

Cost-aware top-k join algorithms

Stefano Ceri Davide Martinenghi Marco Tagliasacchi

Dipartimento di Elettronica e Informazione – Politecnico di Milano
Piazza Leonardo da Vinci, 32 – 20133 Milano, Italy
{ceri,martinen,tagliasa}@elet.polimi.it

June 11, 2009

Abstract

In this paper we consider the problem of joining ranked results produced by search services on the Web. We consider services that output data tuples sorted by score. We cast this problem as a generalization of the traditional rank aggregation problem, i.e., combining several ranked lists of objects to produce a single consensus ranking. The ranked lists may contain a high number of objects, typically presented in pages, and accessing such pages is costly. Therefore, algorithms should determine the best overall results without accessing all the objects. Besides pagination of results, additional difficulties with respect to rank aggregation include the fact that the ranked lists do not, in general, rank the same set of objects, but rather heterogeneous sets of objects that may match on given join attributes to form a join tuple (here called combination). Each combination is assigned a combined score, which is a function of the individual scores.

We propose a model that takes into account the different ways and costs of accessing the services, and an algorithm that generates a cost-aware access strategy to the different services involved in the join so that, at each step of the execution, the expectation to obtain the combinations with the top combined scores is maximized under a cost constraint on the available computational resources. We validate our strategy with experiments conducted on real-world services as well as on synthetically generated datasets.

1 Introduction

Domain-expert search engines are becoming widespread on the Web; their focus upon a specific domain gives them greater capabilities and credibility over general-purpose search engines, but also limits their query answering spectrum just to narrow queries over specific domain knowledge. A new trend in search-oriented research will overcome such limitation, by allowing the co-operation and orchestration of search services in order to answer complex, multi-domain queries. Such new Web programming paradigm, called search computing¹, extends service-oriented architectures by enabling the building of software systems in which ranking is one the main factors for composing services.

Examples of multi-domain queries include the search for conference locations which are serviced by low-cost flights and are near to luxury hotels of given hotel chains offering special packages; or the search for hospitals with world-wide expertise in given speciality combinations; or the search for the best city hosting a road game where a fan of a soccer team can find at the same time low-cost flights and rock concerts. For answering such queries, an integrator system must be capable of understanding the user's goals, express them in a suitable format of sub-goals such that each subgoal addresses a specific

¹Search Computing is an IDEAS Advanced Grant awarded by ERC: <http://www.search-computing.com/>

domain, extract and combine the answers of domain-specific search systems, and then globally rank each combination by means of an aggregation function, so as to present to the user the answers with the top combined scores. This paper is neither concerned with query understanding nor with query decomposition, and assumes that a query is expressed as a top-k join query, i.e., a conjunctive query such that each query atom is routed to a specific search engine and the results are joined to form combinations. In such well-defined scenario, the paper presents a model that takes into account the different ways of accessing the services, each characterized by its access cost, and an algorithm that generates a cost-aware access strategy to services producing the top k combinations with the highest combined scores.

We consider services that may be asked to output data tuples sorted by score, where the score is an explicit field of the tuples. The ranked lists may contain a large number of objects, typically presented in pages, and accessing such pages is costly. Moreover, objects can be accessed according to various methods, that can be broadly classified as sorted, producing a possibly very long ranked list of objects (such that the list’s tail is typically not retrieved by the join method), or attribute-based, producing a narrower set of objects, normally not ranked, which satisfy a selection over the attributes. The top-k join problem has been dealt with in the literature (e.g., [13]) by extending rank aggregation algorithms to the case of join in the setting of relational databases. In particular, [13] extends the threshold algorithm (TA) [9], which, similarly, considers sorted and random accesses; TA, in turn, refines the \mathcal{A}_0 algorithm proposed by Fagin [6] (hereafter denoted FA) by using a better stopping condition. However, previous approaches to top-k join queries, including [13], did not consider some of the distinguishing features of search engines on the Web, namely different access costs.

This paper defines the problem of optimal extraction of top k combinations in our setting, where heterogeneous access costs are to be expected, and then presents its solution. The analytic method for producing the solution is then used to devise an execution strategy that, at each step, selects the next search service to be invoked in order to maximize the expected number of top combinations that can be formed, while explicitly taking access costs into account. In our analysis, we focus on the join of two search services and then extend our results to M -ary joins, with $M > 2$.

We demonstrate our approach at work over two domain-specific Web services. The former, `www.venere.com`, allows listing hotels according to several rankings; we focus on hotels in Paris and use star rating, presenting hotels in decreasing “number of stars”. The latter, `www.eatinparis.com`, extracts good restaurants in Paris, ranked by customer ratings. We join services by street, thus serving a user query whose goal is to find a hotel-restaurant combination in the same Paris street, by maximizing hotel luxury and food excellence. This example shows the potential spreading of multi-domain queries and the concrete possibility of improving query results over generalized search services. We also evaluate the efficiency of the proposed cost-aware access strategy by testing it on synthetically generated data, showing that a cost reduction of approximately 30% can be obtained on average by explicitly taking access costs into account.

The main contributions of the paper can be summarized as follows. (i) We provide a statistical characterization of the cardinalities of the different sets of tuples involved in the top-k join problem, such as the number of combinations and the number of distinct values for the join attributes. (ii) We extend the FA algorithm to the search engine join context so as to solve the top-k join problem. Previous extensions were based on different algorithms, such as TA, and therefore required explicit assumptions on the score distributions in the different services. (iii) We determine, at query formulation time, a *cost-aware* access strategy, based on the FA algorithm, to compute the top k combinations. Such strategy leverages parameters characterizing the involved services that are typical of the Web context, among which pagination of results and different access costs, both per service and per access type. Unlike previous works, we do not require any assumption on the score distributions. Moreover, the proposed access strategy can be adopted regardless of the specific criterion (e.g., FA’s or TA’s stopping condition) used to halt the algorithm when the top k combinations are guaranteed to be found. (iv) We test our

strategy with experiments that validate our approach and show its practicability.

2 The Search Computing Model

In this section, we introduce the setting of search computing based on the service model used by [21, 3], which we use as our starting point. Then we formulate the top-k join problem in this setting.

2.1 A Model for Search Services

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. Such measure may be either hidden or presented to the user along with the results as a score. In the latter case, we say that the service s assigns a score (usually in \mathbb{R} or in the $[0, 1]$ interval) to each tuple it produces.

We consider queries that join two search services, s_1 and s_2 , and assign, to each pair of tuples (henceforth called combination), t_1 from s_1 and t_2 from s_2 , a new combined score according to a given aggregation function that takes into account the scores assigned to t_1 by s_1 and to t_2 by s_2 . Given such a query, the problem we address is how to generate in the most convenient way the k combinations with highest combined score (top k answers).

In the following, without loss of generality, we shall assume each service s involved in a join to have a signature of the form $s(\mathbf{A}, \mathbf{B}, S)$, where \mathbf{A} is a set of attributes specific for service s , \mathbf{B} is a set of attributes used in the join, and S is an attribute carrying the score for the tuples output by s .

Example 2.1 The services considered in the running example of this paper for the join on the same street can be characterized by the following signatures

- $\text{hotel}(\text{HotelName}, \text{Street}, \text{Stars})$ as service s_1 , listing name, street, and stars for Parisian hotels ranked by stars in descending order.
- $\text{rest}(\text{RestName}, \text{Street}, \text{Rating})$ as service s_2 , listing name, street, and rating for Parisian restaurants serving French cuisine, ranked by customers' rating in descending order.

■

2.1.1 Query plans and joins

As in [21, 3], by *query plan* we indicate a sequence of invocations of services and their composition through joins. Additionally, [3] requires every join in the plan to carry an indication of the join strategy to be employed during execution. This includes the expected number of invocations to be performed on each of the joined services as well as the strategy for exploring the results of the joined services in order to compose the results of the join and rank them.

Let \underline{s} be the extension of service s , i.e., the relational set of all tuples that can be returned by s , and let f be an *aggregation function*, i.e., a function mapping a pair of scores into a new score called *combined score*. As in [6], we restrict to queries whose associated aggregation function is monotone, which are best suited for efficient query answering algorithms. A binary aggregation function f is monotone if

$$f(x_1, x_2) \leq f(x'_1, x'_2) \text{ if } x_1 \leq x'_1 \text{ and } x_2 \leq x'_2.$$

Consider now the join between the results of two services s_1 and s_2 . Let $t[A]$ indicate, as usual, the value of tuple t for attribute A . Any tuple t in $s_1 \bowtie_{B_1=B_2} s_2$ adorned with an extra attribute carrying the value $f(t[S_1], t[S_2])$ is called a *combination* from the join between s_1 and s_2 on S_1 and S_2 using f .

Example 2.2 The services of Example 2.1 can be joined, e.g., on the `Street` attribute in order to obtain hotel-restaurant combinations in the same street of Paris. The combination will comprise all the attributes of `hotel` and `rest`, plus an extra attribute for the combined score that is evaluated via an aggregation function combining stars and customers' rating. A possible example of such a function is one that normalizes `Stars` and `Rating` in the $[0, 1]$ interval and takes their product². ■

2.1.2 Access patterns

Unlike data in the relational setting, 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. In order to model such access limitations, we assume that each service is characterized by a given number of combinations of input and output parameters, called *access patterns*, corresponding to the different ways in which it can be invoked.

In [6], the author considers two main ways of accessing elements produced by an information system: *i*) with a sorted access, by requesting, say, the top 10 elements in sorted order, along with their scores, and then the next 10, etc.; *ii*) with a random access, i.e., passing a given element and receiving its score in return. Access patterns allow formulating a similar distinction also in the context of search services, which, however, return tuples of values instead of single elements as in the information systems described in [6]. Accordingly, in this paper we shall assume that each search service can be invoked according to the following two access patterns, where an \circ superscript over an attribute or attribute set indicates that it is an output field, and an \dagger superscript indicates that it is an input field:

- *Sorted access*: $s(A^\circ, B^\circ, S^\circ)$, where all the fields are given as output and no input is required, as in a traditional relational table. The first sorted access returns the first page of tuples sorted by score in descending order, the next access returns the following page, and so on.
- *Attribute-based access*: $s(A^\circ, B^\dagger, S^\circ)$, where a sub-tuple of values for the join attributes B is given as input, and tuples of values for the other attributes are returned as output.

2.2 Parameters Characterizing Search Services

2.2.1 Cardinality and value distribution

Each search service s_i has an associated *cardinality* $N_i = |s_i|$, expressing the total number of results that it can produce.

Another important aspect to consider when dealing with a search service s_i involved in a join is the distribution of values for the join attributes B_i with respect to the values for the service-specific attributes A_i . Together with N_i , we consider two parameters that give a first characterization of this aspect:

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

²Such a function is monotonic in x_1 and x_2 . Reference [6] provides more examples of monotonic aggregation functions.

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 sub-tuples for the join attributes B_1 are covered by s_2 and vice versa. To this end, we shall indicate with $J_{1,2}$ the number of distinct sub-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 `rest` in our previous example).

2.2.2 Pagination

According to [3], services can be further classified as *bulk* or *chunked*; in the former case, they return all their results with a single request, in the latter case they return results in pages (“chunks”) of a fixed size. We observe that search services are generally chunked, since they typically return a large number of results. In our model, we characterize each chunked service s_i with a *page size* \mathcal{P}_i , which expresses the number of tuples returned by each “fetch” (i.e., each sorted invocation of a page of results) performed on s_i .

2.2.3 Costs

We associate each request sent to a search service with an expected cost of invoking it. Such cost may differ for different search services, and also depending on the different access patterns 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 *attribute-based access cost* ac_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, ac_1, ac_2 so as to 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 number of sorted and attribute-based accesses to be made.

The parameters used in the remainder of the paper are summarized in Figure 1. Based on these, we can now state the main problem that we address in this paper. The top-k join problem consists in determining the k combinations with the highest combined scores when joining the results of two search services. Finding a *cost-aware* solution to the problem means devising an optimal execution strategy with respect to the access costs.

<p>sc_i: the cost of a sorted access to s_i.</p> <p>ac_i: the cost of an attribute-based access to s_i.</p> <p>k: the number of top combinations to be determined in the join between s_1 and s_2.</p> <p>N_i: the number of tuples that can be output by s_i.</p> <p>J_i: the number of distinct sub-tuples of values for the join attributes B_i of s_i.</p> <p>$J_{1,2}$: the number of distinct sub-tuples of values for the join attributes B_1 and B_2 in common between s_1 and s_2.</p> <p>Q_i: the average number N_i/J_i of times the same sub-tuple of values for the join attributes B_i occurs in s_i.</p>
--

Figure 1: Summary of notation

3 Finding the top k combinations

In this section we first introduce a well-known algorithm for top-k query answering in the closely related field of rank aggregation. Then we extend and adapt this algorithm to the context of top-k join queries over search services.

3.1 An Algorithm for Rank Aggregation

Rank aggregation is the problem of combining several ranked lists of items in a robust way to produce a single consensus ranking of the items [7]. In the context of rank aggregation, queries are posed over ranked lists of items. These resemble the ranked results output by a search service, but it is usually assumed that all the ranked lists are permutations of one another, i.e., the set of items is the same in all ranked lists. In our context, instead, the tuples output by search services are substantially different in different search services, since they cover different attributes of which only a subset (the join attributes) may be in common.

In rank aggregation, the semantics of a *query* $Q(L_1, L_2)$ combining two ranked lists L_1, L_2 can be defined in terms of an associated binary aggregation function f_Q :

$$Q(L_1, L_2) = \{\langle x, \sigma_Q(x) \rangle \mid x \in \underline{L}_1 \cup \underline{L}_2\} \quad (1)$$

where \underline{L}_i indicates the set of items in list L_i , and $\sigma_Q(x)$ is the combined score of x evaluated as $f_Q(\sigma_{L_1}(x), \sigma_{L_2}(x))$, where $\sigma_{L_i}(x)$ is the score of x in L_i . Variations exist to the above definition (1) in that sometimes it is not desirable to include in the query results items that are not present in all ranked lists. It is typically assumed that if an item x does not occur in L_i then $\sigma_{L_i}(x)$ equals the lowest score possible for L_i .

A well-known algorithm for determining the top k answers to queries with an associated monotone aggregation function is due to Fagin [6]. Such algorithm, commonly referred to as FA, assumes that ranked lists can be accessed both with a sorted access (items are retrieved one by one in ranking order) and with a random access (by requesting the score of a given item). For convenience, let X_n^i denote the set of the n items with highest score in the ranked list L_i . Algorithm FA for the case of two ranked lists is shown in Figure 2. It consists of three phases: (i) the top n items are retrieved from every ranked list by sorted accesses, and n is the minimum number such that there are at least k items common to every ranked list; (ii) scores are determined for every item extracted during phase (i) at every ranked list by random accesses; (iii) the overall score is computed for every extracted item and the top k are output.

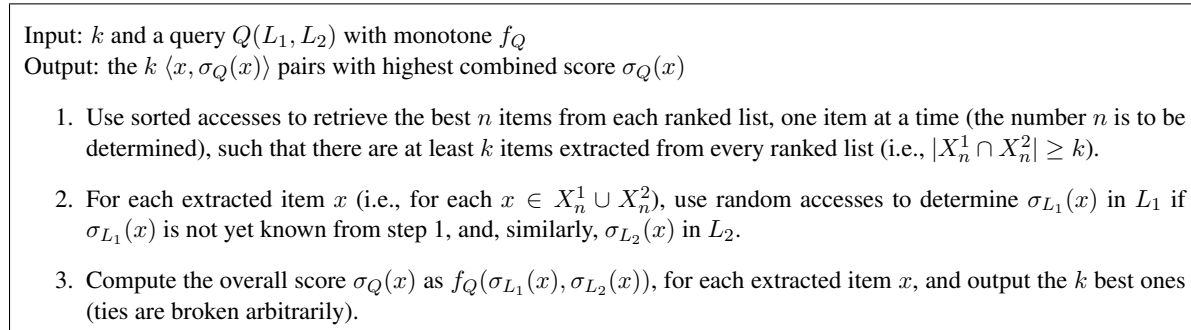


Figure 2: Fagin’s algorithm (FA)

For performance evaluation the author refers to the so-called *middleware cost*, i.e., a linear combination of the total number of sorted accesses and the total number of random accesses executed by FA. It is shown in [6] that, if the two rankings are independent and over the same set of N items, the middleware cost for finding the top k answers (item-score pairs) is $O(\sqrt{N \cdot k})$ with arbitrarily high probability, which is sub-linear in N .

3.2 An Algorithm for the Top-k Join Problem

We now describe an adaptation of algorithm FA to the context of search services that allows determining the top k combinations from a join. The main differences are that *i*) we explore the combinations that can be formed between the two services, instead of items common to the two ranked lists, and that *ii*) the number of sorted accesses is not necessarily the same for the two services, while in FA there is a common number n of sorted accesses to both ranked lists. The former point reflects the different nature of the problem, in which k is the desired number of combinations, not items; the latter point, mentioned in [6] as a possible refinement to further reduce the random accesses, allows us instead to capture the fact that different services may have different access costs and thus it may be convenient to access them differently. We describe a top-k join algorithm in Figure 3, that we call FA-join.

Let us indicate, for convenience, with

- $\mathcal{S}_{n_i}(s_i)$ the set of top n_i tuples in s_i .
- $\mathcal{K}_f(\mathcal{X}_1, \mathcal{X}_2)$ the set of combinations that can be formed between the tuples in a set of tuples $\mathcal{X}_1 \subseteq s_1$ and those in a set of tuples $\mathcal{X}_2 \subseteq s_2$ according to an aggregation function f , i.e.,

$$\mathcal{K}_f(\mathcal{X}_1, \mathcal{X}_2) = \{ \langle t_1, t_2, f(t_1[S_1], t_2[S_2]) \rangle \mid t_i \in \mathcal{X}_i \text{ and } t_1[\mathbf{B}_1] = t_2[\mathbf{B}_2] \}.$$

- $\mathcal{J}_i(n_i)$ the set of sub-tuples of values for the join attributes in the top n_i tuples of s_i , i.e.,

$$\mathcal{J}_i(n_i) = \{t_i[\mathbf{B}_i] \mid t_i \in \mathcal{S}_{n_i}(s_i)\}.$$

Input: k , two services $s_1(\mathbf{A}_1, \mathbf{B}_1, S_1)$, $s_2(\mathbf{A}_2, \mathbf{B}_2, S_2)$, and a monotone aggregation function f
Output: the k combinations with highest combined score

1. Use sorted accesses to retrieve the best n_1 tuples from s_1 and n_2 tuples from s_2 (n_1 and n_2 to be determined), such that at least k combinations are formed with the extracted tuples, i.e., $|\mathcal{K}_f(\mathcal{S}_{n_1}(s_1), \mathcal{S}_{n_2}(s_2))| \geq k$.
2. For each tuple $t_1 \in \mathcal{J}_1(n_1)$ (i.e., for each sub-tuple, extracted from s_1 , of values for the join attributes), make an attribute-based access to s_2 with $\mathbf{B}_2 = t_1$, and call \mathcal{A}_2 the set of all tuples obtained from s_2 in this way. Similarly, access s_1 with the tuples in $\mathcal{J}_2(n_2)$ and call \mathcal{A}_1 the set of obtained tuples.
3. Compute the set of all combinations formable with all the tuples extracted by attribute-based accesses (i.e., $\mathcal{K}_f(\mathcal{S}_{n_1}(s_1) \cup \mathcal{A}_1, \mathcal{S}_{n_2}(s_2) \cup \mathcal{A}_2)$). Output the k combinations with highest scores in the set (ties are broken arbitrarily).

Figure 3: FA-join algorithm

FA-join also consists of three phases: *(i)* sorted accesses are used to retrieve the top n_1 tuples from s_1 and the top n_2 from s_2 so that at least k combinations can be formed (note that n_1 and n_2 are not univocally determined, so FA-join simply identifies a class of $\langle n_1, n_2 \rangle$ pairs); *(ii)* attribute-based accesses are made to each service (s_1 and, respectively, s_2) with every sub-tuples of values for the join attributes extracted during phase (i) from the other service (s_2 and, respectively, s_1); *(iii)* all combinations that can be formed with all the tuples extracted during phases (i) and (ii) are formed (there may be more than k), ranked by their combined score, and the top k are output.

We now state that FA-join is correct, i.e., it outputs the top k combinations (if at least k combinations exist).

Theorem 3.1 *For every monotone aggregation function, FA-join returns the top k combinations.*

Proof. (Sketch) Consider n_1, n_2 (sorted accesses) such that $|\mathcal{K}_f(\mathcal{S}_{n_1}(s_1), \mathcal{S}_{n_2}(s_2))| \geq k$. The last step of the algorithm outputs the top k combinations in the set $\mathcal{K}' = \mathcal{K}_f(\mathcal{S}_{n_1}(s_1) \cup \mathcal{A}_1, \mathcal{S}_{n_2}(s_2) \cup \mathcal{A}_2)$ (see Step 2 for the definition of \mathcal{A}_i). Take now a combination $\langle t_1, t_2, f(t_1[S_1], t_2[S_2]) \rangle$ not in \mathcal{K}' . Then $t_1 \notin \mathcal{S}_{n_1}(s_1)$, so its score $t_1[S_1]$ cannot be greater than that of any of the tuples in $\mathcal{S}_{n_1}(s_1)$; similarly for t_2 . Then, by monotonicity of f , the combined score $f(t_1[S_1], t_2[S_2])$ cannot be greater than that of any of the combinations in $\mathcal{K}_f(\mathcal{S}_{n_1}(s_1), \mathcal{S}_{n_2}(s_2)) \subseteq \mathcal{K}'$. Hence, no combination not in \mathcal{K}' can be among the top k combinations, thus the top k combinations in \mathcal{K}' are the overall top k . \square

We note that, unlike algorithm FA (where random accesses were only required in case of missing score information), attribute-based accesses to s_1 are generally needed with every join sub-tuple extracted in the sorted access phase from s_2 , even if the sub-tuple was also extracted from s_1 , and similarly for the attribute-based accesses to s_2 .

We point out that a better stopping criterion can be implemented at run-time. Similarly to the HRJN (Hash Rank Join) method proposed in [13], the execution can be terminated earlier, yet guaranteeing that the top- k combinations can be formed, when the aggregated score of the k -th combination retrieved so far exceeds a threshold value. The threshold is adjusted adaptively at each execution step, and is set equal to the aggregated score computed based on the scores of the last tuples retrieved by the two services. In the following, we refer to this variant as TA-join (Threshold Algorithm join).

FA-join is proved correct for every pair of n_1 and n_2 satisfying the condition of Step 1, and TA-join for every pair satisfying the threshold condition. Therefore FA-join as well as TA-join actually identify *classes* of algorithms complying with the given conditions, the specific execution of which may be further refined. In the next section, we reformulate the top- k join problem as the problem of finding the pair $\langle n_1, n_2 \rangle$ that maximizes the number of combinations found after sorted accesses subject to a constraint on the costs of both sorted and attribute-based accesses made by the algorithm. In this sense, we propose an execution of FA-join that is “cost-aware” and consequently refer to it as CAFA-join (Cost-Aware FA-join). A simpler, naive way of executing FA-join consists in accessing the services in a “round-robin” fashion, e.g. alternating sorted accesses to s_1 and s_2 in a way that makes the numbers of retrieved tuples on s_1 and s_2 as close as possible to each other (recall that page sizes may be different).

Note that in Section 4 we shall formulate our cost-aware refinement on top of FA-join, because a mathematical characterization based on the more powerful but score-based TA-join would require knowledge on the score distributions at optimization time, which are typically unavailable in the context of services over the Internet. However, as explained in Section 5 and demonstrated in Section 6, our cost-aware refinement adapts as is to TA-join (obtaining CATA-join, i.e., Cost-Aware TA-join) with major gains in terms of performance w.r.t. a non-cost-aware execution of TA-join.

4 Optimizing the retrieval of the top k combinations

The algorithm described in Section 3 provides the guarantee of finding the top k combinations, given that at least k combinations are found after n_i sorted accesses to the service s_i . In a practical system, one might want to progressively increase the number of sorted accesses n_i , up to the point where at least k combinations can be formed. This would enable to construct the result incrementally, in such a way that the top \tilde{k} results, with $\tilde{k} < k$, can be retrieved and displayed before achieving the target number of combinations k . In other words, given a constraint on the available resources, i.e., on the costs involved in accessing the search services, one might devise an optimal strategy that enables to maximize the number of combinations that can be formed after the sorted access phase. More formally, this can be formulated

as the following optimization problem

$$\begin{aligned}
& \text{maximize} && \bar{\mathcal{K}}(n_1, n_2) \\
& \text{subject to} && \bar{C}(n_1, n_2) \leq C_T \\
& && n_i \in \mathbb{N} \cap [0, N_i]
\end{aligned} \tag{2}$$

where $\bar{\mathcal{K}}(n_1, n_2)$ denotes the expected number of combinations that can be formed by retrieving n_i tuples from search service s_i by means of sorted accesses, and $\bar{C}(n_1, n_2)$ is the associated expected cost of performing both sorted and attribute-based accesses, i.e., those needed to find the top k combinations. In both cases, the expectation is taken with respect to all the possible ways of composing the tuples returned by the search services, compatibly with the parameters that characterize them, namely J_i and Q_i .

In order to find a mathematically tractable solution of (2), we make the following simplifications:

- Each join attribute value occurs in service s_i exactly Q_i times, where $Q_i = N_i/J_i$, as previously defined. Our experiments of Section 6, run on realistic settings, show that our solution is effective even without this assumption.
- The ranking order of the tuples in a service is independent from the other service. This is a reasonable assumption in the setting considered in this paper, since we are joining the results of heterogenous search services, which operate on different domains.

In Section 4.1 we derive an expression for $\bar{\mathcal{K}}(n_1, n_2)$, while we obtain an expression of the cost function $\bar{C}(n_1, n_2)$ in Section 4.2.

4.1 Objective function formulation

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

1. Determine the distinct join attribute 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 and j_2 denote the cardinality of the two sets, i.e., $j_1 = 3$ and $j_2 = 3$.
2. Determine the join attribute values in common between s_1 and s_2 , i.e., $\{b^{(1)}, b^{(2)}, b^{(3)}\} \cap \{b^{(1)}, b^{(2)}, b^{(6)}\} = \{b^{(1)}, b^{(2)}\}$. Let j_{12} denote the cardinality of this set, i.e., $j_{12} = 2$.
3. For each join attribute in common, form the combinations. The number of combinations is equal to the product of the multiplicities $q_i^{(r)}$ of the join value $b^{(r)}$ in the first n_i tuples, i.e., $q_1^{(1)} \times q_2^{(1)} = 2 \times 2 = 4$ combinations based on the join value $b^{(1)}$, and $q_1^{(2)} \times q_2^{(2)} = 1 \times 1 = 1$ based on the join value $b^{(2)}$.

Table 1(c) shows the five combinations that can be formed (we omit the individual scores and we do not repeat the join attribute). The combined score has been computed as the minimum between the scores of the joined tuples, but any monotone aggregation function might be adopted.

The aforementioned deterministic example needs to be generalized to a statistical setting, where all the variables of the problems are to be considered discrete-valued random variables. In the remainder of the paper we adopt the following notation: let x denote a discrete-valued random variable and $p(x)$ its probability mass function. 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., $\bar{x} = E[x]$.

Table 1: (a) Tuples in s_1 (black: retrieved, grey: not retrieved). (b) Tuples in s_2 (black: retrieved, grey: not retrieved). (c) Combinations formed with sorted accesses. (d) Additional combinations formed with attribute-based accesses. (e) Top 5 combinations.

(a)			(b)			(c)				(d)				(e)			
A_1	B_1	S_1	A_2	B_2	S_2	A_1	A_2	B	S	A_1	A_2	B	S	A_1	A_2	B	S
$a_1^{(4)}$	$b^{(2)}$	77	$a_2^{(2)}$	$b^{(6)}$	90	$a_1^{(9)}$	$a_2^{(3)}$	$b^{(1)}$	53	$a_1^{(4)}$	$a_2^{(1)}$	$b^{(2)}$	41	$a_1^{(4)}$	$a_2^{(4)}$	$b^{(2)}$	57
$a_1^{(3)}$	$b^{(3)}$	72	$a_2^{(6)}$	$b^{(6)}$	70	$a_1^{(9)}$	$a_2^{(7)}$	$b^{(1)}$	53	$a_1^{(5)}$	$a_2^{(3)}$	$b^{(1)}$	6	$a_1^{(9)}$	$a_2^{(3)}$	$b^{(1)}$	53
$a_1^{(6)}$	$b^{(3)}$	63	$a_2^{(3)}$	$b^{(1)}$	58	$a_1^{(8)}$	$a_2^{(3)}$	$b^{(1)}$	32	$a_1^{(5)}$	$a_2^{(7)}$	$b^{(1)}$	6	$a_1^{(9)}$	$a_2^{(7)}$	$b^{(1)}$	53
$a_1^{(9)}$	$b^{(1)}$	53	$a_2^{(4)}$	$b^{(2)}$	57	$a_1^{(8)}$	$a_2^{(7)}$	$b^{(1)}$	32	$a_1^{(7)}$	$a_2^{(4)}$	$b^{(2)}$	27	$a_1^{(4)}$	$a_2^{(1)}$	$b^{(2)}$	41
$a_1^{(8)}$	$b^{(1)}$	32	$a_2^{(7)}$	$b^{(1)}$	57	$a_1^{(4)}$	$a_2^{(4)}$	$b^{(2)}$	57	$a_1^{(2)}$	$a_2^{(4)}$	$b^{(2)}$	4	$a_1^{(8)}$	$a_2^{(3)}$	$b^{(1)}$	32
$a_1^{(1)}$	$b^{(3)}$	31	$a_2^{(1)}$	$b^{(2)}$	41												
$a_1^{(7)}$	$b^{(2)}$	27	$a_2^{(5)}$	$b^{(7)}$	40												
$a_1^{(5)}$	$b^{(1)}$	6	$a_2^{(8)}$	$b^{(7)}$	35												
$a_1^{(2)}$	$b^{(2)}$	4															

1. For a given number of tuples n_i retrieved with sorted accesses, the number of distinct join attributes $j_i(n_i)$ is a random variable whose p.m.f. $p(j_i; n_i)$ can be determined as shown in Appendix A.
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 4, we want to determine the p.m.f. of j_{12} 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 A. For the sake of the following discussion, we note that the expected value of $j_{12}(n_1, n_2)$ is given by

$$\bar{j}_{12}(n_1, n_2) = \frac{\bar{j}_1(n_1)\bar{j}_2(n_2)J_{1,2}}{J_1J_2} \quad (3)$$

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

$$\mathcal{K}(n_1, n_2) = j_{12}(n_1, n_2) \cdot q_1(n_1) \cdot q_2(n_2) \quad (4)$$

where $q_i(n_i)$ denotes the number of tuples where the same value of a given join attribute occurs in service s_i , when n_i tuples have been accessed with sorted accesses. We notice that the random variables $q_i(n_i)$ and $j_i(n_i)$ are related by the expression

$$j_i(n_i) = \frac{n_i}{q_i(n_i)} \quad (5)$$

Replacing (5) in (4) and the random variables with their expected values we obtain

$$\bar{\mathcal{K}}(n_1, n_2) \simeq \frac{\bar{j}_1\bar{j}_2J_{1,2}}{J_1J_2} \frac{n_1}{\bar{j}_1} \frac{n_2}{\bar{j}_2} = \frac{n_1n_2J_{1,2}}{J_1J_2} \quad (6)$$

4.2 Cost constraint formulation

In order to find the top k combinations, the sorted access phase is followed by the attribute-based access phase. With reference to the example illustrated in Table 1, we need to retrieve the tuples in s_2 whose join attribute value appeared at least once in the first n_1 tuples of s_1 , and vice versa. Table 1(d) shows the additional combination formed by performing an access to s_2 based on the join attribute $b^{(2)}$ and the four additional combinations formed by performing two accesses to s_1 based on the join attributes $b^{(1)}$

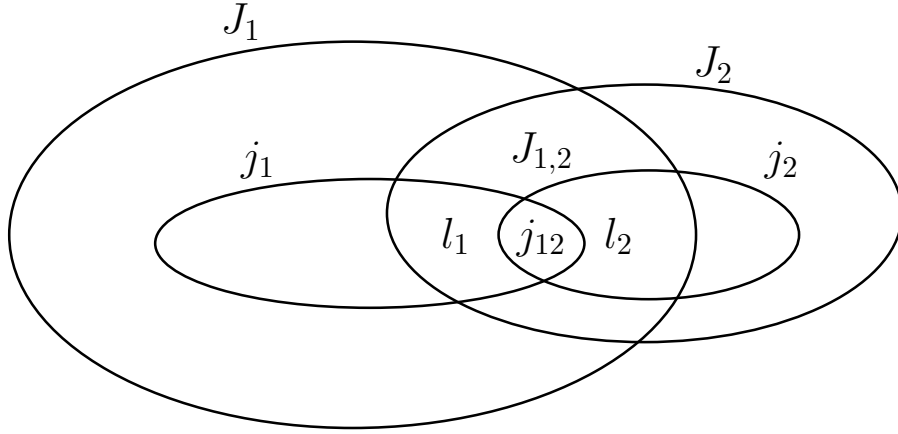


Figure 4: Pictorial representation of the sets of distinct join attributes, used to derive an expression for $p(j_{12}; n_1, n_2)$. Each set is characterized by a cardinality, which is indicated in the figure.

and $b^{(2)}$. It is straightforward to verify that the top 5 combinations listed in Table 1(e) are included in the union of the combinations formed after the sorted accesses (Table 1(c)) and attribute-based accesses (Table 1(d)), as guaranteed by the algorithm described in Section 3.

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 search services are typically accessed remotely on the Internet. We adopt the following additive cost model:

$$C(n_1, n_2) = sc_1 n_1 + sc_2 n_2 + ac_2 j_1(n_1) + ac_1 j_2(n_2) \quad (7)$$

where sc_i denotes the unitary sorted access cost (per tuple) and ac_i the unitary attribute-based access cost (per distinct join attribute sub-tuple). Note that the attribute-based access cost associated to s_1 depends on ac_1 as well as on the number of distinct join attributes $j_2(n_2)$ retrieved from s_2 , and viceversa.

Given n_1 and n_2 , $C(n_1, n_2)$ 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

$$\bar{C}(n_1, n_2) = sc_1 n_1 + sc_2 n_2 + ac_2 \bar{j}_1(n_1) + ac_1 \bar{j}_2(n_2). \quad (8)$$

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 (5) we get

$$\bar{j}_i(n_i) \simeq \frac{n_i}{\bar{q}_i(n_i)}. \quad (9)$$

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 A and it is defined over the range $[0, Q_i]$. Its expected value is equal to $E[p(\tilde{q}_i; n_i)] = Q_i \frac{n_i}{N_i}$.

The random variable $q_i(n_i)$ used in (5) differs from $\tilde{q}_i(n_i)$, since 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.,:

$$p(q_i; n_i) = p(\tilde{q}_i | \tilde{q}_i > 0; n_i) = \frac{p(\tilde{q}_i; n_i)}{1 - p(\tilde{q}_i = 0; n_i)} \quad (10)$$

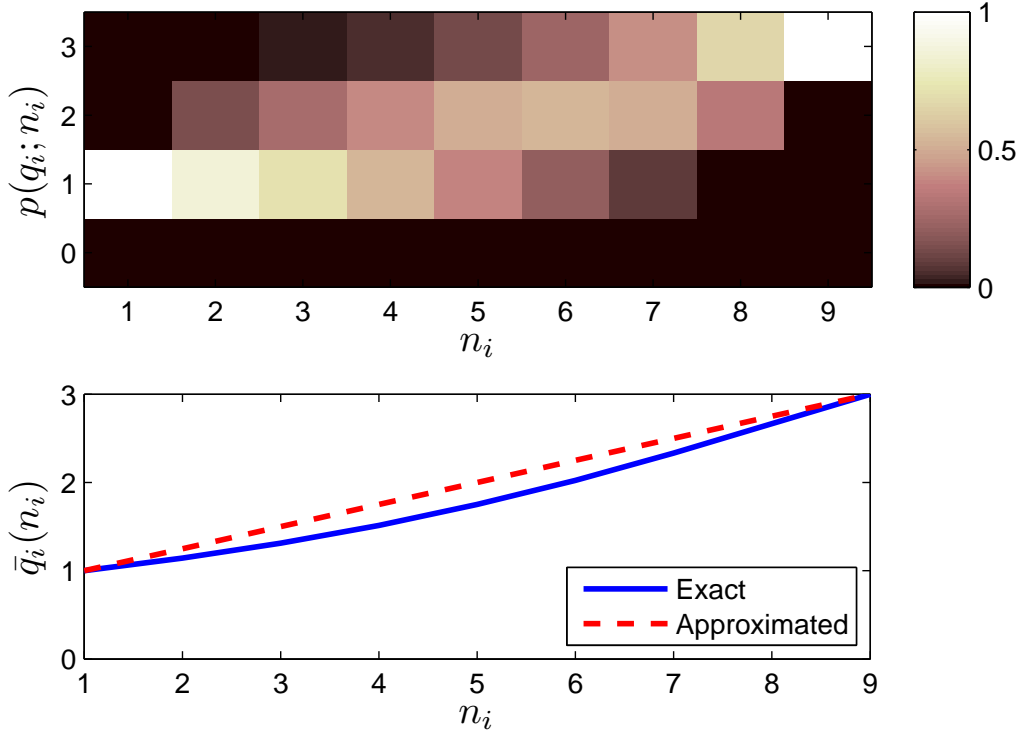


Figure 5: Probability mass function of $q_i(n_i)$ and its expected value $\bar{q}_i(n_i)$. $J_i = 3$, $Q_i = 3$.

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

$$\bar{q}_i(n_i) = \frac{Q_i \frac{n_i}{N_i}}{1 - p(\tilde{q}_i = 0; n_i)} \quad (11)$$

Figure 5 shows the p.m.f. of $q_i(n_i)$ for a service characterized by the same parameters as s_1 used in Table 1, i.e., $J_1 = 3$ and $Q_1 = 3$. Figure 5 also shows the expected value $\bar{q}_i(n_i)$ and an approximation obtained by means of a straight line. We observed, and also verified numerically, that such approximation is generally applicable, and it gets more accurate for larger values of Q_i . Therefore, we shall use the following approximated expression

$$\bar{q}_i(n_i) \simeq \frac{Q_i - 1}{N_i - 1} (n_i - 1) + 1 \quad (12)$$

Substituting (12) in (9) we get

$$\bar{j}_i(n_i) \simeq \frac{n_i}{a_i n_i + b_i} \quad (13)$$

where $a_i = \frac{Q_i - 1}{N_i - 1}$ and $b_i = 1 - a_i$. Figure 6 shows the p.m.f. of $j_i(n_i)$ and $\bar{j}_i(n_i)$ for the same parameter values as in Figure 5. The proposed approximation (13) 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 .

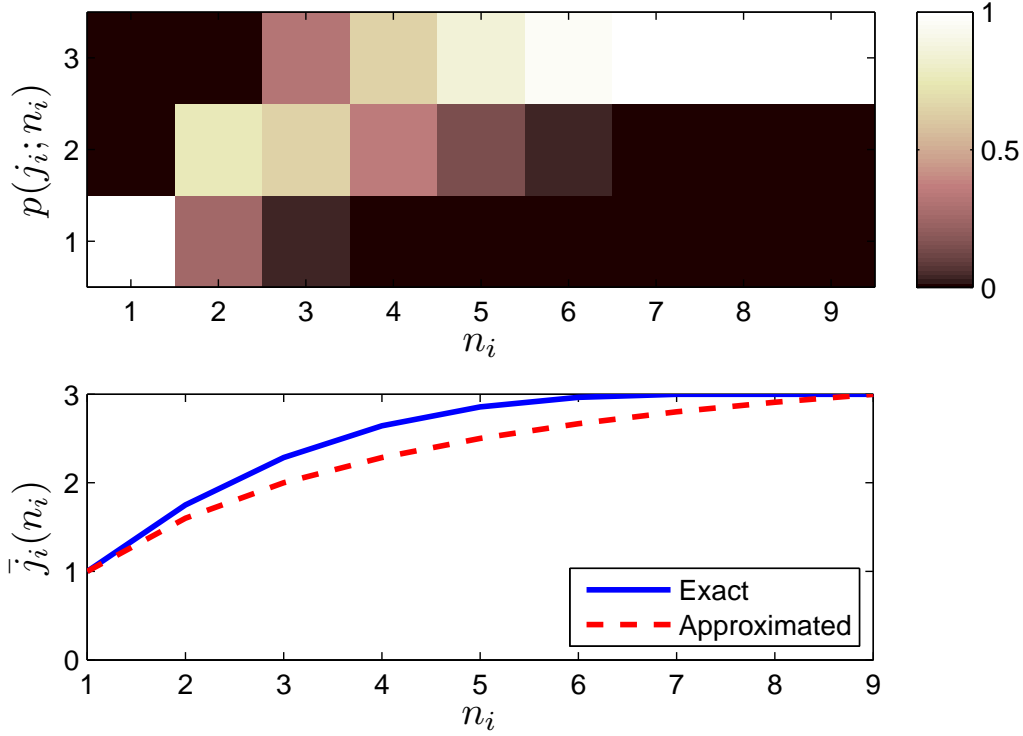


Figure 6: Probability mass function of $j_i(n_i)$ and its expected value $\bar{j}_i(n_i)$. $J_i = 3$, $Q_i = 3$.

4.3 Problem solution

With the approximations derived in Section 4.1 and Section 4.2, the optimization problem (2) can be written:

$$\begin{aligned} & \text{maximize} && \frac{n_1 n_2 J_{1,2}}{J_1 J_2} \\ & \text{subject to} && sc_1 n_1 + sc_2 n_2 + ac_2 \frac{n_1}{a_1 n_1 + b_1} + ac_1 \frac{n_2}{a_2 n_2 + b_2} \leq C_T \\ & && n_i \in \mathbb{N} \cap [0, N_i] \end{aligned} \quad (14)$$

In order to solve (14), we relax the problem by replacing the integer constraint $n_i \in \mathbb{N} \cap [0, N_i]$ with $n_i \in \mathbb{R}$. In addition, we formulate an equivalent optimization problem substituting the objective function with $\log(n_1 n_2)$. Thus, we seek for the solution of the problem

$$\begin{aligned} & \text{maximize} && \log(n_1 n_2) \\ & \text{subject to} && sc_1 n_1 + sc_2 n_2 + ac_2 \frac{n_1}{a_1 n_1 + b_1} + ac_1 \frac{n_2}{a_2 n_2 + b_2} \leq C_T \end{aligned} \quad (15)$$

The Lagrangian function related to the problem (15) is

$$\begin{aligned} L(n_1, n_2, \lambda) = & -\log(n_1 n_2) + \lambda (sc_1 n_1 + sc_2 n_2 \\ & + ac_2 \frac{n_1}{a_1 n_1 + b_1} + ac_1 \frac{n_2}{a_2 n_2 + b_2} - C_T) \end{aligned} \quad (16)$$

If $ac_i \neq 0$ the relaxed problem is not convex. Yet, we can impose the Karush-Kuhn-Tucker conditions, which must be satisfied by the optimal solution $\langle n_1^*, n_2^* \rangle$. In Appendix A we obtain

$$\begin{aligned} n_1^* &= n_1(\lambda^*, a_1, b_1, sc_1, ac_2) \\ n_2^* &= n_2(\lambda^*, a_2, b_2, sc_2, ac_1) \end{aligned} \quad (17)$$

In (17) 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^* . In Appendix A we show that this is achieved by finding one of the roots of a third degree polynomial, whose coefficients are expressed in terms of, respectively, $(\lambda^*, a_1, b_1, sc_1, ac_2)$ and $(\lambda^*, a_2, b_2, sc_2, ac_1)$. 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 dichotomic search over the possible values assumed by λ^* . More specifically, let $j = 1, \dots, \mathcal{J}$ 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 $n_{1,j} = n_1(\lambda_j, a_1, b_1, sc_1, ac_2)$ and $n_{2,j} = n_2(\lambda_j, a_2, b_2, sc_2, ac_1)$, 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 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 large positive constant.
2. for $j = 1 : \mathcal{J}$
 - (a) Set $\lambda_j = \frac{\lambda_l + \lambda_u}{2}$.
 - (b) Find $n_{1,j} = n_1(\lambda_j, a_1, b_1, sc_1, ac_2)$ and $n_{2,j} = n_2(\lambda_j, a_2, b_2, sc_2, ac_1)$.
 - (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 dichotomic 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 $n_i^* \leq N_i$.

4.4 Probability of finding top k combinations

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

In (4) we wrote the number of 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 Appendix A we give exact expressions for the p.m.f.'s of such discrete-valued random variables. Since $j_{12}(n_1, n_2)$ is defined over the interval $\mathbb{N} \cap [0, J_{1,2}]$, and $q_i(n_i)$ over the interval $\mathbb{N} \cap [0, Q_i]$, the resulting p.m.f. $p(\mathcal{K}; n_1, n_2)$ has support $\mathbb{N} \cap [0, J_{1,2}Q_1Q_2]$ and can be evaluated numerically. The probability $P_a(n_1, n_2; k)$ of finding at least k combinations is given by 1 minus the sum of $p(\mathcal{K}; n_1, n_2)$ over $[0, k - 1]$.

Figure 7(a) shows the expected number of combinations for the problem parameters of the example in Table 1, while Figure 7(b) the probability of finding at least $k = 5$ combinations after $\langle n_1, n_2 \rangle$ sorted accesses. As an example, for $\langle n_1, n_2 \rangle = \langle 6, 5 \rangle$, such probability is equal to 0.55.

Although the analytical expression of $P_a(n_1, n_2; k)$ is too complex to be dealt with in optimization, we can still obtain a point-wise value of $P_a(n_1, n_2; k)$ for a given pair $\langle n_1, n_2 \rangle$. Therefore, upon solving problem (2), we can numerically evaluate $P_a(n_1^*, n_2^*; k)$ for a target value of k .

4.5 Extending to more than 2 services

The analytical problem formulation and solution described above for joins involving two services can be straightforwardly extended to M -ary joins (with $M > 2$) over the same join attributes. In the running example, one could, e.g., also require that a highly rated pub showing football games be present on the same street by adding a join with a third service. The FA-join algorithm described in Section 3 can be

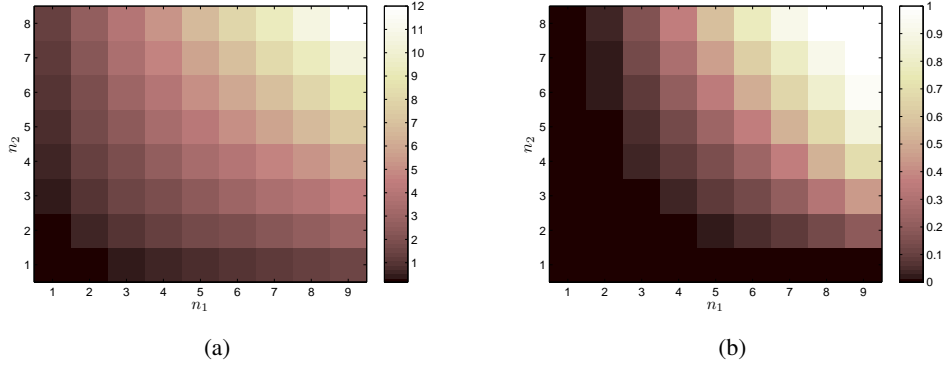


Figure 7: a) Expected number of combinations $\bar{\mathcal{K}}(n_1, n_2)$. b) $P_a(n_1, n_2; k = 5)$.

equally applied in this case, with a minor change: after performing sorted accesses to s_i , proceed with the corresponding attribute-based accesses to all the other services s_m , $m = 1, \dots, M$, $m \neq i$. The stopping condition remains unaltered, i.e. stop after finding at least k combinations with sorted accesses. Therefore, we seek for the solution of the optimization problem:

$$\begin{aligned}
 & \text{maximize} && J_{1,2,\dots,M} \cdot \prod_{i=1}^M \frac{n_i}{J_i} \\
 & \text{subject to} && \sum_{i=1}^M \left(sc_i n_i + j_i(n_i) \sum_{m \neq i} ac_m \right) \leq C_T \\
 & && n_i \in \mathbb{N} \cap [0, N_i]
 \end{aligned} \tag{18}$$

where $J_{1,2,\dots,M}$ denotes the number of distinct join attributes in common to the M services. The problem can be solved similarly to the case $M = 2$, by relaxing the integer constraint and finding the solution that satisfies the Karush-Kuhn-Tucker conditions for the relaxed problem.

5 Execution strategy

Operatively, we are interested in defining a strategy that describes how the search services shall be incrementally accessed, in order to maximize at each execution step the number of combinations found, given the knowledge of the parameters that characterize the services. To this end, instead of solving problem (15) 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. Figure 8 shows such locus for the services in Table 1, when $sc_1 = 1$, $sc_2 = 2$, $ac_1 = 1$ and $ac_2 = 10$. Each point on the curve is the solution of problem (15) for some value of C_T . The black circles indicate the result obtained after rounding.

In Section 2 we pointed out that most search services return paginated results. Therefore each service may return more than one result at a time, i.e., the number of items retrieved from s_i is constrained to be a multiple of a given page size $\mathcal{P}_i \in \mathbb{N}$. Here we devise an optimal execution strategy that takes pagination into account, in order to determine, at each step, the next service to be invoked.

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 search 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 strategies

$$\begin{aligned}
 \langle n_1^{(t)}, n_2^{(t)} \rangle &= \langle n_1^{(t-1)} + \mathcal{P}_1, n_2^{(t-1)} \rangle \\
 \langle n_1^{(t)}, n_2^{(t)} \rangle &= \langle n_1^{(t-1)}, n_2^{(t-1)} + \mathcal{P}_2 \rangle
 \end{aligned} \tag{19}$$

At each step $t > 1$, we use one strategy in (19) 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 defined by the parametric equations (17), traced by varying the target cost C_T .

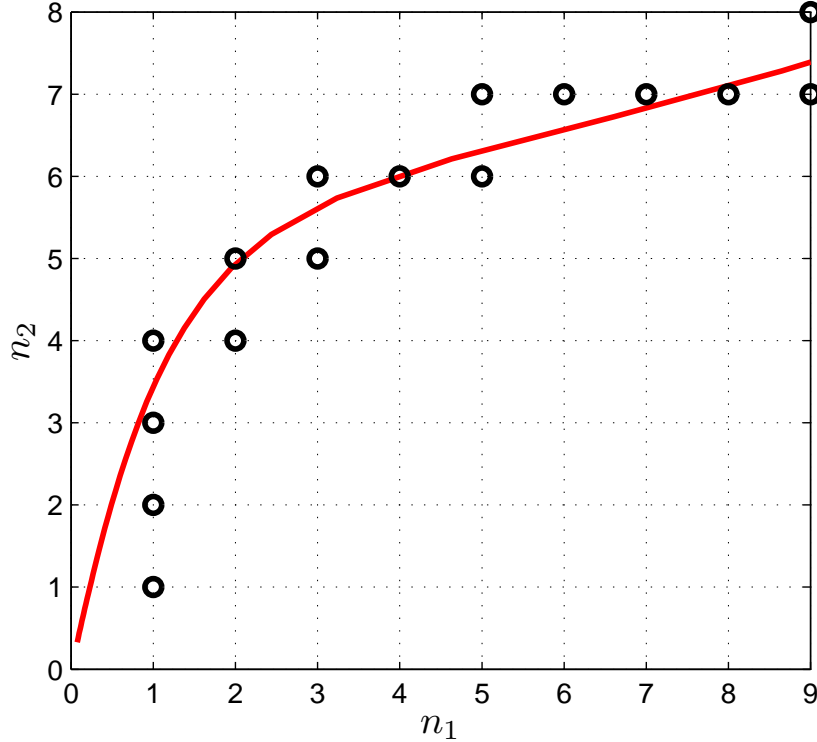


Figure 8: The solid red curve indicates the locus of optimal solutions of the relaxed optimization problem (15) obtained by varying C_T . The black circles indicate the approximate solution obtained after rounding.

Intuitively, this corresponds to finding an approximate solution to (14), 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 $t - 1$. Let $k^{(t)}$ indicate the number of common elements observed at step t . Then, we stop making sorted accesses as soon as we reach a step T such that $k^{(T)} \geq k$. Then, we proceed with attribute-based accesses to find the top $k^{(t)}$ combinations. We also note that the solution sought $\langle n_1^{(T)}, n_2^{(T)} \rangle$ is the one that returns the top k combinations while minimizing (approximately, due to rounding) the overall access cost.

The described strategy characterizes what we referred to as the CAFA-join algorithm. The CATA-join algorithm only differs from CAFA-join wrt. the stopping condition $k^{(T)} \geq k$, which is replaced by a condition on a score-based threshold.

In Section 6 we will show that, whenever the two services are characterized by different access costs, a cost-aware strategy outperforms the corresponding naive strategy, in terms of the overall access cost needed to retrieve a target number of top-k combinations.

6 Experiments

In this section we evaluate the proposed approach on two datasets. First, in order to provide a qualitative insight, we demonstrate the proposed approach on a concrete instantiation of our running example. Second, to validate the usefulness of the approach with quantitative results, we generate data for two synthetic services by means of Montecarlo simulations, and we assess the potential benefits brought by a cost-aware execution strategy in terms of cost reduction to retrieve the top-k combinations.

6.1 Running example

We consider two search services available on the Web, which provide both sorted and attribute-based access, namely `hotel` for hotels as service s_1 and `rest` for Parisian restaurants as service s_2 .

The former service, among the available access modes, 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 attribute-based 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 attributes, 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 attribute-based 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 $Q_2 = 1.23$. All the parameters characterizing the services have been obtained at compile time by executing a few sample queries.

We formulate the following query:

Find the top k combinations of hotels and restaurants serving French food, which are within the same street.

The score aggregation function used in this example is

$$f(t_1[S_1], t_2[S_2]) = t_1[S_1] \cdot t_2[S_2] \quad (20)$$

but any other monotone aggregation function would work. Before computing the combined score, we normalized both scores $t_1[S_1]$ and $t_2[S_2]$ to fall within the $[0, 1]$ interval.

If all the tuples of both search services are retrieved, it is possible to form a total of 590 combinations. In the following, we apply the cost-aware access strategy devised in Section 5 to efficiently retrieve the top combinations, ranked by combined score. We estimated the average access costs involved in both sorted and attribute-based access, and we obtained the following parameters: $sc_1 = 0.01$, $sc_2 = 0.01$ (in seconds/tuple), $ac_1 = 0.10$, $ac_2 = 0.01$ (in seconds/access).

Figure 9(a) illustrates the cost-aware access plan computed with the above problem parameters. The solid curve represents the locus of solutions for the relaxed problem (15), while the dark circles indicate the number of tuples retrieved from the two services during the sorted access phase at each execution step $t = 1, \dots, T$, with $T = 16$. Here, the two services are characterized by equal sorted access costs but different attribute-based access costs. Then, a cost-aware strategy would intuitively request more tuples from s_1 in the sorted access phase, since the attribute-based access to s_2 are cheaper.

Note that in this example both Q_1 and Q_2 are close to 1 and the optimal access strategy approximately follows a straight line. Indeed, in the limit case when $Q_i = 1$, the expected cost function (8) becomes linear in n_1 and n_2 . Solving the problem and tracing the locus of optimal solutions by letting C_T vary results in a straight line whose slope is $\frac{sc_1 + ac_2}{sc_2 + ac_1}$.

Table 2 provides further details about the cost-aware access plan depicted in Figure 9(a). At each step t , we indicate the number of combinations $k_{FA}^{(t)}$ formed after the sorted access phase (such that the top $k_{FA}^{(t)}$ combinations are guaranteed to be retrieved after the attribute-based access phase, i.e. according to a CAFA-join strategy), the number of combinations $k_{TA}^{(t)}$ whose aggregated scores exceed the current threshold value (such that the top $k_{TA}^{(t)}$ combinations are guaranteed to be retrieved after the attribute-based access phase, i.e. according to a CATA-join strategy), the number of such sorted accesses $\langle n_1^{(t)}, n_2^{(t)} \rangle$, the number of distinct streets $j_i^{(t)}$ retrieved from s_i , and the overall access cost $C(n_1, n_2)$ (in

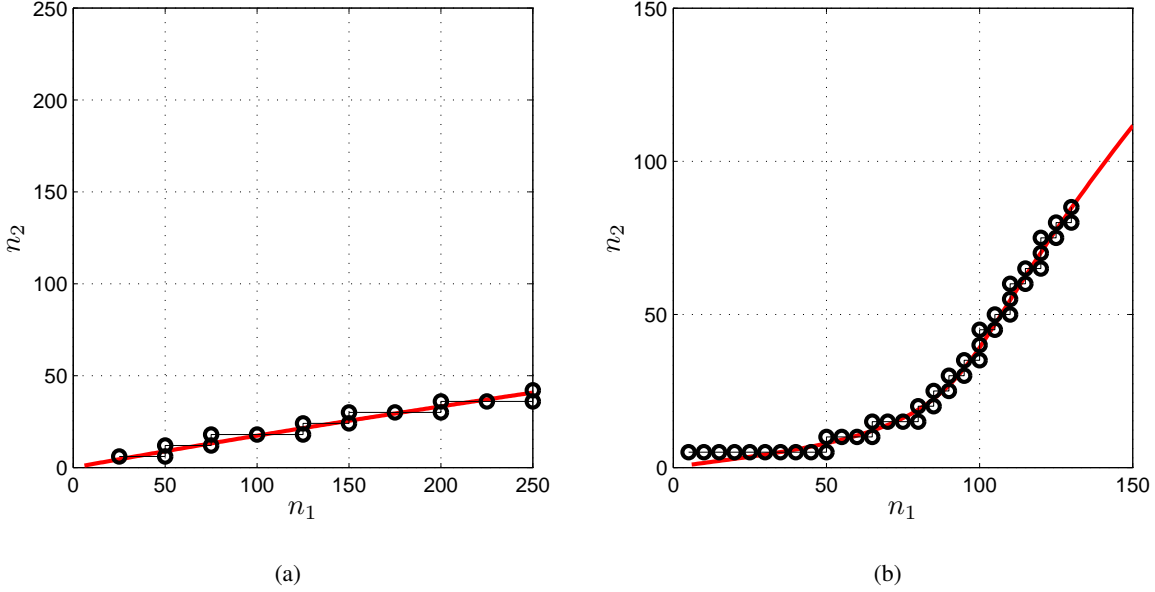


Figure 9: Cost-aware access plans for retrieving the top-k combinations. The solid curve represents the locus of optimal solution of the relaxed problem (15). The dark circles indicate the number of tuples retrieved from the two services during the sorted access phase. a) Running example. b) Simulated services.

seconds) due to both sorted and attribute-based accesses. The leap from $k_{TA}^{(8)} = 4$ to $k_{TA}^{(9)} = 104$ in the table is mainly due to the fact the 9^{th} step accesses the first page in s_1 containing 3-star hotels, which significantly lowers the threshold.

Observe that, within the first 8 seconds, CATA-join (resp., CAFA-join) is able to find the top 104 (resp., 4) combinations. Therefore, even though CATA-join and CAFA-join adopt the same access strategy, the use of a threshold-based stopping criterion at run-time can be extremely helpful. For comparison, consider that the (non-cost-aware) TA-join and FA-join approaches described in Section 3, within the first 8 seconds, only retrieve the top 8 and top 3 combinations, respectively. This shows that a cost-aware strategy can indeed increase the number of top combinations that can be formed.

Last, Table 3 shows the top 5 combinations in our example.

6.2 Simulated data

We synthetically generated data for two services in order to demonstrate the potential gains that can be obtained when a cost-aware access strategy is implemented. In a first experiment, we generated services characterized by the same parameters, e.g., number of distinct join attributes $J_1 = J_2 = 25$, of which $J = 20$ are in common. For each distinct join attribute, we generated a number of tuples according to a Poisson distribution whose mean is set equal to $Q_1 = Q_2 = 20$. We note that in the development of the proposed cost-aware strategy, we assumed that each distinct join attribute in service s_i corresponds to a fixed number of tuples Q_i in order to enable mathematical tractability. Conversely, in these experiments we generate services to mimic real world data. The individual scores of the tuples are sampled from an exponential distribution with mean equal to 10.

Figure 10 shows the number of top-k combinations that can be formed as a function of the overall access cost (expressed in seconds) for the four strategies described in Section 5, when we set sc_i and ac_i

Table 2: Step-by-step cost-aware access plan for retrieving the top hotel-restaurant combinations in the same street.

t	$k_{FA}^{(t)}$	$k_{TA}^{(t)}$	$n_1^{(t)}$	$n_2^{(t)}$	$j_1^{(t)}$	$j_2^{(t)}$	$C^{(t)}(n_1, n_2)$
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
10	2	104	150	30	127	29	5.97 s
11	2	104	175	30	149	29	6.44 s
12	2	104	200	30	169	29	6.89 s
13	4	104	200	36	169	34	7.45 s
14	4	104	225	36	186	34	7.87 s
15	4	104	250	36	204	34	8.30 s
16	5	104	250	42	204	39	8.86 s

Table 3: Top 5 hotel-restaurant combinations in the same street.

Hotel	Restaurant	Street	Score
De Vendôme	Café de Vendôme	Pl. Vendôme	0.6
De Vendôme	Bar 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

as in the running example. The corresponding cost-aware access plan is depicted in Figure 9(b). The scatter points in Figure 10 correspond to the results obtained executing the same access plans (cost-aware vs. naive) on five different data realizations of the same services. The solid curves are the result of fitting the scatter points with a fourth degree polynomial. A cost-aware execution strategy is beneficial, since both CATA-join and CAFA-join outperform, respectively, (non-cost-aware) TA-join and FA-join for a target value of k . For example, if we fix $k = 100$, we observe, respectively, a 22% and a 9% cost reduction.

The cost reduction that can be attained by a cost-aware strategy obviously depends on differences between the actual access costs of the two services. In order to provide a more comprehensive comparison, we let both sorted and attribute-based access cost vary in the set $\{L = 0.01, M = 0.10, H = 1.00\}$ and computed the cost reduction that can be achieved for each of the 81 possible cost combinations.

Figure 11 shows the results obtained averaging five different data realizations for each cost combination, and measuring the cost reduction between CATA-join and (non-cost-aware) TA-join when computing the top-100 combinations. We observe in Figure 11 that, when sorted access costs dominate and they are equal (e.g., $sc_1 = sc_2 = H$), the cost-aware strategy corresponds to the naive strategy, and no cost reductions can be obtained. Conversely, the highest gains, exceeding 70%, are achieved when the sorted access costs of the two services are very different from each other (e.g., $sc_1 = H, sc_2 = L$ and vice versa). More so if the attributed-based access costs contrast the sorted ones (e.g., $sc_1 = H, sc_2 = L, ac_1 = L, ac_2 = H$), since accessing s_2 is cheaper both in terms of sorted access and in terms of the corresponding attributed-based accesses to be made to s_1 . The average cost reduction observed for the cost combinations reported in Figure 11 is around 30%. The cost reductions between CAFA-join and FA-join, not reported in the figure, appear to be even higher.

Similar cost reductions are achieved, both for CATA-join wrt. TA-join and for CAFA-join wrt. FA-join, if different service parameters are used for the different services (i.e., $J_1 \neq J_2$ or $Q_1 \neq Q_2$). We

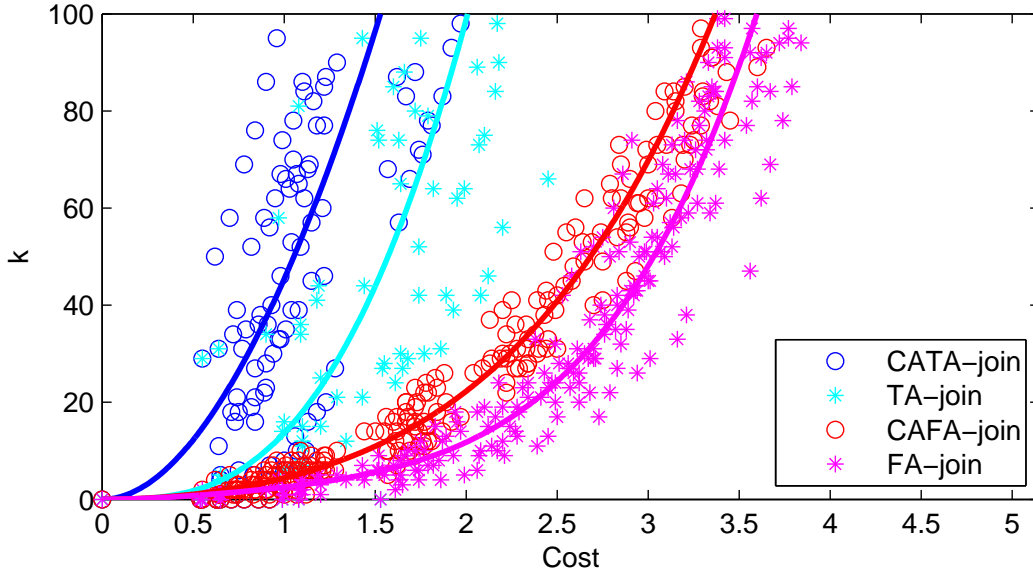


Figure 10: Number of top- k combinations found as a function of access cost.

tested, e.g., $J_1 = 30$ and $J_2 = 10$, leaving the other parameters unchanged, and obtained gains up to 80% averaging more than 30%. We also tested $Q_1 = 40$ and $Q_2 = 20$, obtaining similar gains.

Finally, we tested our approach on instances in which the score distributions on the two services are significantly dissimilar (e.g., two exponential distributions, with average 10 for s_1 and 20 for s_2). Comparisons between CATA-join and TA-join give the same cost reductions as above. For fairness, however, we also compared CATA-join with a “score-aware” variant of TA-join that always makes a sorted access to the service whose last accessed tuple has the highest score (as is done, e.g., by the HRJN* algorithm [13]). In this case, too, CATA-join generally outperforms TA-join, but with lower gains: the cost reduction may still exceed 70%, but averages only slightly more than 10%, since, on some instances, quicker reductions of the threshold may be obtained by this heuristics.

7 Related work

Top- k query answering is a topic that has been addressed by a large body of research during the last years [4, 18, 5, 8, 16, 10, 19, 6, 9, 22, 13, 2, 24, 17, 12, 23, 1, 14, 20]. A comprehensive survey on the subject is found in [15].

In our work we started from the algorithm described in [6]. Fagin’s algorithm (FA) performs a number of sorted accesses in order to find at least k objects in common, and then proceeds with random accesses.

Based on the same principles, the *threshold algorithm* (TA) is introduced in [9], by adding a sufficient stopping condition that limits the number of sorted accesses needed to guarantee finding the top k objects. As soon as an item is extracted with a sorted access, the corresponding random accesses are made on the other lists. TA stops when there are at least k items with a score higher than the overall score (the threshold) computed aggregating the scores of the last items extracted by sorted access. In this way, the cost for sorted accesses in TA is at most that of FA (however TA may make more random accesses than FA). The same algorithm has been independently proposed by [10], under the name of Quick-Combine.

TA is proved to be *instance optimal* for monotone aggregation functions, in the sense that there are

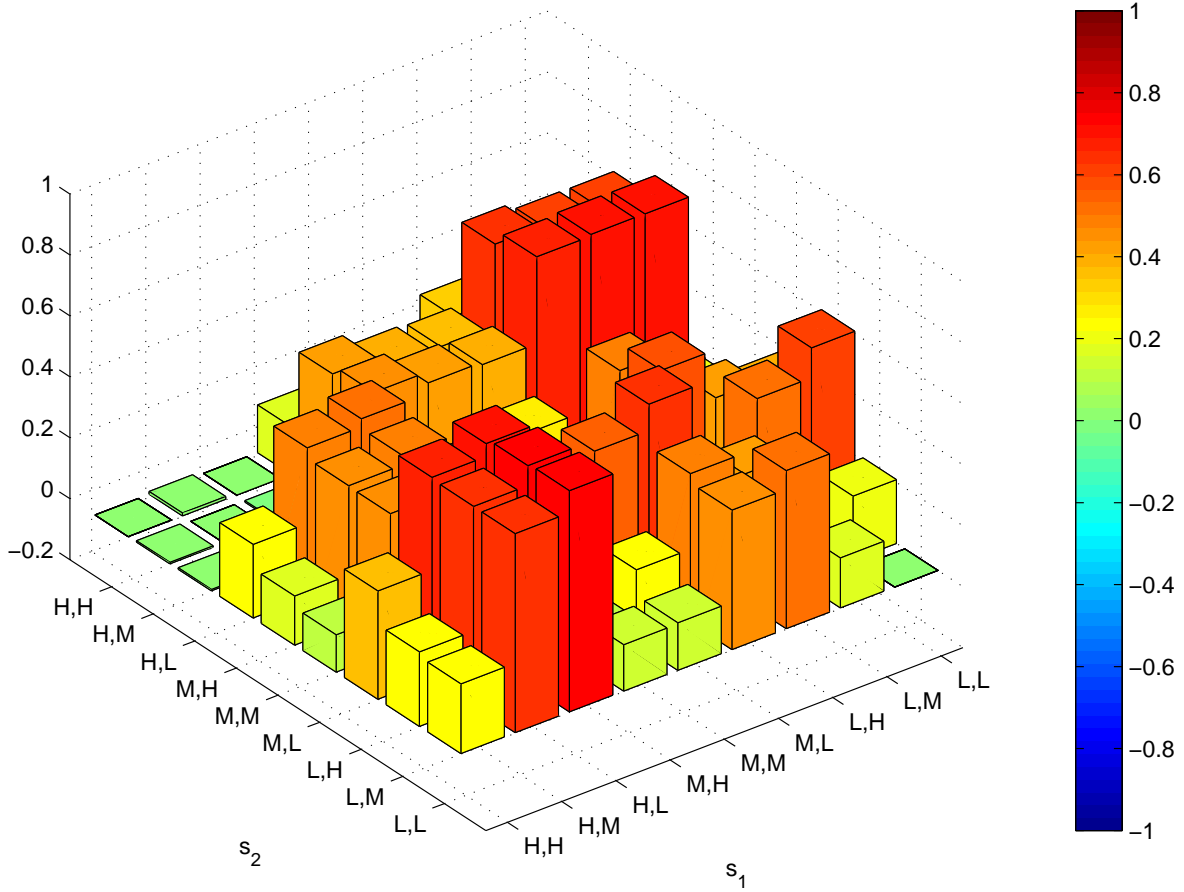


Figure 11: Cost reduction achieved by CATA-join w.r.t. non-cost-aware TA-join to compute the top-100 combinations.

constants c and c' such that, for every database D and algorithm \mathcal{A} , $cost_{TA}(D) \leq c \cdot cost_{\mathcal{A}}(D) + c'$, where $cost_{\mathcal{A}}(D)$ is the overall cost incurred by \mathcal{A} in D . Although instance optimality is a strong theoretical property, different implementations can have significant performance differences in terms of accesses, computational cost, and memory requirements.

In many practical scenarios, the cost associated to sorted and random access might be significantly skewed. Therefore in [9] a cost model is proposed to guide the scheduling 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 scheduling algorithm is described in [2], 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.

Several variants of the TA have been proposed in the literature based on the different data access methods available: no random access (NRA [9], Stream Combine [11]); controlled random probes (MPro [5], Upper and Pick [4]); random access also returning the position (BPA and BPA2 [1]).

The NRA algorithm assumes that data sources can only be accessed with sorted accesses. Based on the knowledge of the score ranges of each source and the specific score aggregation function, NRA computes, for each object retrieved by at least one source, the range of their combined scores. If the range lower bound of an object is above a threshold defined similarly to the TA algorithm, the object is reported in the top-k list. Optimized variants of NRA are proposed in [18], where the authors identify

two phases (growing and shrinking) that must be common to all NRA algorithms.

In [5, 22] the concept of “necessary probe” is introduced, to indicate whether a predicate probe is absolutely required or not. The proposed Minimal Probing (MPro) algorithm uses this idea to minimize the predicate evaluation cost.

In [4] algorithms are proposed for evaluating top-k queries over Web sources. There, exactly one source can be accessed with sorted accesses, while the others allow only random access. The query is processed by first retrieving the top-ranked objects from the sorted source. Then, random accesses are performed to the other sources by selecting the best source to be queried at each step.

One of the first studies of top k answers in join scenarios is [19]. There, algorithm J^* is introduced and proved to be instance optimal among the algorithms that only use sorted access. The algorithm is extended to also use *predicate access*, akin to our attribute-based access, to access all objects from a given list that satisfy a certain predicate.

An instance optimal adaptation of the NRA algorithm to the join context is given in [13]. The threshold value in use changes depending on the strategy used to explore the lists, thus affecting how quickly the top k results can be computed. The authors report that traditional join strategies such as nested-loop joins, merge joins, and hash joins result in poor performance, and therefore suggest new binary join operators that help sweeping the join space in a way that reduces the threshold value, possibly using information on the score distribution in the lists. They also discuss an extension of their algorithm to use random probes, thus providing an adaptation of TA to the join context.

Attempts to provide estimates of the costs of a join in a top-k setting were made in [16, 14]. A probabilistic model is proposed to estimate the number of tuples (called depth) consumed from each input to produce the top-k join 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 in terms of the joint distribution of scores is given in [20].

Most algorithms in the field assume monotonicity of the aggregation function. Strategies for answering queries with non-monotone aggregation functions are studied in [17] and [24].

8 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 accounts for the different access patterns: sorted access and attribute-based access. To this end, we introduce a statistical framework to characterize the number of combinations and the number of attribute-based accesses.

As future work, we are considering extensions of the presented strategy to the case of non-additive cost models capturing the fact that service accesses may, in some cases, be performed in parallel. Moreover, we are investigating the problem of using top-k optimization in the context of general query plans.

References

- [1] R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *VLDB*, pages 495–506, 2007.
- [2] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.
- [3] D. Braga, S. Ceri, F. Daniel, and D. Martinenghi. Optimization of multi-domain queries on the web. *Proceedings of the VLDB Endowment*, 1(1):562–573, 2008.

- [4] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–, 2002.
- [5] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.
- [6] R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.
- [7] R. Fagin, R. Kumar, M. Mahdian, D. Sivakumar, and E. Vee. Comparing partial rankings. *SIAM J. Discrete Math.*, 20(3):628–648, 2006.
- [8] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD Conference*, pages 301–312, 2003.
- [9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.
- [10] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.
- [11] U. Güntzer, W.-T. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.
- [12] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, pages 950–961, 2002.
- [13] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004.
- [14] 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.*, 31(4):1257–1304, 2006.
- [15] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top- query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
- [16] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *SIGMOD Conference*, pages 203–214, 2004.
- [17] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, pages 115–126, 2007.
- [18] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung. Efficient aggregation of ranked inputs. In *ICDE*, page 72, 2006.
- [19] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290, 2001.
- [20] K. Schnaitter, J. Spiegel, and N. Polyzotis. Depth estimation for ranking query optimization. In *VLDB*, pages 902–913, 2007.
- [21] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB’06*, pages 355–366. VLDB Endowment, 2006.
- [22] 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.*, 19(5):646–662, 2007.
- [23] M. Wu and C. Jermaine. A bayesian method for guessing the extreme values in a data set. In *VLDB*, pages 471–482, 2007.
- [24] D. Xin, J. Han, and K. C.-C. Chang. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *SIGMOD Conference*, pages 103–114, 2007.

A Mathematical derivations

A.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:

$$p(j_i = 0; n_i = 0) = 1 \quad (21)$$

$$p(j_i; n_i = 0) = 0 \text{ for } 1 \leq j_i \leq Q_i \quad (22)$$

$$\begin{aligned} p(j_i; n_i) &= p(j_i; n_i - 1) \cdot \frac{Q_i \cdot j_i - (n_i - 1)}{N_i - (n_i - 1)} \\ &+ p(j_i - 1; n_i - 1) \cdot \frac{N_i - Q_i \cdot (j_i - 1)}{N_i - (n_i - 1)} \end{aligned} \quad (23)$$

where (23) 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.

A.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:

$$h(x, M, m_1, m_2) = \frac{\binom{m_1}{x} \binom{M-m_1}{m_2-x}}{\binom{M}{m_2}} \quad (24)$$

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 search engine. If the l_i 's are deterministically known, then the p.m.f. of j_{12} is characterized by the hyper-geometric distribution:

$$p(j_{12}; l_1, l_2) = h(j_{12}, J_{1,2}, l_1, l_2) \quad (25)$$

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

$$\begin{aligned} p(j_{12} | j_1, j_2; n_1, n_2) &= \\ &= \sum_{l_1=0}^{J_{1,2}} \sum_{l_2=0}^{J_{1,2}} p(l_1 | j_1; n_1) p(l_2 | j_2; n_2) h(j_{12}, J_{1,2}, l_1, l_2) \end{aligned} \quad (26)$$

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

$$\begin{aligned} p(j_{12}; n_1, n_2) &= \\ &= \sum_{j_1=0}^{J_1} \sum_{j_2=0}^{J_2} p(j_1; n_1) p(j_2; n_2) p(j_{12} | j_1, j_2; n_1, n_2) \end{aligned} \quad (27)$$

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 (24) is given by $E[x] = \frac{m_1 m_2}{M}$, we obtain

$$\bar{l}_i(n_i) = E[p(l_i|\bar{j}_i; n_i)] = \bar{j}_i(n_i) \frac{J_{1,2}}{J_i} \quad (28)$$

Similarly, from (27) we find that \bar{j}_{12} is equal to

$$\bar{j}_{12} = \frac{\bar{j}_1(n_1)\bar{j}_2(n_2)J_{1,2}}{J_1 J_2} \quad (29)$$

A.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 $p(j_i; n_i)$:

$$p(\tilde{q}_i = 0; n_i) = \prod_{t=1}^{n_i} \left(1 - \frac{Q_i}{N_i - t + 1}\right) \quad (30)$$

$$p(\tilde{q}_i = 1; n_i = 1) = \frac{Q_i}{N_i} \quad (31)$$

$$p(\tilde{q}_i; n_i = 1) = 0 \text{ for } 2 \leq \tilde{q}_i \leq Q_i \quad (32)$$

$$\begin{aligned} p(\tilde{q}_i; n_i) &= p(\tilde{q}_i - 1; n_i - 1) \cdot \frac{Q_i - (\tilde{q}_i - 1)}{N_i - (n_i - 1)} \\ &+ p(\tilde{q}_i, n_i - 1) \cdot \frac{N_i - (n_i - 1) - (Q_i - \tilde{q}_i)}{N_i - (n_i - 1)} \end{aligned} \quad (33)$$

The expected value of $p(\tilde{q}_i; n_i)$ is given by $Q_i \frac{n_i}{N_i}$; intuitively, each joint 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.

A.4 Solution of the relaxed problem

The Karush-Kuhn-Tucker conditions that must be satisfied by the optimal solution $\langle n_1^*, n_2^* \rangle$ of the problem (15) are:

- Stationarity

$$\frac{\partial L(n_1^*, n_2^*, \lambda^*)}{\partial n_i} = 0 \quad (34)$$

- Primal feasibility

$$sc_1 n_1^* + sc_2 n_2^* + ac_2 \frac{n_1^*}{a_1 n_1^* + b_1} + ac_1 \frac{n_2^*}{a_2 n_2^* + b_2} - C_T \leq 0 \quad (35)$$

- Dual feasibility

$$\lambda^* \geq 0 \quad (36)$$

- Complementary slackness

$$\lambda^* (sc_1 n_1^* + sc_2 n_2^* + ac_2 \frac{n_1^*}{a_1 n_1^* + b_1} + ac_1 \frac{n_2^*}{a_2 n_2^* + b_2} - C_T) = 0 \quad (37)$$

From (34) we get for $i = 1$

$$-\frac{1}{n_1^*} + \lambda^* sc_1 + \frac{\lambda^* ac_2 b_1}{(a_1 n_1^* + b_1)^2} = 0 \quad (38)$$

The optimal solution n_1^* must be a root of (38). 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:

$$\alpha_3 = \lambda^* sc_1 a_1^2 \quad (39)$$

$$\alpha_2 = -a_1^2 + 2\lambda^* sc_1 a_1 b_1 \quad (40)$$

$$\alpha_1 = -2a_1 b_1 + \lambda^* sc_1 b_1^2 + \lambda^* ac_2 b_1 \quad (41)$$

$$\alpha_0 = -b_1^2 \quad (42)$$

Among the three roots of the polynomial, we pick the one which is real and positive. Let $n_1^* = n_1(\lambda^*, a_1, b_1, sc_1, ac_2)$ denote such a root. An equivalent procedure can be followed for s_2 . Therefore, we can write

$$\begin{aligned} n_1^* &= n_1(\lambda^*, a_1, b_1, sc_1, ac_2) \\ n_2^* &= n_2(\lambda^*, a_2, b_2, sc_2, ac_1) \end{aligned} \quad (43)$$