SPARQL to SQL Translation Based on an Intermediate Query Language

Sami Kiminki, Jussi Knuuttila, and Vesa Hirvisalo

Aalto University School of Science and Technology

Abstract. We present a structured approach to SPARQL to SQL translation using AQL—a purpose-built intermediate query language. The approach produces a single SQL query for a single SPARQL query. Using AQL, we revisit the semantic mismatch between SPARQL and SQL and present query transformations on AQL presentation which enable the correct translation of some difficult corner cases. By using explicit expression type features in AQL, we also present type inference for expressions. We demonstrate the benefit of type inference as a basis for semantically correct optimizations in translation.

1 Introduction

We present a flexible approach to translating SPARQL queries into SQL queries, and discuss the properties of the approach based on our experimental implementation.

SPARQL is a query language for RDF graphs [14]. RDF data consists of triples expressing relationships between nodes. RDF is semi-structured, in that it does not imply a schema for storage. On one hand, this makes RDF a very flexible mechanism and suitable for representing, *e.g.*, web-related meta-data or other arbitrarily structured information. On the other hand, making efficient queries to access such data is not easy.

Even if native RDF stores are arguably more promising in the long run, there exists a massive amount of data stored in SQL databases with associated technology, infrastructure and know-how. This cannot be ignored in discussions on large scale adoption. During the transition, it is attractive to consider storing the RDF data in SQL databases parallel with non-RDF data in existing systems. The existing data can then be provided as virtual RDF graphs to applications [4], providing unified access to all data.

We address query translation from SPARQL into SQL that enables the use of SQL databases with flexible storage schemas. Storing RDF data in an SQL database is not trivial. For example, there is no single SQL layout for RDF data that is the best in all cases [1, 13]. Because of this, a flexible approach where the SQL layout can be tuned on per-application basis is preferable. Similarly, query translation is not easy, as SQL and SPARQL differ significantly, and for some corner cases, even getting semantically correct translation is hard [6]. Further, producing SQL queries that can be executed efficiently by the SQL database is difficult.

Our approach to the translation is to produce a single SQL query for a single SPARQL query without the need for result post processing, except the presentation. Also, an important design goal has been to produce SQL with the support for using native SQL data types where possible and avoiding subselects. This approach minimizes the amount of communication round-trips and leaves more optimization opportunities for the SQL back-end [11].

To obtain these goals, we devised a translation design based on an intermediate language that we call AQL (Abstract Query Language). As is typical for intermediate languages, it has straightforward basic semantics and has the ability to attach information to support translation. Such properties make it easy to find a translation from the source language into AQL and enable finding efficient translations from AQL to the target language.

AQL targets only the query semantics for translation. It does not address other issues, such as the representation of results like many concrete query languages do (including SPARQL and SQL). AQL has been designed especially in the context of SPARQL to SQL translation. It is used to address the translation of queries into semi-structured data into relational database queries in general.

AQL has language features that enable the use of type information to support translations. Similar type-based static analysis and translation mechanisms have been used for programming languages. We demonstrate how such a methodology can be applied to support the translation of SPARQL queries into SQL queries.

We have created an experimental implementation of our translation approach to test its properties. Our implementation, *Type-ARQuE* (Type-inferring AQL-based Resource Query Engine) [15], is an optimizing SPARQL to SQL query translator. It supports the most important SPARQL language features in order to validate the design. Type-ARQuE is written in C++ and supports PostgreSQL and MySQL back-ends with different database layout options.

Based on the implementation, we show how some demanding cases of RDF queries can be translated into efficient SQL queries. Our demonstration cases underline the challenges raised by alternate variable bindings, variable scoping and determining the required value joins. Especially, we demonstrate how determining the required value joins can benefit from type inference.

We review some of the related background in Sec. 2. The translation design is covered in Sec. 3, containing an overview of the AQL language and the steps of the translation of SPARQL into SQL by using AQL as an intermediate. We give special attention to the use of AQL type information, as type inference is essential for our translation. In Sec. 4 we illustrate the translation with concrete examples. We continue by discussing the properties of our translation (Sec. 5) and end with brief conclusions (Sec. 6).

2 Background

SPARQL[14] is a query language for RDF graphs. It is an official W3C recommendation. SPARQL has syntactic similarity to SQL but with some important differences. Whereas in SQL the query data set is specified by joining tables (`FROM` and `JOIN` clauses), graph match patterns are used in SPARQL.

Relational databases are an important back-end option for storing RDF graphs due to the wide user base of relational databases. A variety of SQL layouts for storing RDF graphs have been suggested, but it seems that no single layout is good for all purposes [1, 13].

Harris was one of the first to systematically consider SPARQL to SQL translation discussing various ways of organizing RDF triple stores and considers especially using SQL back-ends [8]. The opportunity for a number of optimizations is acknowledged and the problem underlined as nontrivial.

To our knowledge, one of the most fundamental works on the problem domain is presented in [6]. The technical report explains the SPARQL algebra with discussion on how to map the SPARQL algebra to the traditional relational algebra. In the report, difficult corner cases of translation are also analyzed.

A rather flexible translation approach is presented in [11]. They consider SPARQL query translation into SQL queries by using a facet-based scheme that is designed to handle filter expressions. They underline that it is desirable that a single SPARQL query is translated into a single SQL statement, and that comparison between results of different data types is useful. They also consider optimization strategies to reduce complexity of translated queries.

Hartig and Heese considered optimization by query translation at SPARQL algebraic level [9]. Their approach was based on translating the query in Jena ARQ into a custom representation (SQGM) for optimization, and then translating back into ARQ.

Lately, Chebotko et al. presented a method for translating a SPARQL query to a single SQL query with preservation of semantics [5]. Their method operates on SPARQL algebraic level, and relies on SQL subqueries on data set declaration.

Left-to-right variable binding semantics are an alternative to bottom-up semantics. This changes variable scoping, enabling queries containing filters in nested graph groups that depend on variables bound by their parents. For a discussion, see [6]. SPARQL utilizes currently the bottom-up semantics.

3 The Translator Design

The query translator in Type-ARQuE translates SPARQL queries into SQL. The translator was designed with three main goals in mind. First, the translator should produce a single SQL query for a single SPARQL query. Second, the type support of the SQL back-end should be utilized. Finally, the translator should not be fixed to some specific SQL schema and SQL dialect.

The translator is based on a multi-stage translation architecture [2], consisting of front-end, intermediate, and back-end translation stages. The front-end parses and translates SPARQL queries into intermediate queries (Sec. 3.2). The intermediate query language, AQL (Sec. 3.1), is specifically designed to stand between SPARQL and SQL. The intermediate translation stage (Sec. 3.3) consists of general query transformation and optimization passes (general preparation) as well as back-end specific transformation passes (specialization), and utilizes type inference (Sec. 3.5). Finally, the translator back-end translates the AQL query into SQL using a specific target dialect (Sec. 3.4).

A main design goal for AQL, the Abstract Query Language, was to be compact with straightforward but high-level semantics. It is relational in nature. The join expressions in AQL are extended from the traditional relational algebra to cover both SPARQL and SQL join semantics.

3.1 The Abstract Query Language

AQL is an intermediate query language, representing the query semantics. In other words, it does not cover representation of the query results, such as SELECT vs. CONSTRUCT forms in SPARQL. Its intended use is machine-only. We begin by introducing the language and then defining the query evaluation constructs and semantics, and finally, we consider expressions in AQL.

An AQL query is represented by a query object, which contains the data set declaration, sort orders, result slicing, and select expressions. The data set declaration consists of a join tree, where each node contains a set of join names, possible child nodes, and join criteria. The data set declaration specifies the data that is used in the query process. The data is a list of query solutions, represented by a 2-dimensional array, columns as solution components and rows as different solutions. Order expressions specify the ordering of the data set. Result slicing (LIMIT and OFFSET) selects a specific range of rows. Select expressions are projections of a solution row to singular values, comparable with SQL select expressions.

A node in a join tree represents joining one or more columns to the data set using the attached join criteria. Child nodes represent nested joins. Each node may be of type INNER or LEFT OUTER join. The columns are named by the join names.

Joins manipulate the rows in the data set. Conceptually, this begins with creating a temporary data set by taking all the triples in the store, and raising the set to the Cartesian power of the number of triple names in the join node. The temporary data set is then joined to the result data set by Cartesian product. Then, the child nodes, if any, are joined. Finally, each join criterion is evaluated per row. The row is eliminated unless it meets all criteria. It is possible that a row that was originally in the result data set is eliminated if all new combinations fail to meet the criteria. In that case, if the join is LEFT OUTER join, the original row is retained and nulls are inserted to new columns.

There are differences in the join processes between the relational algebra and AQL, namely, in the order of operations. In the relational algebra, the data set is created by processing the nested joins first. Joining a table (or a set of tables) and evaluating the join condition is an atomic operation [7]. In AQL, a top-down approach is used instead. The child joins are joined to the parent recursively, and only after that the join criteria are processed. The AQL approach enables referencing more columns in the join criteria than what is possible in the SQL approach. In SQL, the child joins and parent may be referenced in the join criteria. In AQL, the parent (recursive) and every earlier node using in-order join tree traversal may be used, in addition. This extension covers both the SPARQL and SQL semantics.

We now define the AQL and its evaluation semantics, borrowing notation used in SPARQL evaluation definition [14] where applicable.

The query object is defined as follows:

```
(aql-query join-name-group
           join*
           (criterion <expr>)
           sort-expr*
           (result-max-rows <integer>)?
           (result-row-offset <integer>)?
           select-expr*
           distinct?)
```

The parts are:

join-name-group — The join names for the root join node.
join* — Any number of join expressions.
criterion — The filter expression for the data set.
sort-expr — Any number of sort expressions defining the sequence for data set enumeration.
result-max-rows, result-row-offset — Slice specifiers.
select-expr* — The select expressions.
distinct? — Optional result modifier.

A join expression represents joining the set of all triples in the store one or more times into the data set. The join expression is defined as:

```
(join join-type join-name-group expr join*)
```

And the parts:

join-type — The join type, either `INNER` for inner join or `LEFT` for left outer join.
join-name-group — The set of join names.
expr — The join condition expression.
join* — Any number of nested joins.

The `aql-query` object, join expressions and AQL query criteria form a join tree which specifies the data set. The data set is a 2-dimensional array, consisting of solutions as rows and triples as named columns. For the query result data set definition, let T be the set of all triples in the store and I the identity for Cartesian product (I is an empty array of 1 rows, 0 columns).

The data set for a join subtree is produced by function `JoinNode`(D_0, n) where the current data set is denoted as D_0 and the root node of the subtree as n. Return value denotes the data set after joining n to D_0. The following steps define the evaluation of `JoinNode`(D_0, n):

1. Join the Cartesian product specified by join name group of the node:
   $D_r \leftarrow D_0 \bowtie D_1$, where $D_1 = T_{jn_1} \times \cdots \times T_{jn_N}$
2. Join children of n in left to right order into the data set D as:
   $D_r \leftarrow$`JoinNode`$(\cdots($
   `JoinNode`(`JoinNode`($D_0 \bowtie D_1$, n.child1), n.child2), $\cdots$ n.childN)
3. Apply filter:
   $D_r \leftarrow filter(D_r, n.expr)$
4. If the join type of n is LEFT, add row d,ω into D_r for each row d removed from D_0 by the filter expression, where ω contains `nulls` to fill the join columns.
5. Return D_r

The result data set D for the query is produced by `JoinNode`(I, `aql-query`). Fig. 1 illustrates the result data set with query result.

After the result data set is formed, order expressions are used to sort the data set. The first order expression is evaluated for each row and then the rows are ordered so that the rows with smaller order value are enumerated first in ascending order or vice versa in descending order. If two or more rows have the same order value, the second order expression is used to determine the ordering between these rows, and so on. The sort expressions are of form:

```
(sort ascending|descending <expression>)
```

Finally, select expressions are applied for each row in the sorted data set. The set of select expressions for a row produces a result row. When the select expressions are applied for each row in the data sequence, the result sequence is produced. In AQL, select expressions are of form:

```
(select <column-label> <expression>)
```

After the result sequence is produced, if the `distinct` modifier is set, all duplicate rows are eliminated.

The expressions in AQL are of three categories: typed literals, triple property expressions, and function expressions. Function and property expressions are assigned a set of possible types. The expression templates are below:

```
(literal <type> <value>)
(property <type-set> <join-name> subject|predicate|object)
(function <function-name> <type-set> <param-expr>*)
```

`literal` specifies a typed literal, such as 5 (int) or 'abc' (string). `property` specifies access to the subject, the predicate, or the object of a named triple in a solution. `type-set` in `property` makes an assumption of the property type: the type must belong to the assumed type set. `function` represents evaluation of a function returning a value of a type belonging to the type set.

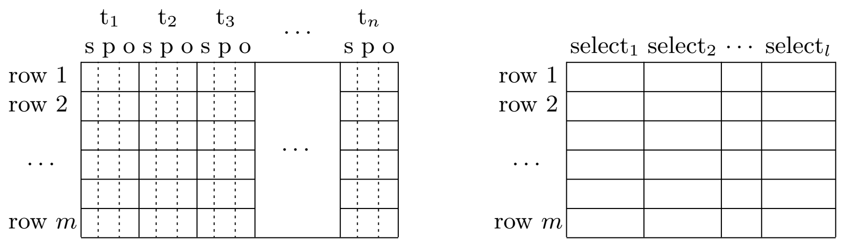

Fig. 1. Illustration of a solution set (left-hand table) and the query result (right-hand table) of an AQL query as a 2-dimensional arrays. Triple columns are added by in-order traversal of join groups, one column and three sub-columns (subject, predicate, object) per triple name. The rows are inserted and filtered by running the data set production algorithm described in Sec. 3.1. Each row represents a single solution to the query. Afterwards, the rows are ordered and sliced, and finally, the select expressions are evaluated per solution row to produce the query result array.

3.2 SPARQL to Abstract Query Language

The translation into AQL is mostly straightforward. First, the SPARQL query is parsed into an abstract syntax tree. After that, variable accesses are checked to conform to bottom-up binding unless left-to-right semantics are enabled. In bottom-up binding, variable may be used in FILTER expression if it is bound by a triple match pattern in the same or a nested graph group. Then, the abstract syntax tree is normalized by merging non-optional SPARQL graph groups with their parents.

For each graph group an AQL join node is created and unique join names are assigned to the triple match patterns in the graph group. Then the join names are inserted into the respective AQL join groups. The graph group hierarchy is naturally preserved in the AQL join group hierarchy.

Variable binding step addresses the mapping of SPARQL variable bindings to AQL property and function expressions. The SPARQL semantics require that variables are bound by the first matching triple match pattern. As non-optional match patterns always bind a variable, that variable can be mapped to the property of the triple join corresponding to the first non-optional match pattern. However, optional match patterns introducing variables require a bit more consideration.

Before a variable is encountered in a non-optional match pattern, the variable may be bound by optional graph groups containing a match pattern mentioning the variable. In this case, coalesce-expressions are used to select the first value-binding triple match visible at point of access.

After the variables are bound to property expressions, join conditions are constructed by triple match patterns. Literals in match patterns impose constraints to triple join properties. If a match pattern introduces a new variable, no condition is rendered. If a variable is already introduced, it is required that the property of the respective triple join is equal to the variable, if the variable was bound. When it is not certain that the variable is bound, additional not-null conditions are added for non-optional match patterns. After translating match pattern conditions, filter expressions are translated as additional join conditions. Thus, match patterns and filters are unified.

Finally, selects and order expressions are translated. This completes the SPARQL to AQL translation.

3.3 Translation Passes on Abstract Query Language

The translation passes prepare an AQL query for SQL translation. The process is called *lowering*. Lowering consists of *general preparation* and *specialization* parts. General preparation is a sequence of generic transformation passes which simplify the AQL, and makes it easier to translate. Specialization part perform back-end-specific transformations such as replacing AQL property expressions with SQL access expressions for a specific layout. We list the passes in both parts briefly below.

General preparation:

- Inner join merge — joins inner joins with parents to simplify the query.
- Logical expression normalization — moves not-expressions inwards using De Morgan's laws and fuses not-expressions with comparisons, *e.g.*, $\neg(a > b \land c = d) \rightarrow a \leq b \lor c \neq d$
- Operators to functions — transforms comparison, typecast, and logical operators to equivalent function expressions.
- Type inference — infers possible types for expressions. Described in more detail in Sec. 3.5.
- Empty type sets to nulls — replaces possible empty type sets with null expressions. An expression with conflicting type requirements may only produce a null.
- Nested join flattening — transforms deep nested joins with many-levels-up accesses to a less deep form. Exemplified in Fig. 4.
- Comparison optimization — transforms equality and non-equality value comparisons to reference comparisons.
- Function variant selection — chooses the most appropriate variants of polymorphic functions in expressions.

Specialization:

- Property value requirer — adds not-null conditions whenever property expression could produce null. Null can be produced if value table(s) must be left-joined or typecast must be used to access value of a triple property.
- Property access resolver — rewrites property access expressions with lower-level back-end-specific equivalents. In the back-end-specific accesses table names, value and index columns are resolved and the required typecasts and `COALESCE`s are added whenever needed.
- Expression optimization — simplifies various expressions and performs common subexpression elimination. Having an explicit clean-up pass simplifies some of the previous passes.

– Function variant selection — this pass is run again to ensure that all variants of functions have been chosen after transformations.
– Typecast injection — inserts typecasts wherever needed in the expressions.
– Property access collection — collects all low-level property accesses. This is used to determine which value-joins are required in the final SQL.

3.4 Translation to SQL

After an AQL query is lowered, the translation into SQL is straightforward. The arrangement resembles code generation in compilers [3]. The AQL expressions in selects, orders, join conditions, and the query criteria are translated into SQL by traversing the expression trees.

The AQL join tree is then transformed into an SQL join tree with value joins and join expressions attached. For multiple triple joins in AQL join group, cross join is used in SQL. The SQL join tree is then serialized into string form.

Finally, all the translation results are inserted into an SQL query template. This completes the SPARQL to SQL translation process.

3.5 Type Inference

In our approach, type inference for all expressions is performed in AQL after the join tree and its expressions are normalized. The inferred information is used, *e.g.*, to optimize value accesses in complex SQL schemas.

We utilize equations on two levels to resolve the possible types for property and function expressions. The lower-level equations infer the possible types in the join conditions. The higher-level equations transfer the inferred type constraints between join conditions in the join tree.

Throughout this section, p denotes the property, *i.e.*, the triple component (subject, predicate, or object), S is the finite set of all known types and $C_{n,p} \subset S$ is the set of the possible types of a property p and expression node n. For example, $C_{n,\mathrm{t5.object}} = \{\mathrm{string}, \mathrm{int}\}$ specifies that the object of the named triple t_5 may only be of type string or int. n is the expression node (*i.e.*, function, literal or property expression) that scopes this constraint.

Type Inference on Join Condition Expressions. The normalized expression tree may contain function, property and literal expression nodes. Property and literal expression nodes are always leaf nodes. The children of function nodes represent function parameters.

For every expression node n, a set of possible expression types $R_n \subset S$ and a set of property type constraints $C_{n,p}$ are assigned. Then, based on expression node types, conditions to R_n and $C_{n,p}$ between adjacent nodes are set, as:

– if n is a literal of type t: $R_n = \{\mathrm{t}\}$, $C_{n,p} = C_{parent(n),p}$
– if n is a property expression p: $R_n = C_{n,p}$, $C_{n,p} = C_{parent(n),p}$

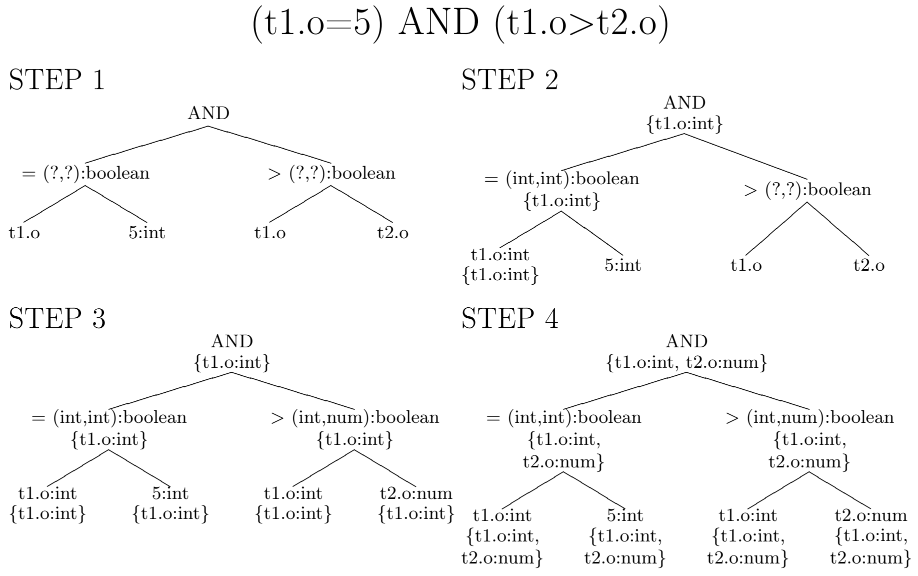
(t1.o=5) AND (t1.o>t2.o)
STEP 1
AND
= (?,?):boolean
> (?,?):boolean
t1.o
5:int
t1.o
t2.o
STEP 2
AND {t1.o:int}
= (int,int):boolean {t1.o:int}
> (?,?):boolean
t1.o:int {t1.o:int}
5:int
t1.o
t2.o
STEP 3
AND {t1.o:int}
= (int,int):boolean {t1.o:int}
> (int,num):boolean {t1.o:int}
t1.o:int {t1.o:int}
5:int {t1.o:int}
t1.o:int {t1.o:int}
t2.o:num {t1.o:int}
STEP 4
AND {t1.o:int, t2.o:num}
= (int,int):boolean {t1.o:int, t2.o:num}
> (int,num):boolean {t1.o:int, t2.o:num}
t1.o:int {t1.o:int, t2.o:num}
5:int {t1.o:int, t2.o:num}
t1.o:int {t1.o:int, t2.o:num}
t2.o:num {t1.o:int, t2.o:num}

Fig. 2. Illustration of type inference for a simple expression. Using the inference procedure described in Sec. 3.5, the set of possible types (in braces $\{\cdots\}$, '?' represents all possible types) are reduced by analyzing expression types and function signatures, and propagating the sets using transfer functions. Property t1.o has been inferred as integral type and t2.o as general numeric type.

– if n is an expression node for the function f and m_i are the parameter nodes. Then, $R_n = F_f(R_{m_1}, R_{m_2}, \ldots)$ where F_f maps the set of possible parameter types to a set of possible return types. The constraint set propagation depends on the type of the function f as follows. If f is
  - or: $C_{n,p} = C_{parent(n),p} \cap (\bigcup_i C_{m_i,p})$
  - not: $C_{n,p} = C_{parent(n),p}$
  - any other function: $C_{n,p} = C_{parent(n),p} \cap (\bigcap_i C_{m_i,p})$

Expression type inference is illustrated in Fig. 2.

Constraint Propagation between Joins. The constraints are transferred between the roots of join condition expressions, as the effective Boolean value of the expression root determines whether the join contributes to the query solution. In the join tree, the constraint transfer between any two nodes must follow the following two simple rules:

1. **Every inferred type constraint of node n applies to every child node m_i of n**, *i.e.*, $C_{m_i,p} \subset C_{n,p}$. This is because the nested joins cannot contribute to the solution if n is null, and therefore, we can assume that inferred type restriction applies also to the nested joins without problems.

2. **Every inferred type constraint made in n for a property of a triple defined by n or some of its children applies everywhere.** This is because the property value can be non-null only if n is not null.

Note that by rule 1, all constraints inferred for the root join apply everywhere.

Solving the Equations. We take a conservative approach by assuming initially that objects of triples may be of any type. Subjects and predicates may only be IRIs in RDF graphs. Then we exclude non-viable type alternatives, *i.e.*, those that cannot appear. The exclusion is performed by iterating the two steps below until a fixed point is found. The steps correspond to the higher and lower level type inference equations. As the sets of viable type alternative constraints are initially finite and may only shrink, the fixed point is guaranteed to be reached. The descriptions of exclusion steps:

1) Condition expression step. First, all function expressions are analyzed in pre-order, separately for all join condition expressions. Based on parameter types and constraints on the return type the union of the viable function variants are computed. This may imply new constraints on function parameters, *e.g.*, function `a=abs(b)` requires that `b` is numeric. If function parameters are property expressions, the type set $C_{n,p}$ of the property is shrunk to reflect function parameter requirements.

After the constraint sets of properties are shrunk, they are propagated in post-order to adjacent expression nodes using the transfer rules described in *Type Inference on Join Condition Expression* above. Finally, every constraint set is intersected by constraints in expression root node.

2) Constraint propagation step. After the possible type sets in join condition expressions are shrunk, the sets are propagated between join conditions. This is straightforward using the two rules in *Constraint Propagation between Joins.*

4 Translation Examples

In this section, we demonstrate some of the techniques utilized by our query compiler. *Alternate binders* demonstrates a translation where a variable may bound by two alternate optional match patterns. *Two levels up access* demonstrates query flattening in AQL. This is required when there is a reference to a variable which is bound by a graph pattern residing two or more levels up in the join tree. *Expression type inference* demonstrates how type inference is used to eliminate superfluous joins to value tables in faceted storage layouts. The examples are translated using Type-ARQuE using the PostgreSQL target. *Alternate binders* and *two levels up access* are motivated by the corner cases discussed in [6].

In examples *alternate binders* and *two levels up access*, the triple graph is stored in a single table, `InlinedTriples`, consisting of three columns for subjects, predicates and objects. In *expression type inference* we use a central triple table, `VI_Triples`, which consists of three index columns. The columns refer to value tables containing the actual property values, such as `VI_Strings` and `VI_BigStrings` for string-typed values.

Alternate binders query (Fig. 3) has two optional graph patterns, both which may bind variable `d`. We call this the alternate binders case. The variable dereference expression depends on where `d` is accessed. In the first optional match, we may assume that `d` is bound there. In the second match there are two alternatives: 1) the first match binds and the triple object must equal to this, or 2) the first pattern did not match and equality comparison should be omitted. In `SELECT` part, `COALESCE`-expression is used to select the binding expression.

Two levels up access example (Fig. 4) demonstrates working around the "two-levels-up" variable access by transforming the join tree. The nested join is translated as a sibling join with additional condition requiring that the former parent must match. In the figure, the join tree of the AQL is displayed before and after flattening. In this example, the extended left-to-right variable binding semantics are assumed. The query is not proper for bottom-up semantics, as in this case, the variable `c` is inaccessible in the filter.

In *Expression type inference* (Fig. 5) we demonstrate the use of type inference to determine the required value joins when using a facet-based storage layout. Without type inference, when dereferencing a variable, all value tables must be joined which can be costly. Using type inference, the number of joins and the complexity of respective coalesce expressions can often be reduced significantly.

5 Discussion

Our translation approach is somewhat different to the approach in popular SPARQL-enabled RDF stores, such as Jena/SDB[10] and Sesame[12], in that we use a purpose-built intermediate language in the translation. The use of an intermediate language enables an approach for translation using relatively small and straightforward passes.

AQL provides a new look into the problem of mismatching semantics between SPARQL and SQL, especially when left-to-right variable binding semantics are used. This is because AQL is more explicit than SPARQL algebra regarding query evaluation, and AQL is less complex than SPARQL algebra in what comes to the language features.

Contrary to SPARQL, AQL does not have query variables. In our approach, variables are translated into triple property access expressions, which refer to parts of the query solutions. Resolving variables to property access expressions is fairly straightforward and eliminates tedious translation problems related to variables altogether. Variables are especially difficult to translate directly into SQL expressions, as they may be bound by different parts of SPARQL query. We demonstrated this in the alternate binders case in Sec. 4.

When using the extended left-to-right variable binding, the "FILTER scope" problem poses a difficult corner case for translation [6]. The naïve SPARQL to SQL translation always fails, because in SQL, the variable bound by a graph

```
SELECT ?a ?d
WHERE {
  ?a ?b ?c
  OPTIONAL { ?a <http://test/surname> ?d }
  OPTIONAL { ?a <http://test/lastname> ?d }
}

SELECT tri_1_1.subj_value AS c0,
  COALESCE(tri_2_1.obj_value,tri_3_1.obj_value) AS c1
FROM InlinedTriples AS tri_1_1
LEFT JOIN InlinedTriples AS tri_2_1 ON
  tri_2_1.subj_value=tri_1_1.subj_value AND
  tri_2_1.pred_value='http://test/surname'
LEFT JOIN InlinedTriples AS tri_3_1 ON
  tri_3_1.subj_value=tri_1_1.subj_value AND
  tri_3_1.pred_value='http://test/lastname' AND
  (tri_2_1.obj_value IS NULL OR
   tri_3_1.obj_value=tri_2_1.obj_value)
```

Fig. 3. Variable `d` is bound by the first matching optional graph pattern. Variable dereference is translated as coalesce of alternate binders. In the latter optional graph pattern, the case where the first graph pattern binds `d` is taken into account by inserting additional null or equals condition.

pattern is not available in an optional join at a nesting distance of two or more levels. In our approach, in these cases the join tree is flattened by a semantically equivalent transformation, reducing the set of untranslatable queries. As a side product of variable elimination and "FILTER scope" workaround, the "Nested `OPTIONAL`s" problem in [6] is also remedied.

Using the left-to-right semantics, there are queries that are structurally untranslatable by our method. This class consists of queries where a parent graph group refers to a variable in nested graph group with a two-levels-up access. We present an archetype of this class, which we call the *2-up-1-down-access* query:

```
SELECT *
WHERE { ?a ?b ?c
        OPTIONAL { ?d ?e ?f
                   FILTER(?i='abc')
                   OPTIONAL { ?g ?h ?i
                              FILTER(?c='def') } } }
```

Within this class of queries, join tree flattening produces access expressions with forward references in the join conditions. In SQL, this is illegal. However, we believe that most of the practical queries do not belong to this class.

The data set construction semantics in AQL can be considered as a process of self-joining the set of all triples in the store with join conditions instead of applying triple match patterns. This unifies the handling of triple match patterns and filters. This also enables a clean isolation of triple store layout from the rest of the translation, as we can consider all the triples to reside in one virtual table, and translate the accesses to the virtual triple table to concrete layout-specific tables and columns.

```
SELECT ?a ?d ?e ?g
WHERE { ?a ?b ?c
 OPTIONAL { ?a ?d ?e
            OPTIONAL { ?e ?f ?g FILTER (?c=45.1) } } }

(aql-query ("tri_1_1") ...   # before flattening
  (join left ("tri_2_1")
    (function "builtin:comp-eq" (boolean)
              (property (IRI) "tri_2_1" subject)
              (property (IRI) "tri_1_1" subject))
    (join left ("tri_3_1")
      (function "builtin:and" (boolean)
        (function "builtin:comp-eq" (boolean)
          (property (IRI) "tri_3_1" subject)
          (property (IRI) "tri_2_1" object))
        (function "builtin:comp-eq" (boolean)
          (property (double) "tri_1_1" object)
          (literal double 45.1)))    ...)

(aql-query ("tri_1_1") ...   # after flattening
  (join left ("tri_2_1")
    (function "builtin:comp-eq" (boolean)
       (property (IRI) "tri_2_1" subject)
       (property (IRI) "tri_1_1" subject)))
  (join left ("tri_3_1")
    (and (function "builtin:and" (boolean)
      (function "builtin:comp-eq" (boolean)
        (property (IRI) "tri_3_1" subject)
        (property (IRI) "tri_2_1" object))
      (function "builtin:comp-eq" (boolean)
        (property (double) "tri_1_1" object)
        (literal double 45.1)))
      (function "builtin:is-not-null" ANY
        (property (reference) "tri_2_1" subject))))
...)

SELECT tri_1_1.subj_value AS c0, tri_2_1.pred_value AS c1,
       tri_2_1.obj_value AS c2, tri_3_1.obj_value AS c3
FROM InlinedTriples AS tri_1_1
 LEFT JOIN InlinedTriples AS tri_2_1 ON
   tri_2_1.subj_value=tri_1_1.subj_value
 LEFT JOIN InlinedTriples AS tri_3_1 ON
   tri_3_1.subj_value=tri_2_1.obj_value AND
   aqltosql_any_to_double(tri_1_1.obj_value)=45.1 AND
   tri_2_1.subj_value IS NOT NULL
```

Fig. 4. The "two-levels-up" variable access in filter condition is flattened by one level by a join tree transformation. The join `tri_3_1` is moved down to the same level with `tri_2_1` with additional condition that `tri_2_1` needs to be non-null. The original semantics are retained but the query becomes translatable into valid SQL. Non-interesting parts of the AQL queries are pruned away for brevity. This query requires the extended left-to-right variable binding semantics.

```
SELECT ?a ?c
WHERE { ?a ?b ?c. FILTER (?c=55 || ?c='David') }
ORDER BY (?c)

SELECT tri_1_1_subj_VC_IRIs.iri_value AS c0,
  COALESCE(tri_1_1_obj_VC_Strings.str_value,
    tri_1_1_obj_VC_BigStrings.text_value,
    CAST(tri_1_1_obj_VC_Integers.int_value AS TEXT)) AS c1
FROM VC_Triples AS tri_1_1
 LEFT JOIN VC_Strings AS tri_1_1_obj_VC_Strings ON
   tri_1_1_obj_VC_Strings.id=tri_1_1.obj
 LEFT JOIN VC_BigStrings AS tri_1_1_obj_VC_BigStrings ON
   tri_1_1_obj_VC_BigStrings.id=tri_1_1.obj
 LEFT JOIN VC_Integers AS tri_1_1_obj_VC_Integers ON
   tri_1_1_obj_VC_Integers.id=tri_1_1.obj
 INNER JOIN VC_IRIs AS tri_1_1_subj_VC_IRIs ON
   tri_1_1_subj_VC_IRIs.id=tri_1_1.subj
WHERE
  (tri_1_1_obj_VC_Integers.int_value=55 OR
   COALESCE(tri_1_1_obj_VC_Strings.str_value,
     tri_1_1_obj_VC_BigStrings.text_value)='David') AND
  (tri_1_1_obj_VC_Strings.str_value IS NOT NULL OR
   tri_1_1_obj_VC_BigStrings.text_value IS NOT NULL OR
   tri_1_1_obj_VC_Integers.int_value IS NOT NULL)
ORDER BY COALESCE(tri_1_1_obj_VC_Strings.str_value,
  tri_1_1_obj_VC_BigStrings.text_value,
  CAST(tri_1_1_obj_VC_Integers.int_value AS TEXT)) ASC
```

Fig. 5. Type inference is especially useful with faceted storage layouts. This becomes obvious when observing translation of variable `c`. It is inferred that `c` must be either integer or string for any solution to the query. Therefore, only string and integer value tables are required to be joined to obtain value for `c`. As variable `a` is used in subject, it is inferred as IRI.

6 Conclusion

We have presented an approach for SPARQL to SQL translation. The translation produces a single SQL query for a single SPARQL query, and does not rely on SQL result post-processing, except in data presentation. This approach reduces the amount of communication round-trips to the SQL server and allows the SQL server do more optimizations.

The translation is structured into three stages (front-end, intermediate, back-end) and the stages themselves are subdivided into self-contained passes. The intermediate stage operates on a purpose-built intermediate language, AQL. Using AQL, we have presented a way to target the queries for different SQL layouts for RDF data, and a strategy for query optimization.

As a basis for optimizations, we introduced type inference and provided an algorithmic view on its implementation. The implementation relies only on the constraints derived from the SPARQL query. Type inference is used to optimize expressions and SQL accesses. Ontology-awareness would likely enhance type inference in many practical scenarios, as, *e.g.*, known predicate of a triple often constrains type possibilities of the object. For triple with predicate `foaf:homepage`, we would expect IRI as the object type, for instance.

Using the extended join semantics of AQL, we provided intermediate-level query transformations for reducing the mismatch between SPARQL and SQL join semantics. The transformations enable translation of the corner cases presented in [6]. However, there is still a class of SPARQL queries that remain untranslatable by our approach, when the extended left-to-right variable binding semantics are used. Of this class, we presented a representative archetype.

To validate our design, we implemented Type-ARQuE, an experimental translator based on the presented design. The translator covers a representative subset SPARQL and demonstrates the translation in detail.

Acknowledgments. This work was supported by the DIEM project, funded by TEKES/Finland and the partner organizations of DIEM.

References

1. Abadi, D., Marcus, A., Madden, S., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: 33rd international conference on Very large data bases. pp. 411–422. VLDB Endowment (2007)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers—Principles, Techniques, & Tools. Addison-Wesley, 2nd edn. (2007)
3. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers (2002)
4. Bizer, C., Seaborne, A.: D2RQ — treating non-RDF databases as virtual RDF graphs (poster). In: ISWC (2004)
5. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving SPARQL-to-SQL translation. Data and Knowledge Engineering 68(10) (Oct 2009)
6. Cyganiak, R.: A relational algebra for SPARQL. Tech. rep., Hewlett-Packard (2005)
7. Groff, J., Weinberg, P.: SQL: The Complete Reference. McGraw-Hill, 2nd edn. (2003)
8. 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. (eds.) WISE Workshops, LNCS, vol. 3807, pp. 235–244. Springer, Heidelberg (2005)
9. Hartig, O., Heese, R.: The SPARQL query graph model for query optimization. In: The Semantic Web: Research and Applications. LNCS, vol. 4519, pp. 564–578. Springer, Heidelberg (2007)
10. Jena 2.6.2, `http://openjena.org/`
11. Lu, J., Cao, F., Ma, L., Yu, Y., Pan, Y.: An effective SPARQL support over relational databases. In: SWDB-ODBIS 2007. LNCS, vol. 5005, pp. 57–76. Springer, Heidelberg (2008)
12. Sesame 2.3.1, `http://www.openrdf.org/`
13. Sintek, M., Kiesel, M.: RDFBroker: A signature-based high-performance RDF store. In: The Semantic Web: Research and Applications. LNCS, vol. 4011, pp. 363–377. Springer, Heidelberg (2006)
14. SPARQL Query Language for RDF, `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`
15. Type-ARQuE 0.2, `http://esg.cs.hut.fi/software/type-arque/`