Optimizing SPARQL query evaluation with a worst-case cardinality estimation based on statistics on the data
Louis Jachiet, Pierre Genevès, Nabil Layaida

To cite this version:
Louis Jachiet, Pierre Genevès, Nabil Layaida. Optimizing SPARQL query evaluation with a worst-case cardinality estimation based on statistics on the data. 2017. hal-01524387

HAL Id: hal-01524387
https://hal.archives-ouvertes.fr/hal-01524387
Submitted on 18 May 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Optimizing sparql query evaluation with a worst-case cardinality estimation based on statistics on the data

Louis Jachiet\textsuperscript{1,2}, Pierre Genevès\textsuperscript{2,3}, and Nabil Layaïda\textsuperscript{1,2,3}

\textsuperscript{1} Inria, France
\{louis.jachiet,nabil.layaïda\}@inria.fr
\textsuperscript{2} CNRS, LIG, France
pierre.geneves@cnrs.fr
\textsuperscript{3} Univ. Grenoble Alpes, France

Abstract. SPARQL is the W3C standard query language for querying data expressed in the Resource Description Framework (RDF). There exists a variety of SPARQL evaluation schemes and, in many of them, estimating the cardinality of intermediate results is key for performance, especially when the computation is distributed and the datasets very large. For example it helps in choosing join orders that minimize the size of intermediate subquery results.

In this context, we propose a new cardinality estimation based on statistics about the data. Our cardinality estimation is a worst-case analysis tailored for SPARQL and capable of taking advantage of the implicit schema often present in RDF datasets (e.g. functional dependencies). This implicit schema is captured by statistics therefore our method does not need for the schema to be explicit or perfect (our system performs well even if there are a few “violations” of these implicit dependencies).

We implemented our cardinality estimation and used it to optimize the evaluation of SPARQL queries: equipped with our cardinality estimation, the query evaluator performs better against most queries (sometimes by an order of magnitude) and is only ever slightly slower.

Keywords: RDF System, Distributed SPARQL Evaluation

1 Introduction

SPARQL is the W3C standard query language for querying data expressed in the Resource Description Framework (RDF). There exists a variety of SPARQL evaluation schemes and, in many of them, estimating the cardinality of intermediate results is key for performance, especially when the computation is distributed and the datasets very large. For example it helps in choosing join orders that minimize the size of intermediate subquery results.

In this context, we propose a new cardinality estimation based on statistics about the data. Our cardinality estimation is a worst-case analysis tailored for SPARQL and capable of taking advantage of the implicit schema often present
in RDF datasets (e.g. functional dependencies). This implicit schema is captured by statistics therefore our method does not need for the schema to be explicit or perfect (our system performs well even if there are a few “violations” of these implicit dependencies).

We implemented our cardinality estimation and used it to optimize the evaluation of queries by SPARQLGX which is a top competitor in distributed SPARQL query evaluation [5]. We benchmark SPARQLGX: equipped with our cardinality estimation, the query evaluator SPARQLGX performs better against most queries (sometimes by an order of magnitude) and is only ever slightly slower.

2 SPARQL & BGP

In this section we present a simplified version of the Basic Graph Pattern (BGP) fragment of the W3C standard SPARQL. The RDF defines three different types of values: Literals, Internationalized Resource Identifiers and Blank nodes. Since the core part of the cardinality estimation problem does not depend on the distinction between those different parts we suppose a set \( V \) of valid values and in this view an RDF graph \( G \) is simply a finite subset of \( V^3 \): the element \((s, p, o)\) \(\in G\) describe an edge between the node subject \(s\) and the node object \(o\) labeled by the predicate \(p\).

A Triple Pattern (TP) is composed of three elements: a subject, a predicate and an object. Each of these three elements is either a variable or a value. The set of valid values is \(V\), we denote the set of valid variable names (or Column names) by \(C\). The set of variables of TP is called its domain.

**Definition 1 (mapping)** A “mapping” is a function from \(C\) to \(V\) with a finite domain. Formally a “mapping” corresponds to a set of the form \(\{k_i \rightarrow v_i \mid i \in I\}\) with \(I\) finite and the \((k_i)_{i \in I}\) all distinct. The set \(\{k_i \mid i \in I\}\) is called the domain of the mapping, written \(\text{dom}(\{k_i \rightarrow v_i \mid i \in I\})\).

**Definition 2 (mapping collection)** A “mapping collection” is a set of mappings with the additional information of a domain \(d_1, \ldots, d_k\) shared by all mappings in the collection (i.e. all the mappings in the collection have \(d_1, \ldots, d_k\) as domain).

**Definition 3 (multiset)** A finite multiset \(M = (m_1, \ldots, m_k)\), or bag, is a generalization of a set but where the order of the elements is unimportant but where the same elements can appear several times. A finite multiset over the set \(S\) (i.e. \(\forall i. \ m_i \in S\)) can be represented via its indicator function \(\chi_M : S \rightarrow \mathbb{N}\), for \(v \in S\), \(\chi_M(v) = |\{i \in 1..k \mid m_i = v\}|\).

The solutions of a TP \((s, p, o)\) on a graph \(G\) is the mapping collection whose domain is the domain of \((s, p, o)\) and a mapping \(m\) belongs to this mapping collection if \((s', p', o') \in G\) (where \(x' = x\) when \(x \in V\) and \(x' = m(x)\) otherwise).
Definition 4 (compatible mappings) Two mappings \( m_1 \) and \( m_2 \) are compatible (written \( m_1 \sim m_2 \)) when \( m_1(c) = m_2(c) \) for all \( c \in \text{dom}(m_1) \cap \text{dom}(m_2) \).

Given two compatible mappings \( m_1 \) and \( m_2 \) we define their sum as the mapping whose domain is \( \text{dom}(m_1) \cup \text{dom}(m_2) \) and \( (m_1 + m_2)(c) = m_1(c) \) when \( c \in \text{dom}(m_1) \) and \( m_2(c) \) otherwise.

Definition 5 (Join) Given two mapping collections \( A \) and \( B \) (written \( A \Join B \)) is defined as \( (m_A + m_B \mid m_A \sim m_B \) and \( (m_A, m_B) \in A \times B) \).

A BGP is a list of TPs \((t_1, \ldots, t_n)\), the solution of a BGP \((t_1, \ldots, t_n)\) is the collection mapping corresponding to the join of solutions of individual TP (the join operation is associative and commutative so all join orders lead to the same mapping collection).

3 Summaries

We will now manipulate projections of mapping collections. If \( c \) in the domain of \( M = m_1, \ldots, m_n \) be a mapping collection, the values \((m_1(c), \ldots, m_n(c))\) form a multiset which is the projection of the mapping collection \( M \) on the column \( c \). Our cardinality estimation is based on summaries of projection of mapping collections.

The projections that we manipulate are often very large (as large as the query answer), that is why, in practice, we manipulate multiset summaries. Multiset summaries are a tight representation that over-approximates multisets.

3.1 Definitions

Given a multiset \( M \) represented via its indicator function \( \chi \) over the set \( S \), we will compute a small set \( S' \subset S \) and represent \( M \) in two parts: the elements of \( M \) belonging to \( S' \) and the other elements (belonging in \( S \setminus S' \)). In order to represent the elements of \( M \) in \( S' \), we simply restrict the indicator function \( \chi \) to this \( S' \) and to represent the multiset \( E \) of elements of \( M \) in \( S \setminus S' \) we use three integers: \( T \) the Total number of elements in \( E \), \( D \) the number of Distinct values in \( E \), and \( M \) the maximal multiplicit\( Y \) of an element in \( E \).

Definition 6 (Multiset summary) Formally, a multiset summary corresponds to a quintuple \((S', \chi', T, D, Y)\) where \( S' \subset S \), \( \chi' \) is a function from \( S' \) to \( \mathbb{N} \), and \((T, D, Y) \in \mathbb{N}^3\). \((S', \chi', T, D, Y)\) is a summary of the multiset represented by \( \chi_M : S \to \mathbb{N} \) when:

- \( \chi' \) over-approximates \( \chi \) on \( S' \), i.e. \( \forall v \in S' \) \( \chi(v) \leq \chi'(v) \);
- \( Y \) is an upper bound on \( \chi(v) \) for \( v \notin S' \), i.e. \( \forall v \in S \setminus S' \) \( \chi(v) \leq Y \);
- \( T \) is an upper bound on the number of elements counted with multiplicity of the multiset not in \( S' \), i.e. \( \sum_{v \in S \setminus S'} \chi(v) \leq T \);
- \( D \) is an upper bound on the number of distinct elements of the multiset that is not in \( S' \), i.e. \( |\{v \in S \setminus S' \mid \chi(v) > 0\}| \leq D \).
**Definition 7 (Column summary)** The multiset summary \(<S, \chi, T, D, Y>\) is a column summary for the column \(c\) of the mapping collection \(m_1, \ldots, m_k\) when \(<S, \chi, T, D, Y>\) is a summary for the multiset \(m_1(c), \ldots, m_k(c)\).

**Definition 8 (Collection Summary)** \(N, s\) is a summary for the collection mapping over the domain \(c_1, \ldots, c_k\) if \(N \in \mathbb{N}\) is greater than the number of mappings in the collection and \(s\) is set of pairs \(s = \{(c_1, S_1), \ldots, (c_k, S_k)\}\) where each \(S_i\) is a column summary of the column \(c_i\).

### 3.2 Example of summaries

Let us consider the following dataset:

<table>
<thead>
<tr>
<th></th>
<th>A memberOfTeam 1</th>
<th>A memberOfTeam 2</th>
<th>A memberOfTeam 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>B teamLeader</td>
<td>C teamLeader</td>
<td>E teamLeader</td>
</tr>
<tr>
<td>Number</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

There are two predicates: `memberOfTeam` and `teamLeader`. A possible collection summary for the TP `(?s memberOfTeam ?o)` is \(?s \rightarrow \langle \{A\}, \{A \rightarrow 3\}, 3, 3, 1 >; ?o \rightarrow \langle \{1\}, \{1 \rightarrow 3\}, 3, 2, 2 >\)` and for the TP `(?s teamLeader ?o)` one possible collection summary is \(?s \rightarrow \langle \{B\}, \{B \rightarrow 1\}, 4, 4, 1 >; ?o \rightarrow \langle \{1\}, \{1 \rightarrow 1\}, 4, 4, 1 >\).

With only the information of these summaries we can deduce that, in this dataset, the relation induced by `memberOfTeam`, is such that the subject `A` might appear several times but – except for this `A` – other team members have only one team. In addition, we know that the team 1 has 3 members and the other teams have less than 2 members.

In the summary of the relation induced by `teamLeader` (i.e. the relation between `?s` and `?o` in `(?s teamLeader ?o)`) we see that this relation is bijective: all the subjects `?s` and all the objects `?o` are each present only once.

Therefore if we need to compute the solutions of:

`(?s member memberOfTeam ?team . ?team teamLeader ?leader)`

we know that there are less than 6 solutions: the relation `teamLeader` is bijective therefore the number of solutions of this BGP is less than the number of solutions for:

`(?s member memberOfTeam ?team)`

More generally the summaries allow us to capture the “subject maximum arity” and the “object maximum arity” while first taking out a few outliers. We designed summaries this way because real datasets often have an implicit schema with a low arity for most subject or object that is violated only for a few categories. For instance, DBpedia contains 28373 triples informing of which artist belongs to which band. In this dataset no artist belongs to more than 8 bands however DBpedia has 22 special names for unknown artists (such as `Guitar`) that appear much more than 8 times (e.g. 164 times for `Guitar`).

Another example with DBpedia is movies and directors, each movie often has a few number of directors and each director has generally a relatively limited
number of directed movies, except for a small number of them. On DBpedia, there are 118448 triples with the information of a movie and one of its director. There are only 150 directors with more than 50 directed movies and only 41 movies with more than 10 directors but the most prolific director has 290 directed movies and one movie has 26 directors.

3.3 Simple operations on summaries

We define the following operations on multiset summaries:

- \( \text{count}(cs, v) \): the function that returns the number of times the value \( v \) can appear in a multiset summarized by \( cs \);

\[
\text{count}(< S, \chi, T, D, Y >, v) = \begin{cases} 
\chi(v) & v \in S \\
Y & v \notin S 
\end{cases}
\]

- \( \text{truncate}(cs, n) \): the multiset summary where we enforce that each value appear at most \( n \) times. Formally, \( \text{truncate}(< S, \chi, T, D, Y >) = < S, \min(\chi, n), \min(T, n), \min(D, n), \min(Y, n) > \) (where \( \min(\chi, n)(x) = \min(n, \chi(x)) \));

- \( \text{limit}(cs, n) \): the multiset summary where we enforce that there are at most \( n \) elements. More precisely we have \( \text{limit}(< S, \chi, T, D, Y >) = < S, \min(\chi, n), \min(T, n), \min(D, n), \min(Y, n) > \);

- \( \text{size}(cs) \): the estimated size of the multiset summarized by \( cs \),

\[
\text{size}(< S, \chi, T, D, Y >) = \sum_{x \in S} \chi(x) + T.
\]

\( \text{the sum of two multisets over } S \) represented by \( \chi_1 \) and \( \chi_2 \) is defined as \( (\chi_1 + \chi_2)(x) = \chi_1(x) + \chi_2(x) \). The sum of two multiset summaries is defined as \( < S_1, \chi_1, T_1, D_1, Y_1 > + < S_2, \chi_2, T_2, D_2, Y_2 > = < S_1 \cup S_2, \chi', T_1 + T_2, D_1 + D_2, Y_1 + Y_2 > \) with:

\[
\chi'(x) = \begin{cases} 
\chi_1(x) + \chi_2(x) & \text{when } x \in S_1 \cap S_2 \\
\chi_1(x) + Y_2 & \text{when } x \in S_1 \setminus S_2 \\
Y_1 + \chi_2(x) & \text{when } x \in S_2 \setminus S_1
\end{cases}
\]

We have the property that if a multiset \( m_1 \) is summarized by \( s_1 \) and \( m_2 \) by \( s_2 \) then \( m_1 + m_2 \) is summarized by \( s_1 + s_2 \).

- Given a multiset \( \{m_1, \ldots, m_k\} \) with its summary \( < S, \chi, T, D, Y > \) we can multiply them by an integer \( n \): \( \{m_1, \ldots, m_k\} \times n \) is the multiset containing \( k \times n \) elements: \( n \) elements \( m_i \) for each \( 1 \leq i \leq k \); this multiset is summarized by \( < S, \chi, T, D, Y > \times n = < S, \chi \times n, T \times n, D, Y \times n > \) (with \( (\chi \times n)(x) = \chi(x) \times n \)).

- Given a collection summary \( s = (t, \{(c_1, s_1), \ldots, (c_k, s_k)\}) \) we note: \( \text{summ}(s, c_k) = s_k \) the summary for the column \( c_k \); \( \text{cols}(s) = \{c_1, \ldots, c_k\} \) the set of columns of the summarized collection and \( \text{size}(s) = t \) the size of the summary over-approximated by \( s \).
4 Computing collection summaries representing the solutions of a single TP

4.1 Computing a multiset summary from a multiset

Given a multiset defined by $\chi$ on the set $S$ we compute its multiset summary of size $K$ by sorting $S$ by $\chi$ decreasing, we extract from this the set $S'$ of the $K$ first elements ($S' \subseteq S$, the set of size $K$ with the biggest $\chi$). Then the computed summary is $< S', \chi, T, D, Y >$ where $T = \sum_{x \in S \setminus S'} \chi(x)$, $D = |\{x \in S \setminus S' | \chi(x) > 0\}|$ and $Y = \max_{x \in S \setminus S'}(\chi(x))$.

Choosing $K$ allows to set a balance between precision of summaries and the time needed to compute them. In practice we adopt the same constant $K$ for all summaries. We notice that computing summaries with several thousands of elements performs well in practice. In section 8, we report on practical experiments with $K = 3000$.

4.2 Gathering statistics

In the RDF format, the predicate carries the “semantic” relationship. In most datasets there is usually a limited number of different predicates in the datasets and in queries variable predicates are relatively rare [2].

During the load phase we compute the list of all predicates $P$ (in one pass over the data) and (in a second pass) we compute for each $p \in P$ (in parallel), a collection summary corresponding to the solution of the TP ($?s \ p \ ?o$). To do that, we compute the list $T_p$ of triples that have $p$ as a predicate, we then compute a multiset summary $o_p$ for the object of $T_p$ and a multiset summary $s_p$ for the subjects of $T_p$, the collection summary is $|T_p|, \{(?s, s_p), (?o, o_p)\}$.

Summaries are computed recursively: we start by computing summaries for individual TP and then combine them. Let $(t_s \ t_p \ t_o)$ be a TP, let us show how to compute its associated summary.

4.3 Fixed predicate $t_p = p$

Let us consider first, the cases where the predicate is fixed to a value $p$, depending on whether the subject is fixed (either the variable $?s$ or the value $s$) and whether the object is fixed (either $?o$ or $o$) we have four cases (the case $(?s, ?o)$ is symmetrical to $(?s, o)$ and thus not treated):

Case $t_s = ?s$ and $t_o = ?o$: the returned summary is simply: $size(s_p), \{(?s, s_p), (?o, o_p)\}$.

Case $t_s = s$ and $t_o = o$: this TP has either 0 or 1 solution and binds 0 columns. The returned summary is $(0, \emptyset)$ when $count(s_p, s) = 0$ or $count(o_p, o) = 0$ and $(1, \emptyset)$ otherwise.

Case $t_s = ?s$ and $t_o = o$: this TP has, at most, $count(o_p, o)$ solutions and only binds the column $?s$. The returned summary is $count(o_p, o), \{(?s, truncate(s_p, 1))\}$. 

4.4 Variable predicate $t_p = ?p$

Let us note $r$, the collection summary for the solutions of $(t_s \ t_p \ t_o)$, the idea is to combine the summaries $s_i$ (summary for $(t_s \ p_i \ t_o)$) for each $p_i \in P$. We build $r$ such that $\text{size}(r) = \sum_{p_i \in P} \text{size}(s_i)$ and for each eventual bounded column $c$ ($c \in \{?s, ?o\}$), then $\text{summ}(r, c) = \sum_{p_i \in P} \text{summ}(s_i, c)$ and, finally, the summary for the column $?p$, is $< P, \chi_p, 0, 0, 0 >$ where $\chi_p(p_i) = \text{size}(s_i)$.

4.5 Duplicated variable

It is possible in SPARQL to have a variable that is present twice (or thrice). Since there are only three parts to If $t_p$ is the duplicated variable, we apply the replacement scheme proposed in 4.4 but we replace all the duplicated part (and not only $t_p$).

If the predicate is not duplicated but we have a duplicated variable then the triple is of the form $(?s \ p \ ?s)$ (if the predicate is variable we first apply the replacement scheme).

Let $T_p, \{(?s, s_p), (?o, o_p)\}$ be the collection summary pre-computed for the TP $(?s \ p \ ?o)$ with $s_p =< S_s, \chi_s, T_s, D_s, Y_s >$. Then we compute the possible number of different values in the intersection of the multiset represented by $s_p$ and $o_p$: $n = \text{min}(n_s, n_o)$ where $n_s = \text{min}(D_s, |\{x \in S_o \setminus S_s \mid \chi_s(x) > 0\}|)$ and $n_o = \text{min}(D_o, |\{x \in S_s \setminus S_o \mid \chi_o(x) > 0\}|)$ the collection summary is $n, \{?s, \chi_s, ?o, \chi_o, 1, \text{min}(n_o, n_s), \text{min}(n_o, n_s), 1 \}$

5 The multiplicative factor

Given two collection summaries $s_A$ (resp. $s_B$) summarizing two collections mappings $A = \{A_1, \ldots, A_n\}$ (resp. $B = \{B_1, \ldots, B_m\}$) we want to compute a collection summary for the solutions of $A \times B$. In order for this summary to be precise, we will introduce in this section the “multiplicative factor” and in the next section we will show how to use the multiplicative factor to compute a relatively precise summary for $A \times B$.

The collection mapping $A \times B$ can be seen as a cartesian product $A \times B$ where we removed mappings $(m_A, m_B)$ that do not agree on all the common columns of $A$ and $B$. Each mapping $m$ is thus built using a unique pair $(m_A, m_B) \in A \times B$ (but such a pair does not necessarily correspond to a mapping of $A \times B$). If we track which mappings of $A$ and $B$ were used to build which mappings of $A \times B$, we can count for each mapping $m_A \in A$ its “multiplicative factor”: the number of mappings in $A \times B$ that were built using $m_A$. The i-th multiplicative factor of $A$ toward $B$ is defined as the i-th greatest multiplicative factor of an element of $A$ (or 0 if $|A| < i$).

As summaries work with over-approximation we will show in this section how to compute, with only the summaries for $A$ and $B$, a bound for the $i$-th multiplicative factor $\text{mult}(s_A, s_B, i)$, i.e. an over-approximation to $\text{mult}(A, B, i)$ for any $A$ and $B$ such that $A$ (resp. $B$) is summarized by $s_A$ (resp. $s_B$).
5.1 A common column

Let c be a column shared between A and B (note that such c does not necessarily exist). A mapping \( m_A \in A \) can only be combined with mappings \( m_B \in B \) to form mappings of \( A \Join B \) when \( m_A(c) = m_B(c) \). Therefore, a mapping \( m_A \in A \) can only be used to build, at most, \( \text{count}(\text{summ}(s_B, c), m_A(c)) \) mappings of \( A \Join B \).

Let \( cs_A =< S_A, \chi_A, T_A, D_A, Y_A > \) (resp. \( cs_B =< S_B, \chi_B, T_B, D_B, Y_B > \)) be the column summary for the column c in \( s_A \) (resp. in \( s_B \)) then for each \( v \in S_A \cup S_B \) there are, at most, \( \text{count}(s_A, v) \) mappings of \( A \) and each can be combined with \( \text{count}(cs_B, v) \) mappings of \( B \). To that we need to add that the, at most, \( T_A \) mappings \( m \in A \) with \( m(c) \notin S_A \cup S_B \) can each be joined with, at most, \( Y_B \) elements.

In total this gives us \( T_A + \sum_{v \in S_A \cup S_B} \text{count}(cs_A, v) \) values that over-approximate the various multiplicative factors of elements of \( A \). By sorting these values, the \( i \)-th greatest value gives us a bound on \( \text{mult}(s_A, s_B, i) \).

Note that the mappings with values for the columns c falling into \( S_B \setminus S_A \) might be counted twice: each of the \( T_A \) elements produce \( Y_B \) mappings of \( A \Join B \), but for each \( v \in S_B \setminus S_A \) we also counted that \( Y_A \) mappings produced \( \chi_B(x) \) each. However, this is not an actual issue since we combine the several bounds on \( \text{mult}(s_A, s_B, i) \), in particular, \( i > \text{size}(s_A) \) implies \( \text{mult}(s_A, s_B, i) = 0 \) and \( \text{size}(s_A) \leq T_A + \sum_{v \in S_A} \text{count}(cs_A, v) \).

5.2 General case

There are no more than \( \text{size}(s_A) \) mappings in \( A \) and each can be used in at most \( \text{size}(s_B) \) mappings of \( A \Join B \), we have a first bound:

\[
\text{mult}(s_A, s_B, i) \leq \begin{cases} 
\text{size}(s_B) & \text{when } i \leq \text{size}(s_A) \\
0 & \text{otherwise}
\end{cases}
\]

Then, we simply use the technique presented earlier on each column shared between the domains of \( A \) and \( B \). Each column giving us a new bound on \( \text{mult}(s_A, s_B, i) \) and we combine all of them: if \( \text{mult}(s_A, s_B, i) \leq k_1 \) and \( \text{mult}(s_A, s_B, i) \leq k_2 \) then \( \text{mult}(s_A, s_B, i) \leq \min(k_1, k_2) \).

Finally we compute a bound \( \text{total}(s_A, s_B, n, k) \) on the number of mappings of \( A \Join B \) that can be built using \( n \) different mappings of \( A \) when we know that each of those mappings has a multiplicative factor bounded by \( k \in \mathbb{N} \cup \{ \infty \} \):

\[
\text{total}(s_A, s_B, n, k) = \sum_{i \leq n} \min(k, \text{mult}(s_A, s_B, i))
\]

6 Joining summaries

Given two collection summaries \( s_A \) (resp. \( s_B \)) summarizing two collections mappings \( A = \{A_1, \ldots, A_n\} \) (resp. \( B = \{B_1, \ldots, B_m\} \)) we want to compute a collection summary for the solutions of \( A \Join B \).
We suppose that we computed a bound for \(\text{total}(s_A, s_B, n, k)\). The total number of mappings of \(A \bowtie B\) can be bounded by \(\min(\text{total}(s_A, s_B, \text{size}(A), \text{size}(B)), \text{total}(s_B, s_A, \text{size}(B), \text{size}(A)))\).

### 6.1 Combining summaries for non common columns

Let \(c\) be a column in the domain of \(A\) but not in the domain of \(B\), let \(cs_A = <S_A, \chi_A, T_A, D_A, Y_A>\) be the summary for the column \(c\) of \(A\), the summary we compute for the column \(c\) of \(A \bowtie B\) is \(<S_a, \chi', T', D_A, Y'>\) where:

- \(T' = \text{total}(s_A, s_B, T_A, \infty)\)
- \(Y' = \text{total}(s_A, s_B, Y_A, \infty)\)
- \(\chi' = \text{total}(s_A, s_B, \chi(x), \infty)\)

Columns that are in the domain of \(B\) but not in the domain of \(A\) are treated symmetrically.

### 6.2 Combining summaries for a common column

Let \(c\) be a column in the domain of both \(A\) and \(B\), let \(cs_A = <S_A, \chi_A, T_A, D_A, Y_A>\) (resp. \(cs_B = <S_B, \chi_B, T_B, D_B, Y_B>\)) be the summary for the column \(c\) of \(A\) (resp. of \(B\)). Each mapping \(m\) of \(A\) will be used in at most \(\text{count}(cs_A, m(c))\) mappings of \(A \bowtie B\), therefore the \(\text{count}(cs_A, m(c))\) mappings sharing the value \(m(c)\) will be used in, at most, \(\text{total}(s_A, s_B, \text{count}(cs_A, m(c)), \text{count}(cs_B, m(c)))\) mappings of \(A \bowtie B\). Note that by symmetry between \(A\) and \(B\) we also have that it will be used in, at most, \(\text{total}(s_B, s_A, \text{count}(cs_B, m(c)), \text{count}(cs_A, m(c)))\) mappings of \(A \bowtie B\).

Our summary for the column \(c\) of \(A \bowtie B\) is \(<S_a \cup S_b, \chi', T', \min(D_A, D_B), Y'>\) where:

- \(T' = \min(\text{total}(s_A, s_B, T_A, Y_B), \text{total}(s_B, s_A, T_B, Y_A))\)
- \(Y' = \min(\text{total}(s_A, s_B, Y_A, Y_B), \text{total}(s_B, s_A, Y_B, Y_A))\)
- \(\chi'(x) = \min(\text{total}(s_A, s_B, \text{count}(cs_A, x), \text{count}(cs_B, x)), \text{total}(s_B, s_A, \text{count}(cs_B, x), \text{count}(cs_A, x)))\)

### 7 Optimization of distributed BGP query plans with summaries

In this section, we present how to use a cardinality estimation to optimize the query plan of SPARQL-GX [5] a distributed SPARQL query evaluator.

#### 7.1 Query plan

We suppose that the evaluator has access to the four following primitives:

- \(TP(t)\) take a TP \(t\) and returns the mapping collection solution of \(t\);
– **HashJoin**(a, b) take two terms a and b and return the join of the mapping collections return by a and b;
– **Distribute**(v, a, b) take two terms a and b, stores the mapping collection returned by a into v and then evaluates b;
– **LocalJoin**(a, v) returns the join of mapping collection returned by a and the mapping collection stored into v.

The idea behind the **Distribute**(v, a, b) primitive is to compute once the solution for a store it into the variable v that is distributed to all computing nodes. This way, during the computation of b, if we come across a **LocalJoin**(c, v) then each computing node holds the whole mapping collection v and the join can be done locally.

### 7.2 Query plan cost

We now present how to compute the cost of a query plan. We note sol(a) the mapping collection return by evaluating the query plan a and size(a) its size. We do not have access to the actual size of the mapping collection solution of a query plan but all query plans corresponds to BGP and we know how to estimate their size: given a BGP, we compute a collection summary s and its estimated cardinality is just size(s). Since our cardinality estimation is a worst case, our query plan cost estimation also constitutes a worst-case analysis.

Our query plan cost analysis is conditioned by three constants:

– **shuffleCost** we suppose that the cost of shuffling a mapping collection has a cost linear in its size with a coefficient **shuffleCost**.
– **distributeCost** we also suppose that the cost of distributing a mapping collection is linear with a coefficient **distributeCost**, however the **Distribute** operation breaks when the mapping collection does not fit into RAM, that is why we have:
– **broadcastThreshold** that indicates the maximum size of a mapping collection that we can distribute.

In our translation, the individual TP are all translated exactly once, we set their cost to 0.

For a **HashJoin**(a, b) we need to compute a and b then shuffle a (resp. b) so that they are hashed on dom(sol(a))\(\cap\)dom(sol(b)), this costs **shuffleCost**\(\times\)size(a) (resp. **shuffleCost**\(\times\)size(b)) but only if a (resp. b) is not already correctly shuffled. We also need to materialize **HashJoin**(a, b, ) (even if both component are already shuffled) which costs size(HashJoin(a, b)).

For a **Distribute**(v, a, b) we need to compute a, distribute it to all workers and then compute b, therefore the cost of **Distribute**(v, a, b) is **distributeCost**\(\times\)size(a) + size(b) (plus the cost of compute a and b). However **Distribute**(v, a, b) can break if a is not small enough to fit into RAM, that is why we impose size(a) < **broadcastThreshold**.

For a **LocalJoin**(a, v) we need to compute a and join it with v (that is already computed) it costs size(LocalJoin(a, v)) (plus the cost of computing a).
7.3 Finding the best query plan

Given a BGP \((t_1, \ldots, t_n)\) the naive translation is to simply join \(TP(t_i)\) using at each step a hash-join algorithm: \(HashJoin(TP(t_1), HashJoin(\ldots, TP(t_n)) \ldots)\).

The naive translation is often not the most efficient plan: we might have to materialize large intermediate results (that could be avoided by using a different join order) and it might also yields shuffles that could be avoided. For instance, if \(t_1\) and \(t_4\) bind \(?a\), \(t_2\) binds \(?a\) and \(?b\), \(t_3\) binds \(?b\) and \(?c\), then the naive translation shuffles \(t_1\) and \(t_2\) on \(?a\) then it shuffles \(t_1 \bowtie t_2\) and \(t_3\) on \(?b\) and finally it shuffles \(t_1 \bowtie t_2 \bowtie t_3\) and \(t_4\) on \(?a\) whilst the order \(((t_1 \bowtie t_4) \bowtie t_2) \bowtie t_3)\) implies one less (potentially costly) shuffle.

Furthermore, the HashJoin algorithm is not always the most appropriate join algorithm [3]: when one mapping collection is large enough compared to the other then a broadcast-join (i.e. Distribute the small dataset to all workers then do a LocalJoin) avoids a costly shuffle of the large dataset to the price of sending a small mapping collection to all workers.

Finally, when a triple \(t_i\) binds only one variable \(v\), has a small number of solutions and at least one of the \(t_j\), with \(j \neq i\), contains \(v\) in its domain, it might be more efficient to broadcast-filter \(t_i\). Broadcast-filtering \(t_i\) consists in computing \(Distribute(v_i, TP(t_i), P)\) where \(P\) is a term computing \(t'_1 \bowtie \ldots \bowtie t'_{i-1} \bowtie t'_{i+1} \ldots t'_n\) and \(TP(t'_j) = LocalJoin(TP(t_j), v_i)\) if \(v\) is in the domain of \(t_j\) and \(TP(t'_j) = TP(t_j)\) otherwise.

To find the query plan minimizing our cost estimation, we essentially enumerate possible plans: that filter-broadcast or not triples, with all possible join orders and for each join we consider the hash-join and the broadcast-joins. Our algorithm thus enumerate an exponential number of plans. However, with a few simple heuristics to cut down the number of plans and the heavy use of memoization, performs well in practice as we now illustrate.

8 Experimental results

8.1 Prototype

We implemented the aforementioned algorithms in the SPARQLGX query evaluator and benchmark the impact our cardinality estimation on the performance. The source code of our prototype is available online at the address: https://github.com/tyrex-team/sparqlgx/.

Our Cardinality Estimator “summaries” was compared to the “stats” version of SPARQLGX [5] that re-organizes TP using statistics about the dataset collected at load time plus general heuristic to obtain a fast query execution plan. We also compared “summaries” to the “NoCartesian” that simply avoid cartesian products.
8.2 Setup

Our prototype uses Spark version 2.1.0 in a cluster of two computing nodes running debian each equipped with 20 GB of RAM and 24 cores of computation. The dataset was stored using Hadoop 2.7.3.
8.3 Datasets and queries

SPARQLGX was tested against the benchmark Lubm [6] 10k that contains 1.38 billions triples and weight 232 GB uncompressed while the WatDiv [1] 10k contains 1.10 billions triples and weight 149GB.

In Lubm, most queries (Q4 to Q13) compute an empty answer unless the dataset is extended with reasoning. We did not extend the dataset with reasoning but we left those queries as it is an interesting use-case of our method to detect empty answers (which it fails to do in Q6 and Q11).

The times spent in evaluating each query of this benchmark are shown in figure 1. These times do not include the translation time that took less than one second for all queries. Notice that the time axis of the figure 1 is logarithmic.

8.4 Experimentation

Between each query the spark cluster was stopped and relaunched. However the OS did not restart between queries and stores a cache of recently read files in RAM. We ran all queries three times and interleaved the different methods so that all methods benefit equally from this cache. We took the best of the three for each method (other metrics would give an advantage to the last tested method). All experiments were stopped after 20 minutes of computation.

The statistics we collected with a double pass on the data during the load phase and the size of the collected statistic weight 2.6MB for WatDiv and 2.8MB for Lubm.

8.5 Results

On all almost all queries, the SPARQLGX query evaluator equipped with our “summaries” goes faster than both the NoCartesian and the “stats” approaches. The exception are the complex WatDiv queries where the “stats” approach beats our “summaries” approach: C1 (40s versus 30s), C2 (49s versus 46s) and C3 (292s versus 263s). But even in those cases our algorithm is slower by only a reasonable margin.

On the opposite there are several queries with a non-empty result where our method vastly outperforms the “stats” module, on Lubmn10k: on Q2 (175s vs 1680s) and on WatDiv10: the star shaped queries (e.g. 13s versus 56s for S1), for L3 (6.5s versus 13.2s), for F5 (19s versus 35s), etc.
<table>
<thead>
<tr>
<th>Lubm Query</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
<th>Q13</th>
<th>Q14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summaries</td>
<td>29.96</td>
<td>23.99</td>
<td>6.72</td>
<td>0.67</td>
<td>14.43</td>
<td>0.72</td>
<td>6.69</td>
<td>0.69</td>
<td>11.72</td>
<td>15.37</td>
<td>0.64</td>
<td>0.698</td>
<td>26.82</td>
<td></td>
</tr>
<tr>
<td>Stats</td>
<td>45.96</td>
<td>1200</td>
<td>60.15</td>
<td>474.78</td>
<td>20.16</td>
<td>14.37</td>
<td>181.1</td>
<td>140.2</td>
<td>304.5</td>
<td>30.67</td>
<td>20.89</td>
<td>31.67</td>
<td>14.88</td>
<td>20.77</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WatDiv Query</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>WatDiv Query</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
</tr>
</thead>
</table>

Fig. 2: Query time results for SPARQL-GX against Lubm10k and WatDiv 10k with the “stats” and “summaries” optimizers

9 Related work

Estimating the number of solutions for a query has long been viewed as a key element in the optimization of queries and it is a well-studied problem in the relational world [13]. Various techniques successful in the relational world (e.g. histograms [8,12]) have been less successful for the semantic web [4,10]. The main reasons for that is the heterogeneous and string nature of RDF [10] and the fact that SPARQL queries usually contain a lot of self-joins that are notoriously hard to optimize [14].

Various works have tackled the specific issue of cardinality estimation for SPARQL. A line of work [15] introduced the “selectivity estimation” now in use in several SPARQL evaluators [16] (and close to what the “stats” module does for SPARQL-GX to which we compare “summaries”). The “selectivity estimation” assumes the statistical independence of the various parts of a TP. Variants of this method have been implemented in popular SPARQL query evaluators (e.g. in RDF-3X [11]).

A second line of work [9] takes as input an actual schema and produces an optimized query plan based on information extracted from the schema.

A third line of work [10] tries to derive the implicit schema of an RDF graph by fitting nodes into characteristic sets, or by summarizing [7] the graph into large entities. These approaches are the closest to our approach in spirit but they tend to focus on finding an implicit schema type for nodes while our approach is more focused on finding an implicit schema for edges.

The “summaries” approach presented in this paper can also be seen as a complement of existing methods. Indeed, we provide a worst-case analysis and a worst-case analysis can always be easily combined with any other analysis.

10 Conclusion

We introduced a new concept: collection summaries. We showed how to compute collection summaries for Basic Graph Patterns and how they can be used
to estimate the cardinality of query answers. Experimental results show that this cardinality estimation allows to improve the performance of SPARQL query evaluation.

References
