Efficient Execution of Top-K SPARQL Queries

Sara Magliacane[1,2], Alessandro Bozzon[1], and Emanuele Della Valle[1]

[1] Politecnico of Milano, P.za L. Da Vinci, 32. I-20133 Milano - Italy
[2] VU University Amsterdam, De Boelelaan 1081a, The Netherlands

Abstract. Top-k queries, i.e. queries returning the top k results ordered by a user-defined scoring function, are an important category of queries. Order is an important property of data that can be exploited to speed up query processing. State-of-the-art SPARQL engines underuse order, and top-k queries are mostly managed with a *materialize-then-sort* processing scheme that computes all the matching solutions (e.g. thousands) even if only a limited number k (e.g. ten) are requested. The $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra is an extended SPARQL algebra that treats order as a first class citizen, enabling efficient *split-and-interleave* processing schemes that can be adopted to improve the performance of top-k SPARQL queries. In this paper we propose an incremental execution model for $\mathcal{S}$PARQL-$\mathcal{R}$ANK queries, we compare the performance of alternative physical operators, and we propose a rank-aware join algorithm optimized for native RDF stores. Experiments conducted with an open source implementation of a $\mathcal{S}$PARQL-$\mathcal{R}$ANK query engine based on ARQ show that the evaluation of top-k queries can be sped up by orders of magnitude.

1 Introduction

As the adoption of SPARQL as a query language for Web data increases, practitioners are showing a growing interest in top-k queries [5,6], i.e. queries returning the top k results ordered by a specified scoring function. Simple top-k queries can be expressed in SPARQL 1.0 by including the ORDER BY and LIMIT clauses, which impose an order on the result set and limit the number of results. SPARQL 1.1 additionally enables the specification of complex scoring functions through the use of projection expressions that can define a variable to be used in the ORDER BY clause. Listing 1.1 provides an example SPARQL 1.1 top-k query that will be used as a running example.

```
1  SELECT ?product ?offer (g1(?avgRat1) + g2(?avgRat2) + g3(?price1) AS ?score)
2  WHERE { ?product hasAvgRating1 ?avgRat1 .
3   ?product hasAvgRating2 ?avgRat2 .
4   ?product hasName ?name .
5   ?product hasOffers ?offer .
6   ?offer hasPrice ?price1 }
7  ORDER BY DESC(?score) LIMIT 10
```

Listing 1.1: A top-k SPARQL query that retrieves the best ten offers of products ordered by a function of user ratings and offer price; g_i are normalization functions and the bold letters represent the abbreviations used in the following examples.

In most of the algebraic representation of SPARQL, the algebraic operators that evaluate the ORDER BY and LIMIT clauses are result modifiers, i.e. operators that alter the sequence of solution mappings *after* the full evaluation of the graph pattern in the WHERE clause. For instance, the query in Listing 1.1 is executed according to the plan in Fig. 1.a: solutions matching the WHERE clause are drawn from the storage until the whole result set is materialized; then, the project expression *?score* is evaluated by the EXTEND operator on each solution and used to order the result set; finally the top 10 results are returned.

This *materialize-then-sort* scheme can hinder the performance of SPARQL top-k queries, as a SPARQL engine might process thousands of matching solutions, compute the score for each of them and order the result set, even if only a limited number (e.g. ten) were requested. Moreover, the ranking criteria can be expensive to compute and, therefore, should be evaluated only when needed and on the minimum possible number of mappings. It has been shown [9] that query plans where the scoring function is evaluated in an incremental, non-blocking manner, access a lower number of mappings, thus yielding better performance. $\mathcal{S}$PARQL-$\mathcal{R}$ANK [1] is a rank-aware algebra for SPARQL that has been designed to address such a need, as it enables the definition of query plans as shown in Fig. 1.b, where the evaluation of the scoring function is split and delegated to rank-aware operators (the ranking operator ρ and the rank-join $\bowtie$) that are interleaved with other operators and incrementally order the mappings extracted from the data store.

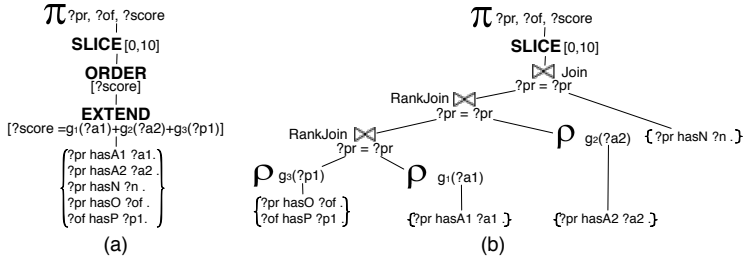


Fig. 1: The (a) standard and (b) $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebraic query plan for the top-k SPARQL query in Listing 1.1.

Contribution. In this paper, we propose an incremental execution model for top-k queries based on the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra. We show how existing work on rank-aware physical operators can be adapted to enable the incremental execution of top-k SPARQL queries over RDF data stores. The focus of our work is on top-k SPARQL query evaluation at the *query engine level*, thus abstracting from the underlying data storage layer. Nonetheless, we show how the availability of dedicated storage data structures for sorted (retrieving mappings sorted by their score) or random access (retrieving mappings matching a join attribute value) can speed-up the execution of top-k queries. While random access is available in both in RDBMS and native RDF storage systems (although with better performance in the latter), sorted access is typically unavailable in RDF stores. However, we show how the presence of sorted access can boost the performance of top-k SPARQL queries, and we elaborate on strategies to obtain it. Based on

the results of a comparative analysis of state-of-the-art physical operators, we propose a rank-aware join algorithm (namely RSEQ) optimized for native RDF stores. We also describe our implementation experience based on ARQ 2.8.9[3] (namely ARQ-$\mathcal{R}$ANK). We provide experimental evidence that the incremental execution model described in the paper speeds up the execution of top-k queries in ARQ-$\mathcal{R}$ANK by orders of magnitude w.r.t. the original ARQ implementation.

Organization of the paper. In Section 2, we discuss the related work. Section 3 reviews the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra proposed in [1]. Section 4 introduces a set of incremental rank-aware physical operators, proposes a rank-join algorithm optimized for native RDF stores, discusses sorted access in RDF stores, and presents three rank-aware optimization techniques. Section 5 focuses on the evaluation based on an extended version of BSBM [3] and reports on the experiments with ARQ-$\mathcal{R}$ANK. Section 6 presents conclusions and future work.

2 Related Work

SPARQL query optimization is a consolidated field of research. Existing approaches focus on algebraic [11,13] or selectivity-based optimizations [14]. Despite an increasing need from practitioners [5,6], few works address top-k query optimization in SPARQL. In state-of-the-art query engines, a basic top-k optimization was introduced in ARQ 2.8.9 [5], but the ORDER BY and LIMIT clauses are still evaluated after the completion of the other operations and on the complete set of solutions. OWLIM and Virtuoso[4] have some basic ranking features that precompute the rank of most popular nodes in the RDF graph based on the graph link density. These ranks are used to order query results by popularity, not based on a user-specified scoring function. Moreover, the query execution is not incremental, even if Virtuoso uses an anytime algorithm.

Our work builds on the results of several well-established techniques for the efficient evaluation of top-k queries in relational databases such as [9,7,8,18,15] where efficient rank-aware operators are investigated, and [4] where a rank-aware relational algebra and the RankSQL DBMS are described. In particular, we considered the algorithms described in [7,8], while building on the discussions about data access types described in [15]. The application of such results to SPARQL is not straightforward, as SPARQL and relational algebra have equivalent expressive power, while just a subset of relational optimizations can be ported to SPARQL [13]. Moreover, relational rank-aware operators require dedicated data structures for sorted access, while they often do not assume to have data structures for random access. In contrast, sorted access is usually not available in native triplestores (as it is rare to have application-specific indexes), while random access is common, as required for the efficient evaluation of schema-free data such as RDF. In a previous work [1], we presented the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra, and applied it to the execution of top-k SPARQL queries on top of virtual RDF

[3] Among many alternatives, we chose Jena ARQ, because of its neat design and the fact that it recently tackled the problem of top-k optimization [5].

[4] OWLIM: http://bit.ly/N9ZRG3 Virtuoso: http://bit.ly/SfRWKm

stores through query rewriting over a rank-aware RDBMS. In this paper, we discuss and evaluate the execution of top-k SPARQL queries also in native RDF stores, offering an extensive discussion on rank-aware operators, optimization techniques, and their application to different data access configurations.

Several works extend the standard SPARQL algebra to allow the definition of ranking predicates [10,20]. AnQL [17] is an extension of the SPARQL language and algebra able to address a wide variety of queries (including top-k ones) over annotated RDF graphs; our approach, on the other hand, requires no annotations, and can be applied to any state-of-the-art SPARQL engine.

Straccia [19] describes an ontology mediated top-k information retrieval system over relational databases, where user queries are rewritten into a set of conjunctive queries, which are translated in SQL queries and executed on a rank-aware RDBMS [4]; the obtained results are merged into the final top-k answers. Another rank-join algorithm, the Horizon based Ranked Join, is introduced [21] and aims at optimizing twig queries on weighted data graphs. In this case, results are ranked based on the underlying cost model, not based on an ad-hoc scoring function as in our work. The SemRank system [12] uses a rank-join algorithm to calculate the top-k most relevant paths from all the paths that connect two resources specified in the query. However, the application context of this algorithm is different from the one we present, because it targets paths and ranks them by relevance using IR metrics, and the focus is not on query performance optimization.

Wagner et al. [2] introduce PBRJ, a top-k join algorithm for federated queries over Linked Data. The proposed processing model is push-based, i.e. operators push mappings to subsequent operators, and, given a set or pre-defined ranking criteria, assumes a-priori knowledge of the upper-bounds and lower-bounds for the scores contained in each data source. By leveraging these bounds it is possible to decide at execution-time which data source to query. The complete content of the involved sources is materialized locally and, if sorted access is not available, it is fully ordered. Due to domain-specific assumptions, PBRJ is able to define a better estimation of unseen join results than standard rank-join algorithms. This work is complementary to our approach, as our work bases on the traditional pull-based processing model, in which operators pull mappings from their input operators; we consider RDF data stored locally either in native or virtual RDF stores, and we make no assumption (e.g., upper- or lower-bound for scores and data access types) on the evaluated data sources. Data can be queried by user-defined ranking criteria, and efficient data access methods like random access (an important optimization factor in rank-join algorithms) can be exploited to further improve the query performance. We are currently investigating a hybrid processing model to address the federated query scenario, combining the strengths of both push and pull-based processing models.

3 Background

To support the following discussion, we review the existing formalization of SPARQL in [11] and our $\mathcal{S}$PARQL-$\mathcal{R}$ANK [1] algebra.

3.1 Basic SPARQL Definitions

In SPARQL, the WHERE clause contains a **graph pattern** that can be constructed using the OPTIONAL, UNION, FILTER and JOIN operators. Given three sets I, L and V (IRIs, literals and variables), a tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a triple pattern. A **Basic Graph Pattern** (BGP) is a set of triple patterns connected by the JOIN operator.

The semantics of SPARQL is based on the notion of **mapping**, defined in [11] as a partial function $\mu : V \rightarrow (I \cup L \cup B)$, where B is the set of blank nodes. The domain of μ, denoted by *dom(μ)*, is the subset of V where μ is defined. Let P be a graph pattern, *var(P)* denotes the set of variables occurring in P. Given a triple pattern t and a mapping μ such that $var(t) \subseteq dom(\mu)$, $\mu(t)$ is the triple obtained by replacing the variables in t according to μ.

Using these definitions, it is possible [11][13] to define the semantics of SPARQL queries with an algebra having a set of operators – Selection (σ), Join ($\bowtie$), Union ($\cup$), Difference($\setminus$) and Left Join ($\bowtie$)– operating on **sets of mappings** denoted with Ω. The evaluation of a SPARQL query is based on its translation into an algebraic tree composed of those algebraic operators.

3.2 The $\mathcal{S}$PARQL-$\mathcal{R}$ANK Algebra

$\mathcal{S}$PARQL-$\mathcal{R}$ANK is a rank-aware framework for top-k queries in SPARQL, based on the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra, an extension of the SPARQL algebra that supports ranking as a first-class construct. The central concept of the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra is the *ranked set of mappings*, an extension of the standard SPARQL definition of a set of mappings that embeds the notion of ranking.

$\mathcal{S}$PARQL-$\mathcal{R}$ANK supports top-k queries in SPARQL with an ORDER BY clause that can be formulated as a scoring function combining several ranking criteria. Given a graph pattern P, a **ranking criterion** $b(?x_1, \ldots, ?x_m)$ is a function defined over a set of m variables $?x_j \in var(P)$. The evaluation of a ranking criterion on a mapping μ, indicated by $b[\mu]$, is the substitution of all of the variables $?x_j$ with the corresponding values from the mapping. A criterion b can be the result of the evaluation of any built-in function of query variables that ensures that $b[\mu]$ is a numerical value. We define as max_b the application-specific maximal possible value for the ranking criterion b.

A **scoring function** on P is an expression $\mathcal{F}(b_1, \ldots, b_n)$ defined over the set B of n ranking criteria. The evaluation of a scoring function $\mathcal{F}$ on a mapping μ, indicated by $\mathcal{F}[\mu]$, is the value of the function when all of the $b_i[\mu]$, where $\forall i = 1, \ldots, n$, are evaluated. As typical in ranking queries, the scoring function $\mathcal{F}$ is assumed to be **monotonic**, i.e., a $\mathcal{F}$ for which holds $\mathcal{F}(b_1[\mu_1], \ldots, b_n[\mu_1]) \geq \mathcal{F}(b_1[\mu_2], \ldots, b_n[\mu_2])$ when $\forall i : b_i[\mu_1] \geq b_i[\mu_2]$.

In order to evaluate the scoring function, all the variables in $var(P)$ that contribute in the evaluation of $\mathcal{F}$ must be bound. Since OPTIONAL and UNION clauses can introduce unbound variables, we assume all the variables in $var(P)$ to be *certain variables*, as defined in [13][5], i.e. variables that are certainly bound

[5] This can be ensured by an efficiently verifiable syntactical condition.

for every mapping produced by P. An extension of $\mathcal{S}$PARQL-$\mathcal{R}$ANK towards the relaxation of the *certain variables* constraint is part of the future work. Listing 1.1 provides an example of the scoring function $\mathcal{F}$ calculated over the ranking criteria g_1*(?avgRat1)*, g_2*(?avgRat2)*, and g_3*(?price1)*.

A key property of $\mathcal{S}$PARQL-$\mathcal{R}$ANK is the ability to retrieve the first k results of a top-k query before scanning the complete set of mappings resulting from the evaluation of the graph pattern. To enable such a property, the mappings progressively produced by each operator should flow in an order consistent with the final order, i.e., the order imposed by $\mathcal{F}$. When the evaluation of a SPARQL top-k query starts on the graph pattern the resulting mappings are unordered. As soon as a subset $\mathcal{B} \subseteq B$, s.t. $\mathcal{B} = \{b_1, \ldots, b_j\}$ (with $j \leq |B|$) of the ranking criteria can be computed (i.e., when $var(b_k) \subseteq dom(\mu)\ \forall k = 1, \ldots, j$), an order can be imposed to a set of mappings Ω by evaluating for each $\mu \in \Omega$ the ***upper-bound*** of $\mathcal{F}[\mu]$ as:

$$\overline{\mathcal{F}}_{\mathcal{B}}(b_1, \ldots, b_n)[\mu] = \mathcal{F}\begin{pmatrix} b_i = b_i[\mu] & \text{if } b_i \in \mathcal{B} \\ b_i = max_{b_i} & \text{otherwise} \end{pmatrix} \forall i$$

Note that if $\mathcal{B} = B$, then $\overline{\mathcal{F}}_{\mathcal{B}}[\mu] = \mathcal{F}_B[\mu]$. Therefore, it is clear that for any subset of ranking criteria $\mathcal{B}$, the value of $\overline{\mathcal{F}}_{\mathcal{B}}[\mu]$ is the upper-bound of the score that μ can obtain, when $\mathcal{F}_B[\mu]$ is completely evaluated, by assuming that all the ranking criteria still to evaluate will return their maximal possible value.

A **ranked set of mappings** $\Omega_{\mathcal{B}}$ w.r.t. a scoring function $\mathcal{F}$ and a set $\mathcal{B}$ of ranking criteria, is the set of mappings Ω augmented with an order relation $<_{\Omega_{\mathcal{B}}}$ defined over Ω, which orders mappings by their upper-bound scores, i.e., $\forall\ \mu_1,\ \mu_2 \in \Omega:\ \mu_1 <_{\Omega_{\mathcal{B}}} \mu_2 \iff \overline{\mathcal{F}}_{\mathcal{B}}[\mu_1] < \overline{\mathcal{F}}_{\mathcal{B}}[\mu_2]$. A set of mappings on which no ranking criteria is evaluated ($\mathcal{B} = \emptyset$) is consistently denoted as $\Omega_{\emptyset}$ or Ω.

The monotonicity of $\mathcal{F}$ implies that $\overline{\mathcal{F}}_{\mathcal{B}}$ is always an upper-bound of $\mathcal{F}$, i.e. $\overline{\mathcal{F}}_{\mathcal{B}}[\mu] \geq \mathcal{F}[\mu]$ for any mapping $\mu \in \Omega_{\mathcal{B}}$, thus guaranteeing that the order imposed by $\overline{\mathcal{F}}_{\mathcal{B}}$ is consistent with the order imposed by $\mathcal{F}$.

Algebraic Operators. The $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra introduces a new *rank operator* ρ, representing the evaluation of a single ranking criterion, and redefines the Selection (σ), Join ($\bowtie$), Union ($\cup$), Difference($\setminus$) and Left Join ($\ltimes\!\!\!\bowtie$) operators, enabling them to process and output ranked sets of mappings. For the sake of brevity, we present ρ and $\bowtie$, referring the reader to [1] for further details.

The *rank operator* ρ_b evaluates the ranking criterion $b \in B$ upon a ranked set of mappings $\Omega_{\mathcal{B}}$ and returns $\Omega_{\mathcal{B}\cup\{b\}}$, i.e. the same set ordered by $\overline{\mathcal{F}}_{\mathcal{B}\cup\{b\}}$. Thus, by definition $\rho_b(\Omega_{\mathcal{B}}) = \Omega_{\mathcal{B}\cup\{b\}}$.

The extended $\bowtie$ operator has a standard semantics for the membership property [11], while it defines an order relation on its output mappings: given two ranked sets of mappings Ω'_{B_1} and Ω''_{B_2} ordered with respect to two sets of ranking criteria B_1 and B_2, the join $\Omega'_{B_1} \bowtie \Omega''_{B_2}$ produces a ranked set of mappings ordered by $\overline{\mathcal{F}}_{B_1\cup B_2}$. Formally $\Omega'_{B_1} \bowtie \Omega''_{B_2} \equiv (\Omega' \bowtie \Omega'')_{B_1\cup B_2}$.

Algebraic Equivalences. Query optimization relies on algebraic equivalences to produce several equivalent formulations of a query. The $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra defines a set of algebraic equivalences that take into account the order property. The rank operator ρ can be pushed-down to impose an order to

a set of mappings; such order can be then exploited to limit the number of mappings flowing through the physical execution plan, while allowing the production of the k results. In the following, we focus on the equivalences that apply to the ρ and $\bowtie$ operators (see [1] for the complete set of equivalences):

1. **Rank splitting** $[\Omega_{\{b_1,b_2,...,b_n\}} \equiv \rho_{b_1}(\rho_{b_2}(...(\rho_{b_n}(\Omega))...))]$: allows splitting the criteria of a scoring function into a series of rank operations $(\rho_{b_1}, ..., \rho_{b_n})$, thus enabling the individual processing of the ranking criteria.
2. **Rank commutative law** $[\rho_{b_1}(\rho_{b_2}(\Omega_B)) \equiv \rho_{b_2}(\rho_{b_1}(\Omega_B))]$: allows the commutativity of the ρ operand with itself, thus enabling query planning strategies that exploit optimal ordering of rank operators.
3. **Pushing** ρ **over** $\bowtie$ [if $\Omega^{''}$ does not contain any variable of the ranking criterion b, then $\rho_b(\Omega^{'}_{B_1} \bowtie \Omega^{''}_{B_2}) \equiv \rho_b(\Omega^{'}_{B_1}) \bowtie \Omega^{''}_{B_2}$; if both $\Omega^{'}$ and $\Omega^{''}$ contain all variables of b, then $\rho_b(\Omega^{'}_{B_1} \bowtie \Omega^{''}_{B_2}) \equiv \rho_b(\Omega^{'}_{B_1}) \bowtie \rho_b(\Omega^{''}_{B_2})$]: this law handles swapping $\bowtie$ with ρ, thus allowing to push the rank operator only on the operands whose variables also appear in b.

The new algebraic laws lay the foundation for query optimization, as discussed in the following section.

4 Execution of Top-K SPARQL Queries

In common SPARQL query engines (e.g. Jena ARQ), a query execution plan is a tree of physical operators designed according to a pull-based processing model. During execution, mappings are extracted iteratively from the root operator, which, in turn, will draw from the child operators only the intermediate mappings needed to produce the output. The same applies for the child operators, recursively up to the evaluation of Basic Graph Patterns (BGPs) in the storage layer. The execution is incremental unless some blocking operator appears in the execution plan (e.g. the ORDER BY operator, which materializes all the intermediate results to order them).

The $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra briefly presented in Section 3.2 enables an execution model in which the blocking ordering operator can be split in several non-blocking rank operators. Using the algebraic equivalences, it becomes possible to push these rank operators inside the execution tree and evaluate the order for each ranking criterion incrementally. The final order of the results, i.e. the order of the scoring function, is ensured by the other rank-aware operators.

In this section, we describe the $\mathcal{S}$PARQL-$\mathcal{R}$ANK incremental execution model and the related physical operators; then, we report on our investigations on a rank-aware optimizer that leverages the algebraic equivalences of Section 3.2.

4.1 Incremental Rank-aware Physical Operators

The $\mathcal{S}$PARQL-$\mathcal{R}$ANK execution model creates rank-aware query plans, i.e. trees of physical operators that incrementally output ranked sets of mappings according to their *upper-bounds*. The execution stops as soon as the requested number

of mappings has been drawn from the root operator. A rank-aware execution model calls for rank-aware physical operators. Some of them are of trivial nature; for instance, the *selection* operator only filters solutions that do not satisfy the FILTER clause, thus guaranteeing the preservation of mappings order. Other operators, e.g. ρ, $\bowtie$, $\rtimes$ and $\cup$, require more complex algorithms.

Rank. The rank operator ρ can exploit rank aggregation algorithms. This class of algorithms orders, in an incremental manner, lists of objects by combining several partial scores (e.g. the score for each ranking criterion) into one final score (e.g.the scoring function). MPro [7] is a state-of-the-art rank aggregation algorithm, which requires sorted access on one of the ranking criteria to be fully effective. To fit the $\mathcal{S}$PARQL-$\mathcal{R}$ANK execution model we adopted MPro as a ρ_b operator that maintains a priority queue containing all the mappings drawn from its input. Within the queue these mappings are ordered by $\overline{\mathcal{F}}_{\mathcal{B}\cup\{b\}}$, where $\mathcal{B}$ is the set of already evaluated criteria on the input set of mappings, while b is the ranking criteria to be evaluated by ρ_b. The operator ρ_b cannot output immediately each drawn mapping, since one of the next mappings could obtain a higher score after evaluation. Instead, it outputs the top ranked mapping of the priority queue μ only when it draws from its input a mapping μ' such that $\overline{\mathcal{F}}_{\mathcal{B}\cup\{b\}}[\mu] \geq \overline{\mathcal{F}}_{B}[\mu']$.

Join. Depending on the ordering and access patterns of its inputs, a rank-aware $\bowtie$ operator can be implemented with several physical operators. In the simplest case when only the left input is a ranked set of mapping, a standard *index join* algorithm can be adopted, which maintains the order of the output mappings according to the one of the left operand. The presence of several indexes in native triplestores (e.g. *s*, *p*, *o*, *spo*, *pos*, *osp*) guarantees fast random access to triples, thus enabling an optimized $n-way$ joining strategy called *streaming index join*[6], which performs lookups on the indexes by substituting already bound variables from previous inputs.

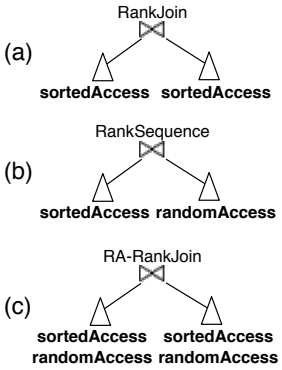


Fig. 2: Rank-join algorithms

Other input configurations require a rank-join algorithm. The idea of rank-join algorithms is to combine ranking and joining, by ordering progressively the results during the join operation. This can be achieved by taking advantage of the individual orders of its inputs to update, after each extraction from the inputs, an upper-bound of the scores of all join combinations not yet seen. A join result is returned only if it has a combined score greater than or equal to the upper-bound, thus ensuring that no other combination could possibly achieve a better score. In this work we consider three algorithms for rank-join (in Fig. 2(a-c)), each one characterized by a different configuration of access patterns on their left and right inputs.

HRJN [8] (and its variant HRJN*) is an example of rank-aware symmetrical hash join that has been shown to provide good performance. The basic HRJN and HRJN* operators assume *sorted access* (retrieving mappings sorted by their

[6] The strategy is named *Sequence* in Jena ARQ, see http://bit.ly/O8e3Rm

score) on both inputs. If the inputs offer *random access* (retrieving mappings matching a join attribute value), then some optimizations can be achieved. We will refer to the generalized version of HRJN endowed with both sorted and random access on both inputs as "RA-HRJN".

Given that efficient sorted access in not commonly available in native triple stores (Section 4.2 provides a detailed discussion on the topic), to exploit available random access mechanisms we propose **RankSequence** (RSEQ), a characterization of the HRJN rank-join template that requires minimum sorted access and leverages random access to improve performances. RSEQ is designed for supporting a configuration where one input provides only sorted access (S), and the other one supports only random access (R). To the best of our knowledge, no previous work on rank-join algorithms addresses these assumptions on data access[7]. The RSEQ algorithm supports a pull-based query model, for which the `getNext` method is presented in Figure 3. The algorithm requires the maintenance of a priority queue Q, which contains all the seen join combinations ordered by a scoring function f. In a rank-aware query plan, f is $\overline{\mathcal{F}}_{B_1 \cup B_2}$, where B_1 and B_2 are the sets of ranking criteria on the input ranked sets of mappings.

Algorithm 1 The RSEQ getNext method

```
S: sorted access input
R: random access input
R_top: maximal possible value for R
loop
  if Q is not empty then
    topScore ← Q.topScore();
    if topScore ≥ Threshold then
      return Q.topMapping();
    end if
  end if
  μ_S ← S.getNext();
  probe R with μ_S;
  for each retrieved join combination
    insert the join combination in Q;
  end for
  Threshold ← f(μ_S, R_top);
end loop
```

Fig. 3: getNext method for *RSEQ*

The algorithm extracts a mapping μ_S from S and it probes R in order to get all corresponding join combinations, inserting them into the priority queue. Then it updates a threshold, which is the upper-bound of the scores for the not yet seen combinations. The mapping with the highest score in the priority queue is output only if it has a score greater or equal to the threshold.

LeftJoin. As all the variables in a $\mathcal{S}$PARQL-$\mathcal{R}$ANK query are assumed to be *certain variables*, all $\bowtie$ operators can have a ranked set of mapping only as left input. Thus, standard *index left-join* algorithms can be adopted, as they will output mappings in the same order of evaluation.

4.2 Sorted access in triplestores

The selection of the most suited rank-join operator is conditioned by the availability of sorted access or random access for its inputs. While random access on triples is the basic operation in native triplestores, sorted access is not typically available at storage level. The ρ operators could be adopted to provide an initial order (according to one ranking criterion) to mappings extracted from the storage layer; however, such a solution is inherently inefficient, as it requires to process all the matching mappings. In the following, we show two cases in which

[7] Similar assumptions are made by Upper [16], which is a *rank aggregation* algorithm, i.e. it is designed for the ρ operator.

we exploit or extend the design of the native triplestores to provide an efficient sorted access on triples and BGPs.

Exploiting existing indexes. Given a triple pattern of the form *(?s p ?o)*, in which p is a given predicate, if a ranking criterion is defined as the variable $?o$ that assumes literal values, then the triplestore native indexes can be exploited. Since triplestores usually provide POS B+ trees, where objects are ordered lexicographically, triples are already stored ordered by the variable value and, therefore, are extracted in the right order. Note that lexicographical order for numerical values means they will be retrieved in ascending order. The same effect can be obtained by reordering the set of triple patterns that compose a BGP so to position the triple pattern involved in the ranking criterion as first.

Creating custom indexes. If a ranking criterion involves the evaluation of an arbitrary function of variable values, or if the literal values should be ordered in descending order, then the native POS index cannot be exploited. Therefore, sorted access can be provided at storage level only by creating custom indexes, which store the evaluations of triples and/or BGPs in the order enforced by one or more ranking criterion. POS indexes can be still exploited by materializing the values of the arbitrary ranking criterion as attributes in the dataset.

4.3 Rank-aware Optimization Techniques

The goal of query optimization is the selection of an efficient execution plan for a given query. Many optimization techniques exist for SPARQL[8], but the addition of the *ranking* logical property brings novel optimization dimensions.

Several works on top-k query processing in RDBMSs propose optimization techniques that attempt to provide a (sub) optimal scheduling of rank [7] or rank-join operators [8] via dataset sampling or ranking selectivity estimation. An optimizer using both operators is presented in [4], where the cost of the generated plans is estimated by executing the plan on a sample of the dataset.

Unfortunately, previous works on top-k query planning in RDBMS cannot be directly ported to SPARQL engines, as data in an RDF storage can be "schema-free"; moreover, in some systems, it is possible to push the evaluations of BGPs down to the storage system, which can be optimized w.r.t. to a standard join by several techniques, like selectivity estimation optimization in native triplestores or SQL rewriting in the case of virtual RDF stores.

The design of a query planner for top-k queries demands for a detailed evaluation of the possible rank-aware configurations that might arise, a topic that we leave to future work. In this paper, we focus on the *rank* and *rank-join* operators and we discuss their application according to several optimization dimensions. Although simple, all of the proposed strategies proved effective in creating plans that considerably reduce the execution time of top-k queries and, therefore, could be easily adopted for heuristic-based query plan selection.

Ranking of BGP strategy. The Ranking of BGP (`ROB`) strategy is a naïve planning technique that only uses rank operators. The evaluation of the BGP is

[8] Refer to [13] for a comprehensive review

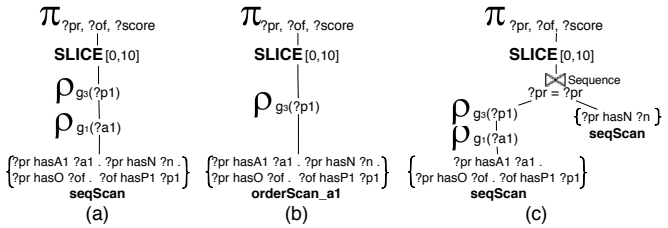


Fig. 4: Three examples of plans for the query in Listing 1.1: (a) `ROB` strategy with sequential access, (b) `ROB` strategy with sorted access on $a1$ (c) `INTER` strategy.

pushed to the storage system, from which the mappings are fetched incrementally through several pipelined rank operators, one per each ranking criterion. The strategy requires little planning overhead, as it exploits only the algebraic transformations on rank operators.

Let P be the graph pattern of a query with a scoring function composed by n ranking criteria, the initial query plan is in the form: $\rho_{\{b_1,b_2,\ldots,b_n\}}(\Omega^P_\emptyset)$, where $\Omega^P_\emptyset$ is the unordered set of mapping resulting from evaluation of P. The rank-aware query plan, produced by the `ROB` strategy applying the splitting law for ρ, is in the form: $\rho_{b_1}(\rho_{b_2}(\ldots(\rho_{b_n}(\Omega^P_\emptyset))))$.

Fig. 4 depicts two example plans for the case study query, where the BGP evaluation is respectively performed with sequential access (Fig. 4.a) and sorted access on the variable $a1$ (Fig. 4.b). If the data source already provides *sorted access* according to one ranking criterion b_i, then the planner can reorder the ρ operators using the rank commutative law to have b_i evaluated first. Since the retrieved set of mappings is not $\Omega^P_\emptyset$, but $\Omega^P_{b_i}$, it can remove the corresponding ρ_{b_i} operator from the plan.

Interleaved Strategy. The `ROB` strategy can be extended to include an additional planning phase that splits the triple patterns containing variables of the ranking criteria ("ranked triple patterns") from the others. This approach is called Interleaved strategy (`INTER`), because it interleaves rank-aware with non-rank-aware operators.

By splitting a set of mappings $\Omega^P_\emptyset$ into two joined sets of mappings $\Omega^{P'}_\emptyset \bowtie \Omega^{P''}_\emptyset$, where P' is the graph pattern containing all the ranked triple patterns and P'' is the pattern containing the rest, we can apply the algebraic law that pushes ρ over $\bowtie$. Applying this law to a `ROB` plan of the form:

$$\rho_{b_1}(\rho_{b_2}(\ldots(\rho_{b_n}(\Omega^{P'}_\emptyset \bowtie \Omega^{P''}_\emptyset))))$$

we obtain a plan of the form:

$$\rho_{b_1}(\rho_{b_2}(\ldots(\rho_{b_n}(\Omega^{P'}_\emptyset)))) \bowtie \Omega^{P''}_\emptyset$$

where the $\bowtie$ operator needs to preserve only the order of the first operand. Thus it is possible to use a streaming index join between the two sets of mappings.

The splitting strategy can be redefined to adopt the standard planning heuristic of avoiding Cartesian joins: if there is no shared variable between two ranked

triple patterns, the strategy must include into the ranked BGP also the "bridge" triple patterns that have a shared variable with each of them (or chains of triple patterns). Fig. 4.c provides an example of application of the `INTER` strategy on the running case. Since the ranked triple patterns *(?pr hasA1 ?a1)* and *(?of hasA1 ?p1)* have no shared variable, we include *(?of hasP1 ?pr)* as a bridge triple pattern inside the ranked sub-plan. The triple pattern *(?pr hasN ?n)* does not influence the final order, so it can be safely put in the non-ranked sub-plan.

Rank-join strategy. The Rank-join strategy (`RJ`) involves the use of one or more rank-join operators in a query plan. This strategy is a variant of the `INTER` strategy, in which the n ranked triple patterns of the query graph pattern P are split each into one BGP P^i (plus the "bridge triple patterns" to avoid Cartesian joins) for $i = 1..n$ and there is a graph pattern P^{n+1} containing the rest of P.

Following this rule, `RJ` first splits a given set of mappings $\Omega_\emptyset^P$ into $n+1$ joined sets of mappings $\Omega_\emptyset^{P^1} \bowtie \cdots \bowtie \Omega_\emptyset^{P^{n+1}}$, where n is the number of ranking criteria. Then, `RJ` can apply the law pushing ρ over $\bowtie$ to obtain a plan of the form:

$$(((\rho_{b_1}(\Omega_\emptyset^{P^1}) \bowtie \rho_{b_2}(\Omega_\emptyset^{P^2})) \bowtie \cdots \bowtie \rho_{b_n}(\Omega_\emptyset^{P^n})) \bowtie \Omega_\emptyset^{P^{n+1}})$$

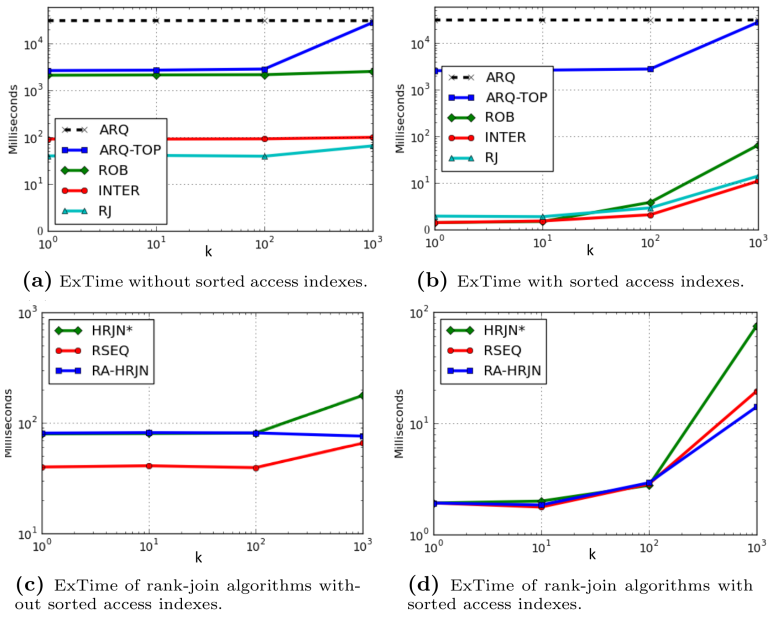
where all the $\bowtie$ operators, except the rightmost, are rank-joins as they operate on ranked sub-plans; the rightmost $\bowtie$ operator needs to preserve only the order of the first operand, thus an index join can be used. If the datastore already provides a *sorted access* according to b_i to $\Omega_\emptyset^{P^i}$, then the retrieved set of mappings is already a ranked set of mappings $\Omega_{b_i}^P$, and the planner can remove the corresponding ρ_{b_i} operator from the plan. Fig. 1.b shows the plan generated by `RJ` on the query in Listing 1.1.

`RJ` builds a left-linear tree of operators and selects each rank-join operator using a simple heuristic based on the availability of sorted or random access to ranked sets of mappings. This heuristic assumes that using more random access leads to better performances; the assumption holds if random access has a low cost (as in native triple stores), because random access allows tightening the upper-bound of the score. `RJ` applies: 1) RA-HRJN, when the left input is a ranked mapping $\Omega_{b_i}^P$, in which b_i is a single ranking criterion and P is a triple pattern or a BGP; 2) RSEQ, when the left input is Ω_B^P, in which B is a set of ranking criteria with $|B| \geq 2$, e.g. if the input is another rank-join operator, or if $|B| = 1$ and P is not a a triple pattern or BGP. A more complex strategy could be devised by taking into account also the cost of the sorted access on the right input, but this is left to future work.

The `RJ` strategy also exploits an optimization of the random access on a BGP as, at run-time, it creates a reordered version of the original BGP as follows: first, the triple pattern containing the joining attribute is evaluated; then a random access on the other triples is performed iteratively.

5 Evaluation

This section presents an evaluation of ARQ-$\mathcal{R}$ANK, a prototype implementation of a $\mathcal{S}$PARQL-$\mathcal{R}$ANK query engine that extends Jena ARQ 2.8.9. The ARQ-



(a) ExTime without sorted access indexes.

(b) ExTime with sorted access indexes.

(c) ExTime of rank-join algorithms without sorted access indexes.

(d) ExTime of rank-join algorithms with sorted access indexes.

Fig. 5: Performance evaluation for the optimization techniques of Section 4.3.

$\mathcal{R}$ANK source code, the test datasets and the queries used in the experiments are available for download at http://sparqlrank.search-computing.org/.

Our experiments are based on a modified version of the Berlin SPARQL Benchmark (BSBM [3]). The BSBM dataset generator has been modified to create additional attributes, generated as aggregates from the existing data (e.g. the average of the ratings for a given product) or according to different probability distributions. We report on the performance of ARQ-$\mathcal{R}$ANK using the *SumDepth* and *wall clock execution time* metrics. The SumDepth metric provides a system-independent measure of the I/O cost of a query, as it sums the total number of processed mappings. The experiments were conducted on an AMD 64bit processor with 2.66 GHz and 4 GB main, a Debian distribution with kernel 2.6.26-2, and Sun Java 1.6.0 with 2 GB maximum heap size for the JVM. The reported execution times are the average of 20 tests, measured after 5 warm ups, removing the outlier values according to the three sigma rule. The experiments were executed using Jena TDB 0.8.11 as a native triple storage.

5.1 Rank-aware optimization strategies evaluation

In the following we evaluate the performance of the `ROB`, `INTER` and `RJ` strategies w.r.t. to the non-optimized Jena ARQ 2.8.8 (`ARQ`) and the TopK optimization [5] introduced in Jena ARQ 2.8.9 (`ARQ-TOP`) on a five million triples dataset.

Fig. 5 reports the results of experiments performed on the query in Listing 1.1, where ranking criteria follow a *normal* ($\mu = 0.5, \sigma = 0.15$) score distribution, at varying values of k (we consider also $k = 1000$, while in real world

applications k is typically less than 100). The reported numbers represent the complete evaluation time of the query, without considering the planning time.

Fig. 5.a depicts the results obtained when no sorted access indexes are available in the data storage. This setting represents the worst-case scenario for the usage of ARQ-$\mathcal{R}$ANK, as rank (e.g. MPro) and rank-join (e.g. HRJN) algorithms are proven to be efficient when sorted access is available for at least one operand. Despite the additional overhead required to provide an initial ordering to the mappings fetched from the data storage, all the techniques show a performance increase (from 0.2x to 10x), a result that is mainly accountable to the reduced number of mappings that flow in the query engine, as shown in Fig. 6.

In Fig. 5.b we report the results of the query when sorted access indexes are available. In this configuration, the number of mappings extracted from the data storage significantly decreases (as shown in Fig. 6 SumDepth ranges from 10-1000 in `RJ`, as opposed to 10^5 in `ARQ-TOP`) as results are already ordered according to one of the ranking criteria, thus exploiting at full potential the features of the adopted rank and rank-join operators: all the proposed strategies consistently outperform `ARQ-TOP` up to three orders of magnitude.

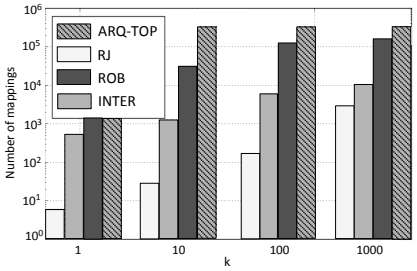


Fig. 6: SumDepth with sorted access indexes

In Fig. 5.c-d we report the results of a comparison analysis for the rank-join operators HRJN*, RA-HRJN, and the proposed RSEQ algorithm. The natural availability of efficient random access to the underlying triple store provides great benefits in terms of accessed mappings and required execution time. Note that the RSEQ and RA-HRJN operators get more effective as k increases. When no sorted access indexes are available, as shown in Fig. 5.c, RSEQ is faster than RA-HRJN, because it requires sorted access only on the left input, while RA-HRJN needs sorted access on both inputs. Fig. 5.d presents the case in which sorted access indexes are available. In this case, RSEQ is comparable to RA-HRJN.

Although the selected case study query may account for the reported numbers, the overall (and consistent) performance increase w.r.t. `ARQ-TOP` provides strong evidence about the general applicability of the proposed approach, also when no sorted access indexes are available. The selection of the best configurations depends on several factors, such as the costs of random and sorted access (which differ based on the storage system), distribution of scores, and others. A detailed study on SPARQL top-k query planning is left to future work.

5.2 Query benchmark evaluation

As far as we know, currently there is no benchmark for top-k queries SPARQL. Therefore, we created a small benchmark of 8 queries[9] having 2 and 3 ranking criteria. Although previous works [18] show that the cost associated with top-k

[9] Queries are available at http://sparqlrank.search-computing.org/

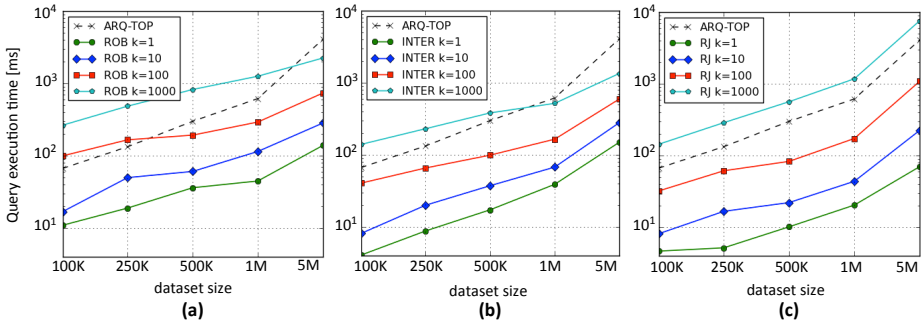


Fig. 7: Performance Evaluation of ARQ-$\mathcal{R}$ANKon the benchmark query mix using the a) `ROB`, b) `INTER`, and c) `RJ` optimization strategies

algorithms is mainly related to the selected k value, to provide asymptotic evidence of the utility of ARQ-$\mathcal{R}$ANK, we performed the evaluation on five datasets of increasing size (from 100K triples to 5M triples). Fig. 7 depicts the average execution times (for each considered dataset) of the query mix in `ARQ-TOP` and using the `ROB`, `INTER`, and the `RJ` strategies.

When $1 \leq k \leq 100$, typical values in top-k queries, ARQ-$\mathcal{R}$ANK consistently outperforms `ARQ-TOP` (up to two orders of magnitude), with a gain proportional to the dataset size. The `RJ` optimization strategy yields considerably better performance for $1 \leq k \leq 10$, while the `INTER` strategy is the best for $k = 100$. Note that the `ROB` strategy provides significant performance gain, while requiring a negligible planning effort.

On the other hand, when $k = 1000$ and the dataset size is below 1M triples, the two systems show comparable performance; nonetheless, the `ROB` and `INTER` strategies show a performance gain w.r.t. `ARQ-TOP` for the biggest dataset in the experiment, and the curves suggest an improvement also for bigger datasets.

6 Conclusion and Future Work

In this paper, we addressed the problem of efficiently executing top-k SPARQL queries; we introduced an incremental execution model for the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra and we presented ARQ-$\mathcal{R}$ANK, a rank-aware SPARQL query engine that builds on the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra and exploits state-of-the-art rank-aware query operators. We proposed RSEQ, a rank-join algorithm specifically designed to exploit fast random access in native triple stores, and we analyzed the behavior of the system under several configurations. Our results show that ARQ-$\mathcal{R}$ANK is consistently able to significantly improve the performance of top-k queries. Although our evaluation did not include other SPARQL engine implementations, the proposed execution model and planning strategies are applicable to other pull-based SPARQL query engine implementations.

The results presented in this paper are part of a broader research work, where we plan to study more advanced, cost-based, optimization techniques that es-

timate the cardinality of intermediate results in multiple pipelined rank-join operator evaluations. Among our goals there is also a study on efficient materialization of RDF views for sorted access, and the effects of the relaxation of the *certain variables* constraint defined in Section 3, as the introduction of potentially unbound variables brings uncertainty in the score evaluation. Finally, we outlook extensions of $\mathcal{S}$PARQL-$\mathcal{R}$ANK w.r.t. SPARQL 1.1 federation extension.

Acknowledgements. This research is supported by the Search Computing project, funded by ERC, under the IDEAS Advanced Grants program.

References

1. A. Bozzon et al. Towards and efficient SPARQL top-k query execution in virtual RDF stores. In *DBRANK workshop in VLDB '11*, 2011.
2. A. Wagner et al. Top-k Linked Data Query Processing. In *ESWC '12*. Springer, 2012.
3. C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2), 2009.
4. C. Li et al. RankSQL: query algebra and optimization for relational top-k queries. In *SIGMOD '05*. ACM, 2005.
5. P. Castagna. Avoid a total sort for order by + limit queries. JENA bug tracker. https://issues.apache.org/jira/browse/jena-89.
6. E. Della Valle et al. Order matters! harnessing a world of orderings for reasoning over massive data. *Semantic Web Journal*, 2012.
7. S.-w. Hwang and K. Chang. Probe minimization by schedule optimization: Supporting top-k queries with expensive predicates. *IEEE TKDE*, 19(5), 2007.
8. I. F. Ilyas et al. Rank-aware Query Optimization. In *SIGMOD '04*. ACM, 2004.
9. I. F. Ilyas et al. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
10. J. Cheng et al. f-SPARQL: a flexible extension of SPARQL. In *DEXA '10*. ACM, 2010.
11. J. Pérez et al. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.
12. K. Anyanwu et al. SemRank: ranking complex relationship search results on the semantic web. In *WWW '05*. ACM, 2005.
13. M. Schmidt et al. Foundations of SPARQL query optimization. In *ICDT '10*. ACM, 2010.
14. M. Stocker et al. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW '08*. ACM, 2008.
15. D. Martinenghi and M. Tagliasacchi. Cost-Aware Rank Join with Random and Sorted Access. *IEEE TKDE*, 2011.
16. N. Bruno et al. Evaluating Top-k Queries over Web-Accessible Databases. In *ICDE '02*. IEEE, 2002.
17. N. Lopes et al. AnQL : SPARQLing up annotated RDFS. In *ISWC '10*. Springer, 2010.
18. K. Schnaitter and N. Polyzotis. Optimal algorithms for evaluating rank joins in database systems. *ACM Transactions on Database Systems*, 35(1), 2010.
19. U. Straccia. SoftFacts: A top-k retrieval engine for ontology mediated access to relational databases. In *SMC '10*. IEEE, 2010.
20. W. Siberski et al. Querying the semantic web with preferences. In *ISWC '06*. Springer, 2006.
21. Y. Qi et al. Sum-Max Monotonic Ranked Joins for Evaluating Top-K Twig Queries on Weighted Data Graphs. In *VLDB '07*, 2007.