Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins

Thomas Neumann [#1], Guido Moerkotte [*2]

[#] *Technische Universität München*
Munich, Germany
[1] neumann@in.tum.de

[*] *Universität Mannheim*
Mannheim, Germany
[2] moerkotte@informatik.uni-mannheim.de

***Abstract*—Accurate cardinality estimates are essential for a successful query optimization. This is not only true for relational DBMSs but also for RDF stores. An RDF database consists of a set of triples and, hence, can be seen as a relational database with a single table with three attributes. This makes RDF rather special in that queries typically contain many self joins.**

We show that relational DBMSs are not well-prepared to perform cardinality estimation in this context. Further, there are hardly any special cardinality estimation methods for RDF databases. To overcome this lack of appropriate cardinality estimation methods, we introduce *characteristic sets* together with new cardinality estimation methods based upon them. We then show experimentally that the new methods are–in the RDF context–highly superior to the estimation methods employed by commercial DBMSs and by the open-source RDF store RDF-3X.

I. Introduction

The semantic web, its underlying data model RDF and its query language SPARQL have received much attention during the last years. In essence, RDF represents data as a set of triples. Hence, an RDF base can be seen as a relational database containing a single relation that has only three attributes. Even the names of these attributes are fixed. They are *subject*, *predicate*, *object*. Let us take a look at a SPARQL query:

```
select ?s ?t ?y
where { ?s <hasTitle> ?t.
    ?s <hasAuthor> "Jane Austen".
    ?s <inYear> ?y }
```

This query asks for information about books written by Jane Austen. Variables start with a question mark, the other terms are constants. Usually, ignoring the `select`, queries are sketched using *triple patterns*, which resembles a datalog query's body without predicates. This is possible since there is only one predicate for RDF triples. For the above query, the triple pattern is

(?s,hasTitle,?t), (?s,hasAuthor,Jane Austen),
(?s,inYear,?y)

Clearly, two self joins are necessary to evaluate this query.

Currently, specialized RDF stores are developed by commercial companies and academics. As a traditional relational database, any RDF store needs a query optimizer that produces an efficient execution plan for a given SPARQL query. The statement that accurate cardinality estimates are vital for a successful query optimization also remains valid for RDF/SPARQL. To see this, consider the following query, which asks for people having books in English and Dutch:

(?p,hasBook,?b1), (?p,hasBook,?b2),
(?b1,hasLanguage,english), (?b2,hasLanguage,dutch).

Clearly, the wrong join order is terrible for a realistic number of occurences.

Since the information on an entity (subject) is distributed on several triples with different predicates (and objects) but the same subject, *star queries*, as in the first example, are quite common. Using triple patterns, they are characterized by several triples with a common subject variable. The reader should not confuse SPARQL star queries with star queries in the relational context. There, a star query is one where the query graph has a star shape, that is there is a single relation, the center, from which join predicates to all other relations exist, and no other join predicate is present. Indeed, star queries in SPARQL exhibit a query graph that is a clique. This is illustrated in Fig. 3.

In this paper, we propose a highly accurate cardinality estimation method for RDF star joins based on a novel synopsis called *characteristic sets*. We also show how this estimation method helps in estimating the cardinality of general queries, i.e., not only RDF star join queries, and how it can be integrated into a query optimizer for SPARQL. We experimentally validate the accuracy and compare it with the estimation accuracy of three major commercial DBMSs and the open-source RDF store RDF-3X.

The rest of the paper is organized as follows. Sec. II discusses related work. Sec. III provides insights into the unique challenges of cardinality estimation for RDF. Sec. V introduces characteristic sets and the basic estimation procedure exploiting characteristic sets for RDF star queries. Sec. VI shows how this basic estimation procedure can be integrated with other methods to cover all RDF queries and how the resulting general method can be integrated into a plan generator. Sec. VII evalutes the accuracy of our approach and others. It also provides some numbers indicating that the achieved increase in accuracy really pays off in terms of query execution times. That is, better estimates lead to better plans. Sec. VIII concludes the paper.

II. Related Work

While cardinality estimation in general is a well-established field [1], work in the context of RDF data hardly exists. Some of the more general-purpose estimation techniques, like multi-dimensional histograms or join histograms [1], would seem useful but are hard to apply due to the heterogeneous and string-oriented nature of RDF.

The RDF-3X system uses specialized histograms for RDF data [2], which are much better suited for RDF than standard synopses, but still have issues in the presence of correlated predicates (see Section III). It also computes the most frequent paths in the RDF graph, which captures some kind of correlations, but this computation suffers from combinatorial explosion and, thus, captures only a few cases. Most of the other systems either ignore cardinality estimation or use very simplistic models. The Jena ARQ optimizer [3] uses single attribute synopsis for individual patterns and pre-computes the number of co-occurrences of all pairs of predicates to estimate join selectivities. However the formulas from [3] basically assume independence between tuple attributes, which quickly leads to severe underestimations. In principle the co-occurrence information between predicates allows for capturing some kinds of correlation between predicates, but the formulas from [3] do not make proper use of this information. Using conditional selectivities would improve the accuracy of the model, but we found in experiments that even then the lack of correlation between predicate and object leads to severe underestimations.

A more refined technique based upon graph summarization was proposed in [4]. It computes the frequency of subgraphs up to a given size and stores it as synopsis. If the synopsis is too large, it prunes path length such that the cut part is distributed as uniformly as possible (i.e., the error resulting from an independence assumption is minimized). When estimating cardinalities, it uses the known fragments and combines them assuming conditional independence. While this is a quite plausible idea, we found that does not work very well in practice. We include more detailed results in Sec. VII, but the main problem is combinatorial explosion. For real world data sets, the number of subgraphs is simply too large, even for very modest subgraph sizes. As the predicates are distributed in a quite chaotic matter in real-world data sets, subgraph counts of 10^9 and more are no exception. The pruning techniques from [4] can shrink this to arbitrary sizes, but the estimation accuracy suffers greatly. Another problem is that they offer no meaningful way to handle bound object values. In theory, their approach allows for adding node labels, which could capture literal semantics, but they do not propose a way to come up with labels. Picking labels is quite crucial, as too fine grained labels would make the combinatorial explosion even worse, while too course labels hurt estimation accuracy. We picked the semantic class (i.e., rdf:type annotation) as label, which matches the examples [4], but this requires schema information which is not available in general. In our experiments we used additional synopsis over object distributions to get reasonable estimates for bound objects. While mining frequent subgraphs work well for small, clean RDF graphs, we had difficulties getting reasonable accurate estimations for real-world data using the approach from [4].

III. The Challenge

Estimating the cardinality of queries over RDF data is very challenging due to the nature of RDF data: First, the nearly complete lack of explicit schema is a serious problem, as all data more or less "looks the same" (i.e., there is no natural way to partition the data for estimation purposes). Second, correlated predicates are not an exception but the rule. This can already be seen for simple queries consisting of a single triple pattern: Consider as illustrational example the triple pattern ($?a$,<isCitizenOf>,<United_States>). On the Yago data set [5] (that contains RDF data derived from the English Wikipedia), we observe the following selectivities:

$sel(\sigma_{P=\text{isCitizenOf}})$	$1.06 * 10^{-4}$
$sel(\sigma_{O=\text{United\_States}})$	$6.41 * 10^{-4}$
$sel(\sigma_{P=\text{isCitizenOf} \wedge O=\text{United\_States}})$	$4.86 * 10^{-5}$
$sel(\sigma_{P=\text{isCitizenOf}}) * sel(\sigma_{\text{O=United\_States}})$	$6.80 * 10^{-8}$

Both the predicate P = isCitizenOf and O = United_States are quite selective in themselves, but we see that the combined predicate is far less selective than predicted by multiplying both selectivities. This is caused by the fact that both predicates are correlated because nearly half of the triples with P = isCitizenOf are about US citizens. Note that this correlation is induced not only by the data source itself (which is somewhat US-centric), but by the very nature of the RDF graph: A triple with an <isCitizenOf> predicate nearly always contains a country as object, which implies a correlation between predicate and object. As RDF graphs are used to model semantics, this kind of correlations is extremely common.

For this reason, standard single-bucket histograms are nearly useless for RDF queries. Virtually all queries that involve more than one constant exhibit correlations, and using single-bucket histograms with independence assumption would lead to severe misestimations. Another severe reason why histograms are inappropriate for RDF data is that histograms are typically applied to numeric data, but RDF data mainly consists of strings.

Thus, we introduce a novel technique for an accurate cardinality estimation for RDF queries. Note that basic estimation techniques for dependent joins are already implemented in the RDF-3X system [2]. The next section briefly describes them. All other techniques are new contributions.

IV. Card. Estimation in RDF-3X

The original work on the RDF-3X system already noticed these issues and tried to address them in the system [2]. For individual triple patterns, the solution is surprisingly simple: For performance reasons, RDF-3X stores not just the complete triples but also the projections to one respectively two attributes, together with the number of occurrences. This information can be used directly for cardinality estimation, one point query into the index structures returns the cardinality of a specific triple pattern.

This solves selectivity estimation of simple patterns, but not for join queries. And SPARQL queries are very join-intensive, 10 patterns and more are not unusual. Again, RDF triples are frequently very correlated. For example, the two patterns

$$(?b, \text{<author>}, ?a), (?b, \text{<title>}, ?t)$$

are strongly correlated; most objects that have an author will have a title. But capturing correlations between joined triple patterns is hard, simply because the number of combinations is so huge. RDF-3X sidesteps this problem by pulling up constants during selectivity estimation [2]:

$$\begin{array}{rcl} \sigma_{p_1}(T) \bowtie_{p_2} \sigma_{p_3}(T) & \equiv & \sigma_{p_3}(\sigma_{p_1}(T) \bowtie_{p_2} T) \\ \Rightarrow \; sel(\sigma_{p_1}(T) \bowtie_{p_2} \sigma_{p_3}(T)) & = & sel(\sigma_{p_1}(T) \bowtie_{p_2} T) * \\ & & sel(\sigma_{p_3}(...)) \end{array}$$

That is, a join between two triple patterns with selections is interpreted as a join between one pattern with a selection and one without a selection. The missing selection is then taken care of after fixing the selectivity of this join. If we assume independence, this is sufficient. As there are only nine possible join predicates p_2 in SPARQL, RDF-3X simple precomputes all predicate/join combinations occurring in the data and stores them in a compressed index structure.

This gives much better estimates than always assuming independence, but it does not solve the fundamental problem. For the two patterns shown above, for example, RDF-3X would not (fully) realize that `author` implies a `title`. More formally, the RDF-3X approach captures the correlation between p_1 and p_2 (i.e., the dependent selectivity of p_2), but then assumes independence for p_3. For just one join this seems reasonable, but the error caused by this approach becomes very noticeable for queries with multiple selection predicates (see Section VII).

V. Characteristic Sets

A. Plain Characteristic Sets

Standard histograms do not capture correlations between join predicates at all, and even using dependent selectivities, as shown in the previous section, only helps for the first join. Therefore, we propose a radically different approach for estimating the selectivity of joins in RDF graphs.

Many of the correlations we observe during selectivity estimation stem from the fact that RDF uses multiple triples to describe the same object. Consider the following sample triples describing a book:

(o_1,title,The Tree and I), (o_1,author,R. Pecker),
(o_1,author,D. Owl), (o_1,year,1996).

All four triples describe the same entity, and accordingly, the individual triple patterns (derived by replacing o_1 with a variable ?b) are strongly correlated. The year is somewhat an exception here, but for the other three triples, searching just for one triple pattern is nearly as selective as searching for all of them. Obviously, this is true for most books. In general, many entities can be uniquely identified by a true subset of their emitting edges.

In most RDF data sets, these emitting edges exhibit a certain structure. While RDF is used usually without a fixed schema, some kind of latent soft schema in the data frequently occurs: Books tend to have `authors` and `titles`, etc. While we might not be able to clearly classify an entity as "book" (due to the lack of schema information), we observe that we can *characterize* an entity by its emitting edges. For each entity s occurring in an RDF data set R, we define its *characteristic set* as follows:

$$S_C(s) := \{p | \exists o : (s, p, o) \in R\}.$$

For many RDF data sets, entities that have the same characteristic set tend to be semantically similar. This is not surprising, as RDF encodes all semantics using edges. But this enables us to predict selectivities based upon the involved characteristic sets.

For a given RDF data set, we define the set of characteristic sets as

$$\mathcal{S}_C(R) := \{S_C(s) | \exists p, o : (s, p, o) \in R\}.$$

For real-world data sets, $\mathcal{S}_C$ is surprisingly small. In theory $|\mathcal{S}_C(R)|$ can be as large as $|R|$, but in practice it is much smaller. We include a more thorough investigation of this in Section VII, but, for example, the UniProt data set with protein information [6] contains only 615 distinct characteristic sets, even though the number of unique triples is 845,074,885.

It is interesting to note that the star-shaped edge structures implied by a characteristic set not only occur in the data itself but also in the queries. As each triple describes only a single aspect of an entity, queries tend to contain multiple triple patterns with identical subject variables. Now, the key idea of using characteristic sets for selectivity estimation is to compute (and count) the characteristic sets for the data and calculate the characteristic set of the query. Subsequently, we find (and count) those characteristic sets of the data that are supersets of the characteristic set of the query.

Let us illustrate this with an example. Assume that R is the set of data triples, $\mathcal{S}_C(R)$ the set of characteristic sets, and $count(S) = |\{s | S_C(s) = S\}|$. Then, we can be compute the result cardinality of a star-join query by looking up the number of occurrences of the characteristic sets:

query:	select distinct ?e
	where { ?e <author> ?a. ?e <title> ?b. }
cardinality:	$\sum_{S \in \{S \| S \in \mathcal{S}_C(R) \wedge \{author,title\} \subseteq S\}} count(S)$

Note that this cardinality computation is exact! Furthermore, this kind of computation works for an arbitrary number of joins, correlated or not, and requires only knowledge about the characteristic sets and their frequency. Memory space is typically no issue either. For example, the characteristic sets for the 57 GB UniProt data set consume less than 64 KB.

This observation makes characteristic sets very attractive for RDF star join cardinality computations. However, this simple and exact computation is only possible due to the keyword `distinct`: In the general case, we have to take into account that the joins can produce duplicate bindings for ?e (due to different bindings of ?a and ?b).

B. Occurrence Annotations

For the above-mentioned reason, we annotate each predicate in a characteristic set with the number of occurrences of this predicate in entities belonging to the characteristic set. This will allow us to compute the multiplicities, i.e., to predict the number of variable bindings per entity caused by a certain join in a star join. Formally, the different annotations of a characteristic set $S = \{p_1, p_2, \ldots\}$ derived from a triple set R can computed as follows:

distinct	$\lvert\{s \mid \exists p, o : (s,p,o) \in R \wedge S_C(s) = S\}\rvert$
count(p_1)	$\lvert\{(s,p_1,o) \mid (s,p_1,o) \in R \wedge S_C(s) = S\}\rvert$
count(p_2)	$\lvert\{(s,p_2,o) \mid (s,p_2,o) \in R \wedge S_C(s) = S\}\rvert$
...	...

Note that in the implementation we compute and annotate the complete set $\mathcal{S}_C$ with only two group-by operators. Thus, its calculation is not expensive.

These counts are very helpful for cardinality estimation. Consider the query from above without the `distinct` clause, and assume that all entities in the RDF data set belong to the following hypothetical annotated characteristic set:

distinct	author	title	year
1000	2300	1010	1090

The first column tells us that with a `distinct` clause, we get 1000 results. That is, there are 1000 unique entities s that occur in triples of the form $(s, author, ?x)$ and in triples of the form $(s, author, ?y)$ (and in `year` triples, of course, but `year` does not occur in the query). The second column says that there are 2300 `author` triples with entities from this characteristic set, i.e., each entity has 2.3 `author` triples on average. Accordingly, the third column says that each entity has 1.01 `title` triples on average. We therefore predict that without the `distinct` clause, the result cardinality is $1000 * 2.3 * 1.01 = 2323$.

In contrast to the case with `distinct`, this computation is no longer exact in general: we average over the whole characteristic set and, therefore, introduce some error. However, entities belonging to the same characteristic set tend to be very similar in this respect. Thus, using the count for multiplicity estimations leads to very accurate predictions (see Section VII).

C. Queries with Bounded Objects

So far, we have only considered star join queries with unbound objects, that is, without restrictions on the object attributes. In many queries, at least some of the objects will be bound. In our running example, binding ?b to some concrete title would be a typical restriction. When a star join contains a triple pattern of the form $(?s, p_1, o_1)$, we can first estimate the cardinality as if the object were unbound, and then multiply with the selectivity of o_1. Note that we have to use the *conditional selectivity*:

$$sel(?o = o_1 | ?p = p_1) = \frac{sel(?o = o_1 \wedge ?p = p_1)}{sel(?p = p_1)}.$$

We do not want to use the simple selectivity of $sel(?o = o_1)$, as this would ignore the correlation between predicate and object. The RDF star join estimation itself already takes the restriction to p_1 into account. Therefore, we have to use the conditional selectivity of the object restriction given the predicate restriction.

However, even using the conditional selectivities is sometimes not accurate enough. First, a given object value might be more frequent within a certain characteristic set than within in the whole data set due to correlations. Second, this still assumes independence between object values, which in practice is frequently not true. Consider, for example, the following query patterns for our running example
$(?b, \text{title}, \text{The Tree and I})$, $(?b, \text{author}, \text{R. Pecker})$

Assuming that the author has written only one book, we would predict that each individual pattern has a selectivity of $\frac{1}{1000}$ (assuming a thousand books in the data set), and that the combined selectivity would therefore be 10^{-6}. But this is obviously not true in this case, as book title and author are highly correlated. The problem, of course, is that we do not know the degree of correlation. The two values might be independent (in which case 10^{-6} would be correct), there might be a functional dependency (in which case the selectivity is 10^{-3}), they might be anti-correlated (in which case the selectivity can be arbitrarily low), or anything in between.

It seems impossible to pre-compute the degree of correlation in full generality with reasonable effort. However, the subjects that fall within the same characteristic set tend to be quite similar, and in our experiments with real world data we noticed that usually one predicate is extremely selective (similar to a key), and then the other predicates (nearly) functionally follow from the selective one. In our book domain, the book title is nearly a key. It is not a perfect key, as there are books with identical titles, but for most books the author name follows from the title. Note that binding just the author would be very selective, too! But once the title of a book is fixed, restricting the authors has usually very little impact on the result cardinality.

When estimating the result cardinality of star joins with multiple bound object values, we only take into account the selectivity of the most selective value, and assume soft functional dependencies for the other values. This means that we ignore their selectivity, but only adjust the predicate multiplicities accordingly (see Section V-D). This leads to a slight over-estimation, but in practice, it is far more accurate than assuming independence (see Section VII for experiments).

Note that one can improve the conditional selectivity estimation of individual object predicates quite easily by storing one additional value per predicate in the characteristic sets: Instead of storing only the multiplicity, we also store the number

```
STARJOINCARDINALITY($\mathcal{S}_C$,$Q = \{(?s, p_1, ?o_1), \ldots,$
                                  $(?s, p_n, ?o_n)\}$)
$S_Q = \{p_1, \ldots, p_n\}$
$card = 0$
for each $S \in \mathcal{S}_C : S_Q \subseteq S$
  $m = 1$
  $o = 1$
  for $i = 1$ to $n$
    if $?o_i$ is bound to a value $o_i$
      $o = \min(o, sel(?o_i = o_i | ?p = p_i))$
    else
      $m = m * \frac{S.count(p_i)}{S.distinct}$
  $card = card + S.distinct * m * o$
return $card$
```

Fig. 1. Cardinality Estimation for Star Joins

of distinct object values. This increases space consumption by about 30%, but gives very useful bounds: For example, there might be 1200 authors in total, but only 50 authors of crime stories. A query involving crime stories should, therefore, base the author selectivity on the smaller domain.

In general, given a characteristic set with d distinct subjects containing (among others) a predicate p with a multiplicity p_m and p_d distinct object values, we can reasonably bound the conditional selectivity as follows:

$$sel(?o = x | ?p = p) \in [\frac{1}{p_d}, 1].$$

The lower bound assumes uniformity over the domain (which is a reasonable assumption for cardinality estimation purposes), and the upper bound assumes that all subjects qualify. Note that it is not strictly necessary to know p_d, as $p_d \leq p_m$, and in many cases $p_d \approx p_m$. Therefore, one can just use p_m instead of p_d, which is available anyway. From a theoretical point of view, using p_d is more desirable (as it leads to tighter bounds), but in our experiments using p_m worked nearly as well and we thus approximate p_d by p_m. The bounds are used to check the plausibility of the conditional selectivity. The general estimation is still useful to differentiate very frequent or very uncommon values, but if the estimate is outside the bounds, we update the estimate such that it stays within the bounds.

D. Using Characteristic Sets for Star Joins

Putting everything together, the complete algorithm for star join cardinality estimation is shown in Figure 1. It gets a set of annotated characteristic sets ($\mathcal{S}_C$) and a star join, i.e., a set of triple patterns with a common subject variable (Q) as input and estimates the result cardinality. It computes the characteristic set of the query (S_Q) and then finds all characteristic sets that are supersets and computes their contribution to the result. For unbound object values, it uses the predicate multiplicities, for bounded object values it assumes that the number of subjects is unchanged (i.e., there exists a functional dependency), but then uses the most selective of these object bindings to adjust the cardinality. As we will see in Section VII, this leads to quite accurate cardinality estimations.

Stars centered around the same entity (i.e., triples with the same subject) occur very frequently both in the data and in queries. However, one can define characteristic sets analogously for stars of incoming edges around a certain object (i.e., triples with the same object). This is less useful in practice, as there are not that many queries which contain a significant number of object-object joins. As characteristic sets are relatively cheap to compute and the space consumption is very low, we decided for RDF-3X to derive and store them, too. It might be unlikely that a user makes heavy use of object-object joins, but such queries are valid SPARQL, and a database system should try to handle all queries gracefully.

E. Handling Diverse Sets

For most data sets we examined, the number of distinct characteristic sets was very low ($< 10,000$). The only exception we found is the Billion Triples data set from the Semantic Web Challenge [7]. It consists of a *union* of many diverse data sources, leading to 484,586 distinct characteristic sets, which is somewhat unwieldy. The problem is not so much the space consumption, which is still low compared to the size of the data set, but the lookup during cardinality estimation. There, we have to find all supersets of the characteristic set of the query, which becomes noticeable as the number of characteristic sets grows.

For the Billion Triples data set, many of these characteristic sets occur only once (i.e., there is only one entity with this exact characteristic set) or a low number of times, which means that these sets do not carry much information. Usually, they are only unique due to one or two predicate combinations that occur only in this set, the remaining predicates would fit into a different characteristic set. We can therefore merge them into other sets without losing much information.

When merging characteristic sets, we have two choices:

1) We can merge directly a characteristic set S_1 into a superset S_2 ($S_1 \subseteq S_2$) by adding the counts.
2) We can split a characteristic set S into two parts S_1, S_2 that can then be merged individually.

We illustrate these two steps with an example. Assume we have four annotated characteristic sets, written as $\{(predicate, count), ..., distinct\}$

$$S_1 = \{(author, 120), 100\}$$
$$S_2 = \{(title, 230), 200\}$$
$$S_3 = \{(author, 2300), (title, 1001), (year, 1000), 1000\}$$
$$S_4 = \{(author, 30), (title, 20), 20\}$$

Now, we want to eliminate S_4. The first alternative would be to merge S_3 into S_4, producing the new set $S_3 = \{(author, 2330), (title, 1021), (year, 1000), 1020\}$. This potentially leads to an *overestimation* of results, as we now predict too many entities for queries asking for $\{author, title, year\}$. Or we could split S_4 into $\{author\}$ and $\{title\}$, and update S_1 into $\{(author, 150, 120)\}$ and $\{(title, 250, 220)\}$. This has

```
MERGECHARACTERISTICSETS($\mathcal{S}_C$,$S$)
$\bar{S} = \{S' | S' \in \mathcal{S}_C \wedge S \subseteq S'\}$
if $\bar{S} \neq \emptyset$
    $\bar{S}' = \{S' | S' \in \bar{S} \wedge |S'| = min(\{|S'| | S' \in \bar{S}\})\}$
    merge $S$ into $\arg\max_{S' \in \bar{S}'} S'.distinct$
else
    $\bar{S} = \{S' | S' \subset S \wedge \exists S'' \in \mathcal{S}_C : S' \subseteq S''\}$
    $S_1 = \arg\max_{S' \in \bar{S}} |S'|, S_2 = S \setminus S_1$
    if $S_1 \neq \emptyset$
        MERGECHARACTERISTICSETS($\mathcal{S}_C$,$S_1$)
        MERGECHARACTERISTICSETS($\mathcal{S}_C$,$S_2$)
```

Fig. 2. Merging Characteristic Sets

the advantage of being accurate for the individual predicates, but potentially leads to *underestimations* for queries asking for both predicates. Which of the two alternatives is better depends on the data and the queries. We use both, but we have decided to favor overestimations, as they usually introduce only a small error and are often less dangerous for the resulting execution plan.

After computing the characteristic sets for a data set, we check if the number of characteristic sets is less than 10,000 (or some other value, but 10,000 is reasonably small). If yes, we just keep all of them. If not, we keep the 10,000 most frequent characteristic sets in $\mathcal{S}_C$ and merge the remaining sets using the algorithm shown in Figure 2. We first try to merge a set into an existing superset. If such a superset exists, we can directly merge. When there are multiple choices, we prefer the smallest superset. If this choice is ambiguous, we choose the most frequent one, as this will cause the least error. If no superset exists, we break the characteristic set into two parts: the largest set that is a subset of a set in $\mathcal{S}_C$, and the rest. Both parts are then merged individually.

While we introduce an error due to these merges the overall accuracy tends to be very good, even when only keeping 10,000 characteristic sets (see Section VII).

VI. Exploiting Characteristic Sets

In Section V, we have introduced the novel technique of characteristic sets and its usage for cardinality estimation. Somewhat surprisingly, characteristic sets allow us to predict the result cardinality of a large number of joins accurately, but require some care for incremental cardinality computations. We therefore sketch how characteristic sets can be used for the incremental estimation process needed in bottom-up plan construction.

Characteristic sets are holistic, in the sense that they predict the cardinality of a large number of operators at once, while the plan construction works on a per-operator basis. We therefore obey the following principles while integrating our selectivity estimation into the plan generator:

1) Compute result cardinalities only once per equivalent plan and independently of how the plan was constructed.
2) Use the maximum amount of consistent correlation information that is available.
3) Assume independence if no correlation information is available.

The first principle means that we want to derive the same result cardinality for every equivalent plan. And not just because we happen to cache the cardinality estimate for the first equivalent plan we encounter, but because cardinality estimation is really independent of the plan structure. This might seem obvious, as two equivalent plans produce the same result, but most database systems build the cardinality estimates incrementally while building plans. As a consequence, the estimate for $|(A \bowtie B) \bowtie C|$ can be different from the estimate for $|A \bowtie (B \bowtie C)|$ in most commercial database systems (depending on how histograms are combined). This is very unfortunate, particularly since it causes a behavior that [8] calls ”fleeing to ignorance”. Cardinality estimates that do not consider correlations tend to underestimate results (which implies lower processing costs) and, thus, the optimizer will favor plans that are based upon poor estimates. We avoid this by deriving the query graph for each partial result, which is independent of the concrete join order, and then deriving cardinalities based upon the graph.

A query graph potentially contains a large number of joins, and we have statistics about different parts of the graph. The second principle says that we will first estimate that part of the graph for which we have the largest consistent information. Or, phrased differently, we try to make the most of the best information we have. This means that we will favor characteristic sets (which are known to be accurate and to capture correlations) over general join statistics. The join statistics are good for individual joins, but in most cases, a characteristic set will be more accurate than a combination of join statistics, simply because the characteristic set captures correlations. And capturing as many correlations as possible is essential for RDF queries.

Finally, as a last resort, we use the independence assumption. This introduces errors, as usual, and there are still many open issues left for future work. However, we use the independence assumption relatively rarely, and ”late” in the estimation process: As we favor using correlation information, if available, large parts of the query are estimated without using the independence assumption. The final result might require assuming independence, but at this point most of the cardinalities are fixed anyway, so the amount of damage that can be inflicted by the independence assumption is limited.

A. Estimation Algorithm

During plan construction, we estimate the result cardinality of partial plans by examining the query graph. Note that there are two different kinds of query graphs for a given RDF query: First, the join graph, and second, the RDF subgraph. Fig. 3 illustrates this. The RDF query graph is the graph where variables form nodes and the triple patterns form edges between the nodes (Fig. 3, left side). It corresponds to a subgraph that the query has to search for. In the join query graph, the triple patterns select ?a ?t where { ?b <author>?a. ?b <title>?t. ?b <year>”2009”. ?b <publishedBy>?p. ?p <name>”ACM”. }

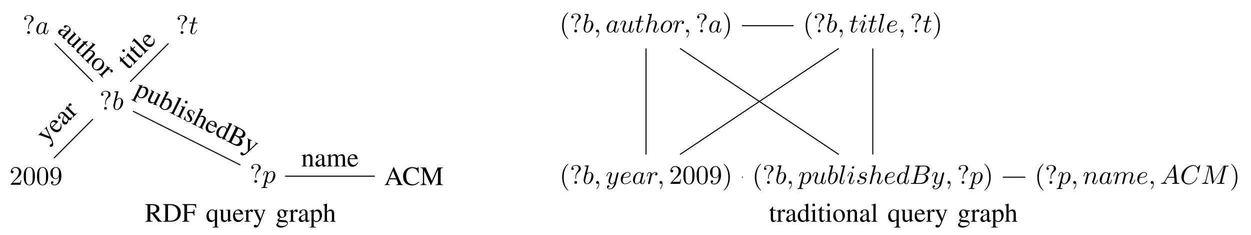

Fig. 3. A SPARQL query and its query graphs

```
ESTIMATEJOINCARDINALITY($Q$,$T$)
$P = \{p|p$ is a triple pattern occurring in $T\}$
$Q_T = Q$ restricted to nodes in $P$
return ESTIMATEQUERYCARDINALITY($Q_T$)
```

Fig. 4. Estimating the Result Cardinality of a Join Tree

form nodes and the join predicates form edges (Fig. 3, right side). It corresponds to the join possibilities within the query. The RDF query graph can be seen as the dual of the join query graph, switching the roles of nodes and edges. However, the RDF query graph is not as expressive: some (rarely used) join conditions cannot be expressed in the RDF query graph. Nonetheless, it is very convenient for cardinality reasoning. We therefore operate on both graphs, omitting constraints from the RDF query graph that cannot be expressed with it.

During query optimization, we use a Dynamic Programming (DP) strategy to find the optimal join tree for a given SPARQL query. When estimating the result cardinality of a partial plan during plan construction, we first check if we already have an equivalent plan in our DP table. If yes, we can re-use the cardinality. If not, we ignore the join structure of the current partial plan and reconstruct the query graph that corresponds to the query fragment (Figure 4). This avoids inducing a bias via a certain join order, as the result cardinality must be independent of the structure of the join tree.

Given a join graph $Q = (V, E)$, we can directly estimate the result cardinality using the independence assumption:

$$card(Q) = \prod_{R \in V} |R| \prod_{p \in E} sel(p).$$

Of course, we know that the independence assumption does not hold in general. Therefore, we try to exploit as much information about correlations as possible. The basic idea is to *cover* the query graph with statistical synopses. We search for synopses that can be used to estimate parts of the query, and then mark the corresponding part in the query graph as covered. As a consequence, we replace a number of factors in the product formula above with the estimate from the synopses. Note that it is not necessary to reconstruct the individual factors during this process. For example, for two correlated joins it is not possible to assign join selectivities without knowing the join order. But as we are only interested in the result cardinality,

```
ESTIMATEQUERYCARDINALITY($Q$)
$Q^R$ = RDF query graph derived from $Q$
$card = 1$
mark all nodes and edges in $Q$ and $Q^R$ as uncovered
while uncovered $Q^R$ contains subject star joins
  $S$=largest subject star join in the uncovered part of $Q^R$
  mark $S$ as covered in $Q^R$ and $Q$
  $card = card*$STARJOINCARDINALITY($\mathcal{S}_C$,$S$)
while uncovered $Q^R$ contains object star joins
  $S$=largest object star join in the uncovered part of $Q^R$
  mark $S$ as covered in $Q^R$ and $Q$
  $card = card*$STARJOINCARDINALITY($\mathcal{S}_C^O$,$S$)
$card = card * \prod_{R \in uncoveredQ} |R| * \prod_{\bowtie_p \in uncoveredQ} sel(\bowtie_p)$
return $card$
```

Fig. 5. Estimating the Result Cardinality for a Given Query Graph

it is sufficient to replace groups of factors with estimates. If we are unable to cover the whole graph, we assume independence and multiply the remaining factors.

The resulting algorithm is shown in Figure 5. It examines the join query graph Q and constructs the induced RDF query graph Q^R. Both graphs are initially uncovered. It then searches Q^R for the largest start join centered around a subject. We know that we can estimate these cardinalities very accurately using characteristic sets, so we prefer using characteristic sets, if possible. If there are multiple star joins, we choose the largest one, as it covers the most correlations. The estimated result size of this star is then multiplied to the factors known so far (initially 1), and the star is marked as covered both in Q and Q^R. This avoids re-applying selectivities multiple times. We repeat this until we find no more star joins. In the next step, the algorithm searches for star joins centered around objects. These estimates tend to be less accurate due to higher heterogeneity, but they still cover correlations nicely. If we find one, we proceed in the same way as for star joins around subjects. Finally, we search for any edge or node that is still uncovered, and multiply with their cardinality/selectivity, assuming independence.

For the example in Figure 3, the algorithm will start by choosing the star centered around b, estimate it using characteristic sets, and mark the whole star as covered. This will leave only the node $(?p, name, ACM)$ and its join edge uncovered in the join graph. The RDF graph does not contain any other uncovered star, so the algorithm checks the join graph for uncovered parts and will find the node and the join edge. Both are estimated using the basic join selectivity estimation, and the complete cardinality is returned.

The whole process favors accurate estimates over inaccurate parts and estimating large portions in one step to cover correlations. As we will see in Section VII, this results in very good estimates for result cardinalities.

VII. Evaluation

To study the accuracy of our new estimation technique, we implemented the characteristic sets estimation in the RDF database system RDF-3X [2] and compared the new estimator with the original RDF-3X estimator. In addition, we loaded the same data sets using the integer triple representation described in [2] into three major commercial database systems. As only one of them (namely DB2) allows us to publish benchmark results, we will anonymize all three and call them DB A, DB B, and DB C. In addition, we included two approaches from the RDF community, namely the approach by Stocker et al. [3], and the subgraph mining approach by Maduko et al. [4]. We first study the accuracy of the different approaches using single join queries in different data sets, and then examine cardinality estimations for more complex queries. These experiments were carried out using two data sets. Sec. VII-D summarizes experiments with other data sets. Sec. VII-E takes a look at the impact of the improved cardinality on the generated plans.

As the experiments were conducted on different systems and under different operating systems (due to the nature of the commercial database systems), we do not give load times for the various systems. The load times for RDF-3X for the different data sets can be found in [2], [9]. The additional effort for the characteristic set computation is negligible compared to the full load, as it requires only a single linear scan over the aggregated SP index of RDF-3X (which directly produces S_C entries) plus an in-memory group-by into a few thousand groups (to insert the individual S_C into the $\mathcal{S}_C$ set). All cardinality estimates were obtained by using the various *explain* features of the different database systems to get the estimates used by the query optimizer. When a database system estimated less than one tuple, we rounded this up to one, as all queries produced non-empty results.

A. Accuracy Experiments

B. Single Join Queries

For the data sets Yago and LibraryThing [9], we generated queries of the form $(?S, p1, ?O1), (?S, p2, ?O2)$, where all possible combinations of $p1$, $p2$ were considered. This resulted in 1751 queries for Yago and 19.062.990 queries for LibraryThing. For each query and cardinality estimation method (system), we calculated the q-error $\max(c/\hat{c}, \hat{c}/c)$ [10], where c is the correct cardinality and $\hat{c}$ the estimate. Thus, the q-error denotes the factor by which the estimate differs from the true value. We then calculate the percentage of all queries where the q-error falls into a bucket $]b_i, b_{i+1}]$, where the b_i are taken from the values $2, 5, 10, 100, 1000, \infty$. Additionally, we kept the maximal occurring q-error. Tables III and IV show the results. All commercial DBMSs, the approach by Stocker at al., as well as RDF-3X tend to provide very bad estimates, which are often (almost in half of the cases, depending on the system) a factor of 100 and more off the true value. The subgraph mining approach from Maduko et al. has a mixed result. For the Yago data set the number of subgraphs is quite low, therefore it effectively materializes the answers for all single join queries, providing perfect cardinality estimations. For the LibraryThing data set however the number of subgraphs with up to two (!) edges is already 1,296,519,484, which cannot be materialized anymore. Here, we used the pruning techniques from [4] to reduce the storage consumption to 50,000 subgraphs with two edges plus all subgraphs containing a single edge. This keeps the space consumption at a reasonable 21.0 MB. However, the estimates becomes less accurate: the error grows up to 17,471! Contrarily, cardinality estimations based on characteristic sets (CS) are very accurate for both data sets: the maximal q-error is less than 3. Furthermore, in 99.9% of all cases the maximal q-error produced is less than 2! The space consumption for the characteristic sets was 0.5 MB for the Yago and 8.3 MB for the LibraryThing data sets.

TABLE I
Cardinality Estimation Errors for LibraryThing Queries

	card	error		
	(g.mean)	median	max	avg
exact	26347			
our	13730	0.77	11.34	1.86
RDF-3X	83	180.56	395397.00	46506.80
Stocker	1	15863.00	$6.45*10^6$	994426.00
Maduko	3	20591.00	$6.50*10^6$	953590.00
DB A	1	15863.00	$6.45*10^6$	994426.00
DB B	71	1464.81	$2.37*10^6$	$1.29*10^6$
DB C	2	7826.75	$2.37*10^6$	$1.61*10^6$

TABLE II
Cardinality Estimation Errors for Yago Queries

	card	error		
	(g.mean)	median	max	avg
exact	1741			
our	1244	0.17	12.60	1.83
RDF-3X	20	64.72	768.86	235.01
Stocker	1	1333.00	520293.00	83145.00
Maduko	75	29.00	3491.55	451.25
DB A	3	336.00	278266.00	31548.10
DB B	722	469.80	70539.20	40098.80
DB C	35	29.85	11547.00	41453.70

C. Complex Queries

The microbenchmark in the previous section studied all possible combinations and, therefore, shows the whole spectrum, but it considers only very simple queries. To study cardinality estimation for more complex queries, we manually constructed queries with up to 6 joins and including additional object constraints. The queries and the detailed results are included in

TABLE III
SINGLE JOIN CARDINALITY ESTIMATION PRECISION FOR THE YAGO DATA SET

	Yago						
q-error	DB A	DB B	DB C	Stocker	Madoku	RDF-3X	CS
≤ 2	16.6	23.4	25.6	0	100[1]	14.9	99.9
≤ 5	12.2	16.0	16.4	0	0	20.7	0.1
≤ 10	7.6	10.2	10.0	2.2	0	16.0	0
≤ 100	40.6	21.4	21.7	1.1	0	38.5	0
≤ 1000	19.7	14.4	14.0	3.2	0	8.9	0
> 1000	3.3	14.6	12.2	93.5	0	0.9	0
max	314275	1731400	783276	$3.8 * 10^{14}$	1[1]	7779527	2.97

[1] all queries were precomputed for this approach

TABLE IV
SINGLE JOIN CARDINALITY ESTIMATION PRECISION FOR THE LIBRARYTHING DATA SET

	LibraryThing						
q-error	DB A	DB B	DB C	Stocker	Maduko	RDF-3X	CS
≤ 2	15.0	23.2	22.9	0	26.7	30.2	100
≤ 5	10.5	27.3	27.8	0	40.6	30.9	0
≤ 10	10.3	17.2	17.7	0	19.7	16.6	0
≤ 100	35.3	28.8	27.8	0.1	12.0	19.9	0
≤ 1000	20.7	3.1	3.3	0	0.8	2.2	0
> 1000	8.1	0.4	0.6	99.9	0.1	0.2	0
max	28367552	1416363	7140611	$1.3 * 10^{15}$	17471	2909310	1.01

the appendix, the estimation errors (p-error $\max(c/\hat{c}, \hat{c}/c) - 1$ [10]) are summarized in Table I and II. Again, the characteristic sets perform very well, with an average estimation error of about 1.8 even for reasonably complex queries, while the other approaches are multiple orders of magnitude worse. The second column in the tables includes the geometric mean of the estimated cardinalities themselves, and it can be seen that the other approaches tend to vastly underestimate the result cardinality.

D. Other Data Sets

Apart from the previously mentioned data sets, we also studied the other data sets used in [2], [9], which are from different domains. Due to space constraints, we can only summarize the findings here.

The Barton data set of library data contains 8,285 distinct characteristic sets and can be estimated very well using characteristic sets. Somewhat surprisingly, the UniProt data set [6] of protein information, which is 57GB in size and contains 845,074,885 unique triples, contains only 615 distinct characteristic sets. As a consequence, characteristic sets lead to an accurate cardinality estimation even for this huge data set using a very small synopsis. The reason for this small size is probably the well-structured and very regular nature of the UniProt data, i.e., the fact that UniProt basically has a schema. This is the ideal case, and characteristic sets work very well in such a setting.

The other extreme is the Billion Triples data set [7], which contains 88GB of data and is a union of twelve different data sources (including web crawls). Accordingly, the data is very noisy, and contains a huge tail of predicate combinations that occur only once. The number of distinct characteristic sets for this data set is 484,586, which is beginning to become unwieldy. The estimation precision is very good, but storing such a large number of characteristic sets seems to be wasteful, in particular since most of them just cover some corner cases. We therefore used the merge strategy from Section V-E to reduce the number of characteristic sets to 10,000. The microbenchmark precision results (similar to Section VII-A, the 10K most frequent predicate parts) are shown below:

	≤ 2	≤ 5	≤ 10	≤ 100	≤ 1000	> 1000
full CS	99.2	0.4	0.1	0.3	0.0	0.0
merged CS	91.7	1.7	0.9	1.3	0.2	4.1

Reducing the number of characteristic sets does effect the accuracy, of course, but overall, the estimates are still very good. More than 90% of the estimates are off by less than a factor of two, even though we reduced the number of characteristic sets by more than a factor of 40. A slight problem remains with predicate combinations that vanish during the merge and for which the characteristic sets, therefore, will predict an empty result (which then tends to end up in the last error bucket). In principle, we could notice this case (assuming that no user will ask for an empty result) and could probably switch to a more conservative cardinality estimate in the future.

E. Effect on Generated Plans

Even though the main focus of this paper is the cardinality estimation itself, we were also interested in the effect of estimation errors on the generated plans. We therefore took the queries used in the previous section (shown in detail in the appendix), and executed them both in the original RDF-3X system and in an RDF-3X variant using our new estimation method. We examined the generated plan and executed the queries on a Dell E6500 (Intel Core2 Duo T9600), measuring the median of ten execution times.

For the Yago queries, the better estimates changed the execution plan in two cases, namely query Q8 and Q9. Both new plans were clearly better, with execution times of 8ms and 1ms respectively, vs. 12ms and 2ms in the original RDF-3X system. For the LibraryThing queries, four queries were changed, namely Q4, Q6, Q9, and Q10. Again, the execution times improved in all cases, from 41ms/3ms/2ms/1ms in the original RDF-3X to 28ms/1ms/1ms/<1ms using our new estimator. The absolute differences are not that large, as RDF-3X was very fast to begin with, but the relative improvement is quite large. It is interesting to note that the commercial database systems frequently needed several minutes to execute the queries or were even unable to execute queries in reasonable time, due to large estimation errors and resulting bad plans.

VIII. Conclusion

We introduced a novel RDF synopsis called *characteristic sets* and experimentally showed that cardinality estimations based upon it are very precise. More specifically, the experiments have shown that they greatly outperform both commerical database systems and other specialized RDF estimation techniques. This gained precision has certainly a positive impact on the runtime of execution plans. Thus, query execution is accelerated.

Acknowledgment

We thank Uwe Steinel for the administration of the three commercial database systems and Sebastian Sindermann for carrying out the experiments with them. We also thank Simone Seeger for her help preparing the manuscript.

References

[1] Y. E. Ioannidis, "The history of histograms (abridged)," in *VLDB*, 2003, pp. 19–30.
[2] T. Neumann and G. Weikum, "Scalable join processing on very large rdf graphs," in *SIGMOD*, 2009, pp. 627–640.
[3] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "Sparql basic graph pattern optimization using selectivity estimation," in *WWW*, 2008, pp. 595–604.
[4] A. Maduko, K. Anyanwu, A. P. Sheth, and P. Schliekelman, "Graph summaries for subgraph frequency estimation," in *ESWC*, 2008, pp. 508–523.
[5] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *WWW*, 2007.
[6] "Uniprot RDF," http://dev.isb-sib.ch/projects/uniprot-rdf/.
[7] "Semantic web challenge 2008. billion triples track," http://challenge.semanticweb.org/.
[8] V. Markl, N. Megiddo, M. Kutsch, T. M. Tran, P. J. Haas, and U. Srivastava, "Consistently estimating the selectivity of conjuncts of predicates," in *VLDB*, 2005, pp. 373–384.
[9] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *VLDB J.*, vol. 19, no. 1, pp. 91–113, 2010.
[10] G. Moerkotte, T. Neumann, and G. Steidl, "Preventing bad plans by bounding the impact of cardinality estimation errors," *PVLDB*, vol. 2, no. 1, pp. 982–993, 2009.

Appendix

For completeness, we include the concrete queries used in Section VII and the individual cardinality estimates of all database systems in this section. In some cases a database system predicted a cardinality of less than one, which makes no sense for non-empty query results. When this happened we rounded the estimates up to one.

A. LibraryThing

The individual cardinality estimates for the following queries are shown in Table V.

Q1: select ?s ?t ?y where { ?s <hasTitle> ?t. ?s < hasAuthor> "Jane Austen". ?s <inYear> ?y }
Q2: select ?s ?t where { ?s <hasTitle> ?t. ?s < hasAuthor> "Jane Austen". ?s <inYear> <2003> }
Q3:
select ?s where { ?s <crime> ?b. ?s <romance> ?b2. ?s <poetry> ?b3. ?s <hasFavoriteAuthor> "Neil Gaiman" }
Q4: select ?s where { ?s <crime> ?b. ?s <politics> ?b2. ?s <romance> ?b3. ?s <poetry> ?b4. ?s <cookbook> ?b5. ?s <ocean\ life> ?b6. ?s <new\ mexico> ?b7 }
Q5: select ?s ?t1 ?t2 where { ?s <inYear> <1975>. ?s <hasAuthor> "T.S. Eliot". ?s <hasTitle> ?t1. ?s <hasTitle > ?t2 }
Q6: select ?s where { ?s < thriller > ?b. ?s < politics > ?b2. ?s <conspiracy> ?b3. ?s <hasFavoriteAuthor> ?a }
Q7: select ?s where { ?s < thriller > ?b. ?s < politics > ?b2. ?s <conspiracy> ?b3. ?s <hasFavoriteAuthor> "Robert B. Parker" }
Q8: select ?s where { ?s < thriller > ?b. ?s < politics > ?b2. ?s <conspiracy> ?b3. ?s <hasFavoriteAuthor> "Noam Chomsky" }
Q9: select ?s where { ?s < politics > ?b1. ?s <society> ?b2. ?s <future> ?b3. ?s <democracy> ?b4. ?s <british> ?b5. ?s <hasFavoriteAuthor> "Aldous Huxley" }
Q10: select ?s where { ?s < politics > ?b1. ?s <society> ?b2. ?s <future> ?b3. ?s <democracy> ?b4. ?s <british> ?b5. ?s <hasFavoriteAuthor> "Aldous Huxley". ?s < hasFavoriteAuthor> "George Orwell" }

B. Yago

The individual cardinality estimates for the following queries are shown in Table VI.

Q1: select ?s ?l ?n ?t where { ?s <bornInLocation> ?l. ?s <isCalled> ?n. ?s <type> ?t. }
Q2: select ?s ?n ?t where { ?s <bornInLocation> < Stockholm>. ?s <isCalled> ?n. ?s <type> ?t. }
Q3: select ?s ?c ?n ?w where { ?s <producedInCountry> ?c. ?s <isCalled> ?n. ?s <hasWebsite> ?w }
Q4: select ?s ?n ?w where { ?s <producedInCountry> < Spain>. ?s <isCalled> ?n. ?s <hasWebsite> ?w }
Q5: select ?s ?l ?n ?d ?t where { ?s <diedInLocation> ?l. ?s <isCalled> ?n. ?s <diedOnDate> ?d. ?s <type> ?t }

TABLE V
INDIVIDUAL CARDINALITY ESTIMATES FOR THE LIBRARYTHING QUERIES

query	true cardinality	our method	RDF-3X	Stocker	Maduko	DB A	DB B	DB C
Q1	9,039	9,107	4,361	1	1,535	1	131,556	2
Q2	1,252	437	54	1	13	1	1,288	1
Q3	2,378,386	1,888,480	6	1	1	1	1	1
Q4	20,592	20,592	18,557	1	1	1	129	26
Q5	16	5	1	1	1	1	58	1
Q6	6,451,809	6,451,810	75,088	1	1	1	2,543,650	28
Q7	11,136	4,872	10	1	1	1	1	1
Q8	2,772	3,269	10	1	1	1	1	1
Q9	302,096	122,331	9	1	1	1	1	1
Q10	302,096	24,466	9	1	1	1	1	1

TABLE VI
INDIVIDUAL CARDINALITY ESTIMATES FOR THE YAGO QUERIES

query	true cardinality	our method	RDF-3X	Stocker	Maduko	DB A	DB B	DB C
Q1	520,294	480,018	26,223	1	484,432	158	435,929,000	24,967
Q2	1,334	1,102	58	1	4	214	1	1
Q3	19,042	18,839	445	1	26,330	1	10,161,000	43,289
Q4	266	113	3	1	1	1	1	1
Q5	278,267	248,081	410	1	225,378	1	19,629,000,000	8,773
Q6	11,548	4,893	15	1	3	1	1	1
Q7	408	30	1	1	1	1	1	1
Q8	264	263	1	1	180	1	440	325
Q9	30	103	1	1	1	1	1	1
Q10	7	8	1	1	1	1	1	1

Q6:
select ?s ?n ?d ?t where { ?s <diedInLocation> <Paris>. ?s <isCalled> ?n. ?s <diedOnDate> ?d. ?s <type> ?t }
Q7: select ?s ?n ?d where { ?s <diedInLocation> <Paris >. ?s <isCalled> ?n. ?s <diedOnDate> ?d. ?s <type> < wordnet_person_100007846> }
Q8: select ?s ?l ?u ?c ?m where { ?s < hasOfficialLanguage> ?l. ?s <hasUTCOffset> ?u. ?s < hasCapital> ?c. ?s <hasCurrency> ?m }
Q9: select ?s ?l ?c ?m where { ?s <hasOfficialLanguage > ?l. ?s <hasUTCOffset> <1>. ?s <hasCapital> ?c. ?s < hasCurrency> ?m }
Q10: select ?s ?c ?m where { ?s <hasOfficialLanguage> <French_language>. ?s <hasUTCOffset> <1>. ?s < hasCapital> ?c. ?s <hasCurrency> ?m }