Top-k Linked Data Query Processing

Andreas Wagner, Thanh Tran Duc, Günter Ladwig,
Andreas Harth, and Rudi Studer

Institute AIFB, Karlsruhe Institute of Technology, Germany
{a.wagner,ducthanh.tran,guenter.ladwig,harth,studer}@kit.edu

Abstract. In recent years, top-k query processing has attracted much attention in large-scale scenarios, where computing only the k "best" results is often sufficient. One line of research targets the so-called *top-k join* problem, where the k best final results are obtained through joining partial results. In this paper, we study the top-k join problem in a Linked Data setting, where partial results are located at different sources and can only be accessed via URI lookups. We show how existing work on top-k join processing can be adapted to the Linked Data setting. Further, we elaborate on strategies for a better estimation of scores of unprocessed join results (to obtain *tighter bounds* for early termination) and for an *aggressive pruning* of partial results. Based on experiments on real-world Linked Data, we show that the proposed top-k join processing technique substantially improves runtime performance.

1 Introduction

In recent years, the amount of Linked Data has increased rapidly. According to the Linked Data principles[1], dereferencing a Linked Data URI via HTTP should return a machine-readable description of the entity identified by the URI. Each URI therefore represents a virtual "data source" (see Fig. 1).

In this context, researchers have studied the problem of Linked Data query processing [3,5,6,10,11,16]. Processing structured queries over Linked Data can be seen as a special case of federated query processing. However, instead of relying on endpoints that provide structured querying capabilities (e.g., SPARQL interfaces), only HTTP URI lookups are available. Thus, entire sources have to be retrieved. Even for a single trivial query, hundreds of sources have to be processed in their entirety [10]. Aiming at delivering up-to-date results, sources often cannot be cached, but have to be fetched from external hosts. Thus, *efficiency* and *scalability* are essential problems in the Linked Data setting.

A widely adapted strategy for dealing with efficiency and scalability problems is to perform top-k processing. Instead of computing all results, *top-k query processing* approaches produce only the "best" k results [8]. This is based on the observation that results may vary in "relevance" (which can be quantified via a ranking function), and users, especially on the Web, are often interested in only a few relevant results. Let us illustrate top-k Linked Data query processing:

[1] http://www.w3.org/DesignIssues/LinkedData.html

Src. 1. ex:beatles

```
ex:beatles
 foaf:name
    "The␣Beatles";
 ex:album ex:
      sgt_pepper.
 ex:album ex:help;
```

Src. 2. ex:sgt_pepper

```
ex:sgt_pepper
 foaf:name
  "Sgt.␣Pepper";
 ex:song "Lucy".
```

Src. 3. ex:help

```
ex:help
 foaf:name "Help!";
 ex:song "Help!".
```

Fig. 1. Linked Data sources describing "The Beatles" and their songs "Help!" and "Lucy"

```
1  SELECT * WHERE
2  {
3   ex:beatles ex:album ?album .
4   ?album ex:song ?song .
5  }
```

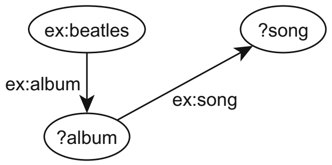

Fig. 2. Example query returning songs in Beatles albums. The query comprises two triple patterns q_1 (line 3) and q_2 (line 4).

Example 1. For the query in Fig. 2, the URIs `ex:beatles`, `ex:help` and `ex:sgt_pepper` are dereferenced to produce results for ?*song* and ?*album*. The results are retrieved from different sources, which vary in "relevance" (i.e., `ex:help` provides the precise name for the song "Help!", while `ex:sgt_pepper` merely holds "Lucy" as name for a song, which is actually called "Lucy in the Sky with Diamonds"). Such differences are captured by a ranking function, which is used in top-k query processing to measure the result relevance. For the sake of simplicity, assume a ranking function assigns triples in `ex:beatles` (s_1) a score of 1, triples in `ex:sgt_pepper` (s_2) score 2, and those in `ex:help` (s_3) a score of 3. Further, assume our ranking function assigns increasing values with increasing triple relevance.

While being appealing, top-k processing has not been studied in the Linked Data (and the general RDF) context before. Aiming at the adaption of top-k processing to the Linked Data setting, we provide the following contributions:

– Top-k query processing has been studied in different contexts [8]. Closest to our work is top-k querying over Web-accessible databases [20]. However, the Linked Data context is unique to the extent that only URI lookups are available for accessing data. Instead of retrieving partial results matching query parts from sources that are exposed via query interfaces (of the corresponding database endpoints), we have to retrieve entire sources via URI lookups. To the best of our knowledge, this is the first work towards top-k Linked Data query processing.
– We show that in a Linked Data setting, more detailed score information is available. We propose strategies for using this knowledge to provide tighter score bounds (and thus allow an earlier termination) as compared to top-k processing in other scenarios [2,13,17]. Further, we propose an aggressive technique for pruning partial query results that cannot contribute to the final top-k result.

– We perform an experimental evaluation on Linked Data datasets and queries to show that top-k processing leads to increased performance (35 % on average). We further show that our proposed top-k optimizations increase the performance compared to a baseline implementation by 12 % on average.

Outline. In Section 2, we introduce the problem of Linked Data query processing. In Section 3, we show how to adapt top-k join processing methods to the Linked Data setting. In Sections 3.3 and 3.4, we propose two optimizations: tighter bounds on future join results and a way to prune unnecessary partial results. We present our evaluation in Section 4 and discuss related work in Section 5, before concluding with Section 6.

2 Linked Data Query Processing

Data Model. We use RDF [9] as our data model. However, for clarity of presentation, we do not consider the special RDF semantics (e.g., RDF blank nodes) and focus on the main characteristics of RDF. Namely, RDF can be considered as a general model for graph-structured data encoded as $\langle s, p, o \rangle$ triples:

Definition 1 (RDF Triple, RDF Graph). *Given a set of URIs U and a set of literals L, $t = \langle s, p, o \rangle \in U \times U \times (U \cup L)$ is a* RDF triple*, and a set of RDF triples is called a* RDF graph*.*

The Linked Data principles used to access and publish RDF data on the Web, mandate that (1) HTTP URIs shall be used as URIs and that (2) dereferencing a URI returns a description of the resource identified by the URI. Thus, a URI d can be seen as a *Linked Data source*, whose content, namely a set of RDF triples T^d, is obtained by dereferencing d. Triples in T^d contain other HTTP URI references (*links*), connecting d to other sources. The union set of sources in U forms a *Linked Data graph* $G = \{t | t \in T^{d_i} \wedge d_i \in U\}$.

Query Model. The standard language for querying RDF is SPARQL [15]. Previous work on Linked Data query processing focused on processing *basic graph patterns* (BGP), which is a core feature of SPARQL.

Definition 2 (Triple Pattern, Basic Graph Pattern). *A* triple pattern *has the form $q = \langle s, p, o \rangle$, where s, p and o is either a URI, a literal or a variable. A* basic graph pattern *is a set of triple patterns $Q = \{q_1, \ldots, q_n\}$.*

Result Model. Often, every triple pattern in a BGP query Q shares a common variable with at least one other pattern such that Q forms a connected graph. Computing *results* to a BGP query over G amounts to the task of *graph pattern matching*. Basically, a result to a query Q evaluated over G (given by $\mu_G(Q)$) is a subgraph of G that matches Q. The set of all results for query Q is denoted by $\Omega_G(Q)$.

Query Processing. Traditionally, a query Q is evaluated by obtaining bindings for each of its triple patterns and then performing a series of equi-joins between bindings obtained for patterns that share a variable. In the Linked Data context, BGP queries are evaluated against all sources in the Linked Data graph G. While some sources may be available locally, others have to be *retrieved via HTTP dereferencing during query processing.*

For this, exploration-based *link traversal* [6,5] can be performed *at runtime.* The link traversal strategy assumes that Q contains at least one URI d as "entry point" to G. Starting from triples in T^d, G is then searched for results by following links from d to other sources. Instead of exploring sources at runtime, knowledge about (previously processed) Linked Data sources in the form of statistics has been exploited to determine and rank relevant sources [3,10] *at query compilation time.* Existing approaches assume a *source index*, which is basically a map that associates a triple pattern q with sources containing triples that match q. Let the result of a lookup in the source index for q be $source(q)$.

Given a source index, Linked data query processing can be conceived as a series of *operators.* We identify the *source scan* as a distinctive operator in Linked Data query processing. Given a source d, $scan(d)$ outputs all triples in T^d. A *selection* $\sigma_{T^d}(q)$ is performed on T^d to output triples that match a triple pattern q. Two triple patterns q_i and q_j that share a common variable are combined via an equi-*join* operator $q_i \bowtie q_j$ (i.e., bindings for q_i respectively q_j are joined). In general, $Q_i \bowtie Q_j$ joins any subexpression $Q_i \subset Q$ with one other $Q_j \subset Q$ ($Q_i \cap Q_j = \varnothing$). Note, in the following, we refer to an equi-join simply as join. Also, we have $\bigcup(I_1, \ldots, I_n)$, which outputs the *union* of its inputs I_i. For clarity of presentation, we assume triple patterns form a connected graph such that a join is the only operator used to combine triples from different patterns. Then, Linked Data query processing can be modeled as a tree-structured plan as exemplified in Fig. 3 (a).

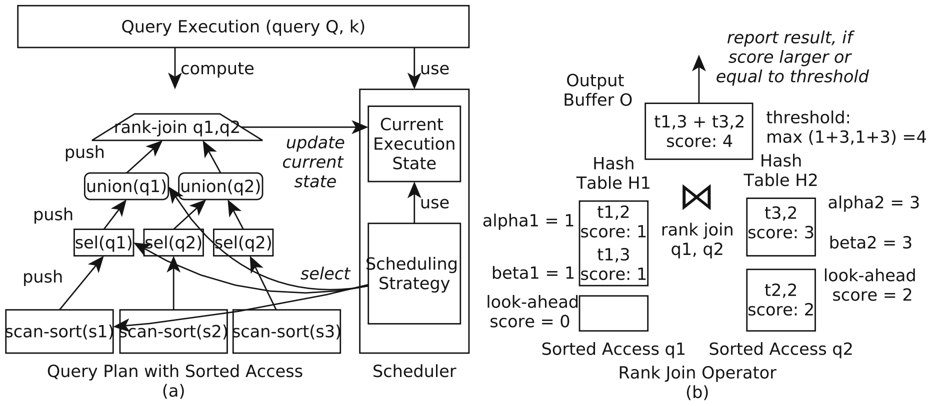

Fig. 3. (a) Query plan providing a sorted access, query execution and the scheduler. (b) Rank join operator with data from our "Beatles" example.

Query plans in relational databases generally consist of access plans for individual relations. Similarly, Linked Data query plans can be seen as being composed of *access plans* at the bottom-level (i.e., one for each triple pattern). An *access plan* for query $Q = \{q_1, \ldots, q_n\}$ is a tree-structured query plan constructed in the following way: (1) At the lowest level, leaf nodes are source scan operators, one for every source that is relevant for triple pattern q_i (i.e., one for every $d \in source(q_i)$). (2) The next level contains selection operators, one for every scan operator. (3) The root node is a union operator $\bigcup(\sigma_{T^{d_1}}(q_i), \ldots, \sigma_{T^{d_n}}(q_i))$, which combines the outputs of all selection operators for q_i (with $d_i \in source(q_i)$). At the next levels, the outputs of the access plans (of their root operators) are successively joined to process all triple patterns of the query, resulting in a *tree of operators*.

Example 2. Fig. 3 (a) shows an example query plan for the query in Fig. 2. Instead of *scan* and *join*, their top-k counterparts *scan-sort* and *rank-join* are shown (explained in the next section). There are three source scan operators, one for each of the sources: `ex:beatles` (s1), `ex:sgt_pepper` (s2), and `ex:help` (s3). Together with selection and union operators, they form two access plans for the patterns q_1 and q_2. The output of these access plans is combined using one join operator.

Push-Based Processing. In previous work [10,11], push-based execution using symmetric hash join operators was shown to have better performance than pull-based implementations (such as [6]). In a push-based model, operators push their results to subsequent operators instead of pulling from input operators, i.e., the execution is driven by the incoming data. This leads to better behavior in network settings, because, unlike in pull-based execution models, the query execution is not *blocked*, when a single source is delayed [11].

3 Top-k Join Linked Data Query Processing

Top-k query processing [14,7,17] aims at a more efficient query execution by focusing on the k best results, while skipping the computation of remaining results. This *early termination* can lead to a significant reduction in the number of inputs to be read and processed, which translates to performance improvements. We now discuss how existing *top-k join* (also called *rank join*) strategies can be be adopted to the Linked Data query processing problem as presented before. Further, we present an optimization towards tighter bounds and an aggressive result pruning. Throughout the query processing we do not approximate, thus, our approach always reports correct and complete top-k final results.

3.1 Preliminaries

Besides the source index employed for Linked Data query processing, we need a *ranking function* as well as a *sorted access* for top-k processing [7,14,17].

Ranking Function. We assume the existence of a ranking function for determining the "importance" of triples and (partial) query results in G:

Definition 3 (Ranking Function). *Let a* ranking function $\upsilon : G \mapsto [0, 1]$ *assign scores to triples in G. Further, let higher scores denote higher triple importance. Given Q as a query over G and $\Omega_G(Q)$ as its results, υ ranks results in $\Omega_G(Q)$ (i.e., $\upsilon : \Omega_G(Q) \mapsto [0, 1]$) as an aggregation of their triple scores: $\mu \in \Omega_G(Q)$: $\upsilon(\mu) = \Delta(\upsilon(t_1), \ldots, \upsilon(t_n)), t_i \in \mu$, where Δ is a monotonic aggregation function.*

Scores for triples can, e.g., be obtained through PageRank inspired ranking [4] or witness counts [1].

Sorted Access. A *sorted access* on a given join input allows to access input elements in *descending* score order. In a database setting, a sorted access can be efficiently provided by using a score index over the input data. In particular, while work on top-k join processing over Web-accessible databases [20] aims at a similar setting, it also assumes such a complete index. However, in the Linked Data context, only source statistics are assumed to be available, while the contained triples are not indexed (e.g., for the sake of result freshness). Following this tradition, we only assume that score bounds are known (i.e., computed at indexing time) for the sources, while triples are ranked and sorted on-the-fly.

Definition 4 (Source Score Bounds). *Given a source $d \in U$, its* upper bound score *$\upsilon_u(d)$ is defined as the maximal score of the triples contained in d, i.e., $\upsilon_u(d) = \max\{\upsilon(t)|t \in T^d\}$. Conversely, the* lower bound score *is defined as $\upsilon_l(d) = \min\{\upsilon(t)|t \in T^d\}$.*

For each triple pattern in the source index, we store its list of relevant sources in descending order of their upper bound scores υ_u. This allows sources for each *union* operator to be retrieved sequentially in the order of their upper bound scores. Further, as triples for a given source are not sorted, we replace each *scan* operator with a *scan-sort* operator. A *scan-sort* operator, after retrieving a source d, sorts its triples T^d according to their scores. However, if two (or more) sources (say, d_i and d_j) have overlapping source score bounds (i.e., $\upsilon_l(d_i) < \upsilon_u(d_j) < \upsilon_u(d_i)$), and both are inputs for the same union, the output of the union will not be ordered if these sources are retrieved individually. We address this problem by treating both sources as "one source". That is, d_i and d_j are scanned and sorted via the same *scan-sort* operator. Fig. 3 (a) shows an access plan with *scan-sort* operators, which provide a sorted access to the bindings of q_1 and q_2. Last, note that $\upsilon_u(d)$ respectively $\upsilon_l(d)$ does not necessarily have to be precise, it could also be estimated (e.g., based on scores of similar sources).

3.2 Push-Based Top-k Join Processing

Based on the ranking function, source index, and our sorted access mechanism, we can now adapt top-k strategies to the Linked Data setting. However, previous work on the top-k join problem uses pull-based processing, i.e., join operators actively "pull" their inputs in order to produce an output [7,17,20]. In compliance with [17], we adapt the *pull/bound rank join* (PBRJ) algorithm template for a *push*-based execution in the Linked Data setting. For simplicity, the following presentation of the PBRJ algorithm assumes *binary* joins (i.e., each join has two inputs).

In a pull-based implementation, operators call a `next` method on their input operators to obtain new data. In a push-based execution, the control flow is inverted, i.e., operators have a `push` method that is called by their input operators. Algorithm 1 shows the `push` method of the PBRJ operator. The input from which the input element r was pushed is identified by $i \in \{1, 2\}$. Note, by input element we mean either a triple (if the input is a *union* operator) or a partial query result (if the input is another *rank join* operator). First, the input element r is inserted into the hash table H_i (line 3). Then, we probe the other input's hash table H_j for valid join combinations (i.e., the join condition evaluates to "true"; see line 4), which are then added to the output queue O (line 5). Output queue O is a priority queue such that the result with the highest score is always first. The threshold Γ is updated using the *bounding strategy* $\mathcal{B}$, providing an upper bound on the scores of future join results (i.e., result combinations comprising "unseen" input elements). When a join result in queue O has a score equal to or greater than the threshold Γ, we know there is no future result having a higher score. Thus, the result is ready to be reported to a subsequent operator. If output O contains k results, which are ready to be reported, the algorithm stops reading inputs (so-called early termination).

As reported in [17], the PBRJ has two parameters: its *bounding strategy* $\mathcal{B}$ and its *pulling strategy* $\mathcal{P}$. For the former, the *corner-bound* is commonly employed and is also used in our approach. The latter strategy, however, is proposed for a pull-based execution and is thus not directly applicable. Similar to the idea behind the pulling strategy, we aim to have control over the results that are pushed to subsequent operators. Because a push-based join has no influence over the data flow, we introduce a *scheduling strategy* to regain control. Now, the `push` method only adds join results to the output queue O, but does not push them to a subsequent operator. Instead the pushing is performed in a separate `activate` method as mandated by the scheduling strategy.

Algorithm 1. PBRJ.*push*(r)

```
Input: Pushed input element $r$ on input $i \in \{1, 2\}$
Data: Bounding strategy $\mathcal{B}$, output queue $O$, threshold $\Gamma$, hash tables $H_1, H_2$
1 if $i = 1$ then $j = 2$;
2 else $j = 1$;
3 Insert $r$ into hash table $H_i$;
4 Probe $H_j$ for valid join combinations with $r$ ;
5 foreach valid join combination $o$ do Insert $o$ into $O$;
6 $\Gamma \leftarrow \mathcal{B}$.update();
```

Bounding Strategies. A bounding strategy is used to update the current threshold (i.e., the upper bound on scores of future join results). As only those results in the output queue can be reported that have a score equal to or greater than the threshold Γ, it is essential that the upper bound is as low (tight) as possible. In other words, a tight upper bound allows for an early termination of the top-k join procedure, which results in less sources being loaded and triples being processed. The most common choice for $\mathcal{B}$ is the *corner bound* strategy:

Definition 5 (Corner-Bound). *For a rank join operator, we maintain an upper bound α_i and a lower bound β_i (both initialized as ∞) on the scores of its input elements from $i \in \{1, 2\}$, where α_i is the score of the first (highest) element r_i^{max} received on input i, $\alpha_i = v(r_i^{max})$, and β_i is the score of the most recently received input element $\hat{r}_i$, $\beta_i = v(\hat{r}_i)$. Then, the threshold Γ for future join results is given by* $\max\{\Delta(\alpha_1, \beta_2), \Delta(\alpha_2, \beta_1)\}$*, i.e., the score for the join between r_1^{max} and $\hat{r}_2$ or between $\hat{r}_1$ and r_2^{max}.*

Scheduling Strategies. Deciding which input to pull from has a large effect on operator performance [17]. Previously, this decision was captured in a *pulling strategy* employed by the join operator implementation. However, in push-based systems, the execution is not driven by results, but by the input data. Join operators are only activated when input is actively being pushed from operators lower in the operator tree. Therefore, instead of pulling, we propose a *scheduling strategy* that determines which operators in a query plan are scheduled for execution. That is, we move the control over which input is processed from the join operator to the query engine, which orchestrates the query execution.

Algorithm 2 shows the `execute` method that takes a query Q and the number of results k as input and returns the top-k results. First, we obtain a query plan P from the `plan` method (line 1). We then use the scheduling strategy $\mathcal{S}$ to obtain the next operator that should be scheduled for execution (line 2). The scheduling strategy uses the current execution state as captured by the operators in the query plan to select the next operator. We then activate the selected operator (line 4). We select a new operator (line 5) until we either have obtained the desired number of k results or there is no operator to be activated, i.e., all inputs have been exhausted (line 3).

Algorithm 2. *execute*(Q, k)

```
  Input: Query $Q$, #results $k$
  Data: Query plan $P$, scheduling
        strategy $\mathcal{S}$
  Output: Query results $\Omega_G$
1 $P \leftarrow$plan($Q$);
2 $op \leftarrow \mathcal{S}$.nextOp($P$);
3 while $|\Omega_G| < k \wedge op \neq null$ do
4     $op$.activate();
5     $op \leftarrow \mathcal{S}$.nextOp($P$);
6 return $\Omega_G$
```

Algorithm 3. PBRJ.*activate*

```
  Data: Output queue $O$, threshold
        $\Gamma$, subsequent operator $out$
1 while $v(O$.peek()$) \geq \Gamma$ do
2     $r \leftarrow O$.dequeue();
3     $out$.push($r$);
```

Algorithm 3 shows the `activate` method (called by `execute`) for the rank join operator. Intuitively, the `activate` method triggers a "flush" of the operator's output buffer O. That is, all computed results having a score larger than or equal to the operator's threshold Γ (line 1) are reported to the subsequent operator (lines 2-3). An `activate` method for a *scan-sort* operator of a source d simply pushes all triples in d in a sorted fashion. Further, `activate` for selection and union operators causes them to push their outputs to a subsequent operator.

Now, the question remains *how* a scheduling strategy should select the next operator (`nextOp` method). We can apply the idea behind the state-of-the-art pulling strategy [17] to perform *corner-bound-adaptive* scheduling. Basically, we choose the input which leads to the highest reduction in the corner-bound:

Definition 6 (Corner-Bound-Adaptive Scheduling). *Given a rank join operator, we prefer the input that could produce join results with the highest scores. That is, we prefer input* 1 *if* $\Delta(\alpha_2, \beta_1) > \Delta(\alpha_1, \beta_2)$*, otherwise we prefer input* 2*. In case of ties, the input with the least current depth, or the input with the least index is preferred. The scheduling strategy then "recursively" selects and activates operators that may provide input elements for the preferred input. That is, in case the chosen input is another rank join operator, which has an empty output queue, the scheduling strategy selects and activates operators for its preferred input in the same manner.*

Example 3. Assume $k = 1$ and let $t_{i,j}$ denote the j^{th} triple in source i (e.g., $t_{1,2} = \langle$`ex:beatles`,`ex:album`,`ex:sgt_pepper`$\rangle$). First, our scheduling strategy prefers the input 1 and selects (via `nextOp`) and activates *scan-sort*(s_1), *sel*(q_1), and *union*(q_1). Note, also input 2 would have been a valid choice, as the threshold (respectively α, β) is not set yet. The rank join reads $t_{1,2}$ and $t_{1,3}$ as new inputs elements from *union*(q_1), and both elements are inserted into H_1 ($\alpha_1 = \beta_1 = 1$). The scheduler now prefers input 2 (as input 1 is exhausted) and selects and activates *scan-sort*(s_3), *sel*(q_2), and *union*(q_2), because source 3 has triples with higher scores than source 2. Now, *union*(q_2) pushes $t_{3,2}$ and α_2 respectively β_2 is set to $v(t_{3,2}) = 3$. Employing a summation as Δ, the threshold Γ is set to 4 (as $\max\{1 + 3, 1 + 3\} = 4$). Then, $t_{3,2}$ is inserted into H_2 and the joins between $t_{3,2}$ and elements in H_1 are attempted; $t_{1,3} \bowtie t_{3,2}$ yields a result μ, which is then inserted into the output queue. Finally, as $v(\mu) = 4 \geq \Gamma = 4$ is true, μ is reported as the top-1 result and the algorithm terminates. Note, not all inputs have been processed, i.e., source 2 has not been scanned (cf. Fig. 3).

3.3 Improving Threshold Estimation

We now present two modifications to the corner-bound bounding strategy that allow us to calculate a more precise (*tighter*) threshold $\widetilde{\Gamma}$, thereby achieving earlier result reporting and termination.

Star-Shaped Entity Query Bounds. A star-shaped entity query is a set of triple patterns Q_s that share a common variable at the subject position. We observed that in Linked Data query processing, every result to such a query is contained in one single source. This is because a result here is an entity, and information related to that entity comes exclusively from the one source representing that particular entity. Exploiting this knowledge, a more precise corner-bound for joins of a star-shaped query (part) can be calculated. Namely, we can derive that, in order to be relevant, sources for Q_s must satisfy all triple patterns in Q_s (because they must capture all information for the requested entities). Given relevant sources for Q_s are denoted as D and the source upper bound is given by $\upsilon_u(d)$ for $d \in D$, the upper bound score $\upsilon_u^Q(Q_s)$ for results matching Q_s can be derived based on the maximum source upper bound $\upsilon_u^{max}(D) = \max\{\upsilon_u(d)|d \in D\}$. More precisely, $\upsilon_u^Q(Q_s) = \Delta(\upsilon_u^Q(q_1), \ldots, \upsilon_u^Q(q_n)$ with $q_i \in Q_s$ and $\upsilon_u^Q(q_i) = \upsilon_u^{max}(D)$, because every triple that contributes to the result must be contained in a source $d \in D$, and thus, must have a score $\leq \upsilon_u^{max}(D)$.

Look-Ahead Bounds. The corner-bound strategy uses the last-seen scores β_i of input elements to calculate the current threshold. We observed that when an input element r_i is received by an operator on input i, the next input element r_i^{next} (and its score $\upsilon(r_i^{next})$) is often already available in the pushing operator. The next element is available because (1) scan-sort operators materialize their complete output before pushing to subsequent operators, (2) rank join operators maintain an output queue that often contains more than one result with scores greater than or equal to the current threshold Γ, and (3) given a source d_i has been pushed by a scan-sort operator, the source score upper bound of d_{i+1} (i.e., the next source to be loaded) is available. By using the score of the next instead of the last-seen input element, we can provide a more accurate threshold Γ, because we can estimate the maximal score of unseen elements from that particular input more accurately. If available, we therefore define $\tilde{\beta}_i = \upsilon(r_i^{next})$ as the score of the next input element. Otherwise, we use the last-seen score β_i, i.e., $\tilde{\beta}_i = \beta_i$ (see Fig. 3 (b)).

Threshold Calculation. By applying both strategies, we can now refine the bound as $\tilde{\Gamma} = \max\{\min\{\Delta(\alpha_1, \tilde{\beta}_2), \upsilon_u^Q(Q_s)\}, min\{\Delta(\alpha_2, \tilde{\beta}_1), \upsilon_u^Q(Q_s)\}\}$. The following theorem (see proof in our report [19]) allows to use $\tilde{\Gamma}$ for top-k processing:

Theorem 1. *$\tilde{\Gamma}$ is correct (i.e., there is no unseen result constituting to the final top-k results) and more precise than Γ (i.e., $\tilde{\Gamma} \leq \Gamma$ holds at all times).*

3.4 Early Pruning of Partial Results

Knowledge about sources can also be exploited to *prune partial results* from the output queues to reduce the cost of a join as well as the memory space needed to keep track of input elements in a join operator. The idea of pruning has been pursued by approximate top-k selection [18] approaches. However, we do not approximate, but only prune those partial results that are *guaranteed* not to be part of the final top-k results. Intuitively, we can prune a partial result, if its score together with the maximal possible score for the "unevaluated" query part, is smaller than the lowest of the k so far computed complete results. Note, the opportunity for pruning arises only when k (or more) complete results have been produced (by the root join operator).

More precisely, let Q be a query and $\mu(Q_f)$ a partial query result, with Q_f as "finished" part and Q_r as "remaining" part ($Q_f \subset Q$ and $Q_r = Q \setminus Q_f$). The upper bound on the scores of all final results based on $\mu(Q_f) \in \Omega_G(Q_f)$ can be obtained by aggregating the score of $\mu(Q_f)$ and the maximal score $\upsilon_u^Q(Q_r)$ of results $\mu(Q_r) \in \Omega_G(Q_r)$. $\upsilon_u(Q_r)$ can be computed as the aggregation of maximal source upper bounds obtained for every triple pattern in $Q_r = \{q_1, \dots, q_m\}$, i.e., $\upsilon_u^Q(Q_r) = \Delta(\upsilon_u^Q(q_1), \dots, \upsilon_u^Q(q_m))$, where $\upsilon_u^Q(q_i) = \max\{\upsilon_u(d) | d \in \mathit{source}(q_i)\}$. A tighter bound for $\upsilon_u^Q(Q_r)$ can be obtained, if Q_r contains one or more entity queries (see previous section) and aggregating their scores in a greedy fashion. Last, the following theorem can be established (see proof in [19]):

Theorem 2. *A result $\mu_f \in \Omega_G(Q_f)$ cannot be part of the top-k results for Q if $\Delta(\upsilon(\mu_f), \upsilon_u^Q(Q_r)) < \min\{\upsilon(\mu) | \mu \in \Omega_G^k(Q)\}$, where $\Omega_G^k(Q)$ are the currently known k results of Q.*

4 Experimental Evaluation

In the following, we present our evaluation and show that (1) top-k processing outperforms state-of-the-art Linked Data query processing, when producing only a number of top results, and (2) our tighter bounding and early pruning strategy outperform baseline rank join operators in the Linked Data setting.

Systems. In total, we implemented three different systems, all based on push-based join processing. For all queries, we generated left-deep query plans with random orders of join operators. All systems use the same plans and are different only in the implementation of the join operator.

First, we have the push-based symmetric hash join operator (*shj*) [10,11], which does not employ top-k processing techniques, but instead produces all results and then sorts them to obtain the requested top-k results. Also, there are two implementations of the rank join operator. Both use the corner-bound-adaptive scheduling strategy (which has been shown to be optimal in previous work [17]), but with different bounding strategies. The first uses the corner-bound (*rj-cc*) from previous work [17], while the second (*rj-tc*) employs our optimization with tighter bounds and early result pruning. The *shj* baseline is used to study the benefits of top-k processing in the Linked Data setting, while *rj-cc* is employed to analyze the effect of the proposed optimizations.

All systems were implemented in Java 6. Experiments were run on a Linux server with two Intel Xeon 2.80GHz Dual-Core CPUs, 8GB RAM and a Segate ST31000340AS 1TB hard disk. Before each query execution, all operating system caches were cleared. The presented values are averages collected over three runs.

Dataset and Queries. We use 8 queries from the Linked Data query set of the FedBench benchmark. Due to schema changes in DBpedia and time-outs observed during the experiments (> 2 min), three of the 11 FedBench queries were omitted. Additionally, we use 12 queries we created. In total, we have 20 queries that differ in the number of results they produce (from 1 to 10118) and in their complexity in terms of the number of triple patterns (from 2 to 5). A complete listing of our queries can be found in [19].

To obtain the dataset, we executed all queries directly over the Web of Linked Data using a link-traversal approach [6] and recorded all Linked Data sources that were retrieved during execution. In total, we downloaded 681,408 Linked Data sources, comprising a total of 1,867,485 triples. From this dataset we created a source index that is used by the query planner to obtain relevant sources for the given triple patterns.

Scores were randomly assigned to triples in the dataset. We applied three different score distributions: uniform, normal ($\mu = 5, \sigma^2 = 1$) and exponential ($\lambda = 1$). This allows us to abstract from a particular ranking and examine the applicability of top-k processing for different classes of functions. We used summation as the score aggregation function Δ.

We observed that network latency greatly varies between hosts and evaluation runs. In order to systematically study the effects of top-k processing, we thus decided to store the sources locally, and to simulate Linked Data query processing on a single machine (as done before [10,11]).

Parameters. Parameter $k \in \{1, 5, 10, 20\}$ denotes the number top-k results to be computed. Further, there are the three different score distributions $d \in \{u, n, e\}$ (uniform, normal and exponential, respectively).

Overall Results. Fig. 4a shows an overview of processing times for all queries ($k = 1, d = n$). We can see that for all queries the rank join approaches perform better or at least equal to the baseline *shj* operator. On average, the execution times for *rj-cc* and *rj-tc* were 23.13s and 20.32s, whereas for *shj* it was 43.05s. This represents an improvement in performance of the *rj-cc* and *rj-tc* operators over the *shj* operator by factors of 1.86 and 2.14, respectively.

The improved performance of the rank join operators is due to top-k processing, because these operators do not have to process all input data in order to produce the k top results, but can terminate early. On the other hand, the *shj* implementation produces all results. Fig. 4b shows the average number of retrieved sources for different values of k. We can see clearly that the rank join approaches retrieve fewer sources than the baseline approach. In fact, *rj-cc* and *rj-tc* retrieve and process only 41% and 34%, respectively, of the sources that the *shj* approach requires. This is a significant advantage in the Linked Data context, where sources can only be retrieved in their entirety.

However, we also see that the rank join operators sometimes do not perform better than *shj*. In these cases, the result is small (e.g., Q18 has only two results). The rank join operators have to read all inputs and compute all results in these cases. For example, for Q20 the rank join approaches retrieve and process all 35103 sources, just as the *shj* approach does.

Bounding Strategies. We now examine the effect of the bounding strategies on overall execution time. The average processing times mentioned earlier represent

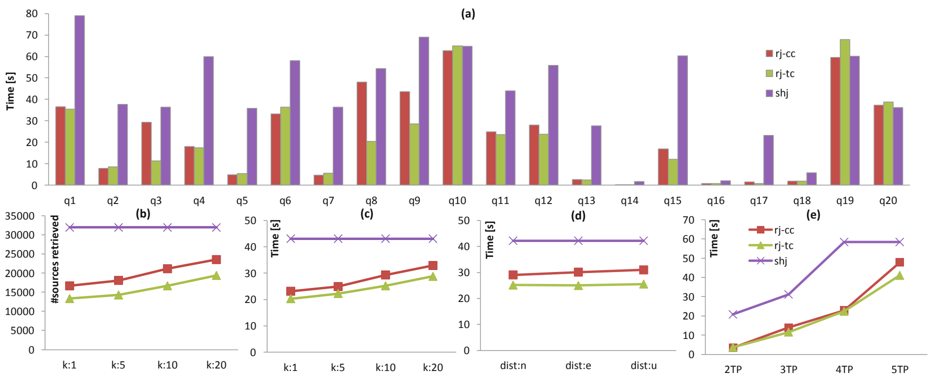

Fig. 4. (a) All queries with their evaluation times ($k = 1, d = n$). (b) Average number of sources over all queries (different k, $d = n$). (c) Average evaluation time over all queries (different k, $d = n$). (d) Average evaluation time over all queries (different score distributions, $k = 10$). (e) Average evaluation time over all queries with varying number of triple patterns ($k = 1, d = n$).

an improvement of 12% of *rj-tc* over *rj-cc*. For Q3, the improvement is even higher, where *rj-tc* takes 11s, compared to 30s for *rj-cc*.

The improved performance can be explained with the tighter, more precise bounding strategy of *rj-tc* compared to *rj-cc*. For example, our bounding strategy can take advantage of a large star-shaped subexpression with 3 patterns in Q3, leading to better performance because of better upper bound estimates. Moreover, we observed that the look-ahead strategy helps to calculate a much tighter upper bound especially when there are large score differences between successive elements from a particular input.

In both cases, a tighter (more precise) bound means that results can be reported earlier and less inputs have to be read. This is directly reflected in the number of sources that are processed by *rj-tc* and *rj-cc*, where on average, *rj-tc* requires 23% fewer sources than *rj-cc*. Note, while in Fig. 4a *rj-tc*'s performance often seems to be comparable to *rj-cc*, Fig. 4b makes the differences more clear in terms of the number of retrieved sources. For instance, both systems require an equal amount of processing times for Q17. However *rj-tc* retrieves 7% less sources. Such "small" savings did not show properly in our evaluation (as we retrieved sources locally), but would effect processing time in a real-world setting with network latency.

Concerning the outlier Q19, we noticed that *rj-tc* did read slightly more input (2%) than *rj-cc*. This behavior is due to our implementation: Sources are retrieved in parallel to join execution. In some cases, the join operators and the source retriever did not stop at the same time.

We conclude that *rj-tc* performs equally well or better than *rj-cc*. For some queries (i.e., entity queries and inputs with large score differences) we are able to achieve performance gains up to 60% compared to the *rj-cc* baseline.

Early Pruning. We observed that this strategy leads to lower buffer sizes (thus, less memory consumption). For instance with Q9, *rj-tc* could prune 8% of its buffered data. However, we also noticed that the number of sources loaded and scanned is actually the key factor. While pruning had positive effects, the improvement is small compared to what could be achieved with tighter bounds (for Q9 73% of total processing time was spent on loading and scanning sources).

Effect of Result Size k. Fig. 4c depicts the average query processing time for all three approaches at different k (with $d = n$). We observed that the time for *shj* is constant in k, as *shj* always computes all results, and that the rank join approaches outperform *shj* for all k. However, with increasing k, more inputs need to be processed. Thus, the runtime differences between the rank join approaches and *shj* operator become smaller. For instance, for $k = 1$ the average time saving over all queries is 46% (52%) for *rj-cc* (*rj-tc*), while it is only 31% (41%) for $k = 10$.

Further, we can see in Fig. 4c that *rj-tc* outperforms *rj-cc* over all values for k. The differences are due to our tighter bounding strategy, which substantially reduces the amount of required inputs. For instance, for $k = 10$, *rj-tc* requires 21% less inputs than *rj-cc* on average.

We see that *rj-tc* and *rj-cc* behave similarly for increasing k. Both operators become less efficient with increasing k (Fig. 4c).

Effect of Score Distributions. Fig. 4d shows average processing times for all approaches for the three score distributions. We see that the performance of both rank join operators varied only slightly w.r.t. different score distributions. For instance, *rj-cc* performed better by 7% on the normal distribution compared to the uniform distribution. The *shj* operator has constant evaluation times over all distributions.

Effect of Query Complexity. Fig. 4e shows average processing times (with $k = 1, d = n$) for different numbers of triple patterns. Overall, processing times increase for all systems with an increasing number of patterns. Again, we see that the rank join operators outperform *shj* for all query sizes. In particular, for 5 queries patterns, we noticed the effects of our entity bounds more clearly, as those queries often contained entity queries up to the length of 3.

5 Related Work

The top-k join problem has been addressed before, as discussed by a recent survey [8]. The J* rank join, based on the A* algorithm, was proposed in [14]. Other rank join algorithms, HRJN and HRJN*, were introduced in [7] and further extended in [12]. In contrast to previous works, we aim at the Linked Data context. As recent work [6,3,10,11] has shown, Linked Data query processing introduces various novel challenges. In particular, in contrast to the state-of-the-art *pull*-based rank join, we need a push-based execution for queries over Linked Data. We therefore adapt pull strategies to the push-based execution model (based on operator scheduling). Further, our work is different from prior work on Web-accessible databases [20], because we rely exclusively on simple HTTP lookups for data access, and use only basic statistics in the source index.

There are different bounding strategies: In [2,17], the authors introduced a new Feasible-Region (FR) bound for the general setting of n-ary joins and multiple score attributes. However, it has been proven that the PBRJ template is *instance-optimal* in the restricted setting of binary joins using corner-bound and a single score attribute [2,17]. We adapt the corner-bound to the Linked Data setting and provide tighter, more precise bounds that allow for earlier termination and better performance.

Similar to our pruning approach, [18] estimates the likelihood of partial results contributing to a final result (if the estimate is below a given threshold partial results are pruned). However, [18] addressed the *selection top-k problem*, which is different to our top-k join problem. More importantly, we do not rely on probabilistic estimates for pruning, but employ accurate upper bounds. Thus, we do not approximate final top-k results.

6 Conclusion

We discussed how existing top-k join techniques can be adapted to the Linked Data context. Moreover, we provide two optimizations: (1) tighter bounds estimation for early termination, and (2) aggressive result pruning. We show in real-world Linked Data experiments that top-k processing can substantially improve performance compared to the state-of-the-art baseline. Further performance gains could be observed using the proposed optimizations. In future work, we like to address different scheduling strategies as well as further Linked Data aspects like network latency or source availability.

References

1. Elbassuoni, S., Ramanath, M., Schenkel, R., Sydow, M., Weikum, G.: Language-model-based ranking for queries on RDF-graphs. In: CIKM, pp. 977–986 (2009)
2. Finger, J., Polyzotis, N.: Robust and efficient algorithms for rank join evaluation. In: SIGMOD, pp. 415–428 (2009)
3. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K., Umbrich, J.: Data summaries for on-demand queries over linked data. In: World Wide Web (2010)
4. Harth, A., Kinsella, S., Decker, S.: Using Naming Authority to Rank Data and Ontologies for Web Search. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 277–292. Springer, Heidelberg (2009)
5. Hartig, O.: Zero-Knowledge Query Planning for an Iterator Implementation of Link Traversal Based Query Execution. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 154–169. Springer, Heidelberg (2011)
6. Hartig, O., Bizer, C., Freytag, J.-C.: Executing SPARQL Queries over the Web of Linked Data. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 293–309. Springer, Heidelberg (2009)

7. Ilyas, I.F., Aref, W.G., Elmagarmid, A.K.: Supporting top-k join queries in relational databases. The VLDB Journal 13, 207–221 (2004)
8. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. 58, 11:1–11:58 (2008)
9. Klyne, G., Carroll, J.J., McBride, B.: Resource description framework (RDF): concepts and abstract syntax (2004)
10. Ladwig, G., Tran, T.: Linked Data Query Processing Strategies. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 453–469. Springer, Heidelberg (2010)
11. Ladwig, G., Tran, T.: SIHJoin: Querying Remote and Local Linked Data. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 139–153. Springer, Heidelberg (2011)
12. Li, C., Chang, K.C.-C., Ilyas, I.F., Song, S.: Ranksql: query algebra and optimization for relational top-k queries. In: SIGMOD, pp. 131–142 (2005)
13. Mamoulis, N., Yiu, M.L., Cheng, K.H., Cheung, D.W.: Efficient top-k aggregation of ranked inputs. ACM Trans. Database Syst. (2007)
14. Natsev, A., Chang, Y.-C., Smith, J.R., Li, C.-S., Vitter, J.S.: Supporting incremental join queries on ranked inputs. In: VLDB, pp. 281–290 (2001)
15. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2008)
16. Schmedding, F.: Incremental SPARQL evaluation for query answering on linked data. In: Workshop on Consuming Linked Data in Conjunction with ISWC (2011)
17. Schnaitter, K., Polyzotis, N.: Optimal algorithms for evaluating rank joins in database systems. ACM Trans. Database Syst. 35, 6:1–6:47 (2010)
18. Theobald, M., Weikum, G., Schenkel, R.: Top-k query evaluation with probabilistic guarantees. In: VLDB, pp. 648–659 (2004)
19. Wagner, A., Tran, D.T., Ladwig, G., Harth, A., Studer, R.: Top-k linked data query processing (2011), `http://www.aifb.kit.edu/web/Techreport3022`
20. Wu, M., Berti-Equille, L., Marian, A., Procopiuc, C.M., Srivastava, D.: Processing top-k join queries. In: VLDB, pp. 860–870 (2010)