Cost-aware rank join with random and sorted access

Davide Martinenghi and Marco Tagliasacchi

Abstract—In this paper we address the problem of joining ranked results produced by two or more services on the Web. We consider services endowed with two kinds of access that are often available: *i)* sorted access, which returns tuples sorted by score; *ii)* random access, which returns tuples matching a given join attribute value. Rank join operators combine objects of two or more relations and output the k combinations with the highest aggregate score. While the past literature has studied suitable bounding schemes for this setting, in this paper we focus on the definition of a pulling strategy, which determines the order of invocation of the joined services. We propose the *CARS* (Cost-Aware with Random and Sorted access) pulling strategy, which is derived at compile-time and is oblivious of the query-dependent score distributions. We cast *CARS* as the solution of an optimization problem based on a small set of parameters characterizing the joined services. We validate the proposed strategy with experiments on both real and synthetic data sets. We show that *CARS* outperforms prior proposals and that its overall access cost is always within a very short margin from that of an oracle-based optimal strategy. In addition, *CARS* is shown to be robust w.r.t. the uncertainty that may characterize the estimated parameters.

Index Terms—Top-k, rank join, sorted access, random access.

1 Introduction

Domain-specific search engines are becoming widespread on the Web. Their focus on a single domain gives them greater capabilities and credibility over general-purpose search engines, but also limits their query answering spectrum to narrow queries over specific domain knowledge. The new trend in search-oriented research [1] aims to overcome such a limitation, by allowing the co-operation and orchestration of search services in order to answer complex, multi-domain queries. Figure 1 shows an example multi-domain query written in an SQL-like language that retrieves the top 5 hotels and French restaurants in Paris located in the same street giving priority to number of stars and user rating. Addressing such queries, called *rank join* queries, requires to *i)* extract the answers of domain-specific search systems, *ii)* join them to form combinations, and *iii)* globally rank each combination by means of an aggregation function, so as to return the answers with the top aggregate scores first.

```
SELECT H.HotelName, R.RestName, R.Street
FROM HotelsInParis H, RestaurantsInParis R
WHERE H.Street = R.Street AND R.Cuisine='French'
RANK BY H.Stars * R.Rating
LIMIT 5
```

Fig. 1. A rank join query (running example)

• *The authors are with Politecnico di Milano, Dipartimento di Elettronica e Informazione, piazza Leonardo da Vinci 32, I-20133 Milano, Italy.*
E-mail: first.last@polimi.it

As customary in top-k query processing, we consider services that may return a large number of tuples, all equipped with a score. The tuples are typically presented in pages, and accessing such pages is costly. Moreover, accesses can occur according to various methods. These can be broadly classified as either *i) sorted*, producing a potentially large list of tuples ranked by score (such that the list's tail is typically not retrieved by the join method), or *ii) random*, producing a narrower set of objects, normally not ranked, which satisfy a selection over the attributes.

In this context, we adopt a model that takes into account the different ways of accessing the services, each characterized by its access cost. Then, we introduce an algorithm that generates at compile-time a cost-aware strategy to access services in order to quickly produce the combinations with the highest aggregate scores.

The rank join problem has been dealt with in the literature (e.g., [2]) by extending rank aggregation algorithms [3] to the case of join in the setting of relational databases. Rank join operators are often ([4]) characterized by *i)* a *bounding scheme* defining an upper bound on the score of unseen combinations, which grants correct termination with the top-k combinations when the score of the current k-th best combination exceeds the upper bound; *ii)* a *pulling strategy* deciding at each step the next service (relation) to be accessed. The well-known HRJN and HRJN* rank join operators [2] adopt a simple, yet efficient, bounding scheme that requires keeping track of the scores of the first and last tuples retrieved from each service.

When only sorted access is available, the optimal pulling strategy (adopted by HRJN*) consists of accessing, at each step, the service with the highest potential

to lower the upper bound. Note that, in this scenario, access costs do not affect the pulling strategy, as the latter depends only on the scores of the retrieved objects.

We remark that random access is often available. This happens, e.g., when the same service is endowed with several input/output signatures representing different access patterns, as is the case with many services on the Web. Also, random access might be provided by a different service than the one providing sorted access; this gives more flexibility over the relational setting, where random access requires the construction of explicit indexes. In addition, random access is naturally available in several contexts, e.g., in similarity join [5]. Here, sorted access takes place by exploring input relations in order of distance from a query object; random access simply means to reuse the same access pattern as sorted access but, instead of a query object, using an arbitrary object in another relation, in order to retrieve the similar objects. Note that the availability of random access enables a much faster termination since *i)* a large number of combinations can be formed after each random access; *ii)* the bounding scheme computes an upper bound that is lower than with sorted access only.

The rank-join problem with both sorted and random access has been mentioned in [2], where a suitable bounding scheme is proposed. The introduction of random access also affects the pulling strategy. This has been recognized in [6], where a cost-aware pulling strategy is envisaged at query execution time. However, this approach only covers the problem of rank aggregation, i.e., a very restricted case of rank join, where all services have the same number of objects, and each of them appears only once. As such, it cannot be readily extended to the more general scenario addressed here. While several other rank aggregation algorithms take both kinds of access into account [3], [7], [6], [8], [9], [10], [11], [12], [13], the recent literature on rank join has mostly focused on cost-aware operators with sorted access only. Considering both kinds of access is relevant since, as pointed out, random access is often available and leads to considerable speed-up of rank-join operators.

To our knowledge, no author has addressed pulling strategies for rank join operators endowed with both sorted and random access. Several works exist that consider both sorted and random access in the context of rank aggregation, and adaptations of these to the rank join scenario are possible; yet, as will be discussed in Sections 3 and 6, some of the main benefits of these methods do not readily carry over to rank join. The design of an optimal pulling strategy that takes into account both access costs and score distributions needs to face practical issues. The main difficulties arise when modeling the score distribution of unseen tuples since: *i)* the score distribution might depend on the query; *ii)* the score of tuples retrieved by means of random access cannot be predicted based on the scores of the seen tuples.

Therefore, in this paper we present a pulling strategy for binary rank join operators that takes into account only the service access costs and rough characterizations of value distributions that are easy to estimate and yet allow us to obtain near-optimal and robust performance.

Contributions. We define the *CARS* (Cost-Aware with Random and Sorted access) compile-time pulling strategy for rank join operators with both sorted and random access. Our model explicitly captures the different access costs and is oblivious of the score distributions. We determine the pulling strategy by solving an optimization problem that is formulated in terms of just a few parameters synthesizing the main aspects of the services (e.g., access costs, join selectivity, number of distinct join attribute values). These parameters might be available at service registration, or estimated using state-of-the-art techniques [14].

We evaluate the performance of *CARS* using both real and synthetic data sets. We observe that *CARS* is very close, in terms of overall access cost, to the optimal strategy computed by an oracle-based algorithm. We compare our solution with the main strategies available at the state of the art: *i)* the round-robin strategy of HRJN, oblivious of costs and score distribution; *ii)* the strategy of HRJN*, oblivious of both costs, which determines at runtime the next move based on the scores of the retrieved tuples; *iii)* a cost-aware strategy that considers only sorted access costs. We show that *CARS* significantly outperforms the other strategies under all the tested experimental conditions. The gains are larger when services are characterized by heterogeneous access costs.

In the main body of the paper we focus on the join of exactly two services and do not consider that services can be accessed in parallel; we remove such restrictions in Appendix A, available as supplemental online material.

2 The Search Computing Model

In this section, we introduce the setting of *search computing* [1], in which we formulate the rank join problem.

We consider the context of multi-domain queries posed on services over information sources available on the Web, in which the results extracted from different services need to be joined and ranked. We are therefore particularly interested in what we call *search* services, i.e., services that answer a request by returning a list of tuples in ranking order, according to some measure of relevance. We assume such a measure to be presented to the user along with the results as a score (usually in $\mathbf{R}$ or in the $[0, 1]$ interval).

We consider queries that join two search services, s_1 and s_2, and assign, to each pair (henceforth called *combination*) of tuples τ_1 from s_1 and τ_2 from s_2, an *aggregate score* according to a given *aggregation function* of the single scores of τ_1 and τ_2. W.l.o.g., we consider natural joins and assume each service s_i to have a signature of the form $s_i(\boldsymbol{A}_i, \boldsymbol{B}, S_i)$, where $\boldsymbol{A}_i$ is a set

of attributes specific for s_i, $\boldsymbol{B}$ is a set of join attributes, and S_i is an attribute carrying the score.

Let R_i be the extension of service s_i, i.e., the relational set of all tuples that can be returned by s_i. Let $\tau[A]$ indicate, as usual, the value of tuple τ for attribute A. Any tuple $\tau \in R_1 \bowtie R_2$ adorned with an extra attribute carrying the value $f(\tau[S_1], \tau[S_2])$, where f is an aggregation function, is called a *combination* from the join between s_1 and s_2. As customary, we deal with monotone aggregation functions, which are best suited for efficient rank join processing.

Example 2.1: The query described in Figure 1 can be formulated over services with the following signatures:

- `hotel(HotelName, Street, Stars)` as s_1, listing name, street, and stars for hotels in Paris ranked by stars.
- `restaurant(RestName, Street, Rating)` as s_2, listing name, street, and rating for restaurants in Paris serving French cuisine, ranked by customers' rating.

Both are easily obtained via suitable interfaces to existing services on the Web, e.g., www.venere.com (restricted to Paris) for hotels and www.eatinparis.com (selecting French cuisine) for restaurants. Services s_1 and s_2 can be joined, e.g., on the `Street` attribute in order to obtain hotel-restaurant combinations in the same street of Paris. Each combination will comprise all the attributes of `hotel` and `restaurant`, plus an extra attribute for the aggregate score that is evaluated via a monotone aggregation function combining stars and customers' rating. A possible example of such a function is one that takes the product of the scores. ■

Unlike relational databases, Web sources typically expose a limited number of access interfaces, in which certain fields must be mandatorily filled in in order to obtain a result. These fields may be the input fields of a form on a data-intensive Web site or the input parameters of a Web Service invocation. We model this by associating each service with given *access patterns* (i.e., input/output signatures), corresponding to the different ways in which it can be invoked.

We assume that each service can be invoked according to the following access patterns, where an O superscript indicates an output field, while I indicates an input field:

- *Sorted access*: $s_i(\boldsymbol{A}_i^{\mathrm{O}}, \boldsymbol{B}^{\mathrm{O}}, S_i^{\mathrm{O}})$, where all the fields are given as output and no input is required. The first sorted access returns the first page of tuples sorted by score in descending order, the next access returns the following page, etc.
- *Random access*: $s_i(\boldsymbol{A}_i^{\mathrm{O}}, \boldsymbol{B}^{\mathrm{I}}, S_i^{\mathrm{O}})$, where a tuple of values for the join attributes $\boldsymbol{B}$ is given as input, and tuples of values for the other attributes are returned.

The main parameters characterizing services are discussed next and summarized in Figure 2. These are either given (e.g., at service registration time) or can be easily estimated by sampling [14].

Cardinality and value distribution. Each service s_i has an associated *cardinality* $N_i = |R_i|$, expressing the total number of results that it can produce. Another important aspect is the distribution of values for the join attributes $\boldsymbol{B}$ with respect to the values for the service-specific attributes $\boldsymbol{A}_i$. Together with N_i, we consider two parameters that give a first characterization of this aspect:

- The number J_i of distinct tuples of values for the attributes $\boldsymbol{B}$ covered by service s_i. For instance, J_2 indicates the number of distinct streets covered by `restaurant` in our running example.
- The average number Q_i of times the same tuple of values for the join attributes $\boldsymbol{B}$ occurs in service s_i. For instance, Q_2 indicates the average number of restaurants (serving French cuisine) per street among those covered by `restaurant`. Clearly, $Q_i = N_i / J_i$.

Also, when considering a join between two services s_1 and s_2, it is relevant to consider to what extent the set of distinct tuples for the join attributes in s_1 are covered by s_2 and vice versa. To this end, we indicate with $J_{1,2}$ the number of distinct tuples of values for the join attributes that are common to s_1 and s_2 (the number of distinct streets in common between `hotel` and `restaurant`, in our example). Within the Search Computing framework [1], all the mentioned parameters (or approximations thereof) are specified when the services are registered on the system. In different frameworks, estimates can be obtained via sampling techniques as described in [14].

Pagination. Search services generally return results in pages of a fixed size, as the number of results may be large. Thus, we characterize each service s_i with a *page size* $\mathcal{P}_i$.

Costs. We associate each request sent to a service with an expected cost of invoking it. Such cost may differ for different services, and also depends on the access pattern used for invoking the service. Since we distinguish two kinds of accesses, for each service s_i, we shall in the following refer to its *sorted access cost* sc_i and its *random access cost* rc_i. These costs may, e.g., correspond to the average service response time, which is a relevant cost indicator both in the case of data sources distributed over a network and in the case of services performing heavy computations. In Section 4, a cost function will be defined in terms of sc_1, sc_2, rc_1, rc_2. With that, we determine the expected overall cost of retrieving the tuples needed to compute the join between s_1 and s_2, according to a strategy that indicates the accesses to be made.

Based on the parameters summarized in Figure 2, we propose a suitable *cost-aware* pulling strategy in Section 4.

3 Rank join operators

Consider two services s_1 and s_2, with signatures of the form $s_i(\boldsymbol{A}_i, \boldsymbol{B}, S_i)$. Let R_1 and R_2 denote their relational extension. We indicate with $\tau_i^{(p_i)}$ the p_i-th tuple extracted

k:	number of top combinations to be found.
sc_i:	cost of a sorted access to s_i (in cost units per tuple).
rc_i:	cost of a random access to s_i (in cost units per access).
N_i:	number of tuples in the extension of s_i.
J_i:	number of distinct tuples of values for the join attributes $\boldsymbol{B}$ of s_i.
$J_{1,2}$:	number of distinct tuples of values for the join attributes $\boldsymbol{B}$ in common between s_1 and s_2.
Q_i:	average number N_i/J_i of times the same tuple of values for the join attributes $\boldsymbol{B}$ occurs in s_i.
$\mathcal{P}_i$:	page size, i.e., number of tuples retrieved in a single sorted access from s_i.

Fig. 2. Summary of notation

from R_i in decreasing order of S_i and with $P_i \subseteq R_i$ the ordered relation containing the tuples already extracted from R_i with sorted access. The number of such tuples, $n_i = |P_i|$ is called *depth*. Let $\tau = \tau_1 \times \tau_2 \in R_1 \bowtie R_2$ denote a combination, whose aggregate score is

$$\mathcal{S}(\tau) = f(\tau[S_1], \tau[S_2]) \tag{1}$$

Definition 3.1: The *rank join* problem is a 4-tuple $(R_1, R_2, \mathcal{S}, k)$ such that *i)* $\mathcal{S}$ is an aggregation function defined as in (1); *ii)* the two relations R_1 and R_2 are accessed with sorted access, in decreasing order of S, and random access, based on an input join attribute value $b \in \boldsymbol{B}$; *iii)* $1 \leq k \leq |R_1 \bowtie R_2|$. A *solution* is an ordered relation O containing the top k combinations from $R_1 \bowtie R_2$ ordered by $\mathcal{S}$ (ties are resolved using a tie-breaking criterion).

Algorithm 1: $RankJoin(R_1, R_2, \mathcal{S}, k)$

```
   Input  : relations R_1 and R_2; function S as
            in (1); result size k
   Output: k combinations with highest aggregate
            score
   Data   : buffers P_1, P_2, RB_1, RB_2, and O
 1 begin
 2   u = ∞;  P_1 = P_2 = RB_1 = RB_2 = O = ∅;
 3   while |O| < k or min_{ω∈O} S(ω) < u do
 4     i = PS.chooseInput();  ℓ = 3 − i;
 5     τ_i = next unseen tuple of R_i;
 6     R_sorted = τ_i ⋈ (P_ℓ ∪ RB_ℓ);
 7     if random access is available ∧ no random
       access was made on R_ℓ with τ_i[B] then
 8       R_random = τ_i ⋈ σ_{B=τ_i[B]} R_ℓ;
 9       Add σ_{B=τ_i[B]} R_ℓ to RB_ℓ;
10     end
11     Add each member of R_sorted and R_random
       to O, retaining only the top k
       combinations;
12     Add τ_i to P_i;
13     u = BS.updateBound();
14   end
15   return O;
16 end
```

An implementation of rank join in line with [2] is illustrated in Algorithm 1. The function $updateBound$ of a given *bounding scheme* BS computes the upper bound u of the aggregate score of the unseen combinations. The unseen combinations are those that can be formed with at least an unseen tuple $\tau_i \in R_i - P_i$. The algorithm halts, returning the correct top-k combinations, if the aggregate score of the k-th combination in the output buffer exceeds or equals u. Note that line 5 corresponds to making a sorted access to s_i, while line 8 is a random access to s_ℓ (where $\ell = 1$ if $i = 2$, else $\ell = 2$). A random access to s_ℓ (if available) is always made after a sorted access to s_i with the tuple $\tau_i[\boldsymbol{B}]$ just extracted, provided that $\tau_i[\boldsymbol{B}]$ is new. The $updateBound$ function depends on the availability of random access. Indeed, when only sorted access is available, HRJN and HRJN* [2] compute u as

$$u = \max\{u_1, u_2\}, \quad \text{with}$$
$$u_1 = f(\tau_1^{(n_1)}[S_1], \tau_2^{(1)}[S_2]), \quad u_2 = f(\tau_1^{(1)}[S_1], \tau_2^{(n_2)}[S_2]) \tag{2}$$

However, when random access is available, we have

$$u = f(\tau_1^{(n_1)}[S_1], \tau_2^{(n_2)}[S_2]) \tag{3}$$

The function $chooseInput$ of a given *pulling strategy* PS decides at each step the next relation to be accessed. The HRJN* rank join operator described in [2] defines a pulling strategy that is optimal when only sorted access is available. At each step, HRJN* performs a sorted access to R_1 (R_2) if $u_1 > u_2$ ($u_1 < u_2$). In case of ties, HRJN* accesses the relation with the least depth. Note that, if only sorted access is available and there are no ties in the aggregate scores, the optimal pulling strategy does not depend on the service access costs[1].

Conversely, when random access is also available, the optimal pulling strategy necessarily depends on both access costs. In the past literature, a specific pulling strategy has not been defined for rank join operators endowed with both sorted and random access. Yet, the pulling strategies of HRJN and HRJN* can also be applied in this case, although sub-optimally.

We conclude this section by showing that state-of-the-art solutions originally conceived for rank aggregation lose some of their benefits when adapted to the case of rank join. Fagin's "combined algorithm" [3] provides a solution that takes into account both sorted and random access. There, the bounding scheme is based on a threshold computed as in (3), whereas the alternation of sorted and random accesses is driven by the cost of random access relative to sorted access. Specifically, a random access phase is triggered after $\lfloor rc/sc \rfloor$ sorted accesses. Then, random accesses are performed only when they might contribute to finding top-k objects. To this end, a lower bound is maintained for each object seen in at

1. Indeed, let $\mathcal{S}^k$ denote the aggregate score of the k-th combination in the ordered output buffer, upon termination. The rank join operator needs to achieve depths n_1^k, n_2^k such that both $u_1 \leq \mathcal{S}^k$ and $u_2 \leq \mathcal{S}^k$. Such depths do not depend on the costs.

least one service. In the case of two services, such a lower bound coincides with the aggregate score, if the object has been seen in both services. Otherwise, it is obtained by aggregating the seen score with the minimum score achievable in the other service. For each object seen in only one service, an upper bound is computed by aggregating the score seen in that service with the score of the last object retrieved by sorted access in the other service. If the upper bound is greater than the k-th largest lower bound, the corresponding random access is performed.

The described idea behind Fagin's combined algorithm cannot be directly extended to the case of rank join, since all random accesses need to be made anyway. A lower bound can be defined only for combinations formed as join results, and necessarily coincides with their aggregate scores. Indeed, a tuple in one service might not contribute to any join result. Hence, its lower bound is undefined. When determining the random accesses to a service, the upper bounds of the tuples retrieved by sorted access from the other service need to be considered. The smallest among those upper bounds is that of the last retrieved object, which coincides with the threshold in (3). Since the bounding scheme has not determined the termination of the algorithm, such an upper bound (and, consequently, all other upper bounds) is greater than or equal to the aggregate score of the k-th seen combination, i.e., the k-th largest lower bound. Hence, all random accesses need to be made.

In the next section we propose a suitable pulling strategy for the problem at hand that is shown to perform very closely to an oracle-based optimal strategy.

4 A COST-AWARE STRATEGY

We introduce *CARS* (Cost-Aware with Random and Sorted access), a pulling strategy defined at compile-time that takes into account access costs. *CARS* can be devised based on just a few parameters, shown in Figure 2, that characterize the joined services. The pulling strategy is obtained by solving an optimization problem that seeks to minimize the cost incurred by Algorithm 1 to find a target number of top combinations. Formally, we solve the following problem

$$\begin{array}{ll} \text{minimize} & \bar{C}(n_1, n_2) \\ \text{subject to} & \bar{\mathcal{K}}(n_1, n_2) \geq \mathcal{K}_T, \text{ with } n_i \in \mathbf{N} \cap [0, N_i] \end{array} \quad (4)$$

where $\bar{\mathcal{K}}(n_1, n_2)$ denotes the expected number of combinations that can be formed by retrieving n_i tuples from service s_i by means of sorted accesses *only*, $\bar{C}(n_1, n_2)$ is the associated expected cost of performing *both* sorted and random accesses, i.e., those needed to find the top k combinations according to Algorithm 1, and $\mathcal{K}_T$ is a target number of combinations. Both expectations are taken with respect to all the possible ways of composing the tuples returned by the services. Note that problem (4) does not constrain the number of *top combinations* directly, but rather the expected number $\bar{\mathcal{K}}$ of combinations formable using sorted access only. These may be considered *good combinations* since, being retrieved by sorted access, they have high scores both for the s_1 and the s_2 sub-tuples. The rationale behind this choice is that *i)* this optimization does not require any information about the score distribution, which might be difficult to obtain; *ii)* once $\mathcal{K}$ good combinations are found after sorted access by Algorithm 1, at least $\mathcal{K}$ top combinations (which include all the $\mathcal{K}$ good combinations) will be formed with random access. In practice, the target $\mathcal{K}_T$ in (4) acts as a conservative lower bound on the expected number $\bar{k} \geq \bar{\mathcal{K}} \geq \mathcal{K}_T$ of top combinations.

In most of the past literature on top-k the target number of top combinations is a user-defined input parameter. Other works [15] prefer instead to enforce a constraint on the cost of computing the result, and incrementally relax such constraint as more resources (e.g., elapsed time) become available. Such an approach can be formalized as the following problem:

$$\begin{array}{ll} \text{maximize} & \bar{\mathcal{K}}(n_1, n_2) \\ \text{subject to} & \bar{C}(n_1, n_2) \leq C_T, \text{ with } n_i \in \mathbf{N} \cap [0, N_i] \end{array} \quad (5)$$

where $\bar{\mathcal{K}}$ and $\bar{C}$ are as before, and C_T is a target cost. The strategies obtained by solving either problem (4) or (5) are equivalent, as shown in Appendix B. In this work, we adopt the formulation in (5) as it comes with a slightly more compact mathematical solution, while preserving the same results. For tractability, we also make the following simplifications:

- The ranking order of the tuples in a service is independent of the other service. This is reasonable in our setting, since we are joining the results of heterogenous services operating on different domains.
- We describe the distribution of the number of tuples per join value with a single parameter $Q_i = N_i/J_i$ representing its expected value.

4.1 Objective function formulation

The example in Table 1 shows a deterministic sample of the results returned by two services characterized by the following parameters: $Q_1 = 3$, $J_1 = 3$, $Q_2 = 2$, $J_2 = 4$, $J_{1,2} = 2$. Suppose, e.g., that we retrieve $n_1 = 6$ tuples from s_1 and $n_2 = 5$ tuples from s_2 (tuples not retrieved are shown in grey). In order to determine the number of good combinations that can be formed, we can proceed as follows:

1. Determine the distinct join values in the retrieved tuples, i.e., $\{b^{(1)}, b^{(2)}, b^{(3)}\}$ for s_1 and $\{b^{(1)}, b^{(2)}, b^{(6)}\}$ for s_2. Let $j_1(n_1) = 3$ and $j_2(n_2) = 3$ denote the cardinality of the two sets.

2. Determine the join values in common between s_1 and s_2, i.e., $\{b^{(1)}, b^{(2)}, b^{(3)}\} \bigcap \{b^{(1)}, b^{(2)}, b^{(6)}\} = \{b^{(1)}, b^{(2)}\}$. Let $j_{12}(n_1, n_2) = 2$ denote the cardinality of this set.

3. For each join value in common, form the combinations. The number of combinations equals the product of the multiplicities $q_i^{(r)}$ of the join value $b^{(r)}$ in the

TABLE 1
(a) Tuples in R_1 (black: retrieved, grey: not retrieved). (b) Tuples in R_2 (black: retrieved, grey: not retrieved). (c) Combinations formed with sorted access. (d) Additional combinations formed with random access.

(a)

$\boldsymbol{A}_1$	$\boldsymbol{B}$	S_1
$a_1^{(4)}$	$b^{(2)}$	77
$a_1^{(3)}$	$b^{(3)}$	72
$a_1^{(6)}$	$b^{(3)}$	63
$a_1^{(9)}$	$b^{(1)}$	53
$a_1^{(8)}$	$b^{(1)}$	32
$a_1^{(1)}$	$b^{(3)}$	31
$a_1^{(7)}$	$b^{(2)}$	27
$a_1^{(5)}$	$b^{(1)}$	6
$a_1^{(2)}$	$b^{(2)}$	4

(b)

$\boldsymbol{A}_2$	$\boldsymbol{B}$	S_2
$a_2^{(2)}$	$b^{(6)}$	90
$a_2^{(6)}$	$b^{(6)}$	70
$a_2^{(3)}$	$b^{(1)}$	58
$a_2^{(4)}$	$b^{(2)}$	57
$a_2^{(7)}$	$b^{(1)}$	57
$a_2^{(1)}$	$b^{(2)}$	41
$a_2^{(5)}$	$b^{(7)}$	40
$a_2^{(8)}$	$b^{(7)}$	35

(c)

$\boldsymbol{A}_1$	A_2	$\boldsymbol{B}$	S
$\mathbf{a}_1^{(9)}$	$\mathbf{a}_2^{(3)}$	$\mathbf{b}^{(1)}$	**53**
$\mathbf{a}_1^{(9)}$	$\mathbf{a}_2^{(7)}$	$\mathbf{b}^{(1)}$	**53**
$\mathbf{a}_1^{(8)}$	$\mathbf{a}_2^{(3)}$	$\mathbf{b}^{(1)}$	**32**
$\mathbf{a}_1^{(8)}$	$\mathbf{a}_2^{(7)}$	$\mathbf{b}^{(1)}$	**32**
$\mathbf{a}_1^{(4)}$	$\mathbf{a}_2^{(4)}$	$\mathbf{b}^{(2)}$	**57**

(d)

$\boldsymbol{A}_1$	A_2	$\boldsymbol{B}$	S
$\mathbf{a}_1^{(4)}$	$\mathbf{a}_2^{(1)}$	$\mathbf{b}^{(2)}$	**41**
$a_1^{(5)}$	$a_2^{(3)}$	$b^{(1)}$	6
$a_1^{(5)}$	$a_2^{(7)}$	$b^{(1)}$	6
$a_1^{(7)}$	$a_2^{(4)}$	$b^{(2)}$	27
$a_1^{(2)}$	$a_2^{(4)}$	$b^{(2)}$	4

first n_i tuples, i.e., $q_1^{(1)} q_2^{(1)} = 2 \cdot 2 = 4$ for $b^{(1)}$, and $q_1^{(2)} q_2^{(2)} = 1 \cdot 1 = 1$ for $b^{(2)}$.
Table 1(c) shows the $\mathcal{K} = 5$ good combinations thus formed, with $\mathcal{S}(\tau) = \min\{\tau[S_1], \tau[S_2]\}$ (individual scores not shown).

We adopt the following notation: let x denote a discrete-valued random variable and $p(x)$ its probability mass function (p.m.f.). We write $p(x; a)$ when the p.m.f. depends on the deterministic parameter a, while $p(x|y)$ denotes the conditional p.m.f. of x with respect to the random variable y. We use $\bar{x}$ as a shorthand for the expectation of x, i.e., $\bar{x} = E[x]$.

1) For a given number of tuples n_i retrieved with sorted accesses, the number of distinct join attribute values $j_i(n_i)$ is a random variable whose p.m.f. $p(j_i; n_i)$ can be determined as shown in Appendix C.
2) The number of common join attributes $j_{12}(n_1, n_2)$ can also be characterized as a random variable. With reference to Figure 3, we want to determine the p.m.f. of $j_{12}(n_1, n_2)$ given the knowledge of the p.m.f.'s $p(j_1; n_1)$ and $p(j_2; n_2)$. The analytical derivation of the p.m.f. is illustrated in detail in Appendix C. For the sake of the following discussion, we note that the expected value of $j_{12}(n_1, n_2)$ is given by

$$\bar{j}_{12}(n_1, n_2) = \frac{\bar{j}_1(n_i)\bar{j}_2(n_2)J_{1,2}}{J_1 J_2} \quad (6)$$

3) The number of combinations that can be formed after the sorted accesses is equal to

$$\mathcal{K}(n_1, n_2) = j_{12}(n_1, n_2) \cdot q_1(n_1) \cdot q_2(n_2) \quad (7)$$

where $q_i(n_i)$ denotes the number of tuples (among the first n_i tuples of service s_i) in which the same value of a given join attribute (among those found in the first n_i tuples of s_i) occurs. We notice that the random variables $q_i(n_i)$ and $j_i(n_i)$ are related by the expression

$$j_i(n_i) = \frac{n_i}{q_i(n_i)} \quad (8)$$

Replacing (8) in (7) and the random variables with their expected values we obtain

$$\bar{\mathcal{K}}(n_1, n_2) \simeq \frac{\bar{j}_1(n_1)\bar{j}_2(n_2)J_{1,2}}{J_1 J_2} \frac{n_1}{\bar{j}_1(n_1)} \frac{n_2}{\bar{j}_2(n_2)} = \frac{n_1 n_2 J_{1,2}}{J_1 J_2} \quad (9)$$

Note that the factor $\frac{J_{1,2}}{J_1 J_2}$ represents the selectivity of the join. Known techniques can be used to estimate such selectivity factor [16], even when the join is general and includes, e.g., selection predicates.

4.2 Cost constraint formulation

In order to find the top k combinations ($k \geq \mathcal{K}$), we now make random accesses. In the example of Table 1, we need to retrieve the tuples in s_2 whose join value appeared at least once in the first n_1 tuples of s_1, and vice versa. Table 1(d) shows the additional combinations formed via random access: one using $b^{(2)}$ on s_2 and four using $b^{(1)}$ and $b^{(2)}$ on s_1. The top combinations are necessarily included in the union of the combinations formed after the sorted accesses (i.e., the good combinations, shown in Table 1(c)) and random accesses (Table 1(d)), since no other combination can score higher than those in Table 1(c). Here, $k = 6$ combinations (in bold) are guaranteed to be top, as their aggregate scores exceed the upper bound $u = f(\tau_1^{(n_1)}[S_1], \tau_2^{(n_2)}[S_2]) = \min\{\tau_1^{(6)}[S_1], \tau_2^{(5)}[S_2]\} = 31$.

In the following, we assume that the cost of retrieving the tuples dominates over the cost of computing the combinations and their scores. This is reasonable in this context, since services are typically accessed remotely on the Web. We adopt the following additive cost model:

$$C(n_1, n_2) = sc_1 n_1 + sc_2 n_2 + rc_2 j_1(n_1) + rc_1 j_2(n_2) \quad (10)$$

where sc_i is the unitary sorted access cost (per tuple), rc_i the unitary random access cost (per distinct join attribute tuple), and $j_i(n_i)$ the number of distinct join attribute tuples retrieved from s_i. Note that the random access cost of s_1 depends on rc_1 and on $j_2(n_2)$, and vice versa.

Given n_1 and n_2, $C(n_1, n_2)$ as of (10) is a random variable, since it is a function of $j_1(n_1)$ and $j_2(n_2)$. We are interested in imposing a constraint on the expected cost, i.e., $\bar{C}(n_1, n_2) = E[C(n_1, n_2)] \leq C_T$, where we can write

$$\bar{C}(n_1, n_2) = sc_1 n_1 + sc_2 n_2 + rc_2 \bar{j}_1(n_1) + rc_1 \bar{j}_2(n_2). \quad (11)$$

An analytical expression of $\bar{j}_i(n_i)$ cannot be easily derived. Therefore, we seek for an approximation based on the related random variable $q_i(n_i)$. Indeed, from (8) we get

$$\bar{j}_i(n_i) \simeq \frac{n_i}{\bar{q}_i(n_i)}. \quad (12)$$

For a given value of n_i and an arbitrarily selected join attribute value, let $\tilde{q}_i(n_i)$ denote the number of times such join attribute value appears in the first n_i tuples. The p.m.f. of the random variable $\tilde{q}_i(n_i)$ is derived in Appendix C and it is defined over the range $[0, Q_i]$. Its expected value is equal to $E[p(\tilde{q}_i; n_i)] = Q_i \frac{n_i}{N_i}$.

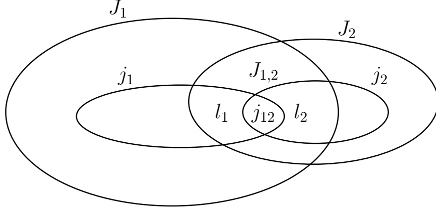


Fig. 3. Pictorial representation of the sets of distinct join attributes, used to derive an expression for $p(j_{12}; n_1, n_2)$. Each set is characterized by a cardinality, which is indicated in the figure.

The random variable $q_i(n_i)$ used in (8) differs from $\tilde{q}_i(n_i)$, since, in the former, it is implied that each of the $j_i(n_i)$ distinct join attributes occurs at least once in the first n_i tuples. Therefore, the p.m.f. of $q_i(n_i)$, written $p(q_i; n_i)$, which is defined over $[1, Q_i]$, is given by the conditional p.m.f. $p(\tilde{q}_i|\tilde{q}_i > 0; n_i)$, which can be easily obtained from $p(\tilde{q}_i; n_i)$, i.e.:

$$p(q_i; n_i) = p(\tilde{q}_i|\tilde{q}_i > 0; n_i) = \frac{p(\tilde{q}_i; n_i)}{1 - p(\tilde{q}_i = 0; n_i)} \quad (13)$$

Therefore, the expected value $\bar{q}_i(n_i)$ is given by

$$\bar{q}_i(n_i) = \frac{Q_i \frac{n_i}{N_i}}{1 - p(\tilde{q}_i = 0; n_i)} \quad (14)$$

We observed, and also verified numerically, that approximating $\bar{q}_i(n_i)$ by means of a straight line is generally applicable, and gets more accurate for larger values of Q_i. Therefore, we shall use the following approximated expression

$$\bar{q}_i(n_i) \simeq \frac{Q_i - 1}{N_i - 1}(n_i - 1) + 1 \quad (15)$$

Substituting (15) in (12) we get

$$\bar{j}_i(n_i) \simeq \frac{n_i}{a_i n_i + b_i} \quad (16)$$

where $a_i = \frac{Q_i - 1}{N_i - 1}$ and $b_i = 1 - a_i$. The proposed approximation (16) tends to slightly underestimate the true value of $\bar{j}_i(n_i)$. We observed that the goodness of fitting improves for larger values of Q_i.

4.3 Problem solution

With the approximations derived in Section 4.1 and Section 4.2, the optimization problem (5) can be written as

$$\begin{aligned} &\text{maximize} \quad \frac{n_1 n_2 J_{1,2}}{J_1 J_2} \\ &\text{subject to} \quad sc_1 n_1 + sc_2 n_2 + rc_2 \frac{n_1}{a_1 n_1 + b_1} + rc_1 \frac{n_2}{a_2 n_2 + b_2} \leq C_T \\ &\qquad\qquad n_i \in \mathbf{N} \cap [0, N_i] \end{aligned} \quad (17)$$

In order to solve (17), we relax the problem by replacing the integer constraint $n_i \in \mathbf{N} \cap [0, N_i]$ with $n_i \in \mathbf{R}$. In addition, we formulate an equivalent optimization problem by substituting the objective function with $\log(n_1 n_2)$. Thus, we seek the solution of problem (18):

$$\begin{aligned} &\text{maximize} \quad \log(n_1 n_2) \\ &\text{subject to} \quad sc_1 n_1 + sc_2 n_2 + rc_2 \frac{n_1}{a_1 n_1 + b_1} + rc_1 \frac{n_2}{a_2 n_2 + b_2} \leq C_T \end{aligned} \quad (18)$$

The Lagrangian function related to problem (18) is

$$\begin{aligned} L(n_1, n_2, \lambda) = &- \log(n_1 n_2) + \lambda(sc_1 n_1 + sc_2 n_2 \\ &+ rc_2 \frac{n_1}{a_1 n_1 + b_1} + rc_1 \frac{n_2}{a_2 n_2 + b_2} - C_T) \end{aligned} \quad (19)$$

The relaxed problem is convex, since the objective function is convex and the constraint is a convex function since it is a sum of linear fractionals, each of which is convex. We can impose the Karush-Kuhn-Tucker conditions, which must be satisfied by the optimal solution $\langle n_1^*, n_2^* \rangle$ of problem (18), i.e.:

- Stationarity

$$\frac{\partial L(n_1^*, n_2^*, \lambda^*)}{\partial n_i} = 0 \quad (20)$$

- Primal feasibility

$$sc_1 n_1^* + sc_2 n_2^* + rc_2 \frac{n_1^*}{a_1 n_1^* + b_1} + rc_1 \frac{n_2^*}{a_2 n_2^* + b_2} - C_T \leq 0 \quad (21)$$

- Dual feasibility

$$\lambda^* \geq 0 \quad (22)$$

- Complementary slackness

$$\lambda^*(sc_1 n_1^* + sc_2 n_2^* + rc_2 \frac{n_1^*}{a_1 n_1^* + b_1} + rc_1 \frac{n_2^*}{a_2 n_2^* + b_2} - C_T) = 0 \quad (23)$$

From (20) we get for $i = 1$

$$-\frac{1}{n_1^*} + \lambda^* sc_1 + \frac{\lambda^* rc_2 b_1}{(a_1 n_1^* + b_1)^2} = 0 \quad (24)$$

The optimal solution n_1^* must be a root of (24). Therefore, it must be a root of a third-degree polynomial $\alpha_3 x^3 + \alpha_2 x^2 + \alpha_1 x + \alpha_0$ with coefficients:

$$\alpha_3 = \lambda^* sc_1 a_1^2 \quad (25)$$

$$\alpha_2 = -a_1^2 + 2\lambda^* sc_1 a_1 b_1 \quad (26)$$

$$\alpha_1 = -2a_1 b_1 + \lambda^* sc_1 b_1^2 + \lambda^* rc_2 b_1 \quad (27)$$

$$\alpha_0 = -b_1^2 \quad (28)$$

Among the three roots of the polynomial, we pick the one which is real and positive. Let $n_1^* = n_1(\lambda^*, a_1, b_1, sc_1, rc_2)$ denote such a root. An equivalent procedure can be followed for s_2. Therefore, we can write

$$\begin{aligned} n_1^* &= n_1(\lambda^*, a_1, b_1, sc_1, rc_2) \\ n_2^* &= n_2(\lambda^*, a_2, b_2, sc_2, rc_1) \end{aligned} \quad (29)$$

In (29) we adopt a shorthand notation to express the fact that the optimal solution $\langle n_1^*, n_2^* \rangle$ can be computed in terms of the problem parameters and the Lagrange multiplier λ^*. This is achieved by finding one of the

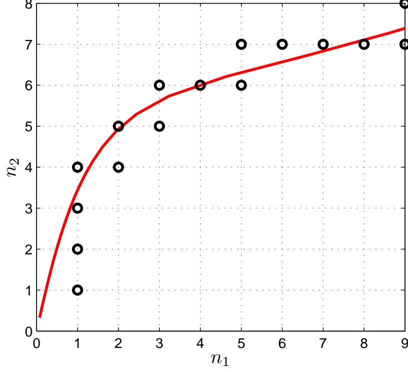


Fig. 4. The solid red curve indicates the locus of optimal solutions of the relaxed optimization problem (18) obtained by varying C_T. The black circles indicate the approximate solution obtained after rounding.

roots of a third degree polynomial, whose coefficients are expressed in terms of, respectively, $(\lambda^*, a_1, b_1, sc_1, rc_2)$ and $(\lambda^*, a_2, b_2, sc_2, rc_1)$. For a given value of the target cost C_T, the optimal value of the Lagrangian multiplier λ^* can be found by means of an iterative procedure performing a binary search over the possible values of λ^*. More specifically, let $j = 1, \ldots, \mathcal{J}$ denote the iteration counter, λ_j the value of the Lagrangian multiplier at the j-th iteration, and $\langle n_{1,j}, n_{2,j}\rangle$ the corresponding candidate solution. We define $n_{1,j} = n_1(\lambda_j, a_1, b_1, sc_1, rc_2)$ and $n_{2,j} = n_2(\lambda_j, a_2, b_2, sc_2, rc_1)$, i.e., the values of n_1 and n_2 computed when we replace λ^* with an arbitrary non-negative value λ_j. The search for λ^* (and, consequently, for $\langle n_1^*, n_2^*\rangle$) proceeds for a large enough number of steps $\mathcal{J}$, until the the candidate solution $\langle n_{1,j}, n_{2,j}\rangle$ approximately satisfies the cost constraint with the equality, i.e., $\bar{C}(n_{1,j}, n_{2,j}) - C_T \simeq 0$

1) Let $\lambda_l = 0$ and $\lambda_u = \Lambda$, where Λ is a positive constant (in our experiments, we set $\Lambda = 10$ when C_T is of the order of seconds).
2) for $j = 1 : \mathcal{J}$
   a) Set $\lambda_j = \frac{\lambda_l + \lambda_u}{2}$.
   b) Find $n_{1,j} = n_1(\lambda_j, a_1, b_1, sc_1, rc_2)$ and $n_{2,j} = n_2(\lambda_j, a_2, b_2, sc_2, rc_1)$.
   c) if $\bar{C}(n_{1,j}, n_{2,j}) - C_T \geq 0$, then $\lambda_l = \lambda_j$. Else $\lambda_u = \lambda_j$.
3) $\lambda^* = \lambda_{\mathcal{J}}$

The number of steps $\mathcal{J}$ of the binary search determines the accuracy of the solution. Typically, $\mathcal{J} = 10$ steps are sufficient for the problem at hand. Finally, we reintroduce the integer constraint by rounding the optimal solution of the relaxed problem and checking that $n_i^* \leq N_i$.

Figure 4 shows the curve obtained for the services in Table 1, when $sc_1 = 1$, $sc_2 = 2$, $rc_1 = 1$ and $rc_2 = 10$. Each point on the curve is the solution of problem (18) for some value of C_T. The black circles indicate the result obtained after rounding.

The cost of finding the solution is in the order of few milliseconds, and thus totally negligible with respect to the typical access times required in practical cases involving remote services. Note also that the cost of solving the problem does not depend in any way on the size of the data sets underlying the services involved in the join.

4.4 Execution strategy

We are interested in defining a strategy that describes how the services shall be incrementally accessed, in order to maximize at each execution step the number of good combinations found, given the knowledge of the parameters that characterize the services. To this end, instead of solving problem (18) for a fixed value of C_T, we let C_T vary over a positive interval so as to trace a locus of optimal solutions. The cost constraint C_T represents the overall time available to fetch tuples from the services. C_T is incremented until the desired number of combinations is achieved; fine increments are possible, so as to trace an almost continuous curve in the n_1/n_2 plane, since (18) is solved very quickly. The curve for the Example in Table 1 was shown in Figure 4.

To address pagination, we constrain the number of items retrieved from s_i to be a multiple of a given page size $\mathcal{P}_i \in \mathbf{N}$. The *CARS* pulling strategy determining, at each step, the next service to be invoked is defined as follows.

Query execution starts at step number $t = 1$, with $\langle n_1^{(1)}, n_2^{(1)}\rangle = \langle \mathcal{P}_1, \mathcal{P}_2\rangle$, i.e., at least one request is sent to each service, otherwise the result would be empty. Then, at any step $t > 1$, the execution proceeds by performing sorted accesses according to one of the following choices

$$\begin{aligned} \langle n_1^{(t)}, n_2^{(t)}\rangle &= \langle n_1^{(t-1)} + \mathcal{P}_1, n_2^{(t-1)}\rangle \\ \langle n_1^{(t)}, n_2^{(t)}\rangle &= \langle n_1^{(t-1)}, n_2^{(t-1)} + \mathcal{P}_2\rangle \end{aligned} \tag{30}$$

At each step $t > 1$, we use one choice in (30) to move from $\langle n_1^{(t-1)}, n_2^{(t-1)}\rangle$ to the new pair $\langle n_1^{(t)}, n_2^{(t)}\rangle$ that lies closest to the curve induced by the solution of (18), traced by varying the target cost C_T. Intuitively, this corresponds to finding an approximate solution to (18), with the additional pagination constraint, for a given cost $C_T^{(t)}$ that is larger than the cost $C_T^{(t-1)}$ implicitly used at step $t-1$. Observe that pagination does not affect the complexity of the proposed method, as it is introduced after problem (18) is solved.

Example 4.1: We now illustrate the execution of *CARS* on the data of Table 1 to determine the top combination. *CARS* starts with one sorted access to s_1 and one to s_2, thus retrieving $\langle a_1^{(4)}, b^{(2)}, 77\rangle$ and $\langle a_2^{(2)}, b^{(6)}, 90\rangle$. Then, it performs one random access to s_2 with value $b^{(2)}$ and one to s_1 with value $b^{(6)}$, thus retrieving $\langle a_2^{(4)}, b^{(2)}, 57\rangle$ and $\langle a_2^{(1)}, b^{(2)}, 41\rangle$. Two combinations are formed (individual scores not shown): $\tau^{(1)} = \langle a_1^{(4)}, a_2^{(4)}, b^{(2)}, 57\rangle$ and $\tau^{(2)} = \langle a_1^{(4)}, a_2^{(1)}, b^{(2)}, 41\rangle$. The threshold computed according to (3) is $u = 77$. According to Figure 4, the next 3 sorted accesses made by *CARS* are on s_2.

The first sorted access returns $\langle a_2^{(6)}, b^{(6)}, 70\rangle$. No sorted access is made this time, since value $b^{(6)}$ has already been used on s_1. The threshold becomes $u = 70$. The next sorted access returns $\langle a_2^{(3)}, b^{(1)}, 58\rangle$. Then, a random access is made to s_1 with value $b^{(1)}$, thus retrieving $\langle a_1^{(9)}, b^{(1)}, 53\rangle$, $\langle a_1^{(8)}, b^{(1)}, 32\rangle$, and $\langle a_1^{(5)}, b^{(1)}, 6\rangle$. The new formed combinations are: $\tau^{(3)} = \langle a_1^{(9)}, a_2^{(3)}, b^{(1)}, 53\rangle$, $\tau^{(4)} = \langle a_1^{(8)}, a_2^{(3)}, b^{(1)}, 32\rangle$, and $\tau^{(5)} = \langle a_1^{(5)}, a_2^{(3)}, b^{(1)}, 6\rangle$. The threshold becomes $u = 58$. The next sorted access returns $\langle a_2^{(4)}, b^{(2)}, 57\rangle$, which determines the already formed combination $\tau^{(1)}$. The threshold becomes $u = 57$, therefore $\tau^{(1)}$ is guaranteed to be the top combination and *CARS* stops. According to (10), the total cost incurred by the execution with the parameters in use ($sc_1 = 1, sc_2 = 2, rc_1 = 1, rc_2 = 10$) with the current depths $n_1 = 1$ and $n_2 = 4$ is $C(1,4) = sc_1 \cdot 4 + sc_2 \cdot 1 + rc_2 \cdot 1 + rc_1 \cdot 3 = 19$.

For comparison, HRJN always alternates sorted accesses to s_1 and s_2. The guarantee to have formed the top combination will be reached only after accessing $n_1 = 4$ tuples from s_1 and $n_2 = 4$ tuples from s_2, with a total cost $C(4,4) = sc_1 \cdot 4 + sc_2 \cdot 4 + rc_2 \cdot 3 + rc_1 \cdot 3 = 45$.

HRJN* accesses s_1 if $u_1 > u_2$, else s_2, where u_1 and u_2 are computed according to (2) (in case of ties, the service with the least depth is accessed, and, on further tie, s_1 is chosen). As with *CARS* and HRJN, a random access is made after every new join attribute value discovered by sorted access. We then focus on sorted access only. After the initial sorted access to s_1 and s_2, we have $u_1 = u_2 = 77$. Then HRJN* makes a sorted access to s_1, i.e., $n_1 = 2$, $n_2 = 1$, $u_1 = 72$, $u_2 = 77$. Then s_2 is chosen, which determines $n_1 = 2$, $n_2 = 2$, $u_1 = 72$, $u_2 = 70$. Then s_1, giving $n_1 = 3$, $n_2 = 2$, $u_1 = 63$, $u_2 = 70$. Then s_2 with $n_1 = 3$, $n_2 = 3$, $u_1 = 63$, $u_2 = 58$. Then s_1, with $n_1 = 4$, $n_2 = 3$, $u_1 = 53$, $u_2 = 58$. Finally s_2 with $n_1 = 4$, $n_2 = 4$, $u_1 = 58$, $u_2 = 57$. Here HRJN* stops, as $u = 57$ coincides with the score of a formed combination. The execution strategy is in this case identical to that of HRJN, and so is the incurred cost. ∎

5 Experiments

We investigate the overall cost, according to the additive cost model in (10), to return the correct top-k combinations when the proposed *CARS* pulling strategy is adopted. We compare this with the optimal cost attained by an oracle-based pulling strategy. We conduct our analysis on both real and synthetic data sets. For the former, *CARS* has been integrated and tested in the SeCo system [1] as one of the strategies available for execution. For the latter, we extensively evaluate the impact, on the overall cost, of the problem parameters, i.e., page size, number of distinct join attribute values, average number of tuples per join attribute value, sorted and random access costs, score distributions.

Data sets. First, in order to provide a qualitative insight, we demonstrate the proposed approach on a concrete instantiation of our running example. We consider two services available on the Web, which provide both sorted and random access, namely `hotel` for hotels as service s_1 and `restaurant` for Parisian restaurants as service s_2. The former service, among the available access patterns, allows requesting hotels as required in the example: by sorted accesses, ranked by stars, in descending order from luxury hotels down to hotels with no star, and by random accesses by searching for hotels in a given Parisian street. Results are paginated with a page size $\mathcal{P}_1 = 25$ tuples/page. The total number of tuples of s_1 is $N_1 = 858$, the number of distinct join values, i.e., different streets, is $J_1 = 592$. The average number of hotels per street is $Q_1 = 1.45$.

The latter service can be invoked by sorted accesses, returning restaurants ranked by customer rating in the $[0, 10]$ range, and by random accesses as above. We restricted our search to restaurants serving French food for a total of $N_2 = 1198$ restaurants. Search results are paginated with a page size $\mathcal{P}_2 = 6$. A total number of $J_2 = 977$ distinct streets can be selected, of which $J_{1,2} = 188$ are in common. The average number of restaurants per street is $Q_2 = 1.23$. We normalized the scores of both services to fall within the $[0, 1]$ interval.

If all the tuples of both services are retrieved, it is possible to form a total of 590 combinations. We estimated the average access costs involved in both sorted and random access, and we obtained $sc_1 = 0.01$, $sc_2 = 0.01$ (in seconds/tuple), $rc_1 = 0.10$, $rc_2 = 0.01$ (in seconds/access).

Second, in order to provide a controlled test bed and obtain quantitative results, we synthetically generated data for two services independently. For each service, we set the number of distinct join attribute values J_i, of which $J_{1,2}$ are in common. For each distinct join value, we generated a number of tuples according to a Poisson distribution whose mean is set equal to Q_i. Therefore, each distinct join attribute value is contained in a potentially different number of tuples to mimic real world data. The individual scores of the tuples are sampled from an exponential distribution with mean μ_i. We also consider data sets whose scores are sampled from either a uniform or a Zipfian distribution. The relevant parameters are summarized in Table 2, with default values in bold.

Methods. We test five different pulling strategies applied to the rank join operator defined in Algorithm 1, endowed with both sorted and random access:

i) Round-robin (*RR*), i.e., alternating sorted accesses to s_1 and s_2. This is the default pulling strategy defined by HRJN [2] and Fagin's "combined algorithm" as regards sorted access. Indeed, as was discussed in Section 3, the extension of Fagin's "combined algorithm" to rank join forces to perform all random accesses. When the two services are characterized by different page sizes, the round-robin strategy is modified to select the service with the least depth, so as to ensure that both services are explored up to, approximately, the same depth;

TABLE 2
Operating parameters (defaults in bold).

description	parameter	tested values
Distinct join values	J_1	10, **25**, 50
Skewness	J_2/J_1	1/2, **1**, 2
Distinct join values in common	$J_{1,2}$	$0.8\min(J_1, J_2)$
Tuples per join value	Q_1	5, **20**, 50
Skewness	Q_2/Q_1	1/2, **1**, 2
Top combinations to be found	k	1, 10, **100**
Sorted access cost	sc_1	**0.01**, 0.1, 1.0
Skewness	sc_2/sc_1	**1/10**, 1, 10
Random access cost	rc_1	0.01, **0.1**, 1.0
Skewness	rc_2/rc_1	1/10, 1, **10**
Page size	$\mathcal{P}_1$	1, **5**, 10
Skewness	$\mathcal{P}_2/\mathcal{P}_1$	1/2, **1**, 2
Score distribution	–	**Exp.**, Uniform, Zipfian
Score distr. parameter	μ_1	**2**
Skewness	μ_2/μ_1	1/2, **1**, 2

ii) Score-aware (SA), i.e., a strategy deciding dynamically which service to access next based on the scores of the retrieved tuples by comparing the bounds in (2): a sorted access to s_1 (s_2) is made if $u_1 > u_2$ ($u_1 < u_2$). In case of ties, the service with the least depth is accessed. This is the strategy adopted by HRJN* [2], which is known to perform very well in practice in the scenario without random access [17]. Although SA usually performs better than RR, its cost-obliviousness and the presence of random access may sometimes cause an exploration that incurs higher costs than RR.

iii) Cost-aware (*CA*), i.e., assuming knowledge of sorted access costs only. The corresponding pulling strategy is obtained by solving problem (5) by setting $rc_i = 0$. The solution is such that $n_1/n_2 = sc_2/sc_1$.

iv) Cost-aware with Random and Sorted access (*CARS*), i.e., the proposed pulling strategy defined in Section 4. We remark that the cost of computing the strategy is negligible with respect to the access costs, regardless of the size of the data set.

v) Oracle-based, i.e., the non-deterministic, prescient pulling strategy characterized by the minimum cost, among those that allow computing the correct top-k combinations. This is obtained *a posteriori* by considering all feasible $\langle n_1, n_2\rangle$ pairs. For each pair, we measure the number k' of combinations that exceed the upper bound defined in (3). If $k' \geq k$ we compute the overall cost. Finally, we select the pair that achieves the minimum cost.

Evaluation metrics. We measure the overall additive cost expressed in (10). We normalize the costs with respect to the cost of the oracle-based pulling strategy. Thus, all relative costs are ≥ 1. A pulling strategy characterized by a relative cost approaching 1 is near-optimal.

Results for real data sets. Figure 5 illustrates the access plan computed with *CARS* for the running example. The solid curve represents the locus of solutions for the relaxed problem (18), while the dark circles indicate the number of tuples retrieved from the two services during the sorted access phase at each execution step $t = 1, \ldots, T$, with $T = 16$. Here, the two services are characterized by equal sorted access costs but different random access costs. Then, a cost-aware strategy would intuitively request more tuples from s_1 in the sorted access phase, since the random accesses to s_2 are cheaper.

TABLE 3
Step-by-step access plan retrieving the top hotel-restaurant combinations in the same street.

t	$\mathcal{K}^{(t)}$	$k^{(t)}$	$n_1^{(t)}$	$n_2^{(t)}$	$j_1^{(t)}$	$j_2^{(t)}$	$C(n_1^{(t)}, n_2^{(t)})$
1	0	0	25	6	25	6	1.16 s
2	0	0	50	6	48	6	1.64 s
3	0	0	50	12	48	11	2.20 s
4	0	0	75	12	70	11	2.67 s
5	1	2	75	18	70	17	3.33 s
6	2	2	100	18	90	17	3.78 s
7	2	2	125	18	106	17	4.19 s
8	2	4	125	24	106	23	4.85 s
9	2	104	150	24	127	23	5.31 s

TABLE 4
Top 5 hotel-restaurant combinations in the same street.

Hotel	Restaurant	Street	Score
De Vendôme	Café de Vendôme	Pl. Vendôme	0.6
Montaigne	Alain Ducasse	Av. Montaigne	0.56
Pont Royal	L'Atelier de Robuchon	Rue Montalembert	0.56
Royal St. Honoré	Au Dauphin	Rue St. Honoré	0.552
Le Walt	Chez Valdo	Av. de La Motte Picquet	0.552

Note that in this example both Q_1 and Q_2 are close to 1, thus the *CARS* strategy approximately follows a straight line. Indeed, when $Q_1 = Q_2 = 1$, the expected cost function becomes linear in n_1 and n_2. The locus of solutions obtained by varying C_T results in a straight line with slope $\frac{sc_1+rc_2}{sc_2+rc_1}$.

Table 3 provides further details. At each step t, we indicate the number of good combinations $\mathcal{K}^{(t)}$ formed after the sorted access phase (such that at least $\mathcal{K}^{(t)}$ top combinations are guaranteed to be retrieved after the random access phase), the number of top combinations $k^{(t)}$ whose aggregate scores exceed the current upper bound, the numbers of tuples retrieved by sorted access $\langle n_1^{(t)}, n_2^{(t)}\rangle$, the number of distinct streets $j_i^{(t)}$ retrieved from s_i, and the overall access cost $C(n_1^{(t)}, n_2^{(t)})$ (in seconds) due to both sorted and random accesses. The leap from $k^{(8)} = 4$ to $k^{(9)} = 104$ is due to the fact the 9th step accesses the first page in s_1 containing 3-star hotels, which significantly lowers the upper bound. Table 4 shows the top 5 combinations in our example.

Observe that *CARS* is able to find the top 100 combinations in 5.31 seconds. For comparison, the simpler round-robin pulling strategy needs 17.45 seconds to return the same result (and zero top combinations can be returned after 5.31 seconds), while the optimal, oracle-based, strategy requires 3.43 seconds. Note that *CA* coincides with RR here, as $sc_1 = sc_2$, while SA slightly deviates from this strategy, with a further cost increase, as the score differences caused a higher number of costly random accesses to s_1. Note that computing the *CARS* strategy by solving (18) takes only 20 ms and is thus totally negligible with respect to the overall access cost.

Results for synthetic data sets. The results obtained on the synthetic data sets are shown in Figure 6. The y-axis reports the overall access cost to return the correct top-100 combinations, relative to the cost of the oracle-

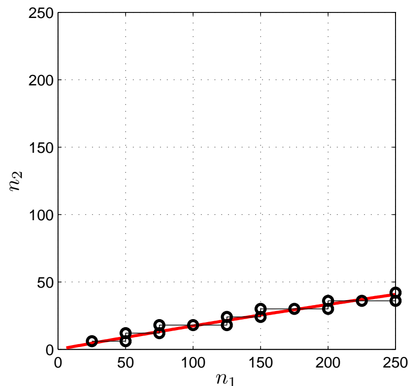


Fig. 5. Cost-aware access plan for retrieving the top-k combinations. The solid curve represents the locus of optimal solutions of the relaxed problem (18). The dark circles indicate the number of tuples retrieved from the two services during the sorted access phase.

based strategy. For clarity, on top of each bar we also report the absolute (non-normalized) overall access cost, expressed in terms of wall-clock time. Each bar represents the average over ten data sets. Here, too, the cost of determining the *CARS* strategy is within 20 ms, and thus totally negligible with respect to the total access cost, since the strategy is computed only once, at compile time, before the execution of the algorithm.

As a general trend, we observe that *i) CARS* outperforms the other (non-oracle-based) strategies, especially when services are characterized by heterogeneous costs; *ii) CARS* always performs close to the oracle-based optimal strategy on all data sets.

In the default configuration, the two services are symmetric with respect to $\mathcal{P}_i$, J_i, Q_i and score distribution. The only asymmetry arises in the access costs, since $sc_2/sc_1 = 1/10$ and $rc_2/rc_1 = 10$. Thus, performing sorted access to s_2 is cost-effective, since it is less expensive and, at the same time, the corresponding random accesses to s_1 are cheaper. On these data sets, *CARS* achieves the same performance as the oracle-based strategy. Conversely, the relative cost overhead of the other strategies is: RR +117%, CA +32%, SA + 76%. In the following we comment on the effect of the individual parameters.

Sorted access cost - sc_i: Figure 6(a) reports the relative cost obtained by varying the sorted access cost sc_1 (when $sc_2/sc_1 = 1/10$, as in the default configuration). For large values of sc_1, sorted access cost dominates random access cost. Thus, the *CARS* strategy is approximated by *CA*. In all cases, the overhead is up to +23%. Figure 6(d) shows that *CARS* is robust to largely skewed sorted access costs, consistently performing close to the oracle-based strategy.

Random access cost - rc_i: Similar observations can be made for Figures 6(b) and 6(e). For small values of rc_1, sorted access cost dominates, and *CARS* is approximated by *CA*. In all cases, the overhead of *CARS* wrt. the oracle-based strategy is up to $+8\%$.

Score distribution: Figure 6(c) shows that *CARS* performs close to the oracle-based strategy independently of the shape of the distribution.

Score distribution skewness - μ_2/μ_1: *CARS* is oblivious of the score distribution. While this enables to compute the pulling strategy at compile-time, losses are to be expected when the score distribution are largely skewed. For large values of μ_2/μ_1, both scores and access costs call for accessing s_2 more frequently. Thus, *CARS* is close to optimal. Conversely, for smaller values of μ_2/μ_1, tuples with larger scores are to be expected to appear in s_1, and the overhead of *CARS* reaches +11%.

Page size - $\mathcal{P}_i$: As expected, for large page sizes, the differences between the tested cost-aware strategies are less significant, as shown in Figure 6(g). Indeed, their differences are partially canceled out by the approximation induced by the page size.

Number of tuples per distinct join value - Q_i: For small values of Q_i, *CARS* achieves nearly optimal cost (Figure 6(h)). For large values of Q_i, *CARS* exhibits a consistent behavior, while *CA* diverges. Similarly, Q_2/Q_1 does not impact the performance of *CARS* (Figure 6(k)).

Number of distinct join values - J_i: Varying the number of distinct join values, as shown in Figures 6(i) and 6(l), affects the data set size, but not the pulling strategies.

Number of top combinations to be found - k: Figure 6(j) shows that the relative performance of *CARS* wrt. the other strategies increases as k grows.

In order to show the robustness to different access cost configurations, we let both sorted and random access cost vary in the set $\{L = 0.01, M = 0.10, H = 1.00\}$ and compute the relative cost that can be achieved for each of the 81 possible cases. Figure 7(a) reports the relative costs for the *CARS* strategy. We observe that *CARS* is always close to the oracle-based strategy (up to +22% cost overhead). *CARS* is also robust wrt. uncertainty in the knowledge of the operating parameters, whose values may indeed come from estimates obtained via sampling. We modeled uncertain parameters by adding Gaussian noise with standard deviation equal to 30% of their nominal value. Figure 7(b) shows that, even under these adverse conditions, performance stays close to the oracle-based strategy.

6 Related work

Top-k query answering is a topic that has been addressed by a large body of research during the last years [18].

The problem originates from rank aggregation, where all relations contain the same objects, sorted in different orders. The objective function in (5) is inspired to Fagin's algorithm [19], which first performs sorted accesses to find at least k matching objects in the input relations, and then proceeds with random accesses to return the top-k objects. The *threshold algorithm* (TA) [3], [20] has a finer stopping condition based on an upper bound.

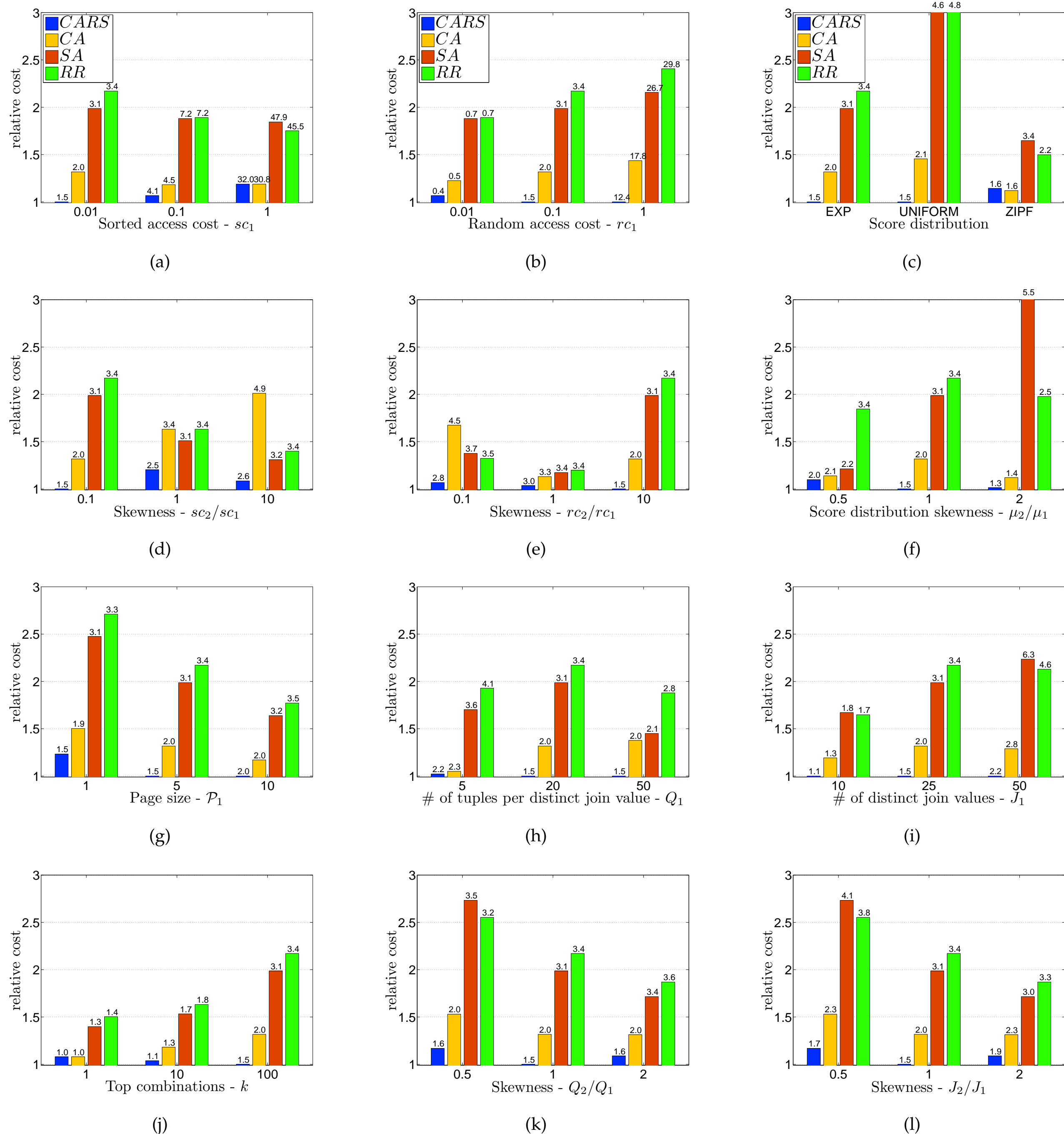
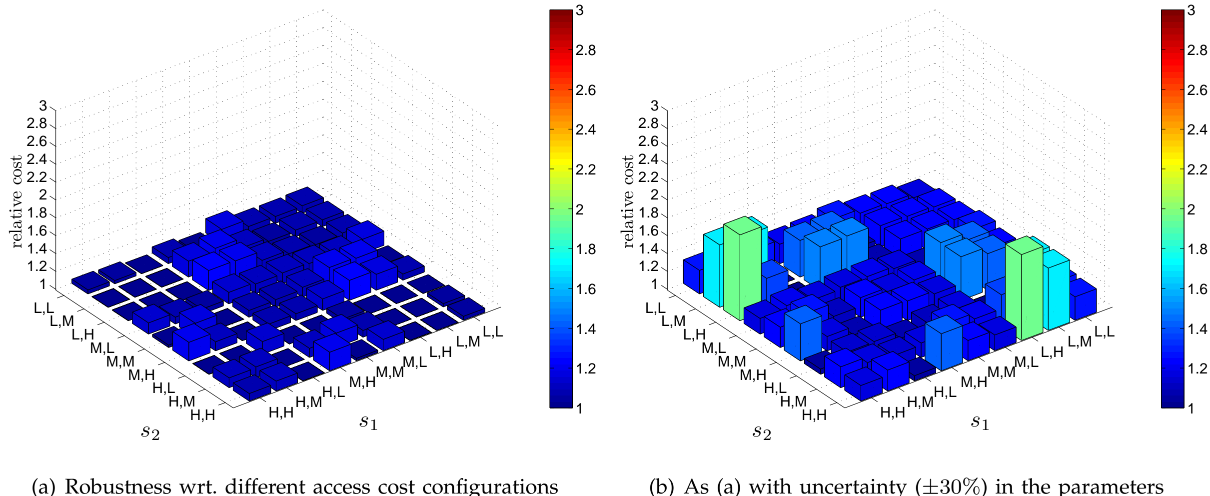


Fig. 6. Bar charts comparing relative costs in different parameters configurations.

TA is *instance optimal* (for monotone functions), i.e., it always achieves a cost within a fixed constant factor of optimal. In many practical scenarios, access costs may be significantly skewed; suitable cost models and pulling strategies for rank aggregation have been proposed. In particular, in [3] a cost model is proposed to guide the pulling strategy, in such a way that the frequency of random accesses is set equal to the ratio between random and sorted access costs. A more sophisticated pulling strategy is described in [7], which explicitly takes into account both access costs (*fixed*, for all ranked lists) and score distributions (modeled as score histograms) to determine the optimal sequence of sorted and random access in terms of overall access cost. A similar approach has been pursued in [6]. Several variants of TA have been proposed in the literature based on the different data access methods available: no random access (NRA [3], Stream Combine [8]); controlled random probes (MPro [9], [10], Upper and Pick [11], [12]); random access also returning the position (BPA and BPA2 [13]). All these algorithms, however, focus on rank aggregation and top-k selection queries, and



(a) Robustness wrt. different access cost configurations

(b) As (a) with uncertainty (±30%) in the parameters

Fig. 7. Relative costs wrt. different access cost configurations and uncertainty.

thus cannot be readily extended to the rank join setting. Indeed, as was pointed out in Section 3, the notion of lower bound does not extend to rank join in a useful way, since partially formed combinations may never become a combination, and thus their lower bound is undefined until the combination is fully formed. Instead, in the case of rank aggregation, every document always has an aggregate score. In addition, the probabilistic models used in rank aggregation algorithms would typically become more complex in rank join, while we have opted for a model that requires very limited knowledge about the data. Finer strategies than the baseline TA algorithm, such as the method of [7], typically exploit knowledge about score distribution (such as score histograms) and precompute pairwise term covariances from terms in frequent queries, which we do not assume to be available in our setting.

One of the first works on the problem, studied in this paper, of top-k query answering with joins is [21]. Their algorithm J^* is proved to be instance optimal among those that only use sorted access.

The HRJN (hash rank join) operator is introduced in [2] and shown there to outperform J^* by a large margin. The authors mainly focus on a scenario where only sorted access is available. In this setting, the bounding scheme summarized in Section 3 is proposed, together with an optimal pulling strategy (HRJN*). For the case of both sorted and random access, a suitable bounding scheme (corresponding to (3)) is briefly described, which turns out to correspond to that used by TA in rank aggregation. In our experiments in Section 5, we have compared both adaptations of HRJN (RR) and HRJN* (SA) to the presence of both sorted and random access with our algorithm. The work in [4] thoroughly analyzes the optimality properties of rank join operators with sorted access, including HRJN. A (costly) tight bounding scheme is proposed, such that the corresponding rank join operator is instance optimal, also when objects carry more than one score value and the join involves more than two relations. Trade-offs between efficiency and cost are further investigated in [17]. The problem of multi-way rank join, i.e., involving more than two services, is typically addressed by pipelining binary rank join operators [2], [4].

Attempts to provide estimates of the costs of a join in a top-k setting were made in [22], [23]. A probabilistic model is proposed to estimate the number of tuples (depth) consumed from each input to produce the top-k results. It is assumed that scores are uniformly distributed, joins are independent, all relations have equal size, and the scoring function is a weighted sum. A more general framework for depth estimation is given in [24].

All the aforementioned algorithms halt when the exact top-k results are guaranteed to be found. In [25], a confidence measure is computed, so as to provide an indication of the likelihood of having found the top-k results, anytime during the execution of the rank join operator. In [15] an approximate result is output if the available resource budget does not allow returning all top-k results.

7 Conclusions

Stimulated by the goal of answering multi-domain queries, in this paper we propose an execution strategy for retrieving the top k combinations that can be formed by joining the results of heterogeneous search engines. We optimize such a strategy with respect to an additive cost model that considers both sorted access and random access. To this end, we introduce a statistical framework to characterize the number of combinations and the number of random accesses.

We are presently considering extensions of the framework to the case of non-additive cost models capturing the fact that service accesses may be performed in parallel, and to pipelining joins in the presence of random access.

ACKNOWLEDGMENT

The authors acknowledge support from the Search Computing (SeCo) project, funded by the ERC.

REFERENCES

[1] M. Brambilla and S. Ceri, Eds., *Search Computing - Challenges and Directions*, ser. LNCS. Springer, March 2010, vol. 5950.
[2] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Supporting top-k join queries in relational databases," *VLDB J.*, vol. 13, no. 3, pp. 207–221, 2004.
[3] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 614–656, 2003.
[4] K. Schnaitter and N. Polyzotis, "Evaluating rank joins with optimal cost," in *PODS*, 2008, pp. 43–52.
[5] Y. N. Silva, W. G. Aref, and M. H. Ali, "The similarity join database operator," in *ICDE*, 2010.
[6] C. A. Lang, Y.-C. Chang, and J. R. Smith, "Making the threshold algorithm access cost aware," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 10, pp. 1297–1301, 2004.
[7] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum, "Io-top-k: Index-access optimized top-k query processing," in *VLDB*, 2006, pp. 475–486.
[8] U. Güntzer, W.-T. Balke, and W. Kießling, "Towards efficient multi-feature queries in heterogeneous environments," in *ITCC*, 2001, pp. 622–628.
[9] K. C.-C. Chang and S. won Hwang, "Minimal probing: supporting expensive predicates for top-k queries," in *SIGMOD Conference*, 2002, pp. 346–357.
[10] S. won Hwang and K. C.-C. Chang, "Probe minimization by schedule optimization: Supporting top-k queries with expensive predicates," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 5, pp. 646–662, 2007.
[11] N. Bruno, L. Gravano, and A. Marian, "Evaluating top-k queries over web-accessible databases," in *ICDE*, 2002, pp. 369–.
[12] A. Marian, N. Bruno, and L. Gravano, "Evaluating top- queries over web-accessible databases," *ACM Trans. Database Syst.*, vol. 29, no. 2, pp. 319–362, 2004.
[13] R. Akbarinia, E. Pacitti, and P. Valduriez, "Best position algorithms for top-k queries," in *VLDB*, 2007, pp. 495–506.
[14] A. Dasgupta, N. Zhang, and G. Das, "Leveraging count information in sampling hidden databases," in *ICDE*, 2009, pp. 329–340.
[15] M. Shmueli-Scheuer, C. Li, Y. Mass, H. Roitman, R. Schenkel, and G. Weikum, "Best-effort top-k query processing under budgetary constraints," in *ICDE*, 2009, pp. 928–939.
[16] P. J. Haas, J. F. Naughton, and A. N. Swami, "On the relative cost of sampling for join selectivity estimation," in *PODS*, 1994, pp. 14–24.
[17] J. Finger and N. Polyzotis, "Robust and efficient algorithms for rank join evaluation," in *SIGMOD Conference*, 2009, pp. 415–428.
[18] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-*k* query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, 2008.
[19] R. Fagin, "Combining fuzzy information from multiple systems," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 83–99, 1999.
[20] U. Güntzer, W.-T. Balke, and W. Kießling, "Optimizing multi-feature queries for image databases," in *VLDB*, 2000, pp. 419–428.
[21] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter, "Supporting incremental join queries on ranked inputs," in *VLDB*, 2001, pp. 281–290.
[22] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid, "Rank-aware query optimization," in *SIGMOD Conference*, 2004, pp. 203–214.
[23] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, H. G. Elmongui, R. Shah, and J. S. Vitter, "Adaptive rank-aware query optimization in relational databases," *ACM Trans. Database Syst.*, vol. 31, no. 4, pp. 1257–1304, 2006.
[24] K. Schnaitter, J. Spiegel, and N. Polyzotis, "Depth estimation for ranking query optimization," in *VLDB*, 2007, pp. 902–913.
[25] B. Arai, G. Das, D. Gunopulos, and N. Koudas, "Anytime measures for top- algorithms on exact and fuzzy data sets," *VLDB J.*, vol. 18, no. 2, pp. 407–427, 2009.

Davide Martinenghi received a MS as Computer Engineer from Politecnico di Milano, Italy, in 1998 and a Ph.D. in Computer Science from Roskilde University, Denmark, in 2005 with a dissertation on integrity checking for deductive databases. Presently, he is assistant professor at Politecnico di Milano. His areas of expertise include ranking queries, data integrity maintenance, knowledge representation, and, in a broad sense, query optimization aspects related to web data access and web search.

Marco Tagliasacchi Marco Tagliasacchi, born in 1978, received the "Laurea" degree (2002, cum Laude) in Computer Engineering and the Ph.D. in Electrical Engineering and Computer Science (2006), from Politecnico di Milano, Italy. Since February 2007 he is Assistant Professor at the Dipartimento di Elettronica e Informazione – Politecnico di Milano. His research interests include multimedia signal processing and information retrieval. Marco Tagliasacchi has been actively involved in several public and private funded projects.

APPENDIX A EXTENSIONS

A.1 Handling more than 2 services

The analytical problem formulation and solution described above for joins involving two services can be extended to M-ary joins, with $M > 2$. We illustrate this for the case of $M = 3$ services in a join $s_1 \bowtie s_2 \bowtie s_3$, where s_1 and s_2 join on attributes $\boldsymbol{B}_{1,2}$, while s_2 and s_3 on attributes $\boldsymbol{B}_{2,3}$. Since s_2 participates in two joins, we shall use two different parameters J_2^L and J_2^R to indicate the number of distinct values in s_2 for $\boldsymbol{B}_{1,2}$ and, respectively, $\boldsymbol{B}_{2,3}$ (and similarly for the random variables j_2^L, j_2^R and their expectations $\bar{j}_2^L$, $\bar{j}_2^R$). Analogously, Q_2^L and Q_2^R indicate the average number of times the same tuple of values for the join attributes $\boldsymbol{B}_{1,2}$, respectively, $\boldsymbol{B}_{2,3}$ occurs in s_2.

Let us consider $s_1 \bowtie s_2$ as a single service $s_{1\bowtie 2}$, whose depth

$$n_{1\bowtie 2} = j_{12}(n_1, n_2) \cdot q_1(n_1) \cdot q_2(n_2) \quad (31)$$

is the number of (partial) combinations formed between s_1 and s_2 with depths n_1 and n_2. In analogy with (7), the number of (full) combinations formed by sorted access is given by:

$$\mathcal{K}(n_{1\bowtie 2}, n_3) = j_{1\bowtie 2,3}(n_{1\bowtie 2}, n_3) \cdot q_{1\bowtie 2}(n_{1\bowtie 2}) \cdot q_3(n_3). \quad (32)$$

By taking the expectation of $\mathcal{K}$ as in (9), we obtain:

$$\bar{\mathcal{K}}(n_{1\bowtie 2}, n_3) \simeq \frac{n_{1\bowtie 2} n_3 J_{1\bowtie 2,3}}{J_{1\bowtie 2} J_3} \quad (33)$$

Note that J_3 is the number of distinct values in s_3 for the attributes $\boldsymbol{B}_{2,3}$ used in the join with $s_{1\bowtie 2}$, i.e., in the join with s_2. Symmetrically, $J_{1\bowtie 2}$ is the number of distinct values in $s_{1\bowtie 2}$ for the attributes $\boldsymbol{B}_{2,3}$ used in the join with s_3; however, this is generally only a subset of the number J_2^R of distinct values in s_2 for $\boldsymbol{B}_{2,3}$, and can be approximated as follows:

$$J_{1\bowtie 2} \simeq \frac{J_{1,2} Q_2^L}{N_2} J_2^R = \frac{J_{1,2}}{J_2^L} J_2^R \quad (34)$$

where $\frac{J_{1,2} Q_2^L}{N_2}$ is the fraction of tuples of s_2 participating in the join with s_1, so that multiplying by J_2^R gives an approximation of the number of distinct values that can be used in the join with s_3. Also, $J_{1\bowtie 2,3}$ is the number of distinct join attribute values in common between $s_{1\bowtie 2}$ and s_3. By a similar argument, we have:

$$J_{1\bowtie 2,3} \simeq \frac{J_{1,2}}{J_2^L} J_{2,3} \quad (35)$$

By replacing $n_{1\bowtie 2}$ in (33) by $\frac{n_1 n_2 J_{1,2}}{J_1 J_2^L}$, according to (9), we obtain

$$\bar{\mathcal{K}}(n_1, n_2, n_3) \simeq \frac{n_1 n_2 n_3 J_{1,2} J_{1\bowtie 2,3}}{J_1 J_2^L J_3 J_{1\bowtie 2}} = \frac{n_1 n_2 n_3 J_{1,2} J_{2,3}}{J_1 J_2^L J_2^R J_3} \quad (36)$$

Equation (36) characterizes the objective function of the optimization problem to solve in this case and is straightforwardly generalized to $M > 3$ services. Note that the factor $\frac{J_{1,2} J_{2,3}}{J_1 J_2^L J_2^R J_3}$ represents the selectivity of the join, which could also be estimated by using techniques known from the literature [16].

The execution and the stopping condition of the algorithm are unchanged, with the proviso that, when a new join attribute value is extracted, say, from s_1, the corresponding random access is performed on "service" $s_{2\bowtie 3}$. This requires activating a "chain" of random accesses: first a random access in made on s_2, which will possibly extract some tuples and thus join attribute values for s_3; then, for each such value (if any), a random access is made on s_3. With this in mind, we can now characterize the cost constraint. The random accesses due to the depth n_1 on s_1 incur the following expected cost:

$$rc_2 \bar{j}_1(n_1) + rc_3 \bar{j}_2^R(\bar{j}_1(n_1) Q_2^L) \quad (37)$$

where $\bar{j}_1(n_1) Q_2^L$ is the expected number of tuples extracted from s_2 by random accesses corresponding to the first n_1 tuples of s_1. Overall, the expected cost is as follows:

$$\begin{aligned} \bar{C}(n_1, n_2, n_3) \quad = \quad & sc_1 n_1 + sc_2 n_2 + sc_3 n_3 + \\ & rc_2 \bar{j}_1(n_1) + rc_3 \bar{j}_2^R(\bar{j}_1(n_1) Q_2^L) + \\ & rc_2 \bar{j}_3(n_3) + rc_1 \bar{j}_2^L(\bar{j}_3(n_3) Q_2^R) + \\ & rc_1 \bar{j}_2^L(n_2) + rc_3 \bar{j}_2^R(n_2) \end{aligned} \quad (38)$$

By substituting $\bar{j}_1$, $\bar{j}_2^L$, $\bar{j}_2^R$, and $\bar{j}_3$ with the approximation given in (16), one gets an expression of $\bar{C}(n_1, n_2, n_3)$ in which the terms are linear fractional in all the n_i's as shown in Appendix D. Thus the corresponding optimization problem

$$\begin{array}{ll} \text{maximize} & \bar{\mathcal{K}}(n_1, n_2, n_3) \\ \text{subject to} & \bar{C}(n_1, n_2, n_3) \leq C_T \\ & n_i \in \mathbf{N} \cap [0, N_i] \end{array} \quad (39)$$

can be rewritten as a convex problem and solved in the same way as Problem (17), by relaxing the integer constraint and finding the solution that satisfies the Karush-Kuhn-Tuker conditions for the relaxed problem. The problem remains convex even with $M > 3$ services.

A special case admitting a concise formulation is that of M-ary joins over the same join attributes. Here, after performing sorted accesses to s_i, we just proceed with the corresponding random accesses to all the other services s_m, $m = 1, \ldots, M$, $m \neq i$. The stopping condition remains unaltered. Therefore, we seek the solution of the optimization problem:

$$\begin{array}{ll} \text{maximize} & J_{1,2,\ldots,M} \cdot \prod_{i=1}^{M} \frac{n_i}{J_i} \\ \text{subject to} & \sum_{i=1}^{M} \left(sc_i n_i + j_i(n_i) \sum_{m \neq i} rc_m \right) \leq C_T \\ & n_i \in \mathbf{N} \cap [0, N_i] \end{array} \quad (40)$$

where $J_{1,2,\ldots,M}$ denotes the number of distinct join attributes values in common to the M services.

A.2 Handling parallel access

The cost model in (10) can be revised when time is used as cost metrics and the services can be accesses in

parallel. In a parallel implementation, sorted access is executed at the same time in both services. Then, whenever a new join attribute value is found, the corresponding random access is immediately made to the other service, and all random accesses can be made in parallel. In this case, the execution strategy is straightforward, as it corresponds to a straight line with slope sc_2/sc_1 in the n_1/n_2 plane.

Therefore, we consider a more interesting scenario in which services can be accessed in parallel, but sorted and random access to the same service cannot be parallelized. In this case, the overall cost can be expressed as

$$C(n_1, n_2) = \max\{sc_1 n_1 + rc_1 j_1(n_2), sc_2 n_2 + rc_2 j_2(n_1)\} \quad (41)$$

Thus, by replacing the expected value of (41) in (5), and relaxing the integer constraint, we seek the solution of the following problem (42):

$$\begin{aligned} \text{maximize} \quad & \log(n_1 n_2) \\ \text{subject to} \quad & t \le C_T \\ & sc_1 n_1 + rc_1 \bar{j}_1(n_2) \le t \\ & sc_2 n_2 + rc_2 \bar{j}_2(n_1) \le t \end{aligned} \quad (42)$$

where the dummy variable t is introduced in such a way that all constraints are expressed as linear fractional functions (hence convex). Problem (42) can be efficiently solved, similarly to problem (18), by imposing the Karush-Kuhn-Tucker conditions.

For a given value of C_T, sorted accesses are executed in parallel, until the desired number of tuples are retrieved. Note that this might cause the two services to stop sorted accesses asynchronously. Then, the corresponding random accesses are made, one after the other. By varying C_T a curve is traced in the n_1/n_2 plane, describing the execution strategy to be followed until the top-k combinations can be reported.

APPENDIX B MORE ON PROBLEM FORMULATION AND SOLUTION

B.1 Equivalence of problems (4) and (5)

When solving (4), we minimize the cost needed to obtain at least $\mathcal{K}_T$ good combinations. As for the solution of (5), we relax the constraint, and seek the solution of the problem

$$\begin{aligned} \text{minimize} \quad & sc_1 n_1 + sc_2 n_2 + rc_2 \tfrac{n_1}{a_1 n_1 + b_1} + rc_1 \tfrac{n_2}{a_2 n_2 + b_2} \\ \text{subject to} \quad & \log(n_1 n_2) \ge \log(\mathcal{K}_T J_1 J_2 / J_{1,2}) \end{aligned} \quad (43)$$

The corresponding Lagrangian function is

$$\begin{aligned} \bar{L}(n_1, n_2, \bar{\lambda}) = & sc_1 n_1 + sc_2 n_2 + rc_2 \frac{n_1}{a_1 n_1 + b_1} + rc_1 \frac{n_2}{a_2 n_2 + b_2} \\ & - \bar{\lambda}(\log(n_1 n_2) - \log(\mathcal{K}_T J_1 J_2 / J_{1,2})) \end{aligned} \quad (44)$$

We observe that, by setting $\lambda = 1/\bar{\lambda}$, the Karush-Kuhn-Tucker conditions of (43) lead to the same solution as for (18). Indeed, there is a one-to-one mapping between the target cost C_T and the number $\mathcal{K}_T$ of good combinations sought.

B.2 Probability of finding top k combinations

In some applications, it might be interesting to evaluate the probability, which we denote $P_a(n_1, n_2; \mathcal{K})$, of finding at least $\mathcal{K}$ good combinations after $\langle n_1, n_2 \rangle$ sorted accesses.

In (7) we wrote the number of good combinations $\mathcal{K}(n_1, n_2)$ in terms of the random variables $j_{12}(n_1, n_2)$, $q_1(n_1)$ and $q_2(n_2)$. In Appendices C.1 and C.2 we give exact expressions for the p.m.f. of, respectively, $j_i(n_i)$ and $j_{12}(n_1, n_2)$, and $q_i(n_i)$ is determined as $n_i/j_i(n_i)$. Since $j_{12}(n_1, n_2)$ is defined over the interval $\mathbf{N} \cap [0, J_{1,2}]$, and $q_i(n_i)$ over the interval $\mathbf{N} \cap [0, Q_i]$, the resulting p.m.f. $p(\mathcal{K}; n_1, n_2)$ has support $\mathbf{N} \cap [0, J_{1,2} Q_1 Q_2]$ and can be evaluated numerically. The probability $P_a(n_1, n_2; \mathcal{K})$ of finding at least $\mathcal{K}$ good combinations is given by 1 minus the sum of $p(\mathcal{K}; n_1, n_2)$ over $[0, \mathcal{K} - 1]$.

For the example shown in Table 1, the probability of finding at least $\mathcal{K}_T = 5$ good combinations after $\langle n_1, n_2 \rangle = \langle 6, 5 \rangle$ sorted accesses is equal to 0.55.

Although the analytical expression of $P_a(n_1, n_2; \mathcal{K})$ is too complex to be dealt with in optimization, we can still obtain a point-wise value of $P_a(n_1, n_2; \mathcal{K})$ for a given pair $\langle n_1, n_2 \rangle$. Therefore, upon solving problem (5), we can numerically evaluate $P_a(n_1^*, n_2^*; \mathcal{K}_T)$ for a target value $\mathcal{K}_T$.

APPENDIX C DERIVATIONS OF P.M.F.'S

C.1 Derivation of the p.m.f. of $j_i(n_i)$

The p.m.f. of the random variable $j_i(n_i)$ can be obtained by means of the following recursive equations:

$$p(j_i = 0; n_i = 0) = 1 \quad (45)$$

$$p(j_i; n_i = 0) = 0 \text{ for } 1 \le j_i \le J_i \quad (46)$$

$$\begin{aligned} p(j_i; n_i) = & p(j_i; n_i - 1) \cdot \frac{Q_i \cdot j_i - (n_i - 1)}{N_i - (n_i - 1)} \\ & + p(j_i - 1; n_i - 1) \cdot \frac{N_i - Q_i \cdot (j_i - 1)}{N_i - (n_i - 1)} \end{aligned} \quad (47)$$

where (47) indicates that in order to have j_i distinct join attribute values in n_i tuples obtained by sorted access we can have two cases: 1) we already have j_i distinct join attributes values after $n_i - 1$ accesses and the next tuple does not contain an unseen value of the join attribute; 2) we have $j_i - 1$ distinct join attributes value after $n_i - 1$ accesses and the next tuple contains an unseen value of the join attribute.

C.2 Derivation of the p.m.f. of $j_{12}(n_1, n_2)$

Let x denote a discrete random variable whose p.m.f. is described by the hyper-geometric distribution, which is defined as follows:

$$h(x, M, m_1, m_2) = \frac{\binom{m_1}{x}\binom{M - m_1}{m_2 - x}}{\binom{M}{m_2}} \quad (48)$$

The hyper-geometric distribution captures the probability that, given a set of M elements and two subsets of, respectively m_1 and m_2 elements, these subsets share exactly x elements in common.

Let l_i denote the number of distinct join attribute values in common with the other service (see Figure 3). If the l_i's are deterministically known, then the p.m.f. of j_{12} is characterized by the hyper-geometric distribution:

$$p(j_{12}; l_1, l_2) = h(j_{12}, J_{1,2}, l_1, l_2) \quad (49)$$

Since the l_i's are random variables with p.m.f. $p(l_i|j_i; n_i) = h(l_i, J_i, J_{1,2}, j_i)$, the conditional p.m.f. of j_{12} can be expressed by

$$\begin{aligned} &p(j_{12}|j_1, j_2; n_1, n_2) = \\ &= \sum_{l_1=0}^{J_{1,2}} \sum_{l_2=0}^{J_{1,2}} p(l_1|j_1; n_1) p(l_2|j_2; n_2) h(j_{12}, J_{1,2}, l_1, l_2) \end{aligned} \quad (50)$$

Finally, since j_1 and j_2 are random variables, we find

$$\begin{aligned} &p(j_{12}; n_1, n_2) = \\ &= \sum_{j_1=0}^{J_1} \sum_{j_2=0}^{J_2} p(j_1; n_1) p(j_2; n_2) p(j_{12}|j_1, j_2; n_1, n_2) \end{aligned} \quad (51)$$

In order to obtain an expression that can be handled in optimization, we can obtain the expected value of j_{12}, i.e., $\bar{j}_{12} = E[j_{12}]$, in terms of the expected values of j_1 and j_2, i.e., $\bar{j}_1$ and $\bar{j}_2$, respectively. Indeed, since the expectation of the hyper-geometric distribution in (48) is given by $E[x] = \frac{m_1 m_2}{M}$, we obtain

$$\bar{l}_i(n_i) = E[p(l_i|\bar{j}_i; n_i)] = \bar{j}_i(n_i) \frac{J_{1,2}}{J_i} \quad (52)$$

Similarly, from (51) we find that $\bar{j}_{12}$ is equal to

$$\bar{j}_{12} = \frac{\bar{j}_1(n_i) \bar{j}_2(n_2) J_{1,2}}{J_1 J_2} \quad (53)$$

C.3 Derivation of the p.m.f. of $\tilde{q}_i(n_i)$

The p.m.f. of the random variable $\tilde{q}_i(n_i)$, which is defined for $n_i > 0$, can be written recursively, using an approach similar to the one used to derive $p(j_i; n_i)$:

$$p(\tilde{q}_i = 0; n_i) = \prod_{t=1}^{n_i} \left(1 - \frac{Q_i}{N_i - t + 1}\right) \quad (54)$$

$$p(\tilde{q}_i = 1; n_i = 1) = \frac{Q_i}{N_i} \quad (55)$$

$$p(\tilde{q}_i; n_i = 1) = 0 \text{ for } 2 \le \tilde{q}_i \le Q_i \quad (56)$$

$$\begin{aligned} p(\tilde{q}_i; n_i) &= p(\tilde{q}_i - 1; n_i - 1) \cdot \frac{Q_i - (\tilde{q}_i - 1)}{N_i - (n_i - 1)} \\ &+ p(\tilde{q}_i, n_i - 1) \cdot \frac{N_i - (n_i - 1) - (Q_i - \tilde{q}_i)}{N_i - (n_i - 1)} \end{aligned} \quad (57)$$

The expected value of $p(\tilde{q}_i; n_i)$ is given by $Q_i \frac{n_i}{N_i}$; intuitively, each join attribute value contributes Q_i tuples, but only a fraction $\frac{n_i}{N_i}$ of them are extracted in the top n_i tuples.

Appendix D
Convexity of cost function with multiple services

We show that the expression characterizing the cost (38) is convex. Indeed, the first three terms are linear. The remaining terms can be rewritten by using the approximation given in (16). All the resulting terms are linear fractional (i.e., a ratio of two linear functions in n_i), and thus convex as long as the denominator is positive, which is always the case here. Trivially, each factor of the form $\bar{j}_i(n_i)$, $\bar{j}_i^L(n_i)$, or $\bar{j}_i^R(n_i)$ is rewritten as a linear fractional expression in n_i by (16). As for the two terms with a nesting of applications of $\bar{j}$, consider, e.g.,

$$rc_3 \bar{j}_2^R(\bar{j}_1(n_1) Q_2^L) \quad (58)$$

By using (16) on $\bar{j}_2^R$, this rewrites into

$$rc_3 \frac{\bar{j}_1(n_1) Q_2^L}{a_2^R \bar{j}_1(n_1) Q_2^L + b_2^R} \quad (59)$$

where $a_2^R = \frac{Q_2^R - 1}{N_2 - 1}$ and $b_2^R = 1 - a_2^R$. Then, by using (16) on $\bar{j}_2$, this rewrites into

$$rc_3 \frac{\frac{n_1}{a_1 n_1 + b_1} Q_2^L}{a_2^R \frac{n_1}{a_1 n_1 + b_1} Q_2^L + b_2^R} = \frac{rc_3 Q_2^L n_1}{(a_1 b_2^R + a_2^R Q_2^L) n_1 + b_1 b_2^R} \quad (60)$$

which is linear fractional in n_1, with a positive denominator, since all its components are positive, and thus convex. The argument is analogous for $rc_1 \bar{j}_2^L(\bar{j}_3(n_3) Q_2^R)$. Finally, the sum of convex functions is a convex function.

The same argument straightforwardly generalizes to cost functions with more than two levels of nesting of applications of $\bar{j}$, which occur when joining more than three services. Therefore, the cost function remains convex even with more than three services.