# Optimal Top-k Generation of Attribute Combinations based on Ranked Lists

Jiaheng Lu†, Pierre Senellart‡, Chunbin Lin†, Xiaoyong Du†, Shan Wang†, and Xinxing Chen†
† School of Information and DEKE, MOE, Renmin University of China; Beijing, China
‡ Institut Télécom; Télécom ParisTech; CNRS LTCI; Paris, France
{jiahenglu, chunbinlin,duyong,swang,xxchen}@ruc.edu.cn, pierre@senellart.com

## ABSTRACT

In this work, we study a novel top-$k$ query type, called top-$k$,$m$ queries. Suppose we are given a set of groups and each group contains a set of attributes, each of which is associated with a ranked list of tuples, with ID and score. All lists are ranked in decreasing order of the scores of tuples. We are interested in finding the best combinations of attributes, each combination involving one attribute from each group. More specifically, we want the top-$k$ combinations of attributes according to the corresponding top-$m$ tuples with matching IDs. This problem has a wide range of applications from databases to search engines on traditional and non-traditional types of data (relational data, XML, text, etc.). We show that a straightforward extension of an optimal top-$k$ algorithm, the Threshold Algorithm (TA), has shortcomings in solving the top-$k$,$m$ problem, as it needs to compute a large number of intermediate results for each combination and reads more inputs than needed. To overcome this weakness, we provide here, for the first time, a *provably instance-optimal* algorithm and further develop optimizations for efficient query evaluation to reduce computational and memory costs and the number of accesses. We demonstrate experimentally the scalability and efficiency of our algorithms over three real applications.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; H.3.3 [**Database Management**]: Information Search and Retrieval—*Search process*

## General Terms

Algorithms, Experimentation, Performance, Theory

## Keywords

keyword search, package search, top-$k$ querying

## 1. INTRODUCTION

During the last decade, the topic of top-$k$ query processing has been extensively explored in the database community due to its importance in a wide range of applications. In this work, we identify a novel top-$k$ query type that has wide applications and requires novel algorithms for efficient processing. Before we give the definition of our problem, we describe an interesting example to motivate it.

Assume a basketball coach plans to build a good team for an important game (e.g., Olympic games). He would like to select a combination of athletes including *forward*, *center*, and *guard* positions, by considering their historical performance in games. See Figure 1 for the example data, which comes from the NBA 2010–2011 pre-season. Each tuple in Figure 1(a) is associated with a pair (*game ID*, *score*), where the *score* is computed by an aggregation of various scoring items provided by the NBA for this game. To build a good team, one option is to select athletes with the highest score in each group (e.g., "Juwan Howard" in the "*forward*" group), or to calculate the average scores of athletes across all games. However, both methods overlook an important fact: *a strong team spirit is critical to the overall performance of a team*. Therefore a better way is to evaluate the performance of athletes by considering their combined scores in the *same* game. We can formalize this problem as to select the top-$k$ combinations of athletes according to their best top-$m$ aggregate scores for games *where they played together*. For example, as illustrated in Figure 1, $F_2C_1G_1$ is the best combination of athletes, as the top-2 games in which the three athletes played together are $G$02 and $G$05, and 40.27 (= 21.51 + 18.76) is the highest overall score (w.r.t. the sum of the top-2 scores) among all eight combinations. We study in this paper a new query type called top-$k$,$m$ query that captures this need of selecting the top combinations of different attributes, when each attribute is described by a ranked list of tuples.

### 1.1 Problem description

Given a set of groups $G_1,\ldots,G_n$ where each group $G_i$ contains multiple attributes $e_{i1},\ldots,e_{il_i}$, we are interested in returning top-$k$ combinations of attributes selected from each group. As an example, recall Figure 1: there are three groups, i.e., *forward*, *center*, and *guard*, and one athlete corresponds to one attribute in each group. Our goal is to select a combination of three athletes, each from a different group.

We suppose that each attribute $e$ is associated with a ranked list $L_e$, where each tuple $\tau \in L_e$ is composed of an ID $\rho(\tau)$ (taken out of an arbitrary set of identifier) and a score $\sigma(\tau)$ (from an arbitrary fully ordered set, say, the reals). The list is ranked in descending order of the scores. In the example NBA data, each athlete has a list containing the game ID and the corresponding score. Given a combination $\epsilon$ of attributes, a *match instance* $\mathcal{I}^\epsilon$ of $\epsilon$ is a set of tuples based on some arbitrary join condition on tuple IDs $\rho(\tau)$ from lists. For example, the game $G$02 (including the three tuples ($G$02, 8.91), ($G$02, 6.01), ($G$02, 6.59)) is a match instance by the equi-join of the game IDs. In the top-$k$,$m$ problem, we are interested in returning top-$k$ combinations of attributes which have the highest

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SIGMOD '12,* May 20–24, 2012, Scottsdale, Arizona, USA.
Copyright 2012 ACM 978-1-4503-1247-9/12/05 ...$10.00.

| | Forward | | Center | | Guard |
|---|---|---|---|---|---|
| $F_1$ | $F_2$ | $C_1$ | $C_2$ | $G_1$ | $G_2$ |
| Juwan Howard | LeBron James | Chris Bosh | Eddy Curry | Dwyane Wade | Terrel Harris |
| (G01, 9.31) | **(G02, 8.91)** | **(G05, 7.21)** | (G01, 3.81) | **(G02, 6.59)** | (G09, 7.10) |
| (G07, 9.02) | (G08, 8.07) | **(G02, 6.01)** | (G06, 3.59) | (G03, 6.19) | (G03, 6.01) |
| (G03, 8.87) | **(G05, 7.54)** | (G06, 5.58) | (G04, 3.21) | (G04, 5.81) | (G04, 3.79) |
| (G04, 5.02) | (G10, 7.52) | (G10, 5.51) | (G07, 3.03) | **(G05, 4.01)** | (G08, 3.02) |
| (G11, 4.81) | (G03, 6.14) | (G04, 5.00) | (G09, 2.07) | (G01, 3.38) | (G05, 2.89) |
| (G08, 4.02) | (G01, 5.05) | (G11, 3.09) | (G11, 1.70) | (G09, 2.25) | (G02, 2.52) |
| (G06, 4.31) | (G04, 5.01) | (G01, 2.06) | (G10, 1.62) | (G06, 1.52) | (G01, 2.00) |
| (G05, 3.59) | (G09, 3.34) | (G08, 2.03) | (G02, 1.59) | (G08, 1.51) | (G10, 1.59) |
| ...... | ...... | ...... | ...... | ...... | ...... |
| (G09, 2.06) | (G06, 3.01) | (G09, 1.98) | (G08, 1.19) | (G07, 1.00) | (G06, 1.52) |

(a) Source data of three groups

40.27(=21.51+18.76) is the largest among the aggregate scores of top-2

| $F_1C_1G_1$ | $F_1C_1G_2$ | $F_1C_2G_1$ | $F_1C_2G_2$ | **$F_2C_1G_1$** | $F_2C_1G_2$ | $F_2C_2G_1$ | $F_2C_2G_2$ |
|---|---|---|---|---|---|---|---|
| (G04, 15.83) | (G04, 13.81) | (G01, 16.50) | (G01, 15.12) | (G02, 21.51) | (G05, 17.64) | (G02, 17.09) | (G02, 13.02) |
| (G05, 14.81) | (G05, 13.69) | (G04, 14.04) | (G07, 12.05) | (G05, 18.76) | (G02, 17.44) | (G04, 14.03) | (G09, 12.51) |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

(b) Top-2 aggregate scores for each combination

**Figure 1: Motivating example using 2010–2011 NBA data. Our purpose is to select one athlete from each of the three groups. The best combination is $F_2C_1G_1$. Values in bold font indicate tuples contributing to the score of this best combination.**

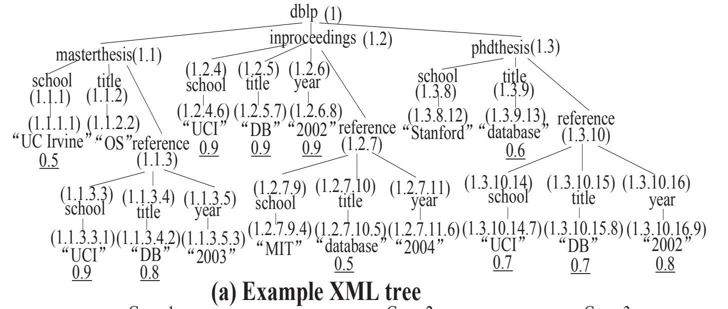
overall scores over their top-$m$ match instances by a monotonic aggregate function. Suppose that $k = 1$, $m = 2$, and that scores are aggregated using the sum function. The top-2 match instances of $F_2C_1G_1$ are $G02$ and $G05$ and their overall score is 40.27, which is the highest overall score among the top-2 match instances of all combinations. Therefore, we say that the answer of the top-1,2 query for the problem instance in Figure 1 is the combination $F_2C_1G_1$.

## 1.2 Applications

Next we give more use-case scenarios of top-$k$,$m$ queries, which shed some light on the generality and importance of top-$k$,$m$ models in practice.

*Use-case 1.* Top-$k$,$m$ queries are useful in *XML databases*. A simple yet effective way to search an XML database is keyword search. But in a real application it is often the case that a user issues a keyword query $Q$ which does not return the desired answers due to the mismatch between terms in the query and in documents. A common strategy for remedying this is to perform some query rewriting, replacing query terms with synonyms that provide better matches. Interestingly, top-$k,m$ queries find an application in this scenario. Specifically, for each keyword (or phrase) $q$ in $Q$, we generate a group $G(q)$ that contains the alternative terms of $q$ according to a dictionary which contains *synonyms* and *abbreviations* of $q$. For example, given a query $Q = \langle \text{DB}, \text{UC Irvine}, 2002\rangle$, we can generate three groups: $G_1$ = {"DB", "database"}, $G_2$ = {"UCI", "UC Irvine"}, and $G_3$ = {"2002"}. We assume that each term in $G(q)$ is associated with a list of node identifiers (e.g., JDewey IDs [7]) and scores (e.g., information-retrieval scores such as tf-idf [4]). See Figure 2 for an example XML tree and scores. The goal of top-$k$,$m$ queries is to find the top-$k$ combinations (of terms) by considering the corresponding top-$m$ search results in the XML database. Therefore, a salient feature of the top-$k$,$m$ model for the refinement of XML keyword query is that it guarantees that the suggested alternative queries have high quality results in the database within the top-$m$ answers.

*Use-case 2.* Top-$k$,$m$ queries also have applications in *evidence combination mining* in medical databases [2]. The goal is to predict or screen for a disease by mining evidence combination from clinical and pathological data. In a clinical database, each evidence refers



**(a) Example XML tree**

| Group1 | | Group2 | | Group3 |
|---|---|---|---|---|
| DB | database | UCI | UC Irvine | 2002 |
| (1.2.5.7, 0.9) | (1.3.9.13, 0.6) | (1.1.3.3.1, 0.9) | (1.1.1.1, 0.5) | (1.2.6.8, 0.9) |
| (1.1.3.4.2, 0.8) | (1.2.7.10.5, 0.5) | (1.2.4.6, 0.9) | | (1.3.10.16.9, 0.8) |
| (1.3.10.15.8, 0.7) | | (1.3.10.14.7, 0.7) | | |

**(b) Sorted lists and groups**

**Figure 2: An example illustrating XML query refinement using the top-$k$,$m$ framework. The original query $Q = \langle \textbf{DB}, \textbf{UC Irvine}, \textbf{2002}\rangle$ is refined into $\langle \textbf{DB}, \textbf{UCI}, \textbf{2002}\rangle$. Each term is associated with an inverted list with the IDs and weights of elements. Underlined numbers in the XML tree denote term scores.**

to a group and different degrees of the evidence act as different attributes. For example *queasiness*, *headaches*, *vomit*, and *diarrhea* are four pieces of evidence (referring to groups) for *acute enteritis*, and each evidence has different degrees (referring to attributes), e.g., *very low*, *low*, *middle*, *high*, *very high*. Each tuple in a list associated to an attribute consists of the patient ID and the probability the patient catches the disease. The goal of the top-$k,m$ query is to find the top-$k$ combinations of different degrees of evidences ordered by the aggregate values of the probabilities of top-$m$ patients for which there is the highest belief this disease is involved.

*Use-case 3.* Top-$k$,$m$ queries may finally have applications in package recommendation systems, e.g., the *trip selection* problem. Consider a tourist who is interested in planing a trip by choosing one hotel, one shopping mall, and one restaurant in a city. Assume that we have survey data provided by users who made trips before. The data include three groups and each group have multiple attributes (i.e., names of hotels, malls, or restaurants), each of which is associated with a list of users' IDs and grades. Top-$k$,$m$ queries recommend top-$k$ trips which are combinations of hotels, malls, and restaurants based on the aggregate value of the highest $m$ scores of the users who had the experience of this exact trip combination.

Generally speaking, top-$k$,$m$ queries are of use in any context where one is interested in obtaining combinations of attributes associated with ranked lists. In addition, note that the model of top-$k$,$m$ queries offers great flexibility in problem definitions to meet the various requirements that applications may have, in particular in the adjustment of the $m$ parameter. For example, in the application to XML keyword search, a user is often interested in browsing only the top few results, say 10, which means we can set $m = 10$ to guarantee the search quality of the refined keywords. In another application, e.g., trip recommendation, if a tourist wants to consider the average score of all users, then we can define $m$ to be large enough to take the scores of all users into accounts. (Of course, in this case, the number of accesses and the computational cost are higher.)

## 1.3 Novelty and contributions

The literature on top-$k$ query processing in relational and XML databases is particularly rich [5–8, 12, 13]. We stress the essential

difference between top-$k$ queries (e.g., as in [8]) and our top-$k$,$m$ problem: the latter returns the top-$k$ *combinations* of attributes in groups, but the former returns the top-$k$ *tuples* (objects). Therefore, a top-$k$,$m$ problem cannot be reduced to a top-$k$ problem through a careful choice of the aggregate function. In addition, contrarily to the top-$k$ problem, we argue that top-$k$,$m$ queries cannot be transformed into a SQL (nested) query, since SQL queries return tuples but our goal is to return *attribute combinations* based on ranked inverted lists, which is not something that the SQL language permits. To the best of our knowledge, this is the first top-$k$ work focusing on selecting and ranking sets of attributes based on ranked lists, which is a highly non-trivial extension of the traditional top-$k$ problem. An extended discussion about the difference between top-$k$,$m$ queries and the existing top-$k$ queries can be found in Section 3.

To answer a top-$k$,$m$ query, one method, called extended TA (for short ETA hereafter), is to compute all top-$m$ results for each combination by the well-known threshold algorithm (TA) [8] and then to pick the top-$k$ combinations. However, this method has one obvious shortcoming: it needs to compute top-$m$ results for *each* combination and reads *more* inputs than needed. To address this problem, we develop a set of *provably optimal* algorithms to efficiently answer top-$k$,$m$ queries.

Following the model of TA [8], we allow both random access and sorted access on lists. Sorted accesses mean to perform sequential scan of lists and random access can be performed only on objects which are previously accessed by sorted access. For example, consider Figure 1 again. When the first tuple $(G05, 7.21)$ of $C_1$ is read, the random access enables us to quickly locate all tuples whose ID are $G05$ (e.g., $(G05, 7.54)$ in $F_2$). We propose an algorithm called ULA (**U**pper and **L**ower bounds **A**lgorithm) to avoid the needs to compute top-$m$ results of combinations (Section 4) and thus to significantly reduce the computation costs compared to ETA.

We then bring to light some key observations and develop an optimized algorithm called ULA$^+$, which minimizes the number of accesses and consequently reduces the computational and memory costs. The ULA algorithm needs to compute bounds (lower and upper bounds) for each combination, which may be expensive when the number of combination is large. In ULA$^+$, we avoid the need to compute bounds for some combinations by carefully designing the conditions to prune away useless combinations without reading any tuple in the associated lists. We also propose a native structure called *KMG graph* to avoid the useless sorted and random accesses in lists to save computational and memory costs.

We study the optimal properties of our algorithms with the notion of *instance optimality* [8] that reflects how well a given algorithm performs compared to all other possible algorithms in its class. We show that two properties dictate the optimality of top-$k$,$m$ algorithms in this setting: including (1) the number of attributes in group $G_i$, namely $|G_i|$, which is part of the input of the problem; and (2) whether wild guesses are allowed. Following [8], wild guesses mean random access to objects which have not been seen by sorted access. Only if each $|G_i|$ is treated as a constant and there are no wild guesses is our algorithm guaranteed to be *instance-optimal*. In addition, we show that the optimality ratio of our algorithms is tight in a theoretical sense. Unfortunately, if either $|G_i|$ is considered variable or wild guesses exist (uncommon cases in practice), our algorithms are not optimal. But we show that in these cases *no* instance-optimal algorithm exists.

To demonstrate the applicability of the top-$k$,$m$ framework, we apply it to the problem of XML keyword refinement. We show how to judiciously design the aggregation functions and join predicates to reflect the semantic of XML keyword search. We then adapt the three algorithms: ETA, ULA and ULA$^+$ (from the most straightforward to the highly optimized one) to efficiently answer an XML top-$k$,$m$ problem (Section 5).

We verify the efficiency and scalability of our algorithms using three real-life datasets (Section 6), including NBA data, YQL trip-selection data, and XML data. We find that our top-$k$,$m$ algorithms result in order-of-magnitude performance improvements when compared to solutions based on the baseline algorithm. We also show that our XML top-$k$,$m$ approach is a promising and efficient method for XML keyword refinement in practice.

To sum up, this article presents a new problem with important applications, and a family of provably optimal algorithms. Comprehensive experiments verify the efficiency of solutions on three real datasets.

## 2. PROBLEM FORMULATION

Given a set of groups $G_1,\ldots,G_n$ where each group $G_i$ contains multiple attributes $e_{i1},\ldots,e_{il_i}$, we suppose that each attribute $e$ is associated with a ranked list $L_e$, where each tuple $\tau \in L_e$ is composed of an ID $\rho(\tau)$ and a score $\sigma(\tau)$. The list is ranked by the scores in descending order. Let $\epsilon = (e_{1i}, \ldots, e_{nj}) \in G_1 \times \cdots \times G_n$ denote an element of the cross-product of the $n$ groups, hereafter called *combination*. For instance, recall Figure 1, every three athletes from different groups form a combination (e.g., {Lebron James, Chris Bosh, Dwyane Wade}).

Given a combination $\epsilon$, a *match instance* $\mathcal{I}^\epsilon$ is defined as a set of tuples based on some arbitrary join condition on IDs of tuples from lists. Each tuple in a match instance should come from different groups. For example, in Figure 1, given a combination {Juwan Howard, Eddy Curry, Dwyane Wade}, then $\{(G01, 9.31), (G01, 3.81), (G01, 3.38)\}$ is a match instance for the game $G01$. Furthermore, we define two aggregate scores: *tScore* and *cScore*: the score of each match instance $\mathcal{I}^\epsilon$ is calculated by *tScore*, and the top-$m$ match instances are aggregated to obtain the overall score, called *cScore*. More precisely, given a match instance $\mathcal{I}^\epsilon$ defined on $\epsilon$,

$$tScore(\mathcal{I}^\epsilon) = \mathcal{F}_1(\sigma(\tau_1), \ldots, \sigma(\tau_n))$$

where $\mathcal{F}_1$ is a function: $\mathbb{R}^n \rightarrow \mathbb{R}$ and $\tau_1, \ldots, \tau_n$ form the matching instance $\mathcal{I}^\epsilon$. Further, given an integer $m$ and a combination $\epsilon$,

$$cScore(\epsilon, m) = \max_{\substack{\mathcal{I}_1^\epsilon,\ldots,\mathcal{I}_m^\epsilon \\ \text{distinct}}} \{\mathcal{F}_2(tScore(\mathcal{I}_1^\epsilon), \ldots, tScore(\mathcal{I}_m^\epsilon))\}$$

where $\mathcal{F}_2$ is a function $\mathbb{R}^m \rightarrow \mathbb{R}$ and $\mathcal{I}_1^\epsilon,\ldots, \mathcal{I}_m^\epsilon$ are any $m$ distinct match instances defined on the combination $\epsilon$. Intuitively, *cScore* returns the maximum aggregate scores of $m$ match instances. Following common practice [8], we require both $\mathcal{F}_1$ and $\mathcal{F}_2$ functions to be monotonic, i.e., the greater the individual score, the greater the aggregate score. This assumption captures most practical scenarios, e.g., if one athlete has a higher score (and the other scores remain the same), then the whole team is better.

DEFINITION 1 (TOP-$k$,$m$ PROBLEM). Given groups $G_1, \ldots, G_n$, two integers $k$, $m$, and two score functions $\mathcal{F}_1, \mathcal{F}_2$, the *top-$k$,$m$ problem* is an $(n + 4)$-tuple $(G_1, \ldots, G_n, k, m, \mathcal{F}_1, \mathcal{F}_2)$. A solution is an ordered set $\mathcal{S}$ containing the top-$k$ combinations $\epsilon = (e_{1i}, \ldots, e_{nj}) \in G_1 \times \cdots \times G_n$ ordered by $cScore(\epsilon, m)$. □

EXAMPLE 2. Consider a top-1, 2 query on Figure 1, and assume that $\mathcal{F}_1$ and $\mathcal{F}_2$ are *sum*. The final answer $\mathcal{S}$ is $\{F_2C_1G_1\}$. This is because the top-1 match instance $\mathcal{I}_1$ of $F_2C_1G_1$ consists of tuples $(G02, 8.91)$, $(G02, 6.01)$ and $(G02, 6.59)$ of the game $G02$ with *tScore* $21.51 = 8.91 + 6.01 + 6.59$. And the second top instance $\mathcal{I}_2$ consists of tuples whose game ID is $G05$ with *tScore*

$18.76 = 7.54 + 7.21 + 4.01$. Therefore, the *cScore* of $F_2C_1G_1$ is $40.27 = 21.51 + 18.76$, which is the highest score among all combinations. □

## 3. BACKGROUND AND RELATED WORK

Before we describe the novel algorithms for top-*k*,*m* queries, we pause to review some related works about top-*k* queries. Top-*k* queries were studied extensively for relational and XML data [5–8, 12, 13, 22]. Notably, Fagin, Lotem, and Naor [8] present a comprehensive study of various methods for top-*k* aggregation of ranked inputs. They identify two types of accesses to the ranked lists: sorted accesses and random accesses. In some applications, both sorted and random accesses are possible, whereas, in others, some of the sources may allow only sorted or random accesses. For the case where both sorted and random accesses are possible, a threshold algorithm (TA) (independently proposed in [10, 20]) retrieves objects from the ranked inputs in a round-robin fashion and directly computes their aggregate scores by using random accesses to the lists where the object has not been seen. Fagin et al. prove that TA is an instance-optimal algorithm. In this paper, we follow the line of TA to support both *sorted* accesses and *random* accesses for efficient evaluation of top-*k*,*m* query. We prove that our algorithm is also an instance-optimal algorithm and its optimality ratio is tight.

There is also a rich literature for top-*k* queries in other environments, such as no random access [9, 18], no sorted access on restricted lists [5, 6], no need for exact aggregate score [11], or ad-hoc top-*k* queries [15, 23]. For example, Mamoulis, Yiu, Cheng and Cheung [18] proposed a family of optimizations for top-*k* queries in the case of *no random accesses*. They impose two phases (*growing* and *shrinking*) that any top-*k* algorithm should go through, and perform optimizations on the *shrinking* phase to reduce the number of accesses. Theobald, Schenkel and Weikum [21] proposed a top-*k* query processor for efficient and self-tuning query expansion, which is related to the XML keyword refinement method described in Section 5. In contrast to our work, Theobald et al. also support a non-fixed number of keywords in the refined query; however, no optimality guarantees are given. Recently, Jin and Patel [13] proposed a novel sequential access scheme for top-*k* query evaluation, which outperforms existing schemes. For more information about top-*k* query evaluation, readers may refer to an excellent survey [12] by Ilyas, Beskales, and Soliman.

In this article, we argue that top-*k*,*m* queries are new types of queries and that the existing top-*k* algorithms cannot be used to solve them. For example, consider the ad-hoc top-*k* queries in [15, 23]. Note the difference regarding the notion of "*group*" between those works and ours. For ad-hoc queries, a group refers to a set of objects (rows) which satisfy certain predicates, while a group in this article means a set of attributes (columns). Therefore, top-*k*,*m* queries, in a different approach from ad-hoc top-*k* queries, focus on the ranking of the combinations of attributes (*not* objects). To the best of our knowledge, this is the first work focusing on selecting and ranking the set of attributes, which is a highly non-trivial extension of the traditional top-*k* problem.

Top-*k* processing in XML databases has recently gained more attention since XML has become the preferred medium for formatting and exchanging data in many domains [1, 7, 19]. There are various types of problems on XML top-*k* processing, including top-*k* twig query processing [1, 19], top-*k* keyword search [4], top-*k* probabilistic query processing [16] and top-*k* keyword cleansing [17]. In this article, we demonstrate how to apply the framework of top-*k*,*m* on the problem of XML top-*k* keyword refinement. Note the difference between keyword cleansing [17] and keyword refinement: the former rewrites the query by fixing spelling errors, but the latter rewrites the query using semantic knowledge such as synonyms and abbreviations. Both approaches are complementary in query processing and can be used together to improve search engine results.

## 4. TOP-K,M ALGORITHMS

In this section we begin our study of an efficient top-*k*,*m* algorithm which can stop earlier than the straightforward algorithm (i.e., ETA mentioned earlier), by avoiding the need to compute the exact top-*m* scores for each combination. We propose a family of optimizations to improve the performance by reducing the number of accesses and computational and memory costs. We also analyze the optimality properties for proposed algorithms.

### 4.1 Access model: sorted and random accesses

As mentioned in the Introduction, given an instance of a top-*k*,*m* problem, following the practice in the top-*k* literature (e.g., [8]), we support both sorted and random access. Sorted accesses read the tuple of lists sequentially and random accesses quickly locate tuples whose ID has been seen by sorted access (assuming the existence of an index to achieve this goal). For example, in Figure 1, at depth 1 (depth $d$ means the number of tuples seen under sorted access to a list is $d$), consider the combination "$F_2C_1G_1$"; the tuples seen by sorted access are $(G02, 8.91)$, $(G05, 7.21)$, $(G02, 6.59)$ and we can quickly locate all tuples (i.e., $(G02, 6.01)$, $(G05, 7.54)$, $(G05, 4.01)$) whose IDs are $G02$ or $G05$ by random accesses.

### 4.2 Baseline algorithm: ETA

To answer a top-*k*,*m* query, one straightforward method (called extended TA, or ETA for short) is to first compute all top-*m* results for each combination by the well-known threshold algorithm TA [8] and then pick the top-*k* combinations. However, this method has one obvious shortcoming: it needs to compute top-*m* results for *each* combination and reads *more* inputs than needed. For example, in Figure 1, ETA needs to compute the top-2 scores for all eight combinations (see Figure 1(b)). Indeed, this method is *not instance-optimal* in this context. To address this problem, we develop a set of *provably optimal* algorithms to efficiently answer top-*k*,*m* queries.

### 4.3 Top-k,m algorithm: ULA

When designing an efficient top-*k*,*m* algorithm, informally, we observe that a combination $\epsilon$ cannot contribute to the final answer if *there exist k distinct combinations whose lower bounds are greater than the upper bounds of* $\epsilon$. To understand this, consider the top-1,2 query in Figure 1 again. At depth 1, for the combination "$F_2C_1G_1$", we get two match instances $G02$ and $G05$ through the sorted and random accesses. Then the lower bound of the aggregate score (i.e., *cScore*) of "$F_2C_1G_1$" is at least 40.27 (i.e., $(7.54 + 7.21 + 4.01) + (8.91 + 6.01 + 6.59)$). At this point, we can claim that some combinations are not part of answers. This is the case of "$F_2C_2G_1$", whose *cScore* is no more than 38.62 ($= 2 \times (8.91 + 3.81 + 6.59)$). Since $38.62 < 40.27$, $F_2C_2G_1$ cannot be the top-1 combination. We next formalize this observation by carefully defining lower and upper bounds of combinations. We start by presenting threshold values, which will be used to estimate the upper bounds for the unseen match instances.

Definition 3 (threshold value). Let $\epsilon = (e_{1i}, \ldots, e_{nj}) \in G_1 \times \cdots \times G_n$ be an arbitrary combination, and $\tau_i$ the current tuple seen under sorted access in list $L_i$. We define the *threshold value* $\mathcal{T}^{\epsilon}$ of the combination $\epsilon$ to be $\mathcal{F}_1(\sigma(\tau_1), \ldots, \sigma(\tau_n))$, which is the upper bound of *tScore* for any unseen match instance of $\epsilon$. □

As an example, in Figure 1(a), consider the combination $\epsilon =$ "$F_2C_1G_1$", at depth 1. The current tuples are $(G02, 8.91)$, $(G05, 7.21)$,

$(G02, 6.59)$. Assume $\mathcal{F}_1 = sum$, we have for threshold value $\mathcal{T}^{\epsilon} = 8.91 + 7.21 + 6.59 = 22.71$.

Definition 4 (lower bound). Assume one combination $\epsilon$ has seen $m'$ distinct match instances. Then the *lower bound* of the *cScore* of $\epsilon$ is computed as follows:

$$\epsilon^{\min} = \begin{cases} \mathcal{F}_2(tScore(\mathcal{I}_1^{\epsilon}), \ldots, tScore(\mathcal{I}_{m'}^{\epsilon}), \underbrace{0, \ldots, 0}_{m-m'}) & m' < m \\ \max\{\mathcal{F}_2(\underbrace{tScore(\mathcal{I}_i^{\epsilon}), \ldots, tScore(\mathcal{I}_j^{\epsilon})}_{m})\} & m' \geqslant m \end{cases}$$ □

When $m' < m$, we use the minimal score (i.e., zero) of unseen $m - m'$ match instances to estimate the lower bound of the *cScore*. On the other hand, when $m' \geqslant m$, $\epsilon^{\min}$ equals the maximal aggregate scores of $m$ match instances.

Definition 5 (upper bound). Assume one combination $\epsilon$ has seen $m'$ distinct match instances, where there are $m''$ match instances ($m'' \leqslant m'$) whose scores are *greater than or equal to* $\mathcal{T}^{\epsilon}$. Then the *upper bound* of the *cScore* of $\epsilon$ is computed as follows:

$$\epsilon^{\max} = \begin{cases} \mathcal{F}_2(tScore(\mathcal{I}_1^{\epsilon}), \ldots, tScore(\mathcal{I}_{m''}^{\epsilon}), \underbrace{\mathcal{T}^{\epsilon}, \ldots, \mathcal{T}^{\epsilon}}_{m-m''}) & m'' < m \\ \max\{\mathcal{F}_2(\underbrace{tScore(\mathcal{I}_i^{\epsilon}), \ldots, tScore(\mathcal{I}_j^{\epsilon})}_{m})\} & m'' \geqslant m \end{cases}$$

□

If $m'' < m$, it means that there is still a chance that we will see a new match instance whose *tScore* contributes to the final *cScore*. Therefore, the computation of $\epsilon^{\max}$ should be padded with $m - m''$ copies of the threshold value (i.e., $\mathcal{T}^{\epsilon}$), which is the upper bound of *tScore* for all unseen match instances. Otherwise, $m'' \geqslant m$, meaning that the final top-$m$ results are already seen and thus $\epsilon^{\max}$=$cScore(\epsilon, m)$ now.

Example 6. This example illustrates the computation of the upper and lower bounds. See Figure 1 again. Assume that $\mathcal{F}_1$ and $\mathcal{F}_2$ are *sum*, and the query is top-1, 2. At depth 1, the combination "$F_2C_1G_1$" read tuples $(G02, 8.91)$, $(G05, 7.21)$, and $(G02, 6.59)$ by sorted accesses, and $(G05, 7.54)$, $(G02, 6.01)$, $(G05, 4.01)$ by random accesses. $m' = m = 2$. Therefore, the current lower bound of "$F_2C_1G_1$" is 40.27 (i.e., $(7.54 + 7.21 + 4.01) + (8.91 + 6.01 + 6.59) = 18.76 + 21.51$), since the two match instances of $F_2C_1G_1$ are $G02$ and $G05$. The threshold $\mathcal{T}^{F_2C_1G_1} = 8.91 + 7.21 + 6.59 = 22.71$ and $m'' = 0$, since $18.76 < 22.71$ and $21.51 < 22.71$. Therefore, the upper bound is 45.42 (i.e., 22.71 + 22.71). In fact, the final *cScore* of "$F_2C_1G_1$" is exactly 40.27 which equals the current lower bound. Note that the values of lower and upper bounds are dependent of the depth where we are accessing. For example, at depth 2, the upper bound of "$F_2C_1G_1$" decreases to 41.78 (i.e., 21.51 + 20.27) and the lower bound remains the same. □

The following lemmas show how to use the bounds above to determine if a combination $\epsilon$ can be pruned safely or confirmed to be an answer.

Lemma 7 (drop-condition). *One combination $\epsilon$ does not contribute to the final answers if there are k distinct combinations $\epsilon_1, \ldots, \epsilon_k$ such that $\epsilon^{\max} < \min\{\epsilon_i^{\min} \mid 1 \leqslant i \leqslant k\}$.*

Proof. The aggregate score of the top-$m$ match instances is no more then the upper bound of $\epsilon$, i.e., $cScore(\epsilon, m) \leqslant \epsilon^{\max}$. And $\forall i \in [1, k]$, $cScore(\epsilon_i, m) \geqslant \epsilon_i^{\min}$, since the $\epsilon_i^{\min}$ is the lower bound of $\epsilon_i$. Therefore, $cScore(\epsilon, m) < \min\{cScore(\epsilon_i', m) \mid |1 \leqslant i \leqslant k\}$, which means that $\epsilon$ cannot be one of the top-$k$ answers, as desired. □

Lemma 8 (hit-condition). *One combination $\epsilon$ should be an answer if there are at least $N_{\text{com}} - k$ ($N_{\text{com}}$ is the total number of the combinations) distinct combinations $\epsilon_1, \ldots, \epsilon_{N_{\text{com}}-k}$, such that $\epsilon^{\min} \geqslant \max\{\epsilon_i^{\max} \mid 1 \leqslant i \leqslant N_{\text{com}} - k\}$.*

Proof. The aggregate score of the top-$m$ match instances of $\epsilon$ is no less than the lower bound of $\epsilon$, i.e., $cScore(\epsilon, m) \geqslant \epsilon^{\min}$. And $\forall i \in [1, N_{\text{com}} - k]$, $\epsilon_i^{\max} \geqslant cScore(\epsilon_i, m)$. Therefore, $cScore(\epsilon, m) \geqslant \max\{cScore(\epsilon_i, m) \mid 1 \leqslant i \leqslant N_{\text{com}} - k\}$, meaning that the top-$m$ aggregate score of $\epsilon$ is larger than or equal to that of other $N_{\text{com}} - k$ combinations. Therefore $\epsilon$ must be one of the top-$k$,$m$ answers. □

Definition 9 (termination). A combination $\epsilon$ can be *terminated* if $\epsilon$ meets one of the following conditions: (i) the drop-condition, (ii) the hit-condition, or (iii) $\epsilon$ has seen $m$ match instances whose *tScore*s are greater than or equals to the threshold value $\mathcal{T}^{\epsilon}$. □

Intuitively, one combination is terminated if we do not need to compute its lower or upper bounds any further. The first two conditions in the above definition are easy to understand. The third condition means that we have found top-$m$ match instances of $\epsilon$. Note that we may not see the final top-$m$ match instances when $\epsilon$ satisfy the drop- or hit-condition.

We are now ready to present a novel algorithm named **ULA** (**U**pper and **L**ower bounds **A**lgorithm), that relies on the dynamic computing of upper and lower bounds of combinations (see Algorithm 1).

**Algorithm 1** The ULA algorithm

**Input**: a top-$k$,$m$ problem instance with $n$ groups $G_1, \ldots, G_n$, where each group has multiple lists $L_{ij} \in G_i$.
**Output**: top-$k$ combinations of attributes in groups.
- (i) Do sorted access in parallel to each of the sorted lists $L_{ij}$. As a tuple $\tau$ is seen under sorted access in some list, do random access to all other lists in $G_j$ ($j \neq i$) to find all tuples $\tau'$ such that $\rho(\tau) = \rho(\tau')$.
- (ii) For each unterminated combination $\epsilon$ (by Definition 9), compute $\epsilon^{\min}$ and $\epsilon^{\max}$, and check if $\epsilon$ can be terminated now.
- (iii) If there are at least $k$ combinations which meet the hit-condition, then the algorithm halts. Otherwise, go to step (i).
- (iv) Let $Y$ be a set containing the $k$ combinations (breaking ties arbitrarily) when ULA halts. Output $Y$.

In the last step of the algorithm, note that the set $Y$ is unordered by *cScore*. In the case where the output set should be ordered by *cScore*, we need to continuously maintain the lower and upper bounds of objects in $Y$ until their order is clear.

Example 10. We continue the example of Figure 1 to illustrate the ULA algorithm. First, in step (i) (at depth 1), ULA performs sorted accesses on one row for each list and does the corresponding random accesses. In step (ii) (at depth 1 again), it computes the lower and upper bounds for each combination, and then three combinations $F_1C_2G_2$, $F_2C_2G_1$ and $F_2C_2G_2$ are safely terminated, since their upper bounds (i.e., $\epsilon^{\max}_{F_1C_2G_1} = 39.42$, $\epsilon^{\max}_{F_2C_2G_1} = 38.62$ and $\epsilon^{\max}_{F_2C_2G_2} = 39.64$) are less than the lower bound of $F_2C_1G_1$ ($\epsilon^{\min}_{F_2C_1G_1} = 40.27$). Next, we go to step (i) again (at depth 2), as there is no combination satisfying the hit-condition in step (iii). Finally, at depth 4, $F_2C_1G_1$ meets the hit-condition and the ULA algorithm halts. To understand the advantage of ULA over ETA, note that ETA cannot stop at depth 4, since $F_2C_2G_1$ does not yet obtain its top-2 match instances. Indeed, ETA stops at depth 5 with 54 accesses, whereas ULA performs only 50 accesses by depth 4. □

## 4.4 Optimized top-k,m algorithm: ULA+

In this subsection, we present several optimizations to minimize the number of accesses, memory cost, and computational cost of the ULA algorithm by proposing an extension, called ULA$^+$.

*Pruning combinations without computing the bounds.* The ULA algorithm has to compute the lower and upper bounds for each combination, which may be an expensive operation when the number of combinations is large. We next propose an approach which prunes away many useless combinations safely *without computing their upper or lower bounds*.

We sort all lists in the same group by the scores of their top tuples. Notice that all lists are sorted by decreasing order. Intuitively, the combinations with lists containing small top tuples are guaranteed not to be part of answers, as their scores are too small. Therefore, we do not need to take time to compute their accurate upper and lower bounds. We exploit this intuitive observation by defining the precise condition under which a combination can be safely pruned without computing its bounds. We first define a relationship between two combinations called *dominating*.

Given a group $G$ in a top-$k$,$m$ problem instance, let $L_e$ and $L_t$ be two lists associated with attributes $e, t \in G$, we say $L_e$ dominates $L_t$, denoted $L_e \succ L_t$ if $L_e.\sigma(\tau_m) \geqslant L_t.\sigma(\tau_1)$, where $\tau_i$ denote the $i$th tuple in the list. That is, the score of the *mth* tuple in $L_e$ is greater than or equal to the score of the *first* tuple in $L_t$.

Definition 11 (domination). A combination $\epsilon = \{e_1, \ldots, e_n\}$ is said to *dominate* another combination $\xi = \{t_1, \ldots, t_n\}$ (denoted $\epsilon \succeq \xi$) if for every $1 \geqslant k \geqslant n$, either $e_i = t_i$ or $L_{e_i} \succ L_{t_i}$ holds, where $e_i$ and $t_i$ are two (possibly identical) attributes of the same group $G_i$. □

For example, in Figure 3, there are two groups $G_1$ and $G_2$. We say that the combination "$A_2B_1$" dominates "$A_3B_2$", because in the group $G_1$, 7.1 > 6.3 and in $G_2$, 8.2 > 8.0. In fact, "$A_2B_1$" dominates all combinations of attributes from $A_3$ to $A_n$ in $G_1$ and from $B_2$ to $B_n$ in $G_2$. Note that the lists in each group here are sorted by the scores of the top tuples.

Lemma 12. *Given two combinations $\epsilon$ and $\xi$, if $\epsilon$ dominates $\xi$ then the upper bound of $\epsilon$ is greater than or equal to that of $\xi$.*

Proof. If $\epsilon$ dominates $\xi$, then for every attribute $e$ in $\xi$, if $e \notin \epsilon$, then there is an attribute $t$ in $\epsilon$, s.t. the $m$-th tuple in the list $L_e$ has a larger score than the first tuple in $L_t$. Therefore, the upper bound of $m$ match instances of $\epsilon$ is greater than or equal to that of $\xi$. More formally, $\epsilon \succeq \xi \Rightarrow \forall i, L_{e_i}.\sigma(\tau_m) \geqslant L_{t_i}.\sigma(\tau_1) \Rightarrow \mathcal{F}_1(L_{e_1}.\sigma(\tau_m), \ldots, L_{e_n}.\sigma(\tau_m)) \geqslant \mathcal{F}_1(L_{t_1}.\sigma(\tau_1), \ldots, L_{t_n}.\sigma(\tau_1))$, since $\mathcal{F}_1$ is monotonic. So $m \times (\mathcal{F}_1(L_{e_1}.\sigma(\tau_m), \ldots, L_{e_n}.\sigma(\tau_m))) \geqslant m \times (\mathcal{F}_1(L_{t_1}.\sigma(\tau_1), \ldots, L_{t_n}.\sigma(\tau_1)))$. Note that

$$\epsilon^{\max} \geqslant m \times (\mathcal{F}_1(L_{e_1}.\sigma(\tau_m), \ldots, L_{e_n}.\sigma(\tau_m))),$$

since the threshold value and the scores of the unseen match instances of $\epsilon$ are no less than $\mathcal{F}_1(L_{e_1}.\sigma(\tau_m), \ldots, L_{e_n}.\sigma(\tau_m))$. In addition, it is easy to verify that $\xi^{\max} \leqslant m \times (\mathcal{F}_1(L_{t_1}.\sigma(\tau_1), \ldots, L_{t_n}.\sigma(\tau_1)))$. Therefore, $\epsilon^{\max} \geqslant \xi^{\max}$ holds, as desired. □

According to Lemma 12, if $\epsilon$ meets the drop-condition (Lemma 7), it means the upper bound of $\epsilon$ is small, then any combination $\xi$ which is dominated by $\epsilon$ (i.e., $\xi$'s upper bound is even smaller) can be pruned safely and quickly.

To apply Lemma 12 in our algorithm, the lists are sorted in descending order by the score of the first tuple in each list, which can be done off-line. We first access $m$ tuples sequentially for each list and perform random accesses to obtain the corresponding match instances. Then we consider two phases. (i) *Seed combination selection*. As the name indicates, seed combinations are used to trigger the deletion of other useless combinations. We pick the lists in descending order, and construct the combinations to compute their upper and lower bounds until we find one combination $\epsilon$ which meets the drop-condition, then $\epsilon$ is selected as the seed combination.

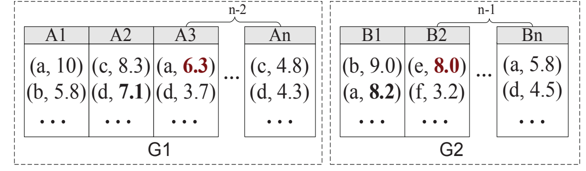



**Figure 3: An example for Lemma 12**

(ii) *Dropping useless combinations*. By Lemma 12, all combinations which are dominated by $\epsilon$ are also guaranteed *not* to contribute to final answers. For each group $G_i$, assuming that the seed combination $\epsilon$ contains the list $L_{ai}$ in $G_i$, then we find all lists $L_{bi}$ such that $L_{ai} \succ L_{bi}$. This step can be done efficiently as all lists are sorted by their scores of first tuples. Therefore, all the combinations which are constructed from $L_{bi}$ can be dropped safely without computing their upper or lower bounds.

Example 13. See Figure 3. Assume the query is top-1, 2 and $\mathcal{F}_1 = \mathcal{F}_2 = sum$. The lists are sorted in descending order according to the score of the first tuple. We access the lists in descending order to find the seed combination, which is $\xi = (A_2, B_1)$ ($\xi^{\max} = 2 \times (7.1 + 8.2) = 30.6 < \epsilon^{\min}$, $\epsilon = \{A_1, B_1\}$). In $G_1$, $\forall i \in [3, n]$ $L_{A_2} \succ L_{A_i}$ (e.g., $L_{A_2} \succ L_{A_3}$, since 7.1 > 6.3). Similarly, in $G_2$, $\forall i \in [2, n]$ $L_{B_1} \succ L_{B_i}$. Therefore all combinations $(A_i, B_j)$ ($\forall i \in [3, n]$, $j \in [2, n]$), as well as $(A_2, B_j)$ and $(B_1, A_i)$ are dominated by $\xi$ and can be pruned quickly. Therefore there are $(n-2)(n-1) + (n-1) + (n-2) = n^2 - n - 1$ combinations pruned without the (explicit) computation of their bounds, which can significantly save memory and computational costs. □

Note that in the ULA$^+$ algorithm (which will be presented later), we perform the two phases above as a preprocessing procedure to filter out many useless combinations.

*Reducing the number of accesses.* We now propose some further optimizations to reduce the number of accesses at three different levels: (i) avoiding both sorted and random accesses for specific lists; (ii) reducing random accesses across two lists; and (iii) eliminating random accesses for specific tuples.

Claim 14. *During query processing, given a list L, if all the combinations involving L are terminated, then we do not need to perform sorted accesses or random accesses upon the list L any longer.*

Claim 15. *During query processing, given two lists $L_e$ and $L_t$ associated with two attributes e and t in different groups, if all the combinations involving $L_e$ and $L_t$ are terminated, then we do not need to perform random accesses between $L_e$ and $L_t$ any longer.*

Claim 16. *During query processing, given two lists $L_e$ and $L_t$ associated with two attributes e and t in different groups, consider a tuple $\tau$ in list $L_e$. We say that the random access for the tuple $\tau$ from $L_e$ to $L_t$ is useless, if there exists a group G ($e \notin G$ and $t \notin G$) such that $\forall s \in G$, either of the two following conditions is satisfied: (i) the list $L_s$ does not contain any tuple $\tau'$, s.t. $\rho(\tau) = \rho(\tau')$; or (ii) the combination $\epsilon$ involving s, e and t is terminated.*

It is not hard to see Claim 14 and 15 hold. To illustrate Claim 16, let us consider three groups $G1$, $G2$ and $G3$ in Figure 4, where $G3$ contains only two lists. The list $L_s$ does not contain any tuple whose ID is $x$ and the combination $\epsilon$ is terminated. Therefore, according to Claim 16, the random access between $L_e$ and $L_t$ for tuple $x$ is unnecessary. This is because no match instances of $x$ can contribute to the computation of final answers. Note that it is common in real life that some objects are not contained in some list. For example, think of a player who missed some games in the NBA

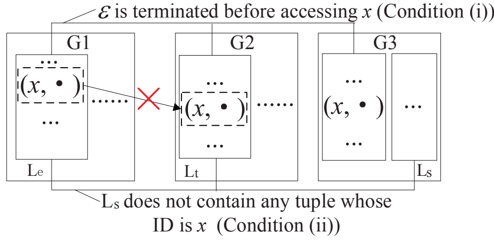



**Figure 4: Example to illustrate Claim 16. Assume there are two lists in group $G_3$. Random access from $L_e$ to $L_t$ is useless, since $\epsilon$ is terminated and $L_s$ does not contain any tuple whose ID is $x$.**

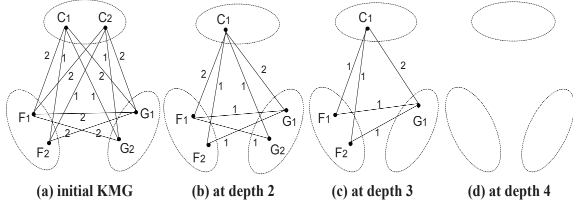



**Figure 5: Example top-k,m graphs (KMG)**

pre-season. Furthermore, to maximize the elimination of useless random accesses implied in Claim 16, in our algorithm, we consider the *Small First Access (SFA)* heuristic to control the order of random accesses, that is, we first perform random accesses to the lists in groups with less attributes. In this way, the random access across lists in larger groups may be avoided if there is no corresponding tuple in the list of smaller groups. As shown in our experimental results, Claim 16 and the SFA heuristic have significant practical benefits to reduce the number of random accesses.

Summarizing, Claim 14 through 16 imply three levels of granularity to reduce the number of accesses. In particular, Claim 14 eliminates both random accesses and sorted accesses, Claim 15 aims at preventing unnecessary random accesses, while Claim 16 comes in to avoid random accesses for some specific tuples.

In order to exploit the three optimizations in the processing of our algorithm, we carefully design a native data structure named *top-k,m graph* (called KMG hereafter). Figure 5(a) shows an example KMG for the data in Figure 1. Formally, given an instance $\Pi$ of the top-$k$,$m$ problem, we can construct a node-labeled, weighted graph $\mathcal{G}$ defined as $(V, E, W, C)$, where (1) $V$ is a set of nodes, each $v \in V$ indicating a list in $\Pi$, e.g., in Figure 5, node $F_1$ refers to the list $F_1$ in Figure 1; (2) $E \subseteq V \times V$ is a set of edges, in which the existence of edge $(v,v')$ means that random accesses between $v$ and $v'$ are necessary; (3) for each edge $e$ in $E$, $W(e)$ is a positive integer, which is the weight of $e$. The value is the total number of unterminated combinations associated with $e$; and finally (4) $C$ denotes a collection of subsets of $V$, each of which indicates a group of lists in $\Pi$, e.g., in Figure 5, $C$=$\{\{F_1, F_2\}, \{C_1, C_2\}, \{G_1, G_2\}\}$. A path of length $|C|$ in $\mathcal{G}$ that spans all subsets of $C$ corresponds to a combination in $\Pi$.

Based on the above claims, we propose three dynamic operations in KMG: (i) decreasing the weight of edges by 1 if one of combinations involving the edge is terminated; (ii) deleting the edge if its weight is 0, which means that random accesses between the two lists are useless (implied by Claim 15); and (iii) removing the node if its degree is 0, which indicates that both sorted and random accesses in this list are useless (implied by Claim 14).

*Optimized top-k,m algorithm.* We are now ready to present the ULA$^+$ algorithm based on KMG, which combines all optimizations implied by Claim 14 to 16. This algorithm is shown as Algorithm 2.

**Algorithm 2** The ULA$^+$ algorithm

**Input**: a top-$k$,$m$ problem instance with multiple groups and each group $G$ has multiple attributes and each attribute $n$ is associated with a list $L_n$.

**Output**: top-$k$ combinations of attributes in groups.

(i) Find the seed combination $\epsilon$ and prune all useless combinations dominated by $\epsilon$ according to the approach in Section 4.4.
(ii) Initialize a KMG $\mathcal{G}$ for the remaining combinations.
(iii) Do sorted accesses in parallel to lists with nodes in $\mathcal{G}$.
(iv) Do random accesses according to the existing edges in $\mathcal{G}$ (note that we need to first access the smaller group based on SFA strategy). In addition, given a tuple $\tau \in L_n$, $n \in G$, if there is another group $G'$ such that each node $n'$ in $G'$ (where $\exists$ edge $(n, n') \in \mathcal{G}$) does not contain the tuple with the same ID of $\tau$, then we can immediately stop all random accesses for $\tau$ (implied by Claim 16).
(v) Compute $\epsilon^{\min}$ and $\epsilon^{\max}$ for each unterminated combination $\epsilon$ and determine if $\epsilon$ is terminated now by Definition 9 using $\epsilon^{\min}$ and $\epsilon^{\max}$. If yes, decrease the weights of all edges involved in $\epsilon$ by 1. In addition, remove an edge if its weight is zero and remove a node $v \in \mathcal{G}$ if the degree of $v$ is zero.
(vi) Add $\epsilon$ to the result set $Y$ if it meets the hit-condition. If there are at least $k$ combinations which meet the hit-condition, then the algorithm halts. Otherwise, go to step (iii).
(vii) Output the result set $Y$ containing top-$k$ combinations.

EXAMPLE 17. We present an example with the data of Figure 1 to illustrate ULA$^+$. Consider a top-1,2 query again. Firstly, in step (i), ULA$^+$ performs sorted accesses to two rows of all lists, and finds a seed combination, e.g., $F_2C_1G_2$, as $\epsilon^{\max}_{F_2C_1G_2} = 40.18 < \epsilon^{\min}_{F_2C_1G_1} = 40.27$. Because $L_{C_1} \succ L_{C_2}$, the combination $\epsilon_{F_2C_1G_2}$ dominates $\epsilon_{F_2C_2G_2}$. Therefore, both $\epsilon_{F_2C_1G_2}$ and $\epsilon_{F_2C_2G_2}$ can be pruned in step (i). Then ULA$^+$ constructs a KMG (see Figure 5(a)) for non-pruned combinations in step (ii). Note that there is no edge between $F_2$ and $G_2$, since both $\epsilon_{F_2C_2G_2}$ and $\epsilon_{F_2C_1G_2}$ have been pruned. By depth 2, ULA$^+$ computes $\epsilon^{\min}$ and $\epsilon^{\max}$ for each unterminated combination in step (iii). Then $\epsilon_{F_1C_2G_1}$ and $\epsilon_{F_1C_2G_2}$ meet the drop-condition (e.g., $\epsilon^{\max}_{F_1C_2G_1} = 37.6 < \epsilon^{\min}_{F_2C_1G_1}$), and we decrease the weights by 1 for the corresponding edges, e.g., $w(F_1, G_1) = 1$. In addition, node $C_2$ should be removed, since all the combinations containing $C_2$ are terminated, (see Figure 5(b)) in step (iv). At depth 3, $\epsilon_{F_1C_1G_2}$ is terminated, since $\epsilon^{\max}_{F_1C_1G_2} = 36.48 < \epsilon^{\min}_{F_2C_1G_1}$, and we decrease the weights of $(F_1, C_1)$, $(F_1, G_2)$ and $(C_1, G_2)$ by 1 and remove the node $G_2$ (see Figure 5(c)). Finally, ULA$^+$ halts at depth 4 in step (vi) and $F_2C_1G_1$ is returned as the final result in step (vii). To demonstrate the superiority of ULA$^+$, we compare the numbers of accessed objects for three algorithms: ETA accesses 54 tuples (at depth 5) and ULA accesses 50 tuples (at depth 4) , while ULA$^+$ accesses only 37 tuples (at depth 4) . □

## 4.5 Optimality properties

We next consider the optimality of algorithms. We start by defining the optimality measures, and then analyze the optimality in different cases. Some of the proofs are omitted here due to space limitation; and most proofs are highly non-trivial.

**Competing algorithms.** Let $\mathbb{D}$ be the class of all databases. We define $\mathbb{A}$ to be all deterministic correct top-$k, m$ algorithms running on every database $\mathcal{D}$ in class $\mathbb{D}$ . Following the access model in [8], an algorithm $\mathcal{A} \in \mathbb{A}$ can use both sorted accesses and random accesses.

**Cost metrics.** We consider the number of tuples seen by sorted access and random access as the dominant computational

factor. Let $cost(\mathscr{A}, \mathscr{D})$ be the nonnegative performance cost measured by running algorithm $\mathscr{A}$ over database $\mathscr{D}$, which represents the amount of the tuples accessed.

**Instance optimality.** We use the notions of instance optimality. We say that an algorithm $\mathscr{A} \in \mathbb{A}$ is *instance-optimal* if for every $\mathscr{A}' \in \mathbb{A}$ and every $\mathscr{D} \in \mathbb{D}$ there exist two constants $c$ and $c'$ such that $cost(\mathscr{A}, \mathscr{D}) \leqslant c \times cost(\mathscr{A}', \mathscr{D}) + c'$.

Following [9], we say that an algorithm makes *wild guesses* if it does random access to find the score of a tuple with ID $x$ in some list before the algorithm has seen $x$ under sorted access. For example, in Figure 1, we can see tuples whose IDs are $G04$ only at depth 3 under sorted and random accesses. But wild guesses can magically find $G04$ in the first step and obtain the corresponding scores. In other words, wild guesses can perform random jump on the lists and locate any tuple they want. In practice, we would not normally implement algorithms that make wild guesses. We prove the instance optimality of ULA (and ULA$^+$) algorithm, provided the size of each group is treated as a constant. This assumption is reasonable as it is mainly about assuming that the *schema* of the database is fixed.

THEOREM 18. *Let $\mathbb{D}$ be the class of all databases. Let $\mathbb{A}$ be the class of all algorithms that correctly find top-k,m answers for every database and that do not make wild guesses. If the size of each group is treated as a constant, then ULA and ULA$^+$ are instance-optimal over $\mathbb{A}$ and $\mathbb{D}$.*

PROOF. According to the definition of instance optimality, the main goal of this proof is to show that for every $\mathscr{A} \in \mathbb{A}$ and every $\mathscr{D} \in \mathbb{D}$ there exist two constants $c$ and $c'$ such that $cost(ULA, \mathscr{D}) \leqslant c \times cost(\mathscr{A}, \mathscr{D}) + c'$. We obtain the values of $c$ and $c'$ as follows.

Assume that an optimal algorithm $\mathscr{A}$ halts by sorted access at most up to depth $d$. Since $\mathscr{A}$ needs to access at least one tuple in each list (otherwise we can easily make $\mathscr{A}$ err), the cost of $\mathscr{A}$ is at least $(d + \sum_{i=1}^{n} g_i - 1)C_s$, where $C_s$ denotes the cost of one sorted access and $g_i$ is the number of attributes in the group $G_i$.

We shall show that ULA halts on $\mathscr{D}$ by sorted access at most up to depth $d + m$. Then the cost of ULA is at most:

$$\begin{aligned} Cost &\leqslant (d+m)(\textstyle\sum_{i=1}^{n} g_i)C_s + (d+m)\sum_{i=1}^{n}\{g_i(\sum_{j=1}^{n} g_j - g_i)\}C_r \\ &= (d+m)(\textstyle\sum_{i=1}^{n} g_i)C_s + (d+m)\sum_{i\neq j}(g_i g_j)C_r \end{aligned}$$

where $C_r$ denotes the cost of one random access. For simplicity of presentation, let $T = \sum_{i=1}^{n} g_i$ and $K = \sum_{i\neq j}(g_i g_j)$. Hence, the cost of ULA is at most: $(d+m)TC_s + (d+m)KC_r$, which is $dTC_s + dKC_r$ plus an additive constant of $mTC_s + mKC_r$. So the optimality ratio $c = \frac{dTC_s + dKC_r}{dC_s} = T + KC_r/C_s$. Correspondingly, $c' = mTC_s + mKC_r - c(T-1)C_s = (T^2 + mT - T)C_s + (kT + mK - K)C_r$. It is easy to see that $c$ and $c'$ are two constants and are independent of the depth $d$.

The following part of the proof aims at showing ULA halts by depth $d + m$ (if the optimal algorithm stops by depth $d$). Let $Y$ be the output set of $\mathscr{A}$. There are now two cases, depending on whether or not $\mathscr{A}$ has seen the exact top-$m$ match instances for each combination when it halts.

**Case 1:** If $\mathscr{A}$ has seen the exact top-$m$ match instances for each combination, then ULA also halts by depth $d < d + m$, as desired.

**Case 2:** If $\mathscr{A}$ has not seen the exact top-$m$ match instances for each combination, then there are still two subcases depending on whether or not the lower bound of each combination $\epsilon \in Y$ is larger than the upper bound of the combinations not in $Y$ when $\mathscr{A}$ halts.

**Subcase 2.1:** For any combination $\epsilon \in Y$ and combination $\xi \notin Y$, $\epsilon^{\min} \geqslant \xi^{\max}$, that is, all the combinations in $Y$ meet hit-condition, and the size of $Y$ is $k$, so ULA halts by depth $d < d + m$, as desired.

**Subcase 2.2:** There exists one combination $\epsilon \in Y$ and one combination $\xi \notin Y$ such that $\epsilon^{\min} < \xi^{\max}$. At this point, our ULA algorithm cannot stop immediately. But since $\mathscr{A}$ is correct without seeing the remaining tuples after depth $d$, we shall prove that ULA algorithm accesses at most more $m$ depths (i.e., $\epsilon^{\min} \geqslant \xi^{\max}$ at that moment), otherwise we can easily make $\mathscr{A}$ err.

Given a list $L_i$ in a combination $\epsilon$, let $\sigma_i^{\epsilon}$ denote the seen minimal score (under sorted or random accesses) in $L_i$ at depth $d$. Assume that $\mathscr{A}$ has seen $m'$ ($m' \leqslant m$) match instances for $\epsilon$. Let $\omega = \mathcal{F}_1(\sigma_1^{\epsilon}, \ldots, \sigma_{|\epsilon|}^{\epsilon})$ denote the possible minimal *tScore*. Then we define an *mScore* as follows.

$$mScore(\epsilon, m) = \mathcal{F}_2(tScore(\mathcal{I}_1^{\epsilon}), \ldots, tScore(\mathcal{I}_{m'}^{\epsilon}), \underbrace{\omega, \ldots, \omega}_{m-m'})$$

Assume that $\mathscr{A}$ has seen $m''$ ($m'' \leqslant m$) match instances for $\xi$. Let $\lambda_i^{\xi}$ denote the unseen possible maximal score ($\lambda_i^{\xi} \leqslant \sigma(\tau)$) below $\tau$ in list $i$ by depth $d + (m - m'')$ of $\xi$. Let $\varphi = \mathcal{F}_1(\lambda_1^{\xi}, \ldots, \lambda_{|\xi|}^{\xi})$ denote the possible maximal *tScore*. Then *hScore* is defined as:

$$hScore(\xi, m) = \mathcal{F}_2(tScore(\mathcal{I}_1^{\xi}), \ldots, tScore(\mathcal{I}_{m''}^{\xi}), \underbrace{\varphi, \ldots, \varphi}_{m-m''})$$

Let us call a combination $\epsilon$ big if its *mScore* is larger than at least $N_{\text{com}} - k$ *hScore* of other combinations. We now show that every member $\epsilon$ of $Y$ is big. Define a database $\mathscr{D}'$ to be just like $\mathscr{D}$, except object unseen by $\mathscr{A}$. In $\mathscr{D}'$, assign unseen objects $V_1, \ldots, V_m$ with the score $\sigma_i^{\epsilon}$ under $\tau$ in each list $L_i \in \epsilon(\epsilon \in Y)$, and assign unseen objects $U_1, \ldots, U_m$ with the score $\lambda_i^{\xi}$ under $\tau$ in a list $L_j \in \xi(\xi \notin Y)$. Then $\mathscr{A}$ performs exactly the same, and gives the same output and accesses the same objects, for databases $\mathscr{D}$ and $\mathscr{D}'$. Then by the correctness of $\mathscr{A}$, it follows that all combinations in $Y$ is big.

Therefore by depth $d + m$, ULA would get at least $m$ match instances, and the lower bound of $\epsilon$ is no less than $mScore$, and the upper bound of $\xi$ is no more than $hScore$. Since $mScore(\epsilon,m) \geqslant hScore(\xi,m)$, so by depth $d + m$, $\epsilon^{\min} \geqslant \xi^{\max}$. Therefore, ULA halts by depth $d + m$, as desired. $\square$

The next theorem shows that the upper bound of the optimality ratio of ULA is tight, provided the aggregation functions $\mathcal{F}_1$ and $\mathcal{F}_2$ are strictly monotone (the proof is omitted due to space limitation, it can be found in a technical report that cannot be referenced due to the anonymous review).

THEOREM 19. *Assume that $\mathcal{F}_1$ and $\mathcal{F}_2$ are strictly monotonic functions. Let $C_r$ and $C_s$ denote the cost of one random access and one sorted access respectively. There is no deterministic algorithm that is instance-optimal for top-k,m problem, with optimality ratio less than $T + KC_r/C_s$, (which is the exact ratio of ULA), where $T = \sum_{i=1}^{n} g_i$, $K = \sum_{i\neq j}(g_i g_j)$, and $g_i$ denotes the number of lists in group $G_i$.*

When we consider the scenarios when an algorithm makes *wild guesses*, unfortunately, our algorithms are not instance-optimal, but we can show that in this case *no* instance-optimal algorithm exists. Note that this appears a somewhat surprising finding, because the TA algorithm for top-$k$ problems can guarantee instance optimality even under wild guesses for the data that satisfies the distinct property. In contrast, the ULA algorithm for top-$k$,$m$ problem is not instance-optimal even for distinct data. The intuition for this disparity is that top-$k$ problem needs to return the exact $k$ objects, forcing all algorithms (including those with wild guesses) to go through the list to verify the results, but an algorithm for top-$k$,$m$ search can correctly return $k$ combinations without seeing their $m$ objects by quickly locating a match instance to instantly boost the lower bound.

THEOREM 20. *Let $\mathbb{D}$ be the class of all databases. Let $\mathbb{A}$ be the class of all algorithms (wild guesses are allowed) that correctly find top-k,m answers for every database. There is no deterministic algorithm that is instance-optimal over $\mathbb{A}$ and $\mathbb{D}$.*

Finally, we consider the case (not so common in practice) when the number of attributes in each group is treated as a variable. While our algorithm is not instance-optimal in this case, we can show that *no* instance-optimal algorithm exists.

THEOREM 21. *Let $\mathbb{D}$ be the class of all databases. Let $\mathbb{A}$ be the class of all algorithms that correctly find top-k,m answers for every database. If the number of elements in each group is treated as a variable, there is no deterministic algorithm that is instance-optimal over $\mathbb{A}$ and $\mathbb{D}$.*

# 5. XML KEYWORD REFINEMENT

In this section, we study XML keyword query refinement using the top-*k*,*m* framework. We show how to judiciously define the aggregate functions and the join predicates in the top-*k*,*m* framework to reflect the semantics of XML keyword search and adapt the aforementioned three algorithms, i.e., ETA, ULA, and ULA$^+$.

## 5.1 XML keyword refinement

Given a set of keywords and an XML database $\mathbb{D}$, we study how to automatically rewrite the keywords to provide users better and more relevant search results on $\mathbb{D}$, as in real applications users' input may not have answers or the answers are not good. In particular, we rewrite the users' queries by two operations: *transformation* by rules and *deletion*. We assume that there exists a table to contain simple rules in the form of $A \rightarrow B$, where $A$ and $B$ are two strings. For example, "UC Irvine"→"UCI", "Database"→"Data base". These rules can be obtained from existing dictionaries, query log analysis [14], or manual annotation. Given a query $q = \{q_1, \ldots, q_n\}$, we scan all keywords sequentially and perform substring match by rules to generate groups.[1] For example, assume that $q$={UC Irvine, Database}; then the two groups are $G_1$={UC Irvine, UCI} and $G_2$={Database, DB}.

We assume that each node in an XML database is assigned with its JDewey identifier [7], which gives the order numbers to nodes at the same level and inherits the label of their ancestors as their prefix. In Figure 2, for example, *school*(1.2.4) shows that the label of its parent is 1.2 and *school* is the fourth node in level 3 from left to right. One good property of JDewey is that the number is a unique identifier among all nodes in the same tree depth.

In general, to convert the problem of XML keyword refinement to the top-*k*,*m* framework, given a keyword query $q = \{q_1, \ldots, q_n\}$, we first produce a set of groups $G_1,\ldots,G_t$ where each element $w_{ij} \in G_i$ is a keyword associated with an inverted list composed of binary tuples $\tau=\langle\rho(\tau), \sigma(\tau)\rangle$, where $\rho(\tau)$ is the JDewey label and $\sigma(\tau)$ is the score of the node (e.g., tf-idf). Then, the XML keyword refinement problem is to return top-*k* combinations of keywords that have the best aggregate scores in their top-*m* search results.

We now present a widely adopted approach (e.g., [7, 19]) to formally define *tScore* and *cScore* in the XML top-*k*,*m* problem. In the XML tree data model, LCA is the lowest common ancestor of multiple nodes and SLCA [1] is the root of the subtree containing matches to all keywords without a descendant whose subtree contains all keywords. In particular, given a combination $\epsilon$ and tuples $\tau_1 \ldots \tau_{|\epsilon|}$ from different groups, one match instance (i.e., keyword search result) is formed by the SLCA node $\tilde{n} = slca(\rho(\tau_1), \ldots, \rho(\tau_{|\epsilon|}))$. Let $x_i$=$\sigma(\tau_i) \times d(l^i - \tilde{l})$, where $l^i$ denotes the depth of node $\rho(\tau_i)$, $\tilde{l}$ is the depth of $\tilde{n}$, and $d(\cdot)$ is a decreasing function to leverage the score of SLCA at different levels (e.g., $d(x) = 0.9^x$ in our implementation). We define *tScore* to compute the score of one match instance $\mathcal{I}^\epsilon$ as

[1] In cases where one word (or a set of words) appears in multiple rules, we need to design an algorithm to generate the mapping from words to groups.

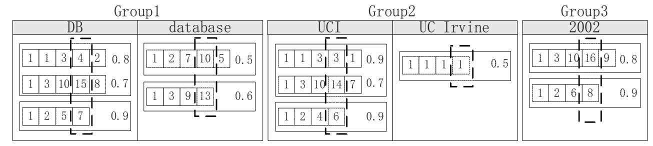



**Figure 6: JDewey labels are clustered by their lengths. The dashed boxes include all data in column 4.**

$tScore(\mathcal{I}^\epsilon) = \min(x_1, \ldots, x_{|\epsilon|})$. Intuitively, *tScore* assigns a greater score to an SLCA subtree with smaller sizes ($l^i - \tilde{l}$) and higher weights ($\sigma(\tau_i)$).

Given a combination $\epsilon$ and an integer $m$, and the *tScore*s of any $m$ match instances $\mathcal{I}_1^\epsilon, \ldots, \mathcal{I}_m^\epsilon$ on $\epsilon$, we define that

$$cScore(\epsilon, m) = \alpha^{|G|-|\epsilon|} \times \max\{tScore(\mathcal{I}_1^\epsilon) + \cdots + tScore(\mathcal{I}_m^\epsilon))\}$$

where $|G|$ is the number of groups, $|\epsilon|$ is the number of attributes in the combination $\epsilon$, and $\alpha$ is a damping constant (e.g., $\alpha = 0.7$ in our experiments). The choice of the component $\alpha^{|G|-|\epsilon|}$ is to give a penalty for *deletion* operation in keyword refinement in the sense that more deletions (i.e., smaller $|\epsilon|$) lead to smaller scores.

Compared to the regular top-*k*,*m* problem (in Section 4), XML top-*k*,*m* follows the same framework to return the top-*k* combinations ordered by *cScore*. But the concrete definitions of *tScore* and *cScore* are changed to cater for the tree-specific XML data. Therefore, to address the challenge from SLCA computation, we *convert SLCA subtree computation to ID equi-join in each level*. In particular, inspired by [7], we put the nodes into segments by the length of their labels and nodes are ordered by their scores at each level. See Figure 6. For example, in the list of "DB", data are clustered into two segments by label lengths (i.e., 5 and 4). The dashed boxes show columns corresponding to levels in XML trees, and numbers in one column uniquely identify nodes at that level (this is because of the feature of JDewey labels). In Figure 6, column 4 contains all the 4th JDewey numbers of the nodes. Note that the complete order of one column from different length segments can be reconstructed online, as we can maintain a cursor for each segment and pick one number with the highest score for all the cursors at each iteration. Therefore, to compute SLCA node, in column $i$, if two numbers have the same value $v$, then they share the same prefix path and their LCA subtree is uniquely identified by $v$. Furthermore, since we access the data in a bottom-up manner in the sense that we first access the column with greater column number, we guarantee that the first seen LCA is the smallest LCA node.

## 5.2 XML top-k,m algorithms

We now describe how to adapt the previous three top-*k*,*m* algorithms to work with XML trees. First, the application of ETA on XML (called XETA) is obvious. For each combination of keywords, we compute their exact top-*m* search answers and return the top-*k* combinations by sorting the final *cScore*. Clearly, this approach has to find the top-*m* search results for *all* the combinations, which is usually prohibitively expensive.

Second, to apply ULA on XML top-*k*,*m*, we iteratively access the JDewey numbers by columns in a bottom-up manner, while continuously computing the lower and upper bounds, until all top-*k* combinations are found. To compute the upper bound, the tricky issue here is that we access the numbers by columns and do not know the maximal values in other columns. However, this issue can be solved by collecting the scores of top nodes in each column in the preprocessing phase. More precisely, given a term (an attribute) $e_i$ and column $l$, let $y_{e_i}^l = \max\{z_l, \ldots, z_M\}$, where $M$ denotes the maximal length of nodes in the list of $e_i$ and $z_j$ =

$s^j \times d(j-l)$ ($l \leqslant j \leqslant M$), where $s^j$ is the top score of nodes in length $j$ and $d(\cdot)$ has been previously defined. For example, see Figure 6, $y^4_{DB} = \max(0.8 \times 0.9^1, 0.9 \times 0.9^0) = 0.9$, where $d(x) = 0.9^x$.

Suppose that we are accessing the JDewey label $\tau$ in column $l$, and the score of current number (representing an LCA node) can be computed as $x^l = \sigma(\tau) \times d(n - l)$, where $n$ is the total number of components in $\tau$. Given a combination $\epsilon$, the threshold value $\mathcal{T}^l_\epsilon$ can be defined as:

$$\mathcal{T}^l_\epsilon = \min\{t^l_{e_1}, \ldots, t^l_{e_{|\epsilon|}}\}$$

where $t^l_{e_i} = max\{x^l_{e_i}, y^1_{e_i}, \ldots, y^{l-1}_{e_i}\}$ ($1 \leqslant i \leqslant |\epsilon|$).

For example, in Figure 6, consider a combination "DB". (Note that a single word can be also considered as a combination due to deletion operation.) Assume that we are accessing the second tuple in column 4, i.e., the current number in list "DB" is 15 and the score $x^4_{DB} = 0.7 \times 0.9^{(5-4)} = 0.63$, then $\mathcal{T}^4_\epsilon = t^4_{DB} = \max\{x^4_{DB}, y^1_{DB}, y^2_{DB}, y^3_{DB}\} = \max\{0.63, 0.9^4, 0.9^3, 0.9^2\} = 0.81$, where $y^i_{DB} = \max\{0.8 \times 0.9^{(5-i)}, 0.9 \times 0.9^{(4-i)}\}$.

We present the *XULA* algorithm to address top-$k$,$m$ queries for XML keyword refinement in Algorithm 3. We iteratively access numbers from different columns in a bottom-up manner.

---

**Algorithm 3** The XULA algorithm

---

**Input**: an XML top-$k$,$m$ problem instance.
**Output**: top-$k$ combinations of keywords.

(i) Initialize a variable $l = H$, the height of XML trees. For each combination $\epsilon$, initialize an empty set $S_\epsilon$ to store SLCA nodes and their scores.
(ii) At column $l$, sorted access in parallel to each list and do the random accesses to get LCA nodes $u$. If $\neg\exists p \in S_\epsilon$, s.t. $u$ is an ancestor of $p$, insert $u$ into $S_\epsilon$.
(iii) For each unterminated combination $\epsilon$, compute the lower and upper bounds $\epsilon_{\min}$ and $\epsilon_{\max}$ (according to the top-$m$ nodes in $S_\epsilon$) and check if $\epsilon$ is terminated by Definition 9. Prune the combination $\epsilon$ by the drop condition (Lemma 7) or confirm $\epsilon$ to be part of the results by the hit condition (Lemma 8).
(iv) Let $Y$ be a set to contain the results. If there are $k$ combinations in $Y$ or all columns have been accessed, the algorithm halts. Output $Y$.
(v) If all nodes at column $l$ have been accessed, $l := l - 1$.
(vi) Go to step (ii).

---

EXAMPLE 22. Given a query $q = \langle \text{DB}, \text{UC Irvine}, 2002 \rangle$, we generate three groups $G_1 = \{\text{"DB"}, \text{"database"}\}$, $G_2 = \{\text{"UCI"}, \text{"UC Irvine"}\}$, and $G_3 = \{\text{"2002"}\}$. See Figure 6. Let $d(x) = 0.9^x$ and $\alpha = 0.7$. Consider a top-1,2 query. In step (i), $l$=5, depth=1, and there are 17 combinations (not 4, due to the deletion operation) and $\forall S_\epsilon = \emptyset$. In step (ii), the sorted accesses find four (trivial) LCA nodes (single node) (e.g., 1.1.3.4.2 in DB) and add them to the corresponding $S_\epsilon$. In step (iii), we compute the upper and lower bounds for all 17 combinations. For example, $\epsilon^{\max}_{DB} = 0.9\alpha^{(3-1)} + 0.9\alpha^{(3-1)} = 0.882$, $\epsilon^{\min}_{DB} = 0.8\alpha^{(3-1)}$=0.392. Then the conditions in step (iv) and (v) are not satisfied. Then we are in step (ii) again ($l$=5, depth=2) and add two new LCA nodes to $S_{DB}$ and $S_{UCI}$ respectively. In step (iii), at this moment, five combinations satisfy the drop-condition and are pruned. For example, $\epsilon^{\max}_{DB,UCIrvine} = 0.7 = 2\alpha^{(3-2)} \times \min(0.9, 0.5)$, which is smaller than $\epsilon^{\min}_{DB} = 0.735 = (0.7{+}0.8)\alpha^{(3-1)}$. Thus, the combination "{DB, UC Irvine}" is pruned. Now all nodes in column 5 have been accessed. Subsequently, we access nodes in column 4, 3 and 2. Finally, in column 2, we find the result is the combination "{DB, UCI, 2002}", which has two SLCA nodes (i.e., 1.3.10 and 1.2) and its lower bound is $0.567 + 0.729 = 1.296$, which is greater than the upper bounds of all other combinations. □

| Data set | # of objects | # of groups | | group size | | # of combinaiton | |
|---|---|---|---|---|---|---|---|
| | | max | avg | max | avg | max | avg |
| YQL | 100,100 | 3 | 3 | 150 | 12 | 3,375,000 | 1,728 |
| NBA | 31,200 | 5 | 5 | 32 | 6 | 33,554,432 | 7,776 |
| DBLP | 3,736,406 | 7 | 2.6 | 12 | 5 | 371,292 | 327 |

**Figure 7: Datasets and their characteristics**

Finally, to optimize the XULA algorithm, we can reuse the optimizations in ULA$^+$, except Claim 16. This is because XML keyword refinement supports the deletion operation and Claim 16 does not hold when an attribute is allowed to be deleted (recall Figure 4). We have to omit the details of the optimized XULA algorithm (called XULA$^+$) here due to the space limitation, but note that XULA$^+$ is implemented and tested in our experiments.

# 6. EXPERIMENTAL STUDY

In this section, we report an extensive experimental evaluation of our algorithms, using three real-life datasets. Our experiments were conducted to verify the efficiency and scalability of all three top-$k$,$m$ algorithms ETA, ULA and ULA$^+$; and their variants for XML keyword query refinement.

**Implementation and environment.** All the algorithms were implemented in Java and the experiments were performed on a dual-core Intel Xeon CPU 2.0GHz running Windows XP operating system with 2GB RAM and a 320GB hard disk.

**Datasets.** We use three datasets including NBA[2], Yahoo! YQL[3], and DBLP to test the efficacy of top-$k$,$m$ algorithms in the real world. Figure 7 summarizes the characteristics of the three datasets. NBA and Yahoo! YQL datasets were employed to evaluate the top-$k$,$m$ algorithms, while DBLP dataset was utilized to test the XML top-$k$,$m$ algorithms.

**NBA dataset.** We downloaded the data of 2010–2011 pre-season in NBA for the "*Point Guard*", "*Shooting Guard*", "*Small Forward*", "*Power Forward*" and "*Center*" positions. The original dataset contains thirteen dimensions, such as opponent team, shots, assists and score. We normalized the score of the data into $[0, 10]$ by assigning different weights to each dimension. There are five groups, and the average size of each group is about 6.

**YQL dataset.** We downloaded data about the hotels, restaurants, and entertainments from Yahoo! YQL[3]. The goal of the top-$k$,$m$ queries is to recommend the top-$k$ combinations of hotels, restaurants, and entertainments according to users' feedback. There are three groups, and the average size of each group is around 12.

**DBLP dataset.** The size of DBLP is about 127M. In order to generate meaningful query candidates, we obtained 724 synonym rules about the abbreviations and full names for computer science conferences and downloaded Babel[4] data including 9,136 synonym pairs about computer science abbreviations and acronyms.

**Choosing the XML queries.** Regarding to the real-world user queries, the most recent 1,000 queries are selected from the query log of a DBLP online demo [3], out of which 219 frequent queries (with an average length of 3.92 keywords) are selected to form a pool of queries that need refinement. Finally, we picked 186 queries that have meaningful refined results to test our algorithms. Here we show 5 sample XML keyword refinement as follows.

$Q_1$:{thomason, huang} is refined by adopting "thomason → thomas".
$Q_2$:{philipos, data, base} can be refined as {philipos, database}.

---

[2] http://www.nba.com/
[3] http://developer.yahoo.com/yql/console/
[4] http://www.wonko.info/ipt/babel.htm

| # of lists in each group | 5 | 10 | 15 | 20 | 25 | 30 |
|---|---|---|---|---|---|---|
| # of total combinations | 3125 | 100000 | 759375 | 3200000 | 9765625 | 24300000 |
| # of pruned combinations | 1875 | 80000 | 494325 | 2332800 | 7604375 | 19756800 |
| Pruning Percentage | 60.0% | 80.0% | 65.1% | 72.9% | 77.9% | 81.3% |

**Figure 8: The performance of optimization to reduce combinations**

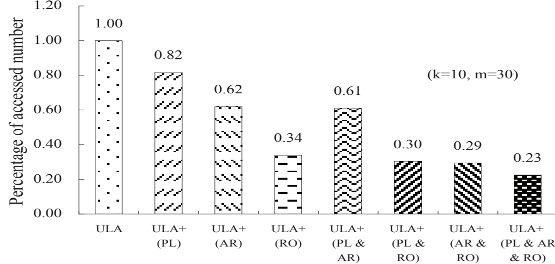



**Figure 9: The performance of different optimizations**

$Q_3$:{XML, key, word, application, 2008} is refined by deleting "2008", followed by a merging of "key" and "word".
$Q_4$:{jimy, kevin} which is refined by substituting "jimmy" for "jimy".
$Q_5$:{world, wild, web, search, engine, 2007}, which is refined by either adopting "world, wild, web" → "www" or deleting "2007".

**Metrics.** Our performance metrics are (1) running time: the cost of the overall time in executing top-*k*,*m* queries; (2) access number: the total number of tuples accessed by both sorted access and random access and (3) number of processed combinations: the total number of combinations processed in memory.

## Experimental Results

We inspected the results returned from all tested algorithms and found that their results are all the same, which verifies the validity of our algorithms. Each experiment was repeated over 10 times and the average numbers are reported here.

*Experimental results on NBA and YQL dataset.* Here we illustrate the performance of algorithms (ETA, ULA and ULA$^+$) on NBA and YQL dataset by varying parameters *k*, *m*, and the data size. In addition, we also deeply study the performance of different optimizations.

**Scalability with database size.** We evaluated the scalability of our algorithms with varying the number of tuples from 10K to 100K in both datasets. As shown in Figure 10(a)(b), both ULA and ULA$^+$ expose an amazingly stable performance without any significant fluctuation both in running time and number of accessed tuples while ETA scales linearly with the size of the database in NBA dataset. And in general, the execution time of ULA$^+$ outperforms ETA by 1-2 orders of magnitude, which verifies the efficiency of the optimizations. In addition, as we can see in Figure 10(e)(f), the results in YQL datasets are similar to that in NBA dataset.

**Performance vs. range of *k*.** In Figure 10(c)(g) we tested the number of accessed tuples for both random accesses (i.e., ULA(R) and ULA$^+$(R)) and sorted accesses (i.e., ULA(S) and ULA$^+$(S)) by varying *k* while fixing *m* on both NBA and YQL datasets. As shown, the number of random accesses are greater than that of sorted accesses for both ULA and ULA$^+$. In addition, ULA$^+$ has less accesses than ULA because of the effects of optimizations. Note that, the number of accessed tuples in the ETA algorithm is the same over all *k* values (i.e., 82K on NBA dataset and 70K on YQL dataset, without shown in the figures), because ETA has to obtain the exact top-*m* match instances for each combination independent of *k*.

**Performance vs. range of *m*.** The results with increasing *m* from 3 to 30 in NBA data and from 10 to 50 in YQL data are shown in Figure 10(d)(h), respectively. In general, both ULA and ULA$^+$ are 1 to 2 orders of magnitude more efficient than ETA method. In addition, ULA$^+$ is more efficient than ULA, which verifies the effects of our optimizations.

**Effect of the optimizations in ULA$^+$.** We then performed experiments to investigate the effects of four different optimizations in ULA$^+$. We fixed the parameters $k = 10$, $m = 30$ and the number of tuples is $100K$. First, to evaluate the approach of pruning useless combinations introduced in Section 4.4, we plotted Figure 8, which shows that the number of combinations processed in memory by our optimized algorithm is far less than that of ULA when the average number of lists in each group is increased from 10 to 50. More than 60% combinations are pruned without computing their bounds, thus significantly reducing the computational and memory costs. Second, to evaluate the effects of Claim 14 to 16, Figure 9 is plotted to evaluate the performance of different optimizations in terms of the number of accessed tuples. In particular, ULA$^+$(PL) uses Claim 14 to prune the whole lists to avoid useless accesses; ULA$^+$(AR) applies Claim 15 to avoid random accesses in some lists; and ULA$^+$(RO) employs Claim 16 to prevent random accesses for some tuples. In Figure 9, the second, third and fourth bars show the results to measure three optimizations individually, while the others are actually a combination of multiple optimizations. As shown, the combination of all optimizations has the most powerful pruning capability, reducing the accesses for almost 80%.

*Experimental results on DBLP dataset.* Finally, we run the experiments to test the scalability and efficiency of XETA, XULA and XULA$^+$ algorithms on DBLP dataset. In Figure 11(a)(b), we varied the size of DBLP dataset from 20% to 100% while keeping $k$=3, $m$=2. As expected, both XULA and XULA$^+$ perform better than XETA and scale well in both running time and number of accesses. In Figure 11(c)(d), we varied *k* from 1 to 5 while fixing $m = 2$ and 100% data size. As shown, both XULA and XULA$^+$ are far more efficient than XETA, and XULA$^+$ accesses 74.2% less objects than XULA and saves more than 35.1% running time, which indicates the effects of our optimizations.

# 7. CONCLUSION AND FUTURE WORK

We proposed a new problem called top-*k*,*m* query evaluation. We developed a family of efficient algorithms, including ULA and ULA$^+$. We analyzed different classes of data and access models, such as group size and wild guesses and their implication on the optimality of query evaluation. We then showed how to adapt our algorithms to the context of XML keyword query refinement. As for future work, extending our problem and algorithms with more access models and query types, e.g., non-random access model and non-monotone weight functions, while preserving optimal property, is a challenging item for future research.

# 8. ACKNOWLEDGEMENTS

This research is partially supported by 973 Program of China (Project No. 2012CB316205), NSF China (No. 60903056), 863 National High-tech Research Plan of China (No. 2012AA011001), and Beijing Natural Science Foundation (No. 4112030).

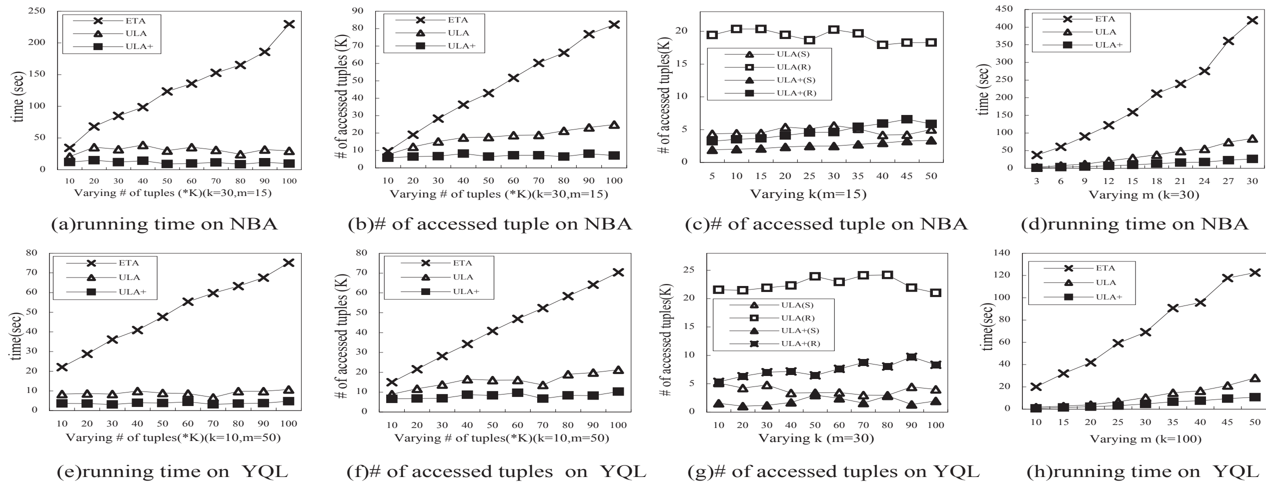



**Figure 10: Performance on NBA and Yahoo! YQL datasets**

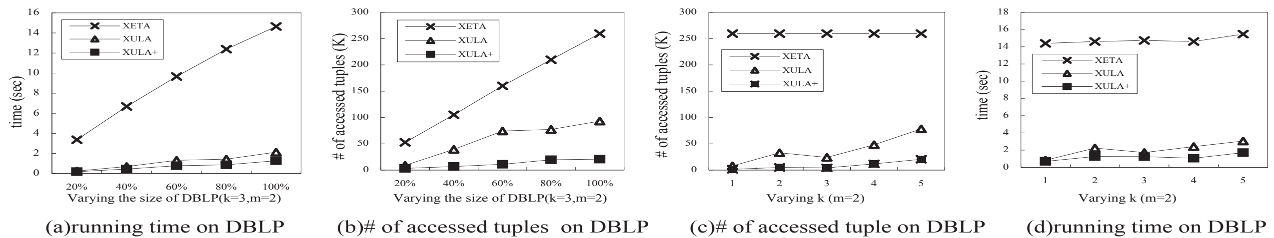



**Figure 11: Performance on XML DBLP dataset**

# 9. REFERENCES


[1] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and content scoring for XML. In *VLDB*, pages 361–372, 2005.

[2] Y. A. Aslandogan, G. A. Mahajani, and S. Taylor. Evidence combination in medical data mining. In *ITCC (2)*, pages 465–469, 2004.

[3] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective XML keyword search with relevance oriented ranking. In *ICDE*, pages 517–528, 2009.

[4] Z. Bao, J. Lu, T. W. Ling, and B. Chen. Towards an effective XML keyword search. *IEEE TKDE*, pages 1077–1092, 2010.

[5] N. Bruno, L. Gravano, and A. Marian. Evaluating top-*k* queries over web-accessible databases. In *ICDE*, 2002.

[6] K. C. C. Chang and S. Hwang. Minimal probing: supporting expensive predicates for top-*k* queries. In *SIGMOD*, 2002.

[7] L. J. Chen and Y. Papakonstantinou. Supporting top-*k* keyword search in XML databases. In *ICDE*, pages 689–700, 2010.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[10] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.

[11] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, 2002.

[12] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[13] W. Jin and J. M. Patel. Efficient and generic evaluation of ranked queries. In *SIGMOD*, pages 601–612, 2011.

[14] R. Jones and D. C. Fain. Query word deletion prediction. In *SIGIR*, pages 435–436, 2003.

[15] C. Li, K. C.-C. Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD*, pages 61–72, 2006.

[16] J. Li, C. Liu, R. Zhou, and W. Wang. Top-*k* keyword search over probabilistic XML data. In *ICDE*, 2011.

[17] Y. Lu, W. Wang, J. Li, and C. Liu. XClean: Providing valid spelling suggestions for XML keyword queries. In *ICDE*, 2011.

[18] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-*k* aggregation of ranked inputs. *ACM Trans. Database Syst.*, 32(3):19, 2007.

[19] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-*k* queries in XML. In *ICDE*, pages 162–173, 2005.

[20] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29, 1999.

[21] M. Theobald, R. Schenkel, and G. Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*, pages 242–249, 2005.

[22] D. Xin, J. Han, and K. C.-C. Chang. Progressive and selective merge: computing top-*k* with ad-hoc ranking functions. In *SIGMOD*, pages 103–114, 2007.

[23] M. L. Yiu, N. Mamoulis, and V. Hristidis. Extracting *k* most important groups from data efficiently. *Data Knowl. Eng.*, 66(2):289–310, 2008.