SPARQL Compiler for Bobox

Miroslav Cermak, Jiri Dokulil and Filip Zavoral
Charles University in Prague, Czech Republic
{*cermak, dokulil, zavoral*}*@ksi.mff.cuni.cz*

***Abstract*—The Bobox framework is a platform for parallel data processing. It can even be used as a database query evaluation engine. However, it does not contain the means necessary to compile and optimize the queries. A specialized front-end is needed. This paper presents one such front-end, which handles queries written using the SPARQL language. The front-end also performs query optimizations taking the specific features of the SPARQL language into account.**

***Keywords*-SPARQL; Bobox; query optimization.**

I. Introduction

The SPARQL language [1] is one of the most popular RDF (Resource Description Framework [2]) query languages. There are several database engines that are capable of evaluating SPARQL. Unfortunately, their performance is still behind state-of-the-art relational and XML databases.

The Bobox parallel framework was designed to support development of data-intensive parallel computations [3]. One of the main motivation was to use it in web semantization research we are currently conducting [4]. The main idea behind Bobox is to connect large number of relatively simple computational components into a non-linear pipeline. This pipeline is then executed in parallel, but the interface used by the computational components is designed in such a way that they do not need to be concerned with the parallel execution – issues like scheduling, synchronization and race conditions. This system may be easily used for database query evaluation, but a separate query compiler and optimizer has to be created for each query language, since Bobox only supports a custom low-level interface for the definition of the structure of the pipeline.

Traditionally (for example in relational databases), query execution plans have the form of directed rooted trees where the edges indicate the flow of the data and all are directed to the root. The nodes of the tree are the basic operations used by the evaluation engine, like full table scan, indexed access, merge join, filter etc. This maps well to the Bobox archtecture, since the tree is a special case of the non-linear pipeline supported by the system. Each operation is mapped to one (or possibly a fixed combination of) components and the components are connected in the same manner as in the original evaluation plan.

We decided to take similar approach for SPARQL. An example of a query evaluation plan is shown in the Figure 1. While the operations used by the SPARQL algebra resemble operations used in the relational algebra, there are

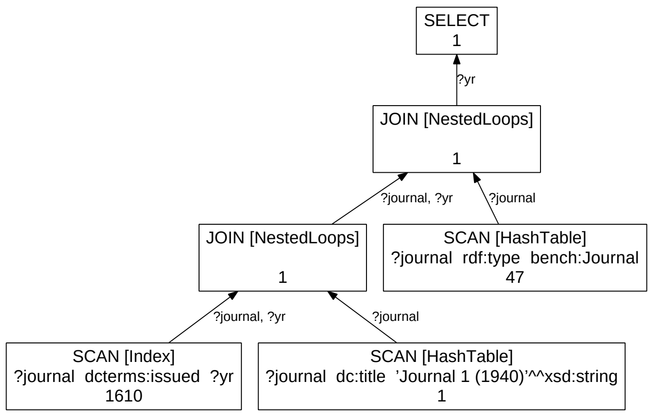

Figure 1. Query evaluation plan example

some more or less significant differences. This prevents the relational algebra from being used directly by the SPARQL evaluation engine. Furthermore, some optimizations used in relational optimizers are not applicable for SPARQL, since the several of the transformations that are used when optimizing relational queries do not preserve the semantics of the SPARQL queries.

The rest of the text is organized as follows: first, the issues related to SPARQL execution, most notably the difference from relational algebra, are discussed in the Section II. Next, the Section III describes data representation used by our system and the statistics we collect about the queried data set. Sections IV and V describe the way in which the query is parsed, transformed and the way in which the final execution plan is generated from the transformed query. The Section VI provides some evaluation of our approach. The last section concludes the paper and discusses future work.

II. SPARQL

The sematics of the SPARQL language is defined using the SPARQL algebra. The algebra is similar to the relational algebra, but there are several important differences.

The relational algebra works with relations (tables), while the SPARQL algebra uses sets of variable mappings. Unlike SQL, there are no NULL values in SPARQL. Instead, the variable is left unbound. This is not just a minor technicality, it significantly affects the way in which some operations behave. There is no difference between an unbound variable and variable that is not present at all. For example the

SPARQL equivalent of left natural join produces a row (a variable mapping to be exact) even when the tested variable is unbound (which would also happen if the variable was not present in the query at all), unlike SQL where the operation is null-rejecting. This may prevent some optimizations like join-reordering to be performed on SPARQL since they could change the results under certain conditions. These optimizations may still be performed, but care must be taken to mind the specific constraints imposed by the SPARQL algebra.

There are two main circumstances when this behavior demonstrates. First, consider the following simple SPARQL query with a OPTIONAL expression:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person, ?contact
WHERE {
      ?person foaf:mbox ?contact
      OPTIONAL(?person foaf:phone ?contact)
}
```

In this query, the `contact` variable is used both inside and outside the OPTIONAL pattern. The resulting behavior is that if the person has an email (foaf:mbox) the mail is returned but if the mail is not present, but a phone number is (foaf:phone), the phone number is returned. Performing this operation is SQL requires a more complicated combination of operations.

The second situation where different definition of left joins demonstrates is when the OPTIONAL branch contains FILTER operation. The operation cannot be performed on just the data from that branch – it is an integral part of the join operation and may filter even the data from the non-optional branch, as in the following example:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person, ?contact
WHERE {
   ?person foaf:age ?x
   OPTIONAL(?person foaf:name ?b FILTER (?x>18))
}
```

On the whole, this means that we cannot directly apply optimization methods that were developed for SQL.

III. Data representation and statistics

Currently, we only work with local data sets. We assume that the data is stored in the most general model – one ”triple” table where all triples are stored. Besides that, we have several indexes, which are in fact the same table but with a specified order (using for example a B-tree). For example, one index is sorted by predicate, subject and the object. This may be used for example when we know the values of predicate and subject and we need to get all objects (and they are already sorted).

Besides the actual data and indexes, we need further information – statistics about data for the cost based optimizations. Since the number of different predicates is usually very limited, we can afford to store the number of distinct

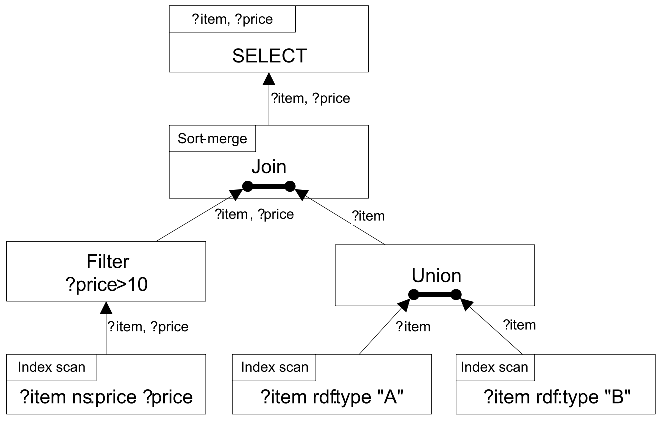

Figure 2. SQGM example

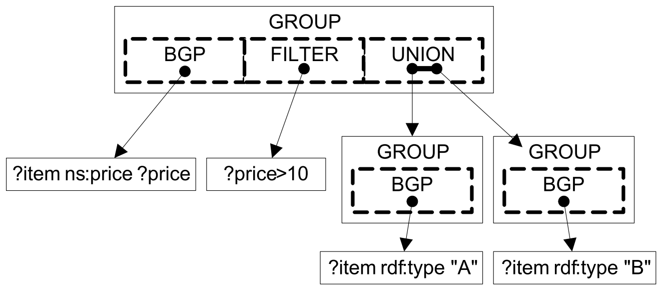

Figure 3. SQGPM example

triples for each predicate. On the other hand, the number of subjects may be very high, so we only store the total number of triples and the number of distinct predicates. For the objects we use equal-height histograms [5]. This provides a balance between the size of the statistics and their precision – we do not store the number of triples for each object value, but if one value is much more common than the others, it would be detected from the histograms. These statistics allow us approximate selectivity of basic graph patterns.

We also try to make some approximations of the results of join operations. We consider the average selectivity of the join of two triples a_1, a_2, a_3 and b_1, b_2, b_3 with the join condition in the form $a_i = b_j$ – we only store one number for each possible form (9 combinations) of the join condition. Besides these general statistics, we use more detailed information for situations where both predicates are known. Then, there are four combinations on join conditions (a_1, p, a_2 and b_1, q, b_2 joined on $a_i = b_j$) for each pair of predicates. This means we have $4 * n^2$ values where n is the number of predicates. This should still be manageable amount of information.

During the optimization, the query evaluation plan is stored in the form of the SQGPM (SPARQL Query Graph Pattern Model) which we designed as an extension of the SQGM model [6]. The difference is that instead of individual operations, the nodes of the tree are formed by groups of operation – a group is a set of operations where the order in which the operations are evaluated does not affect the result of the operation. A SQGM model can be created by replacing each group of operations with a tree composed of those operations. This representation allows us to do transformations that are performed before the order of join operations is determined more easily, since the model is not yet made unnecessarily complicated by the "insignificant" joins (those whose order does not change the result of the query).

An example of the SQGM is shown in the Figure 2 and the corresponding SQGPM model in the Figure 3. They show models of the example query from the previous section.

IV. QUERY PARSING AND REWRITING

The compilation of a query is performed in several consecutive steps. The first step is query parsing. The input stream is parsed into the SQGPM model using standard methods of lexical and syntactical analysis.

The next step is query rewriting. Since the queries to be processed are not expected to be written optimally (duplicities, constant expressions, inefficient conditions, etc.), the goal of this phase is normalization of the model. There are four operations performed on the SQGPM model:

- Merging of included *Group graph patterns*
- Duplicity elimination
- Propagation of `filter`, `distinct` and `reduced`
- Projection of variables

Resulting tree is functionally equivalent to the original one, but its evaluation can be more efficient and better execution plan can be generated.

A. Merging of included group graph patterns

The goal of the merging phase is to detect group graph patterns that can be merged with their parent group patterns while preserving the equivalence of the SQGPM.

Consider the following query:

```
SELECT *
WHERE { ?x ?a ?b . { ?x ?y ?z . { ?a ?b ?c } } }
```

There is only one possible operation ordering when the triples are grouped this way:

Join(?x ?a ?b, Join(?x ?y ?z, ?a ?b ?c))

Using such ordering the nested *Join* operation generates a cartesian product that is consumed by the outer *Join* operation. However, an equivalent representation of the query is as:

```
SELECT *
WHERE { ?x ?a ?b . ?x ?y ?z . ?a ?b ?c }
```

This representation results in wider range of operation ordering, e.g.:

Join(?x ?y ?z, Join(?x ?a ?b, ?a ?b ?c))

Such ordering does not produce the cartesian product; it uses smaller result sets and therefore is more efficient.

Nevertheless, not every *group graph pattern* (*GGP*) can be merged to its parent *GGP*. The problem arises if *GGP* contain both unbound variables and a *Filter* that defines restrictions on the variables. These variables may be bound in another *GGP*, in which case changing the scope of the *Filter* operation may change the result of the query.

Bound variables cannot change their value, they are safe with respect to the *FILTER* operation. The following example IV-A demonstrates a case where merging of *GGP*s is not possible:

```
SELECT *
WHERE {
      ?s rdf:type ?t .
      {P . FILTER(bound(?t)) }
}
```

P represents a graph pattern group for which the result set contains the variable `?s` and possibly unbound variable `?t`. Then the original representation rejects all tuples containing the unbound variable `?t` before joining the *triple* in the parent *GGP*. On the other hand, if we first join the nested *GGP* to the parent one and then perform the *Filter*, the variable `?t` will be bound by the parent *GGP* and the *Filter* never removes such result.

B. Duplicity elimination

The goal of the next phase is to eliminate duplicate graph patters. The following example demonstrates the problem:

```
SELECT DISTINCT *
WHERE {
   ?obj rdf:type ?t .
   ?obj rdf:type ?t
}
```

The query contains two equal triples `?obj rdf:type ?t`. The execution of the second triple and the subsequent join will not generate any new variable mapping not present originally.

If only bound variables are present, there is no combination of rows that would produce a new, unique row. The size of the result set is equal to the input set (possibly increased by duplicates). Then the `DISTINCT` modifier removes all duplicates which makes the result of the join equivalent to the original results of the `?obj rdf:type ?t` pattern.

This optimization may only be performed under the following conditions:

- Duplicate may not under any circumstances generate unbound variables
- The query is of the type `DISTINCT` or `REDUCED`

C. Propagation of Filter

Propagation of *Filter* means that we try to move it to the lowest level (closest to the leaves of the tree that represents the query plan) where all variables used in the *Filter* are still present. Early filtering reduces the size of the result sets which speeds up subsequent operations.

Operation *Filter* where the expression is in a conjunctive form is split into subexpressions using the operator AND. Such splitting reduces the expression domain (the set of the variables used in the *Filter*) and increases the probability of its lower placement in the resulting tree.

Nevertheless, the *Filter* operation cannot be propagated arbitrarily; presence of unbound variables prevents the propagation; see the following example.

```
SELECT DISTINCT *
WHERE { A . FILTER(bound(?y)) . { B }}
```

where:

- A, B are groups of operations with results:
  A={{?x=1,?y=1}}
  B={{?x=1}, {?x=2, ?y=2}}
- FILTER(bound(?y)) is a filter that uses the (possibly unbound) variable ?y

The result is the set {{?x=1, ?y=1}}. If we propagated the filter to the nested pattern, the result set would be empty. Therefore we defined *safe* and *unsafe* variables and conditions for *Filter* propagation:

- *Safe* variable is bound for every possible tuple
- *Unsafe* variable can be unbound for some tuple

If the *Filter*'s domain contains unsafe variables then it is ordered behind the last *group graph pattern* operation in the respective operation tree. If the *Filter* domain does not contain unsafe variables and it is not a part of a *group graph pattern* which forms the OPTIONAL branch of a *LeftJoin* then it:

- can be reordered behind the following operation (in a direction to the root)
- can be reordered before the preceding operation (in a direction to leaves) if it is not an OPTIONAL branch of the *LeftJoin* operation and all used variables are available.

The *Filter* operations that are part of the OPTIONAL branch of the *LeftJoin* operation cannot be reordered, since the SPARQL language defines it to be an integral part of the *LeftJoin* operation.

D. Propagation of Distinct and Reduced

If the query uses DISTINCT or REDUCED modifier, the result set should have no duplicates – they should be eliminated as the last step of the query evaluation. However, under most circumstances, we can add this operation even to deeper levels of the query plan, especially after *Join* (if it is a merge join) and *OrderBy* operations, since the data is ordered and the elimination of duplicates can be done very cheaply.

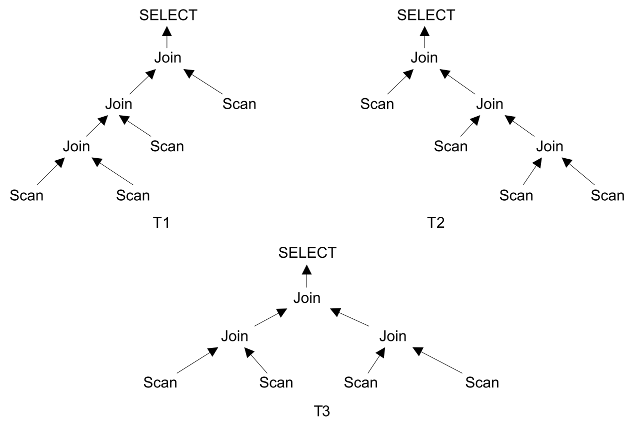

Figure 4. Tree types

V. Execution plan generation

After the transformations described in the previous sections, we still need to transform the groups present in the SQGPM model into a tree of join operations. This is performed by a non-exhaustive search of the space of all possible join types and combinations. We also have to select the best strategy to access the data stored in the physical store – select the best index to use, if any.

The query execution plan is built from bottom to the top using dynamic programming to search a part of the search space of all possible joins. We only consider left-deep trees of join operations, i.e. the right operand of a join operation may not be another join operation. See the Figure 4 for an example – T1 is a left-deep tree, T2 is right-deep and T3 is a bushy tree.

There is one exception to this rule. If there is no other way to add another join operation than adding one that would generate cartesian product, we try building the best plan for the rest of the operations (recursively using the same algorithm) and then join that plan with the one we already have. This may eliminate the need to generate cartesian products and results in an execution plan in the form of a bushy tree. This modification greatly improved plans for some of the queries we have tested and significantly reduced the depth of the trees – some of the results were almost a balanced binary tree.

The whole execution plan generation is performed by the following algorithm according to the statistics and price function that are able to provide an approximate cost for a part of any execution (sub)plan:

```
generate_plan(group_graph_pattern)
begin
  operators:=group_graph_pattern.childs;
  buckets:=empty;
  results:=empty;

  // Rating of feasible data access options
```

```
    foreach op in operators do
      foreach method in op.methods() do
        // Group operator recursive call
        if method is group then
          method := generate_plan(method);
        end if

        c := cost_of(method);
        s := sort_order_of(method);

        // The cheapest only
        if buckets[s].cost > c then
          buckets[s] := method;
        end if
      end for
    end for

    // Tree extension
    for i:=1 to |operators| do
      foreach tree in buckets do
        inops:=not used operators in tree;
        foreach op in inops do

          // Heuristics: skip the operators
          // that generate avoidable
          // cartesian products
          if carthesian && !required then
            continue;
          end if

          // Using join implementations
          foreach jtype in join_types do
            // Heuristics: left-deep tree
            newtree:= op_join(tree, op);
            c:=cost_of(newtree);
            s:=sort_order_of(newtree);

            if buckets[s].cost > c then
              buckets[s] := newtree;
            end if
          end for
        end for
      end for
      buckets:=res;
    end for

    // Result: the cheapest feasible plan
    return min from res;
  end
```

The main goal of the design of this algorithm is to minimize the number of sort operations, make the best use of merge-join operations and avoid joins that generate cartesian products.

VI. EVALUATION

An efficient implementation of the evaluation components for the Bobox system that could execute the generated query plans is not yet available. This allowed us to perform only two types of experiments so far: manually checking the plans generated by the compiler and comparing the cost estimates produced by the compiler with the actual size of the query result and intermediate results.

We have tested the queries provided by the SP^{2}Bench [7] benchmark suite for SPARQL. The Figure 5 shows an example the plan produced for the following query:

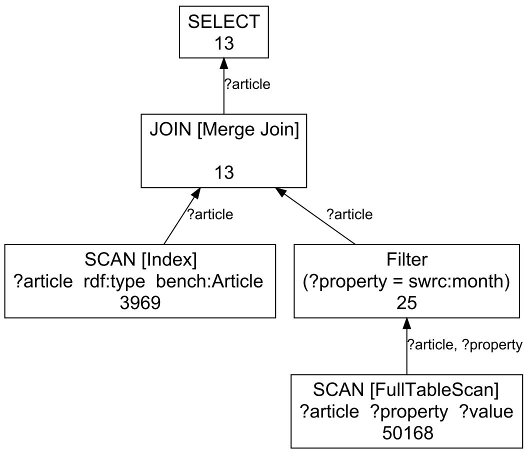

Figure 5. A simple query example

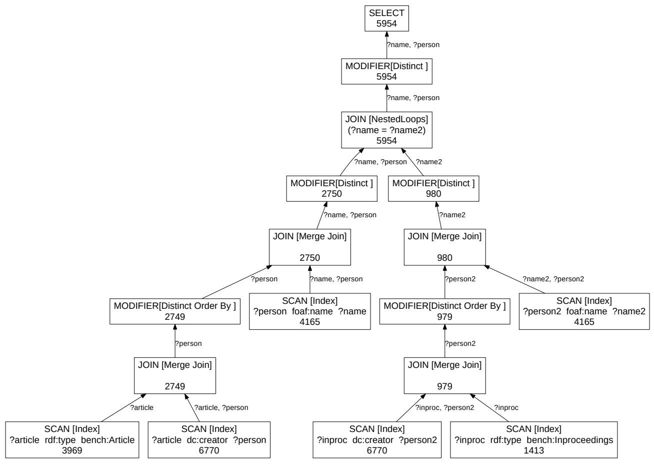

Figure 6. Example of a bushy tree

```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:month)
}
```

A more complex example that demonstrates the bushy trees that may be produced by the compiler is shown in the Figure 6. We were able to compile all `SELECT` queries defined by the SP^{2}Bench benchmark with satisfying results.

VII. CONCLUSION

We have created a working compiler that processes SPARQL queries and generates plans to be executed by the Bobox system. It performs a set of pre-defined optimizations to transform the execution plan into an equivalent but more efficient one. Then the query is further optimized by join reordering using dynamic programing and a cost model to asses the quality of the proposed execution plans.

An obvious next step is to implement the back-end of the SPARQL processor into Bobox and perform experiments on an actual physical RDF store. We have already created a subset of the back-end that can evaluate some of the SP^{2}Bench queries that have been compiled by hand to use only the specified subset of operations. The results of these experiments seem promising especially in comparison to current stat-of-the-art systems like Sesame [8].

ACKNOWLEDGMENT

The work was supported by the grants SVV-2010-261312 and GACR 202/10/0761.

REFERENCES

[1] E. Prud'hommeaux and A. Seaborne, *SPARQL Query Language for RDF*, W3C, 2008, http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. [Online]. Available: http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/

[2] J. J. Carroll and G. Klyne, *Resource Description Framework: Concepts and Abstract Syntax*, W3C, 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[3] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, "Using methods of parallel semi-structured data processing for semantic web," in *3rd International Conference on Advances in Semantic Processing, SEMAPRO*. IEEE Computer Society Press, 2009, pp. 44–49.

[4] J. Dokulil, J. Yaghob, and F. Zavoral, "Trisolda: The environment for semantic data processing," *International Journal On Advances in Software*, vol. 1, no. 1, pp. 43–58, 2009.

[5] Y. Ioannidis, "The history of histograms (abridged)," in *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*. VLDB Endowment, 2003, pp. 19–30.

[6] O. Hartig and R. Heese, "The SPARQL query graph model for query optimization," in *ESWC '07: Proceedings of the 4th European conference on The Semantic Web*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 564–578.

[7] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench: A SPARQL performance benchmark," *CoRR*, vol. abs/0806.4627, 2008.

[8] J. Broekstra, A. Kampman, and F. v. Harmelen, "Sesame: A generic architecture for storing and querying RDF and RDF schema," in *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*. London, UK: Springer-Verlag, 2002, pp. 54–68.