Information Systems 64 (2017) 75–92

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Exploratory product search using top-k join queries

Orestis Gkorgkas[a,*], Akrivi Vlachou[b], Christos Doulkeridis[c], Kjetil Nørvåg[a]

[a] Sem Sælandsv. 7-9, Norwegian University of Science and Technology (NTNU), Trondheim, Norway
[b] Institute for the Management of Information Systems, R.C. "Athena", Athens, Greece
[c] University of Piraeus, Piraeus, Greece

ARTICLE INFO

Recommended by D. Shasha

Keywords:
Exploratory search
Top-*k* queries
Join queries
Product combinations
Combination ranking

ABSTRACT

Given a relation that contains main products and a set of relations corresponding to accessory products that can be combined with a main product, the *Exploratory Top-k Join* query retrieves the k best combinations of main and accessory products based on user preferences. As a result, the user is presented with a set of k combinations of distinct main products, where a main product is combined with accessory products only if the combination has a better score than the single main product. We model this problem as a rank-join problem, where each combination is represented by a tuple from the main relation and a set of tuples from (some of) the accessory relations. The nature of the problem is challenging because the inclusion of accessory products is not predefined by the user, but instead all potential combinations (joins) are explored during query processing in order to identify the highest scoring combinations. Existing approaches cannot be directly applied to this problem, as they are designed for joining a predefined set of relations. In this paper, we present algorithms for processing exploratory top-k joins that adopt the pull-bound framework for rank-join processing. We introduce a novel algorithm (XRJN) which employs a more efficient bounding scheme and allows earlier termination of query processing. We also provide theoretical guarantees on the performance of this algorithm, by proving that XRJN is instance-optimal. In addition, we consider a pulling strategy that boosts the performance of query processing even further. Finally, we conduct a detailed experimental study that demonstrates the efficiency of the proposed algorithms in various setups.

1. Introduction

Nowadays, product databases contain large collections of objects, where each object is characterized by a number of different properties such as price or weight. In many cases, products can be combined with accessories and form combinations with enhanced or extended properties, and in this way generating numerous options for the users. For instance, a user wishing to buy a laptop might find a laptop combined with some additional memory parts and an extra SSD disk more suitable to her needs than any single laptop. However, quite often all potential item combinations are not possible. Some laptops may support the addition of a secondary internal disk while others not. Another indicative example is that of a tourist visiting a city who seeks a highly rated hotel with a restaurant. The tourist might also be interested in highly rated hotels without restaurant that are close to a restaurant. In this example, hotels should only be combined with restaurants, if their distance is relatively short. The plethora of different items and combinations make it challenging for users to explore the available options and find the products that suit their needs.

Ranking queries, such as top-k [1–3] and rank-join queries [4–6], assist users in finding products that are interesting to them by selecting a small set of items or combinations that are highly ranked according to their preferences. In such queries, products are typically modeled as multi-dimensional points, where each dimension corresponds to a specific attribute and the respective value indicates the presence or the performance of a product regarding this attribute [7–9]. User preferences are modeled as multi-dimensional vectors $\mathbf{w}$, where each component (weight) of the vector denotes the importance of the respective attribute for the ranking of the objects [10–12]. The weight values could either be explicitly declared by the user through an interactive user interface [10], or be indirectly estimated [13]. Each item or combination is assigned a score, frequently using a linear scoring function [11,14,15,16–18] of the form $f_{\mathbf{w}}(o) = \sum_{i=1}^{d} \mathbf{w}[i]o[i]$ that assigns scores to products, where $o[i]$ is the normalized value of the i-th attribute of a product o. However, both top-k and rank-join queries present to the users a very limited overview of the available alternatives. Top-k queries, on one hand, focus solely on the products the user is interested in, and ignore the fact that combinations can be

* Corresponding author.
E-mail addresses: orestis@idi.ntnu.no (O. Gkorgkas), vlachou@idi.ntnu.no (A. Vlachou), cdoulk@unipi.gr (C. Doulkeridis), noervaag@idi.ntnu.no (K. Nørvåg).

http://dx.doi.org/10.1016/j.is.2016.09.004
Received 21 April 2015; Received in revised form 30 June 2016; Accepted 3 September 2016
Available online 16 September 2016

Laptops

id	CPU score	RAM (GB)	SSD (GB)	price (USD)	battery (hours)	weight (kg)	RAM type	SSD type
c_1	3346	4	0	-539	4	-2.7	1	1
c_2	3346	8	256	-1119	6.5	-2	1	2
c_3	3941	12	256	-1199	8	-2.5	0	0
c_4	3997	8	0	-1348	13	-2	2	1

Memory

mid	RAM (GB)	price (USD)	RAM type
m_1	4	-71	2
m_2	4	-45	1
m_3	8	-81	1

SSD

sid	SSD (GB)	price (USD)	SSD type
d_1	240	-143	1
d_2	512	-300	1
d_3	120	-83	2

Fig. 1. Sample product database.

Top-3 query	Rank-join query
c_3	$\{c_2, m_3, d_3\}$
c_4	$\{c_4, m_1, d_2\}$
c_2	$\{c_4, m_1, d_1\}$

Fig. 2. Top-k and rank-join query results.

rank	id
1	$\{\mathbf{c2}, \mathbf{m3}\}^*$
2	$\{c2, m3, d3\}$
3	$\{\mathbf{c4}, \mathbf{m1}, \mathbf{d2}\}^*$
4	$\{c4, m1, d1\}$
5	$\{c4, m1\}$
6	$\{\mathbf{c3}\}^*$
7	$\{c2, m2\}$
8	$\{c2, m2, d3\}$
9	$\{\mathbf{c1}, \mathbf{m3}, \mathbf{d2}\}^*$
...	
14	$\{c4\}$
15	$\{c2\}$
...	

Fig. 3. Ranking of all possible combinations.

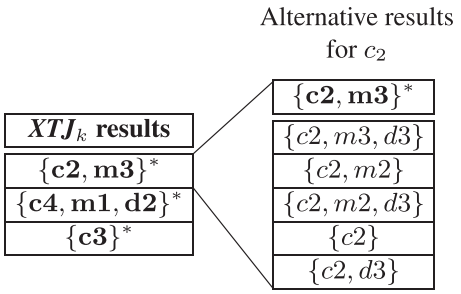

Fig. 4. Grouped results.

better suited to users than single objects. Rank-join queries, on the other hand, focus on fixed-size combinations and do not consider that adding an accessory product does not necessarily result into a more preferable combination, since some not-appealing attributes (e.g., price or weight) may increase with the addition of certain accessories. In addition, they often return similar combinations and thus they present to the user a very limited view of the available products of her interest.

To this end, we propose a new type of query, the eXploratory Top-k Join (XTJ_k) query, which aims at assisting the user to explore the available options by providing her a wide range of the products she is interested in, presented in attractive combinations. In particular, an XTJ_k presents only one combination (the best) per main product, focusing on the products that the user is interested in, while providing possible combinations that may result to a more preferable solution. Different from a rank-join query, where the user specifies the items to be combined, the combination of main and accessory products is performed automatically based on the user preferences. As a result, the user is not required to be aware of the available accessories but she still is presented with interesting combinations. In addition to providing a wide overview of main products, an XTJ_k provides the ability to the user to retrieve more combinations for a specific main product with minimal processing cost, and organizes efficiently the results into groups based on the product the user is interested in.

Consider the example of Fig. 1, which displays the database of an e-shop selling computers, and a user wishing to buy a laptop. We can assume that the user preference vector is equal to $\mathbf{w} = (0.1, 0.2, 0.1, 0.3, 0.2, 0.1)$, where each vector dimension corresponds to a dimension of a laptop item, i.e., $w[1]$ corresponds to CPU, $w[2]$ to RAM, etc. The score of each laptop is equal to $f_{\mathbf{w}} = \sum_{i=1}^{d} \mathbf{w}[i]o[i]$, where $o[i]$ is the normalized value of the i-th attribute of product o. The join attributes "*RAM type*" and "*SSD type*" represent the compatibility between main and accessory items. For instance SSD type 1 represents a normal disk, while SSD type 2 represents an mSATA disk. SSD type 0 represents that no extension can be added to the specific item.[1] A top-3 query and rank-join query would return the results shown in Fig. 2. Both queries do not return the optimal results, as they do not take into account the fact that other combinations may have better scores. Fig. 3 displays the ranking of all possible combinations, and indicates that both queries do not return the best results. In addition, a rank-join query displays two very similar results, limiting the variety of the search results regarding the available products that the user is interested in (laptops). Presenting a ranked list of all possible combinations does help the user to acquire an insight about the available products, as the top-3 combinations (Fig. 3) involve only two laptops.

We argue that a user would be more interested in the combinations presented by an XTJ_3 query as displayed by the first table of Fig. 4. Notice, that an XTJ_k query takes into account all possible combinations, and presents to the user only the best combination for each main product. Upon user request, more combinations for a given product can be presented to the user, providing to her a wide, yet not overwhelming view of the available options. Fig. 4 displays the case where a user wishes to explore alternative combinations for computer c_2. The alternative combinations are organized according to the product of interest of the user, assisting her to explore the available options.

State-of-the-art techniques for processing rank-join queries [6,4,19] return the k highest ranked combinations according to a user-defined preference function. However, the user must be aware of the contents of the database and has to specify at query time the form of combinations. Hence, adaptations of such techniques exhibit sub-optimal performance when applied to our problem. Furthermore the eXploratory Top-k Join is essentially a "star join" of the main product relation and the additional product relations, and existing techniques do not exploit the structure of this join type to achieve performance gains. In this paper, we show how the structure of such queries can be exploited in order to produce efficient query processing algorithms that explore all possible joins, without computing the entire set of possible combinations, and return the correct result.

[1] In the general case, the join condition between two items could be any condition that is evaluated in query-time.

To summarize, the contributions of this paper are the following:

- We propose the use of the pull-bound framework [20] for processing eXploratory Top-k Joins, and we provide a baseline algorithm that processes eXploratory Top-k Join (XTJ_k) queries, by adapting a state-of-the-art rank-join algorithm [4].
- We analyze the properties of eXploratory Top-k Joins and we propose an efficient algorithm (XRJN) that relies on an effective bounding scheme and a plain round-robin pulling strategy.
- We provide strong theoretical guarantees on the performance of our algorithm, namely we prove that XRJN is instance-optimal.
- We present a new algorithm (XRJN*), by introducing a pulling strategy that prioritizes access to relations in a deliberate manner, in order to reduce the overall processing cost.
- We extend the XTJ_k query in order to retrieve multiple combinations (organized in groups) for each main product. Moreover, we provide a generalized version of XRJN*, which supports a wide range of scoring functions, and we prove that XRJN* preserves the property of instance optimality.
- We perform an experimental evaluation that demonstrates the efficiency of our approach.

The definition of the exploratory top-k join together with a basic solution were originally introduced briefly in our previous work [21]. In this paper, we extend our preliminary study substantially and we make several new contributions. We propose a new exploration approach based on XTJ_k queries, which presents initially to the users a wide range of options and subsequently provides them with multiple combinations of a single main product with minimal processing cost. At a conceptual level, we show that our algorithms comply with the pull-bound framework, and can be parameterized by an appropriate bounding scheme and a pulling strategy. In particular, we propose a novel pulling strategy that affects significantly the performance of query processing. Furthermore, we provide theoretical guarantee on the performance of our algorithms and analyze their complexity. Finally, we conduct a thorough experimental study using both synthetic and real data.

The rest of this paper is organized as follows: Section 2 reviews the related work. In Section 3 we formally define the XTJ_k query. Thereafter, in Section 4, we present a baseline technique to answer an XTJ_k query. In Section 5, we provide a more efficient algorithm that exploits the characteristics of the combinations in order to produce faster converging upper and lower bounds. Section 6 describes the generation of multiple combinations per main object, as well as the generalization of the aggregation function. The experimental evaluation is presented in Section 7 and we conclude in Section 8.

2. Related work

Our work is related to top-k and join queries as well as package recommendation. In the following, we present an overview of the related research work.

2.1. Top-k and Rank-join queries

Top-k queries have been well-studied in the last years to enable ranked retrieval of objects based on user preferences. Top-k queries where first studied by Fagin et al. [2]. Das et al. [1] introduced an algorithm using views. Ge et al. [8] follow a similar approach using precomputed views but their goal is to improve the performance on batch top-k queries. Their approach shows a special interest as they avoid linear programming for calculating the upper bound. For a thorough overview of top-k queries we refer to [3].

Rank join queries were first studied by Natsev et al. who introduced the J* algorithm [22]. Ilyas et al. proposed the HRJN* algorithm [4] which outperforms J*. Mamoulis et al. [5] introduced the LARA-J and LARA-J* algorithms which use lattices in order to store partial join results. The performance of LARA-J* is better than HRJN* with respect to access depth but the algorithm induces processing cost which is higher than that of HRJN*. Finger and Polyzotis [6] and Schnaitter and Polyzotis [20,19] study the problem of finding tight bounds for terminating the rank-join algorithms. They also proposed the a-FRPA algorithm, a hybrid approach between a tight bound and HRJN*, which has improved performance over HRJN* in low data dimensionality. In higher dimensionality the performance advantage is minimal and the performance of the algorithm is on the same levels as HRJN*. Martinenghi and Tagliasacchi [23] study the problem of joining results produced by different sources on the Web for which the access cost varies. They assume that both sorted and random access are available and propose an algorithm for determining an efficient pulling strategy at compile time which takes into account the access cost for each source. Habich et al. [24] address the problem of increasing the overall performance of multiple top-k queries over joins. Their main difference is that while the previous approaches assume that the data are sorted according to the preference queries they propose a strategy where they avoid sorting the relations for each top-k query by using a global sorting for merge-joins of the tuples or by using a variation of the hash-join algorithm. Agrawal and Widom [25] discuss the subject of confidence aware rank-join algorithms. Xie et al. [26] study the problem of rank joins with aggregation constraints while in [27] Lu et al. introduce the top-k,m queries. Given a set of groups where each group contains a set of attributes, they study the problem of finding the best combination of attributes. They focus on ranking combinations of attributes and not combinations of objects. Zhang et al. [28] study the problem of finding the best combinations of objects on a graph. Khalefa et al. [29] optimize the performance of preference joins with the use of pruning techniques. Jin et al. study the problem of multi-relational skylines [30] and skylines over equi-joins [31], while Doulkeridis et al. [32] study the problem of rank-join queries over distributed systems.

Our main difference with the aforementioned approaches is that we do not assume that combinations should have a fixed number of elements but combinations of any size can be eligible. In addition, we examine a specific case of joins where all additional objects are combined with a main object. This case of “star”-join has specific characteristics which allow us to improve the performance of processing of such joins. Moreover, we are considering only the best combination of each main object which enables us to offer a wider view of the available main objects which are the objects the user is primarily interested in.

2.2. Package retrieval and recommendation

Angel et al. [33] propose the class of “entity package finder” queries and algorithms that process this query efficiently. Entities can be hotels, cities or airlines, and the query aims to find top-scored combinations of entities based on fixed associations. Entities are scored based on associated documents and their relevance to query keywords. Even if this difference to our work is set aside, their problem setting is practically a join over ranked lists, not relations with multiple common attributes whose values are aggregated.

Guo and Ishikawa [34] try to find packages that are not dominated by other packages. Packages are defined as combinations of objects coming from the same relation, and any two objects can form a combination. Practically, this problem is quite different from our work, as it does not deal with the problem of joining and ranking join tuples.

Our work is also related to package recommendation [35–38]. Xie et al. [37] study the problem of creating the best package out of a set of items given a specific budget. However, the objects are not related to each other and the problem they address is to create the most attractive package of objects for a user. Combinability of objects is not taken into account and it is assumed that all combinations are possible.

Roy et al. [36] suggest a method of constructing combinations

based on a central object and a set of satellite objects but they do not take into account preference vectors. Their effort focuses on creating and presenting a number of packages which maximizes the variety of the contained satellite objects and satisfies at the same time a budget constraint.

Amer-Yahia et al. [38] aim to retrieve combinations of items that satisfy a budget constraint and they are complementary regarding an attribute, i.e., they have different values for that attribute, but they are similar regarding the rest attributes. In their work, they consider all items to belong in one category, e.g., restaurants, and they rank the combinations according to the similarity of the objects, which is totally different than our approach that employs user preferences for ranking. Also they do not study the case of efficient processing of joins, nor the combination of ranking and joins.

Our main difference lies on the fact that the aforementioned approaches aim at finding groups of objects that are attractive to users and possibly satisfy certain constraints, such as budget constraints. However, they do not take into consideration the item the user is searching for and therefore the suggested accessory items may not be relevant to the user query. Current approaches are focused on suggesting accessory items that are likely to interest the user but are not necessarily relevant to the properties the user is interested in. On the contrary, we focus on the item that the user primarily focuses on, and combine it with accessory items that enhance the properties that are most important for the user. Finally, we do not assume that all combinations are possible but we evaluate the join conditions as well. None of the aforementioned approaches examine all these conditions simultaneously.

3. Problem Definition

In this section we formally define the XTJ_k query and all necessary structures used both for the problem definition and the description of the respective algorithms. Table 1 summarizes the main symbols used in this paper.

3.1. Object combinations

Let D be a database of objects and E_M be a relation in D which is connected to a set of relations $\mathcal{E} = \{E_1, \ldots, E_n\}$ of D. E_M has a set of d real valued attributes $A_{E_M} = \{a_1, \ldots, a_d\}$ and each relation E_i contains a subset $A_{E_i} \subseteq A_{E_M}$ of these attributes, i.e., it holds that $\forall\, E_i \in \mathcal{E}, A_{E_i} \bigcap A_{E_M} \neq \emptyset$. Each object in a relation $E \in \mathcal{E} \bigcup \{E_M\}$ is represented as a d-dimensional point $p \in \mathbb{R}^d$ where $p[i] \in \mathbb{R}$ if $a_i \in A_E$ and $p[i] = 0$ if $a_i \notin A_E$. We refer to E_M as the *main relation* and to the rest of the relations as *the additional relations*. The objects of the relations are called *main* and *additional objects*.

Using the main and the additional objects we can form combinations where each combination has exactly one main object and at most one object from each additional relation. We say that an object of the main entity relation $o \in E_M$ and a object $p \in E_i$ of an additional relation E_i are combinable if there is a join of the form $o \bowtie p \in E_M \bowtie E_i$.

Table 1
Table of symbols.

Symbol	Explanation
E_M	The main relation
E_i	An additional relation to E_M
HE	Set of accessed objects of relation E
o	Object of E_M
p	Object of any relation E_M or E_i
c	A combination of objects
$f_{\mathbf{w}}(p), f_{\mathbf{w}}(c)$	Score of an object p and combination c
$m(c)$	The main object of a combination
c_{mp}	The most promising combination
$C(E_M)$	All possible combinations with E_M as main relation
$B(E_M)$	All candidate combinations with E_M as main relation
$XTJ_k(\mathbf{w})$	top-k candidate combinations
$ALT(o)$	Set of alternative combinations for a main object o

Definition 1 (*Combination*). Given a main relation E_M and a set of relations $\mathcal{E}$ we define as a combination a set of objects c such that:

- $\exists\, o \in c$: $o \in E_M$, $|E_M \bigcap c| = 1$,
- $\forall\, p \in c$, $p \neq o$ it holds that $\exists\, E_i$: $o \bowtie p \in E_M \bowtie E_i$ and
- $\forall\, p_i, p_j \in c$, $i \neq j$, $p_i \in E_i$, $p_j \in E_j$ it holds that $E_i \neq E_j$.

We use the notation $m(c)$ to denote the main object of a combination c and $C(E_M)$ to denote the set of all possible combinations that can be formed using E_M as main relation. Note that a main object can participate in many combinations, but each combination has only one main object.

Using the example database in Fig. 1, if a user wishes to buy a laptop and she is interested in CPU, RAM, SSD size and price, then the main relation of her query is *Laptops* and the additional relations are *Memory* and *SSD*. Any other attributes not specified in the query can be considered irrelevant. The set of objects $\{c_2, m_3, d_3\}$ is a valid combination, while $\{c_1, m_1\}$ is not since c_1 and m_1 are not combinable. Fig. 3 shows some of the combinations in C(Laptops) which is the set of all possible combinations with *Laptops* as the main relation.

3.2. Ranking combinations

We can now extend the notion of a top-k query in order to take into account not only single objects but combinations as well. We therefore define the *Exploratory Top-k Join* (XTJ_k) query which returns the top-k combinations with distinct main objects. We consider a user query to be a d-dimensional preference vector $\mathbf{w}$ targeted to relation E_M and each dimension $\mathbf{w}[i]$ of the query to represent the importance of the respective attribute to the user. Without loss of generality we assume that $\mathbf{w}[i] \geq 0$, $\sum_{i=1}^{d} \mathbf{w}[i] = 1$ and if a user is not interested in a specific attribute a_i of the main objects then $\mathbf{w}[i] = 0$. Given a preference vector $\mathbf{w}$ targeted to the main relation E_M, the score of a combination c is defined as: $f_{\mathbf{w}}(c) = \sum_{j=1}^{d} \mathbf{w}[j] \sum_{p \in c} (p[j])$.

An XTJ_k query lists the k main objects with the best combinations and thus each main object can appear at most once in the query's result-set. The following definition formally defines the problem addressed in this paper.

Definition 2 (XTJ_k *query*). Given a main relation E_M, a set of additional relations $\mathcal{E}$, a preference vector $\mathbf{w}$ and an integer k, the result set $XTJ_k(\mathbf{w})$ of an Exploratory Top-k Join query is a set of combinations such that:

- $XTJ_k(\mathbf{w}) \subseteq C(E_M)$ and $|XTJ_k(\mathbf{w})| = k$,
- $\forall\, c_1, c_2 \in XTJ_k(\mathbf{w})$ it holds that $m(c_1) \neq m(c_2)$,
- $\forall\, c_1 \in XTJ_k(\mathbf{w})$, $c_2 \in C(E_M) - XTJ_k(\mathbf{w})$ one of the following necessarily holds:
  - $f_{\mathbf{w}}(c_1) \geq f_{\mathbf{w}}(c_2)$ or
  - $\exists\, c' \in XTJ_k(\mathbf{w})$: $m(c') = m(c_2)$ and $f_{\mathbf{w}}(c') \geq f_{\mathbf{w}}(c_2)$.

Returning to our example, Fig. 3 lists the ranked set $C(E_M)$, while the result of an XTJ_2 query are the combinations $\{c_2, m_3\}$ and $\{c_4, m_1, d_2\}$. There is a combination $\{c_2, m_3, d_3\}$ which has a better score than $\{c_4, m_1, d_2\}$, but it is omitted since it shares the same main object (c_2) with the top-1 result.

3.3. Theoretical properties

In the following, we present some properties of the combinations that helps us to reduce the search space of the XTJ_k query.

A combination that no other tuple can be added to and improve the score of the combination is called *total combination*. A total combina-

tion does not necessarily contain objects from every additional relation. Given the fact that in the general case $p \in \mathbb{R}^d$, the score of an additional object could be negative and the addition of such an object to a combination would make the score of the combination worse. In such cases a total combination may not contain objects from all additional relations. We should note that a main object may also have a negative score, however, a combination should always contain a main object even if its score is negative.

As mentioned before, each object of the main relation can participate in many combinations. However, for each main object we are interested only in the combination with the best score, i.e., the *candidate combination*.

Definition 3 (*Candidate combination***).** Given a main object o of the main relation E_M, the candidate combination c is a combination such that $\forall\ c' \neq c$: $m(c') = m(c) = o$ it holds that $f_{\mathbf{w}}(c) \geq f_{\mathbf{w}}(c')$.

We denote the set of all candidate combinations as $B(E_M)$. Obviously $B(E_M) \subseteq C(E_M)$. Returning to our example, Fig. 3 lists the set $C(E_M)$, while the set of the candidate combinations $B(E_M)$ is indicated with a star (*).

Lemma 1. *A candidate combination is total.*

Proof. By contradiction. Assume that the candidate combination c of object o is not total. Then, there exists a combination $c' \neq c$, such that $m(c) = m(c') = o$ and it holds that , where p is an additional object, and also $f_{\mathbf{w}}(c') > f_{\mathbf{w}}(c)$. This contradicts with Definition 3.□

The opposite does not hold. There can be many total combinations with the same main object and no other common object, but only one of them can be candidate combination as well.

Lemma 2. *It holds that* $XTJ_k(\mathbf{w}) \subseteq B(E_M)$.

Lemma 2 is easy to be proven and it shows that it is sufficient to examine only candidate combinations during the processing of a XTJ_k query.

4. Pull-bound framework

In this paper, we propose a pull-bound framework for XTJ_k queries. The pull-bound framework is based on the assumption that access to the objects of each relation is provided in descending order of score.[2] The score of an object p_i is equal to $f_{\mathbf{w}}(p_i) = \sum_{j=1}^{d} \mathbf{w}[i]p_i[j]$ and it is essentially the contribution to the total score of the combination it belongs to. In other words, for any relation, if object p_1 is accessed before p_2, this means that $f_{\mathbf{w}}(p_1) \geq f_{\mathbf{w}}(p_2)$.

The general structure of the family of algorithms that comply with this framework is shown in Algorithm 1. Their difference lies on (a) the bounding technique that calculates the upper bound of the possible score of any unseen combination, and (b) the pulling technique that determines the next relation to access.

In the following, we first introduce our pull-bound framework for processing XTJ_k queries. Then, we adapt an existing rank-join algorithm, namely *HRJN** [4], in order to be able to process exploratory top-k joins.[3] Since the calculated bounds play an important role on the behavior of the pulling strategy, we are going to analyze the bounding technique first. We employ the modified HRJN* algorithm (MHRJN) as a baseline to compare the performance of our algorithms.

4.1. XTJ$_\mathbf{k}$ *framework*

Algorithm 1. Pull-bound framework.

Input: E_M, $\mathcal{E}$
Output: $XTJ_k(\mathbf{w})$

1: $\widehat{B}(E_M) \leftarrow \varnothing$ //*Set of produced candidate combinations*
2: **while** $|\widehat{B}(E_M)| < k$ OR $LB < UB$ **do**
3: $E \leftarrow$ chooseRelation($\mathcal{E} \bigcup \{E_M\}$)
4: $p \leftarrow E$. pullTuple()
5: $HE \leftarrow HE \cup \{p\}$ //*add p to the accessed objects*
6: $\widehat{B}(E_M) \leftarrow$ update(HE_M, HE_1…, t, …, HE_n)
7: $LB \leftarrow$ kBest($\widehat{B}(E_M)$)
8: $UB \leftarrow$ upperBound($\widehat{B}(E_M)$)
9: **end while**
10: **return** $topK(\widehat{B}(E_M))$

As shown in Algorithm 1, the pull-bound framework consists of a loop which is executed until k join results have been produced and no unseen tuple can produce a join result with better score. In the first step, the next relation to be accessed is selected (line 3) based on a pulling strategy. Given a pulled tuple p from that relation, we update the set of produced combinations (line 6). Finally, the lower bound is set as the score of the k-th join result (line 7) and the upper bound of any unseen join result is computed (line 8) based on the bounding scheme.

Now, we focus on how the combinations are generated (line 6). Since we are interested only in candidate combinations, the method *update*() combines a newly pulled object with a combination only if that is beneficial for the combination. The generated candidate combinations are maintained as a set $\widehat{B}(E_M)$ that contains the best *seen* combination for each main object. In detail, if the accessed tuple p refers to a main object, the method *update*() finds the best additional objects of the already accessed tuples which are combinable with p, creates the combination and adds it to $\widehat{B}(E_M)$. If tuple p refers to an additional object, then we add this object to all combinations that the tuple can be added to, i.e., to all combinations that p is combinable with the main object of the combination and the addition of p results to an improved score of the combination. In that way we ensure that $\widehat{B}(E_M)$ contains only the best combination of each main object, considering of course only the accessed tuples.

As a result, the set $B(E_M)$ is computed incrementally in Algorithm 1, which means that in worst case where all objects of the relations are accessed, then $\widehat{B}(E_M)$ will be equal to $B(E_M)$. However, in practice, the algorithm will halt much earlier, thus avoiding the cost of materializing the set $B(E_M)$.

4.2. *Modified HRJN** *algorithm*

Bounding scheme: Our pull-bound framework evaluates a XTJ_k query by estimating the upper bound of the score that any unseen tuple can produce and terminates when the k-th best join result found is better than the upper bound. Recall that the upper bound (UB_{MHRJN}) of the MHRJN algorithm is the maximum value produced if we combine the worst seen tuple of any relation with the best seen tuples of the remaining relations [4]. For the additional relations we include the best tuples only if their score is positive and they can increase that way the total score of the combination. We must include however the score of the best tuple of the main relation since its presence in the results is necessary. The bounding scheme of MHRJN is formally described in

$$UB_{E_M} = f_{\mathbf{w}}(HE_M[last]) + \sum_{E \in \mathcal{E}} u(f_{\mathbf{w}}(HE[1])) \tag{1}$$

$$UB_{E_i} = f_{\mathbf{w}}(HE_M[1]) + f_{\mathbf{w}}(HE_i[last]) + \sum_{\substack{E \in \mathcal{E} \\ E \neq E_i}} u(f_{\mathbf{w}}(HE[1])) \tag{2}$$

$$UB_{\text{MHRJN}} = \max_{E \in \mathcal{E} \bigcup \{E_M\}} (UB_E) \tag{3}$$

[2] Usually this is achieved by the use of multidimensional indexes or materialized views.
[3] We henceforth use MHRJN to refer to the modified version of HRJN*.

The notation HE denotes the set of accessed objects of relation E and by $HE[1]$, $HE[last]$ we denote the first and last accessed tuples. The function $u(x)$ returns x if $x > 0$ and 0 if $x \leq 0$. The complete algorithm that calculates the bound is described in Algorithm 2.

Algorithm 2. MHRJN bound.

Input: E_M, $\mathcal{E}$
Output: UB_{MHRJN}
1: UB, bound $= -\infty$
2: **for all** $E \in \mathcal{E} \bigcup \{E_M\}$ **do**
3: **if** $E = E_M$ **then**
4: bound $= f_{\mathbf{w}}(HE[last]) + \sum_{E' \in \mathcal{E}} u(f_{\mathbf{w}}(HE[1]))$
5: **else**
6: bound $= f_{\mathbf{w}}(HE[last]) + f_{\mathbf{w}}(HE_M[1]) + \sum_{\substack{E' \in \mathcal{E} \\ E' \neq E}} u(f_{\mathbf{w}}(HE[1]))$
7: **end if**
8: **if** bound $> UB$ **then**
9: $UB \leftarrow$ bound
10: E becomes the next relation to pull from
11: **end if**
12: **end for**
13: **return** UB

Theorem 1 (*Correctness of bound*). *The modified version of HRJN* provides a correct solution to the Exploratory top − k join problem.*

Proof. We assume that MHRJN stops after having accessed d_0 tuples for E_M and d_i tuples for each additional relation E_i. Let $c_k = \{E_M[j_0], E_1[j_1], \ldots, E_n[j_n]\}$ be the k-best combination, i.e., $LB = f_{\mathbf{w}}(c_k)$ and let the upper bound be $UB = UB(E_\zeta)$, $E_\zeta \in \{E_M\} \bigcup \mathcal{E}$. Since the algorithm has stopped then the condition of Inequality (4) holds:

$$f_{\mathbf{w}}(c_k) \geq UB \tag{4}$$

$$\sum_{E_j[i_j] \in c_k} f_{\mathbf{w}}(E_j[i_j]) \geq f_{\mathbf{w}}(E_\zeta[d_\zeta]) + \sum_{\substack{E_j \in \mathcal{E} \bigcup E_M \\ E_j \neq E_\zeta}} f_{\mathbf{w}}(E_j[1]) \tag{5}$$

Based on Eqs. (1)–(3) we conclude that Inequality 5 holds for any relation $E_\zeta \in \mathcal{E} \bigcup \{E_M\}$. Let us assume that there is a non-total combination c' which if combined with an unseen tuple will become better than c_k. Consequently, after joining c' with the unseen tuple it will hold that $f_w(c') > f_w(c_k)$. Since c' contains an unseen tuple, it contains a tuple $E_\tau[d_\tau + x]$ of a relation E_τ. The maximum score of c' is therefore given by either Eq. (1) if $E_\tau = E_M$ or in the opposite case, by Eq. (2) where we consider the last tuple to be $E_\tau[d_\tau + x]$. We have assumed that c' is better than c_k, therefore, if we substitute E_ζ with E_τ in Inequality 5 then we get that $f_{\mathbf{w}}(E_\tau[d_\tau + x]) \geq f_{\mathbf{w}}(E_\tau[d_\tau])$ which is not true because we are accessing the objects in descending order of score. This contradicts with the assumption that a non-total combination combined with an unseen object may produce a top-k result. We reach therefore the conclusion that no unseen object can produce a better combination after the stopping criterion of MHRJN is satisfied.□

Pulling strategy: The set of bounds calculated by MHRJN in Algorithm 2 is used to decide which is the next relation to pull from. The intuition indicates that we should pull from the relation that produced the highest bound since this relation plays an important role in the upper bound of the algorithm. If we pull from the other relations, the upper bound will not be reduced significantly and the algorithm will not terminate fast. Therefore the decision on which is the next relation to pull is taken in line 10 in Algorithm 2.

5. Exploratory rank-join (XRJN)

In this section, we propose the exploratory rank-join algorithm (*XRJN*), which also follows the pull-bound framework. We propose a tighter bounding scheme (Section 5.1) and prove its correctness (Section 5.2), and we show that the bounding scheme of *XRJN* has strong theoretical guarantees on its performance, namely that it is instance-optimal (Section 5.3). In addition, we present a lazy method to compute the bound more efficiently (Section 5.4), and we analyze the complexity of the proposed algorithm (Section 5.5). Finally, we present a pulling strategy that is beneficial for the proposed bounding scheme (Section 5.6).

5.1. Bounding scheme

At a random state of the algorithm, let HE denote the objects of a relation E that have been accessed so far, and $\widehat{B}(E_M)$ the set of all combinations that have been created so far. Recall at this point that only one combination per main object is created and that an additional object is added to a combination only if this produces a better score for the combination. There are two bounds we should consider, denoted as UB_{E_M} and UB_{comb} respectively, and our bounding scheme computes their maximum:

$$UB_{XRJN} = max(UB_{E_M}, UB_{comb}) \tag{6}$$

The first bound (UB_{E_M}) determines the upper bound of any unseen combination, i.e., any unseen object of the main relation E_M. In the best case, the next object of the main relation to be accessed will be combined with the best objects of the additional relations except for those that have a negative score. Obviously, the upper bound for any unseen object of the main relation is the same with the upper bound calculated by the baseline and its value is calculated by Eq. (1).

The second bound (UB_{comb}) represents the best score of a seen main product combined with at least one unseen additional object. For any seen main object, there exists exactly one combination in $\widehat{B}(E_M)$, since any retrieved main object will be added to it as a combination with a single main object if the main object cannot be combined with any additional objects. A combination c in $\widehat{B}(E_M)$ is total based on the seen tuples, if its score cannot be improved further. In other words, c is total if for all relations E_i either (a) there exists $p_i \in E_i$ such that $p_i \in c$, or (b) E_i is exhausted (all tuples have been accessed) or $HE_i[last]$ is negative. In the following, we refer to a combination c of $\widehat{B}(E_M)$ that is not total as non-total combination and we refer as missing relation of c all relations E_i for which (a) or (b) does not hold.

Let c be a non-total combination and E_i be a missing relation of c, then c can be combined with an unseen object of E_i and therefore the maximum contribution of E_i is equal to $f_{\mathbf{w}}(HE_i[last])$. Thus, the upper bound of the score for a non-total combination c can be computed be adding for every missing relation E_i the score of the last accessed object of E_i. We call most promising combination, the non-total combination c_{mp} which has the highest upper bound on its score. The upper bound UB_{comb} of all seen main products is determined by the upper bound of the score of the most promising combination.

Eq. (7) defines formally the most promising combination while Eq. (8) calculates the respective upper bound[4]

$$c_{mp} = \underset{\substack{c \in \widehat{B}(E_M) \\ c \ \ not \ \ total}}{\text{argmax}} \left(f_{\mathbf{w}}(c) + \sum_{\substack{E \in \mathcal{E} \\ E \ missing \ relation}} u(f_{\mathbf{w}}(HE[last])) \right) \tag{7}$$

$$UB_{comb} = f_{\mathbf{w}}(c_{mp}) + \sum_{\substack{E \in \mathcal{E} \\ E \ missing \ relation}} u(f_{\mathbf{w}}(HE[last])) \tag{8}$$

Example 1. Assume the objects shown in Table 2 where E_M is the main relation, E_1, E_2 are the additional relations, and we are looking for the top-1 combination. The table shows the scores of the objects

[4] We should note that this bound is the same for all additional relations, Therefore we call it UB_{comb} because it depends on the current combinations and the last objects accessed.

Table 2
Example.

id	E_M	E_1	E_2
1	10^*	7^+	5
2	10	6	4
3	8	6^*	3
4	7^+	5	$3^{*,+}$
5	6	5	3
6	5	5	3
7	4	3	2
8	3	3	2
9	2	2	1
10	1	1	1

according to a given vector **w**. The two signs (*,+) show two possible combinations. We assume that we read the tuples in a round-robin fashion. The id of each tuple is the row number and the relation letter it belongs to (e.g., $E_M[1]$, $E_1[3]$, etc.). After having read the first row, we have one combination $c_1 = \{E_M[1]\}$ and the upper bound UB is equal to $UB = f_{\mathbf{w}}(E_M[1]) + f_{\mathbf{w}}(E_1[1]) + f_{\mathbf{w}}(E_2[1]) = 22$. The lower bound is the best score of any combination found so far which is the score of the main object $E_M[1]$, thus $LB = f_{\mathbf{w}}(E_M[1]) = 10$.

When the second row is read, the combination $c_2 = \{E_M[2]\}$ is formed and the upper bound of any unseen combination involving any unseen main object is still equal to $UB_{E_M} = 22$. The most promising combination is $c_1 = \{E_M[1]\}$ and the upper bound of its score is equal to $UB_{comb} = f_{\mathbf{w}}(c_1) + f_{\mathbf{w}}(E_1[2]) + f_{\mathbf{w}}(E_2[2]) = 20$. At this point we are not considering the tuples $E_1[1]$, $E_2[1]$ because if any of them were combinable with $E_M[1]$ or $E_M[2]$ the combination would have been formed. Of the two bounds we pick the maximum and therefore the upper bound is equal to $UB = 22$.

Our algorithm continues by creating the combination $c_3 = \{E_M[3]\}$ and then the combination c_1 is updated and becomes equal to $c_1 = \{E_M[1], E_1[3]\}$. After the third row is read, the most promising combination is c_1 and therefore the upper bounds are formed as following: $UB_{E_M} = 20$, $UB_{comb} = 19$. The lower bound LB is now equal to $LB = 16$ due to the update of c_1.

After the fourth row has been read then we have that $UB_{E_M} = 19$. The combinations created are $c_1 = \{E_M[1], E_1[3], E_2[4]\}$, $c_2 = \{E_M[2]\}$, $c_3 = \{E_M[3]\}$ and $c_4 = \{E_M[4], E_1[1], E_2[4]\}$. The lower bound is now equal to $LB = 19$. The most promising combination is c_2 since both c_1 and c_4 are total. The upper bound for the combinations UB_{comb} is equal to $UB_{comb} = 18$ and therefore $UB = UB_{E_M} = LB$ and so the algorithm stops.

According to the MHRJN bound, the upper bound at the fourth row would be equal to $UB_{E_1} = f_{\mathbf{w}}(E_M[1]) + f_{\mathbf{w}}(E_1[4]) + f_{\mathbf{w}}(E_2[1]) = 20$ and therefore extra tuples would be read until $E_1[7]$ and $E_2[7]$ were accessed where the upper bound would become equal to $UB = 19$. At this point it is clear why the new bounding technique offers a tighter estimation of the bound. First, the MHRJN bound technique is overestimating the score of the unseen combinations because it uses uncombinable tuples, and second, because it does not exclude main objects of total combinations whose score cannot be improved.

Algorithm 3. Bounding scheme of XRJN.

Input: $\mathcal{E}$, $\widehat{B}(E_M)$:set of all generated combinations.
Output: UB_{XRJN}
1: $UB_{E_M} \leftarrow f_{\mathbf{w}}(HE_M[last]) + \sum_{E \in \mathcal{E}} u(f_{\mathbf{w}}(HE[1]))$
2: **for all** $c \in \widehat{B}(E_M)$ and c *not total* **do**
3: $c.UB \leftarrow c.score$
4: **for all** $E \in$ missingRelations of c **do**
5: $c.UB \leftarrow c.UB + u(f_{\mathbf{w}}(HE[last]))$ //*max possible score for each comb.*
6: **if** $UB_{comb} < c.UB$ **then**
7: $c_{mp} \leftarrow c$
8: $UB_{comb} \leftarrow c.UB$
9: **end if**
10: **end for**
11: **end for**
12: **return** $max(UB_{E_M}, UB_{comb})$

5.2. Algorithm and correctness

Algorithm 3 calculates the upper bound of the score for any unseen combination. In line 1 we calculate UB_{E_M} which is the upper bound of any combination involving any *unseen* objects of the main relation.

In line 2 we evaluate the already created non-total combinations which can be combined with any unseen tuples. For each non-total combination c in $\widehat{B}(E_M)$ we calculate the upper bound of its score by adding the score of the last seen tuples for all missing relations of c. The combination with the highest upper bound of its score ($c.UB$) is the most promising combination (lines 2–11). UB_{comb} is equal to the score of the most promising combination. The upper bound UB returned by the algorithm is the maximum of these two bounds (line 12).

Theorem 2 (*Correctness of bound***).** *The* XRJN *algorithm provides a correct solution to the* XTJ_k *query.*

Proof. Let us assume that the algorithm has halted, after having accessed d_0 tuples for the relation E_M and d_i tuples for each relation $E_i \in \mathcal{E}$, therefore $UB \leq LB$. Let now c' be an unseen combination for which it holds that $f_{\mathbf{w}}(c') > LB$.

If the unseen combination c' contains an unseen main object, then the score of the combination will be at most equal to UB_{E_M}, thus $f_{\mathbf{w}}(c') \leq UB_{E_M}$. Since $UB_{E_M} \leq UB \leq LB$ it holds that $f_{\mathbf{w}}(c') \leq LB$ which contradicts with the assumption $f_{\mathbf{w}}(c') > LB$. The contradiction that we have reached is due to our assumption that c' contains an unseen main object and it is better than LB.

Alternatively, let us assume that unseen combination c' contains an already accessed main object, which means that there exists a non-total combination $c \in \widehat{B}(E_M)$ that will be combined with at least one unseen additional object and produce c'. In the following we assume that c has been combined with an unseen object from only one relation E_j producing combination c'. Similarly, we can prove the general case of more than one relations. Let $E_j[d_j + x]$ be the first combinable object with c. Then the score of c' when joined with the unseen tuple will be equal to: $f_{\mathbf{w}}(c') = f_{\mathbf{w}}(c) + f_{\mathbf{w}}(E_j[d_j + x]) \leq f_{\mathbf{w}}(c) + f_{\mathbf{w}}(E_j[d_j])$. From Eq. (8) we conclude that $f_{\mathbf{w}}(c') \leq UB_{comb} \leq UB \leq LB$. However in the beginning we assumed that $f_{\mathbf{w}}(c') > LB$ and this is a contradiction. We conclude that an unseen combination cannot have a greater score than the score of the k-th combination in $\widehat{B}(E_M)$, thus our algorithm returns always the correct result set.□

5.3. Instance optimality

Instance optimality is defined by Fagin et al. [2] as follows. Given a class of algorithms $\mathcal{A}$ and a set of databases $\mathcal{D}$, an algorithm $A \in \mathcal{A}$ is instance-optimal if $\forall B \in \mathcal{A}$ and $\forall D \in \mathcal{D}$ it holds that $\text{cost}(A, D) = O(cost(B, D))$. This means that there are constantsc_1, c_2 such that $\text{cost}(A, D) \leq c_1\text{cost}(B, D) + c_2$. Constant c_1 is referred to as *optimality ratio*.

Lemma 3. MHRJN *is not instance optimal for the* XTJ_k *query.*

Proof. Based on the definition of instance optimality it is sufficient to show that the cost of MHRJN is not bounded for one instance database D' compared to an algorithm A. Thus, we construct a data set for which the cost of MHRJN is not bounded compared to XRJN. Denoting the database of Table 2 as D, we consider a database D'. Each relation E_M, E_1, and E_2 of D' has the same first 4 tuples as D. Thus, XRJN will return

the correct answer after accessing the first 4 tuples of each relation but MHRJN does not terminate and will continue reading more tuples. Let us assume that the relations of D' contain more than 4 tuples such that $E_M[i] = E_M[4]$, $E_1[i] = E_1[4]$, $E_2[i] = E_2[4]$ for $i > 4$. MHRJN has to access at least all tuples of relation E_2 before terminating, while XRJN needs to access only the first four rows. Since relation E_2 can be arbitrarily long the cost of MHRJN cannot be bounded and therefore MHRJN is not instance optimal.□

Theorem 3. XRJN *is instance optimal within algorithms following the Pull-Bound Framework with optimality ratio n+1 where n is the number of additional relations.*

Proof. Let A be a random deterministic algorithm solving correctly the XTJ_k query for a vector $\mathbf{w}$. The algorithm halts after having accessed d_0 tuples for E_M and d_i tuples for each additional relation E_i. We define c_k to be the k best candidate combination discovered by A with score $f_{\mathbf{w}}(c_k)$ and d_{max} to be equal to $d_{max} = \max_{0\le i\le n}(d_i)$. We will show by contradiction that XRJN will halt after accessing at most d_{max} tuples from each relation.

Let us assume that XRJN has accessed d_{max} tuples from each relation and has not halted. At this point XRJN has processed at least all combinations evaluated by A and therefore it holds that $LB_{\text{XRJN}} = f_{\mathbf{w}}(c_k)$. Under the assumption that XRJN has not halted, there are two cases to be examined.

The first case is that the upper bound of XRJN at that step is defined by an unseen object of the E_M i.e., $UB_{\text{XRJN}} = UB_{E_M}$ and it holds that $LB_{\text{XRJN}} < UB_{\text{XRJN}}$ and since $LB_{\text{XRJN}} = f_{\mathbf{w}}(c_k)$ it also holds that $f_{\mathbf{w}}(c_k) < UB_{\text{XRJN}}$. As algorithm A is deterministic, it must halt at the same step for all instances of relations that have the same seen tuples HE_M, HE_i. We can construct relation E_M such that $HE_M[d_0 + 1]$ is combinable with the first tuples of all additional relations $HE_i[1]$ and $f_{\mathbf{w}}(E_M[d_0 + 1]) = f_{\mathbf{w}}(E_M[d_0])$. For the combination c' defined by $\{HE_M[d_0 + 1], HE_1[1], \ldots, HE_n[1]\}$ it holds that $f_{\mathbf{w}}(c') = UB_{\text{XRJN}}$, thus $f_{\mathbf{w}}(c_k) < f_{\mathbf{w}}(c')$. Therefore A has halted incorrectly, which leads us to a contradiction.

In the second case it holds that $UB_{\text{XRJN}} = UB_{comb}$ and $LB_{\text{XRJN}} < UB_{\text{XRJN}}$. Let c_{mp} be the most promising combination found by XRJN and $o = m(c_{mp})$ be the main object of c_{mp}. An instance of our database D can be constructed in way that $\forall E_i$: $\nexists p_i \in c_{mp}$, $p_i \in E_i$ it holds that $f_{\mathbf{w}}(E_i[d_{max}]) = f_{\mathbf{w}}(E_i[d_{max} + 1])$ and all $E_i[d_{max} + 1]$ tuples are combinable with o. In this case, when $d_{max} + 1$ tuples has been read from all relations a new combination $c_{mp}{}'$ is produced by updating c_{mp} and adding the newly pulled tuples. It halts that $f_{\mathbf{w}}(c_{mp}{}') = UB_{\text{XRJN}}$ and therefore also $f_{\mathbf{w}}(c_k) < f_{\mathbf{w}}(c_{mp})$. Thus, $c_{mp}{}'$ belongs to the result set and A has halted incorrectly, which leads us to a contradiction.

We conclude that XRJN will halt after accessing at most d_{max} tuples from each relation. Thus, the cost of XRJN in terms of accessed tuples is at most $\text{cost}(\text{XRJN}, D) = (n + 1)*d_{max}$, while the cost of algorithm A is $\text{cost}(A, D) = \sum_{0\le i\le n} d_i$. We can derive that $\text{cost}(\text{XRJN}, D) \le (n + 1)*\text{cost}(A, D)$ since $d_{max} \le \sum_{0\le i\le n} d_i$. Hence XRJN is instance optimal with optimality ratio n+1.□

5.4. Lazy upper bound evaluation

The processing cost of the XRJN* upper bound is determined by the size of the $\widehat{B}(E_M)$ set as it is necessary to search the entire set every time we need to find the most promising combination. However, we can reduce the overall cost of the calculation if we postpone the accurate calculation of the upper bound until it is absolutely necessary. To achieve that, we calculate the highest possible score of a combination when the combination is updated and we update the most promising combination if necessary. This approach does not take into consideration the fact that when a tuple is read, it affects the maximum possible score of possibly all non-total combinations and not only the updated ones. As a result, it is possible that the most promising combination is a combination which was not updated and therefore UB_{comb} and consequently UB_{XRJN} are underestimated. The solution to this problem is to calculate the upper bound UB_{comb} without updating the most promising combination until $UB_{XRJN} \le LB$. When the inequality holds, Algorithm 3 is executed the upper bound is accurately calculated and if it still holds that $UB_{XRJN} \le LB$, XRJN* halts.

5.5. Cost and complexity analysis

Both algorithms described so far follow the pull and bound paradigm presented in Algorithm 1. The I/O cost is mainly determined by the depth each relation is accessed, therefore we expect XRJN to access fewer objects than MHRJN because XRJN provides a better estimation of the upper bound than MHRJN.

The processing cost of each repetition of the pull and bound framework loop is determined by the cost of the lower and upper bound calculations and the cost of updating the candidate combinations. Each combination may contain only one object, therefore the cost for updating the combinations is $O(1)$ for MHRJN. The cost for updating a combination for XRJN is $O(|\mathcal{E}|)$ because the update of a combination c is followed by a calculation of the maximum possible score of c performed by the lazy evaluation. The top-k results are stored in a priority queue. The cost of updating the queue is equal to $O(\log k)$ while the cost of checking the first element of the queue is equal to $O(1)$. Each time a combination is updated it is necessary to verify that the combination is not inserted twice in the queue. The check is performed in $O(1)$ time using a hash table on the elements of the queue. The cost of the upper bound calculation for MHRJN is $O(|\mathcal{E}|)$ as we have to calculate $|\mathcal{E}| + 1$ upper bounds for each tuple update. The cost for XRJN is in the best case equal to $O(|\mathcal{E}|)$ which is the cost of updating the upper bound for the most promising combination while in the worst case the cost is equal to $O(|\widehat{B}_{\text{XRJN}}(E_M)|)$ which is the cost of identifying the most promising combination when the lazy upper bound evaluation produces an upper bound smaller than the lower bound and there is at the same time a large number of non total combinations. The latter however will rarely be the case because frequent updates will create fast total combinations which can be ignored during the upper bound evaluation while in case of infrequent updates the most promising combination is unlikely to change.

The loop in the pull and bound framework is repeated at most $|\mathcal{E} + 1 \parallel \widehat{B}(E_M)|$ times for each algorithm. Therefore, the cost for MHRJN is equal to $O(|\widehat{B}_{\text{MHRJN}}(E_M)|(|\mathcal{E}| + \log k))$ while for XRJN it is in the best case equal $O(\widehat{B}_{\text{XRJN}}(E_M)|(|\mathcal{E}| + \log k))$. In the worst case the cost for XRJN is equal to $O(|\widehat{B}_{\text{XRJN}}(E_M)|^2)$ as the dominating cost is that of the upper bound calculation. The complexity analysis indicates that the processing cost of both algorithms is highly affected by the size of the combinations $\widehat{B}(E_M)$ created by each algorithm. XRJN is expected to generate a significantly smaller number of combinations and therefore we expect XRJN to be more efficient than MHRJN despite the fact that each processing step of XRJN has a higher processing cost than that of MHRJN.

In the following, we will introduce an improved pulling strategy which will reduce the cost of upper bound calculation and ultimately improve the performance of XRJN.

Algorithm 4. XRJN* pulling strategy.

Input: E_M, $\mathcal{E}$
Output: $E \in \mathcal{E} \bigcup E_M$: next relation to pull from
1: **if** $UB_{E_M} > UB_{comb}$ **or** $c_{mp} = null$ **then**
2: **return** E_M //c_{mp}=null: no non-total combinations
3: **end if**
4: $R \leftarrow E_M$
5: max ← − ∞
6: **for all** missing relations E of c_{mp} **do**
7: u←# of non-total combinations not combined with E

```
8:    if u > max then
9:        max ← u
10:       R ← E      //relation with the highest #of uncombined objects
11:   else if max=u then
12:       if f_w(HE[last]) > HR[last] then
13:           R ← E      //in case of tie choose the relation with
      the best last seen score
14:       end if
15:   end if
16: end for
```

5.6. The XRJN* pulling strategy

The objective of the pulling technique is twofold. The first goal is the early convergence of the lower and upper bound in order to minimize the access depth (number of accessed objects) for each relation. The lower bound is increased by the formation of total combinations and is stabilized as soon as the top-k combinations have been formed. The algorithm however, will not halt as soon as the top-k combinations are discovered, but when it is certain that no better combinations can appear. This will be ensured by the decrease of the upper bound. The upper bound is affected by the formation of total combinations since the main objects that participate in total combinations can be excluded from the calculation. The upper bound is also affected by the scores of the last accessed objects. Therefore we should aim at accessing first the relations that have high score according to the user's preferences.

The second goal of the pulling technique is to reduce the processing cost of calculating the upper bound and updating the already formed combinations. As mentioned before, the processing cost of both procedures is determined by the number of non-total combinations existing in each step of the algorithm. Therefore the goal of the pulling technique should be to pull objects from the relations in such order that the number of non total combinations is minimized.

Based on the above observations we propose the pulling strategy described in Algorithm 4. We refer to this variation of XRJN as XRJN*. If $UB_{E_M} > UB_{comb}$ the algorithm reads from the main relation while in the opposite case it reads from an additional relation. In this way, the upper and lower bound converge faster as on each step the highest upper bound is reduced. If the algorithm chooses to read from an additional relation, it examines only the additional relations which can improve the score of the current most promising combination c_{mp}, i.e., it examines only the missing relations of c_{mp}. The algorithm selects the additional relation with the highest number of uncombined non-total combinations while ties are solved by choosing the relation with the highest last seen score $f_{\mathbf{w}}(HE[last])$. By reading from a relation not combined with c_{mp}, it is ensured that the maximum score of c_{mp} and therefore UB_{comb} will be updated and therefore the upper and lower bound will converge more. At the same time, the algorithm tries to maximize the number of non-total combinations which will be updated. Updating a large number of non-total combinations leads to the formation of total combinations and the increase of the lower bound, which forces the upper and lower bounds to converge.

6. Generalizing the XTJ_k query

In this section we discuss how our approach can be extended in order to support more general variants of the $ETOP_k$ query. In particular, we study two orthogonal generalizations: (a) providing more combinations per main object, and (b) supporting different (more general) aggregation functions. In the first case we study an extended $ETOP_k$ query, where we offer the ability to the user to explore more combinations for each of the top-k main objects organized into groups. In the second case, we study how our framework can be generalized to support a wide variety of aggregation functions, other than plain sum, for calculating the score of a combination.

6.1. Providing more results

We extend the proposed XTJ_k query, in order to include multiple combinations with the same main object in the retrieved result set. This generalization is motivated by practical applications, where a user would like to explore the set of best alternatives that contain a specific main object, instead of being presented only with the highest scoring alternative. Notice that presenting such alternatives of inferior score is useful also for comparative purposes, so that the user can assess her benefit, e.g., when choosing the best alternative compared to the second best, etc. Essentially, the requirement is to relax the constraint of presenting only one combination per main object, and instead present *groups of combinations* for each main object.

To support this requirement, we propose to enrich each of the top-k main objects o, with a group of m alternative combinations of o, where m is a user-defined parameter. With respect to the query semantics, the generalized XTJ_k query still returns exactly the same top-k combinations; the difference is that for each main object of the top-k combinations, the result additionally contains the top-m combinations of that particular main object. Consequently, the size of the result set of the generalized XTJ_k query is $k \times m$.

More formally, we define the concept of *alternative combinations* to facilitate the presentation of the generalized query.

Definition 4 (*Alternative combinations*). Given a main object o of the main relation E_M, the alternative combinations $ALT(o)$ is a set of the m combinations of o with the highest scores:

- $|ALT(o)| = m$,
- $\forall\ c_1, c_2 \in ALT(o)$ it holds that $m(c_1) = m(c_2) = o$,
- $\forall\ c \notin ALT(o)$ with $m(c) = o$ it holds that: $\exists\ c_i \in ALT(o)$ with $m(c_i) = o$ such that: $f_{\mathbf{w}}(c_i) \geq f_{\mathbf{w}}(c)$

We also refer to the score of $ALT(o)$ as the score of the best combination of o. Then, the formal definition of the generalized query, denoted $XTJ_{k,m}$, is derived as the k sets $ALT(o)$ with the highest score.

Algorithm 5. Update combination.

```
Input: Object o, Tuple t, E: relation of t
ALT' ← ∅
if t is the first combinable tuple of E with o then
    for all c ∈ ALT(o) do
        c' ← c⋃{t}
        ALT' ← ALT'⋃{c'}
    end for
else
    c ← replace(TOP_1(ALT(o)), t, E)      //replace the respective
    tuple of c with t
    if f_w(c) > f_w(TOP_m(ALT(o))) then      //if the new combi-
    nation is the top-m alternatives
        for all (c' ∈ ALT(o)) and (c'⋂E = TOP_1(ALT(o))⋂E) do
            newComb ← replace(c', t, E)
            ALT' ← ALT'⋃{newComb}
        end for
    else
        mark c as finished for relation E
    end if
end if
ALT(o) ← TOP_m(ALT(o)⋃ALT')
```

In order to be able to discover a group of m combinations for each

main object of the result set, both the algorithm updating the combinations and the stopping criterion of the algorithm need to be modified. When a new main object is accessed, a new combination c is created as well as a set of alternative combinations $ALT(o)$ with c being its only element. When a new additional tuple is accessed, the sets of alternative combinations $ALT(o)$ of all main objects that are combinable with t need to be updated.

Algorithm 5 describes the update procedure of the respective set of $ALT(o)$ when an additional tuple is added. We denote as $TOP_i(ALT(o))$ the i-th combination sorted by their score. Given a main object o and a new accessed tuple t of a relation E, if $TOP_1(ALT(o))$ does not contain a tuple from E, then t is combined with $TOP_1(ALT(o))$. If the result produces a combination with higher score ($f_{\mathbf{w}}(t) > 0$), then all combinations in $ALT(o)$ need to be updated (line 4) since t is the first combinable tuple of E. Thus, the new tuple t is also used to generate a set of new alternative combinations $ALT(o)$ by adding t to all existing alternative combinations. The new set of combinations $ALT(o)$ consists of the already existing and the newly generated combinations and only the top-m elements are maintained.

In the opposite case, when c already contains a tuple from E, a different approach is followed. First, we create a new combination by replacing the tuple of $TOP_1(ALT(o))$ that belongs to relation E with the new tuple t. The score of this combination is an upper bound of the score of any combination that contains t. If this score is smaller than the score of the m-th already retrieved combination, then no combination that contains t can be added to $ALT(o)$. In this case, $ALT(o)$ is not modified and also we do not need to access any more tuples of E as the remaining tuples will create combinations with worse score. If a tuple $t \in E$ does not create an alternative combination which belongs to the $TOP_m(ALT(o))$ set, then no other tuple of E needs to be combined with c and therefore we consider c to be finished for E. Otherwise, given a new tuple $t \in E$ the algorithm creates new alternative combinations by replacing the respective tuple of E in all combinations c' of $ALT(o)$ that share the same tuple of E with $TOP_1(ALT(o))$ (for which it holds that $c \bigcap E = c' \bigcap E$, i.e., the alternative combinations which contain the same tuple of E as c does). When all updates have been made, only the top-m combinations need to be kept.

Naturally, the stopping criterion in Algorithm 1 has to be altered. In more detail, once the lower bound becomes higher than the upper bound, the algorithm stops updating the bounds and accesses the additional relations until the top-m alternative combinations for each main object have been found. The pulling strategy is modified in order to facilitate the creation of the alternative combinations after the k objects with the highest score have been found. At each pulling step, the algorithm chooses a random main object o which is not finished for at least one additional relation and pulls a tuple from a random additional relation for which o is not finished for. The main relation does not need to be accessed any longer as the k best products have already been found.

Theorem 4. *Algorithm 5 provides a correct solution to the $XTJ_{k,m}$ query.*

Proof. The group of alternative combinations is initialized with a combination containing only one main object o. At this stage, the top-m list is correct as there is only one combination in the set. We will denote the top-m set of alternative combinations generated by Algorithm 5 after the n^{th} update as $ALT^n(o)$. We will show that if $ALT^n(o)$ is a correct set of top-m combinations at update n, then $ALT^{n+1}(o)$ is also a correct top-m set. To prove that Algorithm 5 produces the correct result at the n+1 update, it is sufficient to prove that given the set $ALT^n(o)$, Algorithm 5 generates all combinations that can be in $ALT^{n+1}(o)$.

Let t be a new tuple of an additional relation E to be combined with object o. The first case is that t is the first combinable tuple. Then $|ALT^n(o)|$ new alternative combinations containing the new tuple are added to $ALT^{n+1}(o)$ and top-m of both new and existing combinations are selected. In this case all possible combinations were generated and the $ALT^{n+1}(o)$ correctly represents the set of the top-m alternative combinations for object o.

The second case is that t is not the first accessed combinable tuple of E with o. We denote with $C^n(o)$ all combinations with main object o that have been examined up to update n (not only the $ALT(o)$ combinations). We can partition $C^n(o)$ into two partitions based on the best so far combination of $ALT(o)$, $TOP_1(ALT^n(o))$. Let t_0 be equal to $t_0 = TOP_1(ALT^n(o)) \bigcap E$. Note that t_0 may be the empty set. The first partition of $C^n(o)$, Π_1, is the partition such that $\forall\ c \in \Pi_1$ it holds that $c \bigcap E = t_0$. The second partition Π_2 is equal to $C(o) - \Pi_1$, i.e., $\forall\ c \in \Pi_2$ it holds that $c \bigcap E \neq t_0$. Obviously for any combination c in Π_2 there is a better combination $c' \in \Pi_1$ such that $c' = c - (c \bigcap E) \bigcup \{t_0\}$. Therefore, to generate all new combinations based on tuple t we only need to examine the combinations in Π_1. At this point we have shown that upon the n+1 update of the $ALT(o)$ set, we need to examine only the combinations in Π_1. In the following we will show that it is necessary to update only the combinations in $ALT^n(o) \bigcap \Pi_1$.

Given two combinations in Π_1, $c_1 \in ALT^n(o) \bigcap \Pi_1$ and $c_2 \in \Pi_1 - ALT^n(o)$ it holds that $f_{\mathbf{w}}(c_1) \geq f_{\mathbf{w}}(c_2)$. Let c'_i be the combinations that result after the replacement of t_0 with t. Then it holds that $f_{\mathbf{w}}(c'_i) = f_{\mathbf{w}}(c_i) - f_{\mathbf{w}}(t_0) + f_{\mathbf{w}}(t)$. From the last equation we derive that $f_{\mathbf{w}}(c'_1) \geq f_{\mathbf{w}}(c'_2)$ and therefore we conclude that any combination that can be in $ALT^{n+1}(o)$ will either be in $ALT^n(o)$ or it will be generated by the update of a combination that belongs to the set $ALT^n(o) \bigcap \Pi_1$. Therefore Algorithm 5 produces the correct result.□

6.2. Generalizing the aggregation function

The scoring function that was introduced in the previous sections is a special case of the more general scoring scheme $\mathcal{F}_{\mathbf{w}}(c) = \sum_{j=1}^{d} \mathbf{w}[j] \text{aggr}_{j\ p \in c}(p[j])$, where $\text{aggr}_{j\ p \in c}$ is an aggregation function for all items in combination c over dimension j. The aggregation function aggr_j indicates the contribution of an accessory item to the score of the combination it participates for dimension j. In the general case, the contribution of each item in the score of the combination depends on the each attribute and on the values of the rest of items for that attribute. As an example, the addition of an extra memory module to a laptop might result to the substitution of an existing module, thus the total memory of the laptop will be less than the sum of the capacity of the new and the existing modules. Similarly the price can be the total price of the items or possibly a discount may apply.

In such situations the final score of a combination is unknown until the combination is generated. However, the contribution of each accessory item in the final score of a combination is in many cases bounded by the attributes of the item. Therefore, the final score of the combination is bounded by a *bounding function* that calculates the maximum possible score of a combination. In the aforementioned example, the total memory capacity cannot exceed the sum of the capacity of the modules, and the price of each additional item, while the discount may not exceed a certain percentage.

Definition 5. *Bounding function*: Given two d-dimensional functions $f(p)$: $\mathbb{R}^d \rightarrow \mathbb{R}$, $\widehat{f}\ (p)$: $\mathbb{R}^d \rightarrow \mathbb{R}$ we say that $\widehat{f}$ is a bounding function for f if $\forall\ p \in \mathbb{R}^d$ it holds that $f(p) \leq \widehat{f}\ (p)$.

MHRJN and XRJN produce the correct results if the scoring function $f_{\mathbf{w}}$ is bounded by a function of the form $\widehat{f}_{\mathbf{w}} = \sum_{p_i \in c} \hat{g}_w^{E_i}(p_i)$ where $p_i \in E_i$ and $\hat{g}_w^{E_i}$ is a monotonic function indicating the maximum possible contribution in the score of a combination c for an item in relation in E_i. The monotonicity of $\hat{g}_w^{E_i}$ implies that for two objects p's for which it holds that $p_1[j] \leq p_2[j]$, $1 \leq j \leq d$, the maximum contribution of p_1 to the score of a combination c cannot be larger than the maximum contribution of p_2. In other words, if the maximum contribution of a main or accessory item to the score of a combination is bounded by a function that depends only on the attributes of the item, then XRJN produces the correct result. Typical aggregation functions that belong in this category are among others, sum, max, min, and avg.

Modified bounds: The generalized MHRJN and XRJN algorithms access the main and accessory items in descending order of score, but the score of each item is now calculated based using the scoring functions $\hat{g}_w^{E_i}$. Both generalized algorithms employ the bounding function $\hat{f}_{\mathbf{w}}$ to calculate the respective upper bounds. Eqs. (9)–(11) define formally the generalized bounds for XRJN. The bounds for MHRJN are modified accordingly:

$$UB_{E_M} = f_{\mathbf{w}}(HE_M[last]) + \sum_{E \in \mathcal{E}} u\left(\hat{g}_{\mathbf{w}}^{E}(HE[1])\right) \tag{9}$$

$$c_{mp} = \underset{\substack{c \in \widehat{B}(E_M) \\ c \text{ not total}}}{\operatorname{argmax}}\left(f_{\mathbf{w}}(c) + \sum_{\substack{E \in \mathcal{E} \\ E \text{ combinable relation}}} u(\hat{g}_{\mathbf{w}}^{E}(HE[last]) - g_{\mathbf{w}}^{E}(c))\right) \tag{10}$$

$$UB_{comb} = f_{\mathbf{w}}(c_{mp}) + \sum_{\substack{E \in \mathcal{E} \\ E \text{ combinable relation}}} u(\hat{g}_{\mathbf{w}}(HE[last]) - g_{\mathbf{w}}^{E}(c)) \tag{11}$$

Given a non-total combination c, a *combinable* relation E_i is a relation such that (a) $c \bigcap E_i = \varnothing$ or (b) $c \bigcap E_i = p_i$ and $\hat{g}_{\mathbf{w}}^{E_i}(HE[last]) > g_{\mathbf{w}}^{E_i}(c)$ where $g_w^{E_i}(c)$ is the actual contribution of p_i in combination c. As a result, given a combination c and an additional relation E_i such that $c \bigcap E_i = p_1$ when a new combinable item $p_2 \in E_i$ is accessed it is possible for p_2 to replace p_1.

Theorem 5. *MHRJN and XRJN provides a correct solution to the XTJ_k query for any scoring function bounded by a scoring function of the form $\hat{f}_{\mathbf{w}} = \sum_{p_i \in c} \hat{g}_w^{E_i}(p_i)$ where $g_w^{E_i}$ is monotonic for all i.*

Proof. Let us assume that the algorithm has halted, after having accessed d_0 tuples for the relation E_M and d_i tuples for each relation $E_i \in \mathcal{E}$, therefore $UB \leq LB$. Let now c' be an unseen combination for which it holds that $f_{\mathbf{w}}(c') > LB$.

If the unseen combination c' contains an unseen main object, then the score of the combination will be at most equal to UB_{E_M}, thus $f_{\mathbf{w}}(c') \leq UB_{E_M}$. Since $UB_{E_M} \leq UB \leq LB$ it holds that $f_{\mathbf{w}}(c') \leq LB$ which contradicts with the assumption $f_{\mathbf{w}}(c') > LB$. The contradiction that we have reached is due to our assumption that c' contains an unseen main object and it is better than LB.

Alternatively, let us assume that unseen combination c' contains an already accessed main object, which means that there exists a non-total combination $c \in \widehat{B}(E_M)$ that will be combined with at least one unseen additional object and produce c'. In the following we assume that c has been combined with an unseen object from only one relation E_j and c' was created. Similarly, we can prove the general case of more than one relation. Let $E_j[d_j + x]$ be the first combinable object with c. Then the maximum possible score of c' when joined with the unseen tuple will be equal to: $\hat{f}_{\mathbf{w}}(c') = f_{\mathbf{w}}(c) + \hat{g}_{\mathbf{w}}(E_j[d_j + x]) - g_{\mathbf{w}}^{E_j}(c) \leq f_{\mathbf{w}}(c) + \hat{g}_{\mathbf{w}}(E_j[d_j]) - g_{\mathbf{w}}^{E_j}(c)$. From Eq. (8) we conclude that $\hat{f}_{\mathbf{w}}(c') \leq UB_{comb} \leq UB \leq LB$. However in the beginning we assumed that $\hat{f}_{\mathbf{w}}(c') > LB$ and this is a contradiction. We conclude that an unseen combination cannot have a greater score than the score of the k-th combination in $\widehat{B}(E_M)$, thus our algorithm always returns the correct result set.□

Although both MHRJN and XRJN provide the correct solution, MHRJN is not instance optimal as shown earlier. On the contrary, XRJN is instance optimal even in the general case.

Theorem 6. XRJN *is instance optimal for the generalized XTJ_k query within algorithms following the Pull-Bound Framework with optimality ratio n+1 where n is the number of additional relations.*

Proof. Let A be a random deterministic algorithm solving correctly the XTJ_k query for a vector $\mathbf{w}$. The algorithm halts after having accessed d_0 tuples for E_M and d_i tuples for each additional relation E_i. We define c_k to be the k best candidate combination discovered by A with score $f_{\mathbf{w}}(c_k)$ and d_{max} to be equal to $d_{max} = \max_{0 \leq i \leq n}(d_i)$. We will show by contradiction that XRJN will halt after accessing at most d_{max} tuples from each relation.

Let us assume that XRJN has accessed d_{max} tuples from each relation and has not halted. At this point XRJN has processed at least all combinations evaluated by A and therefore it holds that $LB_{\text{XRJN}} = f_{\mathbf{w}}(c_k)$. Under the assumption that XRJN has not halted, there are two cases to be examined.

The first case is that the upper bound of XRJN at that step is defined by an unseen object of the E_M i.e., $UB_{\text{XRJN}} = UB_{E_M}$ and it holds that $LB_{\text{XRJN}} < UB_{\text{XRJN}}$ and since $LB_{\text{XRJN}} = f_{\mathbf{w}}(c_k)$ it also holds that $f_{\mathbf{w}}(c_k) < UB_{\text{XRJN}}$. As algorithm A is deterministic, it must halt at the same step for all instances of relations that have the same seen tuples HE_M, HE_i. We can construct relation E_M such that $HE_M[d_0 + 1]$ is combinable with the first tuples of all additional relations $HE_i[1]$ and $f_{\mathbf{w}}(E_M[d_0 + 1]) = f_{\mathbf{w}}(E_M[d_0])$. For the combination c' defined by $\{HE_M[d_0 + 1], HE_1[1], \ldots, HE_n[1]\}$ it holds that $\hat{f}_{\mathbf{w}}(c') = UB_{\text{XRJN}}$, thus $f_{\mathbf{w}}(c_k) < \hat{f}_{\mathbf{w}}(c')$. Since the score of a combination can be evaluated only after the objects have been accessed, there can be a combination c' such that $f_{\mathbf{w}}(c') > f_{\mathbf{w}}(c_k)$. Therefore A has halted incorrectly and we are lead to a contradiction.

In the second case it holds that $UB_{\text{XRJN}} = UB_{comb}$ and $LB_{\text{XRJN}} < UB_{\text{XRJN}}$. Let c_{mp} be the most promising combination found by XRJN and $o = m(c_{mp})$ be the main object of c_{mp}. An instance of our database D can be constructed in way that $\forall E_i$ such that E_i is combinable for c_{mp}, it holds that $\hat{g}_{\mathbf{w}}(E_i[d_{max}]) = \hat{g}_{\mathbf{w}}(E_i[d_{max} + 1])$ and all $E_i[d_{max} + 1]$ tuples are combinable with o. In this case, when $d_{max} + 1$ tuples have been read from all relations a new combination $c_{mp}{}'$ is produced by updating c_{mp} and adding the newly pulled tuples. It holds that $\hat{f}_{\mathbf{w}}(c_{mp}{}') = UB_{\text{XRJN}}$ and therefore also $f_{\mathbf{w}}(c_k) < \hat{f}_{\mathbf{w}}(c_{mp})$. Thus, $c_{mp}{}'$ may belong to the result and A has halted incorrectly, which leads us to a contradiction.

We conclude that XRJN will halt after accessing at most d_{max} tuples from each relation. Thus, the cost of XRJN in terms of accessed tuples is at most $\text{cost}(\text{XRJN}, D) = (n + 1)*d_{max}$, while the cost of algorithm A is $\text{cost}(A, D) = \sum_{0 \leq i \leq n} d_i$. We can derive that $\text{cost}(\text{XRJN}, D) \leq (n + 1)*\text{cost}(A, D)$ since $d_{max} \leq \sum_{0 \leq i \leq n} d_i$. Hence XRJN is instance optimal with optimality ratio n+1.□

7. Experimental evaluation

In this section, we present the results of the experimental evaluation. All algorithms were implemented in Java and the experiments run on an AMD Opteron 4130 Processor (2.60 GHz), with 32 GB of RAM and 2 TB of disk.

Datasets and metrics: For the data set D, we used both synthetic and real data collections. For the synthetic data we used one uniform and one Zipfian distribution. In particular, for the uniform distribution all object values for all relations and dimensions were generated independently using a uniform distribution generator. Each additional relation has a random subset of attributes of the main relation and contains at least one positive and one negative attribute but in total it contains no more than d–1 attributes where d is the number of attributes of the main relation. The combinations of attributes of each additional relation is unique. Each additional relation has also a joining attribute which does not participate in the ranking of each object while the main relation has $|\mathcal{E}|$ joining attributes, one for each additional relation. The values for the joining attributes are decided based on the join selectivity value σ. For two relations L, R the join selectivity is equal to $\sigma = |L \bowtie R \parallel L \times R|^{-1}$ [39]. If a joining attribute has σ^{-1} different values, then the join contains $\sigma^{-1}\sigma^2|L \parallel R| = \sigma|L \parallel R|$ tuples which gives us a join selectivity of σ. All positive attributes of the main and the additional relations were normalized in the interval [0, 10 000] while the negative attributes of the main relation were normalized in the interval [− 10 000, 0]. The negative attributes of the additional relations were scaled to the size of the relation in order to make the cost of an object proportional to the potential improvement of the main object. In particular, for an additional relation E_i the negative attributes take values in the interval $[-10\,000|A_{E_i} \parallel A_{E_M}|^{-1}, 0]$ where $|A_{E_i}|$ is the

number of attributes included in the relation. Although this might seem counter-intuitive it is quite common for the attributes of accessory objects to have a value range of positive attributes similar to the respective attribute of the main object while their cost is lower. For instance the capacity of hard disks as separate components has similar range as the capacity of disks in laptops. The price of a hard disk however, is lower than a laptop's price carrying a similar disk. Finally, the size of each additional relation is equal to the size of the main relation.

For the Zipfian distribution we used the generator provided by the Apache Commons project[5]. The datasets were generated by giving as parameters the maximum value of an attribute (1000) and the value of the exponent characterizing the distribution. The positive and negative attributes of all relations were generated similarly to the uniform distribution.

In addition, we used the real datasets HOUSE (Household) and NBA. HOUSE consists of 127 930 6-dimensional tuples, representing the percentage of an American family's annual income spent on 6 types of expenditure: gas, electricity, water, heating, insurance, and property tax. NBA consists of 17 265 5-dimensional tuples, representing a player's performance per year. The attributes are average values of: number of points scored, rebounds, assists, steals and blocks. In both relations all attributes were normalized in the interval [0, 10 000]. In each experiment random attributes were considered as negative attributes taking values in the interval [− 10 000, 0]. Each additional relation was created by selecting a random subset of the scoring attributes of the main relation and copying the respective data. Similarly to the uniform distribution the negative attributes were scaled according to the size of each additional relation and each additional relation has at least one positive and one negative attribute and at most $d-1$ attributes in total, while each relation has a unique subset of attributes.

The metrics under which we evaluated the implemented algorithms were: (a) execution time required by each algorithm, and (b) total tuples accessed (depth). We should stress that we do not focus our performance analysis on the size of the data as the performance of the algorithms depends mainly on the number of additional relations, the join selectivity and the distribution of the values of the relations' attributes. We employed a best-case scenario regarding I/O accesses where relations are sorted for each query, stored on the disk, and accessed sequentially. This strategy minimizes the cost of the I/O accesses and allows us to study the minimum performance difference of the proposed algorithms. In practice, access to each relation will be achieved through other means such as materialized views [8] or multi-dimensional indexes [40], which will induce higher cost in terms of I/O and increase the performance gap between MHRJN and XRJN.

Experimental procedure: We run a series of experiments varying the parameters of a) the number of additional relations ($|\mathcal{E}|$) in the interval [3–7], (b) dimensionality (d) in the interval [4–8], (c) number of returned results (k) in the interval [5–100], (d) selectivity (σ) [0.001–0.05], and (e) the number of negative attributes [1–3]. For the Zipfian distribution we varied the value of the characteristic exponent s in the interval [0.1–1.0]. Each experiment was run under 5 different dataset instances and 100 queries were used for evaluating the performance of the algorithms.

The default setup for the experiments was: $|\mathcal{E}| = 5$, d=6, $|E_M| = 100$ K, k=10, σ=0.001 and each relation has one negative attribute. The number of preference queries for each setting was equal to $|W| = 100$. Both the dataset and the preferences set followed the uniform distribution.

[5] http://commons.apache.org/proper/commons-math/

7.1. Pulling technique evaluation

The first series of experiments focuses on evaluating the pulling technique described in Section 5 and here we compare XRJN against XRJN*. XRJN uses round robin as pulling strategy while XRJN* uses the pulling strategy described in Algorithm 4. Figs. 5 and 6 indicate that XRJN* provides an advantage both in processing time and number of accessed tuples. XRJN* pulling strategy forces the upper and lower bound to converge faster, by prioritizing the update of the most-promising combination and by aiming to reduce the highest upper bound. The aggressive update of the most-promising combination induces more frequent invocation of the exact upper bound calculation. As a result, the processing-time gain is smaller than the access-depth gain, fact that becomes more obvious as the number of additional relations increases (Fig. 5).

7.2. Sensitivity analysis

In this section, we provide a detailed sensitivity analysis by varying different parameters that influence the performance of our proposed algorithms. We start by comparing the most important parameters, namely the number of additional relations, the dimensionality of the relations, the number of returned results and the join selectivity of the relations. In addition we examine the performance of the algorithms on a Zipfian distribution.

Varying $|\mathcal{E}|$: As we increase the number of additional relations, the number of accessed tuples and the processing time of all approaches increase. Fig. 7 indicates that XRJN* access consistently less objects than MHRJN and XRJN and the performance of XRJN* is less affected by the increase of the number of additional relations. Due to the improved bounding technique XRJN* and XRJN access nearly an order of magnitude less tuples than MHRJN. The performance of MHRJN is dependent both on the convergence of the upper and lower bound and on the rate the formed combinations become total. If there are many non total combinations then for each accessed tuple there will be an increased processing cost for finding the combinations that the tuple can be added to. On the other hand, XRJN* aims at creating complete combinations fact that helps the lower bound to increase fast and also reduces the updating cost.

Varying dimensionality: Similar conclusions can be drawn when we vary the dimensionality. Fig. 8 indicates that XRJN and XRJN* are nearly an order of magnitude more efficient than MHRJN with respect to time and depth access.

Varying k: The same conclusions hold when we change the number of returned results. Fig. 9 shows the performance of all algorithms. It is noteworthy that the processing cost for MHRJN is increasing significantly as k is increased. The processing cost for XRJN* also increases but with a slower rate and in all cases it remains nearly an order of magnitude less than MHRJN. The same applies for the access depth, where MHRJN accesses more tuples as k increases. Except for the increased number of accessed tuples, the increase of k causes more combinations to be created and evaluated. MHRJN and XRJN create an arbitrary number of non-total combinations which have to be maintained and updated when a new tuple is accessed. XRJN* on the other hand minimizes the effect of the increased number of non-total combinations through its pulling strategy and therefore is less affected by the increase of the result-set size.

Varying selectivity σ: As expected, join selectivity plays an important role in the performance of both algorithms. As Fig. 10 indicates, when the value of join selectivity increases the performance gap between the MHRJN and XRJN* increases as well. The reason lies in the fact that XRJN exploits the fact that total combinations are formed faster as selectivity increases by considering only non total combinations in the upper bound calculation. In addition, frequent combination updates allow XRJN* to reduce the cost of upper bound calculation as the size of $\widehat{B}_{\text{XRJN}*}(E_M)$ is small for high selectivity values. In total, XRJN* benefits from higher selectivity values in two ways; the upper bound of XRJN* converges to the lower bound much faster than the upper bound of MHRJN and the cost of the upper

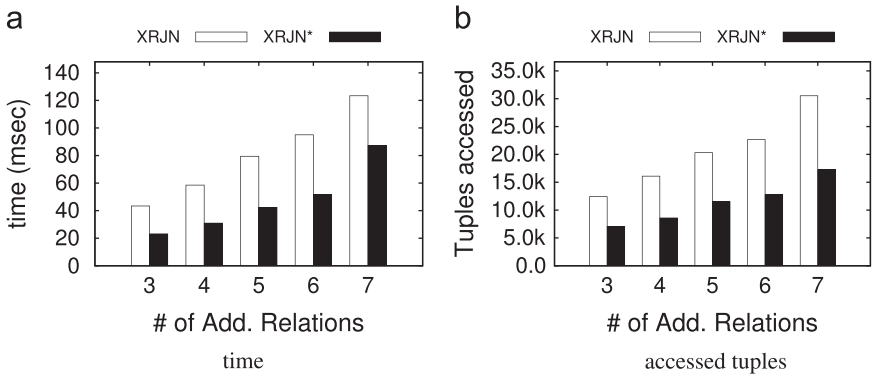

Fig. 5. Pulling strategy evaluation: varying $|\mathcal{E}|$.

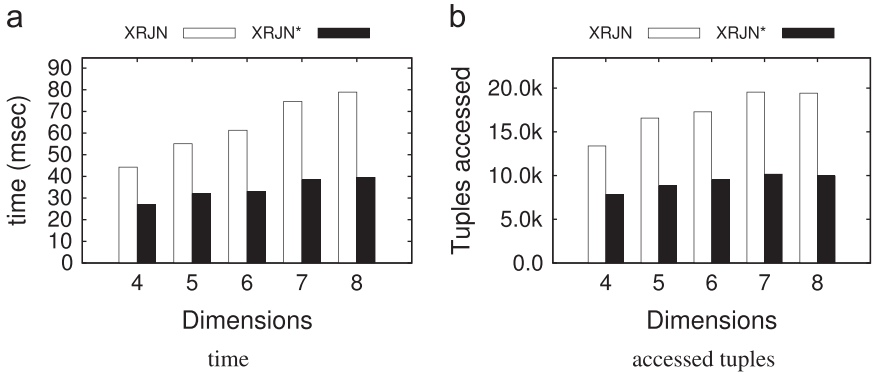

Fig. 6. Pulling strategy evaluation: varying d.

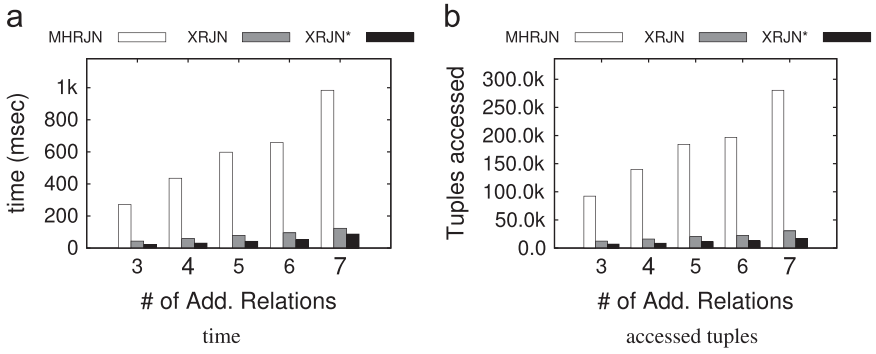

Fig. 7. Sensitivity analysis: varying $|\mathcal{E}|$.

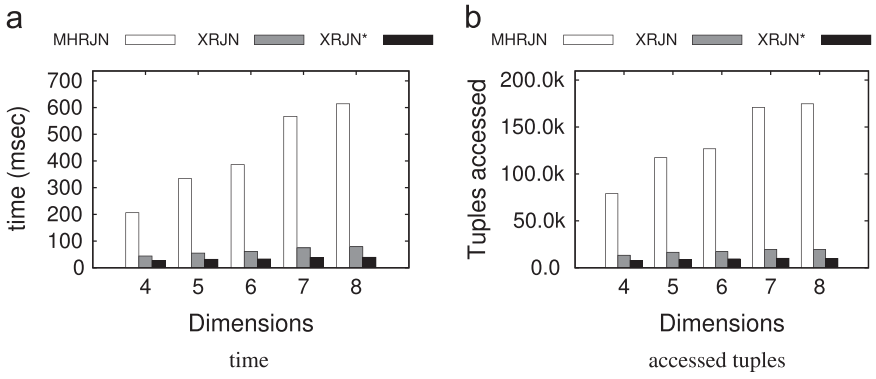

Fig. 8. Sensitivity analysis: varying d.

bound calculation drops as selectivity rises.

Varying the number of negative attributes: Fig. 11 illustrates the performance of all algorithms as we vary the number of negative attributes. When the number of negative attributes increase, the number of tuples which cannot improve a combination increase as well. Naturally, all algorithms benefit from that fact. MHRJN is affected more than XRJN and XRJN* because when an additional tuple with negative score is read, the non accessed part of the relation can be safely discarded. The upper

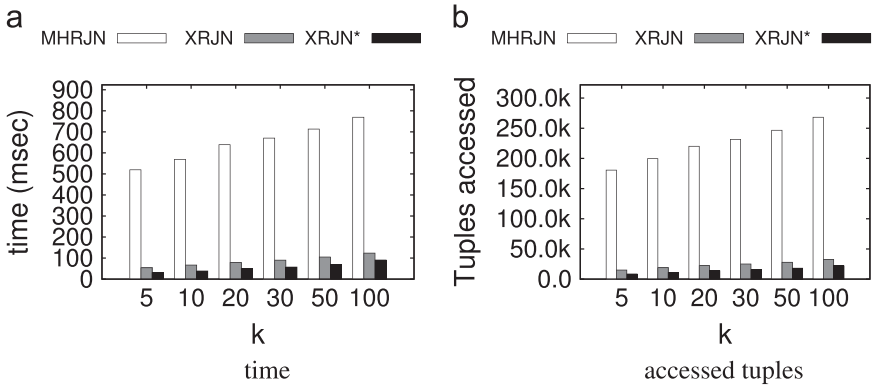

Fig. 9. Sensitivity analysis: varying k.

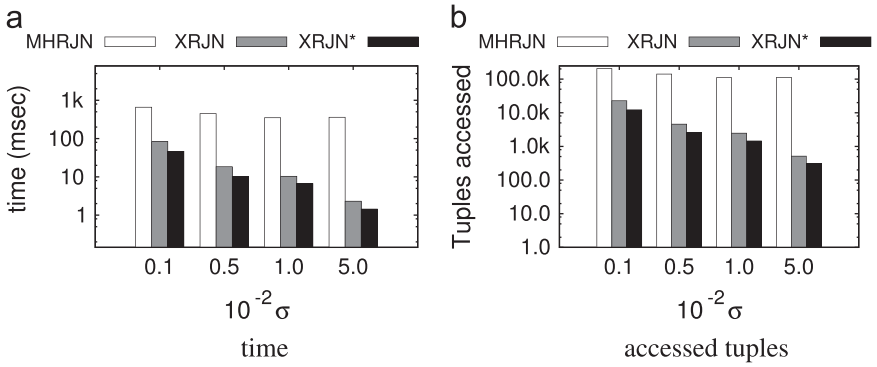

Fig. 10. Sensitivity analysis: varying σ (log scale).

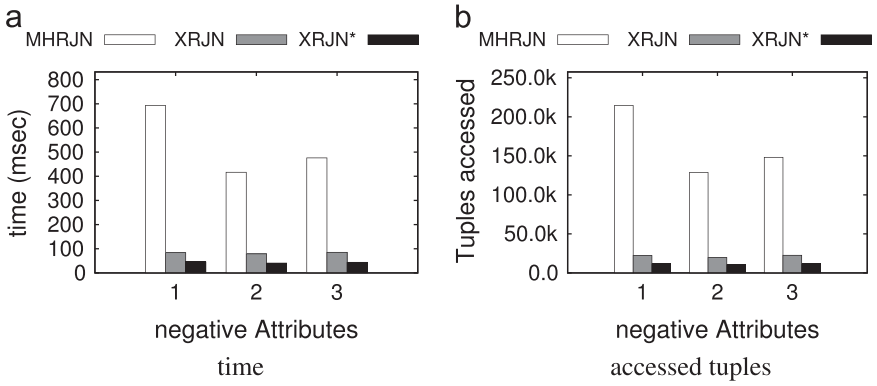

Fig. 11. Sensitivity analysis: varying # of negative attributes.

bound of XRJN allows both XRJN and XRJN* to terminate before the tuples with negative score are accessed and therefore their performance is not affected significantly by the number of negative attributes. They remain however in all cases significantly more efficient than MHRJN.

Zipfian distribution: We evaluated our algorithms against a Zipfian distribution as well. Fig. 12 illustrates the performance of the algorithm against different values of the exponent characterizing the distribution. As the value of the exponent increases, the performance of the all algorithms remains relatively unaffected. In all cases the XRJN and XRJN* remain almost an order of magnitude more efficient than MHRJN both in respect of processing time and accessed tuples.

Real datasets: The results using the real datasets are in accordance with the results of the synthetic ones. As the number of additional relations increases, the performance gain in terms of accessed tuples increases as well. Figs. 13(b) and 14(b) show that processing time and the number of tuples accessed by MHRJN increases much faster than in the case of XRJN*. XRJN* is up to 10 times more efficient for both HOUSE and NBA datasets. We should note at this point that the default value of the join selectivity parameter for the NBA dataset was set to σ=0.01 due to the small size of the dataset.

Figs. 15 and 16 depict the behavior of the algorithms when varying parameter k. Both XRJN and XRJN* are more efficient than MHRJN regarding the access depth. It is noteworthy that XRJN* is not only more efficient than MHRJN but also the performance of XRJN* is minimally affected by the increase of parameter k. On the contrary Figs. 15(b) and 16(b) indicate that the cost of MHRJN increases linearly with respect to k.

Figs. 17 and 18 illustrate the performance of the algorithms when varying the join selectivity σ. All algorithms behave as expected based on the evaluation of the uniform distribution. It is worth noting that the performance of XRJN* improves much faster than MHRJN both in terms of processing time and in terms of access depth. Similarly to the case of the uniform distribution sets the XRJN* forces the upper and lower bound to converge faster than MHRJN as it aims towards creating total combinations fast, fact that helps the lower and upper to converge.

Figs. 19 and 20 illustrate the performance of the algorithms when varying the number of negative attributes. The performance of the

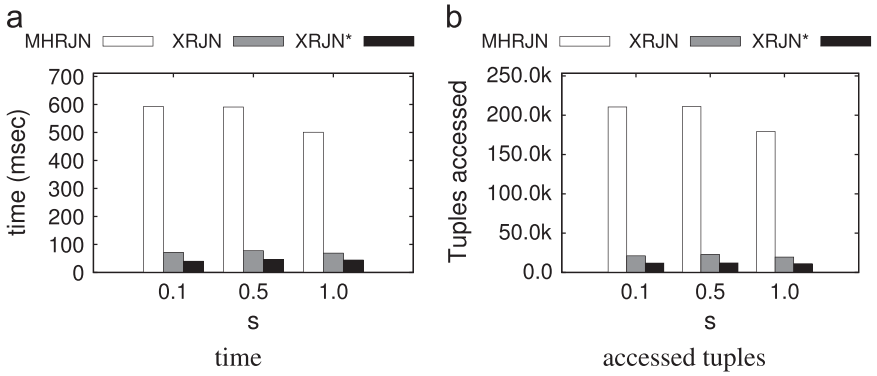

Fig. 12. Sensitivity analysis: Zipfian distribution.

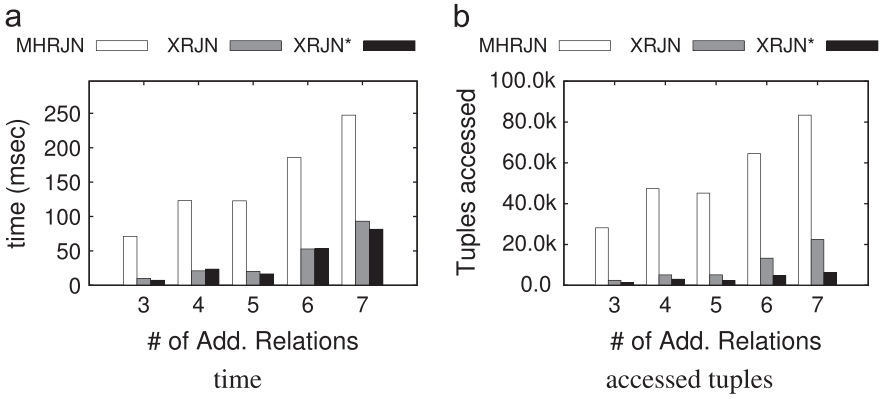

Fig. 13. NBA: varying $|\mathcal{E}|$

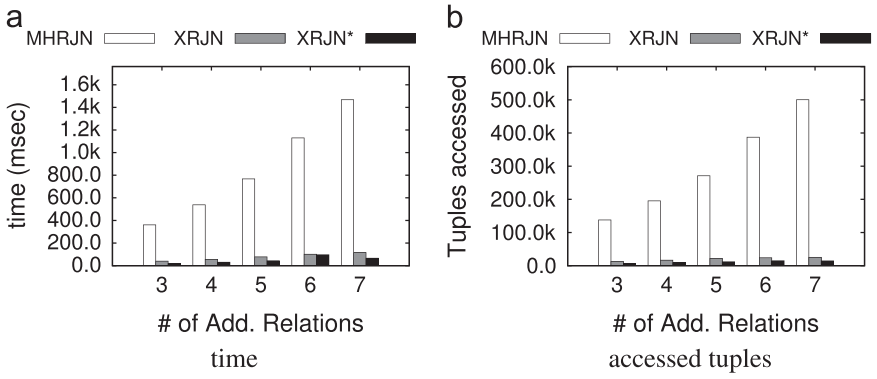

Fig. 14. HOUSE: varying $|\mathcal{E}|$.

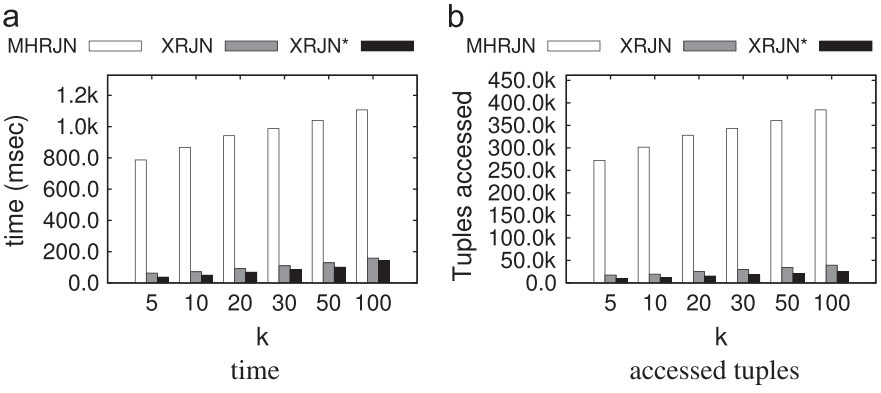

Fig. 15. NBA: varying k.

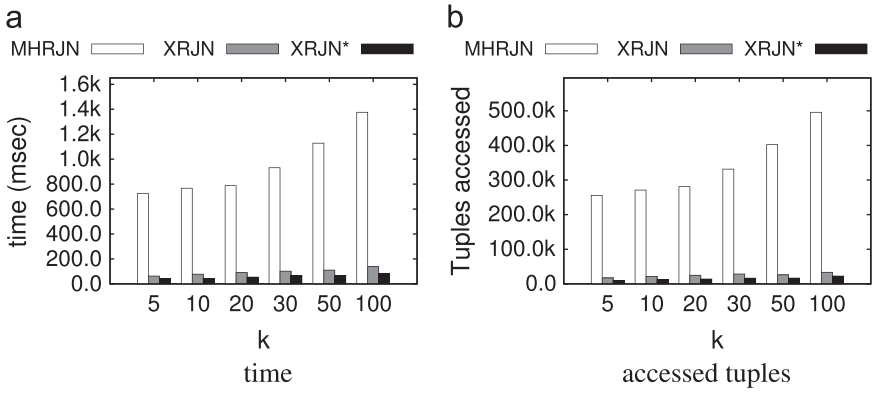

Fig. 16. HOUSE: varying k.

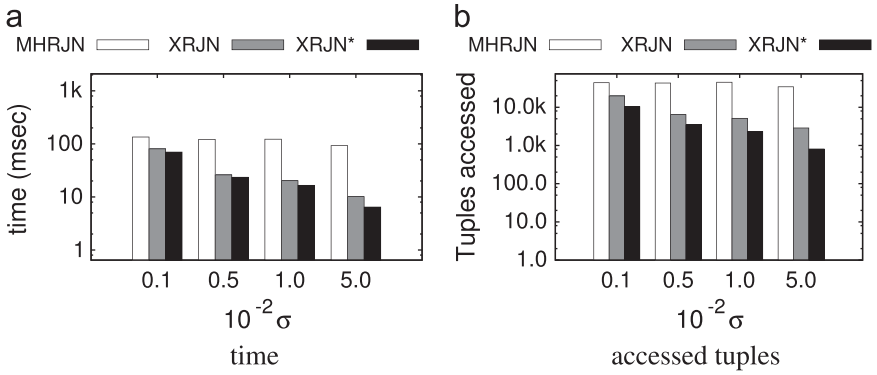

Fig. 17. NBA: varying σ (log scale).

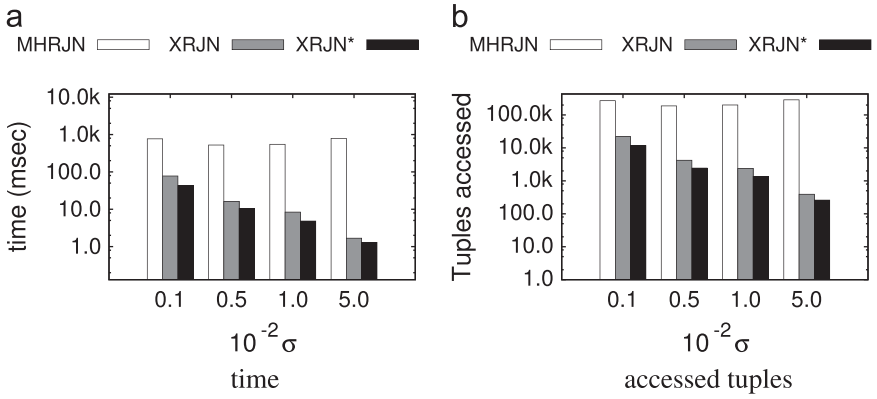

Fig. 18. HOUSE: varying σ (log scale).

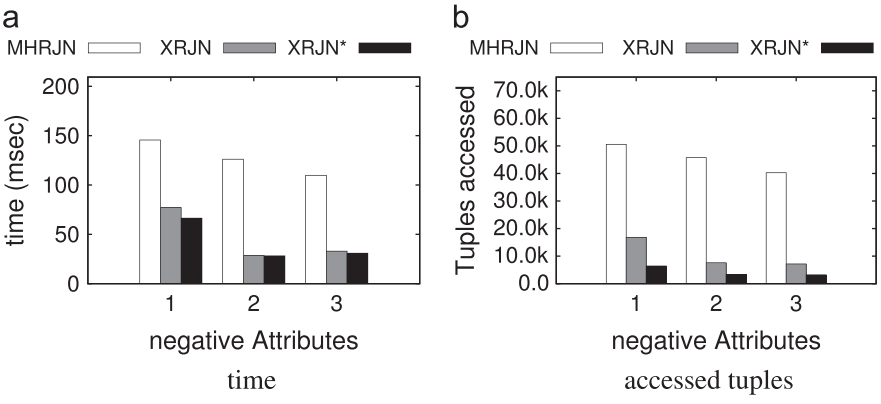

Fig. 19. NBA: varying # of negative attributes.

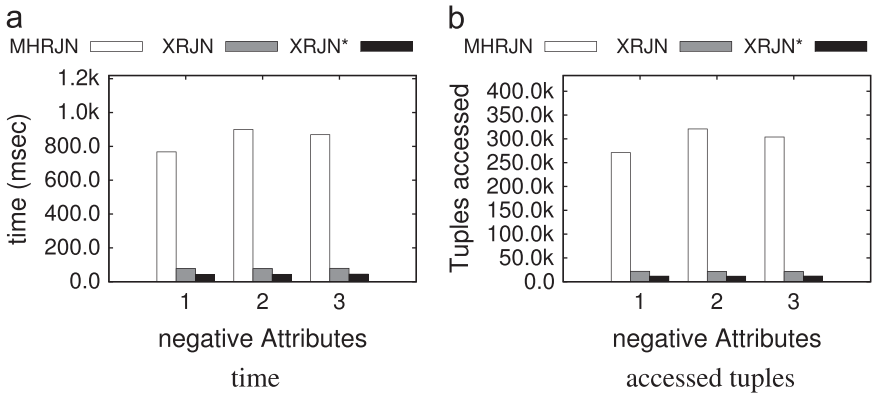

Fig. 20. HOUSE: varying # of negative attributes.

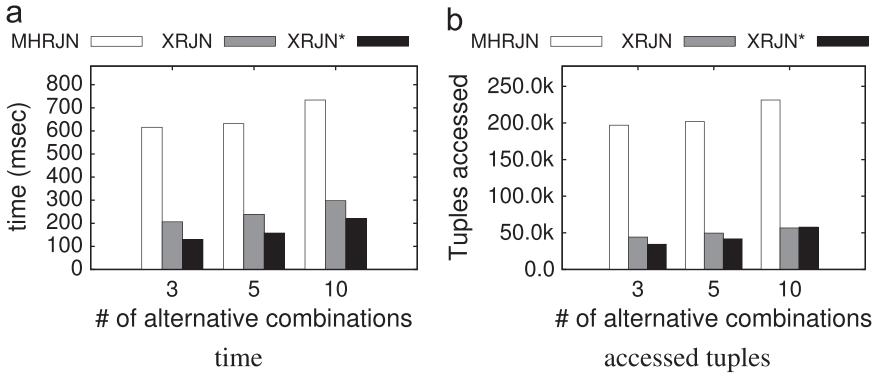

Fig. 21. Varying the number of alternative combinations.

algorithms varies due to the random selection of the negative attributes in each run of the experiments. Nevertheless, XRJN and XRJN* perform in all cases significantly better than MHRJN. Especially in the case of the HOUSE dataset both XRJN and XRJN* are nearly an order of magnitude more efficient than MHRJN.

Alternative combinations generation: We tested both algorithms using the default experimental setup. Fig. 21 illustrates the performance of the two algorithms when a set of m combinations is presented for each main product. As expected the processing cost rises as the number of combinations per main object increases. Interestingly, while the access depth increases for XRJN*, it remains stable for MHRJN. As MHRJN accesses 4 times more tuples than XRJN* to discover the best combination for each main object, it has already accessed enough tuples to form the alternative combinations before the lower bound exceeds the upper bound. Therefore, the only extra cost for MHRJN is that of calculating the top-m alternative combinations. In contrast, XRJN* utilizes a more efficient bounding scheme which allows the algorithm to identify the best combination for each main object after accessing a much smaller number of tuples than MHRJN. Consequently, XRJN* needs to continue reading from the relations after the lower bound has exceeded the upper bound in order to form the top-m combinations. As a result, the access-depth for XRJN* rises as the number of alternative combinations increases. In all cases however, it remains up to 4 times more efficient in terms of access-depth and up to 3 times more efficient in terms of processing time than MHRJN.

8. Conclusion

In this paper, we address the problem of discovering the top-k combinations between a single main product relation and several additional relations that can be joined with the former one. Our approach tries to balance between finding the best combinations and giving the user the ability to explore the products of the database and the possible combinations between them, without the need to specify which relations will be joined to the main relation. To this end, we define the Exploratory Top-k Join query and we present a pull-bound framework for query processing. We propose a bounding scheme that exploits the properties of the formed combinations in order to efficiently calculate the result. The resulting algorithm has strong theoretical guarantees, namely it is instance-optimal. We also propose a more effective pulling strategy than plain round-robin, which further boosts the performance of query processing. In our experimental evaluation, we show that our algorithms perform consistently better than an adaptation of a state-of-the-art rank-join algorithm.

Acknowledgments

A. Vlachou was supported by the Action “Supporting Postdoctoral Researchers” of the Operational Program “Education and Lifelong Learning” (Action’s Beneficiary: General Secretariat for Research and Technology), and is co-financed by the European Social Fund (ESF) and the Greek State.

C. Doulkeridis has been co-financed by ESF and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) – Research Funding Program: Aristeia II, Project: ROADRUNNER

References

[1] G. Das, D. Gunopulos, N. Koudas, D. Tsirogiannis, Answering top-k queries using views, in: Proceedings of VLDB, 2006, pp. 451–462.
[2] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, J. Comput. Syst. Sci. 66 (4) (2003) 614–656.
[3] I.F. Ilyas, G. Beskales, M.A. Soliman, A survey of top-k query processing techniques in relational database systems, ACM Comput. Surv. 40 (4) (2008).
[4] I.F. Ilyas, W.G. Aref, A.K. Elmagarmid, Supporting top-k join queries in relational databases, VLDB J. 13 (3) (2004) 207–221.
[5] N. Mamoulis, M.L. Yiu, K.H. Cheng, D.W. Cheung, Efficient top-*k* aggregation of ranked inputs, ACM Trans. Database Syst. 32 (3) (2007) 19.
[6] J. Finger, N. Polyzotis, Robust and efficient algorithms for rank join evaluation, in: Proceedings of SIGMOD, 2009, pp. 415–428.
[7] S. Chaudhuri, L. Gravano, Evaluating top-*k* selection queries, in: Proceedings of VLDB, 1999, pp. 397–410.
[8] S. Ge, L.H. U, N. Mamoulis, D.W. Cheung, Efficient all top-k computation – a unified solution for all top-k, reverse top-k and top-m influential queries, IEEE Trans. Knowl. Data Eng. 25 (5) (2013) 1015–1027.
[9] Q. Wan, R.C. Wong, I.F. Ilyas, M.T. Özsu, Y. Peng, Creating competitive products, PVLDB 2 (1) (2009) 898–909.
[10] V. Hristidis, Y. Papakonstantinou, Algorithms and applications for answering ranked queries using ranked views, VLDB J. 13 (1) (2004) 49–70.
[11] A. Marian, N. Bruno, L. Gravano, Evaluating top-*k* queries over web-accessible databases, ACM Trans. Database Syst. 29 (2) (2004) 319–362.
[12] T. Wu, D. Xin, Q. Mei, J. Han, Promotion analysis in multi-dimensional space, PVLDB 2 (1) (2009) 109–120.
[13] J. Selke, C. Lofi, W. Balke, Pushing the boundaries of crowd-enabled databases with query-driven schema expansion, PVLDB 5 (6) (2012) 538–549.
[14] Y. Chang, L.D. Bergman, V. Castelli, C. Li, M. Lo, J.R. Smith, The onion technique: indexing for linear optimization queries, in: Proceedings of SIGMOD, 2000, pp. 391–402.
[15] M. Theobald, G. Weikum, R. Schenkel, Top-k query evaluation with probabilistic guarantees, in: Proceedings of VLDB, 2004, pp. 648–659.
[16] M. Xie, L.V.S. Lakshmanan, P.T. Wood, Efficient top-k query answering using cached views, in: Proceedings of EDBT, 2013, pp. 489–500.
[17] D. Xin, C. Chen, J. Han, Towards robust indexing for ranked queries, in: Proceedings of VLDB, 2006, pp. 235–246.
[18] Z. Zhang, C. Jin, Q. Kang, Reverse k-ranks query, PVLDB 7 (10) (2014) 785–796.
[19] K. Schnaitter, N. Polyzotis, Optimal algorithms for evaluating rank joins in database systems, ACM Trans. Database Syst. 35 (1) (2010) 6.
[20] K. Schnaitter, N. Polyzotis, Evaluating rank joins with optimal cost, in: Proceedings of PODS, 2008, pp. 43–52.
[21] O. Gkorgkas, A. Vlachou, C. Doulkeridis, K. Nørvåg, Efficient processing of exploratory top-k joins, in: Proceedings of SSDBM, 2014, p. 35.
[22] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, J. S. Vitter, Supporting incremental join queries on ranked inputs, in: Proceedings of VLDB, 2001, pp. 281–290.
[23] D. Martinenghi, M. Tagliasacchi, Cost-aware rank join with random and sorted access, IEEE Trans. Knowl. Data Eng. 24 (12) (2012) 2143–2155.
[24] D. Habich, W. Lehner, A. Hinneburg, Optimizing multiple top-k queries over joins, in: Proceedings of SSDBM, 2005, pp. 195–204.
[25] P. Agrawal, J. Widom, Confidence-aware join algorithms, in: Proceedings of ICDE, 2009, pp. 628–639.
[26] M. Xie, L.V.S. Lakshmanan, P.T. Wood, Efficient rank join with aggregation constraints, PVLDB 4 (11) (2011) 1201–1212.
[27] J. Lu, P. Senellart, C. Lin, X. Du, S. Wang, X. Chen, Optimal top-k generation of attribute combinations based on ranked lists, in: Proceedings of SIGMOD, 2012, pp. 409–420.
[28] W. Zhang, R. Cheng, B. Kao, Evaluating multi-way joins over discounted hitting time, in: Proceedings of ICDE, 2014, pp. 724–735.
[29] M.E. Khalefa, M.F. Mokbel, J.J. Levandoski, PrefJoin: an efficient preference-aware join operator, in: Proceedings of ICDE, 2011, pp. 995–1006.
[30] W. Jin, M. Ester, Z. Hu, J. Han, The multi-relational skyline operator, in: Proceedings of ICDE, 2007, pp. 1276–1280.
[31] W. Jin, M.D. Morse, J.M. Patel, M. Ester, Z. Hu, Evaluating skylines in the presence of equijoins, in: Proceedings of ICDE, 2010, pp. 249–260.
[32] C. Doulkeridis, A. Vlachou, K. Nørvåg, Y. Kotidis, N. Polyzotis, Processing of rank joins in highly distributed systems, in: Proceedings of ICDE, 2012, pp. 606–617.
[33] A. Angel, S. Chaudhuri, G. Das, N. Koudas, Ranking objects based on relationships and fixed associations, in: Proceedings of EDBT, 2009, pp. 910–921.
[34] X. Guo, Y. Ishikawa, Multi-objective optimal combination queries, in: Proceedings of DEXA, 2011, pp. 47–61.
[35] A.G. Parameswaran, H. Garcia-Molina, Recommendations with prerequisites, in: Proceedings of RecSys, 2009, pp. 353–356.
[36] S.B. Roy, S. Amer-Yahia, A. Chawla, G. Das, C. Yu, Constructing and exploring composite items, in: Proceedings of SIGMOD, 2010, pp. 843–854.
[37] M. Xie, L.V.S. Lakshmanan, P.T. Wood, Breaking out of the box of recommendations: from items to packages, in: Proceedings of RecSys, 2010, pp. 151–158.
[38] S. Amer-Yahia, F. Bonchi, C. Castillo, E. Feuerstein, I. Méndez-Díaz, P. Zabala, Composite retrieval of diverse and complementary bundles, IEEE Trans. Knowl. Data Eng. 26 (11) (2014) 2662–2675.
[39] P.J. Haas, J.F. Naughton, S. Seshadri, A.N. Swami, Fixed-precision estimation of join selectivity, in: Proceedings of PODS, 1993, pp. 190–201.
[40] Y. Tao, V. Hristidis, D. Papadias, Y. Papakonstantinou, Branch-and-bound processing of ranked queries, Inf. Syst. 32 (3) (2007) 424–445.