Interaction of Query Evaluation and Buffer Management for Information Retrieval

Björn T. Jónsson*
University of Maryland
bthj@cs.umd.edu

Michael J. Franklin*
University of Maryland
franklin@cs.umd.edu

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

The proliferation of the World Wide Web has brought information retrieval (IR) techniques to the forefront of search technology. To the average computer user, "searching" now means using IR-based systems for finding information on the WWW or in other document collections. IR query evaluation methods and workloads differ significantly from those found in database systems. In this paper, we focus on three such differences. First, due to the inherent fuzziness of the natural language used in IR queries and documents, an additional degree of flexibility is permitted in evaluating queries. Second, IR query evaluation algorithms tend to have access patterns that cause problems for traditional buffer replacement policies. Third, IR search is often an iterative process, in which a query is repeatedly refined and resubmitted by the user. Based on these differences, we develop two complementary techniques to improve the efficiency of IR queries: 1) Buffer-aware query evaluation, which alters the query evaluation process based on the current contents of buffers; and 2) Ranking-aware buffer replacement, which incorporates knowledge of the query processing strategy into replacement decisions. In a detailed performance study we show that using either of these techniques yields significant performance benefits and that in many cases, combining them produces even further improvements.

*The work of Björn T. Jónsson and Michael J. Franklin was partially supported by the NSF under grant IRI-94-09575, by the Office of Naval Research under contract number N66001-97-C8539 (DARPA order number F475), by Bellcore, and by an IBM Shared University Research award.

1 Introduction

The improved access to document collections and unstructured data due to the proliferation of the World Wide Web has brought information retrieval (IR) techniques to the forefront of search technology. To the average computer user, "searching" now means using IR-based systems for finding information on the WWW or in other document collections.

While database and IR system development efforts have both been traditionally aimed at providing access to large collections of disk resident data, the problems addressed by the two communities have differed substantially. Database query languages have well-defined semantics — for a given query on a given database there is a single, correct answer. As a result, database query processing work has been largely focused on improving the *efficiency* of data access and manipulation. This emphasis is demonstrated by database query processing benchmarks such as TPC-D [Tra95], which measure only the response time and/or cost of query processing; it is assumed that all database systems will return the same *answer set* for each query.

In contrast, the semantics of IR queries are much fuzzier. Information retrieval is based on natural language and as a result, IR systems must cope with the ambiguity and inconsistencies that are inherent in the use of such language. Benchmarks and studies for IR query processing have therefore focused on improving retrieval *effectiveness*, which is a measure of user satisfaction with the returned answers (e.g., see [Tur94, VH97]). Of course, efficiency is also a concern for IR systems, and there has been significant research on indexing techniques and query evaluation heuristics that improve the query response time while maintaining a constant level of effectiveness (e.g., see [Fal85, ZMSD92, WL93, Per94, Bro95, TF95]). This research, however, has not investigated the system mechanisms that underlie the indexing and querying approaches.

Database systems developers have long realized that efficient access to data requires smart algorithms for disk allocation and disk scheduling, buffer management, and process scheduling. As document collections and user communities grow, and the queries themselves become more complicated, it will only become more important to look at all the components of an IR system and to ensure that they work efficiently and in harmony.

In this paper, we apply a "database systems perspective" to an IR query processing algorithm by investigating the impact of buffer management on the efficiency of that algorithm. In the process, we show that the fuzziness of the IR query model allows for development of a new class of query evaluation approaches that are not applicable in a traditional database environment. Furthermore, we show that due to the particular pattern of data access exhibited by the IR query evaluation algorithm, a simple, application-oriented buffer replacement policy can lead to large efficiency benefits.

1.1 Buffer Management

In the database community, it has long been known that buffer management plays a key role in providing efficient access to disk-resident data. This is evidenced by a long list of studies of buffer management techniques for database systems (e.g., see [EH84, CD85, NFS91, CR93, OOW93, JS94, DFJ+96]). The IR community, on the other hand, has largely ignored buffer management issues so far.

We have observed that query evaluation algorithms for IR have access patterns that cause problems for traditional buffer replacement policies. Buffering becomes increasingly important in the presence of a common IR search behavior known as *query refinement*. Query refinement refers to a multiple-query search process where a user repeatedly modifies a query, by adding or removing terms, and resubmits it to the document retrieval system [Fid91, KQCB94]. The combination of bad replacement behavior and query refinement can obviously lead to serious performance[1] problems.

Also, as stated above, the information retrieval context adds a new and interesting dimension to buffering. In the traditional query models for structured data, all data potentially relevant to the query must be read; the only choice is in what order to read it. For IR, on the other hand, the query evaluation engine typically has the added flexibility of choosing *which data to read*, due to the inherent fuzziness of the queries.

To address these performance issues we propose two complementary techniques, *buffer-aware query evaluation* and *ranking-aware buffer replacement*. The first technique uses the flexibility of the query evaluation to place emphasis on using buffer-resident data, which can in turn reduce the need for reading data from disk. We present a query evaluation algorithm that dynamically changes the query evaluation strategy based on buffer contents. The second technique, based on ideas from Semantic Caching [DFJ+96], is a cache replacement policy that makes use of the access patterns of the retrieval algorithm as well as the data on the data pages when making replacement decisions.

[1] In this paper, we use *performance* to refer to retrieval efficiency. This is in contrast to the IR community, where the term is often used to refer to retrieval effectiveness.

1.2 Overview of the Paper

In addition to the two aforementioned techniques, buffer-aware query evaluation and ranking-aware buffer replacement, this paper makes two other important contributions. First, a new *query refinement workload* for studying buffering issues in information retrieval is developed. Since buffering behavior has not been studied by the IR community, no workloads existed previously which emphasized it. Second, in a detailed performance study of our buffering techniques, we show that using either of the techniques will yield significant performance benefits, and that in many cases, combining them can lead to even further improvements.

Note that in the past, Web-search engines typically have not seen query refinement patterns to the same extent as the traditional IR systems on which we focus in this study. This difference has been at least partially due to the lack of tools to facilitate refinement. Recently, however, Web-search engines have been adding facilities to aid users in refining queries. As these tools get better, our results will be more and more applicable to the Web case.

The remainder of the paper is organized as follows. In Section 2 we give an overview of document retrieval, with emphasis on the aspects of query evaluation that we exploit in our techniques. In Section 3 we present our proposed buffering techniques, as well as the document retrieval algorithm to which the performance is compared. Section 4 discusses the experimental environment and the document collection used for the performance experiments. In Section 5 we present our experimental results. In Section 6 we discuss related work, and in Section 7 we give concluding remarks.

2 Overview of Information Retrieval

IR systems have many similarities to database systems, especially to data warehouses. In particular, document collections are often very large and updates are done infrequently and by a central authority, so consistency issues are not very important. As stated in the introduction, however, the problems addressed by the IR and database communities have differed substantially. A brief description of the TREC conference series [Har96, VH97], which serves as the major benchmarking activity for IR systems, illustrates this difference. The TREC conference uses a standard document collection, which is made up of three parts:

Documents There are currently roughly 4 GB of documents in a number of sub-collections.

Topics A topic is a written description from which queries are constructed. There are 300 topics.

Relevance judgments For each topic there are two lists of documents. One contains documents that a group of experts has deemed to be relevant to the topic, the other contains documents that they have deemed irrelevant. Due to the size of the collection, only documents retrieved by at least one system participating in a TREC conference are judged for relevance.

Annually, a number of research groups and commercial vendors meet to compare their systems based on this document collection. The main emphasis of the TREC conference is to compare the *effectiveness* of the participating systems, with very little attention paid to their *efficiency*.

Our buffering techniques exploit properties of queries, answers, physical organization, and query evaluation for IR. We therefore examine these issues in detail in the remainder of this section, focusing on a particular instantiation of a document retrieval system that is used in this paper, rather than on an analysis of all possible design choices.

2.1 Queries and Query Refinement

Early commercial IR systems used a query model based on boolean algebra. For example, the query $t_1 \wedge t_2$ would return, in no particular order, those documents containing both terms t_1 and t_2, whereas query $t_1 \vee t_2$ would return all documents containing either term. The boolean query model is similar to the query model used by relational languages such as SQL. Formulating boolean queries that return result sets of manageable size has been shown to require significant expertise; as the size of the collection grows, formulating queries becomes even harder [Tur94].

Natural language (or *vector space model*) techniques were developed concurrently with the boolean query model, but have only recently been adopted by commercial IR systems [Tur94]. In a document retrieval context, a natural language query consists of a list of terms (implicitly connected by the $\vee$ operator); any document which contains one or more of the terms is perceived to be relevant.[2] In contrast to the boolean approach, documents are *ranked* by perceived relevance to the user query (see Section 2.2). Additionally, many systems restrict the answer to the n most relevant documents, with n typically being set to a number that is manageable for a user (e.g., 200 or fewer). Limiting the result size has two benefits: First, it limits the number of documents the user needs to inspect, and second, it limits the data that needs to be read, which leads to improved response time of the retrieval system. Research has shown that natural language techniques give better query results than boolean techniques, regardless of the size of the document collection [Tur94]. For this reason, we use natural language queries in this paper.

Studies comparing query evaluation algorithms for document retrieval have traditionally used between 35 and 100 terms per query (e.g., see [Bro95, Per94, VH97]). The rationale is that even when users provide fewer terms, the retrieval system will use techniques such as relevance feedback [SB90] or query augmentation with synonyms to improve retrieval effectiveness, resulting in larger queries. While our workloads are based on such queries, we are most interested in *query refinement*, which is an important search behavior in IR systems [Fid91, KQCB94]. When a ranked list of documents does not match what the user had in mind, the user refines the query by adding or removing terms, and resubmits it. This may occur repeatedly, until the user is satisfied with the returned results. As we shall see, this search behavior has a significant effect on the way queries should be processed.

2.2 Answers and Ranking

Many systems accomplish the ranking of documents using *cosine similarity* (see [SB88] for details). Using the cosine similarity measure, the perceived relevance of document d to query q is:

$$relevance_{q,d} = \frac{\sum_t w_{d,t} \cdot w_{q,t}}{W_d}, \qquad (1)$$

where $w_{d,t}$ is the "weight" of term t in d, $w_{q,t}$ is the weight of t in q, and W_d is the "vector length" of document d:

$$W_d = \sqrt{\textstyle\sum_t w_{d,t}^2} \qquad (2)$$

The product $w_{d,t} \cdot w_{q,t}$ is called the *partial similarity* of document d due to term t. As in [Per94], the weight of t in d is defined by:

$$w_{d,t} = f_{d,t} \cdot idf_t \qquad (3)$$

where $f_{d,t}$ is the number of occurrences of t in d and idf_t is the *inverse document frequency* of the term t. An analogous formula applies to $w_{q,t}$; note that terms may have different frequencies in queries, e.g. due to relevance feedback. The inverse document frequency is defined as:

$$idf_t = log_2(N/f_t) \qquad (4)$$

where N is the number of documents in the collection, and f_t is the number of documents in which term t appears at least once. The inverse document frequency assigns a high value to terms that are found only in few documents in the collection, but a low value to the more common terms; it is used by many query evaluation algorithms to decide the order in which terms are processed.

Due to the richness and ambiguity of natural language, it is possible that a document that is highly ranked is not relevant to the user query, and vice versa. The standard way to measure the quality of answers, or *retrieval effectiveness*, is using *precision* and *recall*. Recall is defined as the number of relevant documents returned to the user (where relevance is judged by humans) divided by the number of relevant documents in

[2] In many systems, additional operators, such as proximity operators, which restrict the location of terms in the documents, are provided. For simplicity we have left such operators out of this work; adding support for them is one avenue for future work.

the database; precision is the number of relevant documents returned divided by the number of documents returned. These measures cannot be obtained for arbitrary queries, as that would require the user to judge all documents in the collection for relevance, not just the ones returned by the system. Therefore, to compare systems and query evaluation strategies, collections such as TREC are used, where experts have judged documents for relevance.

2.3 Physical Organization

Since document collections are typically very large, scanning them to search for relevant documents is not practical. Most retrieval algorithms therefore rely on some index structure for efficiency; the index is typically compressed for disk savings [PZSD96]. The most commonly used index structure is the *inverted index* (see [Fal85] for a survey of access methods for text). The inverted index has one *inverted list* for each term t, where all $(d, f_{d,t})$ entries (required for calculating the ranking) are stored. Inverted lists are traditionally ordered by document identifiers, as many query evaluation algorithms use those identifiers (e.g., see [ZMSD92, MZ94, Bro95]). An alternative organization, used in this paper, is a frequency ordering of the inverted lists [WL93, Per94], where those documents in which the terms occur most frequently are stored first in the lists. This organization has the advantage that documents found early in the inverted lists are likely to be highly ranked, and has been shown to give significant performance advantages [WL93, Per94].

2.4 Query Evaluation

Query evaluation is the last aspect of document retrieval systems that influences our proposed techniques. In relational database systems a query has a single *correct* answer. Thus, query optimization cannot transform the query in a way that would cause it to return a different answer. Such optimization is called *safe*, as it does not compromise the quality of the result. Since document retrieval systems do not return a single correct answer, however, IR researchers have developed *unsafe* (or *approximate*) query evaluation algorithms, which improve the response time of the system at the cost of potential degradation in retrieval effectiveness [TF95].

There are three main factors that affect the response time of document retrieval algorithms: Disk reads, CPU cost and memory requirements. While many database systems are disk bound, Turtle and Flood [TF95] report that in natural language systems it is unclear whether disk or CPU cost is more important. Most of the CPU cost, however, is due to decompression of index data and calculations of partial scores, and thus is directly proportional to the number of disk reads. As for memory requirements, the algorithms must allocate accumulators to keep track of partial scores for documents deemed to have the potential to be among the n highest scoring documents. The set of such documents is called the candidate set, and is kept in memory. If it is too large to fit in memory, intermediate results must be written to disk, incurring additional I/O operations. If no precautions are taken, the candidate set frequently includes more than half of the documents in the collection [Bro95].

There are many unsafe optimizations reported in the IR literature (see [TF95] for a survey). Some achieve significant improvement in response time for individual queries, while maintaining acceptable retrieval effectiveness. As all these algorithms read data in a predetermined manner, they do not perform well on query refinement workloads if buffer space is limited (see Section 3.3 for a more detailed discussion). The buffer-aware algorithm proposed in this paper employs an unsafe optimization, which improves performance with limited buffer space by dynamically changing the query evaluation strategy based on buffer contents.

3 The Buffering Techniques

In this section we discuss two new techniques for efficiently buffering inverted index pages. As mentioned in the introduction, buffering is very important for providing efficient access to disk resident data. This is especially true for query refinement workloads (see Section 2.1), where queries are related, and data read by one query is highly likely to be used in the next query.

We have proposed two techniques, buffer-aware query evaluation and ranking-aware buffer replacement. For buffer-aware query evaluation, we have extended an existing algorithm, Document Filtering [Per94], by altering its search strategy to first process buffer-resident data when possible. Document Filtering is presented in Section 3.1, and our extension is discussed in Section 3.2. Ranking-aware buffer replacement is based on the idea of "semantic value functions" from [DFJ+96]. We propose a replacement policy that bases the replacement value of data pages on the actual data on the pages, as well as on the previously posed query. This policy is introduced in Section 3.3.

3.1 The Document Filtering Algorithm

Persin's Document Filtering (DF) algorithm [Per94] is based on the intuition that if there are documents with large similarity values, it is not profitable to consider small partial similarities (see Section 2.2), as they will not be able to substantially change the final rank of documents. Note that, as the heuristic employed by this algorithm is *unsafe*, the final score for a document may not be its full score and the ranking order may be affected.

The DF algorithm is shown in Figure 1. It processes the inverted index one term-at-a-time, accumulating scores for documents as the inverted lists are processed. The first two steps of the algorithm are for initialization. The set of accumulators, $\mathcal{A}$, corresponds to

1. Create an empty set of accumulators, $\mathcal{A}$.
2. Set $S_{max} = 0$.
3. Sort the terms in the query in order of decreasing idf_t (shortest inverted lists first).
4. For each term t in the sorted list of terms,
   (a) Compute the value of the thresholds f_{ins} and f_{add}, according to Equation 5.
   (b) If $f_{max} \leq f_{add}$, go to step 4 and process next term.
   (c) For each $(d, f_{d,t})$ pair in the inverted list,
      i. If $f_{d,t} > f_{ins}$ and $A_d \in \mathcal{A}$, set $A_d = A_d + w_{d,t} \cdot w_{q,t}$.
      ii. Else, if $f_{d,t} > f_{ins}$ and $A_d \notin \mathcal{A}$, set $A_d = w_{d,t} \cdot w_{q,t}$ and add A_d to $\mathcal{A}$.
      iii. Else, if $f_{d,t} > f_{add}$ and $A_d \in \mathcal{A}$, set $A_d = A_d + w_{d,t} \cdot w_{q,t}$.
      iv. Else, if $f_{d,t} \leq f_{add}$, go to step 4 and process next term.
      v. Set $S_{max} = \max(S_{max}, A_d)$.
5. Divide each accumulator A_d in the candidate set $\mathcal{A}$ by the length of the corresponding document, W_d.
6. Identify the n highest values of accumulators, and return
   the document pointers to the user.

Figure 1: The Document Filtering (DF) algorithm.

the candidate set of documents for which scores must be accumulated, while S_{max} stores the maximum accumulated score seen so far for any document. The third step orders the terms for processing; like many other query evaluation algorithms, DF processes inverted lists in order of decreasing idf_t (i.e., it processes shorter inverted lists first). This step requires that the idf_t value of all terms in the collection be maintained in memory.

Step 4 is where the actual accumulation of scores happens. For each query term t, a pair of thresholds is calculated: the *insertion threshold*, f_{ins}, of the term and the *addition threshold*, f_{add} ($f_{ins} \geq f_{add}$). For each $(d, f_{d,t})$ entry in the inverted list, the frequency $f_{d,t}$ is compared to the two thresholds. If it passes the *insertion* threshold ($f_{d,t} > f_{ins}$), then document d is considered important enough to be in the candidate set. If it is already there, the partial similarity due to term t ($w_{d,t} \cdot w_{q,t}$) is added to the accumulator A_d. If the document is not in the candidate set then a new accumulator, A_d, is allocated to keep track of the score of that document and initialized to the partial similarity. If the frequency $f_{d,t}$ only passes the *addition* threshold ($f_{d,t} > f_{add}$), then the partial score is only considered worth keeping track of for documents that are already in the candidate set, so if the document is in the candidate set, the partial similarity due to term t is added to A_d. If the document is not in the candidate set, nothing is done; this is also the case if the frequency passes neither threshold.

The f_{add} and f_{ins} thresholds are called *filtering thresholds*, as they filter out information that does not contribute much to the final answer. As discussed in Section 2.3, using the frequency based organization of inverted lists, the documents with the highest frequency are encountered first. When a document d with $f_{d,t} \leq f_{add}$ is encountered (Step 4(c)iv), no remaining document will pass the addition threshold and there is no need to read further in that inverted list. As the frequencies of terms are highly skewed towards low values, this allows skipping major portions of the inverted index.[3]

The values of the filtering thresholds are based on predetermined constants, properties of the data and the query, and the maximum partial score of any document, S_{max}, seen thus far into the query execution:

$$f_{ins} = \left\lfloor \frac{c_{ins} \cdot S_{max}}{f_{q,t} \cdot idf_t^2} \right\rfloor, f_{add} = \left\lfloor \frac{c_{add} \cdot S_{max}}{f_{q,t} \cdot idf_t^2} \right\rfloor \qquad (5)$$

The constant c_{ins} controls the size of the candidate set, and may be used as a tuning parameter for the algorithm; by increasing c_{ins}, fewer documents will pass the insertion thresholds, and the candidate set will be reduced. The constant c_{add} controls the number of disk reads performed. The higher c_{add} is set, the earlier $f_{d,t}$ will fall below the addition threshold when reading an inverted list, thus omitting a larger portion of it. By setting $c_{add} = c_{ins} = 0$ the optimization is turned off, and all data of the inverted lists of all the query terms are processed. This provides a convenient baseline for gauging the performance of the unsafe optimization.

The c_{ins} and c_{add} parameters must be tuned to the document collection, and the query workload that the algorithm must serve (see [Per94] for details of the effects of varying these parameters). When properly tuned, this algorithm significantly reduces the size of the candidate set, and the number of $(d, f_{d,t})$ pairs processed, with only negligible difference in retrieval effectiveness [Per94]. In Section 5.1.1 we briefly discuss the performance of the DF algorithm on individual queries based on the TREC collection.

3.2 Buffer-Aware Query Evaluation

We now turn to our first technique, buffer-aware query evaluation. Recall that the concept of unsafe optimizations allows considerable flexibility in obtaining an answer to a query as long as satisfactory retrieval effectiveness is maintained. The DF algorithm uses this flexibility to avoid reading data that is likely to have little effect on the ultimate answer. As defined, however, DF reads data in a predetermined order based on the idf_t of the terms. We propose a modification to DF,

[3] The maximum frequency of any document for a given term, f_{max}, is stored separately (with the idf_t values), allowing the algorithm to avoid reading *any* inverted list data, when no document in the inverted list will pass the addition threshold (Step 4b). This occurs very rarely in practice.

called Buffer-Aware Filtering (BAF), which instead of choosing the next term to process based on idf_t value, selects the term which will incur the fewest estimated *disk reads*, based on the buffer contents.

With this modification, when a term is added during query refinement, it is pushed back in the processing order. Pushing the term back has two effects: First, the thresholds are likely to be higher, leading to fewer disk reads for that term. Second, the terms from the previous query will be processed in the same order as before, so the pages already in buffers will be utilized as much as possible. We conjecture that the disk performance of the algorithm will be considerably improved without degradation of the retrieval effectiveness; the validity of this conjecture is demonstrated in Section 5.

The only example we know of any optimization of this sort for document retrieval systems is in the context of Web-search engines. Legend has it that some of the search engines examine the query with respect to the contents of the buffer pool. If the inverted lists for many of the terms are in buffers, then those are used, and the inverted lists for the remaining terms are simply not accessed. This leads to very good response time for the query, but unfortunately removes all guarantees on the quality of the results. Consider the following worst case scenario: A user has submitted a query and decides, after examination of the results, to refine the query by adding one term and then resubmits the query. If the system decides to use the above optimization, the exact same results will be returned. Admittedly, this will have been done very fast, but the user's desire for a change in the results is ignored. As we will see in the following, however, buffer-aware optimization can yield significant performance improvement without degradation in the quality of results, as long as it is done carefully. Before presenting the algorithm, we demonstrate the basic idea with an example.

3.2.1 A Query Refinement Example

Assume that the query "drastic price increases in American stockmarkets" has been posed against the document collection used in our performance study (a collection of articles that appeared in the Wall Street Journal during the years 1987–1992; see Section 4.2 for details). After removing stop-words and stemming terms (see Section 4.2 for details of these two techniques) it becomes "drastic price increas american stockmarket". Let us examine what happens if we then refine the query by adding the term "invest" to it (the stem of "investment"), and run the resulting query[4] while the inverted lists from the initial query are still in buffers.

Term	idf_t	*Pages*	S_{max}	f_{ins}	f_{add}	*Proc.*	*Read*
Stockmarket	12.01	1	0.0	0	0	1	0
Drastic	7.09	4	288.5	1	0	4	0
Invest	**2.36**	**84**	**288.5**	**10**	**1**	**37**	**37**
American	2.34	85	333.0	12	1	38	0
Increas	1.99	109	591.4	29	2	22	0
Price	1.92	114	591.4	32	3	32	0

Table 1: Evaluation of refined query using DF algorithm.

Term	idf_t	*Pages*	S_{max}	f_{ins}	f_{add}	*Proc.*	*Read*
Stockmarket	12.01	1	0.0	0	0	1	0
Drastic	7.09	4	288.5	1	0	4	0
American	2.34	85	288.5	10	1	38	0
Increas	1.99	109	591.4	29	2	22	0
Price	1.92	114	591.4	32	3	32	0
Invest	**2.36**	**84**	**591.4**	**21**	**2**	**20**	**20**

Table 2: Evaluation of refined query using BAF algorithm.

Table 1 summarizes the evaluation using the DF algorithm. The column labeled "Pages" shows the number of pages in the inverted list for each term, "S_{max}" shows the maximum partial score prior to processing an inverted list, "f_{ins}" and "f_{add}" show the thresholds for the term, the column labeled "Proc." shows the number of pages processed of each inverted list, and "Read" shows the number of pages read in from disk. The added term is third in the idf_t order and with DF, 37 pages of the inverted list must be read in from disk. Table 2, on the other hand, shows the evaluation using the BAF algorithm, where the new term is pushed back to the end of the processing order. With BAF, only 20 pages are read from disk — 17 fewer than before. The answers returned in this case by the two executions are similar; of the 20 highest ranked documents, only one document is affected and has a lower score when the term is pushed back.

Note that pushing terms back could potentially lead to more data being processed than with the DF algorithm, leading to increased CPU and memory requirements. In our experiments, however, we did not observe significant differences. On the other hand, if buffer space is limited, the DF algorithm may perform even worse than shown in Table 1. The reading of the inverted list for "invest" may result in the eviction from buffers of pages from the inverted lists for "american", "increas", or "price". This, in turn, requires the system to re-read those pages from disk. When the term is pushed back, however, all buffer-resident pages are processed first.

3.2.2 The Buffer-Aware Filtering Algorithm

The BAF algorithm is presented in Figure 2. The algorithm is identical to DF, except for the addition of

[4]The tuning parameters reported in [Per94] are not suitable for queries with very few terms, as they do not yield any optimization for such queries. We therefore use higher values of the tuning parameters in this example, to make sure the thresholds rise quickly; we have chosen $c_{ins} = .2$ and $c_{add} = .02$, as these values demonstrate well the essence of the algorithm. For our performance experiments, however, we use the values from [Per94].

Step 3a, where in each round of the algorithm, the term that requires the fewest estimated disk reads must be found. Note that this modification can be applied not only to DF, but to any query evaluation algorithm that reads terms in a fixed order. The number of disk pages to read is estimated by calculating, for each term t, the number of pages that will be processed (p_t) and subtracting the number of pages that are in buffers (b_t).[5] As the processing and pruning of each inverted list is identical to that of DF, we focus here on how p_t and b_t are calculated.

To calculate p_t, we first calculate the addition threshold f_{add}, using Equation 5, as if the term t was next in the processing order. Then the p_t value is determined by looking up the f_{add} value in a conversion table, which is maintained in memory and shared by concurrent queries. Maintaining this conversion table for every term and every possible value of f_{add} would result in a very large table. In practice, however, only a fraction of the table needs to be maintained.[6] As written, the algorithm will require $T(T+1)/2$ calculations of f_{add} and p_t for a query with T terms. We note, however, that for a given term, the f_{add} value will not change unless the value of S_{max}, the highest partial document score seen so far, changes. As can be seen in the preceding example, S_{max} is quite often unchanged between processing terms. Furthermore, p_t will only change if f_{add} changes. We therefore keep around two arrays for the terms of each query, one for the addition thresholds and the other for the number of pages, and update these arrays only when necessary.

To obtain the number of pages in buffers, b_t, we simply ask the buffer manager how many of the pages of the inverted list for term t are in buffers. While this feature may not be available in most buffer managers, implementing it is straight-forward. Since b_t may change any time buffer replacement is done, the algorithm must inquire about it for each term each time Step 3a is performed, for a total of $T(T+1)/2$ times. This calls for an efficient implementation of the maintenance of, and access to, the b_t values. This can be achieved using a hash-table or an array of counters, which are updated whenever a page is moved in or out of buffers.

Note that it is possible for BAF to completely avoid reading the inverted list for a newly added term, if the term has very low f_{max}, thus returning the same answer in query refinement situations. We have instrumented our system to notify us when this occurs, however, and in all our experiments it never has. Furthermore, such terms are not likely to have much effect on the final ranking. Finally, an easy fix is to always process at least the first page of each term.

1. Create an empty set of accumulators, $\mathcal{A}$.
2. Set $S_{max} = 0$.
3. Until all terms are marked as "done",
   (a) For each unmarked term,
      i. Calculate f_{add} according to Equation 5.
      ii. From a memory resident conversion table, look up p_t, the number of pages that need to be processed.
      iii. Inquire the buffer manager about b_t, the number of pages of the inverted list for the term in buffers.
      iv. Calculate the expected number of disk reads, $d_t = \max(p_t - b_t, 0)$.

      Select the term t with lowest d_t (fewest disk reads), using higher idf_t as a tie-breaker. Mark t as "done".
   (b) Compute the value of the thresholds f_{ins} and f_{add}, according to Equation 5.
   (c) If $f_{max} \le f_{add}$, go to step 3.
   (d) For each $(d, f_{d,t})$ pair in the inverted list,
      i. If $f_{d,t} > f_{ins}$ and $A_d \in \mathcal{A}$, set $A_d = A_d + w_{d,t} \cdot w_{q,t}$.
      ii. Else, if $f_{d,t} > f_{ins}$ and $A_d \notin \mathcal{A}$, set $A_d = w_{d,t} \cdot w_{q,t}$ and add A_d to $\mathcal{A}$.
      iii. Else, if $f_{d,t} > f_{add}$ and $A_d \in \mathcal{A}$, set $A_d = A_d + w_{d,t} \cdot w_{q,t}$.
      iv. Else, if $f_{d,t} \le f_{add}$, go to step 3.
      v. Set $S_{max} = \max(S_{max}, A_d)$.
4. Divide each accumulator A_d in $\mathcal{A}$ by the length of the corresponding document, W_d.
5. Identify the n highest values of accumulators, and return
   the document pointers to the user.

Figure 2: The Buffer-Aware Filtering (BAF) algorithm

3.3 Ranking-Aware Buffer Replacement

The second technique we propose is ranking-aware buffer replacement. Since most document retrieval systems are built on top of file systems, which use LRU (Least-Recently-Used) for buffer replacement, LRU may be assumed to be the default buffer-replacement policy for document retrieval systems. While LRU may perform well in some cases, it has some problems when

[5] We optimistically assume that the pages in buffers are the first pages in the inverted lists. Using LRU or the ranking-aware buffer replacement policy introduced in Section 3.3, this assumption is guaranteed to hold for all but one term. For MRU the assumption does not hold.

[6] In our experimental setup, $f_{add} = 10$ is the largest threshold of importance, as f_{add} was rarely higher than 10 and $(d, f_{d,t})$ entries with $f_{d,t} > 10$ are very rarely found outside the first page of inverted lists. Furthermore, only 6,060 terms (3.6% of all terms) have more than one page of inverted list data. Since each conversion value is a small integer, the useful parts of the table can therefore be maintained using $6,060 \cdot 10 \cdot 2$ bytes $= 121,200$ bytes, which may easily be kept in main memory; it is much smaller than the arrays of idf_t and f_{max} that must also be kept in memory.

used with query refinement. Since the DF algorithm processes terms in idf_t order and each inverted list is processed in $f_{d,t}$ order, pages that are accessed in consecutive queries are always read in the same relative order. This behavior is analogous to repeatedly reading a relation sequentially in a relational DBMS. It is a well known result that for such access patterns the LRU policy renders buffers useless unless they can hold the entire relation [Sto81]. As will be shown in our experiments, LRU also renders the buffers useless when they cannot hold the inverted list data used by a query refinement workload.[7] The BAF algorithm goes a long way towards solving this problem, but having a better replacement policy is clearly desirable.

In [DFJ+96] we proposed using knowledge of query patterns and the data semantics to determine the replacement value of data in a client cache. Two quick examples demonstrate how that idea is applied to the document retrieval domain:

1. Consider the first and the last page of an inverted list for a term t. While every query which includes t must read the first page, not all queries will be required to read the last page. The first page therefore has more value, and the buffer replacement policy should strive to keep it in memory.

2. In a query refinement workload, terms may be added or removed from the query before resubmitting it. While pages from terms that have been removed are no longer useful and may be evicted, pages from terms that were retained should be kept in buffers.

To address these issues, we propose a Ranking-Aware Policy (RAP) for buffer replacement, which assigns to each page of an inverted index the following replacement value:

$$replacement\ value = w^*_{d,t} \cdot w_{q,t} \qquad (6)$$

where $w^*_{d,t}$ is the highest weight for any document on the page, and $w_{q,t}$ refers to the weight of the term in the query currently being processed. RAP chooses the page with lowest such value as the replacement victim. Using $w^*_{d,t}$ ensures that the first pages of the inverted lists have the highest value. Furthermore, the RAP policy deals easily with pages of terms removed during refinement, as those terms have $f_{q,t} = 0$, and therefore the corresponding pages have replacement value of 0, and will be evicted first (we take care to evict the tail of the list before the head).

As with the BAF algorithm, implementing the RAP replacement policy requires some modifications to the buffer manager. The frame queue must combine the value of the data (the $w^*_{d,t}$ for a page is calculated at database creation/update time, and stored on that page) and the query to calculate the replacement value. Furthermore, as $w_{q,t}$ may change between queries, the replacement value is dynamic so a reorganizing capability is required.[8]

Parameter	*Description*
PageSize	Number of $(d, f_{d,t})$ entries in a page
BufferSize	Size of server buffers (pages)
c_{add}	Tuning constant for f_{add}
c_{ins}	Tuning constant for f_{ins}

Table 3: Experimental parameters.

In this study we have focused on refinement workloads for single users, leaving multi-user workloads for future work. There are at least two options for extending the RAP policy to such workloads. The first is to allocate separate buffer slots to separate queries and use the RAP policy as defined here for each query (if a term is shared by many queries, the highest $w_{q,t}$ could be used). The second option is to maintain a global query history for all users, and manage the buffer pool as a single unit. The trade-offs between these alternatives need to be investigated. Note that in the multi-user case, users may benefit from pages cached in buffers for other users as well.

4 Experimental Environment

4.1 The Document Retrieval Environment

In order to study the relative performance improvements of the proposed techniques, we implemented the DF and BAF algorithms on top of the simulator used in [FJK96, DFJ+96]. The buffer manager of the simulator was modified to use any of the LRU, MRU, or RAP policies. The main parameters used in the experiments are listed in Table 3, and described in the following.

Each inverted list is a separate file,[9] and the $(d, f_{d,t})$ entries are packed into pages, *PageSize* in each page (the setting of *PageSize* is discussed in Section 4.2). In our experiments we vary the size of the buffers (*BufferSize*) from 1 page to as many as are required to avoid buffer replacement (adding buffers after that point has no impact on performance). In all of these experiments the

[7] One well known solution to the problem of repeated sequential reads is to assign segments of buffer space to different queries, and use MRU (Most-Recently-Used) within each segment [CD85]. As will be shown in our experiments, however, MRU faces some additional problems in query refinement workloads. Also, the newer LRU/k [OOW93] and 2Q [JS94] policies will fare no better than LRU in this case.

[8] In a large buffer pool, keeping the page queue completely sorted will be expensive; however, full ordering need not be maintained, as long as the replacement victims are chosen from among the pages with very low values.

[9] As we will show in Section 4.2 most of the terms have inverted lists that are shorter than one page. In a real system, many such lists would share a single disk page, but our system considers each such inverted list to be in a separate page. As queries typically access very few of these terms, however, the likelihood that two of them would be in the same page is very small. In workloads where such terms are frequently accessed, techniques such as dual buffering [KK94] would be appropriate.

tuning parameters of the filtering algorithms, c_{add} and c_{ins}, were set as in [Per94] ($c_{add} = 0.002, c_{ins} = 0.07$).

While disk reads and retrieval effectiveness[10] are our main metrics, the simulator also keeps track of a number of other metrics, including inverted list entries processed (which indicate CPU cost) and the size of the candidate set of documents for which a partial score is kept (which indicates the memory requirements). We report these additional metrics when necessary.

4.2 The Inverted Index

We have indexed the WSJ document collection from the TREC benchmark [VH97]. The WSJ collection contains articles that appeared in the Wall Street Journal during the years 1987–1992. The total size of the collection is roughly 530 MB in 173,252 documents.

Two common techniques for improving performance are stop-word removal [Fox92] and word stemming [Fra92]. Stop-words are terms that occur too frequently in the collection to have any information value and are therefore ignored; as these terms have very long inverted lists, removing them saves disk space and improves response time.[11] Stemming reduces terms to their stem, which may be thought of as the "lowest common denominator" of related terms. For example "computer" and "computing" are both reduced to the stem "comput". Stemming reduces the number of terms in the document collection, thereby reducing the size of the inverted index.

The inverted index was created as follows: First, all non-words (punctuation, numbers, etc.) and stop-words were removed from the documents. All remaining terms were transformed to lower case and stemmed using a Porter stemmer (described in [Fra92]). The occurrences of each term in each document were summed up to form $(d, f_{d,t})$ entries, which were grouped into inverted lists. Finally the inverted lists were sorted, using $f_{d,t}$ as the primary key, and d as the secondary key. After this procedure, there were 167,017 distinct terms (or inverted lists), containing approximately 31.5 million $(d, f_{d,t})$ entries.

As the WSJ collection is relatively small (Turtle and Flood state that "it is now common for users to search single collections containing many tens of gigabytes of text" [TF95]) and we are interested in developing techniques applicable to large collections, we have scaled it tenfold to roughly 5 GB by making each page hold 10 times fewer tuples than it normally would.[12] In the inverted index, each document identifier d is normally represented using 4 bytes, and the frequency $f_{d,t}$ using 2 bytes. Using compression techniques, however, each such 6 byte entry is compressed down to 1 byte [PZSD96]. A page that is one tenth of a 4 KB page with reasonable overhead can hold 404 tuples ($PageSize = 404$). Table 4 breaks up the terms by the length of the inverted lists. Of the 167,017 terms, only 6,060 have inverted lists longer than one page.

Group	idf_t	*Pages*	*Number*
Low-idf_t	1.91–3.10	51–115	265
Medium-idf_t	3.10–5.42	11–50	1255
High-idf_t	5.42–8.74	2–10	4540
Very-high-idf_t	8.74–17.40	$\leq$1	160957

Table 4: Characteristics of inverted lists in the WSJ collection.

5 Performance Experiments

In this section we present performance experiments which analyze the tradeoffs between the DF and BAF algorithms, as well as the LRU, MRU, and RAP replacement policies. The experiments are based on query refinement workloads that we have developed based on queries from the TREC collection. Before presenting details of our results, we define the workloads.

5.1 Workloads

We defined two different workloads, each consisting of a number of query refinement sequences. Each refinement sequence is based on a single TREC query; from the 100 queries studied in [Per94], we obtained 100 refinement sequences for each workload. As average numbers can be hard to explain, however, we have chosen four representative refinement sequences to present in detail. In this subsection we first investigate the four TREC queries behind those refinement sequences, and then describe the query refinement workloads.

5.1.1 TREC Queries

We used the 100 TREC queries used in [Per94] (queries 51–150). The terms and term frequencies ($f_{q,t}$) of the queries were found using the same procedure as was used to construct the inverted index (see Section 4.2). We then used the DF algorithm to run the queries against the WSJ collection, flushing buffers between queries, and compared the performance to a run where DF's optimization is turned off (by setting $c_{add} = c_{ins} = 0$). On average, the unsafe optimization employed by the DF

[10] The metric we use to measure retrieval effectiveness is the *non-interpolated average precision*, which combines recall and precision into a single number. This is one of the metrics used in the TREC conferences [Har96].

[11] We chose the 100 most common words (with highest f_t) as stop-words, as we felt that after 100 terms most of the terms had some value for distinguishing between documents. In retrospect, using a standard stop-word list would have been preferable; however, in [Per94] it is shown that the performance (efficiency) of document filtering is relatively insensitive to the presence or absence of stop-words, since not much additional data in the inverted lists of stop-words is processed by document filtering.

[12] We do the scaling this way, and not by using more document collections from the TREC collection, since our system runs in memory.

Alias	*No.*	*Title*	*Terms*	*Pages*	*Read*	*Savings*
QUERY1	68	Health Hazards from Fine-Diameter Fibers	36	659	150	77.2%
QUERY2	54	Satellite Launch Contracts	31	610	341	44.1%
QUERY3	96	Computer-Aided Medical Diagnosis	31	563	510	9.4%
QUERY4	57	MCI	99	4,093	678	83.4%

Table 5: Details of investigated queries

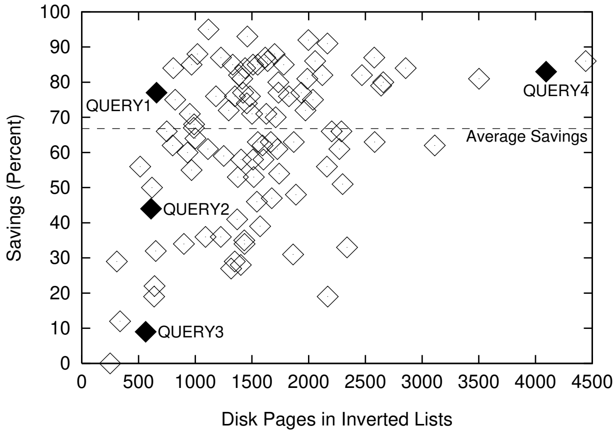

Figure 3: Disk savings of DF, as a function of total length of inverted lists of terms in queries.

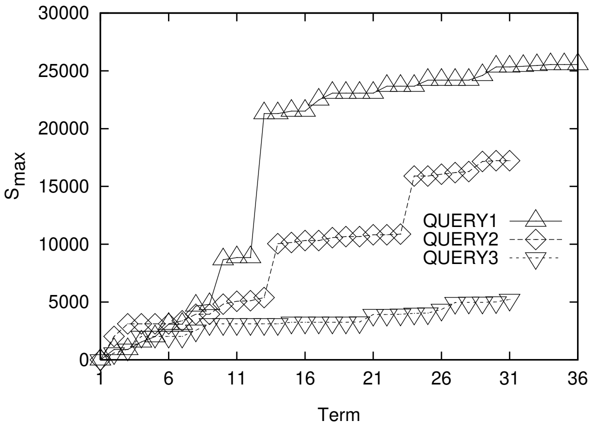

Figure 4: Evolution of S_{max} during processing of query terms.

algorithm is quite effective, reducing disk reads by two-thirds and the number of accumulators by a factor of 50, with negligible difference in retrieval effectiveness.[13] There are, however, significant differences in the performance of individual queries.

Figure 3 indicates the performance improvements for each of the individual TREC queries. The x-axis shows the total number of disk pages in all the inverted lists of all the terms that appear in each query, while the y-axis shows the savings in disk reads, i.e. the fraction of pages which are *not* read and processed using DF. Before moving on to query refinement workloads we briefly examine the performance differences using the four hand-selected TREC queries (described in Table 5 and shown as shaded diamonds in Figure 3).

Looking at the first three queries, which involve similar amounts of inverted list data, but vary widely in performance improvements, the main reason for the savings difference is the relevance of the documents in the collection to the queries. This relevance is reflected in the evolution of S_{max} for the three queries which is shown in Figure 4 (recall that high S_{max} implies high f_{add}, which in turn leads to disk savings). As the figure shows, S_{max} rises fastest and highest for QUERY1. While processing term 12 of QUERY1 (the term "fiber", which has $idf_t = 7.20$ and $f_{q,t} = 5$ and therefore contributes significantly to the ranking) S_{max} more than doubles, and is subsequently higher than S_{max} for QUERY2 ever becomes. For QUERY2 S_{max} is twice lifted significantly, on terms 13 and 23, but for QUERY3 there is no term that contributes much to S_{max}. Turning to QUERY4, the primary reason for the good savings is that the query has a large number of terms with medium or long inverted lists; for those inverted lists great savings are realized due to the low idf_t.

5.1.2 Query Refinement Workloads

Recall that query refinement is an important search behavior in IR systems [Fid91, KQCB94], where a user repeatedly refines the query by adding or removing (dropping) terms, and then resubmits it. We use two types of query refinement workloads, consisting of refinement *sequences* constructed from single TREC queries. Each sequence in turn consists of a number of queries, called *refinements*. These sequences are based on ranking terms according to contribution to document scores, which captures the "importance" of the terms in the TREC query. One workload emphasizes adding terms, while the other also considers dropping terms.

For each of the TREC queries, we ranked the terms by their average contribution to the cosine similarity of the 20 highest ranked documents returned by the DF algorithm when the unsafe optimization is turned off (i.e., all occurrences of all terms are used to calculate the score of documents). Given this ranking, the refinement sequences of the workloads are generated as follows.

ADD-ONLY For each query refinement sequence, the initial refinement query contains the three terms

[13] The results of [Per94] were obtained with stop-words included in the inverted index and queries. We also ran such experiments and obtained results identical to those of [Per94], namely about 90% savings in disk reads, and over 98% savings in accumulators. As stop-words are unlikely to be used in query refinement workloads, we have chosen to omit them in this study.

Group	*idf*$_t$	*Term*	$f_{q,t}$	*Pages*	*Contribution*
1.	7.20	fiber	5	3	5.56
	8.28	asbesto	1	2	0.70
	7.86	insul	2	2	0.39
2.	4.95	fine	3	14	0.36
	3.98	worker	2	28	0.35
	6.08	glass	1	7	0.33
3.	9.67	osha	1	1	0.29
	8.06	lung	1	2	0.28
	6.22	cancer	1	6	0.23
4.	10.18	diamet	3	1	0.22
	3.40	us	2	41	0.21
	5.37	safeti	3	11	0.20
5.	9.77	wool	1	1	0.19
	12.19	cellulos	1	1	0.18
	5.53	install	2	10	0.17
6.	7.75	workplac	1	2	0.15
	3.99	studi	2	27	0.14
	3.56	employe	2	37	0.14
7.	3.18	manufactur	2	48	0.13
	5.04	capac	1	14	0.12
	8.73	ceram	1	2	0.10
8.	2.28	even	2	89	0.09
	6.52	miner	1	5	0.08
	4.17	actual	2	24	0.06
9.	5.21	docum	3	12	0.05
	2.00	concern	2	108	0.04
	6.46	harm	2	5	0.04
10.	5.49	mere	1	10	0.04
	4.82	type	1	16	0.03
	3.42	question	1	41	0.03
11.	3.10	whether	1	50	0.02
	5.81	suspect	1	8	0.02
	4.23	determin	1	23	0.01
12.	10.38	unsubstanti	2	1	0.00
	6.77	diminish	1	4	0.00
	7.60	relev	1	3	0.00

Table 6: Term groups in ADD-ONLY-QUERY1 sequence.

with the highest contribution, the second refinement adds the next three terms, etc. Table 6 shows the term groups in the ADD-ONLY-QUERY1 refinement sequence.

ADD-DROP Terms are added exactly as in the ADD-ONLY workload. However, in each refinement (except the first) the term with the lowest contribution of the previously added group is also dropped. For example using the groups of Table 6, when the second group is added the term "insul" of the first group is removed, and the entire query of five terms is resubmitted.

5.2 The ADD-ONLY Workload

We begin by examining the ADD-ONLY workload. We ran all 100 sequences with varying buffer size. The focus of our presentation will be on the sequences for the TREC queries presented in Section 5.1.1, but we will also briefly describe the results for all 100 refinement sequences. As we have discussed earlier, any performance improvements must be accomplished without harming the retrieval effectiveness. The impact on retrieval effectiveness in this experiment (not shown) is very small. The DF algorithm has the same retrieval effectiveness regardless of replacement policy or buffer size, as its evaluation strategy is not affected by buffer contents at all. Using the BAF algorithm with any of the replacement policies, the relative difference in retrieval effectiveness is within $\pm 5\%$ of DF in over 90% of all runs (refinement sequence/buffer size combinations) and on average it is the same. In the cases where there is a large relative difference, the retrieval effectiveness is usually very low to begin with.

5.2.1 Total Disk Reads

We now look at the total disk reads for entire refinement sequences (note that the cache is cleared before the start of each sequence). Figures 5 and 6 show the disk reads for the ADD-ONLY-QUERY1 and ADD-ONLY-QUERY2 sequences, respectively. The size of the buffers is varied along the x-axis, while the y-axis shows the total number of disk reads. Each figure shows the performance of the DF and BAF algorithms with each of the three replacement policies, LRU, MRU and RAP.

Overall we observe that as buffer size is increased performance improves, until adding more buffers has no effect and the performance remains unchanged. We also note that DF with the LRU policy (the combination used in [Per94]) performs relatively poorly across the range of buffer sizes.[14] By using either BAF or one of the other replacement policies significant improvements are obtained. Finally, in some cases (most notably in Figure 6), even further improvements are seen by using BAF along with one of the better replacement policies. For both sequences, in the best case, the combination of the BAF algorithm and the RAP replacement policy saves more than 70% of the disk reads of DF with LRU. Very similar savings are seen for all the other ADD-ONLY refinement sequences: the best-case savings relative to DF/LRU range from 46% to 90%, with both mean and median around 75%, and 74 sequences (out of 100) showing maximal improvement of over 70%.

As mentioned previously, the reason for the poor performance of DF with LRU is that when the inverted list data read by the refinements does not fit in the available buffers, using a LRU policy makes the buffers useless. BAF is able to improve the performance significantly, as it overcomes this problem with LRU by first reading what is available in buffers, and then reading in what is missing.

[14] Since algorithms that use inverted lists ordered by document identifiers can be expected to read most of the inverted list pages [Bro95], those algorithms would perform significantly worse than DF here.

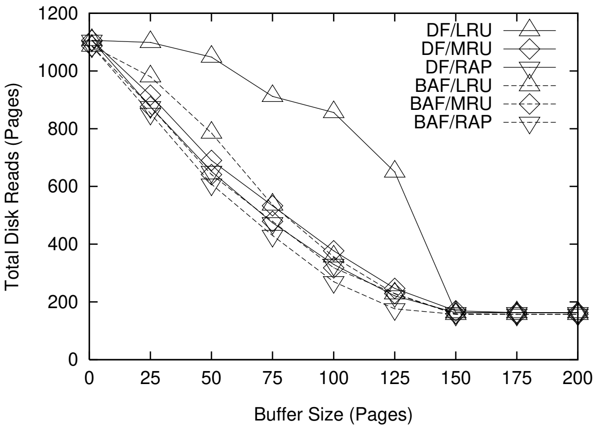

Figure 5: Total disk reads.
ADD-ONLY-QUERY1 sequence, varying buffer size.

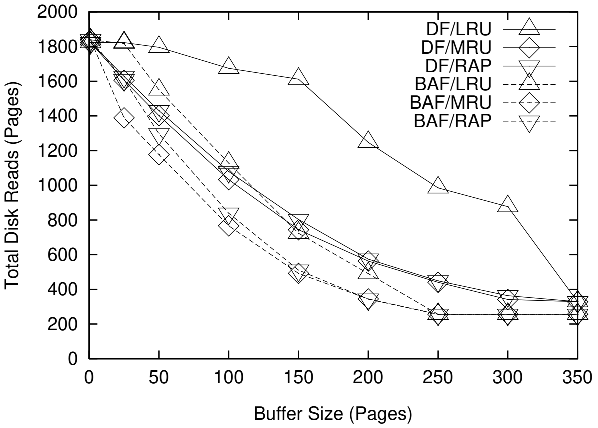

Figure 6: Total disk reads.
ADD-ONLY-QUERY2 sequence, varying buffer size.

Both the MRU and RAP policies also improve the performance substantially; for the ADD-ONLY case there is no clear winner among the two policies. The reason that MRU performs strictly worse than RAP for the ADD-ONLY-QUERY1 sequence (Figure 5) is that early refinements read more of some inverted lists than later refinements. With 100 pages of buffers, the buffer size is reduced by up to 13 pages due to this (in Section 5.3 we show that MRU also has considerable problems with terms that are dropped). For the ADD-ONLY-QUERY2 sequence (Figure 6), on the other hand, any data read is used in all subsequent refinements, and therefore MRU works well. The reason that the RAP policy is somewhat less effective for the ADD-ONLY-QUERY2 sequence is that it occasionally evicts pages from buffers that have not yet been processed when those pages contain entries of lower value than the pages currently being read; those pages must then subsequently be re-read.

The final issue to address for this metric is why using the BAF algorithm and the better replacement policies together sometimes results in improved performance and sometimes not. The key here is that if there is a term which is posed early (and therefore contributes significantly to S_{max}) but is out of idf_t order, then the MRU and RAP policies keep the pages of that term in buffers, and the BAF algorithm uses them efficiently to reduce the number of pages read from other inverted lists. This is the case in the ADD-ONLY-QUERY2 sequence, where the two highest ranked terms are 13th and 22nd in the idf_t order; by using those terms early, even when buffer space is not limited, 20% fewer pages are processed using the BAF algorithm. For the ADD-ONLY-QUERY1 sequence, the highest contributing term is the 12th in the idf_t order, and therefore some benefits are seen, but for the ADD-ONLY-QUERY3 and ADD-ONLY-QUERY4 sequences (not shown) there is no one term with high contribution, and therefore combining the two techniques yields no further improvement.

		Replacement Policy		
		LRU	MRU	RAP
ADD-ONLY-QUERY1	DF	150	38	29
125 buffer pages	BAF	34	32	17
ADD-ONLY-QUERY2	DF	329	80	83
250 buffer pages	BAF	8	8	8

Table 7: Disk reads for the last refinement.

5.2.2 *Disk Reads for the Last Refinement*

While the improvements in the total number of disk reads for all queries are substantial, they are still reduced by the fact that the first few refinements must all read the same amount of data, regardless of the algorithm or the replacement policy; the savings come from the last few refinements. It is therefore interesting to examine the savings for the *last refinement* only.

Table 7 shows the disk reads for the ADD-ONLY-QUERY1 and ADD-ONLY-QUERY2 sequences for the buffer sizes that yield the most improvement. As the table shows, using BAF/RAP produces large savings over DF/LRU, around 90% for ADD-ONLY-QUERY1 and a whopping 97% for ADD-ONLY-QUERY2. It is also interesting to note how much better BAF performs in this case than DF, especially on the ADD-ONLY-QUERY2 sequence.

We also examined the disk reads for the last refinement, when all refinements but the last were collapsed into a single query, resulting in a large first query followed by a small refinement. For this modified ADD-ONLY-QUERY2 sequence, both BAF/LRU and BAF/MRU combinations performed worse than in Table 7, reading around 80 pages, while BAF/RAP still read only 8 pages.

5.2.3 *Other Metrics*

The remaining metrics (not shown), inverted list entries processed and number of accumulators, are largely unaffected by the choice of algorithm and replacement policy (when DF is used, they really are unaffected). The only noticeable difference is for cases when BAF is used with LRU, where the average number of accumulators more than doubles (from 2575 to 5453, which is still quite low). The reason is that occasionally the buffers contain mostly pages from long inverted lists. When BAF reads these lists first, it inserts many documents into the candidate set that are later found to be irrelevant. As with retrieval effectiveness, however, the largest differences occur when DF requires very few accumulators in the first place.

5.3 The ADD-DROP Workload

We now turn to the ADD-DROP workload where terms are dropped as well as added. Again, we ran all 100 ADD-DROP refinement sequences, and again we found ADD-DROP-QUERY1 and ADD-DROP-QUERY2 to be representative. The effects on retrieval effectiveness, number of accumulators, and number of list entries processed, are similar to those for the ADD-ONLY workload, so we will focus on disk reads here.

Figures 7 and 8 show the total disk reads for the ADD-DROP-QUERY1 and ADD-DROP-QUERY2 sequences, respectively. Overall the figures look similar to Figures 5 and 6 from the previous section. The main difference is that here the MRU policy does not perform well; in some cases it even performs worse than LRU.

The reason for the poor performance of MRU on this workload is that it is not able to distinguish between pages that are being used in the current refinement and pages of terms that have been dropped; in fact, since the most recently used page is always replaced, pages of dropped terms are guaranteed *not* to be replaced. This deficiency forces MRU to keep a large number of useless pages in memory; for the ADD-DROP-QUERY1 sequence with 100 buffer pages, a total of 54 pages are useless for the last refinement. Since the RAP policy assigns value to pages with respect to the current refinement, pages from inverted lists of dropped terms are assigned the value 0, and are therefore discarded before any useful pages.

LRU is also able to distinguish between pages of dropped terms and pages of retained terms in the refinement sequence, as the pages of dropped terms age faster in the LRU chain. As long as the total number of pages that are used for the current refinement is smaller than the buffer size, LRU is quite efficient. This effect is evident in Figure 7, where DF/LRU performs better than both DF/MRU and BAF/MRU when the buffers contain 125 or more pages. LRU, however, still suffers from the problem of discarding pages that are useful when the data used by the sequence does not fit in memory.

5.4 Summary

We investigated the performance of the DF and BAF algorithms, using LRU, MRU and RAP replacement

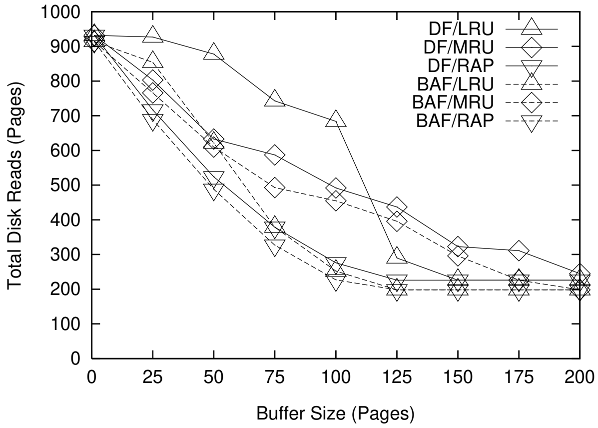

Figure 7: Total disk reads. ADD-DROP-QUERY1 sequence, varying buffer size.

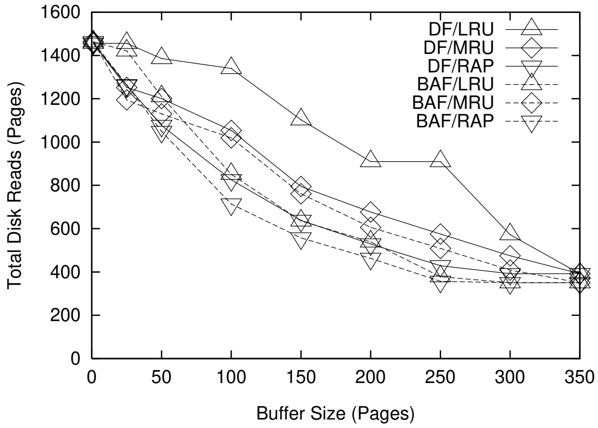

Figure 8: Total disk reads. ADD-DROP-QUERY2 sequence, varying buffer size.

policies. In summary, using DF with LRU (as is done in [Per94]) leads to bad performance with query refinement when buffer space is limited. We observe that while using MRU is sometimes better than using LRU and sometimes worse, using the BAF algorithm or the RAP replacement policy always improves the disk performance significantly; sometimes by as much as 70% for the whole refinement sequence, and as much as 97% for the last refinement. We also observed that in many cases, using both techniques together leads to even further improvements in performance.

6 Related Work

As stated earlier, little work has been done on buffering issues for document retrieval. Client data caching for generic information retrieval systems (where information is not restricted to documents) is studied in [SA87, ABGM90]. Physical index design, inverted index caching, and database scaling for shared-nothing distributed IR systems are studied in [TGM93a, TGM93b]. None of these studies address buffer replacement.

For database systems, however, there are many studies of buffer replacement policies. Some have focused on matching the access patterns of queries (e.g., see [CD85, NFS91, CR93]), while others contend that using access patterns is hard, due to the complexity of SQL, and propose more generic policies (e.g., see [OOW93, JS94]). In document retrieval systems, such as the one studied here, the access patterns are simple and uniform, while presenting serious problems for the policies of [OOW93, JS94].

The concept of using a replacement value function based on the data and the query pattern is presented in [DFJ+96], where it is applied to a relational database in a client caching context. While our techniques are also applicable to client caching, we have focused on server buffering. We have shown that the highly predictable access patterns of IR algorithms allow for a very efficient replacement value function.

The work most related to our buffer-aware algorithm is that of Sarawagi and Stonebraker [SS96], where access to database relation fragments is re-ordered based on the location of the fragments and the time it takes to access them. Experiments with tertiary storage, where multiple users access the same data in a staggered manner, show an order of magnitude decrease in response time. Unlike the IR case, however, for database queries all data must eventually be read. This limits the extent to which they can modify the query evaluation plan.

There are a number of current database research projects which can benefit from techniques similar to those presented here. Buffer-aware evaluation is applicable in projects such as on-line aggregate computation [HHW97], where the emphasis is on producing approximate answers very quickly. Projects that may benefit from buffer-replacement strategies that are tuned to the semantics of the data and queries, include the SQL extensions to restrict answers to highest/lowest values for an attribute presented in [CK97], and the interface for fuzzy answers to queries of [Bro97].

Finally, a project that is looking into the architectural foundations of Web-search engines is the Inktomi project [Ink]. Inktomi's search engine uses a network of workstations (NOW) to efficiently parallelize Web-searches (similar issues were investigated in [TGM93a, TGM93b]), enabling low-cost systems with extreme scalability and significant fault-tolerance.

7 Conclusions and Future Work

The proliferation of the World Wide Web has brought IR techniques to the forefront of search technology. To the average computer user, "searching" now means using IR-based systems for finding information on the WWW or in other document collections. While the IR community has worked on improving the effectiveness and efficiency of query evaluation, little work has been done on the system mechanics that underlie the indexing and querying approaches.

In this paper we have examined an important component of IR systems, that has not been studied by the IR community, namely buffer management. We proposed two complementary techniques, buffer-aware filtering and ranking-aware buffer replacement, and applied them to query refinement workloads. In a detailed performance study, we showed that using either technique significantly improves performance and that in many cases using the two together results in even further performance improvements.

As this is the first study in this area, there are many directions for future work, including implementing our techniques in a fully functional IR system, dealing with more complex queries and query refinement workloads generated using relevance feedback, dealing with other query processing algorithms, investigating performance with other (larger) document collections, and investigating buffering issues in multi-user workloads. Finally, an investigation of the usefulness of buffering techniques in the Web-search environment would be very interesting.

Acknowledgments

We would like to thank Michael Persin for his help with the DF algorithm, Douglas Oard for help with the TREC data and many helpful discussions, and Christos Faloutsos, Donald Kossmann, David Lewis, Amitabh Singhal and Anthony Tomasic for useful references and discussions.

References

[ABGM90] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM TODS*, 15(3), Sept. 1990.

[Bro95] E.W. Brown. Fast evaluation of structured queries for information retrieval. *Proc. ACM SIGIR Conf.*, Seattle, WA, 1995.

[Bro97] K.P. Brown. DBGuide industrial presentation. *ACM SIGMOD Conf.*, Tucson, AZ, 1997.

[CD85] H.-T. Chou and D.J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Proc. of the VLDB Conf.*, Stockholm, Sweden, 1985.

[CK97] M.J. Carey and D. Kossmann. On saying "enough already!" in SQL. *Proc. ACM SIGMOD Conf.*, Tucson, AZ, 1997.

[CR93] C.M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. *Proc. of the VLDB Conf.*, Dublin, Ireland, 1993.

[DFJ+96] S. Dar, M. Franklin, B.T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. *Proc. of the VLDB Conf.*, Bombay, India, 1996.

[EH84] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM TODS*, 9(4), Dec. 1984.

[Fal85] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1), 1985.

[Fid91] R. Fidel. Searchers' selection of search keys: III. Searching styles. *Journal of the American Society of Information Science*, 42(7), 1991.

[FJK96] M.J. Franklin, B.T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. *Proc. ACM SIGMOD Conf.*, Montreal, Canada, 1996.

[Fox92] C. Fox. Lexical analysis and stoplists. In W.B. Frakes and R. Baeza-Yates, editors, *Information Retrieval, Data Structures and Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[Fra92] W.B. Frakes. Stemming algorithms. In W.B. Frakes and R. Baeza-Yates, editors, *Information Retrieval, Data Structures and Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[Har96] D. Harman. Overview of the fourth Text REtrieval Conference (TREC-4). *The fourth Text REtrieval Conference (TREC-4)*, NIST, Gaithersburg, MD, 1996.

[HHW97] J.M. Hellerstein, P.J. Haas, and H.J. Wang. Online aggregation. *Proc. ACM SIGMOD Conf.*, Tucson, AZ, 1997.

[Ink] Inktomi. *The Inktomi Technology Behind Hotbot.* http://www.inktomi.com/Tech/CoupClustWhitePap.html.

[JS94] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. *Proc. of the VLDB Conf.*, Santiago, Chile, 1994.

[KK94] A. Kemper and D. Kossmann. Dual-buffering strategies in object bases. *Proc. of the VLDB Conf.*, Santiago, Chile, 1994.

[KQCB94] J. Koenemann, R. Quatrain, C. Cool, and N.J. Belkin. New tools and old habits: The interactive searching behavior of expert online searchers using INQUERY. *The Third Text REtrieval Conference (TREC-3)*, NIST, Gaithersburg, MD, 1994.

[MZ94] A. Moffat and J. Zobel. Fast ranking in limited space. *Proc. IEEE Conf. on Data Engineering*, Houston, TX 1994.

[NFS91] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. *Proc. ACM SIGMOD Conf.*, Denver, CO, 1991.

[OOW93] E.J. O'Neil, P.E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *Proc. ACM SIGMOD Conf.*, Washington, DC, 1993.

[Per94] M. Persin. Document filtering for fast ranking. *Proc. ACM SIGIR Conf.*, Dublin, Ireland, 1994.

[PZSD96] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society of Information Science*, 47(10), Oct. 1996.

[SA87] P. Simpson and R. Alonso. Data caching in information retrieval systems. *Proc. ACM SIGIR Conf.*, New Orleans, LA, 1987.

[SB88] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5), 1988.

[SB90] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. *Journal of the American Society of Information Science*, 41(4), 1990.

[SS96] S. Sarawagi and M. Stonebraker. Reordering query execution in tertiary memory databases. *Proc. ACM SIGMOD Conf.*, Montreal, Canada, 1996.

[Sto81] M. Stonebraker. Operating system support for database management. *CACM*, 24(7), July 1981.

[TF95] H. Turtle and J. Flood. Query evaluation: Strategies and optimization. *Information Processing & Management*, Nov. 1995.

[TGM93a] A. Tomasic and H. Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. *Proc. ACM SIGMOD Conf.*, Washington, DC, 1993.

[TGM93b] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. *Proc. PDIS Conf.*, San Diego, CA, 1993.

[Tra95] Transaction Processing Performance Council (TPC), 777 N. First Street, Suite 600, San Jose, CA 95112, USA. *TPC Benchmark D (Decision Support)*, May 1995.

[Tur94] H. Turtle. Natural language vs. boolean query evaluation: A comparison of retrieval performance. *Proc. ACM SIGIR Conf.*, Dublin, Ireland, 1994.

[VH97] E.M. Voorhees and D. Harman. Overview of the fifth Text REtrieval Conference (TREC-5). *The fifth Text REtrieval Conference (TREC-5)*, NIST, Gaithersburg, MD, 1997.

[WL93] W.Y.P. Wong and D.L. Lee. Implementation of partial document ranking using inverted files. *Information Processing & Management*, 29(5), Oct. 1993.

[ZMSD92] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. *Proc. of the VLDB Conf.*, Vancouver, Canada, 1992.