned in that column instead.

- A TRIPLE node is translated into a simple `SELECT` query on the triple table. If there are constants in the triple pattern, a corresponding `WHERE` clause is added to the query. Otherwise, the columns are renamed to the corresponding variable names. When one variable appears multiple times in the triple pattern, an equivalent constraint of these columns should be added to the `WHERE` clause. For example, a triple pattern:

  ```
  {?person bm:isFriendOf ?person}
  ```

  is translated into:

  ```
  SELECT subject AS person FROM triple
      WHERE predicate = pID and subject = object
  ```

  Here, the pID stands for the ID the IRI `<bm:isFriendOf>`.
- An AND node is translated into a query on consequent joins of the sub-queries from its child pattern nodes.

  If a child node is optional, a left join is used instead of an inner join. When all the child nodes are optional, a dummy table should be appended as the first table in the joined table list, so that other sub queries can be left joined to this table. The dummy table is the table containing only one line and one column.

  In addition, we keep tracing each variable on whether it could be possibly unbound. The different states will affect the join conditions used. Also, the `COALESCE` function might be used to combine results from multiple queries.

  Finally, if there are child FILTER nodes, they are translated into SQL expressions and added to the `WHERE` clause of the query.
- An OR node is translated into a `UNION` of the sub-queries. If the variable sets of the sub-queries do not match each other, dummy columns containing constant `NULL` should be added to the sub-queries.

3.2 Translation of Filter Expressions

We support the primary operators and functions of filter expressions, which includes:

- (1) IRI constants and literal constants;
- (2) named variables;
- (3) calculation operators (`+`, `-`, `*`, `/`);
- (4) comparison operators (`=`, `!=`, `>`, `<`, `>=`, `<=`);
- (5) logical operators (`&&`, `||`, `!`);
- (6) built-in functions (`bound`, `isIRI`, `isLiteral`, `datatype`, `lang`, `str`, `regex`).

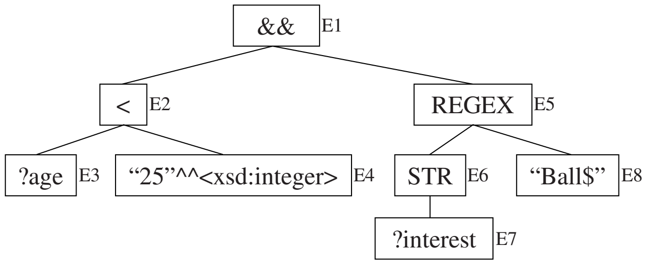

Fig. 4. A filter expression tree

A SPARQL filter expression can be parsed into a filter expression tree. Non-leaf nodes are operators or functions, while leaf nodes are constants or variables. Figure 4 shows the filter expression tree of the `FILTER` clause in Figure 1.

By a filter expression tree, the goal of the translation is to get the SQL representing the Boolean Facet of the root node. For a specific facet of an expression node, its translation result is recursively constructed from generating SQLs of its child nodes.

In the following, we adopt "`X.Lexical`" to represent the generated SQL for the Lexical Facet of "X". Similarly, "`X.ID`" represents the SQL for the ID Facet of "X", and so on.

Literal Constants and IRI Constants. The translation of constants in a filter expression is rather straightforward. The translation results for all kinds of supported facets are just SQL constants as described in the facet definitions. For literal constants, the lexical, language and datatype facet is always available. For typed literals, the numeric, boolean and date time facet might also be available. For IRI reference constants, the only available facet is the IRI facet.

Variables. As internal IDs are used to encode IRIs and literals, the IRI table and literal table should be used to get the actual IRI or literal value of the variable.

In order to effectively translate the Numeric, Boolean and Date Time Facets of variables, we suggest to three additional tables in the database schema, which map the literal IDs to these facet values, respectively.

Therefore, if the SQL for a specific facet of a variable is requested, the corresponding table is added (using left join) to the SQL query, and the corresponding column in that table is provided as the translation result.

In order to avoid unnecessary join operations, during the translation of graph pattern nodes, we keep tracking the possible binding value of each variable. For example, if a variable appears in the subject or predicate position of a triple, the value of the variable cannot be a literal. Thus, if the SQL for a literal related facet is requested, we can directly return a `NULL`.

Comparison Operators. The translation of comparison operators is the most complex in all operators. Because the non-equality operators (`>`, `<`, `>=` and `<=`)

can be used between numeric literals, simple literals, string literals, boolean literals or date time literals, while the equality operators (= and !=) can be further used between all kinds of RDF terms. According to SPARQL specifications, different comparison methods should be used for different types of operands.

CASE Expression. It is often hard to determine the types of operands, especially when the operands are complex expressions. The types of operands need to be bound dynamically. To deal with this problem, we choose to use SQL *CASE Expression* in the translation. A *CASE Expression* often consists of several `WHEN` clauses, in order to provide different results for different cases. The `CASE...WHEN` statement is defined in ANSI SQL-92 standard which is widely supported by many existing DBMSs.

We use the Lexical Facet of operator ">" as an example. For each facet available in both operands, we add a corresponding `WHEN` clause to the `CASE` expression.

1) If the Numeric Facet is available, add

```
WHEN X.Numeric > Y.Numeric THEN ‘true’
WHEN X.Numeric <= Y.Numeric THEN ‘false’
```

2) If the DateTime Facet is available, add

```
WHEN X.DateTime > Y.DateTime THEN ‘true’
WHEN X.DateTime <= Y.DateTime THEN ‘false’
```

3) If the Boolean Facet is available, add

```
WHEN X.Boolean AND NOT Y.Boolean THEN ‘true’
WHEN NOT X.Boolean OR Y.Boolean THEN ‘false’
```

4) If the Lexical Facet is available, add

```
WHEN X.Language = ‘’ AND Y.Language = ‘’ THEN
  CASE WHEN X.Datatype IS NULL
        AND Y.Datatype IS NULL THEN
    CASE WHEN X.Lexical > Y.Lexical THEN ‘true
    WHEN X.Lexical <= Y.Lexical THEN ‘false’ END
  WHEN X.Datatype = xsd:string
   AND Y.Datatype = xsd:string THEN
    CASE WHEN X.Lexical > Y.Lexical THEN ‘true’
    WHEN X.Lexical <= Y.Lexical THEN ‘false’ END
  END
```

** The xsd:string stands for its full URI SQL string.*

After the above steps, we obtain the final translation result, which is probably a complex `CASE` expression. We will discuss how to optimize it in Section 4.

Built-in Functions. We can see that all these built-in functions always return an IRI or a literal with a fixed type as the result (except for error results). For example, the `bound`, `isIRI`, `isLiteral` and `regex` functions always return a boolean typed literal. The `lang` and `str` functions always return a plain literal with no language tag. The `datatype` function always returns an IRI reference.

The translation for Language and Datatype Facet is simply a string constant decided by the function or operator. For other facets, when error happens, it is ensured that a `NULL` is returned instead of the normal results. In order to reduce the complexity of the translation, a constant is always returned, no matter error happens or not. Thus, only the lexical(or boolean) facet is used in the translation to determine if an expression gives an error result or not. We take `bound` functions as an example below.

The `bound` functions always return a boolean typed literal. As described above, the language facet and datatype facet will always return a constant. The boolean facet is translated to an "`IS NULL`" predicate on the ID facet, IRI facet or lexical facet of the operand expression, respectively. In some rare cases, the lexical facet of these functions is required. The result of the lexical facet should be the canonical representation of the boolean values, that is, string "`true`" or "`false`". In this case we have to use a CASE expression over the "`IS NULL`" predicate to give the string result.

Calculation Operators. The calculation operators include `+`, `-`, `*` and `/`. These operators may only be used between numeric literals and always gives a numeric literal as the result. As described above, we always use double as the datatype in our translation for numeric values. Thus, for the numeric facet, the translation result is simply the calculation operator over the numeric facet of the operands. When the lexical facet is required, we have to use a SQL function to convert the numeric result into a string. Note that the numeric-to-string conversion function is not contained in SQL standard and thus may various between different databases.

Logical Operators. The logical operators include `\&\&`, `||` and `!`. These operators may only be used between boolean literals and always returns a boolean literal as the result. The language and datatype facet will always give a constant as the result. For the boolean facet, the translation result is simply the corresponding SQL logical operator (AND, OR or NOT) over the boolean facet of the operands. If the lexical facet is required, we have to use an additional CASE expression to change the result of the SQL predicate into a string "`true`" or "`false`". The SQL will look like this. (`E.boolean` represents the translate result of the boolean facet of the whole expression.)

```
CASE WHEN E.boolean THEN ‘true’
     WHEN NOT E.boolean THEN ‘false’
END
```

Here, a second WHEN clause is used instead of the ELSE clause because when the `E.boolean` returns an UNKNOWN as the result, the result of the lexical facet should be an NULL instead of the "`false`". Furthermore, the SPARQL

Table 1. logical-AND and logical-OR truth table

A	B	A || B	A && B	A	B	A || B	A && B
T	T	T	T	E	T	T	E
T	F	T	F	F	E	E	F
F	T	T	F	E	F	E	F
F	F	F	F	E	E	E	E
T	E	T	E				

Table 2. Translation of build-in functions

Function	Generated SQLs For Different Facets				
	Boolean	**Lexical**	**Lang.**	**Datatype**	**IRI**
bound(X)	`X.ID IS NOT NULL`	`CASE WHEN X.ID IS NOT NULL THEN 'true' ELSE 'false' END`	An empty string.	The IRI of xsd:boolean	(N/A)
isIRI(X)	`X.IRI IS NOT NULL`	`CASE WHEN X.IRI IS NOT NULL THEN 'true' ELSE 'false' END`	An empty string.	The IRI of xsd:boolean	(N/A)
isLiteral(X)	`X.Lexical IS NOT NULL`	`CASE WHEN X.Lexical IS NOT NULL THEN 'true' ELSE 'false' END`	An empty string.	The IRI of xsd:boolean	(N/A)
datatype(X)	(N/A)	(N/A)	(N/A)	(N/A)	`X.Datatype`
lang(X)	(N/A)	`X.Language`	An empty string.	A NULL constant.	(N/A)
str(X)	(N/A)	`COALESCE(X.IRI, X.Lexical)`	An empty string.	A NULL constant.	(N/A)
regex(X, pattern)	`X.Lexical LIKE likePattern` (limited by the expressiveness of LIKE operator)	`CASE WHEN X.Lexical LIKE likePattern THEN 'true' WHEN X.Lexical NOT LIKE likePattern THEN 'false' END`	An empty string.	The IRI of xsd:boolean	(N/A)

* `X.Lexical` represents the generated SQL for the Lexical Facet of "X";
* `X.ID` represents the SQL for the ID Facet of "X", and so on.

logical-AND and logical-OR truth table for true (T), false (F), and error (E) [1] is as following Table 1, which just corresponds with the semantic of SQL logical-AND and logical-OR.

Tables 2 and 3 show the translations of built-in functions, calculation operators and logical operators on various facets.

Table 3. Translation of Calculation Operators and Logical Operators

Ops	Generated SQLs For Different Facets				
	Numeric	**Boolean**	**Lexical**	**Lang.**	**Datatype**
`X` *op* `Y` (*op* is `+`, `-`, `*` or `/`)	`X.Numeric` *op* `Y.Numeric`	(N/A)	`CHAR(X.Numeric` *op* `Y.Numeric)` The number-to-string conversion function may be different in various databases.	An empty string.	The IRI of xsd:double.
`X` *op* `Y` (*op* is `&&` or `||`)	(N/A)	`X.Boolean AND Y.Boolean` for `&&` `X.Boolean OR Y.Boolean` for `||`	`CASE WHEN X.Boolean AND Y.Boolean THEN 'true' WHEN NOT (X.Boolean AND Y.Boolean) THEN 'false' END` For `||`, replace the "`AND`" with "`OR`" in the SQL.	An empty string.	The IRI of xsd:boolean.
`! X`	(N/A)	`NOT X.Boolean`	`CASE WHEN X.Boolean THEN 'false' WHEN NOT X.Boolean THEN 'true' END`	An empty string.	The IRI of xsd:boolean.

* `X.Lexical` represents the generated SQL for the Lexical Facet of "X";
* `X.ID` represents the SQL for the ID Facet of "X", and so on.

4 Optimization

Our facet-based translation may generate very complex result SQL statement. For example, in order to meet the requirement of dynamic types of operands, CASE expressions and lots of constants will appear in the generated SQL. As far as we know, most database optimizers (e.g., DB2) can not perform good optimization over complex constant and CASE expressions, since these expressions are not commonly appeared in hand-written SQLs. Therefore, additional optimization over CASE expression and constant are needed.

4.1 Optimization on CASE Expression

The `CASE` expression in generated SQLs can sometimes be replaced by other expressions which can be well optimized by DBMS engines. For example, by our algorithm, a simple filter expression "`?X != <http://foo/boo>`" is translated into the following SQL expression. The `t1` is a local nickname of the IRI table.

```
(CASE WHEN t1.IRI <> 'http://foo/boo' THEN 'true'
      WHEN t1.IRI = 'http://foo/boo' THEN 'false'
 END) = 'true'
```

However, the above SQL expression can be simply rewritten into another SQL expression with the same semantics:

```
t1.IRI <> 'http://foo/boo'
```

In order to solve this problem, we apply a special optimization on `CASE` expressions. We try to find such "`=`" predicates in the expression which meet all following requirements:

1. One side is a constant while the other side is a `CASE` expression with exactly two `WHEN` clauses.
2. Predicates of the two `WHEN` clauses are exactly the negation of each other. There are two kinds of negations. One is replacing the main operator with the inverse one, while the other is adding a `NOT` outside.
3. One of the results matches the constant on the other side of the "`=`" while the other does not.

If such a predicate is found, we replace the whole predicate with the predicate inside the `WHEN` clause, whose result matches the constant.

4.2 Optimization on Constant Expression

Optimization on constant expressions are especially important for the auto-generated SQLs. One reason is that lots of constants (such as results of Language Facet, Datatype Facet, etc.) may appear in the SQL expression, while most existing database optimizers can not perform good optimization over complex constant expressions.

Another reason is that after this process, it is more likely that the optimization on `CASE` expression can be applied, as useless `WHEN` clauses might be totally removed in the process. Therefore, we recursively optimize the constant expressions in the generated SQL. Table 4 shows several examples. `<A>` and `<B>` represent arbitrary sub-expressions.

Table 4. Constant expression optimization examples

Before Optimization	**After Optimization**
`‘ab’ <> ‘’`	`0=0`
`‘ab’ IS NULL`	`0<>0`
`<A> AND 0=0`	`<A>`
`<A> OR 0=0`	`0=0`
`COALESCE(<A>, ‘ab’, <B>)`	`COALESCE(<A>, ‘ab’)`
`COALESCE(<A>, NULLIF(‘’,‘’), <B>)`	`COALESCE(<A>, <B>)`
`CASE WHEN 0<>0 THEN <A> WHEN <B> ...`	`CASE WHEN <B> ...`
`CASE WHEN 0=0 THEN <A> WHEN <B> ...`	`<A>`

5 An Example of Translation

In this section, we take the query in Figure 1 as an example to explain the translation for filter expressions. The E1 to E8 in Figure 4 are the names of the expression nodes.

(1) The goal of translation is to generate SQLs for the Boolean Facet of the root node E1. E1 is an expression node with `&&` operator. By the translation of `&&` in Table 3, we get

```
E1.Boolean = E2.Boolean AND E5.Boolean
```

(2) E2 is an expression node with `<` operator. By the processes in Section 3.2, we get the (2.1) to (2.3) steps.

(2.1) As E3 and E4 can provide a Numeric Facet, the SQLs for E2.Boolean includes the following `WHEN` clauses:

```
WHEN E3.Numeric < E4.Numeric THEN ‘true’
WHEN E3.Numeric >= E4.Numeric THEN ‘false’
```

(2.2) As E3 and E4 can provide a Lexical Facet, the SQLs for E2.Boolean further includes the following `WHEN` clause:

```
WHEN E3.Language = ‘’ AND E4.Language = ‘’ THEN
  CASE WHEN E3.Datatype IS NULL
        AND E4.Datatype IS NULL THEN
    CASE WHEN E3.Lexical < E4.Lexical THEN ‘true’
         WHEN E3.Lexical >= E4.Lexical THEN ‘false’
  WHEN E3.Datatype = xsd:string
   AND E4.Datatype = xsd:string THEN
    CASE WHEN E3.Lexical < E4.Lexical THEN ‘true’
         WHEN E3.Lexical >= E4.Lexical THEN ‘false’
  END
```

** The xsd:string stands for its full URI SQL string.*

(2.2.1) As E3 `?age` is a variable node, as the description in Section 3.2, two left joins (one for Numeric Facet, the other for Lexical, Language and Datatype Facets) are added to the SQLs of the parent AND node, and the corresponding columns are returned as the result of these facets.

(2.2.2) E4 is a literal constant node, so the results of the requested facets are as follows:

```
E4.Numeric  = 25.0  E4.Lexical  = ‘25’
E4.Language = ‘’    E4.Datatype = xsd:integer
```

(2.3) Adding a “`= ‘true’`” after the big `CASE` expression, we get the complete translation result of `E2.Boolean`.

(3) E5 is a REGEX function, by Table 2 the Boolean Facet is translated to “`E6.Lexical LIKE ‘%Ball’`”. The `‘%Ball’` is translated from the regular expression pattern “`Ball$`”.

(3.1) The Lexical Facet of E6 should be translated to the IRI Facet or Lexical Facet of E7. However, we can know from the triple patterns that variable ?interest is bound to objects of object property “bm:like” or “bm:love”, and thus must be IRI references. So the translation of `E6.Lexical` becomes simply `E7.IRI`. In order to translate `E7.IRI`, one more left join is added in order to retrieve the actual IRI string for variable ?interest.

(4) Now we have the following translation result before the optimization step:

```
(CASE
  WHEN ?age.Numeric < 25.0 THEN ‘true’
  WHEN ?age.Numeric >= 25.0 THEN ‘false’
  WHEN ?age.Language = ‘’ AND ‘’ = ‘’ THEN
    CASE WHEN ?age.Datatype IS NULL
          AND xsd:integer IS NULL THEN
      CASE WHEN ?age.Lexical < ‘25’ THEN ‘true’
           WHEN ?age.Lexical >= ‘25’ THEN ‘false’
    WHEN ?age.Datatype = xsd:string
     AND xsd:integer = xsd:string THEN
      CASE WHEN ?age.Lexical < ‘25’ THEN ‘true’
           WHEN ?age.Lexical >= ‘25’ THEN ‘false’
    END
END) = ‘true’ AND ?interest.IRI LIKE ‘%Ball’
```

Notice that `?age.Numeric` corresponds to the value column in Numeric table, and so forth. The SQL result includes several `CASE` clauses for the comparisons of ?age with different possible data types.

Optimization for the generated SQL. The optimization includes following steps:

(1) By the the optimizations on constant expressions, “`xsd:integer IS NULL`” and “`xsd:integer = xsd:string`” can be replaced with false constants, i.e., “`0<>0`”.

(2) The two surrounding `WHEN` clauses can be totally removed as the condition is never true. Now, the nested `CASE` expression have no more `WHEN` clauses inside and is replaced by “`NULLIF(‘’,‘’)`”.

(3) The clause “`WHEN ?age.Language = ‘’ AND ‘’ = ‘’ THEN NULLIF(‘’,‘’)`” can be removed, as it is the last `WHEN` clause and the result is the same as the `ELSE` part, which is `NULL` by default. The SQL becomes:

```
(CASE
  WHEN ?age.Numeric < 25.0 THEN ‘true’
  WHEN ?age.Numeric >= 25.0 THEN ‘false’
END) = ‘true’ AND ?interest.IRI LIKE ‘%Ball’
```

(4) It can be seen that the “`=`” predicate meets the requirements for the `CASE` expression optimization described in Section 4.1. We get the final translate result after this optimization, which is a very simple SQL:

```
?age.Numeric < 25.0 AND ?interest.IRI LIKE ‘%Ball’
```

6 Experimental Analysis

All experiments were run on a 3.0GHz Core 2 Duo PC with 2GB RAM, running Microsoft Windows XP Professional. IBM DB2 9.1 Enterprise edition is used as the backend store. We implemented the FSparql2Sql in SOR [8]. The Sesame v1.2.6 [3] is extended with DB2 support and adopted for comparison purpose.

To better evaluate filter expressions, we extend the University Ontology Benchmark (UOBM)[2] with a new property "bm:age". Every student and professor are given an integer age. The largest dataset includes 1.1M instances and 2.4M relationship assertions. Six adopted SPARQL queries are shown in Table 5. We further adopted Query 7 to Query 25 for testing the functionality of FSparql2Sql on built-in functions (refer to Table 6), comparison operators (refer to Table 7), logical operators (refer to Table 8) and calculation operators (refer to Table 9), respectively.

Effectiveness Analysis. As shown in Figure 5(a), the execution time of Sesame with a global filter (i.e., Q1) or not (i.e., Q0) is almost the same. Because Sesame uses the same SQL for them and evaluates global filters in Java program. The queries with a nested filter (i.e., Q2) is definitely slower than the ones without it. The reason is that lots of SQL queries are generated and executed for the nested filter. Figure 6(a) show the same tendency for a more complex group of queries.

For FSparql2Sql, we can observe from Figure 6(b) that the queries with a global filter run faster than the one without it, while the queries with a nested

Table 5. SPARQL Queries

Groups	Queries	Notes
Group 1	`Q0. SELECT ?x ?y WHERE { ?x rdf:type bm:GraduateStudent . ?x bm:age ?y }`	Q0 is a simple triple pattern query without Filter.
	`Q1. SELECT ?x ?y WHERE { ?x rdf:type bm:GraduateStudent . ?x bm:age ?y . FILTER (?y < 25) }`	Q1 extends Q0 with global filters.
	`Q2. SELECT ?x ?y WHERE { ?x rdf:type bm:GraduateStudent . OPTIONAL { ?x bm:age ?y . FILTER (?y < 25) } }`	Q2 extends Q0 with nested filters in optional patterns.
Group 2	`Q3. SELECT ?x ?y WHERE { ?x rdf:type bm:GraduateStudent . ?x bm:isAdvisedBy ?y . ?y bm:age ?z }`	Q3 is a complex triple pattern query without Filter.
	`Q4. SELECT ?x ?y WHERE { ?x rdf:type bm:GraduateStudent . ?x bm:isAdvisedBy ?y . ?y bm:age ?z . FILTER (?z > 50) }`	Q4 extends Q3 with global filters.
	`Q5. SELECT ?x ?y WHERE { ?x rdf:type bm:GraduateStudent . OPTIONAL { ?x bm:isAdvisedBy ?y . ?y bm:age ?z . FILTER (?z > 50) } }`	Q5 extends Q3 with nested filters in optional patterns.

Table 6. SPARQL Queries of built-in functions

Groups	Queries	Notes

Group3 (built-in functions)	`Q7. SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (bound(?x))`
	`}`
		function `bound`

	`Q8. SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (isIRI(?x))`
	`}`
		function `isIRI`

	`Q9. SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (isLiteral(?y))`
	`}`
		function `isLiteral`

	`Q10.SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (datatype(?y) = xsd:integer)`
	`}`
		function `datatype`

	`Q11.SELECT ?x WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:like ?interest .`
	`FILTER (lang(?interest) = EN”) ”`
	`}`
		function `lang`

	`Q12.SELECT ?x WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:like ?interest .`
	`FILTER (REGEX(STR(?interest), Ball$”))”`
	`}`
		function `STR`

Table 7. SPARQL Queries of Comparison Operators

Groups	Queries	Notes

Group4 (comparison operators)	`Q13.SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (?y < 25)`
	`}`
		operator `<`

	`Q14.SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (?y <= 25)`
	`}`
		operator `<=`

	`Q15.SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (?y > 25)`
	`}`
		operator `>`

	`Q16.SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (?y >= 25)`
	`}`
		operator `>=`

	`Q17.SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (?y = 25)`
	`}`
		operator `=`

	`Q18.SELECT ?x ?y WHERE {`
	`?x rdf:type bm:GraduateStudent .`
	`?x bm:age ?y .`
	`FILTER (?y != 25)`
	`}`
		operator `!=`

filter use almost the same time as the normal ones. Figure 6(b) show the same tendency for a more complex group of queries. This is because FSparql2Sql successfully translates the global filter directly into a `WHERE` condition, and translates the nested filter expressions into left joins with `WHERE` conditions.

Comparing Figures 5(a) and (b), we can see that the FSparql2Sql is almost 50 times faster than Sesame even for queries without filters. This is mainly due to the differences between two systems, such as different database schemas and the batch strategy adopted in FSparql2SqlȦnd FSparql2Sql clearly outperforms

Table 8. SPARQL Queries of Logical Operators

Groups	Queries	Notes

Group6 (logical operators)	Q19.SELECT ?x ?y WHERE {
	  ?x rdf:type bm:GraduateStudent .
	  ?x bm:age ?y .
	  FILTER (isLiteral(?y) && isURI(?x))
	}
		operator &&

	Q20.SELECT ?x ?y WHERE {
	  ?x rdf:type bm:GraduateStudent .
	  ?x bm:age ?y .
	  FILTER (isLiteral(?y) || isURI(?x))
	}
		operator ||

	Q21.SELECT ?x WHERE {
	  ?x rdf:type bm:GraduateStudent .
	  ?x bm:like ?interest .
	  FILTER (! REGEX(STR1(?interest), Ball$”))”
	}
		operator !

Table 9. SPARQL Queries of Calculation Operators

Groups	Queries	Notes

Group7 (calculation operators)	Q22.SELECT ?x ?y WHERE {
	  ?x rdf:type bm:GraduateStudent .
	  ?y rdf:type bm:GradudateStudent .
	  ?x bm:age ?age1 .
	  ?y bm:age ?age2 .
	  FILTER ((?age1 + ?age2) > 40)
	}
		operator +

	Q23.SELECT ?x ?y WHERE {
	  ?x rdf:type bm:GraduateStudent .
	  ?y rdf:type bm:GradudateStudent .
	  ?x bm:age ?age1 .
	  ?y bm:age ?age2 .
	  FILTER ((?age1 - ?age2) > 0)
	}
		operator -

	Q24.SELECT ?x ?y WHERE {
	  ?x rdf:type bm:GraduateStudent .
	  ?y rdf:type bm:GradudateStudent .
	  ?x bm:age ?age1 .
	  ?y bm:age ?age2 .
	  FILTER ((?age1 / ?age2) = 1)
	}
		operator /

	Q25.SELECT ?x ?y WHERE {
	  ?x rdf:type bm:GraduateStudent .
	  ?y rdf:type bm:GradudateStudent .
	  ?x bm:age ?age1 .
	  ?y bm:age ?age2 .
	  FILTER ((?age1 * ?age2) > 400)
	}
		operator *

Sesame about 100 and 150 times for global and nested filter, respectively. This is because FSparql2Sql fully utilizes the filter information to generate more efficient SQLs.

Scalability and Functionality Test. To test the scalability of FSparql2Sql we adopt a simple query of Query 1 and a more complex query as shown in Figure 1 (we numbered it as Query 6). Figure 7(b) shows the execution time of FSparql2Sql increase slowly with the number of universities. Figure 7(a) shows the results of Sesame over the same queries. Compared Figures 7(a) and (b), we can see that FSparql2Sql achieves average 100 performance gain over Sesame in Queries 1 and 6.

FSparql2Sql supports almost all filter expressions in SPARQL. In order to show the functionality of FSparql2Sql we test the queries defined in Tables 6, 7, 8 and 9 over thress universities. Figure 8, 9, 10 and 11 show our FSparql2Sql can

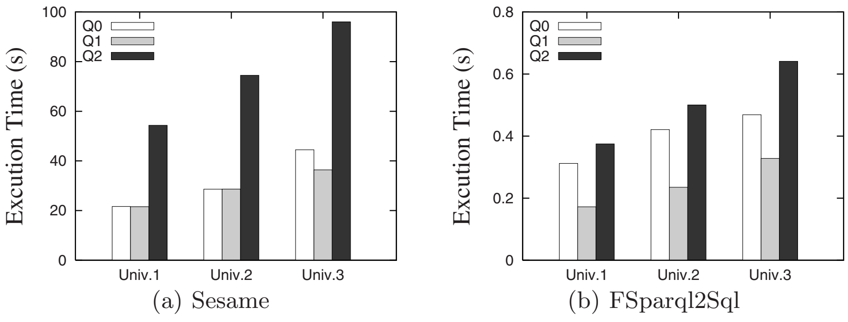

Fig. 5. Execution Time on Query Group 1

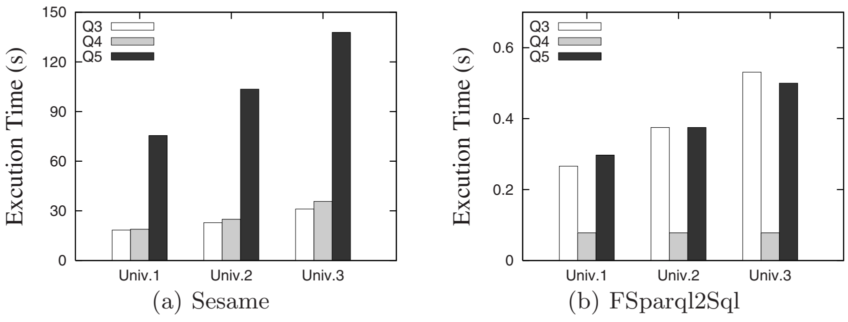

Fig. 6. Execution Time on Query Group 2

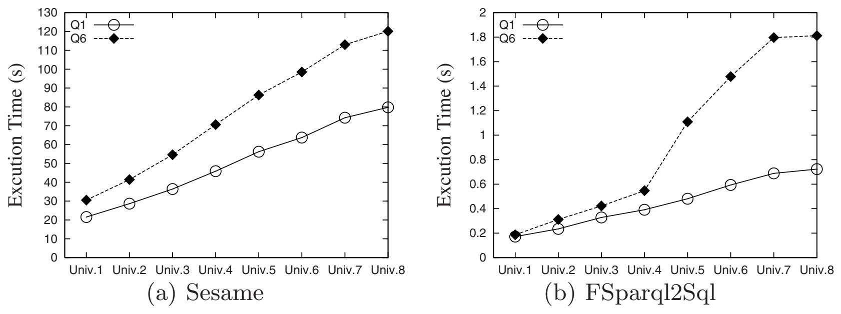

Fig. 7. Execution Time on Queries 1 and 6

well support all these built-in functions and operators and almost show linearly increase over the number of universities.

Therefore, we claim that our FSparql2Sql is an effective method to generate more efficient SQLs and better utilize the DBMS optimizers.

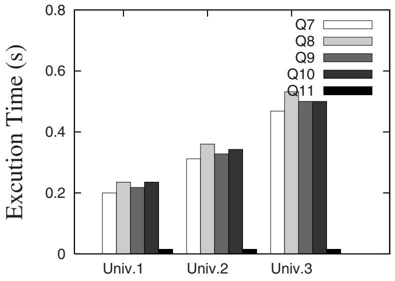

Fig. 8. Built-in Functions

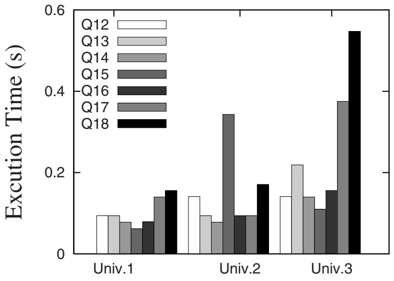

Fig. 9. Comparison Operators

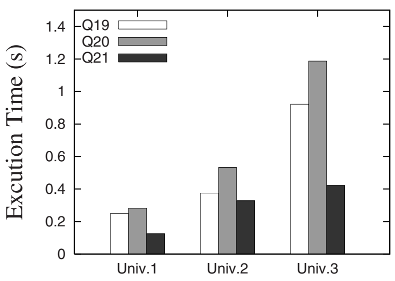

Fig. 10. Logical Operators

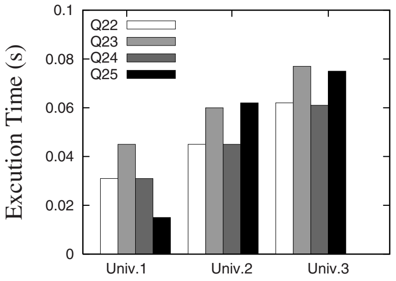

Fig. 11. Calculation Operators

7 Related Work

There are research work around semantics of SPARQL [9,1]. Jorge Perez et al. [9] provided a compositional semantics and complexity bounds, and showed that the evaluation of SPARQL patterns is PSPACE-complete without filter conditions. Different from these works on SPARQL itself, our method mainly focuses on supporting SPARQL queries over relational DBs.

Some other works proposed methods to support basic query patterns of SPARQL [3,4,5,10], including triple patterns, union patterns and optional patterns. For filter expression, They either ignore it due to complexity, or evaluate it based on memory. Cyganiak [4] presented a relational model of SPARQL and defined relational algebra operators (join, left outer join, projection, selection, etc.) to model SPARQL. Using relational algebra operators similar to [4], Harris [5] presented an implementation of SPARQL queries on top of a relational database engine, but only including a subset of SPARQL (there are no UNION operator and nested optional blocks). Sparql2Sql [7] and Sesame [6] query module are two famous query engines for SPARQL. They rewrite SPARQL queries into SQL statements. However, filter expressions are not handled by the database, but evaluated in Java code. In this paper, we show effective schemes to translate practical filter expressions into SQL statements so that a SPARQL query can be completely represented by a SQL query.

A SQL table function is defined in Oracel 10gR2 to accept RDF queries as a subquery [11], instead of supporting new query languages, e.g. RDQL[12]. RDF graph patterns defined in the table function as input are similar to the basic SPARQL patterns and can be considered as a small subset of SPARQL in terms of query capability. Compared with Oracle's RDF query based on triple patterns, we support most SPARQL features, such as nested optional patterns and complex filter expressions.

8 Conclusions

Aiming to a seamless integration of SPARQL queries with SQL queries, we proposed an effective method to translate a complete SPARQL query into a single SQL. The translated SQL query can be directly used as a sub-query by other SQL queries and evaluated by well-optimized relational database engine. In particular, we proposed effective schemes to translate filter expressions into SQL statements, which is ignored or not well addressed by existing methods. Finally, we investigated optimization strategies to improve query performance significantly. Future work is to support more SPARQL features, such as XQuery functions.

References

1. SPARQL Query Language for RDF, `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`
2. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., Liu, S.: Towards a Complete OWL Ontology Benchmark. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 125–139. Springer, Heidelberg (2006)
3. Broekstra, J., Kampman, A., Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002)
4. Cyganiak, R.: A Relational Algebra for SPARQL. HP-Labs Technical Report, HPL-2005-170 (2005)
5. Harris, S., Shadbolt, N.: SPARQL Query Processing with Conventional Relational Database Systems. In: Dean, M., Guo, Y., Jun, W., Kaschek, R., Krishnaswamy, S., Pan, Z., Sheng, Q.Z. (eds.) WISE-WS 2005. LNCS, vol. 3807, pp. 235–244. Springer, Heidelberg (2005)
6. Sesame, `http://www.openrdf.org/`
7. Sparql2Sql, `http://jena.sourceforge.net/sparql2sql/`
8. Lu, J., Ma, L., Zhang, L., Wang, C., Brunner, J., Yu, Y., Pan, Y.: SOR: A Practical System for Ontology Storage, Reasoning and Search. In: 33rd International Conference on Very Large Data Bases, pp. 1402–1405. ACM, New York (2007)
9. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
10. Pan, Z., Heflin, J.: DLDB: Extending Relational Databases to Support Semantic Web Queries. In: 1st International Workshop on Practical and Scalable Semantic Systems. CEUR-WS.org, Florida (2003)
11. Chong, E., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL-based RDF Querying Scheme. In: 31st International Conference on Very Large Data Bases, pp. 1216–1227. ACM, New York (2005)
12. RDQL - A Query Language for RDF, `http://www.w3.org/Submission/RDQL/`