Ranking Objects by Exploiting Relationships: Computing Top-K over Aggregation

Kaushik Chakrabarti
Microsoft Research
kaushik@microsoft.com

Venkatesh Ganti
Microsoft Research
vganti@microsoft.com

Jiawei Han
University of Illinois
hanj@cs.uiuc.edu

Dong Xin*
University of Illinois
dongxin@uiuc.edu

ABSTRACT

In many document collections, documents are related to objects such as document authors, products described in the document, or persons referred to in the document. In many applications, the goal is to find such related objects that best match a set of keywords. The keywords may not necessarily occur in the textual descriptions of target objects; they occur only in the documents. In order to answer these queries, we exploit the relationships between the documents containing the keywords and the target objects related to those documents. Current keyword query paradigms do not use these relationships effectively and hence are inefficient for these queries.

In this paper, we consider a class of queries called the "object finder" queries. Our goal is to return the top K objects that best match a given set of keywords by exploiting the relationships between documents and objects. We design efficient algorithms by developing early termination strategies in presence of blocking operators such as group by. Our experiments with real datasets and workloads demonstrate the effectiveness of our techniques. Although we present our techniques in the context of keyword search, our techniques apply to other types of ranked searches (e.g., multimedia search) as well.

1 INTRODUCTION

In many applications like customer support, digital libraries, e-commerce, personal information management and health care, unstructured documents are often related to objects representing real entities. In a digital library like DBLP, for instance, unstructured documents like papers have objects like author names, publication dates and conference/journal names associated with them. Further, there is an increasing trend of automatically extracting structured information like details of named entities (e.g., names of persons, locations, organizations, products, etc.) from unstructured documents in order to move it up the value chain [7, 18]. The extracted details, being structured, are more amenable to complex querying and analysis. Unstructured documents are therefore usually accompanied by two types of information: (1) *objects*, either as attributes of the documents or automatically extracted from them or both, and (2) *relationship information* that describes which document is related to which object (e.g., paper-author relationship, document-entity relationship).

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2006, June 27-29, 2006, Chicago, Illinois, USA.

Copyright 2006 ACM 1-59593-256-9/06/0006…$5.00.

In many applications, *the goal is to find the objects related to documents that best match a set of keywords*. For example, in a digital library application, one might want to find the top authors in the areas of "databases" and "information retrieval". This is commonly known as the "expert finder" application [15]. A detailed example in the context of named entities (referred to as entities) is shown below.

Example 1.1 (Entity Finder): As shown in Figure 1, consider a database of product reviews. Suppose we extract product names from the reviews using an entity extractor. The database now has two distinct classes of objects: *reviews* with attributes ReviewId and ReviewText (and possibly other attributes) and *Product Entities* with attributes EntityId and EntityName. The *relationships* between reviews and entities are represented by a set of <ReviewId, EntityID> pairs. A pair *<d,t>* is in that set if the review with id *d* is related to the entity with id *t* which, in this case, means *t* has been extracted from *d*. An application might enable users to search for entities matching a set of keywords so that they may find products that best satisfy their desired criteria. In Figure 1, a user might search the reviews/opinions to find laptops using the keywords "lightweight" and "business use". Note that these keywords *do not* occur in the names of laptops. Hence, current keyword search techniques cannot be used to answer such queries.

Such entity finder functionality can be used to search for different object types (e.g., people, locations, companies, products, events, etc.) in a variety of domains. In this paper, we abstract out this functionality and formally define the above class of queries; we refer to them as "*object finder*" (OF) queries. First, we isolate two distinct classes of objects and the relationships among them.

1) **Search Objects** (SOs): These are searched by the keywords (e.g., papers in expert finder, reviews in the product finder)
2) **Target objects** (TOs): These are desired as answers to the query (e.g., authors in expert finder, entities in entity finder)

The **relationships** between the search and the target objects are represented by the set of <SO, TO> pairs as shown in Figure 1. The goal of an OF query is to *return the best K target objects that "match" a given set of keywords.* We address two important questions. First, how does a target object match a set of keywords? Second, how do we compute the relevance score of a target object in order to rank them? Consider the OF query with keywords "lightweight" and "business use" over the product review database in Figure 1. Intuitively, we expect the entities 'Dell Inspiron 700m' and 'Sony VAIO' to qualify as answers since the reviews related to these entities contain the given keywords. Thus, we need to find the reviews that contain the keywords using standard full text search (FTS), and then exploit the relationships between reviews and entities to find the qualifying entities. The relevance of an entity depends on how many of the reviews related to it contain the query

* Work done while visiting Microsoft Research.

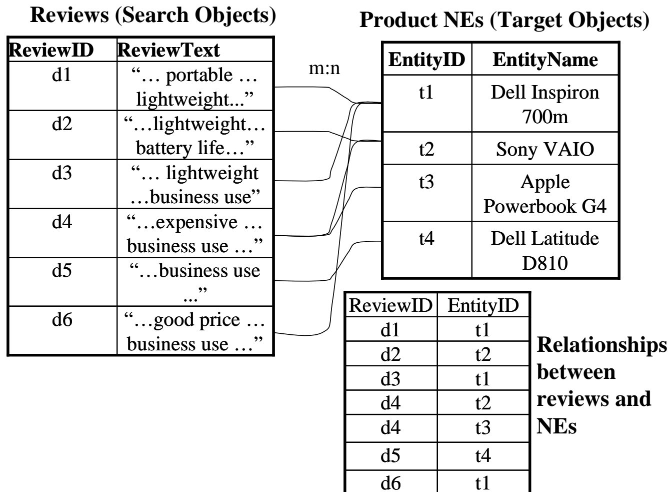


Figure 1: Search objects, target objects and relationships

keywords and how well they match with those keywords. Thus, the relevance score of an entity is an *"aggregate"* of the FTS scores (i.e., the keyword match scores returned by FTS) of all related reviews containing the query keywords. For example, the entity 'Dell Inspiron 700m' in Figure 1 is related to two reviews that contain keyword "lightweight" (d1 and d3) and two reviews that contain keyword "business use" (d3 and d6). So, the relevance score 'Dell Inspiron 700m' for those keywords is obtained by aggregating the FTS scores of d1 and d3 for keyword "lightweight" and those of d3 and d6 for keyword "business use". In this paper, we consider a broad class of scoring functions to compute relevance scores of target objects as aggregations over FTS scores of the related documents. Our OF query evaluation system would then return the K target objects with the best scores according to any chosen scoring function from the above class. Informally, the problem is to compute the *top K target objects with the highest scores, obtained by aggregating over one or more ranked lists.* The techniques we describe in this paper are applicable to this general class of queries, where ranked lists are obtained using any indexing sub-system (not necessarily FTS). However, while describing techniques in this paper, we assume that search objects are documents and the indexing sub-system is FTS, which returns a ranked list of documents for keyword queries.

The requirement that we aggregate (multiple) ranked lists of object scores prevents us from using existing work, especially the TA family of algorithms [10,11]. The threshold algorithm (TA) assumes that an object has a single score in each list. In our case, a target object can have multiple document scores, which need to be aggregated, in each list. (See Section 2 for a detailed discussion).

Most relational DBMSs now support FTS functionality. Hence, OF queries can be implemented in SQL. Figure 4 illustrates the schematic of such a query plan. SQL evaluations performing aggregation over FTS scores would be forced to retrieve all the documents containing the set of query keywords, join them all with the relationships table to find the related target objects, compute the aggregate scores of all these target objects, sort them all based on the score, and return the top K to the user. For large document collections, FTS may return large numbers of documents causing this implementation to be very inefficient. The challenge is to exploit the property that we only require the top K target objects and terminate early.

In this paper, we develop early termination techniques to efficiently evaluate the class of OF queries. We build upon existing infrastructure, FTS and DBMS engines, to support keyword queries over documents. Our approach is based on the following intuition: top scoring documents typically contribute the most to the scores of high scoring target objects. Hence, the target objects related to these top scoring documents are likely to be the best candidate matches. We progressively retrieve documents in the decreasing order of their scores, and maintain upper and lower bound scores for the related target objects. Using these bounds, we first identify *a superset of the top K target objects*. Then, in the second 'pruning' phase, we pick a subset of these candidates and compute their exact scores in order to isolate the exact top K target objects from them. The challenges in this approach are (i) to compute tight bounds in the presence of aggregation over FTS scores, and (ii) to minimize the number of target objects whose exact scores are computed in the pruning phase. We describe an algorithm that *performs the minimum number of such exact score computations*. Overall, the two-phase approach is very efficient when compared with existing techniques.

Our contributions in this paper can be summarized as follows. First, we formally introduce the class of OF queries. Second, we propose a class of scoring functions to compute the relevance scores of target objects. Third, we develop efficient early termination techniques to compute the top K target objects based on a scoring function within the above class. We present an extensive experimental study to determine the effectiveness of our scoring framework and evaluation techniques. Our experiments show that our early termination approach is often 4 to 5 times faster than a corresponding SQL implementation.

The remainder of the paper is organized as follows. In Section 2, we review related work. In Section 3, we provide an overview of the OF query evaluation system and present the class of scoring functions. In Section 4, we discuss the SQL implementation of OF queries. In Section 5, we present our early termination algorithms. We discuss a few extensions of our techniques in Section 6. In Section 7, we present experimental results. We conclude in Section 8.

2 RELATED WORK

Retrieving text documents containing a given set of query keywords has been studied extensively in Information Retrieval [3]. We cannot use these techniques to answer OF queries since the descriptions of target objects usually do not contain the query keywords. The functionality of returning entities for keyword queries to enable faster information discovery has been proposed earlier [8]. However, they do not discuss scoring functions or evaluation techniques for such queries. Recent work on keyword search over databases proposes to return 'joining networks' of related tuples that together contain a given set of keywords where the tuples are related by foreign key-primary key links [1,5,13]. However, these techniques do not consider aggregating the scores of multiple joining networks in order to identify desired target objects based on aggregated scores. But, they could be adapted by restricting the set of 'valid' joining networks to those whose central nodes (the node that connects the keyword nodes) correspond to the desired target objects. Subsequently, we can group those networks by the target objects, compute the aggregate scores and return the top K. This approach is inefficient because first, the number of such valid networks can become very large leading to high grouping cost and second, it, like the SQL implementation, does not have the early termination property.

Our early termination strategies are motivated by the large body of work on top-K queries. One of the most notable algorithms in this area is the TA (threshold algorithm) family of algorithms [10,11,12,14,16]. TA combines[2] the scores of objects in different lists and computes the top-K objects based on the combined score. However, it does not consider aggregation of multiple scores within each list. In our problem, if we know a priori that a target object is related to at most one document, the subsequent aggregation of scores per each target object is redundant; we can use TA in this case to find top-K target objects efficiently. However, in most scenarios a target object is typically related to multiple documents. For example, an entity is typically present in multiple documents, and an author typically writes multiple papers. In such cases, TA cannot be used.

Another potential approach is to pre-aggregate the scores of the target objects for various keywords and materialize them, thereby taking aggregation out of the picture and reducing the problem to combining these materialized lists at query time. This can be done efficiently using the TA algorithm. Such an approach for authority based ranking of objects is proposed in [4]. This approach has several limitations. First, it cannot handle selections on the documents. Second, we found that the pre-aggregation strategy imposes significant space overhead; it might not be feasible to maintain the scores of all target objects for all keywords. Third, this strategy is not applicable to non-keyword ranked searches like multimedia searches or ranked searches over structured data.

3 SYSTEM OVERVIEW AND SCORING FUNCTIONS

We build upon FTS and DBMS systems by indexing documents using FTS, and by storing and querying the relationship and target objects in SQL Server. We first present an overview of the ObjectFinder (OF) query evaluation system to lay the ground for the subsequent discussion on the class of scoring functions.

3.1 System Overview

Figure 2 shows the overview of our system. We describe the functionality we assume from each of these systems.

FTS: We index the text content of the documents using an FTS system (at the preprocessing stage) so that we can support keyword queries on them at query time. We assume that the FTS system supports the following query interface: given a single keyword or a multi-keyword query, it provides *"sorted access" to the ranked list of documents matching with the query*, i.e., the application can retrieve the next best document from the ranked list along with the score. We refer to these documents scores as *DocScores*. For clarity in description, we assume that all documents are indexed by a single FTS index. Our techniques can also be extended to multiple FTS indexes that index different sets of documents.

DBMS: We store the target objects and the relationships in the DBMS in two distinct tables: (i) the target object table *T* which has schema <TOId, TOValue> and stores the ids and values of the target objects, and (ii) the relationships table *R* which has schema <DocId, TOId> and stores the document-target object pairs that are

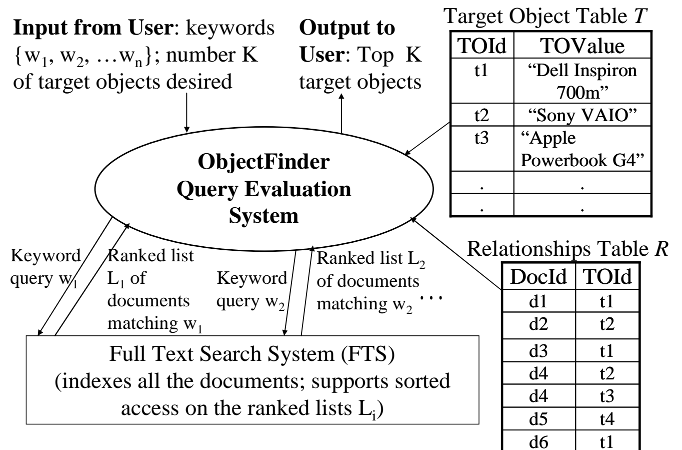


Figure 2: Overview of OF Query Evaluation System

related to each other. The application might have multiple types of target objects (e.g., different types of entities like persons, locations, products, etc. in the entity finder application) and the user might specify the desired type in the OF query. This can be implemented using the above architecture by either storing the target objects and the relationships for each type in separate *T* and *R* tables or by adding a type column to the tables. For the purposes of description, we assume only one type of target objects. Our approach can be extended to take into account static weights and, as discussed further in Section 6, selections on search and target objects and static weights associated with target objects [17].

In evaluating the OF query, we focus on obtaining the identifiers of the top K target objects matching with the keywords; *T* is used only for the final lookup of the TOvalues corresponding to those TOIds before returning to the user. Hence, the ranked lists and *R* are the main inputs to the OF evaluation system. Besides sorted access on the ranked lists from FTS, we require the following two access methods on *R*:

1) *Random access over R on DocId* to find the identifiers of the target objects related to a given DocId or a given set of DocIds.
2) *Random access over R on TOId* to find the identifiers of the documents related to a given TOId.

We assume appropriate physical design for *R* to make the above random accesses efficient.

3.2 Class of Scoring Functions

We now describe the class of scoring functions we consider in this paper. Our OF evaluation system would return the K target objects with the best scores according to the scoring function chosen from this class. Informally, each function in this class computes the score of any target object by aggregating the DocScores of the documents related to it occurring in those ranked lists.

Let $W = \{w_1, w_2, \ldots, w_N\}$ denote the set of N keywords in the OF query. Let L_i denote the ranked list of document identifiers along with DocScores that would be returned by the FTS system for the single keyword query $\{w_i\}$. Let D_t denote the list of documents related to *t*. The DocScores of the objects in D_t in the above lists define the *score matrix* M_t of *t*; the cell $M_t[i,j]$ contains the DocScore of the ith object in D_t in list L_j; it contains 0 if ith object in D_t is not present in L_j. Let *Score(t)* denote the relevance score for the target object *t* (computed using M_t).

[2] TA refers the combination of scores from different lists as "aggregation"; we refer to this as "combination" in this paper. In this paper, "aggregation" refers to aggregation of multiple scores for the same object within a list.

Example 2.1 (Score Matrix): Consider the keyword query {"lightweight", "business use"} in Example 1.1. Suppose FTS returned ranked list L_1={(d1, 0.8), (d2, 0.6), (d3, 0.3)} for keyword "lightweight" and ranked list L_2={(d5, 0.7), (d4, 0.5), (d3, 0.4), (d6, 0.1)} for keyword "business use". Consider the target object "Dell Inspiron 700m"; D_t = {d1, d3, d6}. The score matrix of "Dell Inspiron 700m" for the above query is shown in Figure 3(a). ■

A general scoring function would take the entire score matrix M_t to compute *Score(t)*. However, efficiently retrieving the best top K target objects according to any arbitrary function would be very hard without fetching all relevant documents and target objects. We therefore consider the following two classes of functions. These functions first compute either the row marginals or the column marginals of the score matrix and then aggregate these marginals. We use the term 'marginal' loosely in that the function for computing the row or column marginal may not be the sum function.

1) Row-marginal Class: The overall score *Score(t)* of the target object t is computed in 2 steps. In step 1, we *combine* the scores in each row of the score matrix of t using a *combination* function F_{comb}, i.e., for each document $d \in D_t$, we combine its DocScores in the N lists using F_{comb}.[3] In step 2, we *aggregate* the combined scores of all the documents in D_t using an aggregation function F_{agg} to obtain the overall score. Formally,

$$\text{Score}(t)= F_{agg\; d\in D_t}\left(F_{comb}(\text{DocScore}(d,L_1),\ldots, \text{DocScore}(d, L_N))\right) \quad (1)$$

where DocScore(d, L_j) denotes the DocScore of the document $d \in D_t$ in list L_j (= 0 if $d \notin L_j$). Applications can define a wide variety of scoring functions in this class by plugging in different F_{comb} and F_{agg}; an example of such a scoring function with F_{comb} = MIN and F_{agg}= SUM applied to the score matrix in Figure 3(a) is shown in Figure 3(b).

2) Column-marginal Class: *Score(t)* is computed in 2 steps. In step 1, we aggregate the scores of each column of the score matrix of t using an aggregation function F_{agg}, i.e., for each list, we *aggregate* the DocScores of all documents in D_t in that list. In step 2, we combine the aggregate scores of the N lists using a combination function F_{comb} to obtain the overall score. Formally,

$$\text{Score}(t)= F_{comb}\left(F_{agg\; d\in D_t}(\text{DocScore}(d,L_1)),\ldots, F_{agg\; d\in D_t}(\text{DocScore}(d,L_N))\right) \quad (2)$$

Again, applications can define a wide variety of scoring functions in this class by plugging in different F_{agg} and F_{comb}; an example of such a scoring function with F_{agg}= SUM and F_{comb} = MIN applied to the score matrix in Figure 3(a) is shown in Figure 3(c).

Properties required of F_{agg} and F_{comb}: Our early termination techniques are applicable when F_{agg} and F_{comb} satisfy certain properties. We say that F_{comb} is monotonic if $F_{comb}(x_1,\ldots, x_n) \leq F_{comb}(y_1,\ldots, y_n)$ when $x_i \leq y_i$ for all i. We say that F_{agg} is *subset monotonic* if $F_{agg}(S) \leq F_{agg}(S')$ if $S \subseteq S'$. This implies that, at any stage of aggregation, aggregating additional scores cannot decrease the aggregate score. Sum, count, max, sum_top_D and avg_top_D are examples of subset monotonic functions where sum_top_D (avg_top_D) denote sum (average) over the highest D scores in the set of scores being aggregated; note max is a special case of sum_top_D with D=1. Avg and min are not subset monotonic, and hence we cannot support the instantiation of F_{agg} with avg. Note

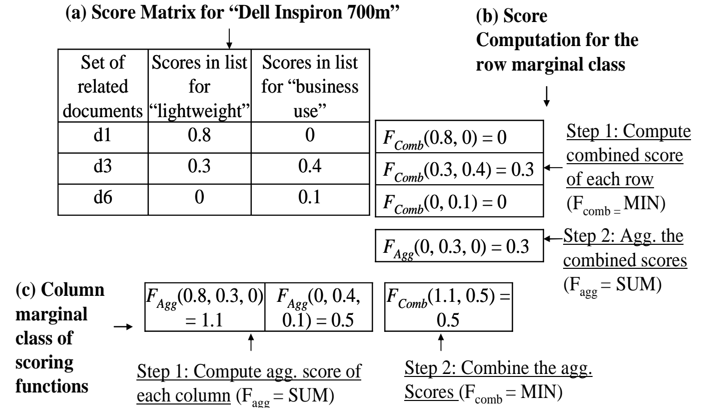


Figure 3: Score Matrix and Classes of Scoring Functions

that we can support avg_top_D which emulates the properties of average. We say that F_{agg} *distributes* over append if $F_{agg}(R_1$ *append* $R_2) = F_{agg}(F_{agg}(R_1), F_{agg}(R_2))$, where *append* denotes the ordered concatenation of lists of tuples. In our case, we invoke this property over ordered (disjoint) fragments of ranked lists.

Our early termination techniques (described later) can be applied if (i) F_{comb} is monotonic, (ii) F_{agg} distributes over append and is subset monotonic.

3.3 Example Instantiations

We now discuss instantiations of scoring functions in order to model certain semantic notions of matching for target objects.

All Query Keywords Present in each Document: Consider the match behavior where we say a target object *t* matches with the keyword query W iff one or more documents related to *t* contains *all* the keywords in W; higher the number of such documents related to it and higher their scores, better the match for *t*. We can implement this notion using the row-marginal scoring framework by choosing an F_{comb} that conserves the standard propositional semantics for conjunction like *min*[4] [10] while F_{agg} can be a subset monotonic function like *sum*. An example for this choice of F_{comb} and F_{agg} is shown in Figure 3(b). "Dell Inspiron 700m" has a non-zero score because one of the documents related to it (d3) contains all the keywords and its final score depends only on the combined keyword score of d3.

The above notion of matching may be restrictive when there is no document related to t that contains all the keywords. For example, consider the target object "Sony VAIO" in Figure 1. None of the documents related to it contains both the keywords "lightweight" and "business use". So it is not a match by the above notion. However, one review related to it contains "lightweight" and another contains "business use". Intuitively, it should match with the query. To handle this, we consider a more relaxed notion of match.

All Query Keywords Present in Set of Related Documents: Consider the match behavior where we say *t* matches with W iff the documents related to it *together cover* all the keywords in W, i.e., each keyword in W is contained in at least one document related to t. The more the number of matching objects related to t for each

[3] The number of arguments of F_{comb} is fixed once the number of keyword queries issued against FTS is known. But, the arity may vary across queries.

[4] Note that we do not require F_{comb} to conserve conjunctive semantics for our techniques to work; we only require it to be monotonic.

keyword and the higher those individual keyword match scores, the better the match for *t*. Clearly, this definition would return "Sony VAIO" as a match in Example 1. We can implement this definition using the column-marginal scoring framework by choosing a subset monotonic function like *sum* as F_{agg} and *min* as F_{comb}. An example for this choice of F_{comb} and F_{agg} is shown in Figure 3(c); "Dell Inspiron 700m" has a non-zero score because the set of objects related to it (i.e., d1, d3 and d6) covers both keywords. Note that this notion cannot be implemented using the row-marginal framework.

Pseudo-document Approach: Consider the following simulation of facilitating keyword search over target objects. Suppose we associate with each target *t* object a pseudo-document created by concatenating all documents that *t* is related to. We can now index these pseudo-documents using FTS and directly facilitate keyword queries over them. Now, the ranked list of "documents" returned by FTS corresponds to a ranked list of target objects, which is our desired goal. However, the overall size of the pseudo-document collection is several times larger because each document is replicated once per target object it is related to. We can instantiate a scoring function within our class to often simulate the same effect as the pseudo-document approach.

Most FTS scoring functions assigning relevance scores to documents have two components: (i) a function F_{score} which scores a document per query keyword, and (ii) a combination (using a function F_{comb}, say, a linear combination based on IDF weights) of these scores across all keywords. TF-IDF scoring functions, commonly used in IR systems, are examples of this type of scoring functions: F_{score} is term frequency (TF) and F_{comb} is a linear combination of document scores per keyword where the coefficients are determined by the IDF weights of the keywords. Suppose F_{score} distributes over concatenation of documents: F_{score}(d1 *concat* d2) = F_{score}(d1) + F_{score}(d2). The term frequency function is such an example. Under the conditions that F_{score} is additive and F_{comb} is fixed (i.e., does not change with document collection), choosing a function within a column marginal framework where F_{agg} is sum, and F_{comb} is the combination used by FTS would achieve the desired functionality.

3.4 Object Finder Problem

Problem statement: Given a list $w_1,\ldots,w_N$ of query keywords, the scoring function *f* in either the row-marginal or the column-marginal class, the interfaces for keyword queries over FTS and for random access on the relationships table R on both DocId and TOId, compute the K target objects with the highest scores.

For the row-marginal class of scoring functions, it is possible to perform the combination inside the FTS system if FTS supports the desired combination function. For the match notion where all query keywords have to be present in each relevant document, we can submit to FTS a single combined keyword query Q = (w_1 AND w_2 AND … AND w_N). The score *Score(t)* is then obtained by aggregating the DocScores of the documents related to *t* occurring in the single ranked list returned by FTS for the above AND query:

$$\text{Score}(t) = F_{agg\; d \in D_t}\left(\text{DocScore}_{\text{AND query}}(d)\right) \quad (3)$$

The advantage here is that the combination over the keywords is performed by FTS and hence can be very efficient. And, such a strategy may be possible for other types of combinations (e.g., disjunction) as well. In this case, the problem for the row marginal class is the same as that for the column marginal class except that there is a single combined keyword query, which returns a single

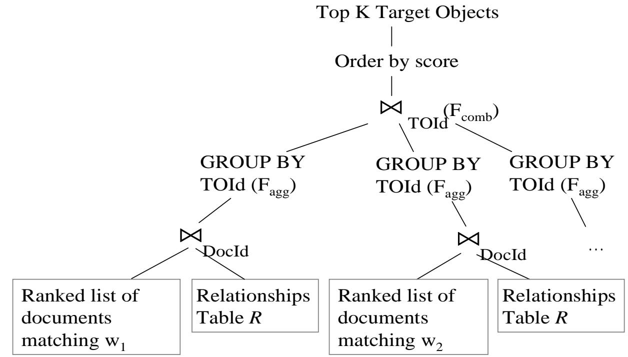


Figure 4: SQL query plan for column marginal class

ranked list of documents. For a general combination function that is not supported by FTS, we obtain a single ranked list corresponding to the combination query as follows. We issue individual keyword queries to FTS, combine the ranked lists of documents using an algorithm such as NRA [11,14], to provide a single combined ranked list for subsequent aggregation. The problem again reduces to a column-marginal class with a single combined keyword query.

For the column-marginal class, it is not possible to perform the combination inside FTS (even if FTS supports the combination function) since the aggregation over documents needs to be done first. Hence, we always need to submit individual keyword queries to FTS. We focus on the column marginal class while describing our evaluation techniques in Sections 3 and 4; we conduct experiments with both classes in our experiments section.

4 SQL IMPLEMENTATION

Commercial DBMSs now support FTS functionality by providing specialized user-defined functions (UDFs) to perform keyword search on text columns of database tables [9]. Therefore, we can implement OF queries in SQL using these FTS UDFs.

Figure 4 shows the execution plan for the column-marginal class. We join each list individually with the relationships table *R* on DocId to get the related target objects. We then group each join result by TOId and aggregate the DocScores using F_{agg}. We then do a full outer join (on TOId) of the aggregation results and compute the combined score of each target object by combining its aggregate score from each list using F_{comb}. Finally, we order the TOIds by the combined scores and return the top K TOIds. A clustered index on *R*.DocId may help the join with *R* to be efficient. Observe that algorithms such as TA may only be applied for the second join above the group by operators.

As discussed earlier, the presence of blocking operators (group by and order by) in the plan makes the evaluation wasteful (cf. Figure 9). Since the user is typically interested in only top K target object, we can significantly reduce these costs by retrieving the top documents from the ranked lists "progressively". Since we cannot do such progressive evaluation using SQL, we implement such an approach in middleware.

5 EARLY TERMINATION APPROACH

In this section, we describe our approaches for OF query evaluation. The idea is to retrieve a small number of the top documents from each ranked list, get related target objects by looking up the relationships table *R*, and determine upper and lower bound scores for those target objects "seen" so far. Often these bounds can guide

us in stopping early. We identify two approaches for leveraging these bounds to stop early.

Generate-only Approach: This approach relies completely on the bounds and stops when it can determine that it has identified the best K target objects based on a "stopping condition". We stop if the condition is met and continue fetching more documents otherwise. This technique is similar in flavor to the NRA algorithm [11]; however, the techniques for computing bounds are different due to the aggregation operator.

Generate-Prune Approach: This approach has two phases: a *candidate generation* phase followed by a *pruning* phase. During the generation phase, we use the bounds to identify a superset of the top K target objects. The condition to check that we have identified a superset is more relaxed than that in the Generate-only approach and hence retrieves fewer documents from the ranked lists and does fewer lookups in *R* (on DocId). During the pruning phase, we isolate the subset of the best K target objects.

The algorithm for the Generate-only approach is identical to the algorithm for the generate phase of the Generate-Prune approach except for the stopping condition. Therefore, we first describe the Generate-Prune approach in detail and then discuss the stopping condition for the Generate-only approach.

5.1 Candidate Generation

The goal of the candidate generation phase is to generate a superset of the top K target objects. We submit keyword queries, one for each keyword w_i, to FTS and obtain ranked lists $L_1,\ldots,L_N$ of documents. We process these lists iteratively. In each iteration, we retrieve more documents from each ranked list L_i and evaluate the query shown in Figure 5 over the prefixes, denoted by *Prefix(L_i)*, of documents retrieved so far from the L_is. Evaluation of the query generates the *SeenTOs* table that contains the current aggregation scores as well as the lower and upper bound scores of all target objects related to one or more documents in any Prefix(L_i). Using the *SeenTOs* table, we check whether or not we can stop further retrieval from the ranked lists. If so, we identify candidates from the *SeenTOs* table, otherwise, we retrieve more documents from each L_i and iterate the above process.

A straight-forward approach is to generate the *SeenTOs* table from scratch every time, i.e., re-evaluate the query shown in Figure 5 over the entire prefixes in every iteration. This is wasteful as it would repeatedly evaluate Group By and combination over the join result of entire prefixes with the relationships table R. We exploit the distributive properties of both F_{agg} and the join operator in order to evaluate the query *incrementally*, i.e., in each iteration, we process just the newly retrieved documents and update the scores in the *SeenTOs* table. The algorithm therefore has 5 steps:

1. Retrieve more documents from each L_i
2. Update *SeenTOs* table for the prefixes Prefix(L_i) retrieved so far from the L_is
3. Compute lower and upper bounds
4. Check stopping condition. If satisfied, go to 4; else go back to 1
5. Identify the candidates among the seen target objects

We now discuss each of these steps in detail.

Step 1 (Retrieve Documents): In each iteration, we retrieve the next chunk C_i of documents from each L_i. We retrieve the documents in chunks in order to reduce the number of join queries (with *R*) issued to the DBMS. The choice of the chunk size presents a tradeoff between the number of "unnecessary" documents (i.e., not necessary to generate candidates) retrieved from FTS and number of join queries issued to DBMS; we choose a chunk size of 100 for all lists in our experiments.

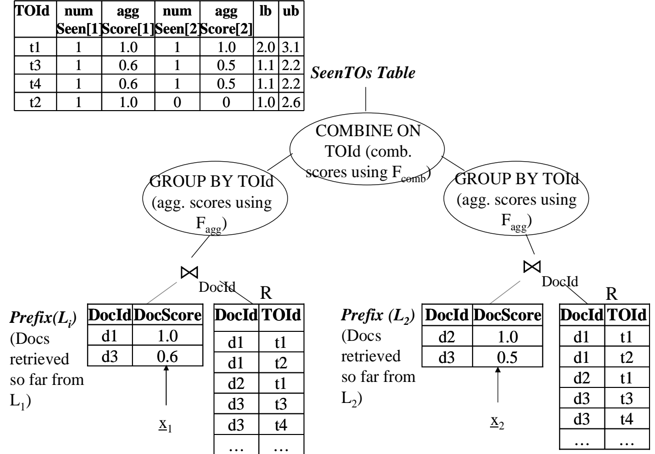


Figure 5: Query Evaluated over the prefixes Prefix(L_i) in each iteration.

Step 2 (Update SeenTOs): We discuss how to process the new chunks C_i retrieved in the current iteration incrementally and update the *SeenTOs* table. This has two parts: incrementally computing the Group By and the combination.

Compute Group By incrementally: As shown in Figure 5, the Group By is computed for each list L_i. For each L_i, we maintain, across iterations, the *AggResult$_i$* table containing the following information for each target object *t* related to one or more documents in Prefix(L_i): the number *numSeen* of documents in Prefix(L_i) related to *t* and the "current" aggregate score *aggScore*, i.e., the aggregate of the DocScores of the documents in Prefix(L_i) related to t. We discuss computing the aggScore column in *AggResult$_i$* incrementally; the *numSeen* column is computed in a similar fashion. Formally, the aggScores in *AggResult$_i$* at the end of any iteration is $\text{GroupBy}_{\text{TOId}}(\text{Prefix}(L_i) \bowtie_{\text{DocId}} R, F_{agg}(\text{DocScore}))$ where $\text{GroupBy}_A(S, F(B))$ denotes Group By over relation S on column A and aggregation on column B using aggregate function F. *AggResult$_i$* is empty at the start of first iteration. The new prefix after this iteration is (Prefix(L_i) *append* C_i), so the new *AggResult$_i$* after this iteration should be: $\text{GroupBy}_{\text{TOId}}((\text{Prefix}(L_i)\ append\ C_i) \bowtie_{\text{DocId}} R, F_{agg}(\text{DocScore}))$. Since both join and F_{agg} distribute over *append*, the new aggScores can be obtained from $\text{GroupBy}_{\text{TOId}}(\text{Prefix}(L_i) \bowtie_{\text{DocId}} R, F_{agg}(\text{DocScore}))$ (the *AggResult$_i$* from the previous iteration) and $\text{GroupBy}_{\text{TOId}}(C_i \bowtie_{\text{DocId}} R, F_{agg}(\text{DocScore}))$ (the AggResult for the current chunk). We first compute the AggResult for the current chunk C_i by joining it with *R* and then aggregating on the join result using F_{agg}. We then merge the AggResult for the current chunk into the AggResult$_i$ table as follows. For each *t* in the AggResult for the current chunk, if *t* is already present in the *AggResult$_i$* of previous iteration, we update the t.aggScore to F_{agg}(previous t.aggScore, t.aggScore for the current chunk). Otherwise, we insert *t* into *AggResult$_i$* setting t.aggScore to be that in the current chunk.

To update AggResult$_i$ efficiently, we maintain AggResult$_i$ as a hash table keyed on TOId. Therefore, in each iteration, we evaluate the join and the Group By over the newly retrieved chunks only.

Combine Incrementally: The combined *SeenTOs* table is a full outer join on TOId of the AggResult$_i$ tables defined above. Since the AggResult$_i$ are anyway in-memory hash tables, we can

performing the outer join simultaneously with the aggregation by maintaining a unified *SeenTOs* hash table and updating the *SeenTOs* table directly instead of maintaining separate $AggResult_i$ tables. The *SeenTOs* table contains any target object t present in any of the $AggResult_i$ tables and all the columns in $AggResult_i$ tables, i.e., it contains t's numSeen and aggScore values for each L_i. We denote these columns as *numSeen[i]* and *aggScore[i]* respectively. If a target object $t \in$ *SeenTOs* is not present in $AggResult_i$, t.numSeen[i] and t.aggScore[i] are set to 0. To efficiently update the lower and upper bound scores of target objects in *SeenTOs* in each iteration, we maintain *SeenTOs* as a hash table keyed on TOId.

Step 3 (Compute bounds): In this step, we take *SeenTO*s table generated in Step 2 and compute the lower and upper bound scores of each target object t in *SeenTOs* (stored as 2 separate columns in *SeenTOs* table). Here, we exploit the subset monotonicity property of F_{agg} and the monotonicity property of F_{comb}. We first consider the computation of lower bound score. Since F_{agg} is subset monotonic, the "current" aggregate score t.aggScore[i] of t for L_i is the lower bound of *t* for L_i. The combined lower bound score of t, denoted by t.lb, is the combination of the individual lower bound scores t.aggScore[i], i.e.,

t.lb = F_{comb} (t.aggScore[1],…,t.aggScore[N]).

We now consider the computation of the upper bound score. The computation of the upper bound scores depends on a crucial constant *B* called the *cardinality bound. B* is the maximum number of documents in any ranked list L_i that can contribute to the score of any target object *t*. For the following discussion, we assume *B* is known; we discuss its computation in Section 5.1.2. Since there are *t.numseen[i]* documents related to t in Prefix(L_i), there can be at most (*B - t.numseen[i]*) documents in (L_i – Prefix(L_i)) that can contribute to the aggregate score of *t* for L_i. Furthermore, the DocScores of such unseen documents is upper bounded by the DocScore $\underline{x}_i$ of the last document retrieved from L_i as shown in Figure 5. The upper bound score of t for list L_i, denoted by t.ub[i], is therefore aggregation of the current aggregate score (i.e., t.aggScore[i]) and the upper bound of the remaining contribution:

$$t.ub[i] = F_{agg}\left(t.aggScore[i],\quad F_{agg}\left(\underline{x_i}, \underline{x_i}, \ldots, (B - t.numseen[i])\, times\right)\right)$$

The combined upper bound score, denoted by t.ub, is:

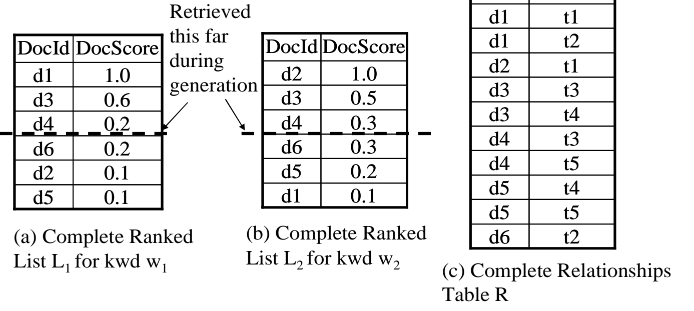
t.ub = F_{comb}(t.ub[1],…,t.ub[N]).

Step 4 (Stopping Condition): We can stop when there are at least K objects in *SeenTOs* whose lower bound scores are higher than the upper bound score of any unseen target object (i.e., target object not in *SeenTOs*). This guarantees that no unseen object can qualify for the final top K, i.e., *SeenTOs* is guaranteed to contain the final top-K target objects. Let *UnseenUB* denote the upper bound score of any unseen target object. Using the same logic as t.ub computation:

$$UnseenUB \;=\; F_{comb}\left(F_{agg}(\underline{x_1}, \underline{x_1}, \ldots, B\, times), \ldots, F_{agg}(\underline{x_N}, \underline{x_N}, \ldots, B\, times)\right)$$

Let LB and UB denote the list of all target objects in *SeenTOs* sorted in decreasing order of their lower and upper bounds, respectively and let LB_j (UB_j) denote the jth largest lb (ub) value in LB (UB).The stopping condition is: $LB_K \geq$ *UnseenUB*.

Step 5 (Identify candidates): In this step, we filter out objects from *SeenTOs* which cannot be in the final top K. Consider an object in *SeenTOs* whose upper bound score is less than the lower bounds of at least K target objects. This object cannot be in the final top K and hence can be filtered out. Let Top(List, X) denote the top X elements in the list. The set of candidates is defined by



DocId	DocScore
d1	1.0
d3	0.6
d4	0.2
d6	0.2
d2	0.1
d5	0.1

(a) Complete Ranked List L_1 for kwd w_1

DocId	DocScore
d2	1.0
d3	0.5
d4	0.3
d6	0.3
d5	0.2
d1	0.1

(b) Complete Ranked List L_2 for kwd w_2

DocId	TOId
d1	t1
d1	t2
d2	t1
d3	t3
d3	t4
d4	t3
d4	t5
d5	t4
d5	t5
d6	t2

(c) Complete Relationships Table R

Figure 6: Complete ranked lists and Relationships Table for Examples

Top(UB, h) where h is the least value which satisfies (i) $LB_K \geq UB_{h+1}$ and (ii) Top(LB, K) $\subseteq$ Top(UB, h). To ensure Top(LB,K) $\subseteq$ Top(UB,h), we order objects in LB and UB as follows. For any two objects O and O', if their lower bound scores (upper bound scores) are equal, we order them in LB (UB) in the decreasing order of their upper bound score (lower bound score). If both their upper bound and lower bound score are same, we rank them based on their TOId.

Memory Requirement: We assume that the *SeenTOs* table fits in memory. If *SeenTOs* becomes too large, we need to use a disk-resident hash table.

Lemma 1*:* With the above ordering rule, if $LB_K \geq UB_h$, then the final top-K objects are within the top-h objects of UB.

Example 5.1: Consider the OF query whose complete ranked lists and relationships table *R* is shown in Figure 6. Let F_{agg} = SUM, F_{comb} = SUM, B = 2, $|C_i|$ (chunk size) = 1, K = 3. During the first round, we retrieve (d1, 1.0) from L_1 and (d2, 1.0) from L_2. We look them up in *R* and get (t1, 1.0) and (t2, 1.0) for L_1; (t1, 1.0) for L_2. At this stage, $\underline{x}_1$ = 1.0 and $\underline{x}_2$ = 1.0; the bounds of target objects seen so far are (sorted by lb):

	num Seen[1]	agg Score[1]	ub[1]	num Seen[2]	agg Score[2]	ub[2]	lb	ub
t1	1	1.0	2.0	1	1.0	2.0	2.0	4.0
t2	1	1.0	2.0	0	0	2.0	1.0	4.0

Also, *UnseenUB* = F_{comb}(F_{agg}(1.0, 1.0), F_{agg}(1.0, 1.0)) = 4.0. LB_3 = 0 (because there are only two target objects in LB list), LB_3 not $\geq$ *UnseenUB*, so we get more documents. We retrieve (d3, 0.6) from L_1 and (d3, 0.5) from L_2. The new join result is (t3, 0.6) and (t4. 0.6) for L_1; (t3, 0.5) and (t4. 0.5) for L_2. We compute the bounds for the target objects seen so far; the stop condition still does not hold. So, we continue retrieving more documents; we retrieve (d4, 0.2) from L_1 and (d4, 0.3) from L_2. We get (t3, 0.2) and (t5. 0.2) for L_1; (t3, 0.3) and (t5. 0.3) for L_2. At this stage, $\underline{x}_1$ = 0.2 and $\underline{x}_2$ = 0.3; the bounds of target objects seen so far are (sorted by lb):

	num Seen[1]	agg Score[1]	ub[1]	num Seen[2]	agg Score[2]	ub[2]	lb	ub
t1	1	1.0	1.2	1	1.0	1.3	2.0	2.5
t3	2	0.8	0.8	2	0.8	0.8	1.6	1.6
t4	1	0.6	0.8	1	0.5	0.8	1.1	1.6
t2	1	1.0	1.2	0	0	0.6	1.0	1.8
t5	1	0.2	0.4	1	0.3	0.6	0.5	1.0

UnseenUB = F_{comb}(F_{agg}(0.2, 0.2), F_{agg}(0.3, 0.3)) = 1.0. LB_3 = 1.1 $\geq$ *UnseenUB*, so we go to Step 4. *h* turns out to be 4, the candidate set is Top(UB, 4) = {t1, t2, t3, t4}. ■

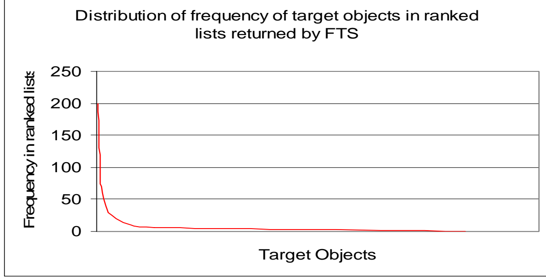


Figure 7: Frequency distribution of target objects

5.1.1 Stopping Condition for Generate-Only Approach

The algorithm for the Generate-Only approach is identical to the candidate generation algorithm presented above except that the stopping condition in Step 3 is $LB_K \geq UB_{K+1}$ *and Top(LB, K) = Top(UB, K)* instead of $LB_K \geq$ *UnseenUB*. That is, we stop when the K target objects with the highest lower bound scores have lower bound scores greater than or equal to the upper bound score of any target object outside those K target objects; these are guaranteed to be the final K target objects.

Lemma 2*:* The top-K objects in UB list is the final top-K if and only if $LB_K \geq UB_{K+1}$ and Top(LB, K) = Top(UB, K).

5.1.2 Computation of Cardinality Bound B

The bound B on the number of documents in a list L_i that can contribute to the score of a target object may be computed in one of the following ways: using properties of aggregation functions, data characteristics, and materialized statistics.

Using Properties of Aggregation Functions: Consider the example where F_{agg} is max. Then, B=1. Another bounded aggregation function is sum_top_D. Recall that sum_top_D computes the aggregate score of a target object t for list L_i by summing of the DocScores of the top D documents in L_i related to t. In this case, B = D. Sum and Count are examples of aggregation functions where B is unbounded.

Using Data Characteristics: In many real scenarios, each target is related to a bounded number of documents. For example, in the entity finder application, we might know that an entity can appear in at most M documents or, in the expert finder application, an author has written at most M papers. This bounds the number of documents related to a target object t than can occur in L_i (referred to as the *frequency* of T in L_i); so B = M. In cases where both aggregation function and data are bounded, B is simply the minimum of the two as shown in table below:

	F_{agg}=Max	F_{agg}=sum_top_D	F_{agg} unbounded
M bounded	B = 1	B = min(D, M)	B = M
M unbounded	B = 1	B = D	B unbounded

Materialized Statistics: In many real scenarios, only a few target objects have frequency M or close to M. The majority of the target objects have much lower frequency. A typical distribution of frequency of target objects in the ranked lists for single keyword queries is shown in Figure 7. While M is about 200 in the above example, more than 99% of the target objects have frequency less than 80. We propose an optimization, referred to as *frequent target object materialization (FTM)*, to obtain a tight bound on B. The *frequency* of a target object is the number of documents it is related to. For each keyword *w*, we materialize the target objects with frequency above a threshold θ in the ranked list returned by FTS for the single keyword query {*w*}. We store their TOIds along with their final aggregated score for w.[5] For any keyword *w*, we know that the target objects not found in the materialized set for *w* have frequency less than or equal to θ. So the bound B for such target objects is B = min(d, M, θ). The value of the cardinality bound B is used in the candidate generation algorithm as explained earlier. The materialized (exact) scores are used as the lower and upper bound scores for these objects. During the algorithm, we do not update the bounds for these materialized objects. Note that the FTM optimization has two-fold benefit: (1) we have the exact scores of the materialized target objects (t.lb[i] = t.ub[i] = materialized score) and (2) we have better bounds for the remaining target objects. Both these factors lead to earlier termination of the algorithm. The choice of θ offers a tradeoff between efficiency of candidate generation and space overhead; lower values of θ imply faster termination but higher space overhead and vice-versa. In our experiments, we allow space to store an average 1% of the target objects for each keyword which results in choice of θ = 80.

5.2 Pruning to the Final Top-K

The goal of this phase is to isolate the exact top K target objects from the superset of candidates identified in the generation phase. Our main insight is that it is sufficient to compute the exact scores of a subset of the candidate target objects and then stop. Computing the exact score of a target object entails queries against the relationships table *R* and accessing (DocId, DocScore) pairs in the lists returned by FTS. The challenge is to compute the exact scores of as few candidates as possible and still return the top K objects with exact scores. We now present an algorithm that computes the score for the smallest possible number of candidate target objects. In some applications, it is sufficient to return the top K target objects even if their scores are not exact. In such scenarios, we show that we can improve the algorithm even further.

5.2.1 Exact Top-K with Exact Scores

When the exact score of the Kth best target object is greater than or equal to the upper bound scores of any candidate target object whose score is not yet computed, the current best K target objects are the overall best K objects. We show that the best pruning strategy (i.e., the one that computes the exact scores of least number of target objects) is to iteratively sort the candidate target objects in the decreasing order of their upper bound scores and compute exact scores in that order until we can stop. The pseudocode of the algorithm is shown in Table 1.

We initially mark all candidate target objects as 'uncomputed' (Step 1). In Step 2, we sort the candidate target objects in the decreasing order of their upper bound scores using lower bound scores to break ties. In Step 3, we stop if all the top K target objects in UB have exact scores computed and return these K objects. (We save on computing exact scores of the remaining candidates.) This is correct because all other candidates have upper bounds less than these exact scores and hence cannot be better than them. If not, in Step 4, we compute the exact scores of all 'uncomputed' target objects in UB with the K best upper bounds. We update their upper bound scores (replace them with exact scores), mark them as computed, and go back to Step 2.

[5] For uncommon keywords, i.e., keywords contained in very few documents, there are typically no frequent target objects, so nothing is stored.

Algorithm *Prune_Exact_Scores*

1. Mark all candidate target objects as 'uncomputed'.
2. Sort all the candidate target objects by their upper bound scores, get sorted list UB (break ties based on lower bound scores)
3. If all target objects in top K of UB are computed, return these and stop.
4. Otherwise, compute exact score of all 'uncomputed' target objects within top-K in UB, update their upper bound scores, mark them as computed, and go to step 2.

Table 1: Pruning Algorithm for Exact top-K, Exact scores

Computing Exact Score of a Candidate Target Object: To compute the exact score of a candidate t, we first get the set D_t of documents related to *t* by looking up *R*. Subsequently, we obtain the DocScore of each document in D_t in each list L_i and compute the exact score using Equation 2. Since FTS systems do not usually provide random access to documents in the ranked lists, we retrieve all document identifiers with scores from each L_i using sorted access and store them in a hash table or a temporary relation (depending on the size) keyed on DocId to provide that random access. However, unlike in the candidate generation phase, these additional documents retrieved are *not* joined with *R*.

Example 5.2: Continuing with Example 5.1, the candidate set is {t1, t2, t3, t4}. *Pruning_Exact_Scores* will retrieve all the remaining documents from each L_i shown in Figure 6. After sorting by upper bound scores, UB = {t1, t2, t3, t4} with upper bound scores 2.5, 1.8, 1.6, and 1.6 respectively (tie between t3 and t4 broken based on their lower bound scores). Since K=3, the algorithm computes the exact scores of the top 3 objects in UB, i.e., t1, t2 and t3. Their exact scores evaluate to 2.2, 1.6, and 1.6 respectively. We go back to step 2; the top K objects in UB are again {t1, t2, t3} and their exact scores are already computed. So the final top-K results are {t1, t2, t3}. ■

Pruning_Exact_Scores compute the exact scores of the minimum possible number of candidates. The intuition is that no exact pruning strategy can isolate the final top K objects without computing the exact scores of the candidates that are in the top K in UB at any stage of the algorithm.

Theorem 1: Given a set of candidate target objects with correct upper and lower bound scores, no exact scores pruning strategy can isolate the final top K with fewer exact score computations than *Pruning_Exact_Scores*.

5.2.2 Exact Top-K with Approximate Scores

In some applications, it might be sufficient to return the top K target objects even if their scores are not exact. In such cases, we can be more efficient than *Pruning_Exact_Scores* by computing the exact scores of fewer candidates. For example, consider the candidate t1 in Example 5.1. Since its lower bound score (2.0) is greater than the $(K+1)^{th}$ highest upper bound score (1.4), it is guaranteed to be in the final top K. Hence, we do not need to compute its exact score. Note that it was not possible to avoid such exact score computations in *Pruning_Exact_Scores* because we wanted their exact scores. We first identify the set of candidates whose score *must* be computed to isolate the final top K target objects. These are classified into *crossing* and *boundary* candidates.

Definition 5.1 (Crossing Objects): A target object is *crossing* if its rank in LB is more than K and its rank in UB list is K or less.

Definition 5.2 (Boundary objects): A pair of target objects (A, B) is called *boundary* objects if the exact scores of neither A nor B has been computed, and before their exact score computation:

1) The top K objects in UB and LB are same (i.e., there are no crossing objects)
2) A is the Kth object in LB list and u^{th} object in UB list ($u \leq K$)
3) B is the $(K+1)^{th}$ object in UB and l^{th} object in LB list $l \geq (K+1)$
4) $LB_K < UB_{K+1}$ ■

It is sufficient to iteratively compute the exact scores of the crossing and boundary objects (first A and then, if necessary, B) until these sets are empty for identifying the top K target objects. The intuition is as follows. Recall from Section 5.1 that the necessary and sufficient condition for identifying the final top K is $LB_K \geq UB_{K+1}$ and Top(LB, K) = Top(UB, K). Note that this condition is not satisfied if there are crossing objects or boundary object pairs. Computing exact scores of non-crossing objects cannot change the status of a crossing object *t* because it can only lower the rank of other objects in UB list (i.e, can only raise t's rank in UB list) and raise the rank of other objects in LB list (i.e., can only lower t's rank in LB list). Only computing the exact score of *t* can change its status; hence exact scores of crossing objects must be computed for the stopping condition to hold. Similarly, computing exact scores of non-boundary objects can only raise the ranks of boundary objects in UB list and lower their ranks in LB list. Hence they can either stay as boundary objects or at least one of them becomes crossing (in which case we *must* compute its exact score).

Reducing the number of documents retrieved: We can be more efficient by updating the bounds of crossing and boundary candidates based on the documents retrieved so far and checking the stopping condition in Step 3 instead of computing their exact scores right away. Thus, we retrieve just enough additional documents from the lists L_i necessary to satisfy the stopping condition (instead of retrieving all of them as in *Pruning_Exact_Scores*), thereby saving FTS costs.

6 DISCUSSION

In this section, we discuss three important issues: the handling of selection predicates, the choice of aggregation functions, and the application to other types of ranked search.

Selection predicates on documents: We assume that the ranked lists L_i of documents contain only the objects that satisfy the selection condition: either by pushing the selection to FTS (if it supports) or by filtering the documents returned by FTS. Our basic flow of the algorithms remains unchanged. The bound computation however may have to be modified when frequency target materialization (FTM) is used. Note that for the materialized target objects, we cannot use their materialized exact scores as the lower bound scores, so the lower bound score is initialized to 0 and updated during candidate generation like the non-materialized target objects. We can use their materialized scores as upper bound scores but they could be weaker because of the presence of selection predicates. Therefore, we also compute their upper bound scores during candidate generation like the non-materialized objects and use the less of the two. Note that the bound θ becomes weak because the actual frequencies in the ranked list are lower due to the selection. This may result in weaker upper bound scores for very selective conditions (cf. Figure 16).

Selection predicates on target objects: For selection predicates on target objects, we apply an additional filter at the candidate generation step. We could, in principle, apply it while joining the

Keywords	Desired Entity Type	Top 5 Results
New York Yankees	Person	Joe Torre, Alex Rodriguez, Derek Jeter, Gary Sheffield, Hideki Matsui
Boston Red Sox	Person	David Ortiz, Curt Schilling, Manny Ramirez, Terry Francoma, Jason Varitek
Los Angeles Lakers	Person	Shaquille Oneal, Phil Jackson, Kobe Bryant, Karl Malone, Gary Payton
Google executives	Person	Larry Page, Sergey Brin, Eric Schmidt, Marissa Mayer, David Garrity
Wimbledon champion 2004	Person	Maria Sharapova, Roger Federer, Serena Williams, Andy Roddick, Lindsay Davenport
Mobile phones	Organization	Nokia, Motorola, T-Mobile, Sprint, AT&T Wireless

Table 2: Examples of OF queries and their results

ranked list of documents with the relationships table R. But, that forces a join with the target objects table T as well. We have not explored this option in our prototype.

Choice of aggregation function: The materialized scores for frequent target objects may be useful even if F_{agg} specified at query time is different from that used for materialization.

- If we materialize the scores using the SUM aggregation function, we can use the materialized scores as the upper bound scores for the class of SUM_TOP_D functions.
- If we also materialize, for each keyword, the frequencies of the frequent target objects in the corresponding ranked list, we can use these frequencies (for bounding the B value) to compute the upper bounds for the materialized objects for any subset monotonic aggregation function.

Other types of ranked search: Our techniques apply beyond keyword search paradigms involving FTS, and both our scoring functions and evaluation techniques apply to other types of ranked searches (e.g., multimedia search [10, 11], ranked search on structured attributes [2, 6]). For example, the search objects can be homes where each home has price and neighborhood information associated with it. Suppose there is a 'ranking subsystem' that supports ranked search on price, i.e., returns the ranked list of homes that best 'match' a given price. An application might want to find the top neighborhoods that have homes with price similar to $350K; we can answer such queries using our techniques. FTS is substituted with the appropriate ranking subsystem which generates the ranked lists. Our techniques can subsequently be used.

7 EXPERIMENTS

We now present the results of an extensive empirical study to evaluate the techniques described in this paper. We conduct our experiments in the context of the entity finder application presented in Example 1.1 over a large collection of news articles, using keyword queries from "Google top sports queries". The major findings of our study can be summarized as follows:

1. **Faster than SQL**: The Generate-Prune approach is 4-5 times faster than the SQL implementation for small values of K (≤ 25) and about 2-3 times faster for larger values of K (25-100).
2. **Faster than Generate-Only**: The Gen-Prune approach significantly outperforms the Generate-Only approach.
3. **Robust to number of keywords and selections**: The Generate-Prune approach is robust to the number of keywords and selection conditions on documents
4. **Intuitive Results**: Using anecdotal evidence on a small sample of queries, we show that the scoring functions we instantiate produce meaningful results for OF queries.

All experiments reported in this section were conducted on a Compaq XW8200 dual-processor machine with 2 XEON 3.2 GHz processors and 2.5 GB RAM, running Windows 2003 Server.

7.1 Experimental Methodology

Dataset and Preprocessing: Our documents comprise of a collection of 714,192 news articles from 2003-2004 which we obtained from MSNBC news portal. We index those news articles inside SQL Server FTS engine so that we can get ranked lists of documents for keyword queries using SQL. We extract 3 types of named entities, viz. PersonNames, OrganizationNames and LocationNames, from the news articles using a Named Entity Extractor tool; these entities are our target objects. The tool extracted 435,838 PersonNames, 93,256 OrganizationNames and 158,246 LocationNames from the above collection. We store the entities and relationships of each type in separate target object and relationships tables. The relationships tables for PersonNames, OrganizationNames and LocationNames have 4,118,256, 798,956 and 3,078,421 tuples respectively. In order to study the benefit of frequent target materialization (FTM), for certain experiments we materialize the target objects with frequency above $\theta = 80$ for each keyword; this choice was based on allowed space overhead of 1%.

Queries: To get realistic OF queries, we picked the following top 10 sport news queries on Google in 2004 as reported on "Google Zeitgeist".

1) Dallas Cowboys	6) Los Angeles Lakers
2) New York Yankees	7) Philadelphia Eagles
3) Chicago Cubs	8) New England Patriots
4) Boston Red Sox	9) Green Bay Packers
5) Atlanta Braves	10) Oakland Raiders

We specify "PersonName" as the desired entity type for all the queries. All our measurements are averaged across the 10 queries.

Comparison: We have implemented all the 3 approaches to evaluate OF queries: SQL implementation, Generate-Prune approach and Generate-Only approach (abbreviated Gen_Prune and Gen_Only in plots). We compare these approaches against each other for both classes of scoring functions. For the row marginal class, we issue an 'AND query' to FTS. For the column marginal case, we use SUM as combination function F_{comb} in all the experiments. The experiments use SUM as the aggregation function unless otherwise mentioned; FTM optimization is used in these cases. We use chunk size $|C_i| = 100$. All the queries were run with a cold buffer cache.

7.2 Experimental Results

Quality of Answers: While a thorough user study is beyond the scope of this paper, we present anecdotal evidence that our OF query semantics and scoring functions produce intuitive results. Table 2 shows the top 5 results of some entity finder queries on the news collection. The scoring function used is a column marginal function with F_{agg}=SUM and F_{comb}=SUM. The results are, not surprisingly, quite meaningful.

Comparison with SQL: Figure 8 shows the execution times of the Generate-Prune approach and the SQL implementation for various values of K for both the column marginal and row marginal classes. For the column marginal class, the Generate-Prune approach is 4-5 times faster than SQL for small values of K (≤ 25) and about 2-3 times faster for larger values of K (25-100). This establishes that the early termination property of Generate-Prune leads to

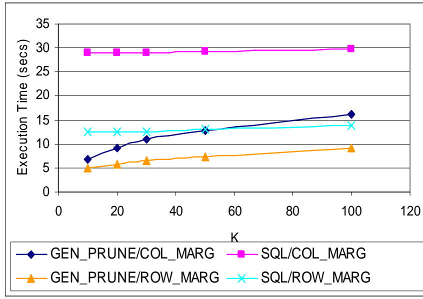


Figure 8: Execution times of GENERATE_PRUNE and SQL approaches for column and row marginal scoring functions

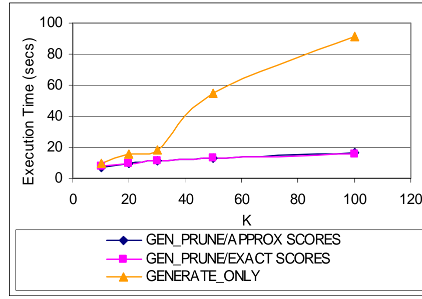


Figure 9: Execution times of GEN_ONLY, GEN_PRUNE/APPROX SCORES and GEN_PRUNE/EXACT SCORES

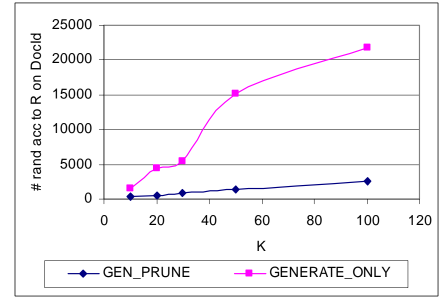


Figure 10: Number of random accesses to relationships table R on DocId by GEN_PRUNE and GEN_ONLY approaches

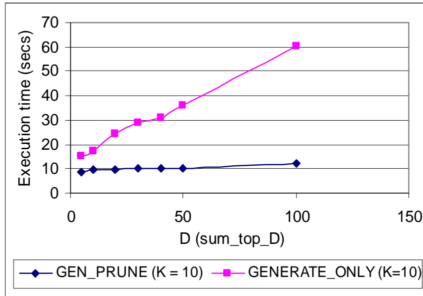


Figure 11: Execution times of GEN_PRUNE and GEN_ONLY approaches for various values of D (sum_top_D)

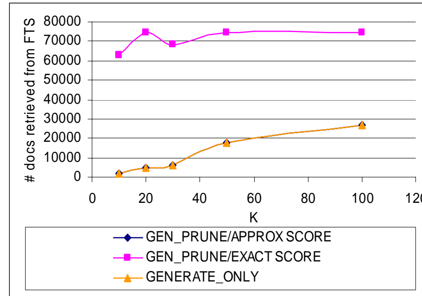


Figure 12: Number of docs retrieved from FTS by GEN_PRUNE/APPROX SCORES, GEN_PRUNE/EXACT SCORES and GEN_ONLY

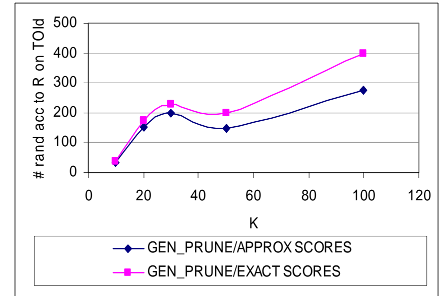


Figure 13: Number of rand. accesses to R on TOId by GEN_PRUNE/APPROX and GEN_PRUNE/EXACT SCORES

significant reduction of execution cost of OF queries. For the row marginal class, both approaches are faster compared to the column marginal case because the combination is pushed down into the FTS and documents are retrieved/looked up from a single ranked list. Even in this case, the Gen-Prune approach is 2-3 times faster compared to SQL.

Comparison with Generate-Only Approach: Figure 9 compares the Generate-Prune approach with the Generate-Only approach for the column marginal framework. The Generate-Prune approach significantly outperforms the Generate-Only approach for all values of K but the gap widens for larger values of K. This is because the Generate-Only approach ends up retrieving a large number of documents from the ranked lists and looking them up in the relationships table *R* (i.e., doing random access to *R* on DocId) in order to satisfy the "ideal" stopping condition.[6] The Generate-Prune approach, due to its relaxed stopping condition, retrieves much fewer documents during the generation phase and hence does much fewer random accesses to *R* on DocId. This is confirmed by Figure 10 which shows the number of random accesses to *R* on DocId for the two approaches. We observe this same performance gap between the two approaches even for small values of K when the cardinality bound B becomes weak. Figure 11 compares the two approaches for the SUM_TOP_D aggregation function for various values of D. We turn off the FTM materialization for this experiment; so the cardinality bound B = D. Even for K=10, Generate-Only approach rapidly degrades with increasing D; this is because it again ends up retrieving a large number of documents from FTS and looking them up in *R* to satisfy the "ideal" stop condition due to the weak upper bounds. The Generate-Prune approach, on the other hand, is robust due to the relaxed stopping condition. Note that the Generate-Prune approach has the additional cost of computing exact scores of candidates but that cost is small compared to the difference of cost in the generation phase.

Comparison between Exact Scores and Approximate Scores: Recall that the Generate-Prune technique returning the top K objects with approximate scores is expected to reduce cost in 2 ways: (a) compute exact scores of fewer candidate target objects and (b) retrieve fewer documents from the lists. Figure 13 shows the savings due to (a); the approximate scores approach compute exact scores of fewer candidates (by almost 25-50%). Figure 12 shows the savings due to (b); the approximate scores approach retrieves much fewer documents compared to exact scores which retrieve all documents (but do not lookup in the relationships table). However, the surprising result was that their execution times as shown in Figure 9 are almost identical. Investigating this anomaly, we found that the SQL UDF we are using to get the ranked lists from FTS for the various keywords actually gets the whole ranked list in one go. We confirmed this by varying the number of documents retrieved from FTS for various keyword queries and measuring the response times; the execution times are independent of the number of documents retrieved. Hence, the savings in the cost due to (b) is not reflected in the execution time. Furthermore, we found that retrieving the ranked list from FTS accounts for about half the execution time; the remaining time is evenly split between the generation and pruning phases. Therefore, in an FTS which does not retrieve the whole ranked lists in one go, we expect the approximate scores approach to be even better compared to all the other approaches including SQL.

Sensitivity to materialization: Figure 14 shows the execution times of the Gen-Prune approach for various values of θ. Lower the value of θ, more the number of frequent target objects materialized, better the upper bound scores, faster the execution.

[6] The Generate-Only approach sometimes performs worse than SQL although it does not retrieve any more search objects or do more joins on SOId than SQL. This is because SQL does the join in one go while Generate-Only does it in chunks, thereby incurring higher costs of communication with server, parsing costs, etc. The execution time of Gen-Only can be reduced by choosing the chunk size judiciously but is still no better than SQL.

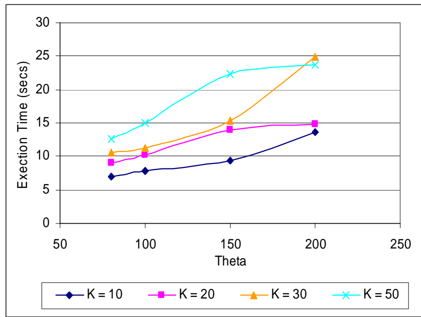


Figure 14: Sensitivity of GEN_PRUNE approach to number of frequent target objects materialized (θ)

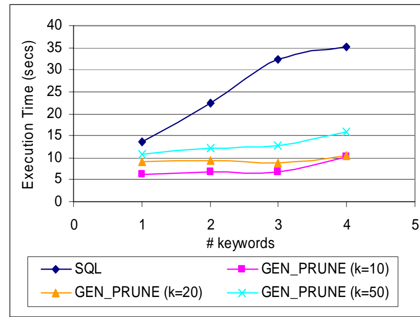


Figure 15: Sensitivity of GEN_PRUNE and SQL approaches to the number of keywords

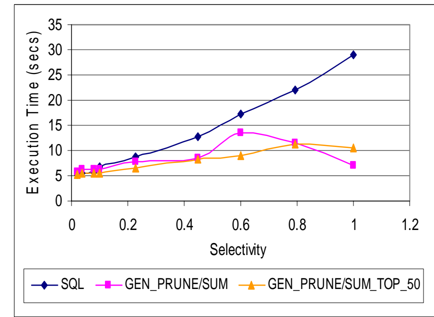


Figure 16: Sensitivity of GEN_PRUNE and SQL approaches to selectivity of selection conditions on search objects

Sensitivity to number of keywords: Figure 15 shows the execution times of the Gen-Prune approach and SQL implementation for different number of keywords in the OF query. We used the 2 and 3 keyword queries from the 'Google top sports queries' in addition to some 1 and 4 keyword queries from "Google Zeitgeist". The Gen-Prune is more robust to the number of keywords since it does partial retrieval on the ranked lists; SQL, on the other hand, has to retrieve more lists of documents and lookup more documents in the relationships table and hence becomes even more expensive.

Sensitivity to selections on documents: Figure 16 compares the execution times of the SQL implementation, Gen-Prune approach with F_{agg}=SUM (with FTM optimization) and Gen-Prune approach with F_{agg}=SUM_TOP_D (without FTM optimization) in presence of selection conditions on documents. We pose a range selection condition on the 'date' attribute of the news articles and vary its selectivity by changing the date ranges. For selectivity < 10%, the execution times are identical for the 3 approaches. This is because the cardinality bound B based on θ, although correct, is too weak in presence of selective search conditions; hence the Gen-Prune approach ends up retrieving as many documents from FTS as SQL. For selectivity > 10%, the Gen-Prune approach outperforms SQL because the bounds start getting stronger resulting in earlier terminations. The Gen-Prune with F_{agg}= SUM_TOP_D performs better than the F_{agg}=SUM case. This is because, in the former case, the bound B comes from D which is unaffected by selections while, in the latter case, it comes from θ which is weakened by selections. We observe that the Gen_Prune with F_{agg}=SUM has a bell-shaped curve because the weak bounds have the most impact when the selectivities are high but not high enough for the bounds to be tight.

8 CONCLUSIONS

In many applications, the goal is to find the top K objects related to documents that best match a set of keywords. We introduced the class of object finder queries and defined its semantics. We present two broad classes of scoring functions, which exploit relationships between documents and objects, to compute the relevance score of the target objects for a given set of keywords. Our query evaluation system would return the K target objects with the highest scores. We present early termination techniques to efficiently evaluate these queries. Our experiments show that our approach is 4-5 faster than the SQL implementation that does not have this early termination property.

9 ACKNOWLEDGEMENTS

We thank Raghav Kaushik for his insightful comments on our work. We thank Saliha Azzam and Kevin Humphreys of the Microsoft Natural Language Group for providing us the entity extractor and Gary Nease of MSNBC for providing us the news dataset.

10 REFERENCES

[1] S. Agrawal, S. Chaudhuri and G. Das. *DBExplorer: A System for Keyword Search over Relational Databases.* In Proc. of ICDE, 2002.

[2] S. Agrawal, S. Chaudhuri, G. Das and A. Gionis. *Automated Ranking of Database Query Results.* In Proc. of CIDR, 2003.

[3] R. Baeza_yates and B. Ribiero-Neto, *Modern Information Retrieval*, ACM Press, 1999.

[4] A. Balmin, V. Hristidis and Y. Papakonstantinou, *ObjectRank: Authority-Based Keyword Queries in Databases,* In Proc. of VLDB, 2004.

[5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti and S. Sudarshan, *Keyword Searching and Browsing in Databases using BANKS*, In Proc. of ICDE, 2002.

[6] S. Chaudhuri and L. Gravano, *Evaluating Top-k Selection Queries*, In Proc. of VLDB, 1999.

[7] S. Chaudhuri, R. Ramakrishnan and G. Weikum, *Integrating DB and IR Technologies: What is the Sound of One Hand Clapping?*, In Proc. of CIDR, 2005.

[8] J. Conrad and M. H. Utt, *A System for Discovering Relationships by Feature Extraction from Text Databases*, In Proc. of SIGIR, 1994.

[9] S. Dessloch and N. Mattos, *Integrating SQL Databases With Content-Specific Search Engines*. In Proc. of VLDB, 1997.

[10] R. Fagin, *Combining fuzzy information from multiple systems*. In Journal of Computer and System Sciences, 1999.

[11] R. Fagin, A. Lotem and M. Naor, Optimal Aggregation Algorithms for Middleware, In Journal of Computer and System Sciences, 2003.

[12] U. Guntzer, W. Balke and W. Kieβling, *Optimizing Multi-Feature Queries for Image Databases*. In Proc. of VLDB, 2000.

[13] V. Hristidis and Y. Papakonstantinou, *DISCOVER: Keyword Search in Relational Databases*, In Proc. of VLDB Conference, 2002

[14] I. Ilyas, W. Aref and A. Elmagarmid, *Supporting Top-k Join Queries in Relational Databases*. In Proc. of VLDB, 2003.

[15] D. Mattox, *Expert Finder*. MITRE Publications, *'The Edge'*, http://www.mitre.org/news/the_edge/june_98/third.html, Jun 1998

[16] S. Nepal and M. V. Ramakrishna, *Query Processing Issues in Image (Multimedia) Databases*, In Proc. of ICDE, 1999.

[17] Z.Nie, Y. Zhang, J. Wen and W. Ma, "Object-Level Ranking: Bringing Order to Web Objects", In Proc. of WWW, 2005.

[18] E. Voorhees, *Introduction to Information Extraction and Message Understanding Conferences*, http://www.itl.nist.gov/iaui/894.02/related_projects/muc/