# Skip-and-Prune: Cosine-based Top-K Query Processing for Efficient Context-Sensitive Document Retrieval*


Jong Wook Kim
Comp. Sci. and Eng. Dept.
Arizona State University
Tempe, AZ, 85287, USA
jong@asu.edu

K. Selçuk Candan
Comp. Sci. and Eng. Dept.
Arizona State University
Tempe, AZ, 85287, USA
candan@asu.edu



## ABSTRACT

Keyword search and ranked retrieval together emerged as popular data access paradigms for various kinds of data, from web pages to XML and relational databases. A user can submit keywords without knowing much (sometimes nothing) about the complex structure underlying a data collection, yet the system can identify, rank, and return a set of relevant matches by exploiting statistics about the distribution and structure of the data. Keyword-based data models are also suitable for capturing user's search context in terms of weights associated to the keywords in the query. Given a search context, the data in the database can also be re-interpreted for semantically correct retrieval. This option, however, is often ignored as the cost of re-assessing the content in the database naively tends to be prohibitive. In this paper, we first argue that top-$k$ query processing can help tackle this challenge by re-assessing only the relevant parts of the database, efficiently. A road-block in this process, however, is that most efficient implementations of top-$k$ query processing assume that the scoring function is monotonic, whereas the cosine-based scoring function needed for re-interpretation of content based on user context is not. In this paper, we develop an efficient top-$k$ query processing algorithm, *skip-and-prune (SnP)*, which is able to process top-$k$ queries under *cosine-based* non-monotonic scoring functions. We compare the use of proposed algorithm against the alternative implementations of the context-aware retrieval, including naive top-$k$, *accumulator*-based inverted files, and full-scan. The experiment results show that while being fast, naive top-$k$ is not an effective solution due to the non-monotonicity of underlying scoring function. The proposed technique, *SnP*, however, matches the precision of *accumulator*-based inverted files and full-scan, yet it is orders of magnitude faster than these.



*Supported by NSF Grant "Archaeological Data Integration for the Study of Long-Term Human and Social Dynamics (0624341)"



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'09,* June 29–July 2, 2009, Providence, Rhode Island, USA.



Copyright 2009 ACM 978-1-60558-551-2/09/06 ...$5.00.



## Categories and Subject Descriptors

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

## General Terms

Experimentation, Performance

## Keywords

Top-K, Ranking

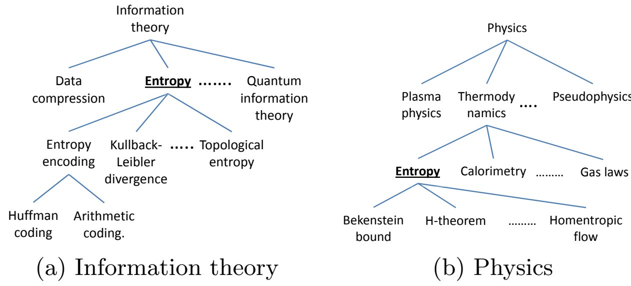

## 1. INTRODUCTION

Recently, there has been growing interest in integrating keyword search into systems that manage different types of data [1, 6, 8]. Keyword-based searches have become popular, especially for naive users (i.e., most of us) who do not want to master sophisticated query languages to access the data they need. Through a keyword-based search interface any user can issue a query without having to know a particular query language or the structure of the data.

While there are many keyword-based query models, including Boolean [21] and fuzzy [22] ones, the most commonly used keyword-based query model is the vector model [20]. Given a dictionary, $\mathcal{L}$, with $l$ distinct words, the set, $Q$, of query keywords are represented in the form of an $l$-dimensional vector $\vec{q}$, where each vector dimension corresponds to a unique word in this dictionary[1]. The vector $\vec{q}$ is constructed such that only those positions in the vector that correspond to the query words are set to 1 and all others are set to 0. A similar vector-based model is also used to represent documents in the database: the set of words in each document is mapped onto a vector in the same $l$-dimensional space.

While the simplest vector construction approach is to record the existence of a word in a given document/query using a 1 and the absence using a 0, a more commonly used technique is to encode the importance of the keyword using real-valued weights between 0 and 1. Most often, the weight of a keyword in a vector represents how often the keyword occurs in the document (the more often the keyword occurs, the higher is the weight) and how discriminatory/rare the keyword is in the database (the rarer the keyword is, the higher its weight). The popular TF/IDF keyword weights rely on this principle [22]. The vector representation also provides a

[1]In practice, many words (such as "a" and "the") in a real dictionary would be omitted as *stop-words* because of their high frequencies, most derived words would be clustered by *stemming* them into their roots, and sometimes sets of words (or stems) that are commonly used together would be combined into *compound words*.




(a) Information theory (b) Physics

**Figure 1: The keyword "entropy" appears in two different taxonomies, each specifying a different knowledge domain**

natural way to capture the user's search context by adjusting the weights of the keywords in the query: if we know, for example, that in a 2-keyword query "SIGMOD Conference", "SIGMOD" is more important for retrieval than "Conference", instead of using 1 for both query keywords, we can assign a lower weight (say 0.5) to "Conference". This property of the keyword search is commonly used to implement feedback-based, transparent query revision algorithms [19].

In most retrieval systems, given a query vector $\vec{q}$ and a document vector $\vec{d}$, the match between the vectors is computed using the so called *cosine similarity*:

$$sim_{cos}(\vec{d},\vec{q}) = cos(\vec{d},\vec{q}) = \frac{\vec{d}\cdot\vec{q}}{\|\vec{d}\|\ \|\vec{q}\|},$$

where "$\cdot$" denotes the *dot product* operator. Intuitively, $sim_{cos}(\vec{d},\vec{q})$ measures the degree of alignment between the interests of the user and the characteristics of the document: in the best case, when the vectors are identical, the similarity score is 1. In fact, to get a perfect match, the vectors do not need to be identical; as long as the angle between the vectors is $0^o$ degrees (i.e., their relative keyword compositions are similar) then the document and query vectors are said to match perfectly ($sim_{cos}(\vec{d},\vec{q}) = 1$). Naturally, as the angle between the two vectors grow, the difference between the query and the document also gets larger.

## 1.1 Search Context

Let us consider two users who are searching for documents having to do with the keyword "entropy". For the sake of the example, let us assume that the first user is a computer science student, who is interested in document related to "entropy" within CS context (Figure 1(a)). Let us further assume that the second user is a physics student, who, naturally, interprets the term within its physics context (Figure 1(b)). For these two students, the keywords in the documents in the database as well as the keywords in the query carry different meanings and connotations. Consequently, when the proper contexts are taken into account, the search should return different results for these two users:

$$sim_{cos}(\vec{d}_{cs},\vec{q}_{cs}) = cos(\vec{d}_{cs},\vec{q}_{cs}) \neq cos(\vec{d}_{p},\vec{q}_{p}) = sim_{cos}(\vec{d}_{p},\vec{q}_{p}),$$

Above the subscript "$cs$" denotes the interpretation of the keywords in the vector within the computer science context and the subscript "$p$" denotes the physics-based interpretation of the vectors.

Let $R$ be an $l \times c$ matrix which relates the $l$ terms in the dictionary with the $c$ concepts in the user's context (based on a criterion, such as string similarity [18]). Let $C$ be a $c \times c$

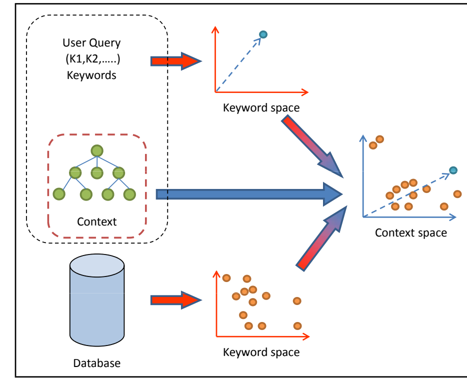



**Figure 2: Mapping of documents and the query from the keyword space into the *context space***

matrix, which relates the concepts in the user context to each other (i.e., the relationship between different concepts based on taxonomy analysis [18]). Given these matrices, $R$ and $C$, the similarity value between the user query, $q$, and a document, $d$, in the database can be computed by revising the similarity formula as follows[2]:

$$sim_{cos}(\vec{d},\vec{q},R,C) = cos(CR^T\vec{d},CR^T\vec{q}),$$

where $R^T$ is the transpose of the matrix, $R$. Intuitively, by multiplying the original $l$ dimensional query vector with $CR^T$, we obtain a $c$ dimensional query vector, which collectively captures the keywords weights of the original vector, the mapping between the original dictionary and the significant concepts in the user's context, as well as relationships between these concepts in this context. The document vector is also brought to the same $c$ dimensional concept space by multiplying it with $CR^T$ (Figure 2). Matching and ranking is, then, performed in this $c$ dimensional context space. In the rest of the paper, we use the shorthand notation $\mathtt{U}$ to refer to $CR^T$ (i.e., $\mathtt{U} = CR^T$).

## 1.2 Why is Using "Context" in Searches Hard?

As described above, given $R$ and $C$ matrices (or simply $\mathtt{U} = CR^T$), redefining the search criterion is not hard:

$$sim_{cos}(\vec{d},\vec{q},\mathtt{U}) = cos(\mathtt{U}\vec{d},\mathtt{U}\vec{q}),$$

While obtaining the context-specific matrices, $R$ and $C$, is challenging, there are also plenty of techniques for discovering them[3]. The main obstacle in leveraging the "context" in searches stems from efficiency concerns: reinterpreting a large number of document vectors based on a user specific matrix in run-time can be prohibitively costly.

To avoid reinterpretation of document vectors in run-time, an alternative scheme,

$$sim'_{cos}(\vec{d},\vec{q},\mathtt{U}) = cos(\vec{d},\mathtt{U}^T\mathtt{U}\vec{q}),$$

is sometimes used. This brings the query vector into the concept space and then re-aligns it with the dictionary. How-

[2] A slightly simpler alternative formulation, $sim_{cos}(\vec{d},\vec{q},R,C) = cos(R^T\vec{d},CR^T\vec{q})$ is also sometimes used. Similar results hold in that case as well.

[3] While, in our example, we used an explicit taxonomy to describe the significant concepts and their relationships, the matrix $C$ can also be learned from the user through other mining techniques, such as [23].

ever, since

$$\frac{(\mathtt{U}\vec{d}) \cdot (\mathtt{U}\vec{q})}{\|\mathtt{U}\vec{d}\| \; \|\mathtt{U}\vec{q}\|} \neq \frac{\vec{d} \cdot (\mathtt{U}^T \mathtt{U}\vec{q})}{\|\vec{d}\| \; \|\mathtt{U}^T \mathtt{U}\vec{q}\|},$$

as long as U is not the identity matrix (i.e., the user's context is null), this is semantically incorrect. To see the impact of reinterpreting the document vectors, consider Figure 2: as visualized in this figure, the distribution of the document vectors in the original keyword space and the space representing the user's context can be different and this difference cannot be captured by modifying the query vector alone.

### 1.3 Our Contribution: Cosine-based Ranked Query Processing for Retrieval within the User Context

The goal in this paper is to develop efficient query processing algorithms to locate the best documents in a database that match a given keyword query *within a user specific context*. Complete reinterpretation of the entire set of document vectors in the database is clearly too expensive to be a viable option. A more promising alternative is to leverage top-$k$ query processing techniques, such as [9, 10], to reinterpret only the necessary documents, on a per-need basis. This could help prune large portions of the database and provide significant savings.

The key obstacle, however, is that most top-$k$ ranked query processing schemes assume that the underlying scoring function is monotonic (e.g *max*, *min*, *product*, and *average*). These scoring functions guarantee that a candidate dominating (or equal to) the other one in its sub-scores will have a combined score better than (or as good as) the other one. This, however, is not the case for a scoring function based on *cosine* similarity. For example, given two pairs, $\langle 0.2; 0.2\rangle$ and $\langle 0.2; 0.8\rangle$, the second pair is dominating the first one, yet we have

$$cos(\langle 1;1\rangle, \langle \mathbf{0.2;0.2}\rangle) = 1 \;>\; 0.857 = cos(\langle 1;1\rangle, \langle \mathbf{0.2;0.8}\rangle).$$

Thus, a scoring function of the form, $score(x) = cos(\langle 1;1\rangle, x)$ would not be monotonic. We can easily generalize this and state that, in general, a cosine-based scoring function, which compares documents in the database to the user's query, is not monotonic. *Since a cosine-based scoring function is not monotonic, the use of existing top-k algorithms would lead to errors in the top-k results.*

In this paper, we develop an efficient query processing algorithm, *skip-and-prune (SnP)*, to process top-$k$ queries with a *cosine-based* scoring function and use it for keyword based retrieval in the presence of a user specific search context. In particular, the approach presented in this paper is able to identify and eliminate those documents that are guaranteed not to produce the top-$k$ highest scores at an early phase. We evaluate the performance of the proposed algorithm for various parameters, using synthetic as well as real data collections, and show that the proposed technique provides significant execution time gains over existing solutions to this problem.

The rest of this paper is structured as follows: In the next section, we formalize the problem and introduce the notation we will use to develop and describe our algorithms. In Section 2.5, we present a naive approach to highlight the underlying efficiency challenges. In Section 3, we present our algorithm for processing top-$k$ queries with the *cosine* scoring function and in Section 4, we experimentally evaluate our approach using synthetic as well as real data sets. In Section 5, we present the related work not already covered in the paper and, then, we conclude the paper.

## 2. PROBLEM AND THE NAIVE SOLUTIONS

In this section, we formally define the problem and introduce the key terminology which we will rely on when describing the algorithms proposed in this paper.

### 2.1 Document-Keyword Matrix

Given a dictionary, $\mathcal{L}$, the document collection, $\mathcal{DB}$, is represented in the form of a $m \times l$ *document-keyword matrix* D, where $m$ is the number of documents and $l$ is the number of distinct keywords in the dictionary. Without loss of generality, we assume that each row vector[4]. $\mathtt{D}[\mathtt{i}, -] = [w_{i,1}, w_{i,2}, w_{i,3}, \cdots, w_{i,l}]^T$ of D is normalized into unit length for efficient query processing.

### 2.2 Query-Keyword Vector

Given a dictionary, $\mathcal{L}$, user's keyword query is represented as an $l$-dimensional vector, $\vec{q}$. Again, without loss of generality, we assume that the query vector is also normalized into a unit length.

### 2.3 User Specific Context Matrix

As described in the previous section, user's context is abstracted in the form of a $c \times l$ matrix, U, where $c$ is the number of significant concepts in the current context. Intuitively, given any $l$-dimensional vector $\vec{v}$ in the original keyword space, $\vec{v}_u = \mathtt{U}\;\vec{v}$, is its interpretation in the $c$-dimensional user specific concept space. Once again, without loss of generality, we assume that each column vector $\mathtt{U}^T[-, j]$ is normalized into unit length.

### 2.4 User-Context Aware Retrieval Problem

The document retrieval problem in the presence of user-specific contexts can be stated as follows: "*Given*

- *a dictionary,* $\mathcal{L}$,
- *a document-keyword matrix,* D, *representing the document database,* $\mathcal{DB}$,
- *a keyword query vector,* $\vec{q}$, *and*
- *a user specific context matrix,* U,

*find a set* $Res = \{d_1, \ldots, d_k\}$ *of* $k$ *documents in* $\mathcal{DB}$, *such that the following holds:*

$$\forall_{d_i \in Res, d_j \notin Res} \quad cos(\mathtt{U}\,\mathtt{D}^T[-, i], \mathtt{U}\,\vec{q}) \geq cos(\mathtt{U}\,\mathtt{D}^T[-, j], \mathtt{U}\,\vec{q}).$$

In other words, answers to the user's query consists of the documents having the highest cosine similarity score with the query in the user specific concept space.

### 2.5 Naive Solution Strategies

The set of documents satisfying the query within the user-context can be identified in various ways.

**Scan and Choose.** A straight-forward query processing strategy would be to scan the vectors of the entire document base, reinterpret them according to the user's context by multiplying each document vector with U, and maintaining only the $k$-best solutions. An equivalent strategy can be implemented by representing all

[4]Given a matrix A, we use $\mathtt{A}[\mathtt{i}, -]$ to represent the $i$-th row vector and $\mathtt{A}[-, \mathtt{j}]$ to represent the $j$-th column vector.

matrices and vectors in the form of database tables, $D(doc_id, word_id, weight)$, $U(concept_id, word_id, weight)$, and $Q(word_id, weight)$ and perform a sequence of join and group-by operations to obtain a list of documents from which the top $k$ document can be selected. Both schemes would be quite expensive since all the database vectors are scanned and re-interpreted with U at least once. Moreover, since only the best $k$ results are needed, potentially, most of the computation will be wasted.

**Accumulator-based Inverted Files.** A potentially more efficient query processing strategy would benefit from *inverted files*, commonly used in IR systems. An inverted file index [13] is an access structure containing all the distinct words that one can use for searching. For each word, a pointer to the corresponding inverted list (containing IDs of documents that contain this word) is maintained. A search structure (hash file, B$^+$-tree) is used to enable quick access to the directory of inverted lists. Given a query with multiple keywords, the matching and ranking processes can be tightly-coupled to reduce the retrieval cost [27]:

1. Each document has an associated *similarity accumulator*. Initially each accumulator has a score of zero.
2. Each query word is processed one at a time. For each word, the accumulator values for each document in the corresponding inverted index is increased by the contribution of the word to the similarity of the corresponding document. For example, if the cosine similarity measure is used, then the contribution of keyword $t$ to document $d$ for query $q$ can be computed as

$$contrib(t, d, q) = \frac{w(d, t)w(q, t)}{\sqrt{\sum_{t_i \in d} w^2(d, t_i)}\sqrt{\sum_{t_i \in q} w^2(q, t_i)}}.$$

   Here $w(d, k)$ is the weight of the keyword $k$ in document $d$ and $w(q, k)$ is the weight of $k$ in the query.
3. Once all query words have been processed, the accumulators for documents with respect to the individual words are combined into "global" document scores. For example, if the cosine similarity measure is used as described above, the accumulators are simply added up to obtain the document scores. The set of documents with the largest scores are then returned.

Given a user context that relates each keyword, $t$, to concept, $c$, the contribution equation, $contrib(t, d, q)$, used by the inverted file can be modified as follows to consider the user's context in addition to the query:

$$\frac{\left(\sum_{c_j} w(d, t)w(c_j, t)\right)\left(\sum_{c_j} w(q, t)w(c_j, t)\right)}{\sqrt{\sum_{t_i \in d}\left(\sum_{c_j} w(d, t_i)w(c_j, t_i)\right)^2}\sqrt{\sum_{t_i \in q}\left(\sum_{c_j} w(q, t_i)w(c_j, t_i)\right)^2}}.$$

While this approach can leverage index structures and a query plan specifically designed for this purpose, it will still perform redundant computations for those documents that will not even appear in the top-$k$.

**Naive Ranked Processing.** Both of the above approaches fail to eliminate redundant work. Since the users are often interested in a few most relevant documents, we can achieve significant time gains if we can eliminate those documents that are guaranteed not to have sufficient scores early on.

Such ranked query processing is important in many application domains where results need to be ordered based on a *matching score*, including information retrieval and multimedia, and various top-$k$ join algorithms have been developed to prune unpromising candidates from consideration [9, 10, 12, 16]. These join algorithms generally assume that individual data sources to the top-$k$ join operator are sorted in non-increasing order of scores. Some algorithms also assume that it is possible to perform random access on the individual inputs.

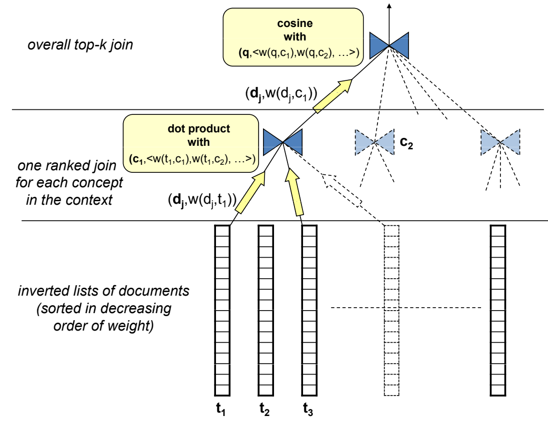



**Figure 3: Top-$k$ implementation based on a ranked join operator requires two layers, where the merge function for the top-layer is cosine**

Given a ranked join algorithm, such as TA [9, 10], the inverted file scheme above can be modified by combining Steps 2 and 3: (a) Instead of processing each query word one at a time, all query words would be considered simultaneously and, for each query word, the documents would be enumerated in the order of their contributions; (b) the stream of documents for each query keyword would then be fed into a sequence of top-$k$ join algorithms which would combine the scores and would stop processing when the best $k$ documents are found (Figure 3):

- *Inverted lists:* As before we use inverted lists, but now these are maintained in the database in sorted order of document-keyword weights.
- *Per-concept ranked joins:* Given a user context with $c$ concepts, we deploy $c$ ranked join operators, one for each concept. The set of inputs to each ranked join operator is the set of keywords that contribute to that concept. Each ranked join operator (TA) combines input streams based on document IDs into document-keyword vectors and for each obtained vector, $\vec{d_j}$, assigns a score based on the *dot product* of $\vec{d_j}$ with the concept-keyword vector, $\vec{c_i}$, corresponding to this join operator. The result is a stream of documents that are ranked in their matching against this concept.

  Note that the basic TA algorithm requires the value of $k$ as input to operate, but at this level we do not know how many documents we need to enumerate for each concept. Thus, the $k$ threshold is constantly updated and the ranked join process continues until the join algorithm receives a stop signal from the top-level join operator, described next.
- *Ranked join against the user query:* The per-concept streams of documents (ordered in decreasing order of weights) are fed into a final ranked join operator (TA) which combines input streams based on document IDs into document-concept vectors and for each obtained

vector, $\vec{d}_j$, assigns a score based on the *cosine similarity* of $\vec{d}_j$ with the query-concept vector, $\vec{q}$. The result is a stream of documents that are ranked in their matching against the query within the user-context.

This approach would save time by eliminating redundant word-document contribution evaluations, redundant joins of documents, and the entire post-sorting step. Unfortunately, however, as discussed in Section 1.3, almost all top-$k$ join algorithms assume that the individual scores are combined using a **monotone** function (e.g *max*, *min*, *product*, and *average*). This, however, is not the case for the *cosine* merge function (Section 1.3). To address this problem, in the next section, we propose a novel ranked query processing algorithm to efficiently compute top-$k$ results with a *cosine*-based scoring function and discuss its use in the user-context aware retrieval problem.

# 3. SKIP-AND-PRUNE: EFFICIENTLY PRUNING UNPROMISING DOCUMENTS USING "SKIP SETS"

In this section, we describe the *skip-and-prune (SnP)* algorithm for efficiently computing top-$k$ query results in the presence of a user context. The general outline of the *SnP* algorithm is similar to that of the ranking based scheme introduced in the previous section. However, since we cannot rely on TA family of algorithms [9, 10] for ranked joins, we develop a novel ranked query processing scheme which can prune its inputs under a cosine-based merge function.

## 3.1 Preliminaries: Sorted Inverted Lists

TA-family of algorithms generally require two access methods: random-access and sorted-access. In our approach, to support random-accesses, we use a table, $D(doc_id, word_id, weight)$, indexed using a $\texttt{B}^+$-tree index on $(doc_id, word_id)$. To support sorted-accesses, for each keyword, $t_j$, we maintain an *inverted* list $\langle d_i, \texttt{D[i,j]}\rangle$, where as described in Section 2.1, $\texttt{D[i,j]}$ is the weight of the keyword, $t_j$, in document $d_i$. This inverted list is maintained in descending order of weights to support sorted access.

The overhead of creating and maintaining these weight-sorted inverted document lists is low since this is an offline process and the sorted list for each keyword can be implemented using a $\texttt{B}^+$-tree index.

## 3.2 Phase 1: Mapping Documents to the User's Concept-Space using Ranked Joins

As described in Section 2.3, we represent the user's search context in the form of a $c \times l$ matrix, $\texttt{U}$, where $c$ is the number of significant concepts in the current context and $l$ is the number of keywords in the dictionary. Given the $m \times l$ document-keyword matrix, $\texttt{D}$, and the $c \times l$ concept-keyword matrix, $\texttt{U}$, we obtain the corresponding $m \times c$ document-concept matrix, $\texttt{D}_\texttt{u}$ (which maps the documents into the user's concept space) as $\texttt{D}_\texttt{u} = \texttt{D}\,\texttt{U}^T$. In other words,

$$\texttt{D}_\texttt{u}[\texttt{i},\texttt{h}] = \sum_{1 \le j \le l} \texttt{D}[\texttt{i},\texttt{j}] \times \texttt{U}^T[\texttt{j},\texttt{h}] = \sum_{\texttt{U}^T[\texttt{j},\texttt{h}] \neq 0} \texttt{D}[\texttt{i},\texttt{j}] \times \texttt{U}^T[\texttt{j},\texttt{h}].$$

Naturally, computing the entire $\texttt{D}_\texttt{u}$ would be extremely expensive and would potentially include significant amount of computations that will eventually prove to be redundant. Instead, we use ranked joins to compute only the necessary parts of the matrix, $\texttt{D}_\texttt{u}$, in a progressive manner. In particular, we assign a ranked join operator for each of the $c$ concepts. The role of each of these per-concept ranked join operators is to independently produce a stream of documents that are sorted in decreasing order of their weights with respect to this concept:

- If we re-consider the above equation, we can see that the value of $\texttt{D}_\texttt{u}[\texttt{i},\texttt{h}]$, for document $d_i$ and concept $c_h$, can be computed by considering only those keywords that contribute to concept $c_h$. Thus, the inputs to the ranked join operation corresponding to concept $c_h$ need to include only streams of sorted documents from keywords, $t_j$, such that $\texttt{U}^T[\texttt{j},\texttt{h}] \neq 0$.
- Given the (multi-)set of non-zero keyword weights $W_h = \{w_{h,j} = \texttt{U}^T[\texttt{j},\texttt{h}] \; s.t. \; \texttt{U}^T[\texttt{j},\texttt{h}] \neq 0\}$ for concept $c_h$, the score $\texttt{D}_\texttt{u}[\texttt{i},\texttt{h}]$ for document $d_i$ can be computed as the weight sum of its keyword weights:

$$\texttt{D}_\texttt{u}[\texttt{i},\texttt{h}] = \sum_{w_{h,j} \in W_h} w_{h,j}\; \texttt{D}[\texttt{i},\texttt{j}].$$

  In other words, the merge function for each of per-concept ranked join operators is *weighted sum*, which is known to be monotonic.

Given these, we slightly modify the TA algorithm [10] to let it produce results in a progressively decreasing order of document-concept scores without requiring a bound, $k$, on the number of results:

1. We maintain a $|W_h|$ dimensional score vector, $\vec{\sigma}$ and a threshold, $\tau$. The score tuple is initialized as $\vec{\sigma} = \langle 1, 1, \ldots, 1\rangle$. Given this, the threshold is initialized as follows:

$$\tau = \sum_{w_{h,j} \in W_h} w_{h,j} \times \vec{\sigma}[j];$$

   i.e., the total weights of the keywords that contribute to the concept $c_h$.
2. The input streams are visited in a round-robin manner as in the original TA algorithm.
3. For each new document, $d_i$, seen in the $j^{th}$ stream, we compute the corresponding score $\texttt{D}_\texttt{u}[\texttt{i},\texttt{h}]$ by accessing the random-access table $D(doc_id, word_id, weight)$ using the key $(doc_id = i, word_id = s)$ for all input streams, $s$, such that $s \neq j$.
4. The combined concept score for $d_i$ is computed using the weights obtained through the random-accesses and the document is inserted into a priority queue based on its concept score.
5. The score vector, $\vec{\sigma}$ is updated such that $\vec{\sigma}[j] = \texttt{D}[\texttt{i},\texttt{j}]$ (i.e., the keyword-weight of document $d_i$ in stream $j$) and $\tau$ is recomputed based on the updated $\vec{\sigma}$.
6. All the documents, in the priority queue, whose scores are above the new threshold $\tau$ are extracted from the priority queue and sent to the next phase **along with** the threshold value, $\tau$.

Note that Phase I is almost the same as the TA algorithm [10], except that it does not require a $k$ as input and, also, it outputs the current threshold along with the documents and their combined concept scores. The value of the threshold will be used to construct an *upper-bound* in the next phase of the process.

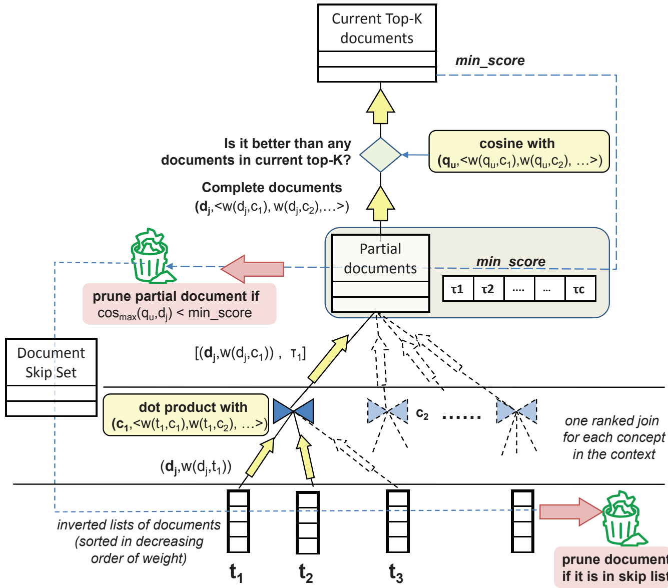



**Figure 4: An overview of the proposed approach: partially available candidate documents can be pruned based on their maximum possible scores.**

## 3.3 Phase 2: Top-K Ranked Join of Document-Concept Streams using a Cosine-based Scoring Function

The output of the $h^{th}$ ranked join operator in the previous phase is a stream of documents ($d_i$s) sorted in the decreasing order of contributions ($\mathtt{D_u[i,h]}$) to the concept $c_h$. In this phase, we combine the various concept contributions of the documents and compare resulting concept-vectors against the user query in the concept space. In other words, one of the inputs to this phase is the query-keyword vector, $\vec{q}$, mapped onto the user-specific context space by multiplying with the concept-keyword matrix, $\mathtt{U}$: i.e., $\vec{q}_u = \mathtt{U}\,\vec{q}$.

Given $\vec{q}_u$ and a complete document vector $\mathtt{D_u[i,-]}$ (i.e., all concept contributions, $\mathtt{D_u[i,1]}, \mathtt{D_u[i,2]}, \ldots, \mathtt{D_u[i,c]}$), we can compute the degree of match between the document, $d_i$, and the user's query as

$$sim_{cos}(\mathtt{D_u[i,-]}, \vec{q}_u) = cos(\mathtt{D_u[i,-]}, \vec{q}_u).$$

However, since the document streams coming from the ranked join operators in Phase 1 are not sorted in document IDs, in most cases, the concept-vector $\mathtt{D_u[i,-]}$ is only partially available. The complete score for document $d_i$ cannot be computed until all concept-scores for $d_i$ from all Phase I streams arrive. Traditional ranked join algorithms deal with this problem in different ways. FA [9] and TA [10] perform random accesses. NRA [10] is able to enumerate top-$k$ join results without having to perform random accesses, however, it requires a monotonic merge function. As we have shown in Section 1.3, however, the cosine similarity described above is not monotonic. Therefore, in order to identify results without waiting for all the concept-scores of all documents from all streams, we need a novel pruning strategy which does not require conventional monotonicity. Figure 4 provides an overview of the proposed approach for pruning unpromising documents based on their maximum possible cosine scores:

- As described in Subsection 3.2, in Phase 1, the rank join operators create a stream of document-concept scores and constantly updated thresholds.
- In Phase 2, we maintain partial vectors of the candidate documents based on the concept-scores fed through Phase I operators. A cut off score, $min_score$, corresponding to the lowest score in the current top-$k$ candidate list, is also maintained.

  Given the user's query, $\vec{q}_u$, and the $min_score$,
  - for any candidate document whose concept-scores are fully available, we can compute the matching score between $d_i$ and $\vec{q}_u$ and update the current list of top-$k$ documents if $d_i$'s score is better than $min_score$, and
  - any document, $d_j$, whose concept-vector, $\mathtt{D_u[j,-]}$, is partially available, is pruned from consideration if the maximum possible score it can eventually have is less than $min_score$.

    Moreover, if $d_j$ is eliminated from consideration in Phase 2, there is no need for the Phase 1 operators to compute the remaining concept scores of $d_j$. Thus, the algorithm maintains a **skip set**, which consists of the IDs of the documents pruned in Phase 2, which can also be eliminated from consideration in Phase 1.

Unlike the TA family of algorithms, the above process cannot end before the sorted inverted files are completely scanned due to "non-monotonicity" of cosine function. In that sense, Phase 2 is similar to the inverted-file based scheme discussed in Section 2.5. On the other hand, the *skip set* mechanism ensures that documents are pruned early from the input streams, thus the total processing needed by the algorithm is much less than that of the inverted-files. In the rest of this section, we develop this pruning strategy.

### 3.3.1 Partially Available Document-Concept Vectors Viewed as Bounded Hyperplanes

Consider $\mathtt{D_u[i,-]}$ representing document $d_i$ in the $c$-dimensional concept space. If all the concept weights of $d_i$ are known, then $\mathtt{D_u[i,-]}$ corresponds to a point (or vector) in this space. If one of the weights is not known, on the other hand, the possible vectors will define a line in the space; if two weights are not known, then the possible vectors will define a plane, and so on. These lines, planes, and so on, however, are not unbounded. Let us consider the case of a line, where a single concept score, say $\mathtt{D_u[i,h]}$, is not known yet. Obviously, $0 \leq \mathtt{D_u[i,h]} \leq 1$. However, the current threshold, $\tau_h$, emitted by the Phase 1 operator corresponding to the concept $c_h$ is nothing but an upper-bound on the scores of any future documents, including $d_i$, that will be output by this operator. Thus, we can see that, $0 \leq \mathtt{D_u[i,h]} \leq \tau_h$. Moreover, as the corresponding operator in Phase 1 identifies new documents and their progressively decreasing concept scores, the upper-bound, $\tau_h$, on the possible values for $\mathtt{D_u[i,h]}$ will also get increasingly tighter.

### 3.3.2 Pruning Partial Vectors based on Maximum Possible Scores

As described above, a document, $d_i$, whose concept scores are partially available defines a hyperplane in the concept space. Given a query vector, $\vec{q}_u$, in the same concept-space, computing maximum possible score $sim_{cos}^{max}(\mathtt{D_u[i,-]}, \vec{q}_u)$ requires measuring the minimum possible angle between the $\vec{q}_u$ and the hyper-plane $\mathtt{D_u[i,-]}$. Although, in theory, it is possible to compute such a minimum angle numerically, this

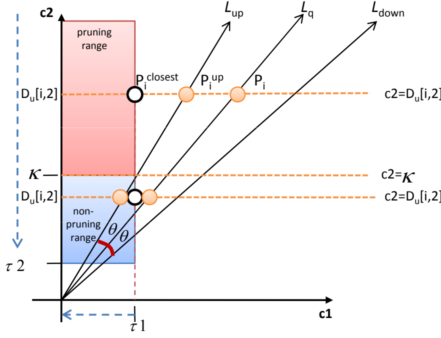



**Figure 5: When $\mathtt{D_u}[\mathtt{i},2]$ is known, but $\mathtt{D_u}[\mathtt{i},1]$ is not, the line segment defined by $\mathtt{D_u}[\mathtt{i},-]$ is horizontal. If any point on this line segment under the current bounds is between $L_{up}$ and $L_q$, the line cannot be pruned.**

would require solving complex equations and would be impractical. Therefore, given a cut off score, *min_score*, we need an efficient mechanism to check whether

$$MAX\{sim_{cos}(\mathtt{D_u}[\mathtt{i},-],\vec{q}_u)\} \leq min_score$$

without explicitly quantifying the minimum possible angle between the query vector and the document hyper-plane.

Example 3.1. *Let us consider the example shown in Figure 5. In this example, the concept-space consists of two concept dimensions: $c_1$ and $c_2$. $L_q$ is a line segment starting from the origin and extending in the direction of the query vector, $\vec{q}_u$. Also, $L_{up}$ and $L_{down}$ are two line segments extending from the origin along the directions specified by $\vec{up}$ and $\vec{down}$, which are two distinct unit-length vectors furthest away from the query vector, but still within the score threshold, $min_score = cos(\theta)$; i.e.,*

$$cos(\vec{up},\vec{q}_u) = cos(\vec{down},\vec{q}_u) = min_score.$$

*In this figure, $\tau_1$ and $\tau_2$ are the current concept score thresholds for $c_1$ and $c_2$, respectively.*

*Let $d_i$ be document. If both $\mathtt{D_u}[\mathtt{i},1]$ and $\mathtt{D_u}[\mathtt{i},2]$ are known, $\mathtt{D_u}[\mathtt{i},-]$ is a vector and $cos(\mathtt{D_u}[\mathtt{i},-],\vec{q}_u)$ can be quickly computed and verified against $min_score$.*

*If neither $\mathtt{D_u}[\mathtt{i},1]$ nor $\mathtt{D_u}[\mathtt{i},2]$ is yet known, $\mathtt{D_u}[\mathtt{i},-]$ is not a candidate yet.*

*If $\mathtt{D_u}[\mathtt{i},1]$ is known, but $\mathtt{D_u}[\mathtt{i},2]$ is not known yet, then $\mathtt{D_u}[\mathtt{i},-]$ defines a vertical line. If $\mathtt{D_u}[\mathtt{i},2]$ is known, but $\mathtt{D_u}[\mathtt{i},1]$ is not, the line defined by $\mathtt{D_u}[\mathtt{i},-]$ is horizontal. Let us consider this latter case, where $\mathtt{D_u}[\mathtt{i},-]$ defines a horizontal line[5]. Since $\mathtt{D_u}[\mathtt{i},2]$ is known, it must have already been output by the corresponding Phase 1 operator, thus it is greater than or equal to the current threshold, $\tau_2$; i.e., $\tau_2 \leq \mathtt{D_u}[\mathtt{i},2] \leq 1$. On the other hand, the unknown value $\mathtt{D_u}[\mathtt{i},1]$ must be less than or equal to $\tau_1$; i.e., $0 \leq \mathtt{D_u}[\mathtt{i},1] \leq \tau_1$.*

*As shown in Figure 5, the threshold $\tau_1$ and the line segment $L_{up}$ split the possible values of $\mathtt{D_u}[\mathtt{i},2]$ into two:* a pruning *range* and a non-pruning *range. Let $\kappa$ be a value such that the vertical line $c_1 = \tau_1$ and the horizontal line $c_2 = \kappa$ intersect on the line $L_{up}$. Clearly, if $\mathtt{D_u}[\mathtt{i},2] > \kappa$, then document $d_i$'s similarity score cannot be above $min_score$ and it can be safely pruned. If $\mathtt{D_u}[\mathtt{i},2] \leq \kappa$, then there is a chance that $d_i$ may have a similarity score above the cut-off, $min_score$, thus cannot be eliminated from consideration.*

In the above example the hyper-planes are simple lines. Therefore, computing angles is cheap. When there are a higher number of missing concept scores, however, the above process may require measuring the minimum possible angle between a line and a hyperplane, which is expensive. Since measuring the distance between points in space is much cheaper, in this paper, we reformulate the above pruning condition in terms of distances between points:

Example 3.2. *Let us reconsider the hyperplane, $c_2 = \mathtt{D_u}[\mathtt{i},2]$, in Figure 5. Let $P_i$ and $P_i^{up}$ be two points obtained by projecting of the vectors, $\vec{q}_u$ and $\vec{up}$, onto this hyperplane. Also, let $P_i^{closest}$ be the closest point to $P_i$ on the line $c_2 = \mathtt{D_u}[\mathtt{i},2]$ within the current bounds ($0 \leq c_1 \leq \tau_1$). If*

$$\Delta(P_i^{closest}, P_i) \geq \Delta(P_i^{up}, P_i),$$

*then the document is in the pruning range; otherwise the document cannot be pruned.*

We leverage this intuition to develop pruning criteria for documents whose concept scores are only partially known:

1. Given a query vector, $\vec{q}_u$, and a partially available document, $\mathtt{D_u}[\mathtt{i},-]$, first we identify a significant-hyperplane, $H_i$, which will lead to the most pruning opportunities.
2. The query vector is projected onto this significant-hyperplane, $H_i$, to identify the corresponding projection point, $P_i$. This projection point represents the query on this hyperplane.
3. The point, $P_i^{closest}$, representing the document on the significant-hyperplane, $H_i$, and closest to $P_i$ under the current bounds, is found next.
4. Then, the point, $P_i^{max}$, on the significant-hyperplane, $H_i$, and representing the current cut-off (based on the current top-$k$ candidates), is identified.
5. Since all the points are on the same hyperplane, $P_i^{closest}$ can be compared to $P_i^{max}$ to decide whether the document can be pruned or not. In particular, the document is pruned if $\Delta(P_i^{closest}, P_i) \geq \Delta(P_i^{max}, P_i)$.

We now explain and describe each of these steps in detail.

#### 3.3.3 Step 1: Construct the Significant-Hyperplane of Document $d_i$

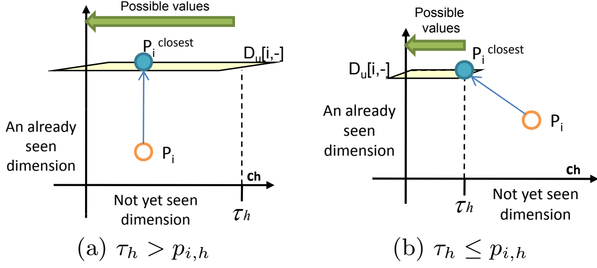
As described in Example 3.2, we first need to define an hyperplane on which the query vector, $\vec{q}_u$, is to be projected. Given the query vector, $\vec{q}_u$, we first identify the index, $h_{sig}$, of the *most significant* concept dimension [6], along which the query vector has the highest weight:

$$\vec{q}_u[h_{sig}] = \underset{1 \leq s \leq c}{MAX}\{\vec{q}_u[s]\}$$

Intuitively, this provides the concept dimension that provides less flexibility for the partially available documents to satisfy and thus provides more opportunities for pruning.

[5]The case for the vertical line is similar.

[6]The concept dimension selected is the one which leads to more pruning opportunities: given that we need to pick a starting dimension, picking the highest valued one increases the chance of the selected dimension having a contribution larger than the contributions of the others.




(a) $\tau_h > p_{i,h}$ (b) $\tau_h \leq p_{i,h}$

**Figure 6: The position of closest point, $P_i^{closest}$, to $P_i$ on $\mathtt{D_u}[\mathtt{i}, -]$ depends on the current bounds along the concept dimensions not yet seen**

Given a document, $d_i$, and a corresponding partially available $\mathtt{D_u}[\mathtt{i}, -]$, we compute

$$w_{i,h_{sig}} = \mathtt{D_u}[\mathtt{i}, \mathtt{h_{sig}}].$$

If $\mathtt{D_u}[\mathtt{i}, \mathtt{h_{sig}}]$ is already known, this is trivial. If $\mathtt{D_u}[\mathtt{i}, \mathtt{h_{sig}}]$ is not known yet, then this requires a random-access to the table table, $D(doc_id, word_id, weight)$ (Section 3.1). The significant-hyperplane, $H_i$, of document $d_i$ is defined by the equation $c_{h_{sig}} = w_{i,h_{sig}}$ in the concept space.

Note that the significant-hyperplane, $H_i$, is much simpler than the document hyperplane defined by $\mathtt{D_u}[\mathtt{i}, -]$. Therefore, finding the angle between a given line and $H_i$ is much cheaper than finding the minimum angle between the same line and $\mathtt{D_u}[\mathtt{i}, -]$.

### 3.3.4 Step 2: For Each Candidate Document, $d_i$, Identify the Projection Point, $P_i$

Once the significant-hyperplane, $H_i$, is identified, next step is to project the query vector, $\vec{q}_u$, onto this hyperplane to find a representative point of the query on the hyperplane. This gives a vector, $\vec{q}_{u,i}$, defined by the projection point, $P_i = (p_{i,1}, p_{i,2}, \ldots, p_{i,c})$ where

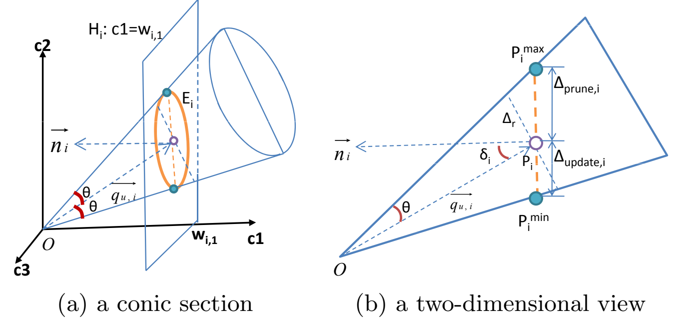
$$p_{i,k} = \begin{cases} w_{i,h_{sig}} & if\ k = h_{sig} \\ \dfrac{w_{i,h_{sig}} \times \vec{q}_u[k]}{\vec{q}_u[h_{sig}]} & otherwise \end{cases}$$

The projection point, $P_i$, and the corresponding vector, $\vec{q}_{u,i}$, are unique for each document $d_i$, and thus are computed only once and stored for future references for each candidate document identified in Phase 2.

### 3.3.5 Step 3: Computing the Minimum Possible Distance between $P_i$ and $\mathtt{D_u}[\mathtt{i}, -]$

As described earlier, $\mathtt{D_u}[\mathtt{i}, -]$ defines a (bounded) hyperplane in the concept space if all of its components are not known yet. To compute the minimum possible distance between $P_i$ and $\mathtt{D_u}[\mathtt{i}, -]$, we first identify the closest point, $P_i^{closest} = (p'_{i,1}, p'_{i,2}, \ldots, p'_{i,c})$, to $P_i$ on the hyper-plane $\mathtt{D_u}[\mathtt{i}, -]$ *within* the current bounds. The distance between $P_i$ and $P_i^{closest}$ gives us the minimum possible distance between $P_i$ and $\mathtt{D_u}[\mathtt{i}, -]$.

Figure 6 shows two cases to illustrate the impact of the dimensions whose weights are not known yet on $P_i^{closest}$. In the first case, the current bound, $\tau_h$, associated with the dimension, $c_h$ (whose value is not known yet), is larger than $p_{i,h}$ of $P_i$. This implies that $p'_{i,h} = p_{i,h}$. On the other hand, if the bound on $c_h$ is less than the $p_{i,h}$, the closest point is obtained when $p'_{i,h} = \tau_h$. Bearing these two cases in mind, $P_i^{closest}$ is computed as follows:




(a) a conic section (b) a two-dimensional view

**Figure 7: (a) The intersection of the significant-plane $H_i$ and the cone defined by $min_score$ is an ellipse and (b) the distance between the point, $P_i^{max}$ ($P_i^{min}$) and $P_i$ can be computed based on the two angles, $\theta$ and $\delta_i$.**

- if $c_h$ is a concept dimension whose score is already known for $d_i$, then

$$p'_{i,h} = \mathtt{D_u}[\mathtt{i}, \mathtt{h}].$$

- else,

$$p'_{i,h} = \begin{cases} p_{i,h} & if\ \tau_h > p_{i,h} \\ \tau_h & otherwise \end{cases}$$

Then, the current minimum distance between $P_i$ and $\mathtt{D_u}[\mathtt{i}, -]$ is equal to $\Delta(P_i^{closest}, P_i)$.

Note that as we get to know more about $d_i$ or as the bounds gets tighter, we need to recompute the position of $P_i^{closest}$. Along the dimensions whose values are known, however, $P_i^{closest}$ stays constant. To speed up the computation, for any dimension, $c_h$, whose value is already known, the value $(p'_{i,h} - p_{i,h})^2$ is computed once and stored for future use.

### 3.3.6 Step 4: Computing the Projection Distance between the Query Vector and the Cut-Off Cone

As shown in Figure 5, in a two dimensional space, the query vector $\vec{q}_u$ and a similarity cut-off $min_score$ defines a triangle. In a multidimensional space, on the other hand, $\vec{q}_u$ and $min_score$ defines a hyper-cone. Figure 7(a) shows an example in a three-dimensional concept space. In this example, $\theta$ is such that $cos(\theta) = min_score$. When $H_i$ cuts this cone, this creates either a closed hyper-ellipse (as shown in Figure 7(a)) or an open hyper-parabola.

Let us first consider the case where the intersection is an hyper-ellipse. Let $E_i$ be the hyper-ellipse of points that are both on the significant-hyperplane $H_i$ and the hyper-cone. Let $P_i^{max} \in E_i$ be a point where $\Delta(P_i^{max}, P_i) = MAX_{P \in E_i}\{\Delta(P, P_i)\}$. In other words, the $P_i^{max}$ is the point of the hyper-ellipse $E_i$ lying on the major axis (Figure 8). The pruning condition in Section 3.3.2 can be relaxed as follows:

$$\Delta(P_i^{closest}, P_i) \geq \Delta(P_i^{max}, P_i).$$

As can be seen in Figure 8, any point that is outside of the hyper-ellipse can be pruned. Points inside the hyper-ellipse, on the other hand, cannot be pruned. The pruning distance, $\Delta_{prune,i} = \Delta(P_i^{max}, P_i)$ can be computed as follows: Let $\Delta_r$ be the radius of the hyper-sphere [7] produced by cutting the

[7] In the case of a three-dimensional space, this is a circle.

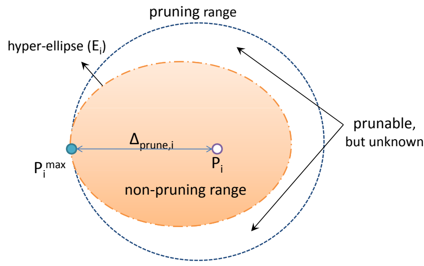



**Figure 8: The pruning condition relies on the *maximum* distance from $P_i$ to the point lying on the hyper-ellipse $E_i$.**

cone by a plane perpendicular to the query vector, $\vec{q}_u$, and through the point, $P_i$. Given the vector $\vec{q}_{u,i}$ defined by the point $P_i$, $\Delta_r$ can be computed as follows (Figure 7(b)):

$$\Delta_r = |\vec{q}_{u,i}| \times \tan(\theta).$$

Let $\delta_i$ be the angle between the query vector, $\vec{q}_u$, and the vector, $\vec{n_i}$, normal to the significant-hyperplane. Then, the configuration in Figure 7-(b) can be used to obtain the following[8]:

$$\Delta_{prune,i} = \Delta(P_i^{max}, P_i) = \Delta_r \times \cos(\delta_i) + \frac{\Delta_r \times \sin(\delta_i)}{\tan(90 - \delta_i - \theta)}.$$

As described earlier, given this value of $\Delta_{prune,i}$ and the previously computed $\Delta(P_i^{closest}, P_i)$, we can prune document $d_i$ from consideration if

$$\Delta(P_i^{closest}, P_i) \geq \Delta_{prune,i}.$$

Note that it is in fact possible that $\Delta_{prune,i}$ as computed above is less than 0. This occurs when the conic section defined by the intersection of the cut-off hyper-cone and the significant-hyperplane $H_i$ is not a closed hyper-ellipse, but an open hyper-parabola. In that case, we do not have sufficient information to support pruning, and thus we set $\Delta_{prune,i}$ to $\infty$.

#### *3.3.7 Step 4: Updating the List of Phase 2 Candidates and the Phase 1 Skip Set*

Once we identify that a document, $d_i$, can be pruned based on its currently known concept scores, the current thresholds, and the $min_score$ value[9], we can prune this document from further consideration. More importantly, however, since $d_i$ is eliminated from further consideration, we also do not need its yet-unknown concept scores anymore. Thus, there is no reason to use Phase 1 resources to compute the remaining concept scores for $d_i$.

We leverage this observation by maintaining a *skip set*, consisting of the IDs of documents that have been eliminated in Phase 2. This continuously updated skip set (implemented as an efficient hash table) is used by the ranked join operators in Phase 1 to filter out eliminated documents from their incoming document-keyword streams. Thus, we can efficiently retrieve the top-$k$ results by avoiding to compute a complete set of concept scores for all the documents in the database. We note that since the skip set only needs to contain document IDs and nothing else, commonly it is small enough to fit into the main memory.

[8]The details of this are omitted.

[9]If there are less than $k$ documents in the current top-$k$ list, $min_score$ is set to 0.

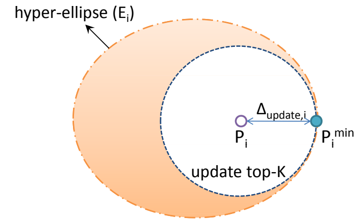



**Figure 9: The condition to update the current top-k list is determined by the *minimum* distance from $P_i$ to the point lying on the hyper-ellipse $E_i$.**

#### *3.3.8 Step 3 Further Refined*

The efficiency of the pruning algorithm described in the previous section depends on how tight the $\Delta_{prune,i}$ values for the candidate documents are. As described in Section 3.3.6, this value depends on the score cut-off value, $min_score$, which corresponds to score of the $k^{th}$ ranking documents for which the complete document score has already been computed. However, since (especially at the initial stages of the process) we do not have enough complete documents, the $min_score$ value will be too low to be effective. One way to ensure that we can prune documents as early as possible is to redefine $min_score$ in such a way that it does not require complete documents but can also reflect scores of the partially seen documents.

Let $d_k$ denote the document in the current top-$k$ list with the lowest score (i.e., $min_score = sim_{cos}(\mathtt{D_u}[\mathtt{k}, -], \vec{q}_u)$). We can remove the $d_k$ from the current top-$k$ list and replace it with $d_i$ if we can be sure that $d_i$ cannot have a lower score than the score of $d_k$:

$$MIN\{sim_{cos}(\mathtt{D_u}[\mathtt{i}, -], \vec{q}_u)\} > min_score.$$

Thus, if we can compute $MIN\{sim_{cos}(\mathtt{D_u}[\mathtt{i}, -], \vec{q}_u)\}$, then we can use this to remove $d_k$ from the current top-$k$ list and update $min_score$ with higher possible score.

Given the hyper-ellipse of points, $E_i$, let $P_i^{min} \in E_i$ be a point where $\Delta(P_i^{min}, P_i) = MIN_{P \in E_i}\{\Delta(P, P_i)\}$ (Figure 7-(b)). Using a reasoning similar to the one described in Subsection 3.3.6, we can state that the above condition holds when

$$\Delta(P_i^{furthest}, P_i) < \Delta(P_i^{min}, P_i),$$

where $P_i^{furthest}$ is the furthest point from $P_i$ on the hyperplane $\mathtt{D_u}[\mathtt{i}, -]$ under the current bounds. As can be seen in Figure 9, if $P_i^{furthest}$ is within the inner circle, it is guaranteed that the possible minimum score of $\mathtt{D_u}[\mathtt{i}, -]$ can not be less than $min_score$. Thus, we can safely replace $d_k$ with $d_i$ in the current top-$k$ list.

Similar to the case described in Section 3.3.6, $P_i^{furthest} = (p''_{i,1}, p''_{i,2}, \ldots, p''_{i,c})$ can be computed as follows:

- if $c_h$ is a concept dimension whose score is already known for $d_i$, then

$$p''_{i,h} = \mathtt{D_u}[\mathtt{i}, \mathtt{h}].$$

- otherwise,

$$p''_{i,h} = \begin{cases} 0 & if\ p_{i,h} > |\tau_h - p_{i,h}| \\ \tau_h & otherwise \end{cases}$$

Once the furthest point, $P_i^{furthest}$, is identified, we can easily compute the left hand side of the above condition.

**Table 1: Synthetic data generation: dense *document-keyword matrices* contain about 50% non-zero entries, while sparse *document-keyword matrices* have 1% non-zero entries.**

| | the size of matrix | |
|---|---|---|
| | dense | sparse |
| *document-keyword matrix* | $0.01M \times 100$ | $0.01M \times 10000$ |
| | $0.1M \times 100$ | $0.1M \times 10000$ |
| | $1M \times 100$ | $1M \times 10000$ |
| *user context matrix* | $100 \times 8$ | $10000 \times 8$ |
| | $100 \times 16$ | $10000 \times 16$ |
| | $100 \times 32$ | $10000 \times 32$ |
| | $100 \times 64$ | $10000 \times 64$ |

The next step is to compute $\Delta(P_i^{min}, P_i)$ using a process similar to the computation of $\Delta(P_i^{max}, P_i)$ described in Subsection 3.3.6. Given the hyper-ellipse of points, $E_i$, that are both on the significant-hyperplane $H_i$ and the hyper-cone, the $P_i^{min}$ is on the major axis of the $E_i$, but on the opposite side of $P_i^{max}$ (Figure 9). Thus, the configuration in Figure 7-(b) can be used to obtain the following[10]:

$$\Delta_{update,i} = \Delta(P_i^{min}, P_i) = \frac{|\vec{q_{u,i}}| \times \sin(\theta)}{|\cos(\theta - \delta_i)|}.$$

Once $\Delta_{update,i}$ is computed, this is compared against $\Delta(P_i^{furthest}, P_i)$ to see if we can be sure that $d_i$ cannot have a lower score than the score of $d_k$. If so, we can remove $d_k$ from the candidate list and we compute the full score of $d_i$ using random access. After this, $d_i$ is inserted into the top-$k$ list and a new *min_score* value is computed based on the new lowest score in the current top-$k$ list.

# 4. EXPERIMENTAL EVALUATIONS

In this section, we describe the experiments we carried out to evaluate the efficiency of the proposed approach. In order to evaluate the proposed approach in a controlled manner, we first systematically generated a large number of data with varying parameters and use these data in our initial experimental evaluation. Secondly, we also used a real data set to verify the practical utility of our approach. In the experiments, in addition to reporting results for the skip-and-prune (*SnP*) algorithm presented in this paper, we also report results for the *scan-and-choose* (SC), *accumulator-based inverted file* (AIF), and *naive ranked processing* (NRP) schemes discussed in Section 2.5. We ran all experiments on a machine with 2.33 GHz of CPU and 2GB of memory and stored relevant tables and indexes in a MySQL DBMS.

## 4.1 Experiments with Synthetic Data

We first evaluated the proposed approach with synthetic data sets. For our experiments, we generated two different types of document-keyword matrix: *dense* and *sparse* matrices. As shown in Table 1, for the dense case, we considered three different matrix sizes: 0.01M-by-100, 0.1M-by-100 and 1M-by-100. These contain about 50% non-zero entries. On the other hand, for the sparse case, we created 0.01M-by-10000, 0.1M-by-10000 and 1M-by-10000 matrices where 1% of entries have non-zero value. The weights for each row vector in these matrices, which corresponds to an individual document, is generated based on a Zipfian distribution.

We considered scenarios with 8, 16, 32 and 64 significant concepts. Each concept is linked to 3-5 keywords in the original dictionary. For the synthetic data set, the user query keyword-vectors are also randomly generated. We run each query three times and the processing times reported in the experiment are the averages of all runs for all queries.

[10]The details of this are omitted.

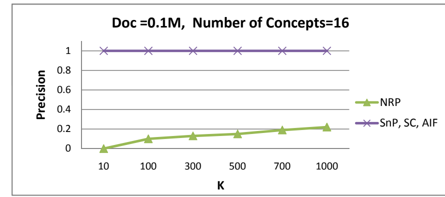



**Figure 10: Average precision on synthetic data**

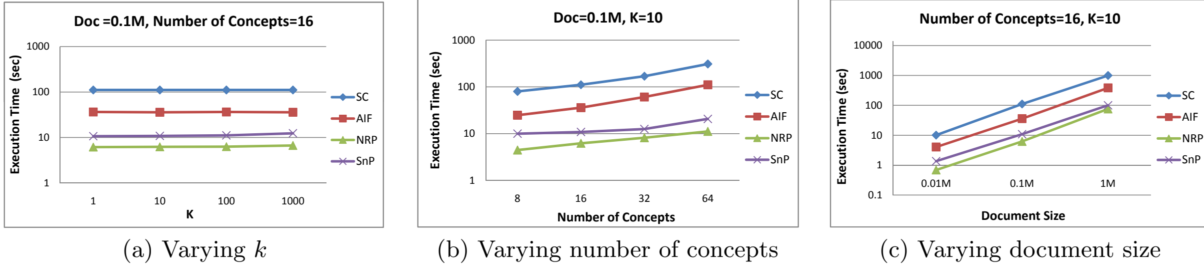
### 4.1.1 Verifying the Need for SnP

Before reporting the execution time results, we first verify the need for the *SnP* approach by comparing the precision obtained by *SnP* against the precision obtained by using a naive ranked-join scheme, which assumes monotonicity. Figure 10 compares the average precision of top-$k$ results by our *SnP* and naive ranked processing (NRP) as a function of $k$ (the number of documents to be returned to users). Here, the size of documents in document-keyword matrix is 0.1M and the concept-space consists of 16 dimensions; results for other cases were similar. As can be seen in this figure, while *SnP* approach proposed in this paper correctly computes the top-$k$ results, a naive ranked processing scheme would result in a very low precision.

### 4.1.2 Execution Times

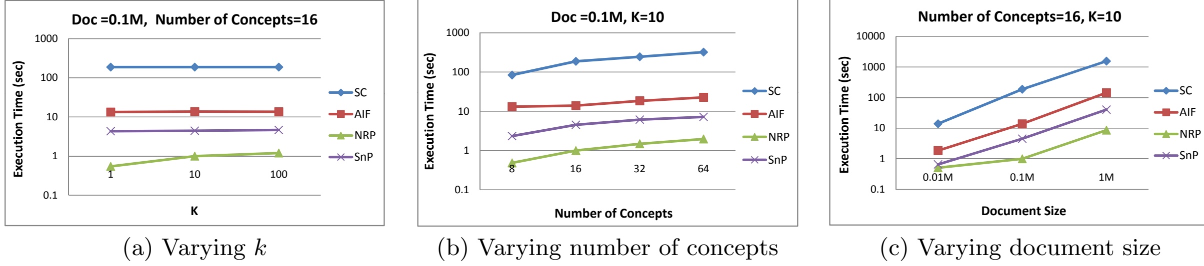
Figures 11 and 12 show the execution times, for varying (a) $k$, (b) number of significant concepts, and (c) the number of documents, for dense- and spare document-keyword matrices respectively. Key observations based on Figures 11 and 12 can be summarized as follows:

- As expected, NRP outperforms other alternatives in execution time. However, as evaluated in Subsection 4.1.1, this scheme performs very poorly in terms of the result quality and thus is not a viable solution.
- Among the three alternatives that can correctly compute the top-$k$ results, *SnP* scheme proposed in this paper shows the best performance. The performance gaps between *SnP* and AIF are larger in dense data sets (representing documents with many distinct keywords). This is because the time gain achieved by skipping unpromising documents is likely to be larger in when there are more keyword streams in Phase 1.
- The effect of $k$ on the execution time of the *SnP* approach is very slight. This shows that the skip-and-prune scheme is highly effective.
- In Figures 11-(b) and 12-(b), the number of significant concepts varies from 8 to 64. For all alternatives, the execution time increases as the number of concepts increases. In the cases of AIF, NRP, and *SnP*, this is because the number of input streams to be scanned and merged at run time depends on the number of concepts. Especially for dense documents, the performance gaps between *SnP* and AIF/SC schemes get larger as the number of concepts increases.
- Figures 11-(c) and 12-(c) compares execution times as the number of documents varies from 0.01M to 1M. In this experiment, we used a concept space consisting of 16 concepts and set the value of $k$ to 10. Once again, as




(a) Varying $k$ (b) Varying number of concepts (c) Varying document size

**Figure 11: Dense document-keyword matrices: the execution times on various parameters.**




(a) Varying $k$ (b) Varying number of concepts (c) Varying document size

**Figure 12: Sparse document-keyword matrices: the execution times on various parameters.**

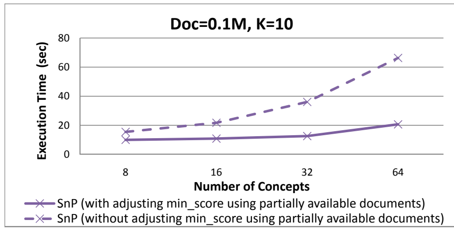



**Figure 13: The impact of adjusting *min_score* by leveraging partially seen documents (Section 3.3.8)**

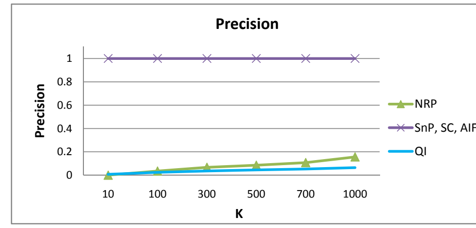



**Figure 14: Average precision on real data sets**

the database gets larger, the performance gap between *SnP* and others increases.

In summary, the *SnP* scheme proposed in this paper is more efficient and scales better with the document sizes and the number of dimensions in the concept space.

### 4.1.3 Impact of Adjusting *min_score* based on Partially Seen Documents

Figure 13 shows that adjusting *min_score* using partially available documents significantly improves the efficiency of top-$k$ query processing. As expected, redefining *min_score* in such a way that it does not require complete documents but can reflect scores of the partially seen documents enables the system to have *min_score* values early and this results in more effective pruning of unpromising documents.

## 4.2 Experiments with Real Data

To evaluate our proposed approach with real data sets, we used the abstracts of papers collected from the ACM digital library [28] and ScienceDirect [29]. This data contains 123,708 documents and 114,611 distinct words, which corresponds to a $\sim 100K \times 100K$ document-keyword matrix where 0.1% of entries have non-zero value.

We considered a scenario in which a user who is reading papers in a specific area is interested in finding more related papers. Thus, the user context is obtained from the papers the user is currently reading: (a) given an abstract, we extracted a set keywords as significant concepts and (b) we created the `U` matrix by considering the co-occurrence relationships of these keywords in the sentences of the abstract.

In the experiments, the number of significant concepts varied between 32, 46, 68, and 94. Also, each result presented here is the average of 30 different random queries.

### 4.2.1 Results and Discussion

Figure 14 compares the average precision of top-$k$ results for *SnP*, SC, NRP, and AIF schemes. In addition, we also evaluate a query-only interpretation scheme, QI, which rewrites the query based on user context, but does not reinterpret the documents in the database. In Section 1.2, we had seen that, while it avoids costly reinterpretation of document vectors in run-time, this scheme is not semantically equivalent to comparing query and the documents in the same user-specific concept space. The figure confirms that QI performs poorly (even worse than NRP); this is because the modification of the query vector alone is not sufficient to capture the distribution of the document vectors in the user specific context space. As in the case of synthetic data sets, the precision of NRP is also poor. This confirms the fact that NRP is not a viable solution, however fast it executes. The proposed *SNP* scheme and the naive approaches SC and AIF are able to obtain perfect precision.

Figure 15 shows the execution times of the different schemes for different numbers of significant concepts. As expected, a naive ranked processing outperforms the other alternatives in execution time. However, as discussed above, NRP results are too poor quality to be useful. Among the three schemes that correctly compute the results, the *SnP* scheme proposed in this paper is always 1 to 2 orders faster than the SC and AIF schemes.

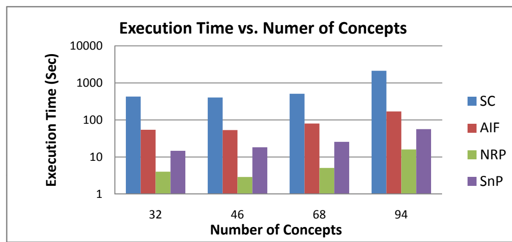



**Figure 15: Execution time with real data**

## 4.3 Skip Set Size

In the experiments, for dense synthetic data, the skip set contained IDs of 80% of the documents in the database (∼800,000 IDs out of 1M). For the real data, which is sparse, the skip set contained about 20% of document IDs.

# 5. RELATED WORK

The most well-known method for top-$k$ queries is TA (threshold algorithm) [9, 10, 12, 16]. TA algorithm assumes that given $M$ sorted-lists (such as $L_1, L_2, \cdots, L_M$), each object has a singe score in each list and an aggregation function, which combines independent object's scores in each list, is monotone. Many variants of TA algorithm have been proposed [2, 3, 6, 7, 15, 25, 26]. An approximate-based algorithm relies on the probabilistic confidence to terminate earlier than the original TA algorithm [2, 25]. *Bansal et al.* [3] introduced an approximation algorithm to aggregate the score of terms in the presence of hierarchies. In their approach, each term score is accumulated into the more commonly used term, which is located at a higher level of the hierarchies, and the probabilistic framework is used for early stopping. With the development of web, there have been studies to determine the ranking of objects based on the score of text data which are related with the specific objects [6, 7]. [6] is similar with our approach in that it maintains the upper- and lower-bound scores of partially seen objects. However, [6] uses these bounds to decide the stopping condition, while in our approach upper- and lower-bounds are used to eliminate unpromising documents.

The *sky-line* operator returns a set of objects that are not dominated by other objects [4]. The existing algorithms to compute results of a sky-line queries can be categorized into block-nested loop [4], divide and conquer [4], spatial index (such as R-tree) [17, 24], and bitmap [24] schemes. While being related to top-$k$ querying, the difference between the sky-line queries and top-$k$ queries is that a there is no explicit scoring function in, while top-$k$ query algorithms described above need a scoring function to be specified.

$k$-nearest neighbor queries [5, 11, 14] are also related with our problem. The most popular method to compute the $k$-nearest neighbors is based on the *branch and bound* method where objects are hierarchically partitioned [11]. As in *SnP*, when computing $k$-nearest neighbor queries, pruning techniques are commonly used to eliminate unpromising objects from further considerations. On the other hand, $k$-nearest neighbor query processing is different from the problem discussed in this paper in that the score function of $k$-nearest neighbor queries is generally based on Euclidean distance, while the retrieval requires a cosine score function being used. Consequently, the pruning technique presented in this paper is different from the pruning schemes of $k$-nearest neighbor queries. The most significant difference is that *SnP* can prune objects even with partially available information.

# 6. CONCLUSION

In this paper, we proposed a novel *skip-and-prune (SnP)* algorithm for efficiently computing the top-$k$ queries in presence of user provided contexts. Our proposed approach enables to prune unpromising documents, despite of the fact that the underlying scoring function is cosine-based, and thus, is not monotonic. Experiment results show that the proposed technique provides orders of magnitude execution time gains over existing solutions. Future work will include extension of the *skip-and-prune* framework presented in this paper to other non-monotonic scoring functions.

# 7. REFERENCES


[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: a system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top-k algorithms. In *VLDB*, 2007.

[3] N. Bansal, S. Guha, and N. Koudas. Ad-hoc aggregations of ranked lists in the presence of hierarchies. In *SIGMOD*, 2008.

[4] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[5] A.J. Broder. Strategies for efficient incremental nearest neighbor search. Pattern Recognition, 23(1):171 - 178, 1990.

[6] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects by exploiting relationships: computing top-K over aggregation. In *SIGMOD*, 2006.

[7] T. Cheng, X. Yan, and K.C.C Chang. EntityRank: searching entities directly and holistically. In *VLDB*, 2007.

[8] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: a semantic search engine for XML. In *VLDB*, 2003.

[9] R. Fagin. Combining fuzzy information from multiple systems. JCSS, 58(1):83-99, 1999.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. JCSS, 66(4):614-656, 2003.

[11] K. Fukunaga and P.M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. IEEE Transactions on Computers, 1975.

[12] U. Guntzer, W. Balke, and W. Kiebling. Optimal Aggregation Algorithms for Middleware. In *VLDB*, 2000.

[13] D. Harman, E.A. Fox, R.A. Baeza-Yates, and W.C. Lee. Inverted files. Information Retrieval: Data Structures and Algorithms. pp. 28-43, 1992.

[14] G.R. Hjaltason, and H. Samet. Distance browsing in spatial databases. TODS, 24(2):265-318, 1999.

[15] I.F. Ilyas, W.G. Aref, and A.K. Elmagarmid. Supporting top-K join queries in relational databases. In *VLDB*, 2003.

[16] S. Nepal and M.V. Ramakrishna. Query processing issues in image(multimedia) databases. In *ICDE*, 1999.

[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. TODS, 30(1):41-82, 2005.

[18] P. Resnik. Semantic similarity in a taxonomy: an information-based measure and its application to problems of ambiguity in natural language. JAIR, 11:95-130, 1999.

[19] S.E Robertson and K.S Jones. Relevance weighting of search terms. JASIS, 27(3):129-146, 1976.

[20] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. CACM, 18(11):613-62, 1975.

[21] G. Salton, E.A. Fox, and H. Wu. Extended boolean information retrieval. CACM, 26(11):1022-1036, 1983.

[22] G. Salton and M.J. McGill. Introduction to modern information retrieval. McGraw-Hill, 1983.

[23] J.T. Sun, H.J. Zeng, H. Liu, Y. Lu, and Z. Chen. CubeSVD: a novel approach to personalized web search. In *WWW*, 2005.

[24] K.Lee Tan, P.K. Eng, and B.C. Ooi. Efficient progressive skyline computation. In *VLDB*, 2001.

[25] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, 2004.

[26] P. Tsaparas, N. Koudas and T. Palpanas. Ranked join indices. In *ICDE*, 2003.

[27] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. TODS, 23(4):453-490, 1998.

[28] ACM Digital Library. http://portal.acm.org.

[29] ScienceDirect. http://www.sciencedirect.com.