Supporting Full-Text Information Retrieval with a Persistent Object Store*

Eric W. Brown, James P. Callan, W. Bruce Croft, J. Eliot B. Moss

Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA

Abstract. The inverted file index common to many full-text information retrieval systems presents unusual and challenging data management requirements. These requirements are usually met with custom data management software. Rather than build this custom software, we would prefer to use an existing database management system. Attempts to do this with traditional (e.g., relational) database management systems have produced discouraging results. Instead, we have used a persistent object store, Mneme, to support the inverted file of a full-text information retrieval system, INQUERY. The result is an improvement in performance along with opportunities for INQUERY to take advantage of the standard data management services provided by Mneme. We describe our implementation, present performance results on a variety of document collections, and discuss the advantages of using a persistent object store to support information retrieval.

1 Introduction

The task of a full-text information retrieval (IR) system is to satisfy a user's information need by identifying the documents in a collection of documents that contain the desired information. This identification process requires a means of locating documents based on their content. A well known mechanism for providing such means is the inverted file index [15].

An inverted file index consists of a record, or inverted list, for each term that appears in the document collection. A term's record contains an entry for every occurrence of the term in the document collection, identifying the document and possibly giving the location of the occurrence or a weight associated with the occurrence. Inverted file indices can become quite large. Some commercial systems contain millions of full-text documents, occupying gigabytes of disk space. An inverted file index for such a collection will contain hundreds of thousands of records, ranging in size from just a few bytes to millions of bytes.

Typically, an IR system that depends on an inverted file index will use custom data management software built from scratch to support the index. An advantage of this approach is that the software is designed specifically to meet the requirements of the particular information retrieval strategy used in the system. A disadvantage is that building such software is difficult and tedious, particularly if it must provide sophisticated features such as concurrency control or recovery.

* This work is supported by the NSF Center for Intelligent Information Retrieval at the University of Massachusetts. Email: {brown, callan, croft, moss}@cs.umass.edu

Instead, we propose using an "off-the-shelf" data management facility, in the form of a persistent object store, to provide the inverted file index service. We have taken the INQUERY full-text retrieval system [20, 3], which originally used a custom B-tree package to provide the inverted file index support, and replaced the B-tree package with the Mneme persistent object store [14]. The result is a system that reaps the benefits of using an existing data management facility without sacrificing performance or functionality. The integrated system actually demonstrates a performance improvement, and the features of the persistent object store offer potential solutions to some of the difficult problems associated with inverted list management.

In the next section we take a closer look at the characteristics of inverted files that make them difficult to support. Next, we describe our integrated software architecture, including details of INQUERY and Mneme. Following that, we present a performance evaluation of the integrated system and discuss the results. In the last two sections we review previous and related work, and offer some concluding remarks. The principle contribution of our work is a demonstration that data management facilities for IR systems need not be custom built in order to obtain superior performance. Additionally, we show how the size distribution characteristics of records in an inverted file index, along with the characteristics of inverted file record access during query processing, can be used to guide decisions regarding persistent store organization and buffer management policy selection.

2 Inverted File Indices

There are three basic operations performed on an inverted file index: creation, lookup, and modification. The operation performed most often is lookup. As the IR system processes queries a lookup is typically performed at least once for each term in the query. Modifications occur less frequently as new documents are added to the collection and old or irrelevant documents are retired from the collection. Creation occurs once when a document collection is first indexed by the IR system, although it may be considered a special case of modification where a number of document additions are batched together.

If we optimize for the common case, lookup should be given the most careful consideration. Efficient lookup requires knowing the size distribution of the records in the file and a characterization of the record access patterns. The size of an inverted list depends on the number of occurrences of the associated term in the document collection. Zipf [23] observed that if the terms in a document collection are ranked by decreasing number of occurrences (i.e., starting with the term that occurs most frequently), there is a constant for the collection that is approximately equal to the product of any given term's frequency and rank order number. The implication is that nearly half of the terms have only one or two occurrences, while a few terms occur very many times.

Figure 1 shows the distribution of inverted list sizes for the TIPSTER document collection used in our performance evaluation below (see Table 1). For a given inverted list size, the figure shows how many records in the inverted file are less than or equal to that size, and how much those records contribute to the total file size. The figure also shows the distribution of sizes for the inverted lists accessed by TIPSTER query set 1.

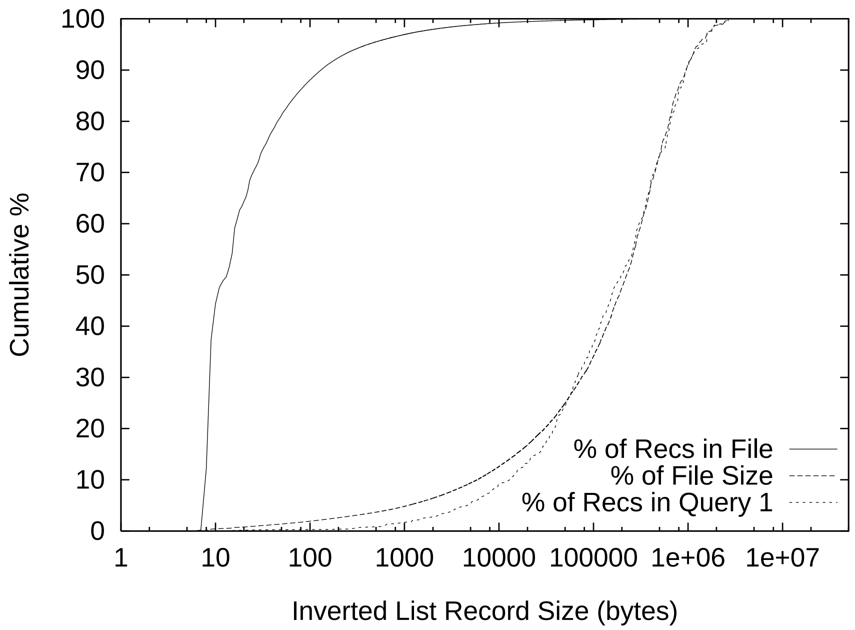

Fig. 1. Cumulative distributions over inverted list size for TIPSTER

The majority of the records accessed are between 10 Kbytes and 1 Mbyte. This size range represents a small percentage of the total number of records in the file, but a large percentage of the total file size. Therefore, we must be prepared to provide efficient access to the majority of the raw data in the file. The strong similarity between the inverted file distribution as a percentage of file size and the query distribution indicates that the former might be used to predict the latter. The same plots for the other collections used in the performance evaluation (not shown here) have similar shapes.

We also observe that there is significant repetition of the terms used from query to query. This can be expected for two reasons. First, a user of an IR system may iteratively refine a query to obtain the desired set of documents. As the query is refined to more precisely represent the user's information need, terms from earlier queries will reappear in later queries. Second, IR systems are often used on specialized collections where every document is related to a particular subject. In this case, there will be terms that are common to a large number of queries, even across multiple users.

Support for modification has traditionally been lacking in IR systems due to their archival nature. The recent trend, however, is towards incremental and dynamic update of document collections and associated index structures. Frequent addition of documents to a collection and modification of existing documents in a collection pose challenging problems for inverted list management. These operations require sophisticated database management features such as concurrency control and versioning, lending further credence to the approach we take below.

3 Architecture

In this section we describe the software architecture that resulted when the B-tree package of INQUERY was replaced by Mneme. We begin with a description of INQUERY, followed by a brief overview of Mneme, and conclude with a discussion of the issues addressed during integration of the two systems.

3.1 INQUERY

INQUERY is a probabilistic information retrieval system based upon a Bayesian inference network model [20, 3]. The power of the inference network model is the consistent formalism it provides for reasoning about evidence of differing types. Extensive testing has shown INQUERY to be one of the best IR systems, as measured by the standard IR metrics of recall and precision [10, 20]. INQUERY is fast, scales well to large document collections, and can be embedded in specialized applications.

The bottlenecks in IR are retrieving and ranking the documents that match a query. Retrieval identifies the (possibly large) subset of the collection that may be relevant to the query. Document ranking orders the documents so that a user can examine first those documents that are most likely to satisfy the information need. In INQUERY, document ranking is a sorting problem, because the Bayesian method of combining evidence assigns a numeric value to each document. Other functionality, for example sophisticated query processing and presentation of results, generally does not affect the speed of the system.

Two of INQUERY's data storage facilities affect the speed of retrieval: a hash dictionary and an inverted file index. INQUERY uses an open-chaining hash dictionary to map text strings (words) to unique integers called *term ids*. The hash dictionary also stores summary statistics for each string and resides entirely in main memory. The inverted file index is organized as a keyed file, using term ids as keys and a B-tree index. There is one record per term. A record has a header containing summary statistics about the term, followed by a list of the documents, and the locations within each document, where the term occurs. The record is stored as a vector of integers in a compressed format. The average compression rate for the four collections in Table 1 is about 60%.

During retrieval, INQUERY performs "term-at-a-time" processing of evidence. That is, it reads the complete record for one term and merges the evidence from that term with the evidence it is accumulating for each document. Then it processes the next term. This approach is fast because it minimizes I/O. However, it requires large amounts of memory for large collections because several inverted lists must be kept in memory simultaneously. A "document-at-a-time" approach, which gathered all of the evidence for one document before proceeding to the next, might scale better to large collections. However, it would be cumbersome with the current custom B-tree package.

3.2 Mneme

The Mneme persistent object store [14] was designed to be efficient and extensible. The basic services provided by Mneme are storage and retrieval of objects, where an object is a chunk of contiguous bytes that has been assigned a unique identifier. Mneme has no notion of type or class for objects. The only structure Mneme is aware of is that objects may contain the identifiers of other objects, resulting in inter-object references.

Objects are grouped into files supported by the operating system. An object's identifier is unique only within the object's file. Multiple files may be open simultaneously, however, so object identifiers are mapped to globally unique identifiers when the objects are accessed. This allows a potentially unlimited number of objects to be created by allocating a new file when the previous file's object identifiers have been exhausted. The number of objects that may be accessed *simultaneously* is bounded by the number of globally unique identifiers (currently 2^{28}).

Objects are physically grouped into *physical segments* within a file. A physical segment is the unit of transfer between disk and main memory and is of arbitrary size. Objects are also logically grouped into *pools*, where a pool defines a number of management policies for the objects contained in the pool, such as how large the physical segments are, how the objects are laid out in a physical segment, how objects are located within a file, and how objects are created. Note that physical segments are not shared between pools. Pools are also required to locate for Mneme any identifiers stored in the objects managed by the pool. This would be necessary, for instance, during garbage collection of the persistent store. Since the pool provides the interface between Mneme and the contents of an object, object format is determined by the pool, allowing objects to be stored in the format required by the application that uses the objects (modulo any translation that may be required for persistent storage, such as conversion of main memory pointers to object identifiers). Pools provide the primary extensibility mechanism in Mneme. By implementing new pool routines, the system can be significantly customized.

The base system provides a number of fundamental mechanisms and tools for building pool routines, including a suite of standard pool routines for file and auxiliary table management. Object lookup is facilitated by *logical segments*, which contain 255 objects logically grouped together to assist in identification, indexing, and location. A hash table is provided that takes an object identifier and efficiently determines if the object is resident in main memory. Support for sophisticated buffer management is provided by an extensible buffering mechanism. Buffers may be defined by supplying a number of standard buffer operations (e.g., allocate and free) in a system defined format. How these operations are implemented determines the policies used to manage the buffer. A pool *attaches* to a buffer in order to make use of the buffer. Mneme then maps the standard buffer operation calls made by the pool to the specific routines supplied by the attached buffer. Additionally, the pool is required to provide a number of "call-back" routines, such as a modified segment save routine, which may be called by a buffer routine.

3.3 The Integrated System

The Mneme version of the inverted index was created by allocating an object for each inverted list in the B-tree file. The Mneme identifier assigned to the object was stored in the INQUERY hash dictionary entry for the associated term. When the inverted list for a term is needed by the query processor, the object identifier for the list is retrieved from the hash dictionary and used to obtain the desired object.

Based on the analysis in Sect. 2 and the features of Mneme, we created three distinct groups of inverted list objects. First, in all of the test collections, approximately 50% of the inverted lists are 12 bytes or less. By allocating a 16 byte object (4 bytes for a size field) for every inverted list less than or equal to 12 bytes, we can conveniently fit a whole logical segment (255 objects) in one 4 Kbyte physical segment. This simplifies both the indexing strategy used to locate these objects in the file and the buffer management strategy for these segments. Inverted lists in this category were allocated in a *small object pool*. Second, a number of inverted lists are so large it is not reasonable to cluster them with other objects in the same physical segment. Instead, these lists are allocated in their own physical segment. All inverted lists larger than 4 Kbytes were allocated in this fashion in a *large object pool*. The remaining inverted lists form the third group of objects and were allocated in a *medium object pool*. These objects are packed into 8 Kbyte physical segments. The physical segment size is based on the disk I/O block size and a desire to keep the segments relatively small so as to reduce the number of unused objects retrieved with each segment.

This partitioning of the objects allows the indexing and buffer management strategies for each group to be customized. Each object pool was attached to a separate buffer, allowing the global buffer space to be divided between the object pools based on expected access patterns and memory requirements. The buffer replacement policy for all of the pools is least recently used (LRU) with a slight optimization. As queries are parsed by INQUERY, a tree is constructed that represents the query in an internal form. Before the tree is processed, we scan it and "reserve" any objects required by the query that are already resident, potentially avoiding a bad replacement choice.

With the above partitioning, the large object pool will still contain a huge range of object sizes. We experimented with further partitioning the large object buffer, but found the best hit rates were achieved with a single buffer of the same total size.

4 Performance Evaluation

We evaluated the persistent object store based INQUERY system by comparing it with the original system. Traditionally, IR system performance has been measured in terms of recall and precision. The portion of the system that determines those factors is fixed across the two systems we are comparing. Instead, we are concerned with execution time, which we measured on a variety of document collections and query sets. Below we describe the execution environment, the experiments, and the results.

4.1 Platform

All of the experiments were run in single user mode on a DECstation 5000/240 (MIPS R3000 CPU[2] clocked at 40 MHz) running ULTRIX[3] V4.2A. The machine was configured with 64 Mbytes of main memory, a 426 Mbyte RZ25 SCSI disk, and a 1.35 Gbyte RZ58 SCSI disk. The machine mounts many of its bin files from another host via NFS,

[2] MIPS and R3000 are trademarks of MIPS Computer Systems.

[3] DECstation and ULTRIX are registered trademarks of Digital Equipment Corporation.

Table 1. Document collection statistics and Mneme buffer sizes (all sizes are in Kbytes)

Collection	# of Docs	Collection Size	# of Terms	Inverted File Size		Object Buffer Sizes		
				B-Tree	Mneme	Small	Med	Large
CACM	3204	2136	5944	641	556	12.7	24.4	24
Legal	11953	290529	142721	65840	71296	12.7	97.7	1098
TIPSTER 1	510887	1225712	627078	460836	476904	12.7	341.8	4596
TIPSTER	742358	2103574	846331	768406	789344	12.7	702.5	7806

and so could not be isolated from the network. In fact, the INQUERY system executables were stored on a remote host, although all of the data files accessed during the experiments were stored locally on the 1.35 Gbyte disk. The INQUERY system was compiled with the GNU C compiler (gcc) version 2.3.2 at optimization level 2.

4.2 Experiments

We measured the execution time of both systems on a number of query sets using the document collections described in Table 1. The documents in CACM [8] are abstracts and titles of articles that appeared in *Communications of the ACM* from 1958 to 1979. The three query sets used with CACM are different representations of the same 50 queries. TIPSTER is volumes 1 and 2 of the *TIPSTER* document collection, containing full-text news articles and abstracts on a variety of topics from news wire services, newspapers, Federal Register announcements, and magazines. The query set was generated locally from *TIPSTER topics 51-100* using automatic and semi-automatic methods. TIPSTER 1 consists of volume 1 only and uses the same query set. Both TIPSTER and CACM are standard test collections in the IR community. Legal is a privately obtained collection of legal case descriptions. The first query set for the Legal collection was supplied with the collection. The second query set was generated locally by supplementing the first query set with dictionary terms, phrases, and weights. In all cases the query sets are designed to evaluate an IR system's recall and precision and are representative of queries that would be asked by real users.

Each query set was processed by the two versions of INQUERY in batch mode, using appropriate relevance and stop words files. A relevance file lists the documents that should have been retrieved for each query and is required for determining recall and precision. A stop words file lists words that are not worth indexing because they occur so frequently or are not significantly meaningful.

Since the B-tree version of INQUERY does no user space main memory caching of inverted lists across record accesses, we measured the Mneme based version of INQUERY both with and without inverted list caching. For the version with caching, the main memory buffer sizes are shown in Table 1 and were determined for each collection as follows. The large object buffer size was 3 times the size of the largest inverted list in the collection. This heuristic was meant to allocate a reasonable amount of buffer space, in a somewhat regulated fashion, for each collection. Merely allocating a percentage of the total inverted file size would be inappropriate given the range of

Table 2. Wall-clock times (all times are in seconds)

Collection	Query Set	B-Tree	Mneme, No Cache	Mneme, Cache	Improvement
CACM	1	6.49	6.02	5.93	9%
	2	7.41	6.40	6.37	14%
	3	11.73	9.34	8.32	29%
Legal	1	62.84	51.36	50.55	20%
	2	65.82	53.46	52.01	21%
TIPSTER 1	1	2683.20	2568.24	2519.55	6%
TIPSTER	1	4132.34	3973.45	3894.74	6%

inverted file sizes. For the three larger collections, the medium object buffer size was 9% of the size of the large object buffer. This allocation was based on object access behavior observed during query processing, where the number of accesses to medium objects equaled roughly 9% of the number of accesses to large objects. For the CACM collection, 9% of the large object buffer would not have been large enough to hold a single medium object segment. Therefore, we made the CACM medium object buffer large enough to hold 3 medium object segments. The small object buffer was made large enough to hold 3 small object segments since small object access was insignificant.

Timings were made using the system clock via calls to ftime() and getrusage(). Timing was begun just before query processing started, after all files had been opened and any initialization was complete. Timing ended when the query set had been processed, before any files were closed. Each query set was run 6 times, and mean times from all six runs are reported below. In all cases, the result of any particular run differed from the mean by less than 1% of the mean. Before each query set was run, a 32 Mbyte "chill file" was read to purge the operating system file buffers and guarantee that no inverted file data was cached by the file system across runs. The measured I/O inputs for each run indicate that this was accomplished.

4.3 Results

Table 2 shows the wall-clock time required by the different versions of INQUERY to process each of the query sets. The Mneme version without caching achieves a noticeable improvement in performance over the B-tree version. The addition of caching to the Mneme version increases the performance further, yielding the improvements shown in the final column of the table. Improvement is calculated as (B-tree time – Mneme with cache time) / B-tree time.

A more precise measure of the portion of the system that varies across the different versions is system CPU time plus time spent waiting for I/O to complete. This was obtained by subtracting user CPU time from the wall-clock time. User CPU time approximates the time spent in the inference engine. This time should be comparable for all versions, and in fact varies by less than 1% across the versions. System CPU plus

Table 3. System CPU plus I/O times (all times are in seconds)

Collection	Query Set	B-Tree	Mneme, No Cache	Mneme, Cache	Improve-ment
CACM	1	1.97	1.48	1.41	28%
	2	2.56	1.53	1.52	41%
	3	5.22	2.82	1.90	64%
Legal	1	24.59	13.67	12.77	48%
	2	26.38	14.70	13.21	50%
TIPSTER 1	1	586.12	479.86	430.58	27%
TIPSTER	1	861.75	723.00	646.92	25%

I/O time is reported in Table 3. Again, the Mneme version without caching is faster than the B-tree version, and the Mneme version with caching is fastest.

For the end user, the reduction in wall-clock time is most significant. However, our goal was to demonstrate that the inverted file sub-system of an IR system could be efficiently supported by an "off-the-shelf" data management system. The system plus I/O times represent the time spent in the sub-system we have replaced, and the significant improvement shows that we have met our goal. It is also apparent from Tables 2 and 3 that as the collection becomes larger, the time spent in the inference engine starts to dominate the overall time, reducing the impact of improvements in system and I/O time.

To help explain the performance improvement of the Mneme versions, Table 4 gives some I/O statistics for each query set and INQUERY version. "I" is the number of I/O inputs measured with getrusage(), which counts the number of 8 Kbyte blocks actually read from disk. "A" is the average number of file accesses per inverted list lookup. Note that this does not represent actual disk activity since some file accesses are satisfied by the Ultrix file system cache. "B" is the total number of Kbytes read from the inverted list file during query processing. Again, this does not represent actual bytes read from disk since some file accesses are satisfied by the Ultrix file system cache.

We can make a number of observations from this table. The Mneme version without caching is faster than the B-tree version because it makes fewer accesses to the file (therefore fewer system calls) and, more importantly, fewer accesses to the disk. The B-tree version does limited and unsophisticated caching of index nodes, such that every record lookup requires more than one disk access. This problem gets worse as the file grows and the height of the index tree increases. Mneme, however, requires close to 1 file access per record lookup. Mneme locates objects based on their logical segments using compact multi-level hash tables. This lookup mechanism requires slightly more computation, but the reduced table size allows the auxiliary tables to remain permanently cached after their first access[4]. It is interesting to note that the Mneme version reads substantially more bytes from the file for the CACM queries than does the B-tree version. This is because the CACM queries generate more activity in the small and medium object pools, which have multiple objects clustered in physical segments. Accessing a given

[4] The TIPSTER collection requires only 512 Kbytes to cache all of the auxiliary tables.

Table 4. I/O statistics: I = I/O inputs, A = ave. file accesses / record lookup, B = total Kbytes read from file

Collection	Query Set	B-Tree			Mneme, No Cache			Mneme, Cache		
		I	A	B	I	A	B	I	A	B
CACM	1	82	1.89	585	63	1.02	1700	64	0.89	1496
	2	82	1.89	940	64	1.01	2430	64	0.85	2056
	3	83	1.44	2030	65	1.00	7890	65	0.45	3600
Legal	1	2747	2.92	20700	1626	1.07	20652	1625	0.96	17346
	2	2776	2.61	24526	1626	1.06	24668	1626	0.80	18594
TIPSTER 1	1	68280	2.89	503546	61308	1.03	503520	59917	0.60	271272
TIPSTER	1	96352	3.09	841304	87876	1.04	841516	84568	0.61	456062

object will cause the entire physical segment to be read in. This is less expensive than it appears because the physical segment size is tuned to the disk block transfer size. Each disk access causes 8 Kbytes to be read from disk, so in fact, based on the number of I/O inputs, the B-tree version transfers more raw bytes from disk even though it attempts to read far fewer bytes in the file.

Caching of inverted lists increases the performance of the Mneme version by further reducing the number of file and disk accesses. For CACM and Legal, the file system cache is able to satisfy enough file accesses so that there is no difference in "I" between the two Mneme versions. However, the reductions in "A" and "B" mean fewer system calls, less data copying between system and user memory space, and a savings in system CPU time. The TIPSTER collections are large enough that the Mneme version with inverted list record caching requires fewer I/O inputs than the versions that have file system caching only.

It is clear that caching of inverted lists to reduce disk accesses is advantageous, whether provided by the file system cache or the data management subsystem. It is also clear (and well known [17]) that caching provided by the file system is an inferior solution for data management problems. The buffer management requirements of inverted list data are better satisfied by the custom, domain tailored mechanisms in Mneme. The effectiveness of these caching mechanisms can be seen in Table 5, which shows the hit rates achieved in each of the buffers for each of the queries. The hit rates are fairly significant given that the buffer sizes allocated could be considered modest.

To further investigate the effects of buffer size, we measured the hit rates achieved in the large object buffer over a range of buffer sizes for the TIPSTER query set (see Fig. 2). The figure shows that increasing the buffer size gradually produces diminishing returns, but the knee of the curve can be used to guide buffer allocation.

5 Related Work

A great deal of work has been done in the area of supporting IR with a relational database management system (RDBMS). Some of the earliest work was done by Crawford and

Table 5. Buffer hit rates for the query sets

Collection	Query Set	Small Obj Buffer			Medium Obj Buffer			Large Obj Buffer		
		Refs	Hits	Rate	Refs	Hits	Rate	Refs	Hits	Rate
CACM	1	15	4	0.27	191	16	0.08	14	9	0.64
	2	11	2	0.18	191	17	0.09	25	17	0.68
	3	5	3	0.60	221	109	0.49	30	25	0.83
Legal	1	0	0	0.00	29	2	0.07	296	33	0.11
	2	0	0	0.00	35	9	0.26	366	95	0.26
TIPSTER 1	1	1	0	0.00	158	36	0.23	2112	938	0.44
TIPSTER	1	0	0	0.00	106	25	0.24	2137	923	0.43

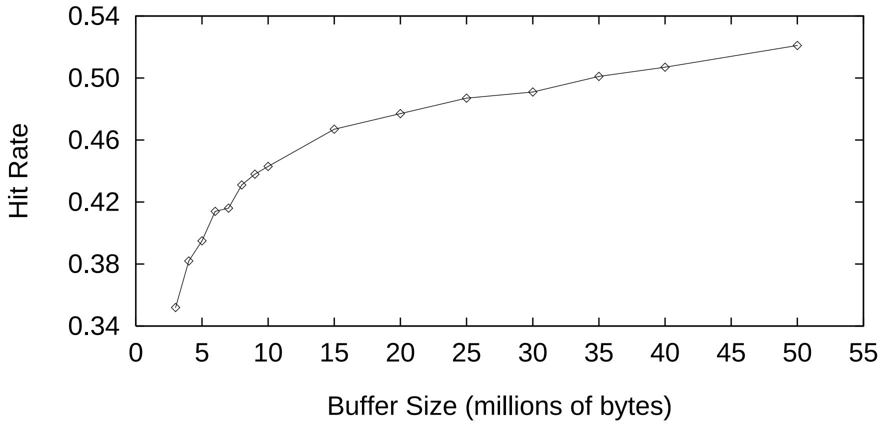

Fig. 2. Large object buffer hit rates for the TIPSTER query set

MacLeod [5, 12, 4, 13], who describe how to use the relational model to store document data and construct information retrieval queries. Similar work was presented more recently by Blair [1] and Grossman and Driscoll [9]. Others have chosen to extend the relational model to allow better support for IR. Lynch and Stonebraker [11] show how a relational model extended with abstract data types can be used to better support the queries that are typical of an IR system.

In spite of evidence demonstrating the feasibility of using a standard or extended RDBMS to support information retrieval, IR system builders have still chosen to build production systems from scratch. This is due to the belief that superior performance can be achieved with a custom system, a belief which is substantiated by a lack of results proving otherwise and anecdotal evidence. Additionally, most of the work described above deals only with document titles, author lists, and abstracts. Techniques used to support this relatively constrained data collection may not scale to true full-text retrieval systems. We desire to support full-text retrieval with high performance. Our approach, while similar in spirit to the above work, differs in both the data management technology chosen to support IR and the extent to which it is applied for that task. The data management technology we use is a persistent object store, and currently it is only used to manage an inverted file index.

Other work in this area has attempted to integrate information retrieval with database management [6, 16]. The services provided by a database management system (DBMS) and an IR system are distinct but complementary, making an integrated system very attractive. The integrated architecture consists of a DBMS component and a custom IR system component. There is a single user interface to both systems, and a preprocessor is used to delegate user queries to the appropriate subsystem. Additionally, the DBMS is used to support the low level file management requirements of the whole system. This architecture is similar to ours in that a separate data management system is used to support the file management requirements of the IR system. However, our data management system is a persistent object store and we focus on supporting high performance IR, with no support for traditional data management.

Efficient management of full-text database indices has received a fair amount of attention. Faloutsos [7] gives an early survey of the common indexing techniques. The two techniques that seem to predominate are signature files and inverted files, each of which implies a different query processing algorithm. Since the INQUERY system uses an inverted file index, and we are not interested in changing the query processing algorithm, we do not discuss signature files. Zobel et al. [24] investigate the efficient implementation of an inverted file index for a full-text database system. Their focus is on compression techniques to limit the size of the inverted file index. They also address updates to the inverted file and investigate the different inverted file index record formats necessary to satisfy certain types of queries. In our work, the format of the inverted file index records and the compression techniques applied to those records are pre-determined by the existing INQUERY system. Our approach is to replace the subsystem that manages these records, without changing the format of the records.

Tomasic and Garcia-Molina [19] study inverted file index performance in a distributed shared-nothing environment. Their simulation results show that caching inverted file index records in main memory can significantly improve performance. This is consistent with our results obtained from measuring an actual system, where the performance improvement of INQUERY integrated with Mneme is partly due to caching. This result implies that there is significant repetition of terms from query to query. This fact has severe implications for any IR study which assumes a uniform distribution over the term vocabulary when selecting query terms, such as the study in [18].

Properly modeling the size distribution of inverted file index records and the frequency of use of terms in queries is addressed by Wolfram in [21, 22]. He suggests that the informetric characteristics of document databases should be taken into consideration when designing the files used by an IR system. We have tried to take this advice to heart by developing appropriate file organization and buffer management policies based on the characteristics of the data and the data access patterns.

Buckley and Lewit [2] studied the effects of query optimization on an IR system and found that it can reduce the amount of data read from the inverted file. Reduced I/O could change the cache hit rates for the Mneme version of our IR system or alter the size distribution of inverted lists read by the system. We believe that these effects might require some adjustments in Mneme buffer sizes, but would not reduce the advantage of Mneme over the B-tree package.

6 Conclusions

Information retrieval systems development is reaching a point where further progress requires the use of more sophisticated data management services, such as concurrency control, dynamic update, and a complex data model. IR system builders are faced with the choice of developing these services themselves, or looking to "off-the-shelf" products to provide these services. Previous attempts at using standard DBMSs to provide these services have produced discouraging results due to poor performance. We have shown here that with the proper data management technology, sophisticated data management services can be supplied to an IR system by an "off-the-shelf" data management system without a performance penalty. In fact, the performance measurement results presented in Sect. 4 demonstrate that a performance improvement can be obtained.

Much of the performance improvement enjoyed by the Mneme version can be attributed to careful file allocation sympathetic to the device transfer block size and intelligent caching of auxiliary tables and inverted lists. While these features could be added to the B-tree package to achieve a similar improvement, it is exactly this type of effort we are trying to avoid by using an existing data management package.

Mneme offers other advantages besides data caching and smart file allocation. The extensibility of Mneme allows customization based on the characteristics of the data being stored. This capability is a clear advantage in an environment where the data management requirements are non-traditional, and was mandatory for satisfying the individual management needs of the different object groups in the inverted file. The more standard data management services provided by Mneme include recovery and support for a richer data model. Inter-object references allow structures such as linked lists to be used to break large objects into more manageable pieces. This could provide better support for inverted list updates and allow incremental retrieval of large lists.

The current version of Mneme is a prototype and does not provide all of the services one might expect from a mature data management system, such as concurrency control and transaction support. However, the nature of access to the data we are supporting here is predominately read-only. We expect that the addition of these services would not introduce excessive overhead or change the results reported above.

For future work we plan to implement some of the standard data management services not currently provided by Mneme and verify the above claim. We will also make use of the services that are currently provided by Mneme but not used to advantage above, such as the richer data model. Furthermore, it would be worthwhile to investigate other store and buffer organizations, looking for more opportunities to tune the system to the unique data management requirements of information retrieval.

References

1. D. C. Blair. An extended relational document retrieval model. *Inf. Process. & Mgmnt.*, 24(3):349–371, 1988.

2. C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proc. of the 8th Inter. ACM SIGIR Conf. on Res. and Develop. in Infor. Retr.*, pages 97–110, June 1985.
3. J. P. Callan, W. B. Croft, and S. M. Harding. The INQUERY retrieval system. In *Proc. of the 3rd Inter. Conf. on Database and Expert Sys. Apps.*, Sept. 1992.
4. R. G. Crawford. The relational model in information retrieval. *J. Amer. Soc. Inf. Sci.*, 32(1):51–64, 1981.
5. R. G. Crawford and I. A. MacLeod. A relational approach to modular information retrieval systems design. In *Proc. of the 41st Conf. of the Amer. Soc. for Inf. Sci.*, 1978.
6. J. S. Deogun and V. V. Raghavan. Integration of information retrieval and database management systems. *Inf. Process. & Mgmnt.*, 24(3):303–313, 1988.
7. C. Faloutsos. Access methods for text. *ACM Comput. Surv.*, 17:50–74, 1985.
8. E. A. Fox. Characterization of two new experimental collections in computer and information science containing textual and bibliographic concepts. Technical Report 83-561, Cornell University, Ithaca, NY, Sept. 1983.
9. D. A. Grossman and J. R. Driscoll. Structuring text within a relational system. In *Proc. of the 3rd Inter. Conf. on Database and Expert Sys. Apps.*, pages 72–77, Sept. 1992.
10. D. Harman, editor. *The First Text REtrieval Conference (TREC1)*. National Institute of Standards and Technology Special Publication 200-207, Gaithersburg, MD, 1992.
11. C. A. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. In *Proc. of the 14th Inter. Conf. on VLDB*, pages 306–317, 1988.
12. I. A. MacLeod. SEQUEL as a language for document retrieval. *J. Amer. Soc. Inf. Sci.*, 30(5):243–249, 1979.
13. I. A. MacLeod and R. G. Crawford. Document retrieval as a database application. *Inf. Tech.: Res. Dev.*, 2(1):43–60, 1983.
14. J. E. B. Moss. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, Apr. 1990.
15. G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.
16. L. V. Saxton and V. V. Raghavan. Design of an integrated information retrieval/database management system. *IEEE Trans. Know. Data Eng.*, 2(2):210–219, June 1990.
17. M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, July 1981.
18. A. Tomasic and H. Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. Technical Report STAN-CS-92-1434, Stanford University Department of Computer Science, 1992.
19. A. Tomasic and H. Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. In *Proc. of the ACM SIGMOD Inter. Conf. on Management of Data*, Washington, D.C., May 1993.
20. H. Turtle and W. B. Croft. Evaluation of an inference network-based retrieval model. *ACM Trans. Inf. Syst.*, 9(3):187–222, July 1991.
21. D. Wolfram. Applying informetric characteristics of databases to IR system file design, Part I: informetric models. *Inf. Process. & Mgmnt.*, 28(1):121–133, 1992.
22. D. Wolfram. Applying informetric characteristics of databases to IR system file design, Part II: simulation comparisons. *Inf. Process. & Mgmnt.*, 28(1):135–151, 1992.
23. G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley Press, 1949.
24. J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proc. of the 18th Inter. Conf. on VLDB*, Vancouver, 1992.

This article was processed using the LaTeX macro package with LLNCS style