Aggregation-Aware Top-k Computation for Full-Text Search

Mingjie Zhu[2*], Shuming Shi[1], Zaiqing Nie[1], Ji-Rong Wen[1]
Microsoft Research Asia[1]
University of Science and Technology of China[2]
{shumings, znie, jrwen}@microsoft.com[1], mingj.zhu@gmail.com[2]

ABSTRACT

A typical scenario in information retrieval and web search is to index a given type of items (e.g., web pages, images) and provide search functionality for them. In such a scenario, the basic units of indexing and retrieval are the same. Extensive study has been done for efficient top-*k* computation in such settings. This paper studies top-*k* processing for many emerging scenarios: efficiently retrieving top-*k* items of one type based on the inverted index of another type of items. It would be very inefficient by directly utilizing traditional top-*k* approaches. Here we follow TA (the Threshold Algorithm) in this scenario. We present an aggregation-aware top-*k* computation framework with three pruning principles upon the conventional inverted index and a novel inverted index type *HybridRank*, which employs the item information of both types. Experimental results show that our proposed new index structure and the aggregation-aware top-*k* strategy provide an efficient solution for this aggregation-aware top-*k* problem.

Keywords

Top-*k*, Aggregation-aware, Aggregation function

1. INTRODUCTION

A typical scenario in information retrieval and web search is to index a given type of items (e.g., web pages, images) and provide search functionality for them (Figure 1). In such scenarios, the basic units of indexing and retrieval are the same. For example, the primary units for indexing and retrieval in general web search are web pages. A lot of work has been done for efficient top-*k* computation in such settings.

However many web applications require top-*k* processing for another scenario where we need to retrieve top-*k* items of one type based on the inverted index of another type of items. For example, consider an academic search engine (Google Scholar [10], Libra [16]) whose functionality is to retrieve top papers and authors with respect to a given query. The natural choice for retrieving top papers is to build an inverted index of all papers and rank papers using current web page ranking techniques. It is somewhat tricky, however, to support searching for top authors (e.g., retrieve top-10 authors given query "information retrieval"). One straightforward approach is to construct author objects (by combining all papers published by one author) and build an inverted index specialized for authors. Since one academic paper is likely written by multiple authors, its contents will be contained in multiple author objects and therefore appear in the author index for multiple times. As a result, the inverted index for authors would be several times bigger (than the paper index) with this approach. Large index will lead to long query processing time and low throughput. If top-*k* authors can be generated efficiently based on the paper index (without building the author index), we may profit from the merit of small index.

Another example in web search is searching for top *sites* (instead of *web pages*) with respect to a query (e.g., retrieve top-10 *sites* related to query "xbox ps3 wii"), based on the inverted index of *web pages*. Indexing sites directly is doable in this case. But building a page-level index gives us the possibility of exploiting the content duplication of web pages to reduce index size. Therefore the choice between page index and site index depends on how efficiently the top-*k* *sites* can be computed based on the *page* index.

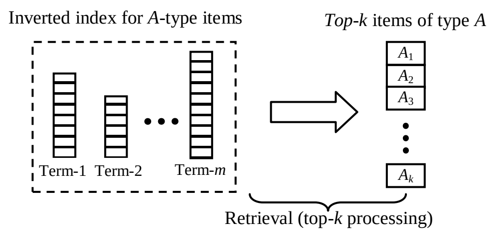

Figure 1. Classical top-*k* processing

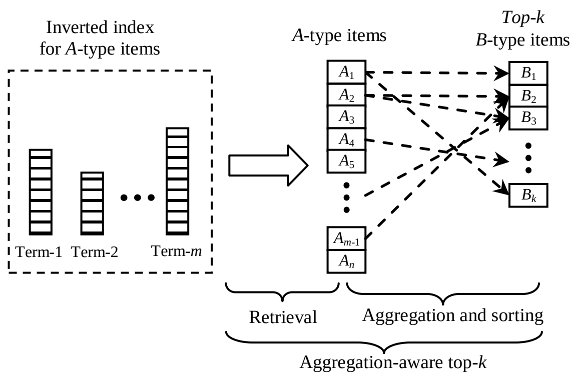

Figure 2. Aggregation-aware top-*k* processing

Both of the above two scenarios require efficiently retrieving top-*k* items of one type (type *B*) based on the inverted index of another type (type *A*), as illustrated in Figure 2. In the first (academic search) example above, papers are *A*-type items and authors are *B* type. While in the second example, web pages are *A* type and sites are *B* type. As only *A*-type items are able to be acquired directly from the inverted index (via inverted list intersection) in answering a query, *B*-type items have to be indirectly constructed on the fly from the information contained in

Technical Report. Microsoft Research. **MSR-TR-2009-47**.

* This work was done when the author was an intern at Microsoft Research Asia.

A-type items. We call such a process *aggregation-aware top-k processing*. There are possibly many-to-many relationships between *A*-type and *B*-type items. That is, one *A*-type item (e.g., A_1 and A_2 in Figure 2) may participate in the construction of multiple items of *B*-type; and one *B*-type item (e.g., B_2 and B_3) may also aggregate the information of multiple *A*-type items. In contrast, classical top-*k* processing (Figure 1) only involves one type of data items and has nothing to do with information aggregation.

Without considering search efficiency, top-*k* items of type-*B* can naturally be retrieved in two stages: Retrieval stage and Aggregation stage. In the Retrieval stage, the inverted lists related to query *Q* are accessed and all relevant *A*-type items are retrieved and scored. In the Aggregation stage, *B*-type items are constructed from these intermediate items (of type *A*) and top-*k* of them are returned. The problem will become much more complicated, however, if efficient top-*k* computation is required. To retrieve top-*k* *B*-type items efficiently, we need to early stop the search process and avoid processing all items of type-*A* related to the query. Existing top-*k* mechanisms designed for single type items couldn't handle this problem well, because generating top-*k* results of type *A* is not sufficient for the needs of constructing top-*k* items of *B* type. If the construction of top-*k* *B*-type items requires top-k_1 items of type-*A*, then generally k_1 will be much larger than *k* in common settings. The value of k_1 depends on index structure, ranking functions being used, and other factors. There may be situations where k_1 is close to the number of all *A*-type items related to the query.

This paper investigates the problem of efficiently retrieving top-*k* items of *B*-type based on the inverted index of *A*-type items. We propose an aggregation-aware top-*k* computation strategy, in which three pruning principles are adopted for reducing the number of items processed and the amount of data loaded from disk. Experimental results show that query processing time can be reduced with our approach based on existing state-of-the-art index structures. In addition, we propose to organize the inverted index of *A*-type items by considering the static rank of *B*-type items in sorting *A*-type items. More performance gain is achieved with such a way of index organization. We also analyze the impact of the aggregation function on performance gain.

We make the following contributions in this paper,

1). Propose an aggregation-aware top-*k* strategy which adopts TA with three pruning principles to efficiently reduce the processing cost;

2). Introduce a novel inverted index structure to further improve the performance of the top-*k* aggregation problem.

The rest of this paper is organized as follows. Section 2 introduces some preliminary knowledge and related work. We describe our approach in Section 3. In Section 4 we present the experimental results. Finally Section 5 is the conclusions and future work.

2. BACKGROUND AND RELATED WORK

In this section, we first give some background concepts related to top-*k* computation. Then some existing work is briefly described.

2.1 Background

Indexing and ranking are two key components of a web search engine. To support efficient retrieval, a web collection needs to be offline indexed. One primary (and most frequently used) way of indexing large scale web documents is inverted index, which organizes document collection information into many *inverted lists*, corresponding to different terms. For a term *t*, its inverted list includes the information (DocIds, occurring positions, etc) of all documents containing the term. Given a query, one ranking function is adopted to compute a relevance score for each document based on the information stored in the inverted index. The documents are then sorted by their scores and the top *k* documents with the highest scores are returned to end users. Although there are often millions of documents which are somewhat related to the query, end users only care about top *k* (e.g., *k*=10, 20, 50, 100) results.

2.2 Related Work

Efficient top-*k* processing for *one data type* has been extensively studied in database and information retrieval communities. In database community, R. Fagin et al. have a series of pioneer work [7, 8, 9] on efficient top-*k* computation. As Fagin [9] demonstrates, if multiple inverted lists are sorted by attributes value and the aggregation function is monotonic, then efficient top-*k* algorithms exist. In information retrieval area, index efficiency related work could go back to 1980s. Some early works are described in [4, 5, 11, 17, 19, 24]. The topic is still active now and some recent works [1, 2, 15, 22, 25, 26] focus on proposing new index structures and pruning strategies for web search, involving in new factors (e.g., page rank, term proximity), etc. In addition to getting exact top-*k* results, approximate or probabilistic top-*k* computation is also important to web search, which is studied in [8, 9, 23]. In all these efforts, the data types for indexing and retrieval are the same. Differently, we examine the case of indexing one data type (type-*A*) while retrieving top-*k* results of another data type (type-*B*).

There is similar work in the database community about dealing with ad-hoc top-*k* aggregation or grouping queries [6, 13, 14]. Li et al. [14] propose to generalize the "*group-by*" operator (in a *SELECT* SQL statement) to enable flexible grouping (or clustering) and address the problem of getting top-*k* tuples within each group (cluster). Li, Chang and Ilyas [13] propose an approach for efficiently processing top-*k* aggregate queries. In other words, they aim at efficiently executing queries like this,

```
SELECT t.g, func(T.score) as gscore  FROM  t
WHERE <conditions>
GROUP BY t.g  ORDER BY gscore  LIMIT k
```

Such a scenario is similar to our *aggregation-aware top-k* processing problem, if we treat *A*-type items as tuples in table *t*, and *B*-type items as groups (yielded by the group-by operator). One big difference is that, the problem in [13] is a traditional SQL problem related to DB-style index structures (e.g., B+ tree). To enable instant web search service for the users all over the world, it's impractical to employ the DB technologies, e.g. building the complex DB-style index, performing *pre-computation* or doing many random accesses to the index. In this paper, we demonstrate how to perform efficient aggregation-aware top-*k* computation for a large scale full-text search engine built on the flat inverted index. The work in [6] addresses the top-*k* aggregation problem by assuming that the full-text search (FTS) module of the database is available and a black-box. Differently, we study how to achieve performance gain by changing the structure of the full-text index.

2.3 Index Structures and Pruning Strategies for Classic Top-K Computation

The following kind of ranking functions are assumed in most papers studying efficient top-*k* computation,

$$F(D,Q) = \alpha \cdot G(D) + \beta \cdot T(D,Q) \\ = \alpha \cdot G(D) + \beta \cdot \sum_{t \in Q} \omega_t \cdot T(D,t) \qquad (2.1)$$

where $F(D,Q)$ represents the overall relevance score of document D to query Q, $G(D)$ is the (query-independent) static rank (e.g., PageRank [3]) of the document, $T(D,Q)$ is the overall term score of D to Q, and $T(D,t)$ is the term-score of D to query term t, computed via a term-weighting function (e.g., BM25 [20]). And α, β and ω_t are parameters satisfying $\alpha + \beta = 1$ and $\sum_{t \in Q} \omega_t = 1$.

The organization of inverted index is the key to index pruning. Efficient top-k strategies are commonly based on specific index structures. Here we briefly summarize the index structures utilized in classical top-k processing.

Globally ordered by static rank: One simple way of organizing inverted lists for improving top-k computation is to have documents sorted in descending order by static rank scores. The intuition here is that, since static rank is an important factor in a ranking function (Formula 2.1), documents with relatively high static rank scores are more likely to appear in the final top results. Therefore, documents with high static rank should appear in top positions of inverted lists.

Multi-segment divided by impact scores: In multi-segment index structure, an inverted list is divided into multiple segments by one specific kind of term impact scores (e.g., TF, IDF, term weighting score, etc). Each segment is sorted by DocId or static rank. As the weight of static rank scores are commonly smaller than that of term weighting scores in real ranking functions, splitting inverted lists by impact scores might be more effective than purely sorting documents by static rank.

3. OUR APPROACH

In this section we formally define the aggregation-aware top-k processing problem and describe our approach to the problem. We discuss the pruning strategies and index structures for aggregation-aware top-k processing, and analyze some key factors affecting the pruning performance.

3.1 Problem Definition

The aggregation-aware top-k computation problem involves two types of interrelated items: A-type and B-type. As illustrated by Figure 2, an inverted index will be built for all items of type A, and the problem is how to efficiently retrieve top-k results of B type given a query. We make more assumptions about the two types of data items and their relationships for convenience of further analysis.

We assume an A-type item has a static rank and a couple of text attributes (or fields). Static rank is a query independent measure indicating how important the item is. For an A-type object a and a query q, we assume the following ranking function for evaluating the score of a with respect to the query, just as most existing IR systems have been doing in the literature,

$$S(a) = \lambda_1 G(a) + (1-\lambda_1) \cdot T(a,q) \\ = \lambda_1 G(a) + (1-\lambda_1) \cdot \sum_{t \in q} w_t \cdot T(a,t) \qquad (3.1)$$

where G(a) is the static rank of a, $T(a, q)$ is the overall term score of a to q, and $T(a,t)$ is the term-score of a with respect to query term t, computed via a term-weighting function (e.g., BM25 [20]). Parameter λ_1 is the static rank weight. And w_t $(t \in q)$ are term score weighting parameters satisfying $\sum_{t \in Q} w_t = 1$.

For each B-type item b, we may also assign or compute a static rank $G(b)$ for it (refer to [18]). An A-type item a is said to be *related to* another B-type item b, if the information of a is needed to construct b at retrieval time. We assume that one A-type item can be related to one or multiple B-type items. And reversely, one B-type item can also be related to one or multiple items of type A. We use $a.CB$ to denote All B-type items related to a, and use $b.CA$ to denote all A-type items related to b.

For a given query, we assume the score of a B-type item is determined by its static rank and the score(s) of all A-type item(s) *related to* it. In this paper, we are interested in the following score expression,

$$S(b) = \lambda_2 \cdot G(b) + (1-\lambda_2) \cdot Agg(b.CA) \qquad (3.2)$$

Agg is an aggregation function for combining the scores of all elements in *b.CA* into an overall score. And λ_2 is the b's static rank weighting parameter.

There are of course other possible expressions for $S(a)$ and $S(b)$. As the expressions listed above are natural and widely utilized in existing information retrieval and web search systems, we follow such expressions in the remaining part of the paper. The score aggregation function *Agg* can vary from application to application. As we will see in the following sections, the *Agg* function has great impact on the performance of top-k processing approaches.

3.2 Pruning Principle and Strategy

In this section we first introduce the main operations in aggregation-aware top-k processing and then present our pruning strategies for improving performance. Specifically, we propose three principles to guide the pruning.

3.2.1 Time-consuming operations

For this problem, the query processing performance is mainly determined by the cost of three operations. The first (referred to as COST-1) is loading the inverted lists corresponding to the query from disk to memory and doing intersection among the inverted lists. The second one (COST-2) is computing scores for A-type items relevant to the query. And the third one (COST-3) is computing the scores of B-type items. The key of designing an efficient top-k processing approach to this problem is therefore to reduce the cost of the three kinds of operations.

3.2.2 Item states and pruning principles

Our top-k strategy keeps track of the *states* of all B-type items in processing a query. One B-type item can be in one of the following four states,

Not-evaluated For an item b in this state, none of the A-type items related to it (i.e. items in $b.CA$) has been processed.

Top-*k*-member The item is in the top-k list *at the moment*.

Top-*k*-candidate This item is not in current top-k, but still has the possibility of being in top-k in the future (along with more A-type items being processed).

Discarded This item is impossible to be in top-k from the information currently available. Items in this state will be skipped without further evaluation.

In processing a query, a B-type item's state could be updated from one state to another. Based on the item states defined above, three pruning principles are utilized for reducing the cost of the three operations mentioned in section 3.2.1.

Some notations are defined here before the pruning principles are described. Let *R* and *C* denote the list of all *B*-type items with state *Top-k-member* and *Top-k-candidate*, respectively. For some items in *R* or *C*, we may not be able to compute their exact scores without accessing or scoring all *A*-type items related to it. Instead, we can have an estimation of the possible minimal and maximal score of the item according to current information we have. Let *b*.*min_score* and *b*.*max_score* denote the estimated minimal and maximal score of *b*. Let R_i be the item in *R* with the *i*'th largest (estimated) minimal score and its minimal and maximal score be $S_{i,min}$, and $S_{i,max}$ respectively.

Pruning principle-1 (the *A*-skipping principle): An item *a* of type *A* can be skipped without being scored if,

$$\forall b \in a.CB,\ b.state = Discarded \qquad \blacksquare$$

This principle means that, if we know all the *B*-type items related to *a* have no chance to be in top-*k* beforehand, we can skip this item without evaluating it (by a ranking function). This principle is for reducing the number of *A*-type items been evaluated (to save COST-2).

Pruning principle-2 (the *B*-discarding principle): For a *B*-type item *b*, its state can be set to *Discarded* (therefore it can be removed from set *C* if it has been a member of it) if,

$$|R| \geq k \ \textbf{ and } \ b.max\_score \leq S_{k,min} \qquad \blacksquare$$

Intuitively, for an item *b*, if there are at least *k* items whose estimated minimal scores are greater than *b*'s maximal possible score, then *b* is not going to be in top-*k*. Therefore, we can set *b*'s state to be *Discarded* and remove it from the top-*k* candidate set *C*. By this principle, we can determine that one *B*-type item will not be in the final top-*k* list even if we have not got its exact score (to save COST-3).

Pruning principle-3 (the early-stopping principle): The search process can stop if the following conditions are satisfied.

C3-1). $S_{k,min} \geq S_{upp}$

C3-2). $\forall b \in C,\ S_{k,min} \geq b.max\_score$

C3-3). $\forall\ 1 \leq i < k,\ S_{i,min} \geq S_{i+1,max}$

where S_{upp} is the score upper bound of all un-processed items of type *B*. ■

The first condition indicates that all un-processed *B*-type items are not going to be in the final top-*k* results (because their score upper bounds do not exceed $S_{k,min}$). The satisfaction of this condition allows us to seek for the final top-*k* items only in *R* and *C*. The second condition says that all *B*-type items with state *Top-k-candidate* will not be in top-*k* (because the maximal possible score of each item does not exceed $S_{k,min}$). If condition C3-1 and C3-2 are both satisfied, *R* is guaranteed to contain the final top-*k* items, but the relative order between the items are not insured (please keep in mind that items in *R* are now sorted by their estimated minimal possible scores instead of real scores). The final condition implies that the relative order among items in *R* can be determined.

All the above pruning principles are guaranteed to return the exact same results by TA adopted in our top-*k* strategy in the next subsection. The proof for the correctness of TA can be found in Fagin's related work [8].

3.2.3 Our top-k strategy

To apply TA here, we utilized the above three pruning principles and illustrate our top-*k* computation strategy in Figure 3. Two data structures are maintained: *R* (current top-*k* list of *B*-type items) and *C* (the set of all *B*-type items that are possibly in the final top-*k*). Elements in *R* are sorted in descending order by their minimal possible scores. *R* and *C* are initialized to be empty, and the initial states of all *B*-type items are set to be *Not-evaluated*. The algorithm sequentially accesses the inverted index and retrieves relevant *A*-type items one by one. For each item *a* retrieved, we first check with the A-skipping principle (pruning principle-1). If all corresponding *B*-type items have been in *Discarded* state, we skip it and retrieve the next *A*-type item. Otherwise, we send it to a ranking function and update its minimal and maximal possible scores. For every *B*-type item related to *a*, if it is not in *Discarded* state, its score range (minimal to maximal possible scores) is re-computed and updated with the information of *a*. The score change of a *B*-type item may result in a new top-*k* list and/or the update of the candidate set, which is handled by a routine called *OnBScoreUpdated* (refer to Figure 4).

```
Algorithm:  Aggregation-aware top-k computation
    Input: Query q
    Output: Top-k items of B-type
    Background data: The inverted index of A-type items
    Variables:
        R: Current Top-k list of B-type items (with $R_i$ the i'th item)
        C: The candidate set (the set of B-type items which have the
possibility of being in top-k)
        $S_{k,min}$, $S_{k,max}$: The estimated minimal and maximal score of $R_i$.
    Clear R and C; Set $S_{k,min} = S_{k,max} = 0$
    Set the states of all B-type items to be Not-evaluated
    for each A-type item a retrieved from the index
        if all_have_discarded_states(a.CB)
            Continue;
        end-if
        Evaluate a    //Update a.min_score and a.max_score
        for each item b in a.CB
            if (b.state != Discarded)
                Update b.min_score and b.max_score according to a
                OnBScoreUpdated(R, C, b);
            end-if
        end-for
        if (it is time to check batchly)
            for each item e in C
                Update e.min_score and e.max_score
            end-if
            //check with the early-stop principle
            bCanStop = TryEarlyStop(R, C)
            if(bCanStop)
                return R;
            end-if
        end-if
    end-for
    return R.
```

Figure 3. Strategy for *aggregation-aware top-k* computation

Routine *OnBScoreUpdated* is invoked every time the score boundaries a *B*-type item *b* being updated. It does corresponding actions according to its scores and states. If *b* has already been in current top-*k* list, its position in *R* may change (from the *k*'th position to the first one, for example) after its minimal score is updated. If *b* is not in current top-*k* list but its minimal score exceeds the minimal score of the *k*'th element in *R*, it will be added into the top-k list or replace the k'th element in R if $|R| = k$. Finally, item b might be discarded if the conditions of the B-discarding principle(pruning principle-2) are satisfied.

```
//Parameters:
//    R: Top-k items at the moment; C: The candidate set
Routine bool TryEarlyStop(ref R, ref C)
    for each item b in C
        Update b.max_score
        OnBScoreUpdated(R, C, b);
    end-for
    bSatisfied = Test using the first condition of Pruning Principle-3
    if (bSatisfied)
        bSatisfied = Test the second condition of Pruning Principle-3
    end-if
    if (bSatisfied)
        bSatisfied = Test the third condition of Pruning Principle-3
    end-if
    return bSatisfied
end-routine //TryEarlyStop

//Parameters:
//    R: Top-k items at the moment; C: The candidate set; b: an B-type
//item whose minimal score or maximal score has just been updated
Routine void OnBScoreUpdated(ref R, ref C, ref b)
    if ($b \in R$)
        Adjust the position of b in R;
    else
        if ($|R| < k$)
            b.state = Top-k-member;
            $R = R + b$;
        else if($b.min\_score > S_{k,min}$)
            b.state = Top-k-member;
            $R_i$.state = Top-k-candidate;
            $C = C + R_i$;
            $R = R - R_i + b$;
        else if ($b \notin C$ and $b.max\_score \geq S_{k,min}$)
            b.state = Top-k-candidate;
            $C = C + b$;
        else if ($b \in C$ and $b.max\_score \leq S_{k,min}$)
            //applying the skipping principle
            b.state = Discarded;
            $C = C - b$;  //remove b from C
        end-if
    end-if
end-routine //OnBScoreUpdated
```

Figure 4. Routine TryEarlyStop and OnBScoreUpdated

The early-stopping principle(pruning principle-3) is crucial for reducing processing cost. The principle is checked by routine *TryEarlyStop* (refer to Figure 4). This routine first update the maximal possible scores of all items in C (according to the new information acquired), and then check the conditions one by one. It returns *true* if all the conditions are satisfied. The routine is time-consuming, because all items in R and C have to be checked and updated. Therefore we choose to invoke the routine after a certain number of A-type items have been retrieved and processed.

From the three pruning principle and the above algorithm, we can see that the key to achieve efficient pruning is to make a good estimation of S_{upp} (the score upper bound of all un-evaluated B-type items), and to properly estimate the minimal and maximal possible scores of a partially-processed item b. Inverted index structure should be involved in making all these estimations, because different index structure may have different capability of supporting good estimation-making. In the next section, we will illustrate some candidate index structures, based on which "good" estimations might be made and therefore efficient top-k processing is probably achieved.

3.3 Index Structures Based on HybridRank

One top-k strategy relies on properly organized index structure to achieve good performance. We start with the two state-of-the-art index structures (for classical top-k processing) mentioned in Section 2.3. If the two index structures are directly utilized in our aggregation-aware top-k processing problem, all A-type items in each segment will be sorted in descending order of their static rank. For further efficiency improvement, we propose to consider the static rank information of B-type items in ordering A-type items in the index. We define *HybridRank* as a combination between A-type item's static rank and the static rank of B-type items *related to* the A-type item. Then we sort A-type items by HybridRank in each index segment.

3.3.1 HybridRank

For an A-type item a, let $b(a)$ denote the item which has the largest static rank in all the B-type items *related to a*. We assign a HybridRank $H(a)$ to item a as follows,

$$H(a) = \max(w_1 \cdot G(a), w_2 \cdot G(b(a))) \tag{3.3}$$

In other words, the HybridRank $H(a)$ is defined as the weighted maximum of a's static rank and the static rank values of all its corresponding B-type items. When $w_1 = 1$ and $w_2=0$, $H(a)$ represents a's static rank. Therefore, sorting items by A-type static rank is a special case of sorting items by HybridRank.

3.3.2 Organize A-type items in index by HybridRank

Two index structures exploited in our approach are shown in Figure 5. For the first index structure (Figure 5(a)), all (A-type) items in each inverted list are sorted by HybridRank. Items are split into two segments in the second index structure, where items with term weighting scores higher than a threshold are placed in the high-impact segment, while other items are placed in the low-impact segment. Items are sorted by HybridRank in each segment.

The basic idea of organizing items by HybridRank is to make a "good" estimation of the score upper bounds of all unseen items in query processing. According to the definition, the HybridRank of item a is the (weighted) maximum of a's static rank and the static rank values of all its corresponding B-type items. Therefore when we scan the inverted lists with respect to a query, the static rank upper bounds of not only unseen A-type items but also their corresponding B-type items will decrease. According to Formula 3.1 and 3.2, it is intuitive that reducing both $G(b)$ and $G(a)$ helps to reduce score upper bound S_{upp} more efficiently than solely reducing $G(b)$ or $G(a)$.

Let us illustrate how to estimate score bounds for index structures based on HybridRank. Take, for example, the estimation of S_{upp} (see Pruning Principle -3) for the index structure in Figure 5(a).

Let H_t be the HybridRank of the last accessed A-type item. Because items in all inverted lists are ordered in descending order by HybridRank, the HybridRank of each remaining item will not be greater than H_t. That is, for each remaining A-type item a and any B-type item b_i *related to* it, we have,

$$\begin{aligned} w_1 \cdot G(a) &\leq H_t \\ w_2 \cdot G(b_i) &\leq H_t \end{aligned} \tag{3.4}$$

The maximal possible score of all subsequent A-type items can be estimated by the following formula,

$$max_a = \lambda_1 \cdot \min(\frac{H_t}{w_1}, 1.0) + (1 - \lambda_1) \cdot 1.0 \quad (3.5)$$

Here the maximal possible value of $T(a, q)$ is estimated as 1.0 because no more information is available for a better estimation for this kind of index structure.

And the maximal possible score of all unseen B-type items are,

$$S_{upp} = \lambda_2 \min(\frac{H_t}{w_2}, 1.0) + (1-\lambda_2) Agg(U_m(max_a)) \quad (3.6)$$

where m is the maximal number A-type items a B-type item can possibly be *related to,* and $U_m(s)$ denote the m-dimension vector with all elements having uniform value s.

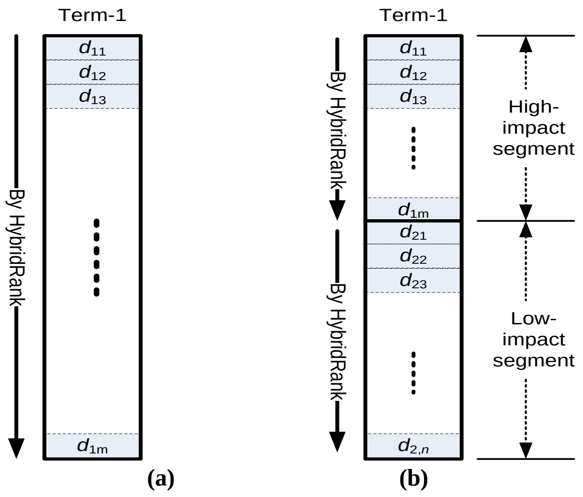

Figure 5. Index structures based on HybridRank: (a). One segment with all documents ordered by HybridRank; (b). Two segments divided by term-impact scores

3.4 Aggregation Functions

Given one specific index structure, the effectiveness of our top-k strategy may depend on the aggregation function adopted for combining A-type items into a B-type item (refer to Formula 3.2).

Consider, for example, the estimation of S_{upp} using Formula 3.6. Suppose we are lucky to get a small H_t/w_2 (say 0.3) and a small max_a (say 0.1). Also suppose λ_2=0.5 and m=100. Given these assumptions, S_{upp} is estimated as,

$$S_{upp} = 0.5 \cdot 0.3 + (1 - 0.5) \cdot \mathrm{Agg}(U_{100}(0.1)) \quad (3.7)$$

Let's examine the following aggregation functions:

SUM: $\mathrm{Agg}(S) = \sum_1^{|S|} s_i$; MAX: $\mathrm{Agg}(S) = \max_1^{|S|} s_i$

If SUM is adopted as our aggregation function, we will have S_{upp}=5.15. While if MAX is adopted, the estimation value of S_{upp} will be 0.2, a small enough value for condition C3-1 to be satisfied. SUM is a typical aggregation function which yields very high score upper bounds, while MAX is another extreme case of generating very small S_{upp}. Many real-case aggregation functions may be in between them. We are interested in the *long-tail-favoring* property of an aggregation function, i.e., whether the aggregation of a larger number of tiny scores could yield a high aggregated score. SUM is a long-tail-favoring aggregation function, while MAX is not.

Shi et al. [21] proposed and studied a spectrum of score aggregation formulas with SUM and MAX as its two ends. For a score vector S=(S_1, …, S_n) where scores are ordered in descending order, their aggregated score is computed as,

$$Hsc(S) = \sum_{i=1}^{n-1} \frac{(h+1) \cdot i}{h+i} \cdot (S_i - S_{i+1}) \quad (3.8)$$

It is easy to verify that MAX and SUM are two special cases of the *Hsc* function (when parameter h=0 and h=+∞, respectively). Parameter h is originally K in [21]. We use h here simply for reducing ambiguity, because K (or k) has been used to represent the number of results retuned to users. Generally, the upper bound of the aggregated score rises as h increases. Intuitively, smaller h helps to lower down the upper bound estimation and makes pruning easier. We can generate different aggregation functions by varying parameter h. In the following section, we will give the related experimental results.

4. EXPERIMENTS

In this section, we evaluate our work by conducting experiments in the academic search scenario, based on our academic search system (references omitted due to blind review). In such a scenario, papers and authors are respectively A-type and B-type items. In other words, we are required to retrieve top-k authors, based on an inverted index of papers. We would like to add experiments in other scenario(s) in the future work.

4.1 Experimental Setup

Dataset and query set: We crawled about 0.6 million papers in computer science domain. The papers are indexed according to the index structures described in Section 3.3, with different values of w_1 and w_2 (including the special case of w_1=1 and w_2=0). The static rank of all papers and authors are calculated by the methodology in [18]. In retrieval time, the relevance score of a paper and an author are computed via Formula 3.1 and 3.2 respectively, where the term weighting score $T(a,t)$ is calculated by BM25 [20]. The *Hsc* function series illustrated in Section 3.4 are utilized as our aggregation functions. From the most popular queries in our academic search system's log, we remove the queries not fit in our scenario (e.g. author names, one particular paper's full title) and select the representative 68 queries like "content based image retrieval" as the testing query set.

Approaches for comparison: We compare our approach with the baseline method as:

Baseline The Baseline method makes a *complete* scan of the inverted lists corresponding to each query to get *all* relevant A-type items, and then computes top-k items of B-type by traversing all the processed A-type items. No efficient top-k computation techniques are adopted in the baseline approach. Different index structures do not affect the performance of the baseline approach too much.

Parameters: three key parameters are critical for the search resuls quality: λ_1 (the static rank weight of A-type items in Formula 3.1), λ_2 (the static rank weight of B-type items in Formula 3.2), and h (*aggregation factor*, the parameter for controlling our aggregation function, see Section 3.4). We set $\lambda_1 = = \lambda_2 = 0.4$ from our empirical study. In practice we usually set the h value in from 1 to 10 to get the best results. However, without loss of generality, we compare more h values in experiments: 0, 0.5, 1, 2, 4, 10, 20, +∞.

Evaluation Metrics: The search result quality is beyond the domain of this paper. For different groups of ranking parameters setting, the search result may be different. While for one certain group of parameters, different top-k algorithms are guaranteed to return the identical top-k list of B-type items as the non-top-k baseline according the TA basis. The primary metric used in our experiments is the average query processing time over all the queries in our query set.

Hardware and software environment: All experiments were carried on one single workstation with Dual 2.4GHz AMD Opteron Processor 250, 8GB RAM and 800GB local SATA Disk, running Windows Server 2003 Enterprise x64 Edition. And the programs are written and compiled via Visual C++ 2005.

4.2 Experimental Results

4.2.1 Overall performance comparison

Figure 6 shows the performance comparison between the baseline approach and our approach, respectively based on two basic index structures: *one-seg* and *two-seg*, with *A*-type items sorted by HybridRank. We can observe from the figure that, for both index structures, our approach performs better than the baseline. More performance gain is achieved on the *two-seg* index structure for most aggregation factors. Especially for small aggregation factor values, about 70% query processing time can be saved.

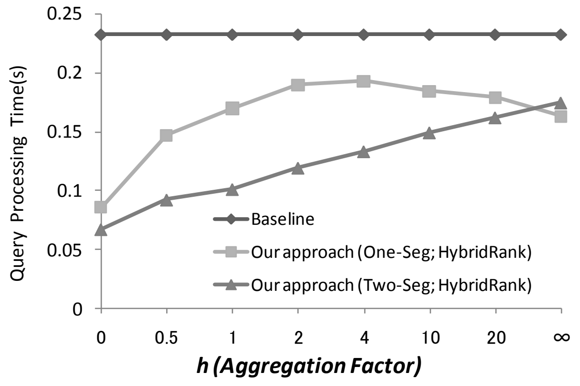

Figure 6. Performance comparison between the baseline and our approach, for various aggregation functions ($\lambda_1 = \lambda_2$ =0.4, Top-5 results)

We study the relationship between the aggregation function and pruning performance, by adopting different h values of the above *Hsc* function. We do not intend to discuss which aggregation function is best given a search scenario, but focus on studying the relationship between different types of aggregation functions and their corresponding optimal top-k computation strategies.

More performance gain is typically achieved for small h values as the bound is tighter according to formula 3.8. While for the *one-seg* index structure, query processing time also decreases with very large h values. Such observations can be partially explained by condition C3-1 in Pruning Principle-3. This condition says that $S_{k,min} \geq S_{upp}$ should be satisfied to achieve early stop. On one hand, according to Formula 3.6 and 3.8, large h value leads to large estimation of S_{upp}. On the other hand, the value of $S_{k,min}$ would also slightly increase when the value of h gets high. For the *two-seg* index structure, since the value $S_{k,min}$ has already been large after the high-impact segment is processed, parameter h has relatively small impact on $S_{k,min}$. As a result, when h gets larger, query processing time increases accordingly due to bigger S_{upp}. For the *one-seg* index structure, however, parameter h has high impact on both $S_{k,min}$ and S_{upp}. The relationship between query processing time and h is therefore a little bit complex. We would like to leave it for future work.

4.2.2 Comparison among different index structures

In this subsection, we study the performance of our approach over different index structures. Figure 7 shows the performance comparison among different item ordering schemes over the *one-seg* index structure. One observation is that HybridRank performs better than A.Rank and B.Rank. The B.Rank scheme only has slight performance improvement over the baseline. Please pay attention that adopting A.Rank even performs worse than the baseline for some h values. This means the top-k algorithm doesn't' work for the inverted index sorted by item A's static rank, and the cost of the extra top-k computation makes the performance even worse than the naïve baseline. Results on *two-seg* index structure are shown in Figure 8. All top-k algorithms benefit from the *two-seg* index structure while HybridRank scheme has the best performance.

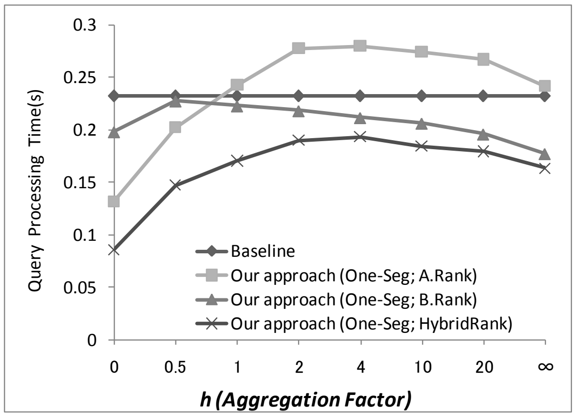

Figure 7. Performance comparison among different item ordering schemes (Index structure: *one-seg*, $\lambda_1 = \lambda_2$ =0.4, Top-5 results)

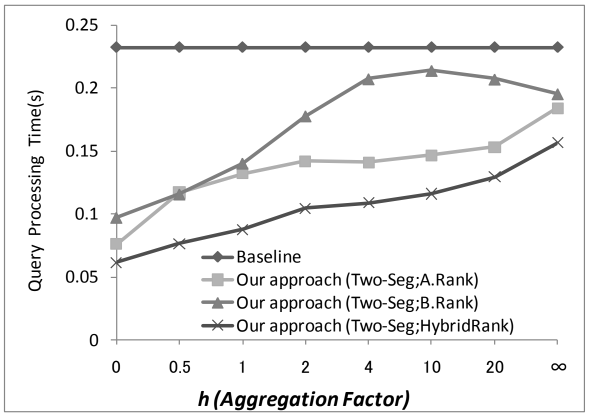

Figure 8. Performance comparison among different item ordering schemes (Index structure: *two-seg*, $\lambda_1 = \lambda_2$ =0.4, Top-5 results)

HybridRank has better performance, as demonstrated in previous experimental results. While the A.Rank ordering scheme has more flexibility because it keeps the original index structure for retrieving A-type items in classical top-k processing (see Figure 1). As a result, the same index structure can be shared for retrieving both A-type and B-type items. In addition, more item types (type-

C, type-D, etc.) can also be retrieved by re-using the same index structure. Due to the specific application requirements, it may be interesting to further compare HybridRank with the A.Rank ordering scheme with different values of λ_1 and λ_2. Table 1 shows the comparison between HybridRank and A.Rank over the *two-seg* index structure, where "+", "0", and "-" respectively mean that the performance of HybridRank is better than, equal to, or worse than that of the A.Rank ordering scheme. From Table 1, we can see that in most cases HybridRank outperforms A.Rank ordering scheme.

Table 1. Winning area for A.Rank and HybridRank ordering over the *two-seg* structure with different λ_1, λ_2 values (h = 2)

λ_1 \ λ_2	0	0.25	0.5	0.75	1
0	+	+	+	+	+
0.25	+	+	+	+	+
0.5	+	+	+	+	+
0.75	0	0	-	-	+
1	-	-	-	-	+

5. CONCLUSIONS AND FUTURE WORK

We studied the problem of retrieving data items of one type based on the inverted index of another data type. Adopting the TA algorithm for this problem, we propose three pruning principles and an efficient top-*k* computation strategy. For most index structures examined in this paper, the strategy can have apparent performance gains over the baseline approach. Ordering *A*-type items by HybridRank can help achieve better performance than other ordering schemes.

HybridRank has two parameters w_1 and w_2. It will be interesting to study which parameter values result in the best performance for a given scenario. The conclusion may depend on the static rank distributions of *A*-type and *B*-type items, the value of λ_1 and λ_2, the properties of the aggregation function, and other factors. We plan to leave this for future study. We would like to try more types of aggregation functions in the future. Building HybridRank index doesn't require too much additional efforts according to our implementation experience. However, a detailed study on the building and updating of the HybridRank index will be addressed in the future work. We also plan to make further efforts to apply our proposed framework in more scenarios.

6. REFERENCES

[1] V. N. Anh and A. Moffat. Pruned Query Evaluation Using Pre-Computed Impacts. Proc. 29th Annual International ACM SIGIR, 2006.

[2] V. N. Anh and A. Moffat. Pruning Strategies for Mixed-Mode Querying. In Proc. of 2006 CIKM Int. Conf. on Information and Knowledge Management (CIKM'06), 2006.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In Proc. of the Seventh World Wide Web Conference (WWW'98), 1998.

[4] C. Buckley and A. Lewit. Optimization of inverted vector searches. Proc. of 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'85), 1985.

[5] M. Burrows. Method for Parsing, Indexing and Searching World-Wide-Web Pages. US Patent 5,864,863, 1999.

[6] K. Chakrabarti, V. Ganti, J. Han, D. Xin. Ranking Objects by Exploiting Relationships: Computing Top-K over Aggregation. In SIGMOD 2006.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In Proc. of the Twentieth ACM Symposium on Principles of Database Systems, 2001.

[8] R. Fagin. Combining Fuzzy Information from Multiple Systems. In Proc. of the Fifteenth ACM Symposium on Principles of Database Systems, 1996.

[9] R. Fagin. Combining Fuzzy Information: an Overview. SIGMOD Record, 31(2): 109-118, June 2002.

[10] Google Scholar: http://scholar.google.com/

[11] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. Journal of the American Society for Information Science, 41(8): 581-589, August 1990.

[12] R. Kaushik, R. Krishnamurthy, J. Naughton and R. Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. SIGMOD 2004.

[13] C. Li, K. C.-C. Chang, and I. F. Ilyas. Supporting Ad-hoc Ranking Aggregates. In SIGMOD 2006.

[14] C. Li, M. Wang, L. Lim, H. Wang, and K. C.-C. Chang. Supporting Ranking and Clustering as Generalized Order-By and Group-By. In SIGMOD 2007.

[15] X. Long and T. Suel. Optimized Query Execution in Large Search Engines with Global Page Ordering. In VLDB 2003.

[16] Libra Academic Search: http://libra.msra.cn/

[17] A. Moffat and J. Zobel. Self-Indexing Inverted Files for Fast Text Retrieval. TOIS 14(4), 1996.

[18] Z. Nie, Y. Zhang, J.-R. Wen, and W.-Y. Ma. Object-Level Ranking: Bring Order to Web Objects. In WWW, 2005.

[19] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered Document Retrieval with Frequency-Sorted Indexes. Journal of the American Scociety for Information Science, 47(10): 749-764, May 1996.

[20] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In Proceedings of TREC-3, November 1994.

[21] S. Shi, R. Song, and J.R. Wen. Latent Additivity: Combining Homogeneous Evidence. Technique report, MSR-TR-2006-110, Microsoft Research, August 2006.

[22] T.Strohman and W. B.Croft. Efficient Document Retrieval in Main Memory. Proc. 30th Annual International ACM SIGIR , 2007.

[23] M. Theobald, G. Weikum, and R. Schenkel. Top-K Query Evaluation with Probabilistic Guarantees. In Proceedings of the 30th VLDB conference, 2004.

[24] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. Information Processing and Management, 31(6):831-850, 1995.

[25] M. Zhu, S. Shi, M. Li, and J.-R. Wen. Effective Top-K Computation in Retrieving Structured Documents with Term-Proximity Support. CIKM2007.

[26] M. Zhu, S. Shi, N. Yu, and J.-R. Wen. Can Phrase Indexing Help to Process Non-Phrase Queries? In ACM 17th Conference on Information and Knowledge Management (CIKM'08). Napa Valley, California, USA, 2008.