Efficient Inverted Lists and Query Algorithms for Structured Value Ranking in Update–Intensive Relational Databases

Lin Guo
Jayavel Shanmugasundaram
Cornell University
{*guolin,jai*}*@cs.cornell.edu*

Kevin Beyer
Eugene Shekita
IBM Almaden Research Center
{*kbeyer,shekita*}*@almaden.ibm.com*

Abstract

We propose a new ranking paradigm for relational databases called Structured Value Ranking (SVR). SVR uses structured data values *to score (rank) the results of keyword search queries over text columns. Our main contribution is a new family of inverted list indices and associated query algorithms that can support SVR efficiently in update–intensive databases, where the structured data values (and hence the scores of documents) change frequently. Our experimental results on real and synthetic data sets using BerkeleyDB show that we can support SVR efficiently in relational databases.*

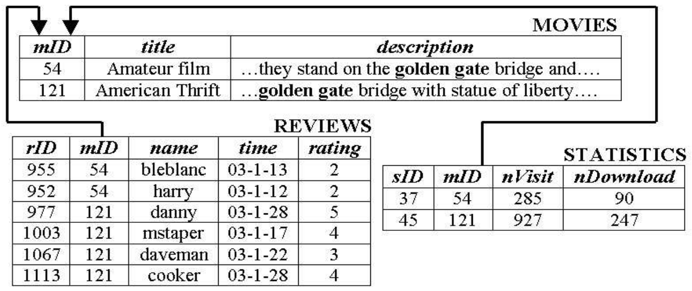
1 Introduction

SQL/MM [22] is the standard extension to SQL for processing text data. Current relational databases that implement SQL/MM [9, 10] treat the text management component as a "black box" and use traditional ranking techniques such as TF–IDF [25]. Such ranking techniques, however, were originally developed by the IR community for *stand–alone* document collections and often do not produce meaningful results for text stored in relational databases. As an illustration, consider the relational database in Figure 1, which contains fragments of data from the Internet Archive (http://www.archive.org) database, and the SQL/MM query below, which returns the top 10 movies whose description contains the keywords "golden gate". Using TF–IDF, the two movies in the database will have about the same score in the query result because the keywords "golden gate" occur the same number of times (i.e., once) in both of them and their text field lengths are about the same. In other words, the two movies will have similar scores despite the fact that they have very different popularities based on their review ratings and number of visits/downloads in the database.

```
SELECT    *
FROM      Movies m
ORDER BY  score(m.desc, "golden gate")
FETCH TOP 10 RESULTS ONLY
```

In this paper, we propose *Structured Value Ranking* (SVR), which uses the *structured data* values related to the text data in order to rank keyword search query results. In our example, using SVR, we can score a movie based on a combination of its (a) average reviewer ratings, (b) number of visits by users, and (c) number of downloads. Thus, the movie "American Thrift" will be clearly ranked above "Amateur Film" since it has received higher ratings, visits and downloads. One of the contributions of this paper is a SQL–based framework for specifying and integrating SVR in a relational database. A system administrator or an automatic ranking tool can use SQL queries to specify how text columns are to be scored based on the values of related structured data (such as average reviewer ratings). Our framework also enables ranking based on a *combination* of SVR and TF–IDF.

The main contribution of this paper is a new family of inverted list indices and query algorithms that can support SVR efficiently in relational databases. The main challenge in devising these indices is to efficiently support top–k queries when the scores of the text columns change frequently and possibly dramatically. Our initial motivation for this problem arose in the context of the Internet Archive database, where the reviewer ratings, number of visits and number of downloads were updated often and significantly changed the scores of the associated text columns (frequently, these changes are due to "flash crowds" on the Internet, where an item suddenly gains popularity due to some external event such as an award announcement [2]). From the user's point of view, they would still like to see the top ranked results based on the *latest* scores so that they do not miss important recent information. We believe that this is also true for a large class of other update–intensive databases where SVR is applicable, such as stock databases (where volume of trade can be used to rank results), online auctions such as e–bay (where time to completion and the current bid can be used to rank results) and a host of other web–based applications where the popularity of an item sometimes changes dramatically in a span of a few


MOVIES

mID	*title*	*description*
54	Amateur film	…they stand on the **golden gate** bridge and….
121	American Thrift	…**golden gate** bridge with statue of liberty….

REVIEWS

rID	*mID*	*name*	*time*	*rating*
955	54	bleblanc	03-1-13	2
952	54	harry	03-1-12	2
977	121	danny	03-1-28	5
1003	121	mstaper	03-1-17	4
1067	121	daveman	03-1-22	3
1113	121	cooker	03-1-28	4

STATISTICS

sID	*mID*	*nVisit*	*nDownload*
37	54	285	90
45	121	927	247

Figure 1. Example database

minutes [19].

Unfortunately, traditional inverted list indices and associated query processing algorithms are not designed to support top–k queries efficiently when document scores are updated frequently. Specifically, we cannot use regular top–k query processing algorithms such as the Threshold Algorithm [11] and its variants [23, 26] because these algorithms require the keyword inverted lists to be sorted (and processed) in document score order – when scores are updated often, maintaining the inverted lists in sorted order becomes very expensive. We thus propose a new family of inverted list indices that are maintained in *approximate* score order, and devise special query processing algorithms that *correct* for the score inaccuracy during query processing.

The proposed indices and algorithms can trade off query time for update time, can support both conjunctive and disjunctive keyword search queries, can support a combination of SVR and term scores (such as TF–IDF), and can be tightly integrated with a relational database by reusing relational features such as B+–trees. Further, the indices can also efficiently support incremental *document* insertions, deletions and updates. Our implementation using BerkeleyDB and our evaluation using both synthetic and real data sets indicate that the proposed indices can efficiently support SVR in update–intensive relational databases.

While the focus of this paper is on SVR in relational databases, we believe that the indices are more generally applicable. Specifically, they can be used in any top–k search application where the document scores change frequently.

2 Related Work

Our work differs in three main aspects from recent work on keyword search in relational databases, such as DBXPlorer [1], BANKS [4], and DISCOVER [16]. First, these techniques only use simple ranking schemes based on the number of joins [1, 16], or traditional IR ranking schemes such as TF–IDF [14] and variants of PageRank [3, 5]. In contrast, SVR exploits the rich semantics and relationships of *structured values* to rank keyword search results (in addition to supporting traditional IR–style ranking). Second, these techniques do not consider score updates, which is one of the main technical contributions of our work. Finally, these techniques only support "pure" keyword search, but can query over text data that spans multiple tables by automatically exploiting primary–foreign key relationships. In contrast, SVR is designed to be tightly integrated with SQL/MM, which deals with text data in a user specified table, but can support a mix of keyword search and structured queries. Other related work includes top–k parametric queries in relational databases [6, 7, 15], but these are not designed to support keyword search queries.

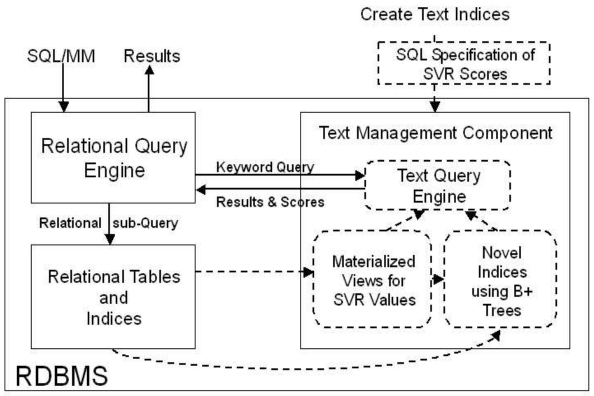

Create Text Indices
SQL/MM
Results
SQL Specification of SVR Scores
Relational Query Engine
Text Management Component
Keyword Query
Results & Scores
Text Query Engine
Relational sub-Query
Relational Tables and Indices
Materialized Views for SVR Values
Novel Indices using B+ Trees
RDBMS

Figure 2. System architecture

There is a vast body of work on devising inverted list indices [12, 25, 29] and associated query processing algorithms [11, 23, 24, 26, 28] for top–k queries. However, as mentioned in the introduction and discussed in further detail in Section 4, these approaches are not designed to process top–k queries efficiently when the document scores change frequently. There has also been some work on devising inverted lists that can efficiently handle *document* insertions, deletions and updates [18, 20, 27], but these are not designed to handle *score* updates.

3 System Architecture

One of our primary design goals was to tightly integrate SVR with a relational database. Towards this end, we build upon the architecture used by many commercial relational database systems for managing text data; this architecture is shown in Figure 2 (ignore the dashed lines and boxes for now). The text fields are indexed by a text management component (called "extender" in DB2, "cartridge" in Informix, and "data blade" in Oracle). Users can issue SQL/MM queries that contain *both* keyword search and other structured SQL sub–queries to the relational database engine. Given this joint query, the relational query engine first optimizes the query (rank–aware optimization can be used [17]). Then, during query evaluation, the relational query processor invokes the text management component with the query keywords to obtain the top–ranked (or all) text documents along with their scores, and merges these results with the other structured sub–query results.

In this paper, we focus on the text management component of the above architecture and show how it can be extended to efficiently support SVR. There are two main extensions that need to be made. First, we need to specify how the text keyword search results are to be ranked based on structured data values. Towards this end, we present a SQL–based framework for specifying SVR scores when creating a text index on a relational table (see dashed box in top right of Figure 2); system administrators or automatic ranking tools can use this framework to specify SVR scores for a given application. Second, we need efficient index structures and associated query processing algorithms that produce the top–ranked results for keyword search queries based on the *latest* structured data values. Towards this end, we devise new index structures that can be rapidly updated while still providing good performance for top–k keyword search queries. These indices can be implemented using traditional B+–trees (see dashed box in Figure 2) and can thus be easily integrated with a relational database.

We now describe our SQL–based framework for specifying and maintaining SVR scores using relational materialized views. In the next section (which constitutes the bulk of the paper), we describe our index structures and query processing algorithms.

3.1 SVR Score Specification

Consider a relation $R(C_1, C_2, ..., C_n)$ in which column C_t is the text column to be scored, and C_k is the primary key. Specification of the SVR score for C_t requires the specification of the following aspects:

1. $< S_1, S_2, ..., S_m >$, where each S_i is a SQL–bodied function [8] that takes in a C_k value (i.e., a primary key value of R), and returns the score for the text column. Intuitively, each S_i corresponds to one of the components of the SVR score.
2. $Agg(s_1, ..., s_m)$, where Agg is a SQL–bodied function that takes in m scores and returns a single aggregated score. Intuitively, Agg specifies how the scoring components are combined to obtain the final score.

Consider the example in Figure 1. Here, R is the *Movies* table, C_k is the *mID* column, and C_t is the *description* column. We can specify three scoring components using the SQL–bodied functions given below:

```
create function S1 (id: integer) returns float
return SELECT avg(R.rating)
       FROM   Reviews R WHERE  R.mID = id

create function S2 (id: integer) returns float
return SELECT S.nVisit
       FROM   Statistics S WHERE  S.mID = id

create function S3 (id: integer) returns float
return SELECT S.nDownload
       FROM   Statistics S  WHERE  S.mID = id

create function Agg(s1,s2,s3:float) returns float
return (s1*100 + s2/2 + s3)
```

S1 computes the average review rating for a movie, S2 computes the number of visits, S3 computes the number of downloads, and Agg computes the overall weighted score. As mentioned earlier, we also allow for the score to depend both on structured values and TF–IDF scores. In this case, the specification can also include the built–in scoring function *TFIDF()* (in addition to S1, S2 and S3). The Agg function can then aggregate all four scores as shown below (s4 is the TF–IDF score value).

```
create function Agg (s1,s2,s3,s4:float)
returns float
return (s1*100 + s2/2 + s3 + s4/2)
```

3.2 Efficient SVR Score Maintenance

One of the main challenges in using SVR scores is that the score values (which depend on structured data values) can change frequently and possibly dramatically. Fortunately, our SQL–based specification language allows us to leverage existing relational database infrastructure, specifically incrementally maintained materialized views [13], in order to maintain SVR scores efficiently.

Consider a relation R with primary key C_k, text column C_t, SVR score components $S_1, ..., S_m$, and aggregation function Agg. The following materialized view can be created for associating a score with each value of C_k (In case Agg contains the TF–IDF term, this term is not included in the materialized view specification, but is handled by the query algorithm as described in Section 4.3.3).

```
create materialized view Score as
   SELECT R.Ck, Agg(S1(R.Ck), ..., Sm(R.Ck))
   FROM   R
```

Since $S_1, ..., S_n, Agg$ are SQL–bodied functions, they are visible to the relational engine. Thus the view can be incrementally maintained [13] in the face of score updates.

4 Indexing and Query Processing

We now propose index structures and associated query processing algorithms that can support top–k queries efficiently even when document scores are updated frequently. Along the way, we also illustrate why traditional inverted lists are not appropriate for this scenario (and experimentally validate this claim in Section 5). Although our current focus is SVR in relational databases, the index structures are more general in scope and apply to any top–k keyword search application where scores change frequently.

4.1 Index Operations and Assumptions

We would like our index structures to efficiently support the following operations:

- **Document score updates:** Index structures should be able to handle frequent updates to document scores.
- **Top–k queries:** Index structures should be able to effi ciently evaluate conjunctive and disjunctive keyword search queries and return the top–k documents ordered by the latest values of their scores (may include IR–style term scores).

- **Content updates, insertions and deletions**. Index structures should be able to handle incremental content updates, insertions and deletions to documents.

In this paper, we focus on document score updates (because they are fundamental to SVR) and top–k queries (because this is a direct measure of query performance). Our index structures can also support incremental content updates, insertions and deletions. We describe these in Appendix A .

All of the described index structures can handle a combination of SVR scores and term scores (such as TF–IDF). The index structures can also support both conjunctive (i.e., return documents that contain *all* of the query keywords) and disjunctive (i.e., return documents that contain *at least one* of the query keywords) queries. However, to better illustrate the fundamental tradeoffs, we initially focus solely on SVR scores and conjunctive queries. Then, in Section 4.3.3, we show how the best of the indices (the Chunk method) can be extended to support a combination of SVR and term scores, and support both conjunctive and disjunctive queries.

We make the following assumptions. We assume that scores are non–negative real numbers. We assume that the SVR score for each text document is incrementally maintained using a materialized view $Score$ as described in Section 3.2. We also assume that the index structures are notified whenever the score of a document is updated in the materialized view (this can be done by the part of the code that actually updates the materialized view).

4.2 Traditional/Naive Inverted Lists

We introduce our index structures as refinements over traditional inverted lists, illustrating along the way why the traditional approaches are not applicable in our scenario.

4.2.1 ID Method

The ID method builds an inverted list for each term t, where each inverted list entry (posting) contains the ID of a document containing the term t. The postings for each term are sorted in increasing ID order. This data structure is one of the most commonly used in IR systems [12, 25]. In addition, a Score table is used to store the ID and score of each document (there is only one such table for the entire collection, not one per term). In fact this can be the same materialized view Score that stores SVR scores (see Section 3.2). An index is built on the ID column of this table so that score lookups by ID are efficient. Figure 3(a) shows an example inverted list for the term *news* and the Score table.

Score updates are very efficient in this method; when the score of a document is updated, we only need to update the score for the document in the Score table. Query processing, however, is more expensive. Given a query with terms $k_1, ..., k_n$, we need to merge the n lists corresponding to the query terms and determine the IDs of the documents that

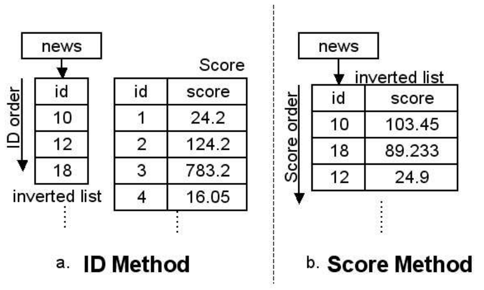


Figure 3. ID method and Score method

contain all of the query terms. For each such ID, we need to look up the Score table to determine the score of the associated document. Since users are usually only interested in the top–k results, a result heap is used to keep track of the top–k results during the scan. The main disadvantage of this method is that we need to scan all the postings in the query term inverted lists (which may be large) even if the user only wants the top–k results.

4.2.2 Score Method

A different way to organize inverted lists is to order postings by decreasing score instead of increasing ID – this ordering is required by efficient top–k query processing algorithms such as [11, 23, 24, 26, 28]. Here, each posting contains the document score in addition to the document ID since the inverted lists are sorted and merged in score order (note that we cannot simply use the document score as the document ID because scores can change in our setting). Figure 3(b) shows this inverted list organization for the term *news*.

Score updates are very inefficient in this method. When the score of a document d is updated, the new score has to be updated in the inverted list for *every distinct term* in d. This is likely to be very expensive because documents usually have hundreds to thousands of terms, and each update requires a random probe in the inverted list. In addition, since the inverted list for each term is ordered by the score, the ordering of the postings may have to be updated too. However, top–k queries can be processed very efficiently using the techniques proposed in [11, 23] since the inverted lists are in decreasing score order.

4.3 Novel Inverted Lists and Query/Update Algorithms

The ID and Score methods can be viewed as two ends of a spectrum. The ID method is very efficient for score updates but not for queries, while the Score method is very efficient for queries but not for score updates. Since score updates are relatively frequent and we still want to reduce query time, it is interesting to explore whether there are different update–query tradeoffs.

Devising index structures that provide such a tradeoff is a non–trivial task because of the following reason: we want the inverted lists to be ordered by the score (for query performance) but we still do not want to touch them for every score update (for update performance). We now present two novel index structures that address this apparent dilemma. These index structures can be implemented using regular relational tables and B+–trees, as described in Section 5.2.

4.3.1 Score–Threshold Method

The Score–Threshold method builds upon the Score method. There are two main ideas behind Score–Threshold method. The first idea is to allow the scores in the inverted list to be out–of–date by up to a threshold (hence the name Score–Threshold). Thus, not every score update requires an update to inverted list postings. Query processing, however, becomes slightly less efficient. Since the scores in the inverted list may be inaccurate by up to the threshold value, we may need to scan the inverted lists even after the first *k* results are found in order to correct for this inaccuracy. Note that the update–query tradeoff can be controlled by varying the size of the threshold.

The second idea behind this method addresses the following scenario: a document's new score exceeds its original score by *more* than the threshold value (say, for a newly popular document). This scenario is the main reason why we cannot directly use query algorithms like the Threshold Algorithm [11]. Specifically, since the change in score is potentially unbounded, any threshold function [11] has to be overly conservative thereby requiring a scan of the *entire* list for each query (since the document with the lowest score *could* have been updated to get the highest score).

Our solution to this problem is as follows. We maintain *two* inverted lists for each term. The first inverted list is the same as in the Score method and is never updated. The second inverted list has the same structure as the first, but only contains postings for those documents whose scores have been updated by more than the threshold value (hence, this inverted list is likely to be shorter and more efficient to update than the first inverted list). We note that while short inverted lists are routinely used in IR systems for efficient document insertions and deletions, our focus is to devise a new query algorithm that can deal with *score* updates. This leads to new problems because the scores in the inverted list can be inaccurate, and the same document can have different scores in the short and long inverted lists. Note that although the scores in the inverted lists may be inaccurate, our algorithm will nevertheless produce the correct top–*k* results based on the latest accurate score values.

Figure 4 shows the data structures used. For each term, there is a long and a short inverted list, both ordered by the score. Since the scores in the inverted list can be inaccurate by up to a threshold, a separate Score table (like in the ID method) is used to track the current accurate score of each document. In addition, a ListScore table contains an entry for each document whose score has been updated. Each entry contains the ID of the document, its score in the (short or long) inverted list, and a "inShortList" field; the "inShortList" field indicates whether the document has postings in the short list because its score changed by more than the threshold value.

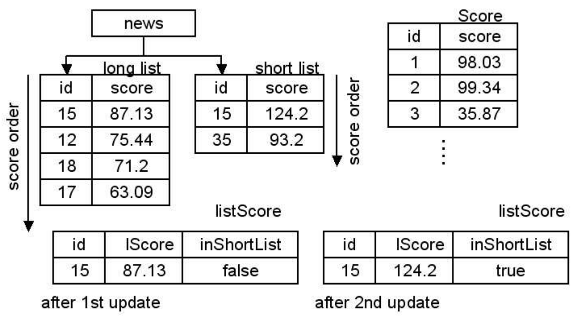


Figure 4. Score–Threshold method

Algorithm 1 shows how score updates are processed. Given a score update for a document d with ID id, the old score of d is retrieved, and the score is updated in the Score table (lines 7-8). We then determine the list score of d (i.e., the current score of d in the short or long inverted list). To do this, we first check whether the document d's score has ever been previously updated (line 9). If so, we look up the list score for d from the ListScore table (lines 10-12). If d's score was never previously updated, then we set its list score to be its old score (lines 14-16). Given the list score and the new score, we determine if d's score has increased by more than the threshold (the threshold is computed using the *thresholdValueOf* function – more will be said about this function soon). Only if the score has changed by more than the threshold do we add the new score to the short list. If d already contains postings in the short list, we update the scores of these postings (lines 20-22), else we add new postings for d in the short list (lines 24-26).

We now walk through an example in Figure 4. Suppose document 15 contains the term *news* and its initial score is 87.13. Further, assume that *thresholdValueOf*(87.13) = 100. First, if the document's score is updated to 91.4, its score is updated in the Score table and an entry is added to the ListScore table with the list score 87.13 and flag inShortList set to false. However, postings for this document with the new score are *not* added to the short list. Now if the document's score is updated again to 124.2, this is updated in the Score table. Now, since the new score is greater than *thresholdValueOf*(87.13) (where 87.13 is the list score), postings for the document with the new score are added to the short lists. The ListScore table is updated with the new value of the list score (124.2) and records the fact that postings for the document are now in the short lists.

Algorithm 2 shows how queries are processed. Consider a query of n terms $t_1, ..., t_n$, with long lists $LL(t_1), ..., LL(t_n)$ and short lists $SL(t_1), ..., SL(t_n)$. We use the notation $LL(t_i) \oplus SL(t_i)$ to denote the logical union of the postings of $LL(t_i)$ and $SL(t_i)$ in score order. We do

Algorithm 1 : `ScoreUpdate`(id, newS)

```
1: id : document ID of the updated document;
2: newS: new score of the updated document;
3: oldS: old score of the updated document;
4: Content(id): text content of the updated document;
5: SL($t$): the short list of term $t$
6:
7: oldS = Score.$getScore(id)$;
8: Score.$updateScore(id, newS)$;
9: if ListScore.$hasValue(id)$ then
10:    entry = ListScore.$lookupEntry(id)$;
11:    lScore = entry.lScore;
12:    inShortList = entry.inShortList;
13: else
14:    lScore = oldS;
15:    ListScore.$insert(< id$, oldS,false $>)$
16:    inShortList = false;
17: end if
18: if newS $>$ thresholdValueOf(lScore) then
19:    if inShortList then
20:       for each term $t$ in Content(id) do
21:          $SL(t).update(< id$, newS $>)$;
22:       end for
23:    else
24:       for each term $t$ in Content(id) do
25:          $SL(t).insert(< id$, newS $>)$;
26:       end for
27:    end if
28:    ListScore.$update(< id$, newS,true $>)$
29: end if
```

Algorithm 2 : `Query`($t_1, ..., t_n$, k)

```
1: $t_1, ..., t_n$ : query terms;
2: $k$: desired number of results;
3: $SL(t)$: the short list of term $t$;
4: $LL(t)$: the long list of term $t$;
5:
6: threshold = -1; // score has non-negative value
7: while $true$ do
8:    Merge $(SL(t_1) \oplus LL(t_1)), ..., (SL(t_n) \oplus LL(t_n))$ until
      find a candidate $d$ (with document ID $id$ and score $l_s$)
9:    if threshold $>$ thresholdValueOf($l_s$) then
10:      return;
11:   end if
12:   if $d$ is from $SL(t_1), ..., SL(t_n)$ then
13:      currScore = Score.$lookup(id)$;
14:      resultHeap.$add$($id$, currScore);
15:   else
16:      entry = ListScore.$lookupEntry(id)$;
17:      if entry == null or entry.inShortList == false then
18:         currScore = (entry==null)? $l_s$:Score.$lookup(id)$;
19:         resultHeap.$add$($id$, currScore);
20:      end if
21:   end if
22:   if enoughResult($k$, $l_s$) && threshold $< 0$ then
23:      threshold = $l_s$;
24:   end if
25: end while
```

a merge of the lists $SL(t_1) \oplus LL(t_1), ..., SL(t_n) \oplus LL(t_n)$ by scanning the short and long inverted lists for the query terms in parallel (line 8). For each document d that appears in all of the inverted lists, we have two cases. If d is a result due to postings in the short lists, then its current score is looked up in the Score table (recall that the scores in the inverted lists may be out of date), and it is added to the result heap (lines 12-14). If d is a result due to postings in the long list and if either d's score has not been updated or d is not in the short list (lines 16-17), then the latest score of d is obtained and d is added to the result heap (if d is in the short list, then the long inverted list postings can be ignored).

The interesting aspect of the algorithm is that it does not stop after the first k results are found. This is necessary because that scores in the inverted lists are not always accurate, and there may exist a posting further down the inverted list whose latest score is actually greater than the score of the current document. However, since we know that this inconsistency is bounded by a threshold, we only need to scan the inverted lists until we are sure that the current score of the posting cannot possibly exceed the lowest score in the top–k results (lines 9-11, 22-24).

As an example, consider the evaluation of a top–k query, where the k'th document added to the result heap has list score 100. Instead of stopping at this k'th result, the algorithm continues to scan the inverted lists until it reaches a posting for a document whose list score l_s satisfies the following property: *thresholdValueOf*(l_s) <100; this is the additional work that the query processing algorithm has to do since the list scores are not accurate.

We now turn to the choice of the *thresholdValueOf* function. It should be a monotonic function that satisfies the following property: *thresholdValueOf(score)*$\geq$ *score*. In other words, the threshold score should be at least as large as the original score. If *thresholdValueOf(score)* = *score*, then every positive score update causes the short list postings for the document to be updated (note that negative score updates would not require updates to the short list). On the other hand, if *thresholdValueOf(score)* = ∞, then the Score–Threshold method behaves similar to the ID method because postings would never be added to the short list and the entire inverted list would have to be scanned for every query. Of course, there are many choices for the *thresholdValueOf* function that fall in between. From our experiments, we found that using *thresholdValueOf(score)* = r $\times$*score* for some constant $r \geq 1$ worked well. We call r the *threshold ratio*.

Theorem 1:(Correctness of top–k Search) Algorithm 2 produces the top–k query results based on the latest values of the document scores. (Please see Appendix B for the proof.)

4.3.2 Chunk Method

Although the Score–Threshold method offers a trade–off between update and query performance, it suffers from one drawback compared to the ID method: Score–Threshold requires scores to be stored in the long inverted lists because query merging is done in score order. This requires larger inverted lists (because scores are replicated with each term in a document), which could negatively impact performance[1]. We now introduce the Chunk method that avoids storing scores in the inverted lists while still offering the desired update–query tradeoff.

The main idea is to divide the document collection into “chunks” based on the original document scores. For example, if there are 10000 documents, the lowest 5000 documents (based on score) could be in the first chunk, the next higher 3000 documents could be in the second chunk, and the top 2000 documents could be in the third chunk. Thus, documents in higher chunks always have higher scores than documents in lower chunk (at least, before score updates). Now, the key idea is to organize the inverted lists so that *within* a chunk, postings are in document ID order. Thus, during query time, we first merge all the documents in the last (third) chunk in ID order, then merge all the documents in the previous (second) in ID order, and so on until we find the top–k results at the end of some chunk. Note that since we do not merge based on the scores, we do not need to store the scores in the inverted lists; the scores only have to be stored in the Score table.

The other issue that needs to be addressed is when to add/update postings for a document in the short list. A simple solution is to add/update postings only when the score of a document moves into the boundary of a higher chunk (since within a chunk, the scores do not matter as the documents are ordered by ID). However, this creates a problem with boundary cases. Specifically, a small score update to the document with top–most score within a chunk can easily move the document into the next higher chunk, thereby causing its postings to be updated in the short list. We thus employ the strategy that a document's postings in the short list is updated only when its score causes it to move up *two* chunks. This avoids corner cases like in the above example. Note, however, that this has implications for query processing. We can no longer stop at the end of a chunk after we have found the top–k results; we need to scan an additional chunk to compensate for the inaccurate chunk IDs in the inverted lists (similar to inaccurate scores in the Score–Threshold method).

[1]Note that storing the scores in a separate lookup table instead of in the long inverted lists is not an efficient option either because this would require a *random* score lookup (as opposed to more efficient sequential access in Score–Threshold) for *each* processed posting in the inverted lists.

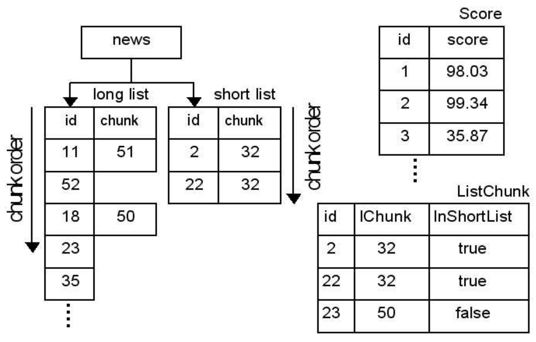


Figure 5. Chunk method

The Chunk method has the data structures shown in Figure 5. There is one long and one short inverted list for each term. However, unlike the Score–Threshold method, the postings are first ordered by decreasing chunk ID (or CID), and within each chunk, are ordered by increasing document ID. Note that we only have to store the CID at the beginning of a chunk, and not with each posting. This method also has a ListChunk table (analogous to the ListScore table in the Score–Threshold method) and the Score table.

The update and query processing algorithms are similar to that for the Score–Threshold method (Algorithms 1 and 2). The main difference is that the threshold function is now specified in terms of chunks instead of scores. Specifically, for a chunk c, *thresholdValueOf*(c)=c+1, which indicates that a document's postings in the short list need to be updated only if the score exceeds more than one chunk boundary. Since we use chunks instead of scores, the *newS* in Algorithm 1 is replaced by *newChunk* and *oldS* is replaced by *oldChunk*, and l_s in Algorithm 2 is replaced by the CID. We can also prove a theorem similar to Theorem 1 about the correctness of this algorithm.

The main tradeoff between update and query processing in this method comes about by setting chunk boundaries. If the chunks are very large, it behaves like the ID method with very little update overhead but slower query processing due to scanning large chunks. If the chunks are very small, however, it will have to update the short list postings on every score update, although queries will be faster. We experimented with various methods for specifying chunk boundaries (including equal sized chunks, exponentially growing/shrinking chunks) and determined that a good strategy was to set the chunks based on the actual score distribution. Specifically, we found that it was usually best to set chunk boundaries so that for two adjacent chunks $c+1$ and c, the ratio of the lowest score in $c+1$ to the lowest score in c is a constant r ($r \geq 1$). We call r the *chunk ratio*, and this is similar to the threshold ratio for the Score–Threshold method. Under very skewed score distributions, some chunks have only a few documents in them. So, we also set a minimum size of a chunk so that each chunk has at least 100 documents (or some other constant). We evaluate the impact of different threshold and chunk ratios in the experimental section.

One point to note is that although the Chunk method has the advantage of smaller long inverted lists, the Score–Threshold method has the advantage that its ratio can be changed without having to regenerate the long inverted lists. However, this is likely to be useful only when the score update distribution is unknown.

4.3.3 Chunk–TermScore Method

Thus far, we have focused solely on SVR scores and conjunctive keyword search queries. We now show how the Chunk method can be extended to support a *combination* of SVR and term–based scores such as TF–IDF, and support both conjunctive and disjuctive queries (the generalization for the Score–Threshold method is similar).

More formally, consider a result document d that contains all (conjunctive query) or some (disjunctive query) of the query terms for a keyword search query Q=$\{t_1$,...,$t_m\}$. Let the SVR score of d be $S_{svr}(d)$ and its term score be $f_t(d)$ for each term $t \in Q$. f_t could represent, say, the TF–IDF score (note that the SVR score of d is independent of the query terms, while the term score depends on the query terms). We consider the following combination scoring function F: $w \times S_{svr}(d) + \Sigma_{t \in Q} f_t(d)$ (although our technique generalizes to any monotonic F).

In designing the Chunk–TermScore method, we build upon the Fancy–ID method recently proposed by Long and Suel [21] for efficiently combining document global scores (such as PageRank) with term–based scores. However, their method assumes that the long inverted list is sorted based on the global score and hence, cannot support efficient score updates. Thus, our contribution is showing how this techniques can be adapted to work with the Chunk method to efficiently support score updates.

The data structures for the Chunk–TermScore method are similar to the Chunk method, with two changes. First, each posting in the long and short inverted lists also contains the term score (such as the normalized TF score) in addition to the ID. Second, each term has an additional ID ordered inverted list called the fancy list [21], which is a small list of postings that have the highest term scores for that specific term. The score update algorithm for the Chunk–TermScore method is the same as the Chunk method. However, the query processing algorithm has to be adapted to account for the new scoring function.

Algorithm 3 shows the query processing algorithm. Given a query, the basic idea is to first merge the fancy lists corresponding to the query keywords and determine the IDs of the documents that contain *all* of the query keywords (even for disjunctive queries). These IDs, along with their corresponding combined scores are tentatively added to the result heap because they are highly likely to be the top–k results due to their high term scores (line 8). In addition, the IDs that appear in some but not all of the query keyword fancy lists are added to a data structure called the remainList (line 9). The intuition is that the IDs in the remainList have a high term score for at least one query term, and could thus still make it to the top–k results. Once the fancy lists are merged, query processing proceeds similar to the Chunk method by merging the short and long inverted lists (line 11). Once an ID appears in any of these lists, it no longer needs to be remembered in the remainList since it is currently being processed and is thus removed (line 12). If the ID contains all (conjunctive queries) or some (disjunctive queries) of the query keywords, its score is computed using a combination of SVR and term scores, and added to

Algorithm 3 : `Query`($t_1, ..., t_n$, k)

```
1:  $t_1, ..., t_n$ : query terms;
2:  $k$: desired number of results;
3:  $SL(t)$: the short list of term $t$;
4:  $LL(t)$: the long list of term $t$;
5:  $FL(t)$: the fancy list of term $t$;
6:  $f_{min}(t)$: minimum term score in $FL(t)$;
7:
8:  Merge $FL(t_1), .., FL(t_n)$ and put candidate documents and
    their scores into resultHeap;
9:  remainList = $FL(t_i) \cup ... \cup FL(t_n)$ − resultHeap;
10: while $true$ do
11:     Merge $(SL(t_1) \oplus LL(t_1)), ..., (SL(t_n) \oplus LL(t_n))$ to find
        next $d$ (with document ID $id$, chunk CID and term scores
        $f(id, t_1), ..., f(id, t_n)$ where $d$ contains at least one term,
        i.e. $\exists i.f(id, t_i) > 0$);
12:     remainList.$remove(id)$;
13:     if $d$ contains some/all terms $t_1, ..., t_n$ then
14:         $S_{term} = \sum_{i=1}^{n} f(id, t_i)$;
15:         if $d$ is from $SL(t_1), ..., SL(t_n)$ then
16:             $S_{svr}$ = Score.$lookup(id)$;
17:             resultHeap.$add(id, F(S_{term}, S_{svr}))$;
18:         else
19:             entry = ListScore.$lookupEntry(id)$;
20:             if entry==null or entry.inShortList == false then
21:                 $S_{svr}$ = Score.$lookup(id)$;
22:                 resultHeap.$add(id, F(S_{term}, S_{svr}))$;
23:             end if
24:         end if
25:     end if
26:     if reach the end of chunk with id CID then
27:         $S_{svr}^{max}$ = thresholdValueOf(CID);
28:         remainList.$prune(S_{svr}^{max})$;
29:         if remainList.$empty()$ then
30:             $S_{term}^{max} = \sum_{i=1}^{n} f_{min}(t_i)$;
31:             if $F(S_{term}^{max}, S_{svr}^{max}) \leq$ resultHeap.minScore($k$)
32:             then return; endif
33:         end if
34:     end if
35: end while
```

Data	Num of tuples = 50k, **100k**, 150k	Score zipf factor =0.25, 0.50, **0.75**
Upd	Mean upd size = **100**, 1000, 10000	Focus set upd (%) = 0, **10**, 20, 40, 80
	Focus inc upd = 0, 50%, **100%**	Focus set size = 1%
Qry	Num of terms = 1, **3**, 5	Terms selectivity = **unsel**, medsel, sel
	Num of results = 1, **10**, 100, 1000, 10000, 100000	
	Type of queries = **conjunctive**, disjunctive	

Figure 6. Experimental parameters

the result heap as for the Chunk method (lines 13-25).

At the end of each chunk, we need to determine whether we need to continue further to find the correct top–k results. To do this, we first prune the remainList to remove the IDs of documents that cannot be in the top–k results (using the condition presented in [21]). If remainList is non–empty, then we continue processing because the IDs in the remainList could potentially be among the top–k results. However, if remainList is empty, then we can stop processing so long as the scores of the IDs in the remaining chunks cannot exceed the current top–k scores (lines 29-33).

Theorem 2:(Correctness of top–k Search) Algorithm 3 produces the top–k query results based on the latest scores computed using the combination scoring function F.

5 Experimental Evaluation

We now experimentally evaluate the performance of different methods described in the previous section.

5.1 Experimental Setup

We used two primary evaluation metrics: the time to update an inverted list due to a score update, and the time to evaluate a top–k keyword search query. We do not measure the time required to merge the short inverted lists (in the methods that use this data structure) with the long inverted lists because this is done offline and does not impact the performance of the operational system.

We generated synthetic data sets using the parameters shown in the first row of Figure 6 (default values are in bold face). The generated data table is *R(Id, StructuredColumn, TextColumn)*, where *Id* is an integer primary key, *StructureColumn* is a 100 byte column that simulates the presence of structured data columns, and *TextColumn* is a text document. The total number of distinct terms in the data set was 200000, which is approximately the number of terms in the English language. Each text document contains 2000 terms (possibly duplicates) and the term frequency follows the Zipf's law with parameter 1.0 (as in English). For the default settings, the total size of the database was approximately 805MB. In addition to R, we also generated a score table: *Score(Id, score)*. The value of *Score* ranged from 0 to 100,000, and the scores were generated using the Zipf distribution with default parameter 0.75; this zipf parameter is the same as what we experimentally observed in the real Internet Archive data set using the SVR specification in Section 3.1. For the real data set, we used the Internet Archive database. The total data set size was 60MB, and the two tables with indexable text columns had a total size of 10MB. Most of the experiments reported here use the synthetic data set where we could vary various parameters.

We studied three classes of keyword search queries: unselective queries in which the keywords were randomly chosen from the 350 most frequent terms; medium–selective queries were randomly chosen from the top 1600 most frequent terms, and selective queries were chosen randomly from the top 15000 terms. We varied the number of top–ranked results to be returned for each query.

The score update workload followed a Zipf distribution, whereby documents with higher scores were updated more frequently; this is consistent with the update logs in the Internet Archive. The *mean update size* controls the size of a score update; a value of 100 implies that the score of a document increases or decreases by 100 on the average, with the distribution of the update size varying uniformly between 0 and 200 (twice the mean). Score increases and score decreases are equally likely. We also model updates to a sub–set of the documents called the *focus set*, which is expressed as a percentage of the collection. The focus set contains documents that temporarily receive a lot of attention, independent of their actual current score. This reflects newly popular documents, such as a song that recently made it to the top–5 list (other research shows that many such scenarios occur on the Web [19]). The *focus set update* reflects that percentage of score updates that go to one of the focus set documents. The *focus increase update* controls whether the focus set updates are strictly increasing (default), strictly decreasing, or strictly increasing for 50% of the documents and strictly decreasing for the other 50%.

5.2 Inverted List Implementation

We implemented the five inverted list structures described in Section 4 on top of BerkeleyDB. As a basefile for comparison with Chunk–TermScore, we also implemented ID–TermScore, which is an extension of the ID method to additionally store term–based scores. To ensure a fair comparison with the base–line methods, we included various optimizations for the ID, Score and ID–TermScore methods as described in [29], including early termination methods and merging starting from the shortest lists.

The long inverted lists were stored as binary objects in the database since they are never updated; they were read in a page at a time during query processing. For the Score method alone, since the long inverted list is updated, it was implemented as a clustered B+–tree. The short lists, ListScore and ListChunk were implemented as tables with B+–tree indices built on the appropriate columns. The tables R and $Score$ also had a B+–tree index on the *Id* columns.

Table 1 shows the size of the long inverted list for different methods. The Score method has the largest space requirement because its inverted list needs to be updated; it

Method	**IL Size**	**Method**	**IL Size**
ID	145MB	ID–TermScore	428MB
Score	2,768MB	Score–Threshold	847MB
Chunk	146MB	Chunk–TermScore	430MB

Table 1. Size of Long Inverted Lists

thus suffers from the associated indexing and storage overhead in BerkeleyDB. The Score–Threshold method stores both the document ID and document score in the inverted list, for *each term* in the document; hence it suffers from additional overhead compared to the ID method, which does not store scores in the inverted list. The ID method also gets additional compression due to differential encoding of IDs since the postings are in ID order. The Chunk method has roughly the same space overhead as ID, but has a small additional overhead for storing the chunk ID once for each chunk. The size of the inverted list varies slightly for different chunk ratios, but the difference is not significant.

All our experiments were run on a 2.8 GHz Pentium IV processor with 1GB of main memory and 80GB of disk space. The size of the BerkeleyDB cache was set to 100MB. For updates, we report the total update time divided by the number of updates. Queries were run after all the updates using a cold cache for the long inverted lists to simulate a non memory–resident data set, and were averaged over 50 independent measurements. Unless otherwise stated, for each experiment, we varied one of the parameters, and used default values for the rest.

5.3 Experimental Results

5.3.1 Threshold and Chunk Ratios

Recall that the Score–Threshold and Chunk methods offer knobs, the threshold and chunk ratios, respectively, which can be tuned to trade off query vs. update performance. Thus, to compare the performance of these methods with the others, we first need to determine appropriate ratios. The appropriate ratio for a given workload depends on the nature of the updates (small or large updates), the actual number of updates (before merging with the long inverted lists), and the score distribution (uniform or skewed). To quantify this tradeoff, in Table 2, we tabulate the performance of the Chunk method for various ratios when varying the size of the update. Note that each measurement is the average time *per operation* (not the total time).

The general trend shows that for an mean score update step of 100, the time to perform a single score update increases as the chunk ratio decreases. This is expected because larger chunk ratios imply larger chunks, which in turn implies that the probability of updating the short lists due to score updates is lower. The interesting aspect to note is that the update time first increases almost imperceptably until a ratio of 6.12, and then increases dramatically because the smaller chunk sizes cause a lot of updates to the short lists.

	Step 100		Step 1000		Step 10000	
Ratio	Upd	Qry	Upd	Qry	Upd	Qry
164.84	0.01	138.64	0.01	135.68	0.01	134.4
82.92	0.01	136.53	0.01	133.99	**0.01**	**132.3**
41.96	0.01	46.204	0.24	54.32	160.4	90.8
21.48	0.01	43.938	**0.25**	**45.85**		
11.24	0.12	39.512	34.45	57.25		
6.12	**0.19**	**35.37**	222.18	83.558		
3.56	0.76	32.774				
2.28	145.35	30.938				
1.56	277.54	30.572				

Table 2. Effect of Chunk Ratio (times in ms)

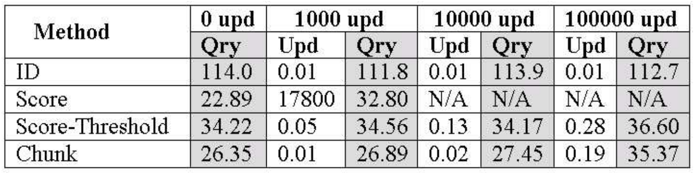
Query performance, on the other hand, decreases steadily as the ratio decreases. Thus, the optimal ratio for updates with mean step size 100 is around 6.12 (assuming the default score distribution and 100000 score updates).

When the mean step size is increased to 1000, we note that the optimal ratio increases, because the index has to tolerate more dramatic changes in the score; the optimal ratio in this case is around 21.48. In fact, even query time increases after this ratio because the lengths of the short lists increase rapidly. Note that the optimal query time with the mean update step size of 1000 is larger than step size 100 because the chunk sizes are larger for a larger ratio. A similar trend occurs when the mean size of the update is changed to 10000 (which is 10% of the entire domain of scores). Thus, the Chunk method essentially adapts to the update distribution, thereby allowing the appropriate query–update tradeoff. We also observe a similar tradeoff for the Score–Threshold method (figures not shown). For the rest of this section, unless otherwise stated, we fix chunk ratio at 6.12 and the threshold ratio at 11.24 (which is the optimal ratio for Score–Threshold using the default settings).

The observant reader would have noted that the time to perform queries is more than the time to perform updates. This is because query evaluation is performed on a cold cache of the long inverted lists to simulate a non memory resident data set. However, the Score table and the short lists are much smaller than the long inverted lists (the size of the Score table is only 2.7MB), and are easily maintained in the database cache. Since score updates for the Chunk method only access the Score table in most cases, it is faster than a query. This suggests that the Chunk method is likely to have a low overhead even in update–intensive databases.

5.3.2 Varying Number of Updates

Figure 7 shows the average update and query times, respectively, for the different methods when the number of updates is varied from 0 to 100000 (ignore ID–TermScore and Chunk–TermScore for now). The most striking thing to note is that the update performance of the Score method deteriorates dramatically because of the overhead of updating the long inverted lists. In fact, the cost per update is about 17



Method	0 upd	1000 upd		10000 upd		100000 upd	
	Qry	Upd	Qry	Upd	Qry	Upd	Qry
ID	114.0	0.01	111.8	0.01	113.9	0.01	112.7
Score	22.89	17800	32.80	N/A	N/A	N/A	N/A
Score-Threshold	34.22	0.05	34.56	0.13	34.17	0.28	36.60
Chunk	26.35	0.01	26.89	0.02	27.45	0.19	35.37

Figure 7. Varying # updates (times in ms)

seconds for Score method, as compared to 0.01 ms for the best methods! Since the performance of the Score method is always orders of magnitude slower than the best methods, we do not consider it further.

The ID method has the best update performance because score updates only require a single update in the Score table. However, its query performance suffers because it always scans the entire long inverted list even for top–*k* queries. The Score–Threshold and Chunk methods have the best overall performance because they avoid frequent updates to the short lists while still processing top–*k* queries efficiently. Of the two, the Chunk method has slightly better query performance because it has shorter inverted lists.

5.3.3 Varying Number of Desired Results

Figure 8 compares the query processing time for ID, Score–Threshold and Chunk methods when varying the number of desired top ranked results, *k*. As expected, the performance of the ID method remains roughly the same since it has to scan the entire inverted list regardless of *k*. In contrast, the performance of the Score–Threshold and Chunk methods is better with smaller *k* because they only scan the initial part of the inverted lists. When *k* is large, the performance of Chunk becomes the same as the ID method, while the performance of Score–Threshold is worse because it has the overhead of scanning larger inverted lists (that contain scores). Since Chunk always dominates Score–Threshold in this manner, we do not consider Score–Threshold further.

5.3.4 Varying Mean Update Step Size

Recall from Table 2 that the chunk ratio for the Chunk method needs to be set based on the expected magnitude of the score updates. Larger updates require larger chunk ratios. The interesting thing to note is that for a given update workload, the Chunk method with the optimal ratio for that workload always dominates or is very close to the ID method (the query performance of the ID method is always constant – about 114ms – regardless of the size of the updates). Thus, Chunk essentially adapts to the update distribution, thereby allowing for a query–update tradeoff.

5.3.5 Performance of Chunk–TermScore

So far we have focused on SVR scores in isolation. We now study the performance impact of including term scores using the Chunk–TermScore method. As a baseline for comparison, we compare with the ID–TermScore method, that is similar to the ID method but with term scores stored in

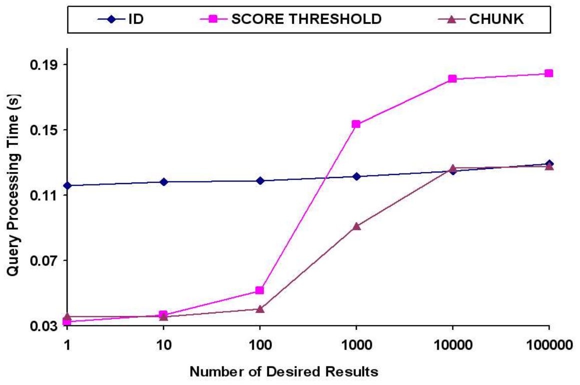


Figure 8. Varying # desired results

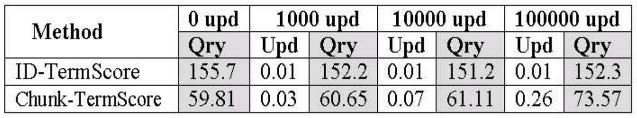
the inverted list so that it can compute the combined score. As shown in Figure 9, the query performance of Chunk–TermScore is significantly better than ID–TermScore (due to early stopping for Chunk–TermScore), while still having comparable update performance. Chunk–TermScore has slightly worse query performance than Chunk (in Figure 7) because Chunk–TermScore has larger inverted lists that store the term scores (Table 1) and also scans larger parts of the inverted list due to the combined scoring function. Note, however, that the query performance of Chunk–TermScore is even better than that of the ID method, which does not support term based ranking.

5.3.6 Performance of Disjunctive Queries

So far, we have focused on conjunctive queries. We now look at disjunctive queries. For the default parameter settings, the performance of the Score, Score–Threshold, Chunk and Chunk–termScore methods was only slightly better – less than 1ms better – for disjunctive queries than for conjunctive queries (results are thus not shown). The reason for this behavior is that even though disjunctive queries scan a smaller number of postings, they access about the same number of disk pages as conjunctive queries do (which is usually just the first page for each keyword) since multiple postings are packed into the same page. Since disk access dominates the evaluation time, the performance differnce is not significant. The performance of the ID and ID–TermScore methods, however, is worse for disjunctive queries (see Figure 10). The reason is that there are many more potential results in the disjunctive case, and for the ID methods, the overhead of processing these additional results in the result heap degrades performance.

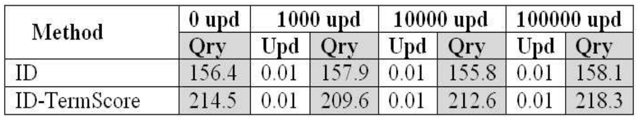
5.3.7 Summary of Other Results

We ran other experiments varying all the parameters described in Section 5.1. The conclusion was essentially the same: the update and query performance of the Chunk method was the best or close to the best. As mentioned earlier, we also ran experiments on the Internet Archive real data set. The original data set size was just 10MB of text



Method	0 upd	1000 upd		10000 upd		100000 upd	
	Qry	Upd	Qry	Upd	Qry	Upd	Qry
ID-TermScore	155.7	0.01	152.2	0.01	151.2	0.01	152.3
Chunk-TermScore	59.81	0.03	60.65	0.07	61.11	0.26	73.57

Figure 9. Combining term scores



Method	0 upd	1000 upd		10000 upd		100000 upd	
	Qry	Upd	Qry	Upd	Qry	Upd	Qry
ID	156.4	0.01	157.9	0.01	155.8	0.01	158.1
ID-TermScore	214.5	0.01	209.6	0.01	212.6	0.01	218.3

Figure 10. Disjunctive query results

data, and this was too small to illustrate the tradeoffs between the different approaches. So, we scaled up the data set by replicating the text data 10 times, and generating scores using the same distribution as the 10MB data set. The results that we obtained were very similar to those obtained using the synthetic data set.

6 Conclusion

We have introduced SVR, a new and alternative method for ranking keyword search queries in relational databases based on structured data values. We have also proposed new inverted list indices, notably the Chunk method, that can efficiently implement SVR in update–intensive relational databases. The Chunk method has a knob that can tradeoff query performance for update performance based on the application needs. In addition, an extension of the Chunk method (Chunk–TermScore) can support scoring using a combination of SVR and term scores (such as TF–IDF). Our experimental results show that SVR can be efficiently implemented in update–intensive relational databases.

7 Acknowledgments

We thank Jonathan Aizen and Brewster Kahle for giving us access to the Internet Archive data set. Lin Guo and Jayavel Shanmugasundaram were partially supported by the Cornell/AFRL Information Assurance Institute, NSF CAREER Award IIS-0237644, and an IBM Faculty Award.

References

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

[2] J. Aizen, D. Huttenlocher, J. Kleinberg, and A. Novak. Traffic-based feedback on the web. In *Proceedings of National Academy of Sciences*, 2004.

[3] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, 2004.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 1998.

[6] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. In *TODS*, 2002.

[7] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *VLDB*, 2004.

[8] C. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley Publishing Company, 1997.

[9] S. Dessloch and N. Mattos. Integrating sql databases with content-specific search engines. In *Data Engineering*, 2001.

[10] P. Dixon. Basics of oracle text retrieval. In *VLDB*, 1997.

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

[12] W. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[13] A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin*, 18(2), 1995.

[14] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, 2003.

[15] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, 2001.

[16] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[17] I. Ilyas, R. Shah, W. Aref, J. Vitter, and A. Elmagarmid. Rank-aware query optimization. In *SIGMOD*, 2004.

[18] N. Kabra, R. Ramakrishnan, and V. Ercegovak. The quiq engine: A hybrid ir-db system. In *ICDE*, 2003.

[19] J. Kleinberg. Bursty and hierarchical structure in streams. In *KDD*, 2002.

[20] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Agarwal. Dynamic maintenance of web index using landmarks. In *WWW*, 2003.

[21] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *VLDB*, 2003.

[22] J. Melton and A. Eisenberg. Sql multimedia and application packages(sql/mm). In *SIGMOD*, 2001.

[23] A. Natsev, Y. Chang, J. Smith, C. Li, and J. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, 2001.

[24] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. In *JASIS*, 1996.

[25] G. Salton. *Automatic Text Processing: The Transaction, Analysis and Retrieval of Information by Computer*. Addison Wesley, 1989.

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

[27] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *SIGMOD*, 1994.

[28] A. Vo, O. Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR*, 2001.

[29] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishing, 1999.

A Insertions, Deletions and Content Updates

We describe how the Chunk method can be extended to handle document insertions, deletions, and content updates. The extension for the other methods is similar.

A.1 Content Updates

To handle content updates to documents, we need to add an extra field to each posting in the short lists called *op*. The *op* field indicates whether a term has been added (ADD) or removed (REM) from the corresponding document.

Now consider an update to the content of a document d. Assume that the set of distinct terms in d before the update is T_{old} and the set of distinct terms in d after the update is T_{new}. The set of added terms is $T_{add} = T_{new} - T_{old}$ and the set of removed terms is $T_{del} = T_{old} - T_{new}$. For each term t_1 in T_{add}, we insert the posting $< d, ADD >$ (or update the existing posting to be $< d, ADD >$ if a posting for d already exists) in the short list for t_1. For each term t_2 in T_{del}, we insert (or update) the posting $< d, REM >$ in the short list for t_2.

During query time, when we union the long list l_t and the short list s_t for term t, we discard postings marked as REM in s_t. For example, when we see $< d >$ in l_t and $< d, REM >$ in s_t in the same chunk, we will not return d as a result because t was removed from d during a content update. ADD postings in s_t are treated like regular postings.

A.2 Insertions and Deletions

Document insertions are easily handled by the Chunk method because the insertion is simply treated as updates to the short lists, with the operation specified as ADD. Deletions are a bit more complex. First, we need to add a new field in the Score table that indicates whether a document with a given ID is deleted. Whenever the Score table is probed for a document during query processing, we do not add the document to the result heap if it is marked as deleted. If the relational system can reuse deleted IDs, we also need to delete all postings of the document from the short lists so that these are not interpreted as terms in a document that is later inserted with the same ID.

A.3 Experimental Results for Insertions

Table 3 shows how the query, score update and insertion update performance for the Chunk method varies with the number of document insertions (queries are timed right after the document insertions, so are score updates). The query performance remains robust even after 10000 document updates since the Chunk method effectively avoids having to scan all of the inverted list for top–N queries. Score updates performance degrades somewhat because of the increased length of the short lists (due to document insertions). However, the cost per update is still very low at 10-17 milliseconds because score updates do not require frequent updates to the short list for the Chunk method. Insertion performance is very fast for up to 2000 documents, but then degrades to about 0.5 seconds per document insertion for 4000 document and beyond (remaining stable at about 0.6 seconds after 4000 insertions). The reason for the degradation in performance beyond 4000 document insertions is that the size of the short lists increases significantly due to the document insertions. Note however, that even an insertion time of 0.5 seconds is still likely to be acceptable considering the fact that each document has 2000 terms that need to be indexed. In fact, we expect that *any* technique that supports incremental document insertions will incur a similar overhead because it will have to insert the 2000 terms into an inverted list, and if the inverted list does not fit into memory, it will incur disk access costs to insert each posting. Note also that the short lists will be periodically merged with the long lists bringing down document insertion cost again.

Inserted Docs	Query	Score Update	Insertion
1000	27.45	0.25	12.06
2000	28.45	1.21	12.86
4000	27.75	14.12	525.0
8000	27.74	11.38	531.5
10000	28.16	17.17	660.6

Table 3. Varying # Insertions (times in ms)

The results for document deletions and content updates are similar, and are omitted.

B Proof of Theorem 1

Theorem 1. *(Correctness of top–k Search) Algorithm 2 produces the correct top–k results based on the latest scores of the documents.*

Before we prove this theorem, we need to first define some terminology. For an arbitrary candidate document d with document ID id, we define:

- $oScore(id)$ is d's original score stored in long inverted lists;
- $cScore(id)$ is d's current(latest) score stored in Score table;
- $lScore(id)$ is d's short or long list score. Formally,

$$lScore(id) = \begin{cases} e.score & \text{if } P_1 \\ oScore(id) & \text{otherwise.} \end{cases}$$

where P_1 is e=ShortList.lookup(id, t) is not empty (t could be any term contained in document d).

Proof. Algorithm 2 involves two key list scores: the threshold list score (line 22-24) and the stopping list score (line 9-11). Suppose the first one is $threshold$, and the second one is l_{stop}. From line 22-24, we know that at the time $threshold$ is initialized, we already obtain k candidate documents whose current scores are above $threshold$. Thus, to prove the theorem, we only need to prove that the current score of any upcoming candidate document is smaller than *threshold* (Lemma 1.3). Lemma 1.1 and 1.2 are necessary to prove Lemma 1.3. □

Lemma 1.1.

$$lScore(id) = F(id) = \begin{cases} e.lScore & \textit{if } P_2 \\ oScore(id) & \textit{otherwise.} \end{cases}$$

where P_2 is e=ListScore.lookup(id) is not empty.

Proof. $F(id)$ can be transformed to

$$F(id) = \begin{cases} e.lScore & \text{if } P_2 \wedge P_1 \\ e.lScore & \text{if } P_2 \wedge !P_1 \\ oScore(id) & !P_2. \end{cases}$$

Based on Algorithm 1 (line 18-27), whenever a document is stored in short lists (P_1), we insert or update an entry to ListScore table (P_2). Thus, there is $P_1 \Rightarrow P_2$ and $!P_2 \Rightarrow\ !P_1$. $F(id)$ is equal to

$$F(id) = \begin{cases} e.lScore & \text{if } P_1 \\ e.lScore & \text{if } P_2 \wedge !P_1 \\ oScore(id) & !P_2. \end{cases}$$

Now we prove $P_2 \wedge !P_1 \Rightarrow oScore(id) = e.lScore$ where e=ListScore.lookup(id). The predicate $P_2 \wedge !P_1$ means that the document's new score is smaller than *thresholdValueOf (lScore(d))*. Based on Algorithm 1 line 9-17, $e.lScore$ is never changed and equals to the document's original score $oScore(id)$. Note that $e.lScore$ is only updated when its new score exceeds the threshold (line 28). Thus, $F(id)$ is equal to

$$F(id) = \begin{cases} e.lScore & \text{if } P_1 \\ oScore(id) & \text{if } P_2 \wedge !P_1 \\ oScore(id) & !P_2. \end{cases}$$

Since $P_2 \wedge !P_1 \vee !P_2 =!P_1 \vee !P_2$ and $!P_2 \Rightarrow !P_1$, we have $P_2 \wedge !P_1 \vee !P_2 =!P_2$, and thus $F(id) = lScore(id)$. □

Lemma 1.1 indicates that the value *lScore* obtained in either line 11 or 14 of Algorithm 1 is exactly the list score of the updated document.

Lemma 1.2.

$$cScore(id) < \textit{thresholdValueOf}(lScore(id))$$

Proof. The proof is obvious based on Lemma 1.1 and Algorithm 1 (line 18-28). □

Lemma 1.3. *In Algorithm 2, suppose the current candidate document obtained while merging the inverted lists is d_0 with document ID id_0 and list score $lScore(id_0)$. Given any candidate document obtained after d_0, say d, with document ID id and current score $cScore(id)$, there is*

$$cScore(id) < \textit{thresholdValueOf}(lScore(id_0))$$

Proof. Since both long and short inverted lists are ordered by decreasing scores and document d is obtained after d_0, there is $lScore(id) < lScore(id_0)$. Also, function *thresholdValueOf* is monotonic. As a result,

$$\text{threshodValueOf}(lScore(id)) < \text{thresholdValueOf}(lScore(id_0))$$

Based on Lemma 1.2, there is

$$cScore(id) < \text{thresholdValueOf}(lScore(id))$$

Then we get

$$cScore(id) < \text{thresholdValueOf}(lScore(id_0))$$

□

C Proof of Theorem 2

Theorem 2. *(Correctness of top–k Search) Algorithm* **??** *produces the correct top–k results based on the latest scores of the documents and the term based scores.*

Proof. There are three stopping conditions:
(1) Reach the end of chunk with chunk id CID;
(2) After prunning, remainList is empty;
(3) $F(S_{term}^{max}, S_{svr}^{max}) \leq$ *threshold* =resultHeap.minScore(k), where S_{svr}^{max} = *thresholdValueOf*(CID) and $S_{term}^{max} = \sum_{i=1}^{n} f_{min}(t_i)$.

Similar to the proof of Theorem 1, we only need to prove that given any candidate document obtained after this stopping point, its combined score $F(S_{term}, S_{svr})$ is smaller than *threshold* (=resultHeap.minScore(k)). Since the combined score function $F()$ is monotonic, we only need to prove that $S_{term} < S_{term}^{max}$ and $S_{svr} < S_{svr}^{max}$.

step 1. Prove that $S_{term} < S_{term}^{max}$. In Algorithm **??**, for each query term t, $f_{min}(t)$ is the mimimum term score of fancy list $FL(t)$, and at the same time the maximum term score of those documents in $LL(t) \cup SL(s) - FL(t)$. Since after prunning remainList is empty (condition 2), $f_{min}(t)$ is the maximum term score of any upcoming candidate document. Thus,

$$S_{term} < \sum_{i=1}^{n} f_{min}(t_i) = S_{term}^{max}$$

step 2. Prove that $S_{svr} < S_{svr}^{max}$. Since the stopping point is the end of chunk with id CID (condition 1), any upcoming candidate document has chunk id cid < CID. Thus, $S_{srv} <$ *thresholdValueOF*$(cid) <$ *thresholdValueOf*(CID) $= S_{svr}^{max}$. □