Summary-based optimization in semantic graph databases

Isak Czeresnia Etinger

Ecole polytechnique / INRIA

Summary-based optimization in semantic graph databases

Isak Czeresnia Etinger[a]

[a]Ecole Polytechnique,
Palaiseau, France

E-mail: isak.czeresnia-etinger@polytechnique.edu
Advisor: Ioana Manolescu

Abstract. RDF is the data model of choice for the semantic Web. SPARQL is the standard of the W3C for querying RDF graph. As the structure of RDF graph is heterogeneous and they lack a schema, evaluating queries over such graphs may be hard. Toward making it more efficient, in this project we study the problem of optimizing SPARQL queries based on RDF quotient summaries.

Contents

1 Introduction

The RDF format is the graph format most frequently used to encode structured data on the Web. As RDF graphs can be large and heterogeneous, evaluating queries over them can be costly. In this project, we study algorithms for optimizing queries in semantic graph databases through the use of quotient summaries: these are compact graphs whose structure is computed from the original RDF graph, such that all the paths (and graphs) present in the original graph are also present in the summary. Graph summaries have been investigated in the past, in particular for semistructured graph data models such as OEM (the Object Exchange Model), XML and more recently for RDF.

SPARQL is the standard query language for RDF graphs; it is a complex language with many features, including graph pattern queries, wildcard path queries, SQL-style aggregation etc. SPARQL evaluation and its queries are often complex. There is therefore an interest to optimize the evaluation of SPARQL queries.

We want to exploit summaries in order to help optimization of those queries. In particular, we focus on bisimulation-based summarization. And among bisimulation summaries, we focus on backward bisimulation summaries due to experiments showing its efficiency in the tested cases [ČGM17].

For a family of SPARQL dialects, going from the simplest to the most complex, and given a summary, we devise an algorithm which enumerates all the ways in which a query can be evaluated, based on (a) a simple RDF storage model which has been shown to be efficient in many prior works, and (b) the extents of summary nodes. Each such plan will then be ported into optimizer of SQL-style operations. For each type of summary analyzed, we separate all the possible graph queries by a defined type, and then for each of those types we list plans to solve such queries. The graph query types and the plans to solve them are listed in increasing order of complexity, such that we recursively use already defined query types plans to solve new graph queries.

After the listing of the possible plans to evaluate a query, statistical analysis should be applied to estimate which plan likely is the most efficient to be chosen, and therefore optimize the query execution.

The report is organized as follows:

- In section 2, we describe tools and formalism used on the paper.
- In section 3, we present research work closely related to our topic.
- In section 4, we give a formal definition of the problem statement we are trying to solve.
- In section 5, we present theory and algorithms about exhaustive plan enumeration based on backward bisimulation summaries.
- In section 6, we describe the implementation used for the algorithms proposed in section 5.
- In section 7, we give additional tools to find the plan with the lowest expected run time to evaluate a SPARQL query.
- In section 8, we conclude the paper.

2 Preliminaries

2.1 Algebraic plans

We consider a set of algebraic plans, denoted $\mathcal{P}$, built over a given set of relations, and using the following operators:

- $Scan(R)$ or simply R returning all the tuples of the relation R;
- $\sigma_{pred}(e)$ (selection), where e is an algebraic expression, returning all tuples of e that satisfy the predicate $pred$;
- $\pi_{cols}(e)$ (projection) returning tuples derived from those returned by e, which have exactly the columns (attributes) $cols$;
- $e_1 \bowtie_{pred} e_2$ (join), the join of the two algebraic expressions e_1 and e_2 on a given logical predicate $pred$;
- $e_1 \cup e_2$ (union), returning all the tuples returned by e_1 or e_2. If a tuple is returned by both, then it is included once.

2.2 RDF, RDFS and SPARQL

RDF (Resource Description Framework) [w3ca] graphs are the W3C standard for representing data and knowledge.

At a first level, RDF graphs can be seen as simply encoding data in labeled graphs.

RDF graphs can also be interpreted with the help of RDF Schema [w3cb]; this schema (or ontology) language allow expressing knowledge, that is, relations between the properties and types present in an RDF graph. In the presence of an RDF schema, an RDF graph contains implicit and explicit relations, in which the implicit relations are implications of the explicit ones. A "saturated" RDF graph is defined as a RDF graph in which all possible implicit relations have already been made explicit.

SPARQL (SPARQL Protocol and RDF Query Language) is a query language for RDF databases. It is widely used for queries in RDF graphs, and therefore will be the language for which we design our work. SPARQL queries allow searching for specific nodes according to the graph structure connecting them, and to the values of their properties; they can return either tuples of nodes or values (constants), or they can build new RDF triples (in which case, a query builds a new RDF graph). For example, a SPARQL query can search for nodes N_0, N_1 such that there is an arc P_0 from N_0 to N_1. There are more complex queries present in SPARQL, and we will present types of SPARQL queries as we implement solutions to them, usually using recursion to define a type of query based on already defined types of queries.

Definition 2.1. We define a *simple linear-path query* as a query of the following form. In SPARQL variables are preceded by a "?".

SELECT $?A_0$, ..., $?A_K$
WHERE {
    $?N_0$ P_0 $?N_1$.
    . . .
    $?N_{N-1}$ P_{N-1} $?N_N$.
}

such that $\{?A_0, ..., ?A_K\} \subset \{?N_0, ..., ?N_N\}$ and that $?N_i \neq ?N_j \; \forall \; i \neq j \in \{0, ..., N\}$. The set of all simple linear-path queries is noted S_{linear}.

Definition 2.2 (Simple queries)**.** A *simple query* is defined as a query that is equivalent to a selection of a union of queries L_0, ...,$L_N \subset S_{linear}$. The set of all simple queries is denoted S_{simple}.

Definition 2.3 (Regular queries)**.** We define the set of *regular queries*, denoted RQ, as follows:

- A query whose result is an empty set (denoted $\emptyset_q$) belongs to RQ;
- any query $Q \in S_{simple}$ belongs to RQ;
- For every $A, B \in RQ$:
  - A / B is in RQ, and is equivalent to the join of A and B, on the condition that *the last attribute* of A is equal to the *first attribute* of B. While in principle the attributes of a relation are not ordered, that is, "the first" and "the last" attribute are not well defined, when showing concrete relation signatures, we make the convention that "the last" attribute is the last shown in the table signature.
  - $A \mid B$ is in RQ, and is equivalent to the union of A and B;
  - $A?$ is in RQ, and is equivalent to the union of A and $\emptyset_q$;
  - $A*$ is in RQ, and denotes 0 or more repetitions of the RQ A;
  - $A+$ is in RQ, denotes 1 or more repetitions of the RQ A.

Definition 2.4 (Projection of Regular Queries)**.** We note S_{SPARQL} as the set of all SPARQL queries that are equal to a projection over a regular query.

It is easy to see that $S_{linear} \subset S_{simple} \subset S_{SPARQL}$. We will use this property to recursively implement SPARQL queries.

A formal semantics of SPARQL, which we will likely base our work on, is introduced in [PAG09].

3 Related work

In this section, we present research work closely related to our topic.

3.1 Dataguides

Dataguides [GW97] are a form of structural graph summaries introduced prior to the RDF data model. The dataguide $D(G)$ of a graph G is basically a graph equal to the union of all the possible paths present in the graph G.

Definition 3.1. A *Dataguide* of a graph G is a graph $D(G)$ such that each node N of $D(G)$ indexes the maximal set of nodes present in G that can be retrieved by the same arc path as N. That set of nodes in the original graph is called the *extent* of N.

Dataguide graphs are usually much smaller than the graphs they are derived from, and they have been introduced as a way to speed up the evaluation of queries in large OEM (Object Exchange Model) graphs, as explained in the paper [GW97]. For example, instead of searching for all the nodes that have some path in the original graph, one can search for a node in a DataGuide graph that has such path, and then retrieve its extent.

It is useful to have information on each node in $D(G)$ describing a property of a respective path in G. This information is called an *annotation* of the path.

Once a DataGraph $D(G)$ is built, any changes at the original graph G can be reflected by incremental maintenance of $D(G)$ instead of doing the complete process of summarization over again. Ideally only small and fast changes should be applied on $D(G)$ to turn it into $D(G')$.

Definition 3.2. A specially useful type of DataGuide is one called *strong DataGuide*. A DataGuide $D(G)$ is defined as "strong" iff for each path P of G, the set of paths in G that reach exactly the same points reached by P in G is equal to the set of paths in $D(G)$ that reach exactly the same points reached by P in $D(G)$.

The interest of a strong dataguide is that every target set in G has exactly one representative in $D(G)$. So, to this node, one can attach information about the target set: how many nodes, which nodes, what are their values if any etc.

3.2 Quotient-based RDF graph summarization

In [ČGM17], the authors consider building RDF summaries as quotient graphs. Naive application of quotient summarization to RDF graphs with type and schema information prevents a summary from being representative.

It is therefore useful to work with a notion of node equivalence specifically designed for RDF graphs, called *RDF equivalence*. Any class node is RDF equivalent only to itself, and any property node is equivalent only to itself.

The authors then introduce:

Definition 3.3. The summary of an RDF graph G by an RDF node equivalence relation $\equiv$ is the RDF graph $G_{\equiv}$ defined as the quotient graph of G by $\equiv$, in which every class (resp. property) preserved its URI from G URI. In contrast, non-class nodes and non-property nodes in $G_{\equiv}$ have fresh URIs.

We say a summary S is *quotient* if $\forall$ arc P_j from N_i to N_k on G there is an arc P_j in $S(G)$ from a node in $S(G)$ indexing N_i to a node in $S(G)$ indexing N_k (i.e.: it represents all arcs in G).

Every summary has the same sets of classes names, of property names, and of schema triples as the original RDF graph.

3.3 Bisimulation-based RDF summarization

The concept of (bi)simulation has been introduce to characterize self-similar structure within an RDF graph [HHK95].

Bisimulation summarization of RDF graphs is a summarization concept that groups nodes based on the set of paths *from* the nodes (also called *outgoing paths*), *to* the nodes (also called incoming paths), or both. There are three types of bisimulation summaries: *Forward Bisimulation*, *Backward Bisimulation*, and *Forward-and-Backward Bisimulation*.

To define those bisimulation summaries, it is useful to at first define the concepts of *forward, backward, and forward-and-backward bisimilarities.*

Definition 3.4. Two nodes N_1, N_2 are defined as being *forward bisimilar (FB)* iff the set of arc paths *from* N_1 and the set of arc paths *from* N_2 are the same. Likewise, two nodes N_1, N_2 are defined as being *backward bisimilar (BB)* iff the set of arc paths *to* N_1 and the set of arc paths *to* N_2 are the same. Two nodes N_1, N_2 are defined as being *forward-and-backward bisimilar (FBB)* iff they are both forward bisimilar and backward bisimilar.

All those types of bisimilarities are transitive. I.e: if N_1 and N_2 are forward (resp. backward, forward-and-backward) bisimilar, and N_2 and N_3 are also forward (resp. backward, forward-and-backward) bisimilar,then N_1 and N_3 are also forward (resp. backward, forward-and-backward) bisimilar. We can therefore group together nodes that are bisimilar to each other in one of those three types, and call a set of *pairwise* FB (resp. BB, FBB) nodes simply as a set of FB (resp. BB, FBB) nodes.

Definition 3.5. A *forward (resp. backward, forward-and-backward) bisimulation summary* of a graph G is a graph G' such that for every maximal set of forward bisimilar (resp. BB, FBB) nodes $S = \{N_1, N_2, \ldots, N_n\}$ in G, there exists exactly one node N'_S in G' such that N'_S is forward bisimilar (resp. BB, FBB) to S. In that situation S is called the *extent* of N'_S.

3.4 Query optimization based on summaries and possibly other indexes

A graph summary, in particular one for which we know the extent of each node, can be leveraged in order to make query evaluation (more) efficient on a given data graph. For instance, [MW99] shows how to leverage (*i*) dataguide node extents, (*ii*) value indexes and (*iii*) reverse arc indexes in order to enumerate *alternative data access paths*, that is: alternative data structures (collections, tables) to be scanned or accessed in order to answer the query.

Query optimization based on bisimulation-based summaries has been studied in the context of XML databases in [KBNK02]. While some main ideas from that work carry over in [ČGM17] and the present work, they did not consider RDF graphs, nor semantics, and the query language was also significantly different from SPARQL we consider.

3.5 The role of summaries in RDF query evaluation

The relationship between the two can be described as follows.

1. A generic RDF storage scheme, used in many works, creates a relation $T_a(s, p)$ for every data property a which appears in the graph. Then, conjunctive queries can be expressed as join expressions over such T_a tables.
   Note that this is independent of (and does not require) any summarization.

2. When a summary exists, one can store *node extents*, i.e. tables of the form $T_{sn}(n)$ where sn is a summary node, and n is (the identifier of) a data node in the graph, represented by the summary node sn. Then, to answer a query, join expressions can be built over the T_a tables (as above) and/or T_{sn} tables.

   - It is easy to see that to each such join expression corresponds (at least) a SQL query.
   - *Depending on the query, the data, and the summary definition*, one or another of these possible expressions will lead to the most efficient query evaluation. Finding the best alternative is the broad goal of the work (see below).

4 Problem statement

In the last years a huge number of large semantic graphs has been produced, both in the commercial sector and in academia. As this type of data structure is continuously rising, there is a need to optimize queries in databases structured in that format.

The problem to address is to design an algorithm which takes a graph, a SPARQL query and a summary of the graph, and returns the maximal set of relational algebraic plans computing the query answers based on a persistent store of the graph as well as a summary of the graph. We focus on plans as they represent a good abstraction for concrete operations which may be implemented by a system. More formally, We use the following two definitions:

Definition 4.1 (Triple tables and pair tables)**.** We consider two types of tables to store an RDF graph. One is called *triple tables*, or *subject–predicate–object* tables, denoted T, and containing columns S, P, O representing respectively the subject, predicate and object of an RDF relation. From a graph perspective, that tuple is a P arc from node S to node O. The other type is called *pair tables*, or *subject–object* tables, denoted T_P for a predicate P, and containing columns S, O representing respectively the subject and object of a RDF relation of predicate P. For a graph G, we denote its triple table as $T(G)$ and its pair table for predicate P as $T_P(G)$. When both $T_{P_i}(G)$ and $\sigma_{P=P_i}(T(G))$ may be used in an expression, we use $T'_{P_i}(G)$ to represent any of them.

Let $\mathcal{S}$ be the set of all possible summaries of an RDF graph, $\mathcal{Q}$ be the set of SPARQL queries, $\mathcal{G}$ the set of semantic graphs, and $\mathcal{P}$ is the set of algebraic plans (Section 2.1).

Definition 4.2. We consider that two plans are *trivially-equivalent* iff they differ only by the order of algebraic operations they perform, or by their projections.
It may happen that two plans differ only by an operation that has no effect on the resulting table (e.g.: the $\sigma_{a<10}(.)$ operation in $\sigma_{a<10}(\sigma_{a=5}(A))$). Without loss of generality, we assume this does not happen.

For instance, (A $\bowtie$ B) $\bowtie$ C is trivially-equivalent to A $\bowtie$ (B $\bowtie$ C), and therefore we will denote the set of all the possible plans to do those operations simply by A $\bowtie$ B $\bowtie$ C. Also, two plans differing only by projection operations are trivially-equivalent (assuming no essential information is lost during a projection).

An useful way to represent plans are by *n-ary unions* and *n-ary joins*, defined as follows:

Definition 4.3. We define *N-ary joins* as the operation of joining N tables with a set of join conditions, independently of the set of physical join operations. For instance, the n-ary join of tables A, B, C on join conditions $A.o = B.s \wedge B.o = C.s$ is trivially equivalent to joining at first A and B, then C; and joining at first B and C, then A.

Additionally, we can note the name of the columns of the new joined table by preceding an "=" condition between columns with [name]. For instance, $[column_1]A.o = B.s \wedge [column_2]B.o = C.s$ represents an n-ary join in which the table returned contains columns $column_1$ (which is derived from an "=" condition on columns A.o and B.s) and $column_2$ (which is derived from an "=" condition on columns B.o and C.s).

Definition 4.4. We define *N-ary unions* as a union of N tables.

Definition 4.5 (Optimization function)**.** We call *optimization function* $\mathcal{O} : \mathcal{S} \times \mathcal{Q} \times \mathcal{G} \rightarrow \mathcal{P}$ a function such that $\mathcal{O}(s, q, G)$ is the set of all non-trivially-equivalent plans evaluating query q on graph G using at most $T(G)$, $T_P(G)$ and the summary s.

More precisely, the problem to address is to define the optimization $\mathcal{O}(s, q, G)$ for different values of s, q and G.

Each algebraic plan built by $\mathcal{O}$ can be translated by existing software developed within the CEDAR team, into SQL queries, which will be sent for evaluation to a relational database management system (RDBMS). The RDBMS stores the data originating from the RDF graph, but is not aware of the fact that this represents graph data. A plan thus translated to a query evaluated by Postgres will leverage its existing cost-based optimization techniques.

A more complex problem statement is to *choose* which plan from the output of $\mathcal{O}$ is thus translated and evaluated. We will address this in a second stage of our work.

5 Exhaustive plan enumeration based on backward bisimulation summaries

We will focus on the backward bisimulation summaries due to experiments showing it more efficient than the backward bisimulation in the tested cases (some results from recent updates of the work to which we have access; unpublished at this time) [ČGM17].

We use the fact that nodes in a backward bisimulation summary are BB to their respective maximal sets of BB nodes in the original graph. We can therefore search for a certain path in the original graph by searching the same path in its backward bisimulation summary.

In this section, we will study possible plans for different shapes of queries and list the possible non-*trivially-equivalent* plans to process queries of given shapes.

For a graph G and a query q we will list in this section the elements of the set $\mathcal{O}(S, q, G)$. Each subsection gives $\mathcal{O}(S, q, G)$ to a specific shape of q (i.e.: to a specific subset of $\mathcal{Q}$).

5.1 Simple Linear-Path Queries

For a graph G and query $q \in S_{linear}$ we will list in this subsection the elements of the set $\mathcal{O}(S, q, G)$. First, note that we want to avoid enumerating a set of plans, if they are essentially equivalent to each other, that is, they can be obtained easily from one another through standard algebraic equivalence laws. From each set of plans thus obtained, we only want to see one. Specifically, we define:

5.1.1 Direct Joins on the Triple Table

We recall that a simple linear-path query is defined as a query of the form:

SELECT $?A_0$, ..., $?A_K$
WHERE {
    $?N_0$ P_0 $?N_1$.
    . . .
    $?N_{N-1}$ P_{N-1} $?N_N$.
}

such that $\{?A_0, ..., ?A_K\} \subset \{?N_0, ..., ?N_N\}$ and that $?N_i \neq ?N_j \; \forall$ i $\neq$ j $\in \{0, ..., N\}$.

Definition 5.1 (**Last-first join** $\overline{\bowtie}$)**.** Let T^N be the space of tables of N columns. We define the *last-first join operator*, noted $\overline{\bowtie}$, as follows: $\overline{\bowtie} : T^N \times T^M \rightarrow T^{N+M-1}$ such that for A $\in T^N$, B $\in T^M$, we have A $\overline{\bowtie}$ B is the table formed by the usual join operator ($\bowtie$) on the last column of A and the first column of B, yielding a table with the N columns of A at the same positions and the M-1 last columns of B at the last positions.

One plan is to simply use direct join on the triple table:

- Let $FullData = \sigma_{P=P_0}(T) \overline{\bowtie} \sigma_{P=P_1}(T) \overline{\bowtie} \ldots \overline{\bowtie} \sigma_{P=P_{N-1}}(T)$
- Then, $\mathcal{O}(S, q, G) \ni \pi_{?A_0,\ldots,?A_K}(FullData)$
- Additionally, $\{\ \sigma_{?A_0,\ldots,?A_K}(FullData)\ \}$ contains all possible plans for evaluating q using only $T(G)$.

5.1.2 Direct Joins on Pair tables

Another plan is to use direct join on pair tables.

- Let $FullData = T_{P_0} \overline{\bowtie} T_{P_1} \overline{\bowtie} ... \overline{\bowtie} T_{P_{N-1}}$
- We have $\mathcal{O}(S, q, G) \ni \pi_{?A_0,\ldots,?A_K}(FullData)$
- Additionally, $\{\ \pi_{?A_0,\ldots,?A_K}(FullData)\ \}$ contains all possible plans for evaluating q using at most $T_P(G)$.

5.1.3 Joins in Summary Table and Use of Indexes

We can leverage the backward bisimulation summary as follows:

Definition 5.2. Let us define the *full-extent* of a tuple of nodes ($?n_0$, ... , $?n_N$) in a summary as:

$$\text{full-extent}((?n_0, \ldots, ?n_N)) = \prod_{i=0}^{N} \text{extent}(?n_i)$$

Each node in the summary is BB to a maximal set of BB nodes in the original graph. We have therefore that for every tuple of nodes resulting of a linear path query in the original graph, that tuple is a selection of the full-extent of a tuple resulting from the same linear path query in the summary.

For example, let us consider the following query in the graph represented in figure 1.

```
SELECT ?n0, ?n1, ?n2
WHERE {
      ?n0 A n1 .
      ?n1 C n2 .
}
```

The result is the set of tuples {(1,2,6), (1,3,7), (1,4,8), (1,5,9)}.

The BB summary of that graph is represented in figure 2, with the notation "&{n_0,...,n_N}" meaning a node whose extent is the set {n_0,...,n_N} of nodes of the original graph. The query applied on the summary yields the result {(&{1},&{2,3},&{6,7}), (&{1},&{4,5},&{8,9})}. The full-extent of (&{1},&{2,3},&{6,7}) is (1,2,6), (1,2,7), (1,3,6), (1,3,7), and the full-extent of (&{1},&{4,5},&{8,9}) is {(1,4,8), (1,4,9), (1,5,8), (1,5,9)}.

Definition 5.3. We define the *full-extent* of a query q in a summary $S(G)$ as the union of the full-extents of the tuples obtained by evaluating q on $S(G)$.

The following property holds:

Property 5.1. The result of a query q in a graph G is equal to a selection of the full-extent of q on any quotient summary of G.

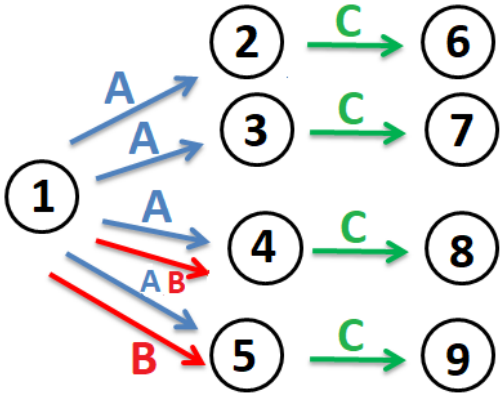

Figure 1. Original graph.

Proof. (property 5.1) We recall definition 3.3 (quotient summaries). Suppose there is a tuple t such that $t \in result(q, G)$, $S(G)$ a quotient summary of G.

$t \in result(q, G) \implies \forall$ WHERE triple $?N_iP_j?N_k$ in q, $?N_iP_j?N_k$ is satisfied by t and that there is an arc P_j from N_i to N_j in G.

$\underset{S \text{ is quotient}}{\implies} \forall$ WHERE triple $?N_iP_j?N_k$ in q, $?N_iP_j?N_k$ is satisfied by t and that there is an arc P_j in $S(G)$ from a node in $S(G)$ indexing N_i to a node in $S(G)$ indexing N_k.

$\implies t \in \text{full-extent}(q, S(G))$.

Therefore $\left(t \in result(q, G) \underset{\text{if } S \text{ quotient}}{\implies} t \in \text{full-extent}(q, S(G)) \right)$.

In other words, the result of a query q in a graph G is equal to a selection of the full-extent of q on any quotient summary of G. □

Corollary: As S, FBS, FS, and Strong Dataguides are quotient summaries, the result of a query q in a graph G is equal to a selection of the full-extent of q on any of those summaries.

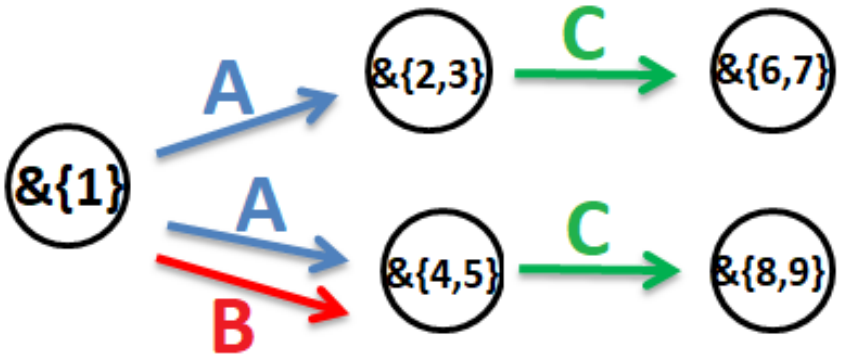

Figure 2. Backward bisimulation summary.

In the example, the query result is (1,2,6), (1,3,7), (1,4,8), (1,5,9)} and the summary-super-extent is {(1,2,6), (1,2,7), (1,3,6), (1,3,7), (1,4,8), (1,4,9), (1,5,8), (1,5,9)}.

We can then identify two more plans belonging to $\mathcal{O}(S, q, G)$:

1. Using $T(G)$:
   - $FullData = extent(?N_0) \overline{\bowtie} \sigma_{P=P_0}(T) \overline{\bowtie} extent(?N_1) \overline{\bowtie} \sigma_{P=P_1}(T) \overline{\bowtie} extent(?N_2) \overline{\bowtie} \ldots \overline{\bowtie} extent(?N_{N-1})$.
   - Then, $\mathcal{O}(S, q, G) \ni \pi_{?A_0,\ldots,?A_K}(FullData)$
2. Using $T_P(G)$:
   - Let $FullData = extent(?N_0) \overline{\bowtie} T_{P_0} \overline{\bowtie} extent(?N_1) \overline{\bowtie} T_{P_1} \overline{\bowtie} extent(?N_2) \overline{\bowtie} \ldots \overline{\bowtie} extent(?N_{N-1})$
   - We have $\mathcal{O}(S, q, G) \ni \pi_{?A_0,\ldots,?A_K}(FullData)$

Additionally, we can show that there is no other plan evaluating q using at most $T(G)$, $T_P(G)$ and $S(G)$ without direct join between two elements of $\{T(G)\} \cup T_P(G)$.

5.1.4 Hybrid Joins

Another plan to evaluate the query is to partition it into subqueries, then for each partition set evaluate it with one of the plans listed above, and then join together the results obtained from each part. Let H denote the set of those plans, then it follows that $\mathcal{O}(S, q, G) \supset H$.

For instance, in the example described above:

SELECT $?n_0$, $?n_1$, $?n_2$
WHERE {
  $?n_0$ A n_1 .
  $?n_1$ C n_2 .
}

we could partition this into:

{
  Q1 = "SELECT $?n_0$, $?n_1$ WHERE {$?n_0$ A n_1 .}",
  Q2 = "SELECT $?n_1$, $?n_2$ WHERE {$?n_1$ C n_2 .}"
}

Then we could use two different plans to perform each of those queries (e.g.: use "direct joins on pair tables" for Q1 and "joins in summary table and use of indexes" for Q2).

The result of Q_1 is ($?n_0$, $?n_1$) ϵ {(1,2), (1,3), (1,4), (1,5)}, and the result of Q_2 is ($?n_1$, $?n_2$) ϵ {(2,6), (3,7), (4,8), (5,9)}. The result of the original query is simply $Q_1 \bowtie Q_2$, in which $\bowtie$ is the usual join operator (and in this case will join on $?n_1$). The result of the original query will then be ($?n_0$, $?n_1$, $?n_2$) ϵ {(1,2,6), (1,3,7), (1,4,8), (1,5,9)}.

H is therefore the set of all the plans of the form:

1. determine $?N_i^S$s if needed
2. $\pi_{?A_0,\ldots,?A_K}\left(\overline{\bowtie}_0\ T'_{P_0}(G)\ \overline{\bowtie}_1\ \ldots\ \overline{\bowtie}_{N-1}\ T'_{P_{N-1}}(G)\ \overline{\bowtie}_N\right)$, in which

$$\overline{\bowtie}_0 = \begin{cases} \epsilon \text{ (empty string)} \\ \underline{extent(?N_0^S)\ \overline{\bowtie}} \end{cases}, \qquad \overline{\bowtie}_N = \begin{cases} \epsilon \text{ (empty string)} \\ \underline{\overline{\bowtie}\ extent(?N_N^S)} \end{cases}, \text{ and}$$

$$\overline{\bowtie}_i = \begin{cases} \overline{\bowtie} \\ \underline{\overline{\bowtie}\ extent(?N_i^S)\ \overline{\bowtie}} \end{cases} \forall i \in \{1, \ldots, N-1\}$$

Table 1. Form of hybrid joins plans.

In the above, each $\overline{\bowtie}_i$ is expressed as a choice between two options: one using the summary (shown as underlined) and one not using it. It is possible to just ignore the summary and rely on the T_P tables (this is the ϵ option), or join a table with the respective summary table to check that the node satisfies all the required conditions. $?N_i^S$s are determined by evaluating q (or a valid query whose set of WHERE triples is contained in that of q, see alternative discussion below) on $S(G)$.

Alternatives We note that there are several different plans for the same set of $\overline{\bowtie}_i$s chosen, because there are different (non-equivalent) plans for calculating $?N_i^S$s. For instance, for the query:

```
SELECT ?n0, ?n1, ?n2
WHERE {
    ?n0 A n1 .
    ?n1 B n2 .
}
```

in choosing $\overline{\bowtie}_0 = \epsilon$, $\overline{\bowtie}_1 = \overline{\bowtie}$ and $\overline{\bowtie}_1 = \overline{\bowtie}\ extent(?n_2^S)$, there are 4 non-equivalent plans differing by how they query $S(g)$ to calculate $?n_2^S$. We will use the following definitions to explicit those plans:

Definition 5.4 (T^S **and** T_P^S **tables**)**.** We note $T^S(g)$ a subject-predicate-object table with columns S, P, O that stores triple on $S(g)$. We also note $T_P^S(g)$ a subject-object table with columns S, O for predicate P. When both $T_{P_i}^S(g)$ and $\sigma_{P=P_i}(T^S(g))$ are valid, we use $T_{P_i}^{S\,\prime}(g)$ or $S(P_i)$ (when g and S are implicit) to represent any of them.

Definition 5.5 (**Representation table** $SRep$)**.** We note $SRep(g)$ a graphNode-summaryNode table with columns G, S such that it maps nodes in the original graph g with nodes in the summary $S(g)$ that represent them.

Using those last two definitions and the notation from definition 4.3, we can explicit the plans. For the query

```
SELECT ?n0, ?n1, ?n2
WHERE {
    ?n0 A n1 .
    ?n1 B n2 .
}
```

in choosing $\overline{\bowtie}_0 = \epsilon$, $\overline{\bowtie}_1 = \overline{\bowtie}$ and $\overline{\bowtie}_1 = \overline{\bowtie}\ extent(?n_2^S)$, there are non-equivalent plans differing by how they query $S(g)$ to calculate $?n_2^S$:

- $?n_2^S = \pi_{S(B).o}(S(B))$
- $?n_2^S = \pi_{S(B).o}(S(A) \overline{\bowtie} S(B))$

For each of those plans, let $newT$ denote the new joined table. There will be a column in $newT$ named $?n_2$. We obtain the plan by substituting:

$$extent(?n_2^S) = \pi_{SRep.G}(newT\ {}_{?n_2}\overline{\bowtie}_S\ SRep(g))$$

Definition 5.6. For a chosen set of $\overline{\bowtie}_i$ values, we denote the set of possible plans with those $\overline{\bowtie}_i$ values as $\mathcal{O}_{\{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\}}(S,q,G)$. We call each $\mathcal{O}_{\{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\}}(S,q,G)$ a $family$ of plans.

For instance, the 4 plans just presented above are from the same family of plans.

A basic property is that $\mathcal{O}(S,q,G)$ can be partitioned into families of plans. In other words, :

- $\mathcal{O}_{\{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\}}(S,q,G) \subset \mathcal{O}(S,q,G)$
  (i.e.: each family is a subset of $\mathcal{O}(S,q,G)$)
- $\bigcup_{\text{all } \{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\} \text{ options}} \mathcal{O}_{\{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\}}(S,q,G) = \mathcal{O}(S,q,G)$
  (i.e.: the union of all families is equal to $\mathcal{O}(S,q,G)$)
- $\mathcal{O}_{\{\overline{\bowtie}_0^i,\ldots,\overline{\bowtie}_N^i\}}(S,q,G) \cap \mathcal{O}_{\{\overline{\bowtie}_0^j,\ldots,\overline{\bowtie}_N^j\}}(S,q,G) = \emptyset\ \forall \{\overline{\bowtie}_0^i,\ldots,\overline{\bowtie}_N^i\} \neq \{\overline{\bowtie}_0^j,\ldots,\overline{\bowtie}_N^j\}$
  (i.e.: families are pairwise disjoint)

All plans $\in \mathcal{O}(S,q,G)$ that use a summary (i.e.: use at least one extent) are composed of two parts (or *subplans*):

1. determine N_i^Ss by evaluating q (or a valid query whose set of WHERE triples is contained in that of q) on $S(G)$.
2. $\pi_{?A_0,\ldots,?A_K}\left(\overline{\bowtie}_0\ T'_{P_0}(G)\ \overline{\bowtie}_1\ \ldots\ \overline{\bowtie}_{N-1}\ T'_{P_{N-1}}(G)\ \overline{\bowtie}_N\right)$

Part 1. is called *summary evaluation* **(we denote it by *summary evaluation subplan*, or *SES*)**. It uses the summary to determine N_i^S values for using in part 2.
Part 2 is called *graph evaluation* **(we denote it by *graph evaluation subplan*, or *GES*)**, and it uses the $?N_i^S$ values determined in part 1 to query on G.
All plans within a same family have the same expression for the original-graph evaluation part (the expression defined in table 1). However, the values for $?N_i^S$s appearing in those expressions may be different between plans from a same family, because they are results of different queries on S performed in the summary evaluation part of each plan.

For instance, for the query

```
SELECT ?n0, ?n1, ?n2
WHERE {
      ?n0 A n1 .
      ?n1 B n2 .
}
```

We have the following GES and SES:

GES:
$\bowtie_{E(?n_1).e=A.o}\ (A, B, E(?n_1))$
SES:
A

GES:
$\bowtie_{E(?n_1).e=A.o \wedge E(?n_0).e=A.s}\ (B, E(?n_1), A, E(?n_0))$
SES:
A

GES:
$\bowtie_{E(?n_1).e=A.o \wedge E(?n_0).e=A.s \wedge E(?n_1).e=B.s}\ (B, E(?n_1), A, E(?n_0))$
SES:
$\bowtie_{S(A).o=S(B).s}\ (A, B)$

GES:
$\bowtie_{E(?n_2).e=B.o \wedge E(?n_1).e=A.o \wedge E(?n_0).e=A.s \wedge E(?n_1).e=B.s}\ (B, E(?n_2), E(?n_1), A, E(?n_0))$
SES:
$\bowtie_{S(A).o=S(B).s}\ (A, B)$

GES:
$\bowtie_{E(?n_1).e=A.o \wedge E(?n_0).e=A.s \wedge E(?n_2).e=B.o}\ (E(?n_1), A, E(?n_0), E(?n_2), B)$
SES:
$\bowtie\ (A, B)$

GES:
$\bowtie_{E(?n_1).e=A.o \wedge E(?n_1).e=B.s}\ (E(?n_1), A, B)$
SES:
$\bowtie_{S(A).o=S(B).s}\ (A, B)$

GES:
$\bowtie_{E(?n_1).e=A.o \wedge E(?n_1).e=B.s \wedge E(?n_2).e=B.o}\ (E(?n_1), A, E(?n_2), B)$
SES:
$\bowtie_{S(A).o=S(B).s}\ (A, B)$

GES:
$\bowtie_{E(?n_1).e=A.o \wedge E(?n_2).e=B.o}\ (E(?n_1), A, B, E(?n_2))$
SES:
$\bowtie\ (A, B)$

GES:
$\bowtie_{E(?n_0).e=A.s}\ (A, B, E(?n_0))$
SES:
A

GES:
$\bowtie_{E(?n_0).e=A.s \wedge E(?n_1).e=B.s} (E(?n_1), A, B, E(?n_0))$
SES:
$\bowtie (A, B)$

GES:
$\bowtie_{E(?n_0).e=A.s \wedge E(?n_2).e=B.o \wedge E(?n_1).e=B.s} (E(?n_1), A, E(?n_0), B, E(?n_2))$
SES:
$\bowtie (A, B)$
$\bowtie_{S(A).o=S(B).s} (A, B)$

GES:
$\bowtie_{E(?n_0).e=A.s \wedge E(?n_2).e=B.o} (A, E(?n_0), B, E(?n_2))$
SES:
$\bowtie (A, B)$
$\bowtie_{S(A).o=S(B).s} (A, B)$

GES:
$\bowtie_{E(?n_1).e=B.s} (E(?n_1), A, B)$
SES:
B

Now we will find the possible summary evaluation subplans for a given family (i.e.: a given $\{\overline{\bowtie}_0, \ldots, \overline{\bowtie}_N\}$ set of chosen options). We will denote the set of such subplans as $\mathcal{O}^{-}_{\{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\}}(S, q, G)$.

Theorem 5.1. $\mathcal{O}(S, q, G)$ is the union of all plans having q_{SES} as their summary evaluation subplan such that q_{SES} meets the following conditions:

1. the set of WHERE triples of q_{SES} is a subset of the set of WHERE triples of q,

2. the graph formed by the set of WHERE triples of q_{SES} contains all $?N_i^S$s nodes whose extents are used in the original-graph evaluation subplan.

3. the graph formed by the set of WHERE triples of q_{SES} has no arc *disconnected* from all $?N_i^S$s nodes whose extents are used in the original-graph evaluation subplan,
(We consider two nodes are connected iff there is a path of arcs, ignoring their orientation, between them. An arc is connected to a node iff at least one of its nodes is.)

Proof. (theorem 5.1) We want to prove that each plan with a q_{SES} that follows the theorem's conditions appears exactly once in $\mathcal{O}(S, q, G)$. This is equivalent to prove that every plan yielded by a q_{SES}: (A): is present in $\mathcal{O}(S, q, G)$, and (B): is not trivially equivalent to another plan yielded by a q_{SES} following the theorem's conditions. (a plan is valid iff it follows (A) and (B)).

1. If there is an WHERE triple of q_{SES} not present in q, then the result of q_{SES} may not contain nodes whose extents have a combination present on the result of q ($\implies$ (A) is false $\implies$ q_{SES} is invalid).

2. If there is a $?N_i^S$ node whose extent is used in the original-graph evaluation subplan that is not contained in any WHERE triple of q_{SES}, then the plan is trivially-equivalent to a changed version of itself in which q_{SES} does not use that $?N_i^S$ node and the original-graph evaluation subplan does not use the summary on $?N_i^S$ ($\implies$ (B) is false $\implies$ q_{SES} is invalid).

3. If there is an WHERE triple of q_{SES} which is an arc in $S(G)$ disconnected from all $?N_i^S$s nodes whose extents are used in the original-graph evaluation subplan, then that triple is trivially equivalent to a changed version of itself in which all disconnected triples of q_{SES} are discarded ($\implies$ (B) is false $\implies$ q_{SES} is invalid).

We reach therefore that if at least one of the conditions is not met, then q_{SES} is invalid. Now we will prove that if all conditions are met, then q_{SES} is valid.

If the first condition is met, the result of q_{SES} on $S(G)$ contains the result of q on $S(G)$. This implies that q_{SES} yields a plan whose result is in $\mathcal{O}(S, q, G)$ ($\implies$ (A) is true). Additionally, if the second and third conditions are met, the plan is not trivially equivalent to another plan yielded by a q_{SES} following the theorem's conditions ($\implies$ (B) is true).

Therefore, q_{SES} is valid iff the three conditions are met.

Therefore each plan with a q_{SES} that follows the theorem's conditions appears exactly once in $\mathcal{O}(S, q, G)$. □

Table 2 presents an algorithm to calculate $\mathcal{O}^-_{\{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\}}(S, q, G)$, i.e.: to list all the distinct summary evaluation subplans for a chosen set of $\{\overline{\bowtie}_0, \ldots, \overline{\bowtie}_N\}$ (i.e.: for a same family).

```
receive query q and chosen options {⋈̄_0, ..., ⋈̄_N} as input;
construct immutable qGraph as graph of WHERE triples of q;
construct immutable nodesWithExtents as set of the nodes that have their extents
used in {⋈̄_0, ..., ⋈̄_N};
construct variable currSES initialized as empty graph of N-joins;
construct variable toReturn initialized as empty list of graph of N-joins;
call auxiliary function DFS;
return toReturn;

DFS:
    if currSES contains nodesWithExtents:
        add currSES to toReturn;
    for each arc A in qGraph connected to currSES or to nodesWithExtents:
        join A on currSES
        call DFS;
        remove A join on currSES
```

Table 2. Summary evaluation subplans listing algorithm.

We will now prove that the algorithm returns $\mathcal{O}^-_{\{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\}}(S, q, G)$.

Proof. Each summary evaluation subplan it returns follows the three conditions of theorem 5.1: It follows condition (1) because all arcs it adds to its q_{SES} are contained in A. It follows condition (2) because it only adds to *toReturn* subplans that contain *nodesWithExtents*.

And it follows condition (3) because it only adds arcs to its q_{SES} if they are connected to $currSES$ or to $nodesWithExtents$. Therefore all summary evaluation subplans returned $\in \mathcal{O}^{-}_{\{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\}}(S,q,G)$.

Additionally, the algorithm performs an exhaustive depth first search on all the arcs connected to $CurrSES$ or to $nodesWithExtents$, adding each possible subplan exactly once to $toReturn$.

Therefore the algorithm returns $\mathcal{O}^{-}_{\{\overline{\bowtie}_0,\ldots,\overline{\bowtie}_N\}}(S,q,G)$.

□

For each of those plans, let the new joined table be noted $newT$. For each column $?n$ whose extent is used on newT, we obtain the plan by substituting:

$$extent(?n^S) = \pi_{SRep.G}(newT\ {}_{?n}\overline{\bowtie}_S\ SRep(g))$$

For each chosen set of $\overline{\bowtie}_i$ values, the number of possible plans is between 1 and 2^N (inclusive). Therefore $2^{N+1} \leqslant \#\mathcal{O}(S,q,g) \leqslant 2^{2N+1}$ when either T or T_P is available and $4^{N+1} \leqslant \#\mathcal{O}(S,q,G) \leqslant 2^N 4^{N+1}$ when both are available.

For every plan p evaluating $q \in S_{linear}$ using at most $T(G)$, $T_P(G)$ and $S(G)$, p can be partitioned into smaller plans such that each one of them will either use only $T(G)$; or use only $T_P(G)$; or do not use direct join between two elements of $\{T(G)\} \cup T_P(G)$. As for each of those cases we presented on previous subsections the complete set of possible evaluating plans, H is the complete set of possible evaluating plans. $\mathcal{O}(S,q,G) = H$.

5.2 Simple Queries

For a graph G and query $q \in S_{simple}$ we will list in this subsection the elements of the set $\mathcal{O}(S,q,G)$.

The plans $\in \mathcal{O}(S,q,G)$, for q a simple query are analogous to the plans $\in \mathcal{O}(S,q,G)$ for q a linear query, but having as set of GES the plans produced by the following algorithm:

```
Start with GES oges = GES of g without using summaries;
repeat the following operation as many times as wished:
    for a node ?n not having its summary used:
        //(use the summary of ?n)
        substitute (on oges) that ?n by extent(?n^S)
        for each table being joined on ?n:
            join that table (on oges) with extent(?n);
            delete the join between that table and ?n;
return oges;
```

Table 3. GES listing algorithm for given simple query.

For instance, the query

```
SELECT ?n0, ?n1, ?n2, ?n3
WHERE {
      ?n0 A n1 .
      ?n0 B n2 .
      ?n0 C n3 .
}
```

has 16 possible GES (we note A (*resp.* B, C) $= T'_{A\ (resp.\ B,C)}(g)$, and E(?n) $= extent(?n^S)$ for simplicity:

1. $\bowtie_{A.s=B.s=C.s}\ (A, B, C)$
2. $\bowtie_{E(?n_0).e=A.s \wedge E(n_0).e=B.s \wedge E(?n_0).e=C.s}\ (C, A, B, E(?n_0)$
3. $\bowtie_{E(?n_0).e=A.s \wedge E(?n_0).e=B.s \wedge E(?n_3).e=C.o \wedge E(?n_0).e=C.s}\ (A, B, E(?n_0), E(?n_3), C)$
4. $\bowtie_{E(?n_1).e=A.o \wedge E(?n_0).e=A.s \wedge E(?n_0).e=B.s \wedge E(?n_3).e=C.o \wedge E(?n_0).e=C.s}\ (E(?n_1), A, B, E(?n_0), E(?n_3), C)$
5. $\bowtie_{E(?n_1).e=A.o \wedge E(?n_2).e=B.o \wedge E(?n_0).e=A.s \wedge E(?n_0).e=B.s \wedge E(?n_3).e=C.o \wedge E(?n_0).e=C.s}$
   $(E(?n_1), A, E(?n_2), B, E(?n_0), E(?n_3), C)$
6. $\bowtie_{E(?n_2).e=B.o \wedge E(?n_0).e=A.s \wedge E(?n_0).e=B.s \wedge E(?n_3).e=C.o \wedge E(?n_0).e=C.s}\ (A, E(?n_2), B, E(?n_0), E(?n_3), C)$
7. $\bowtie_{E(?n_0).e=A.s \wedge E(?n_0).e=B.s \wedge E(?n_0).e=C.s \wedge E(?n_1).e=A.o}\ (C, E(?n_0), A, B, E(?n_1))$
8. $\bowtie_{E(?n_2).e=B.o \wedge E(?n_0).e=A.s \wedge E(?n_0).e=B.s \wedge E(?n_0).e=C.s \wedge E(?n_1).e=A.o}\ (C, E(?n_2), B, E(?n_0), A, E(?n_1))$
9. $\bowtie_{E(?n_0).e=A.s \wedge E(?n_2).e=B.o \wedge E(?n_0).e=B.s \wedge E(?n_0).e=C.s}\ (A, C, E(?n_0), E(?n_2), B)$
10. $\bowtie_{A.s=B.s=C.s \wedge E(?n_3).e=C.o}\ (A, B, C, E(?n_3))$
11. $\bowtie_{A.s=B.s=C.s \wedge E(?n_3).e=C.o \wedge E(?n_1).e=A.o}\ (B, A, C, E(?n_3), E(?n_1))$
12. $\bowtie_{A.s=B.s=C.s \wedge E(?n_3).e=C.o \wedge E(?n_2).e=B.o \wedge E(?n_1).e=A.o}\ (A, B, C, E(?n_3), E(?n_2), E(?n_1))$
13. $\bowtie_{A.s=B.s=C.s \wedge E(?n_2).e=B.o \wedge E(?n_3).e=C.o}\ (A, B, C, E(?n_2), E(?n_3))$
14. $\bowtie_{A.s=B.s=C.s \wedge E(?n_1).e=A.o}\ (A, B, C, E(?n_1))$

15.

$$\bowtie_{A.s=B.s=C.s \wedge E(?n_1).e=A.o \wedge E(?n_2).e=B.o} (A, B, C, E(?n_1), E(?n_2))$$

16.

$$\bowtie_{A.s=B.s=C.s \wedge E(?n_2).e=B.o} (A, B, C, E(?n_2))$$

In fact, the number of GES is equal to 2^N, where N is the number of variables. This is because, for given query q and graph g, there a node can "have or not have" its extent used. There are 2 possibilities for each node (independently of the other nodes), therefore there are 2^N different GES for given query q and graph g.

Except for the set of GES, all the discussion and results given in the Simple Linear-Path Queries subsection are valid for general simple queries, including the summary exploration subplans and the calculation of the extents by:

$$extent(?n^S) = \pi_{SRep.G}(newT \ {}_{?n}\overline{\bowtie}_S \ SRep(g))$$

where $newT$ is the result of the SES n-ary join.

5.3 SPARQL Queries with Regular Path Expressions

For a graph G and query $q \in S_{SPARQL}$ we will list in this subsection the elements of the set $\mathcal{O}(S, q, G)$.

We will use an auxiliary function to decompose the given query $\in S_{SPARQL}$ into a set of queries $\subset S_{simple}$.

Definition 5.7. We define $\mathcal{QS} : S_{SPARQL} \rightarrow Powerset(S_{simple})$ a function such that for $\mathcal{QS}(Complex) = Simple$, the evaluation of Complex is equal to the union of evaluations of queries in Simple. (The $Powerset$ of a set is the set of all its subsets).

Let N be a positive integer. Then, $\mathcal{QS}$ can be implemented by the algorithm shown in Table 4. As we can call the algorithm with N arbitrarily large, we have therefore $\mathcal{O}(S, q, G) \supset \mathcal{QS}(q)$.

receive Complex $\in S_{SPARQL}$ as input;
create variable Curr $\leftarrow \{Complex\} \in Powerset(S_{SPARQL})$;
while (true):
for some RPEs A, B, C:
If $Complex$ contains an instance of A(B)?C, then Curr $\leftarrow$ (Curr transforming each query in it to {AC, ABC});
If $Complex$ contains an instance of A(B)*C, then Curr $\leftarrow$ (Curr transforming each query in it to {AC, A(B)C, A(BB)C, ..., A $(\underbrace{B \ldots B}_{N})$ C});
If $Complex$ contains an instance of A(B)+C, then Curr $\leftarrow$ (Curr transforming each query in it to {A(B)C, A(BB)C, ..., A $(\underbrace{B \ldots B}_{N})$ C});
if no "if" condition has been activated in this iteration, then return Curr.

Table 4. Query decomposition algorithm.

We will now proof that the algorithm terminates and returns $\mathcal{QS}(Complex)$.

Proof. Curr is initialized as a set containing only the input ($Complex$). the union of evaluations of queries in Curr is a loop-invariant (because in each iteration queries are substituted according to the definition of '?', '*', and '+'). And the number of '?', '*' and '+' symbols in a Curr query is a loop-variant starting ≥ 0 and decreasing by 1 in each iteration until reaching 0. Therefore the algorithm terminates and returns $\mathcal{QS}(Complex)$. □

6 Implementation

We recall that for a SPARQL query (maybe with regular-path expressions), we can divide that query in a set of simple queries (without regular-path expressions). For each simple query, we can partition the set of its resulting non-trivially-equivalent plans into families. Plans within a same family share a common Graph Evaluation Subplans (GES), and each of those plans has a Summary Evaluation Subplan.

It is therefore interesting to write a modular code, with separate methods that can each independently perform one of the following operations:

1. For a given SPARQL query (maybe with regular-path expressions), list its decomposition queries (the simple queries, without regular-path expressions, that satisfy the SPARQL query).
2. For a given simple query, list its GE subplans.
3. For a given GE subplan, list its SE subplans.

This way, an user can do any of the following operations:

- call the three methods to get all plans for a given SPARQL query,
- call only one method to get all simple queries for a given SPARQL query,
- call only two methods to get all families of plans for a given SPARQL query,
- call only one method to get all plans within a given family,
- etc.

We wrote code with this modular property to implement the algorithms proposed on this paper. The code totals approximately 700 lines of C++ in a header and a cpp file (files CJTgraph.h and CJTgraph.cpp), and is explained in the following subsections.

It can be retrieved here: https://github.com/ICEtinger/SPARQL_BBS_plans.

6.1 Data Structures

In this section we will explain the data structure used to implement the methods.

6.1.1 Includes and typedefs

All arrays, vectors, strings, unordered_maps, unordered_sets and stacks used and referred to in this section are from the standard C++ libraries (e.g.: std::array, std::vector, ...).

We use the following standard libraries:

```
#include <cstdio>
#include <array>
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <functional>
#include <stack>
```

Table 5. Includes from standard library

We define two types. The first is used to represent a triple (subject, predicate, object), and is an array of 3 strings. The positions 0, 1 and 2 contain respectively the subject, predicate and object of the represented triple.

The second is a bitwise representation of an array of bits. We call it a *bitArray* and define it as an int_fast64_t, which is a type required by the C++ standard and that is the fastest type possible containing at least 64 bits (on most systems it should be an integer of size exactly 64 bits). A further description of the bitArray is given in the "List Summary Exploration Subplans for given GES" subsection.

```
// each triple is (subject, predicate, object)
typedef std::array<std::string, 3> triple;

// represents an array of 64 bits.
typedef int_fast64_t bitArray;
```

Table 6. Type definitions

6.1.2 Column-Join-Table Bipartite Graph

Definition 6.1. We define a *column-join*, also noted *cjoin* as the operation of joining N nodes over a same column. We denote $\bowtie\!\!\!\!\bowtie(T_1(a_1), \ldots, T_N(a_N))$ as the join of tables $T_1, \ldots, T_N$ on their respective columns $a_1, \ldots, a_N$.

For instance, $\bowtie\!\!\!\!\bowtie(A(a), B(b), C(c))$ is trivially equivalent to all of the following:

1. $A \ {}_a\!\bowtie_b \ B \ {}_b\!\bowtie_c C$
2. $A \ {}_a\!\bowtie_c \ C \ {}_c\!\bowtie_b B$
3. $B \ {}_b\!\bowtie_a \ A \ {}_a\!\bowtie_c C$
4. $B \ {}_b\!\bowtie_c \ C \ {}_c\!\bowtie_a A$
5. $C \ {}_c\!\bowtie_a \ A \ {}_a\!\bowtie_b B$
6. $C \ {}_c\!\bowtie_b \ B \ {}_b\!\bowtie_a A$

As we can see, column-joins are very useful to succinctly represent sets of trivially-equivalent multiple join operations. To make the representation of plans as succinct as possible, we will represent each plan as the following data structure:

Definition 6.2. We define a *Column-Join-Table bipartite graph*, also noted CJT graph, as a graph in which:

- nodes are either a column-join or a table,
- all edges are between a column-join and a table (thus the graph is bipartite),
- an edge between a table T and a column-join CJ is labeled with the column of T taking part in CJ.

Table 7 depicts its definitions:

```
struct Table {
    std::string name;
    std::unordered_set<CJoin*> edges;
    Physical_table* phys_table;
};

struct CJoin {
    std::string name;
    std::vector< std::pair<Table*, char> > edges;
};
```

Table 7. data structure definitions for tables and cjoins

We store the edges of a table as an std::unordered_set because the complexity of accessing an edge (with a specific CJoin) from a Table is a critical performance factor in the algorithm of listing GES for given simple query. Conversely, we store edges of a CJoin as an std::vector because we do not need efficient accessing to an edge with a specific Table from a CJoin. We only need an iterable collection class, and std::vector is the most compact one available. We only store the column of the Table joined in the edges of CJoin to avoid redundancy and unnecessary operations.

Every Table can have a pointer to a Physical_table, which would be a structure containing the data (and possibly statistics) of the real table being represented by that Table object. If a Physical_table is used, then the TGraph can be used to evaluate column joins of the query (or evaluate all the query if all cjoins are evaluated). Alternatively, one can also choose not to use a Physical_table, because it is not required in order to perform the operations: list SES for given GES, list GES for given simple query, and list simple queries for given SPARQL query

```
struct Physical_table { /* e.g.: std::vector<std::tuple<>> data; int statistics; */ };
```

Table 8. Example of physical table implementation

We then create the CJTgraph class. It stores pointers to its Tables and CJoins and *owns* them (i.e.: deletes them from memory on destruction). There are many auxiliary methods implemented that are not represented here (as methods to efficiently add edges to the graph, efficiently copy a graph and create a mapping between both graphs, etc).

```
class CJTgraph {
private:
    std::unordered_set<Table*> tables;
    std::unordered_set<CJoin*> cjoins;
public:
    void evaluate(CJoin* cj);
    ...
};
```

Table 9. Basic structure of CJTgraph class

In the case where Physical_tables are used, one can evaluate a column-join in the query by calling the *evaluate* method of a CJTgraph. That method substitutes the N-join and all the tables it has edges with by a new table (equal to its evaluation), and redirects all edges from the substituted tables to the new table.

evaluate calls the method physical_cjoin_evaluator, which gives the physical result of a column-join between given physical tables in their given columns. Additionally, it gives the position of the original columns (of each table being joined) in the new joined table. If that information was lost, it would be impossible to interpret the new joined table or to further join it with other tables.

This is detailed in Table 5.1.3:

```
// first element of returned pair: a pointer to the resulted Physical_table
// second element: vector<vector<char> M such that M[i][j] is the new column posi-
tion (in the resulted table) of column j of v[i].
std::pair<Physical_table*, std::vector<std::vector<char>> > physical_cjoin_evaluator
(std::vector<Physical_table*> v, std::vector<char> positions_columns_to_join) {
    // implementation
}
```

Table 10. Structure of physical cjoin evaluator

6.2 List Simple Queries for Given SPARQL Query

In this section we will describe the method used to list the simple queries (i.e.: without regular-path expressions) yielded by the decomposition of a SPARQL query.

```
static std::vector<std::vector<triple>> decompose_query(const std::string& q,
const int limit_regular_exp);
```

Table 11. Structure of decompose query function

The method in question, *decompose_query* takes as parameter a SPARQL query (possible with regular-path expressions) as a string and an integer that represents the maximum number of repetitions to be considered for the "*" and "+" expressions. For instance, if the limit is 5, the method considers "A+" as {"A", "AA", "AAA", "AAAA", "AAAAA"}. The method returns a vector of vector of triple. I.e.: a vector of simple queries (yielded from the decomposition), in which each simple query is a vector of WHERE triples.

The method begins by parsing the SPARQL query into a vector of WHERE triples (possibly containing RPE). E.g.: it parses

```
SELECT ?n0, ?n1, ?n2
WHERE {
    ?n0 A+ n1 .
    ?n1 B n2 .
}
```

into:

```
vector<triple>{
    {"?n0", "A+", "n1"},
    {"?n1", "B", "n2"}
}
```

The method then creates an empty vector of vector of triples *total_decomposed* as a vector of an empty vector of triples. That variable will be changed during the function's execution and then will be returned. We use vector because it is the most compact iterable collection class, but order of elements is not important. total_decomposed is a listing of simple queries (to be returned), and is initialized as listing a single query, which is empty.

```
std::vector<std::vector<triple>> total_decomposed std::vector<triple>{} ;
```

Table 12. Structure of total_decomposed auxiliary function

We then repeat the following operation: for each triple parsed from the input, we decompose it and set-multiply total_decomposed by it. For the input of:

```
{
    {"?N0", "A+", "?N1"},
    {"?N1", "B", "?N2"}
}
```

the following states are present:

1. total_decompose is initialized as a listing containing only an empty query.
2. total_decompose is set-multiplied by {"?N0 A ?N1", "?N0 AA ?N1", "?N0 AAA ?N1", "?N0 AAAA ?N1", "?N0 AAAAA ?N1"}.
3. total_decompose is now {"?N0 A ?N1", "?N0 AA ?N1", "?N0 AAA ?N1", "?N0 AAAA ?N1", "?N0 AAAAA ?N1"}.
4. total_decompose is set-multiplied by {"?N1 B ?N2"}
5. total_decompose is now {"?N0 A ?N1 B ?N2", "?N0 AA ?N1 B ?N2", "?N0 AAA ?N1 B ?N2", "?N0 AAAA ?N1 B ?N2", "?N0 AAAAA ?N1 B ?N2"}.
6. total_decompose is returned.

For decomposing a query, the method calls an auxiliary function *decompose_predicate*, which takes a predicate string as input, tokenizes it (e.g.: transforms "(pred1/pred2)?/pred3+" into ("(", "pred1", "/", "pred2", ")", "?", "/", "pred3", "+")), then iterates through the tokens applying appropriates operations for each token (between "?", "*", "+", "|" , "/", "(", ")", or predicate name). It uses stacks to treat parenthesis "(" and ")". For "?", "*" and "+" it uses set-multiplication. And for "|" and "/" it uses set union or concatenation of predicates/triples. The implementation uses C++11's move semantics to efficiently perform those operations.

For transforming composed predicates into simple triples, extra variables are added. For instance, triple{"?N0", "ABC", "?N1"} results in vector{triple{"?N0", "A", "?extra1"}, triple{"?extra1", "B", "?extra2"}, triple{"?extra2", "C", "?N1"} }

6.3 List Original-Graph Evaluation Subplans for Given Simple Query

In this section we will describe the method used to list the original-graph evaluation subplans for a given simple query.

static void for_each_GES(const std::vector<triple>& q, std::function<...> function_to_apply);

Table 13. Structure of main function for listing GES for given simple query

The *for_each_GES* is a void function that takes as parameter a simple query and a function to apply on each GES. The function to be applied is a void, and takes will be called having as parameters variables constructed (and changing) during the function execution.

We chose to have a foor_each function instead of returning a listing of GES to increase modularity, maintainability and speed, and to lower memory usage. As the number of different GES for a given simple query is equal to 2^N, there could be memory issues arising from returning a collection of graphs. For N = 40 nodes, there are $2^{40} (\approx 10^{12})$ different GES. The storage of those GES would take at least 4 terabytes, and be therefore clearly impractical.

The method has worst-case complexity of $O(\frac{m}{N}\ 2^N)$, where m is the number of triples. If we consider $\frac{m}{N}$ to be $O(1)$, which holds true for almost all SPARQL queries typically used (e.g.: any query whose WHERE triples form a tree), then the method worst-case complexity is $O(2^N)$. An interesting result is that **that worst-case complexity is the theoretical lower bound for the problem of implementing a function that explores all GES** (for each GES in the result, the function has to explore it at least once). In fact, the method is so efficient that even printing all explored GES would make the complexity (N + m) times slower (it would be $O((N + m)\ 2^N)$).

We will now prove that the worst case complexity for that implementation is $O(\frac{m}{N}\ 2^N)$.

Proof. The method passes through the following states:

1. At first the methods initializes auxiliary variables with complexity $O(N + m)$.
2. The method starts with the GES that does not uses any summary.
3. The method then performs a DFS on the hypercube of possible GES (in which each dimension of the hypercube is whether a variable is having its summary used or not).

During the DFS on the hypercube of possible GES, each CJoin in the CJTgraph has the operation "set its summary to be used; wait for other CJoins; set its summary to not be used" a number of times $\in \{1, 2, 4, \ldots, 2^{N-1}\}$. The i^{th} CJoin performs that operation 2^i times, with $i \in \{0, \ldots, N-1\}$. This is analogous to how many times a bit of an *int* is changed when that int goes from 0 to 2^N by consecutively being increased by 1.
Each operation performed by CJoin cj has complexity $O(degree(cj))$, in which degree(cj) is the number of edges of cj. The total complexity is therefore $O(degree(cj_0)\ 2^0 + \ldots + degree(cj_{N-1})\ 2^{N-1})$. To reduce the number of operations performed with CJoins of high degrees, the method creates a vector of the cjoins to be applied the DFS on, and sorts that vector by the CJoins degrees. The worst-case complexity is therefore when the degrees are equally distributed among each CJoins. The worst case complexity is then $O(\frac{m}{N}\ (1 + 2 + 4 + ... + 2^{N-1})) = O(\frac{m}{N}\ 2^N)$. □

Table 14. Algorithm explanation and proof of complexity

There are many interesting possibilities of functions for being applied on for_each_GES. One can use a function to count the total number of GES:

```
int CJTgraph::count_all_GES(const std::vector<triple>& q) {
    int count = 0;
    for_each_GES(q, [&count](...) {++count; });
    return count;
}
```

Table 15. Function to count all GES

One can use a function to print all GES:

```
void CJTgraph::print_all_GES(const std::vector<triple>& q, FILE * pFile = std-
out, char format = 'O') {
    for_each_GES(q, [&pFile, &format](...){GES.print_itself(pFile, format); });
}
```

Table 16. Function to print all GES

In this case, we use the *print_itself* function of a CJTgraph. That function makes a CJTgraph object prints itself into a file (or default stdout) in one of 3 available formats: the GraphViz format for graph visualization, the n-ary join format for GES, and the n-ary join format for SES.

For instance, for the SPARQL query

```
SELECT ?N0, ?N1, ?N2
WHERE {
      ?N0 A ?N1 .
      ?N1 B ?N2 .
}
```

the n-ary join format gives the following result:

1. $\bowtie_{A.o=B.s} (A, B)$
2. $\bowtie_{A.o=B.s \wedge E(?n_0).e=A.s} (A, B, E(?n_0))$
3. $\bowtie_{A.o=B.s \wedge E(?n_2).e=B.o \wedge E(?n_0).e=A.s} (A, E(?n_2), B, E(?n_0))$
4. $\bowtie_{E(?n_1).e=A.o \wedge E(?n_2).e=B.o \wedge E(?n_0).e=A.s \wedge E(?n_1).e=B.s} (A, E(?n_1), B, E(?n_2), E(?n_0))$
5. $\bowtie_{E(?n_0).e=A.s \wedge E(?n_1).e=A.o \wedge E(?n_1).e=B.s} (B, E(?n_0), E(?n_1), A)$
6. $\bowtie_{A.o=B.s \wedge E(?n_2).e=B.o} (A, E(?n_2), B)$
7. $\bowtie_{E(?n_2).e=B.o \wedge E(?n_1).e=A.o \wedge E(?n_1).e=B.s} (E(?n_1), A, B, E(?n_2))$
8. $\bowtie_{E(?n_1).e=B.s \wedge E(?n_1).e=A.o} (A, E(?n_1), B)$

If we use the GraphViz format for printing, we get the results shown in Figure 3. The figure shows GES in the Table-CJoin Graph representation, which is the internal structure our implementation uses to process GES. The red number assigned to each family in the figure refers to the number of the 8 GES just listed above.

Finally, one can also use a function to get the set of all SES for a given GES, and perform an operation on it:

```
void CJTgraph::print_all_SES_and_GES(const std::vector<triple>& q, FILE * pFile
= stdout, char format = 'O') {
      for_each_GES(q, [&pFile, &format](...) {
            GES.print_itself(pFile, format);
            fprintf(pFile, "SES:\ n");
            print_all_SES_for_given_GES(..., pFile, 'S');
            fprintf(pFile, "\ n");
      });
}
```

Table 17. Function to print all SES and GES

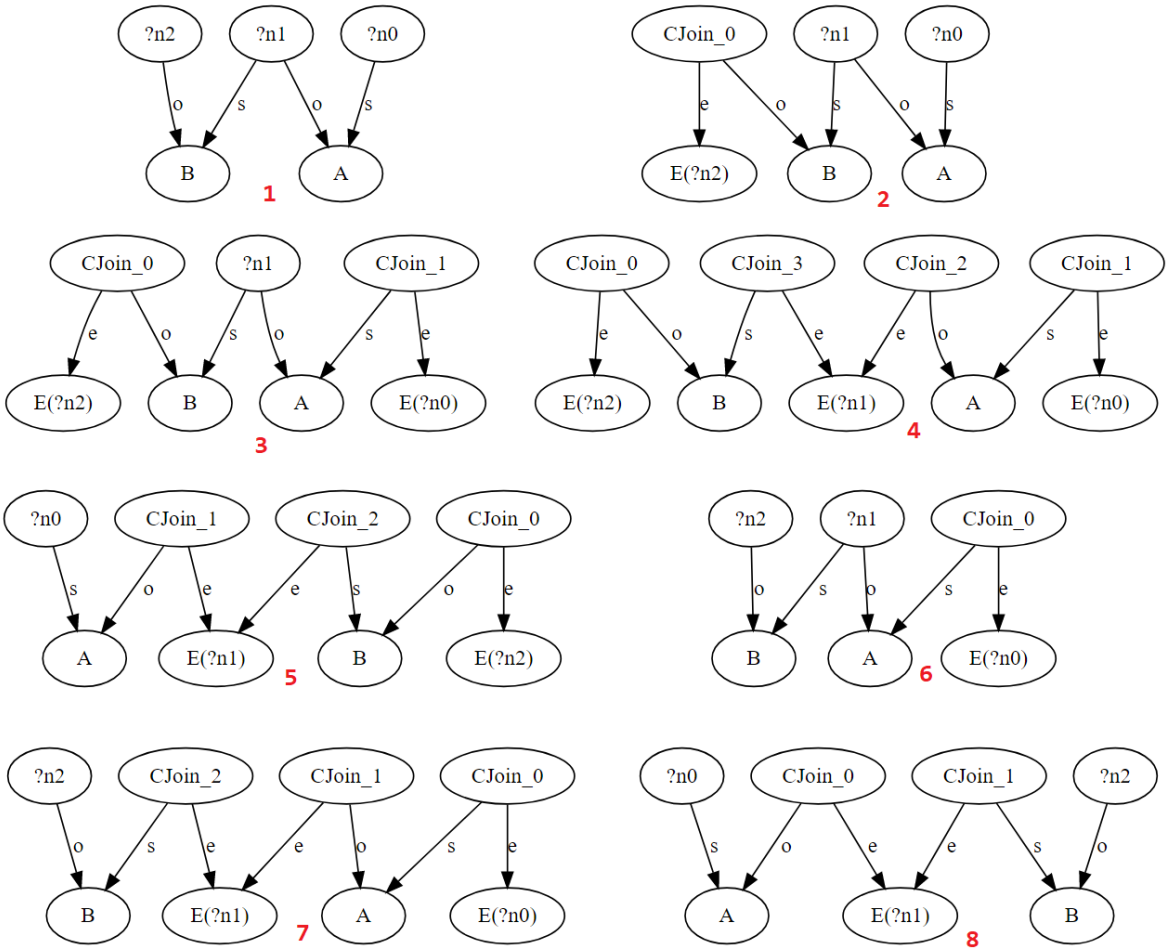

Figure 3. The 8 families in Table-CJoin Graph representation

Here the function *print_all_SES_for_given_GES(...)* uses a method which will be discussed on the next subsection:

6.4 List Summary Evaluation Subplans for given GES

In this section we will describe the method used to list the summary evaluation subplans for a given GES.

static std::unordered_set<bitArray> SES_set(CJTgraph& GES, ...);

Table 18. Function returning a set of all SES for a given GES

A SES is a subgraph of the simple query CJTgraph. We numerate the edges of the simple query CJTgraph, and we represent a SES by an array of bits in which the i^{th} bit is true iff the i^{th} edge is present in the subgraph. The array of bits is implemented as an int_fast64_t, which is a type required by the C++ standard and that is the fastest type

possible containing at least 64 bits (on most systems it should be an integer of size exactly 64 bits). In this we assume that the number of edges will be at most 64, which is usually the case for queries evaluated in SPARQL. In the case of a bigger query, it suffices to change the definition of bitArray to one of the following types implemented by C++'s Boost:

```
#include <boost/multiprecision/cpp_int.hpp>

typedef boost::multiprecision::int128_t bitArray;
//typedef boost::multiprecision::int256_t bitArray;
//typedef boost::multiprecision::int512_t bitArray;
//typedef boost::multiprecision::int1024_t bitArray;
```

Table 19. Alternative bitArray definitions

The implementation uses the following algorithm:

```
create std::unordered_set<bitArray> SES_visited;
create std::vector<std::pair<bitArray, bitArray>> initial_SES;
// (∀ pair in initial_SES: 1st element represents a SES, 2nd represents its directly connected edges.)
adds to initial_SES all SES having exactly 1 edge directly connected to each CJoin whose summary
is used in the given GES;
adds those SES to SES_visited;
∀ std::pair<bitArray, bitArray> SES_connections_pair in initial_SES:
    call DFS(SES_connections_pair)
return SES_visited;

// auxiliary function:
// edges_connections is such that edges_connections[i] = bitArray with j^th bit true iff edges i and j
// are directly connected.
DFS(curr_SES, connected_edges):
    create bitArray possible_next_edges = ~ curr_SES & connected_edges;
    ∀ true bit edge (i^th position) in possible_next_edges:
        create bitArray next_SES = curr_SES | edge;
        adds next_SES to SES_visited (if it was not already present)
        if it was not already present:
            DFS(next_SES, connected_edges | edges_connections[i]);
```

Table 20. Algorithm used in SES_set function

The algorithm uses many bitwise operations for enhanced performance.

- "create bitArray *possible_next_edges* = ~ curr_SES & connected_edges;" creates a bitArray possible_next_edges as: a bit in possible_next_edges is true iff it is false in curr_SES and it is true in connected_edges (i.e.: the edge is not present in curr_SES, and is in connected_edges).

- "create bitArray *next_SES* = curr_SES | edge;" creates a bitArray next_SES as: a bit in next_SES is true iff it is true in curr_SES or in edge (i.e.: it is next_SES is equal to curr_SES, except that its bit corresponding to edge is true).

- "DFS(next_SES, connected_edges | edges_connections[i]);" calls DFS having as secondary parameter the union of edges in connected_edges and in edges_connections[i], which is the edges directly connected to the i^{th} edge.

7 Finding the Plan with the Lowest Expected Run Time

In the previous sections, we calculated $\mathcal{O}(S, q, G)$ given a query q on a graph G. The objective of this section is now to present additional theoretical and algorithmic tools to find the plan from $\mathcal{O}(S, q, G)$ with the lowest expecetd runtime. We denote such plan as $\mathcal{O}(S, q, G)_{fastest}$.

Summary-based query optimization for general SQL queries is already an extensively studied topic. One approach, we will call the *Exhaustive Algorithm*, is to simply use those already established results for generic queries on each plan found, so expected runtimes are attributed to each plan, and then select the plan with the lowest expected runtime. It will be $\mathcal{O}(S, q, G)_{fastest}$.

In the next subsections we will present *additional* approaches.

In the next subsections we will consider $\mathcal{O}(S, q, G)_{fastest}$ for a given $\mathcal{O}(S, q, G)$ in which $q \in S_{linear}$. In this case $\mathcal{O}(S, q, G)$ is calculated in section 5.1, and is denoted H.

7.1 Exhaustive Algorithm

The Exhaustive Algorithm approach is to simply use the already established results for generic queries on each plan found in section 5.1, so expected runtimes are attributed to each plan, and then select the plan with the lowest expected runtime. It will be $\mathcal{O}(S, q, G)_{fastest}$. With N the number of predicates in the linear query, this approach has exponential complexity on N. We recall from section 5.1 that H is the set of all the plans of the form:

$\sigma_{?A_0,\ldots,?A_K}\left(\overline{\bowtie}_0\, T'_{P_0}(G)\, \overline{\bowtie}_1 \ldots \overline{\bowtie}_{N-1}\, T'_{P_{N-1}}(G)\, \overline{\bowtie}_N\right)$, in which

$\overline{\bowtie}_0 = \begin{cases} ""\text{(empty string)} \\ \underline{extent(?N_0^S)\ \overline{\bowtie}} \end{cases}$, $\overline{\bowtie}_N = \begin{cases} ""\text{(empty string)} \\ \underline{\overline{\bowtie}\ extent(?N_N^S)} \end{cases}$, and

$\overline{\bowtie}_i = \begin{cases} \overline{\bowtie} \\ \underline{\overline{\bowtie}\ extent(?N_i^S)\ \overline{\bowtie}} \end{cases}\ \forall i \in \{1, \ldots, N-1\}$

in which options using S are noted as underlined, and $?N_i^S$s used are obtained by querying q (or a query whose set of WHERE triples is contained by that of q) on $S(G)$.

And we also recall from 5.1 that $2^{N+1} \leqslant \#\mathcal{O}(S, q, G) \leqslant N * 2^{N+1}$ when either T or T_P is available and $4^{N+1} \leqslant \#\mathcal{O}(S, q, G) \leqslant N * 4^{N+1}$ when both are available.

The Exhaustive Algorithm has at least exponential complexity on N, and is therefore undesirable for queries with large N.

7.2 Exhaustive Algorithm with Pruning, and Heuristic Algorithms

Another approach is to not call the SQL query analyzer on some plans $\in \mathcal{O}(S, q, G)$ if we know beforehand that they are not an optimal plan. We will call this other approach *Exhaustive Algorithm with Pruning*, and we will be constructing it in the next subsections. Solutions found with this method are guaranteed to be an optimal plan.

We will also present *Heuristic Algorithms*. I.e.: algorithms considering only a subset of $\mathcal{O}(S, q, G)$ (usually much smaller than the one considered in the Exhaustive Algorithm with Pruning method), which find a solution probably close to the optimal one.

7.3 Possible Order of Join Operations

We recall that $\overline{\bowtie}_i = \left\{ \underline{\overline{\bowtie}\ extent(?N_i^S)\ \overline{\bowtie}} \right. \forall i \in \{1, \ldots, N-1\}$

Definition 7.1. We note $A \leftarrow$ (resp $A \rightarrow$) as joining table A with the table on its left (resp. right) and returning the result.

Definition 7.2. For a $\overline{\bowtie}_i$, with $i \in \{0, \ldots, N\}$, we define its *immediate join tree* as:

- if a summary is used: the linear tree of successively applying $\leftarrow$ and/or $\rightarrow$ on $extent(?N_i^S)$ until the resulting join tree contains all elements of $\overline{\bowtie}_i$.
  i.e.: both $\overline{\bowtie}$s of $\overline{\bowtie}\ extent(?N_i^S)\ \overline{\bowtie}$ if $i \in \{1, \ldots, N-1\}$, or the only $\overline{\bowtie}$ if i=0 or i=N.
- if no summary is used: the tree containing only $\overline{\bowtie}_i = \overline{\bowtie}$ as join operation, and containing as tables only the two tables being joined by $\overline{\bowtie}_i$.

We present the possible immediate join trees for a $\overline{\bowtie}_i$, with $i \in \{0, \ldots, N\}$, in figures 4 and 5. When $\overline{\bowtie}_i$ does not use the summary, the (only) possibility of the join tree is trivial and is represented in figure 4. When $\overline{\bowtie}_i$ uses the summary, there are only 2 possibilities: (and those are represented in figure 5):

$$extent(?N_i^S) \underbrace{\leftarrow \ldots \leftarrow}_{m} \rightarrow \qquad \text{and} \qquad extent(?N_i^S) \underbrace{\rightarrow \ldots \rightarrow}_{m} \leftarrow$$

For $i \in \{1, \ldots, N-1\}$, we have $m \geq 1$ and both possibilities are valid.
For $i = 0$, we have m = 0 and the only valid possibility is $extent(?N_i^S)\ \rightarrow$.
For $i = N$, we have m = 0 and the only valid possibility is $extent(?N_i^S)\ \leftarrow$.

Figure 4. the only possibility for $\overline{\bowtie}_i$ without summary use

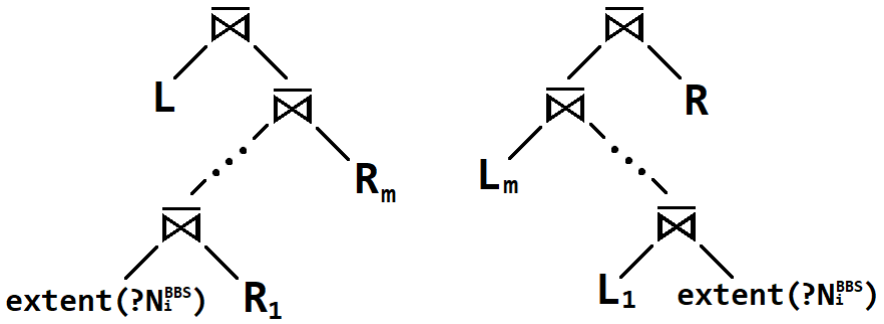

Figure 5. the 2 possibilities for $\overline{\bowtie}_i$ with summary use

7.4 Cost model

We model the total cost of a plan (denoted C_{total}) as the sum of its join operations (denoted C_{joins}), plus the cost of accessing $T'_{P_i} \forall i \in \{0, \ldots, N-1\}$ (denoted C_{tables}), plus the cost of calculating $?N_i^S$s used (denoted $C_{summary}$). i.e.: $C_{total} = C_{joins} + C_{tables} + C_{summary}$.

We model the cost of a join operation $A \bowtie B$ as $3\,(n(A) + n(B))$ in which n(A) (resp. n(B)) = number of pages used to store A (resp. B).

Definition 7.3. We call the *immediate evaluation cost* of a $\overline{\bowtie}_i$ (noted $C_{join}^{\overline{\bowtie}_i}$), with $i \in \{0, \ldots, N\}$, as the cost of evaluating its immediate join tree.

For a given $\overline{\bowtie}_i$ with $i \in \{1, \ldots, N-1\}$, its immediate evaluation cost $\overline{\bowtie}_i$ is 3(n(L) + n(R)) when no summary is used for $\overline{\bowtie}_i$.

When a summary is used for $\overline{\bowtie}_i$, its immediate evaluation cost is the following (or with $L, L_1, \ldots, L_m$ and $R, R_1, \ldots, R_m$ inversed):

$$C_{join}^{\overline{\bowtie}_i} = 3\Big(\underbrace{n(extent(?N_i^S)) + n(R_1)}_{\text{from joining } R_1} + \underbrace{n(extent(?N_i^S) \overline{\bowtie} R_1) + n(R_2)}_{\text{from joining } R_2} + \ldots + \underbrace{n(extent(?N_i^S) \overline{\bowtie} R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_{m-1}) + n(R_m)}_{\text{from joining } R_m} + \underbrace{n(extent(?N_i^S) \overline{\bowtie} R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_m) + n(L)}_{\text{from joining } L}\Big)$$

Reordering the terms, we get that its immediate evaluation cost is the following:

$$C_{join}^{\overline{\bowtie}_i} = 3\Big(\underbrace{n(R_1) + \ldots + n(R_m) + n(L)}_{\text{from original tables}} + \underbrace{n(extent(?N_i^S)) + \ldots + n(extent(?N_i^S) \overline{\bowtie} \ldots \overline{\bowtie} R_m)}_{\text{from tables with } extent(?N_i^S)}\Big)$$

7.5 Substitution estimation for additional summary costs

We now try to estimate the additional cost of using the summary over the case where no summary is used. An estimation for that cost is

$$C_{additional}^{substitution} = \sum_{\overline{\bowtie}_i \; uses \; S} \left(C_{join}^{\overline{\bowtie}_i} - \underline{C_{join}^{\overline{\bowtie}_i}}\right) + C_{summary}$$

where $\underline{C_{join}^{\overline{\bowtie}_i}}$ is the cost of the join tree (which will be noted *substitution join tree* of $\overline{\bowtie}_i$) obtained by substituting $(extent(?N_i^S) \overline{\bowtie} R_1)$ to $R1$ or $(L_1 \overline{\bowtie} extent(?N_i^S))$ to $L1$ in the immediate join tree of $\overline{\bowtie}_i$.

That cost is an *upper bound* of the optimal join cost because the considered join tree (obtained from transforming all immediate join trees into substitution join trees) is one of the possible join trees without using the summaries.

$\forall i \in \{0, \ldots, N\}$, we have (symmetrically for $L, L_1, \ldots, L_N$ and $R, R_1, \ldots, R_N$):

$$\underline{C_{join}^{\overline{\bowtie}_i}} = 3\Big(\underbrace{n(R_1) + n(R_2)}_{\text{from joining } R_2} + \underbrace{n(R_1 \overline{\bowtie} R_2) + n(R_3)}_{\text{from joining } R_3} + \ldots + \underbrace{n(R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_{N-1}) + n(R_N)}_{\text{from joining } R_N} + \underbrace{n(R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_N) + n(L)}_{\text{from joining } L}\Big)$$

Reordering, we get:

$$\underline{C_{join}^{\overline{\bowtie}_i}} = 3\left(\underbrace{n(R_1)+\ldots+n(R_N)+n(L)}_{\text{from original tables}} + \underbrace{n(R_1 \overline{\bowtie} R_2)+\ldots+n(R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_m)}_{\text{from joined tables}}\right)$$

Subtracting that equation from the expression we obtained for $C_{join}^{\overline{\bowtie}_i}$ in the last subsection, we get that, $\forall i \in \{0,\ldots,N\}$, we have:

$$C_{join}^{\overline{\bowtie}_i} - \underline{C_{join}^{\overline{\bowtie}_i}} = 3\Bigg(\underbrace{n(extent(?N_i^S)) + \ldots + n(extent(?N_i^S) \overline{\bowtie} \ldots \overline{\bowtie} R_m)}_{\text{from tables with } extent(?N_i^S)} - \underbrace{n(R_1 \overline{\bowtie} R_2) - \ldots - n(R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_m)}_{\text{from joined tables}}\Bigg)$$

And reordering:

$$C_{join}^{\overline{\bowtie}_i} - \underline{C_{join}^{\overline{\bowtie}_i}} = 3\Bigg(\underbrace{n(e(?N_i^S)) + n(e(?N_i^S) \overline{\bowtie} R_1)}_{\text{fixed tables}} + \underbrace{(n(e(?N_i^S) \overline{\bowtie} R_1 \overline{\bowtie} R_2) - n(R_1 \overline{\bowtie} R_2)) + \ldots + (n(e(?N_i^S) \overline{\bowtie} R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_m) - n(R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_m))}_{\text{tables depending on m}}\Bigg)$$

7.6 Heuristics

We can infer the following from that last equation:

- for every summary used, there is a fixed positive additional cost (equal to $n(extent(?N_i^S))+ n(extent(?N_i^S) \overline{\bowtie} R_1(resp.L_1)$), and a variable cost depending on m, equal to $\sum(n(e(?N_i^S) \overline{\bowtie} R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_i) - n(R_1 \overline{\bowtie} \ldots \overline{\bowtie} R_i))$.
- Each term of the sum in the variable cost is non-positive. If $(n(extent(?N_i^S) \overline{\bowtie} R_1 = n(R_1)$, then the summary failed to reduce the number of considered tuples, and all terms of the variable cost are equal to zero. If $(n(extent(?N_i^S) \overline{\bowtie} R_1 < n(R_1)$, then the summary succeeded in reducing the number of considered tuples, the first term is less than zero, and all the other terms are likely also negative.
- As for every summary used there is a fixed positive additional cost and there is a non-positive variable cost which usually is negative and decreases as m increases and $(n(extent(?N_i^S) \overline{\bowtie} R_1 < n(R_1)$, using extends with high values for m and with small cardinality is the most effective in reducing the cost.

We propose therefore the following heuristics:

1. Use q to calculate $?N_i^S$s (instead of a query whose set of WHERE triples is contained by that of q). This is to decrease $n(extent(?N_i^S) \overline{\bowtie} R_1)$. In a large graph, the added cost of calculating q on the S should be compensated by the negative variable costs.

2. Avoid using extents for all or almost all of $?N_i^S$s with $i \in \{0, \ldots, N\}$. If extents are used for an excessive amount of $?N_i^S$s, then the average m of immediate join trees of $?N_i^S$s will be small, so the average variable costs will be small and they should not compensate the fixed costs.

3. Avoid using extents if their cardinality are big, and prioritize those who are small. Do this by e.g.: selecting the K extents with smallest cardinalities, or every extent with cardinality greater than a threshold.

7.7 Disatvantage of using the summary when N = 2

When N = 1, i.e.: a query of the form:

```
SELECT (a subset of ?N0, ?N1)
WHERE {
      ?N0 A ?N1 .
}
```

the optimal plan is trivially a projection of $T_A^{'}$.

The first interesting case occurs when N = 2, i.e.: a query of the form:

```
SELECT (a subset of ?N0, ?N1, ?N2)
WHERE {
      ?N0 A ?N1 .
      ?N1 B ?N2 .
}
```

Without using summaries, the plan for evaluating the query is $T_A^{'} \bowtie T_B^{'}$, which has a total cost of $3(\mathrm{n}(T_A^{'}) + \mathrm{n}(T_B^{'}))$.

Using a summary, the plan would have at least two joins: a join of $T_A^{'}$ with another table (with cost greater than $3\mathrm{n}(T_A^{'})$), and a join of $T_B^{'}$ with another table (with cost greater than $3\mathrm{n}(T_B^{'})$). The total cost would be greater than $3(\mathrm{n}(T_A^{'}) + \mathrm{n}(T_B^{'}))$. Another way to arrive at that result is to use the cost model described in its subsection above, and limit the value of m.

We reach therefore in an interesting result: assuming the join cost to be linear on the number of pages of the joined tables, for N $\leq$ 2, using a summary is **always** slower than not using it.

8 Conclusion

We have presented theoretical and algorithmic tools to optimize the evaluation of SPARQL queries leveraging summaries, in particular backward bisimulation summaries. And we have implemented a C++ program to enumerate SPARQL plans using BBS, which can be retrieved here: https://github.com/ICEtinger/SPARQL_BBS_plans.

The program implemented can be used to enumerate the possible plans, and the theoretical and algorithmic tools presented can be used to find the plan with the fastest expected evaluation, therefore optimizing the evaluation of the SPARQL query.

References

[ČGM17] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. Query-Oriented Summarization of RDF Graphs. Research Report RR-8920, INRIA Saclay ; Université Rennes 1, June 2017.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[HHK95] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[KBNK02] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, and Henry F Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 133–144, New York, NY, USA, 2002. ACM.

[MW99] Jason McHugh and Jennifer Widom. Query optimization for XML. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 315–326, 1999.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[w3ca] RDF. https://www.w3.org/TR/rdf11-concepts.

[w3cb] RDF Schema. https://www.w3.org/TR/rdf11-mt.