Integrating IR and RDBMS Using Cooperative Indexing

Samuel DeFazio, Amjad Daoud, Lisa Ann Smith, and Jagannathan Srinivasan
Oracle Corporation

Bruce Croft and Jamie Callan
University of Massachusetts at Amherst

Abstract

The full integration of information retrieval (IR) features into a database management system (DBMS) has long been recognized as both a significant goal and a challenging undertaking. By full integration we mean: i) support for document storage, indexing, retrieval, and update, ii) transaction semantics, thus all database operations on documents have the ACID properties of atomicity, consistency, isolation, and durability, iii) concurrent addition, update, and retrieval of documents, and iv) database query language extensions to provide ranking for document retrieval operations. It is also necessary for the integrated offering to exhibit scaleable performance for document indexing and retrieval processes. To identify the implementation requirements imposed by the desired level of integration, we layered a representative IR application on Oracle Rdb and then conducted a number of database load and document retrieval experiments. The results of these experiments suggest that infrastructural extensions are necessary to obtain both the desired level of IR integration and scaleable performance. With the insight gained from our initial experiments, we developed an approach, called *cooperative indexing*, that provides a framework to achieve both scaleability and full integration of IR and RDBMS technology. Prototype implementations of system-level extensions to support cooperative indexing were evaluated with a modified version of Oracle Rdb. Our experimental findings validate the cooperative indexing scheme and suggest alternatives to further improve performance.

1 Introduction

Providing information retrieval (IR) features within a database management system (DBMS) framework is a highly desirable goal. Many (if not most) of the applications involving text databases also have a major component involving structured data and require the types of guarantees about concurrency control and recovery that are provided in commercial database systems.

There are a number of technical challenges associated with the integration of IR and database systems. Of these, the most significant obstacles include supporting transaction semantics for IR operations, providing scaleable indexing and query processing for large text databases, and extending database query languages to accommodate ranking. Providing transaction semantics for IR systems in a database environment involves enabling the DBMS to handle document storage, update, indexing, and retrieval. Some current "integrated" systems actually consist of a complete IR system and a DBMS with the integration mainly occurring in a common interface. In this approach, the IR system maintains completely separate index structures on the text components of the database and the IR portion of a query is evaluated using these indices. The separate text indices are very difficult to coordinate with documents stored in the database system, and guaranteeing concurrency control and recovery is virtually impossible. The alternative is to store text indices in the database itself [BLAIR88, CROFT85, HARPER92, LYNCH88, MCLEOD83]. Under this approach, performance is a problem when the indexing mechanisms provided by the DBMS do not satisfy the requirements of full-text IR systems. This is generally the case when any commercial relational database management system (RDBMS) is used. In the work reported by [LYNCH88] and [HARPER92], the DBMS-provided indexing features seem to more closely satisfy IR system requirements. However, we are not aware of any studies that discuss either the performance or scaleability of these approaches.

To accommodate large collections of text, database systems that provide IR functionality must deliver scaleable performance for indexing and retrieval transactions. Thus, the initial focus of our work is scaleability, as opposed to raw performance. Tuning and/or additional hardware (which continues to decrease in cost dramatically) can frequently be used to improve any performance metric of interest. However, scaleability for DBMS products is directly related to system-level capabilities and algorithm design.

Extending database systems to support document retrieval predicates has major implications with respect to query language design and query evaluation. The incorporation of uncertainty (e.g., probabilities) into database query languages results in a more general information system and is essential for effective information retrieval. It is difficult, however, to change current database systems. Despite some proposals in this area [FUHR90], most integration efforts impose a Boolean filter on the ranked results of IR queries before incorporating those results in the standard database query processing. An example of this would be to treat the top N ranked text objects as the retrieved set. Emerging database standards such as SQL/Multimedia [SQLMM], while acknowledging the importance of ranking, are oriented towards the Boolean logic view of retrieval. Although extending SQL to handle IR query semantics is important, it is outside our current focus and will not be discussed further.

SIGIR'95 Seattle WA USA© 1995 ACM 0-89791-714-6/95/07.$3.50

The overall goal of our work is to develop a general DBMS framework that enables an efficient and effective integration of IR technology. This paper concentrates on identifying the system-level extensions that enable RDBMS products to provide transaction semantics for IR operations including document storage, update, indexing, and retrieval along with scaleable performance. We are particularly interested in formulating RDBMS extensions to provide a generic approach for supporting different types of IR systems. A simplistic viewpoint would be that the RDBMS provides support for inverted text indices, since this is the overwhelming choice of IR system implementors. However, since different IR systems have specific approaches to the extraction of index terms from documents and the type of data stored in a text index, it is important that the indexing support provided by database systems not be defined too narrowly. We describe an approach called *cooperative indexing*, where the IR components of the system define what is extracted from documents along with the related index structure, and the database system provides efficient access to the index.

The cooperative approach is important not only for the initial indexing of the text, but also for maintaining consistency as the text is updated. The update problem has been largely overlooked in traditional IR, with the emphasis being on periodically adding batches of new documents. Document modification and deletion are rarely mentioned. In a number of important applications involving collaborative authoring, the situation is significantly more dynamic. Parts of documents will be added, modified and deleted very rapidly, and there will be a significant need for real-time updates to the text indices associated with these documents.

The following section describes the experimental approach that we employed to study the integration of IR technology into Oracle Rdb[ORACLERdb]. In Section 3, we discuss our initial database load and query experiments using a representative IR system that is layered on Oracle Rdb. The lessons learned from this study led to the cooperative indexing approach, which is described in Section 4. In Section 5, we present the experimental findings for the prototype implementation of cooperative indexing and discuss the implications.

2 Experimental Approach

To help identify the system-level extensions that facilitate the integration of RDBMS and IR technology, the following experimental approach was employed:

- *Characterize the IR Workload.* Using Oracle Rdb Version 6.1, we conducted experiments to characterize the workload generated by indexing and retrieval transactions. These experiments were driven with SQL-based application programs that load the database and execute queries. For this study, the text index structure was mapped to an Oracle Rdb table.
- *Develop an Experimental Version of Oracle Rdb.* An experimental version of Oracle Rdb was developed that contains extensions to improve the scaleability of indexing and retrieval transactions, and reduce the secondary storage requirements for text indices. This version of Rdb is not production quality software, and served essentially as a mechanism to help validate the extensions under consideration.
- *Evaluate the Extensions.* Using the modified version of Rdb, we again ran the database load and retrieval experiments. The resulting workload characteristics were compared to the initial experiments to evaluate the proposed extensions.

It should be noted that minimal effort was extended to "tune" Oracle Rdb or the application software. As such, our experimental findings do **not** reflect the actual performance properties of Oracle Rdb for IR applications. Given this, the transaction times presented below are expressed in relative units of measure. The intent of these experiments was simply to characterize the workload and evaluate the proposed extensions. During subsequent work, we plan to measure the performance of Oracle Rdb for IR applications using published benchmark specifications [DEFAZIO93].

2.1 Test Data

The data for our experiments was extracted from the *News* collection [TOMASIC93]. In Table 1, we present some of the statistical properties for this corpus. It should be noted that the *News* collection exhibits typical word usage properties. That is, the word usage patterns can be characterized with a Zipf [ZIPF49] distribution.

Table 1: Statistical Properties of the *News* Collection

Total Raw Text	686 Mbytes
Total Documents	138,578
Average Document Size	4,950 Bytes
Total Unique Words	788,256
Total Word Occurrences	48,526,577
Average Occurrences per Word	61
Frequent Words	39,413
Infrequent Words	748,843
Frequent Word Occurrences	93.6%
Infrequent Word Occurrences	6.4%

2.2 Database Schema

The database design we employ for our experiments is representative of the table-based schemes that are used with RDBMS products to build IR systems. This database design supports the probabilistic IR approach [CALLAN92] and consists of the following two tables:

DOCUMENTS. This table contains the *News* articles and the related descriptive (meta) data. Each row represents one article and has the following attributes:

- *text* - varying length character string (64,000 byte maximum) containing the content of one *News* article.
- *ID* - four byte integer containing a unique identifier for the *News* article.
- *maxfreq* - two byte integer containing the maximum word usage frequency for the article. This value is used as the normalizing factor for the probabilistic IR scheme being modeled. Note that we could have selected some other normalizing factor such as the length of the *News* article.
- *unqwords* - two byte integer containing the number of unique words in the *News* article

DOCINDEX. This table contains an inverted index for the database. Each row represents one index entry and is composed of the following attributes:

- *token* - fixed character string (20 characters in length) containing the token itself.
- *ID* - four byte integer containing a unique identifier for the *News* article.
- *frequency* - two byte integer containing the frequency of the token within the *News* article.
- *occlist* - varying length character string (512 byte maximum size) containing the occurrence data that corresponds to this token within the associated *News* article.

Observe that for the DOCINDEX table, the scope of each index entry is a single document, as opposed to the entire database. This design reflects our desire to model data structures that support document-level update and delete operations. That is to say, documents can be deleted or updated without rebuilding the entire index structure. Notice also that the occurrence list is stored in a varying length character string with a maximum size of 512 bytes. This may seem very small, however, 512 bytes easily handles the occurrences for one word within a single *News* article. For other collections of text, additional storage may be required. In such cases, the string length could be increased to 64 Kbytes, or even larger (i.e., multiple terabytes) by using an Oracle Rdb Binary Large Object (BLOB).

To improve data retrieval performance, Oracle Rdb provides both SORTED (B-tree) and HASHED indices that may be defined on one or more attributes of a table. Indices that include multiple attributes are termed *multi-segment.* An index defined for only one attribute is called *single-segment.* For the experiments discussed below, we employ both single and multi-segment indices on the database tables.

2.3 Driver Software

The experiments that we conducted were driven with two SQL Module Language [ORACLERdb] programs that are representative of existing software for integrating IR and RDBMS technology. One of these programs, called *TextLoad,* builds the experimental database. This program reads *News* articles from disk, tokenizes the text, and builds the set of index entries for the DOCINDEX table. Using calls to SQL Module procedures, a single insert action adds the *text* and related attributes to one row of the DOCUMENTS table. Before insertion into the database, the *text* attribute is compressed using a Huffman [HUFFMAN52] encoding scheme. After a document is tokenized, the resulting words are filtered through a stop word list *and stemmed* to generate the text index entries. One or more inserts are performed to load these index entries into the DOCINDEX table. Prior to insertion the text index entries are sorted. This sorting improves insert performance by increasing the locality of database write operations. Also, the *occlist* attribute of the DOCINDEX table is compressed using a simple variable length encoding scheme.

The other SQL Module Language program, *TextRetrieval*, is designed to be representative of software that selects documents of interest from the database. A query is presented to this software as a list of words. For each word, the corresponding entries in the DOCINDEX table are selected using SQL Module Language procedures. To model a probabilistic IR system in a dynamic environment, documents are indexed as separate entities independent of the collection. That is, the unit of inversion is a document, not the entire *News* collection. Term frequency and maximum term frequency within the document are pre-computed and stored in the database tables. At search time, the other statistics required to generate ranked output (e.g. IDFs) are computed.

2.4 Computing Environment

The experiments that we conducted were performed on a dedicated DEC Alpha 7000, Model 720 machine operating under OpenVMS Version 6.1. The hardware configuration included one 275 mhz Alpha processor, 1,024 Mbytes of memory, and approximately 10 Gbytes of disk storage. Oracle Rdb operated with an I/O size of 8 Kbytes, a database buffer cache of 128 Mbytes, and a maximum process address space of 256 Mbytes. Our experimental database was striped across three DEC RZ28 disks, each having 2 Gbytes of capacity and mean access time of about 10 milliseconds. The driver software for our experiments was written in DEC C Language.

3 Initial Experiments

To characterize the workload for our IR application, we conducted multiple database load and retrieval experiments. The load experiments build the database, and can be viewed conceptually as a sequence of large batch transactions. Each load transaction inserts 100 *News* articles into the database. This mode of operation was selected to model the operation of an incremental "batch" update program. Clearly, the selection of 100 articles per load transaction is an arbitrary choice on our part. This choice, however, does represent a compromise between loading each article under a distinct transaction, and loading all the articles as one transaction.

The retrieval experiments select documents from the database using a set of sample queries. Each query is processed as a distinct transaction and consists of 6 tokens that were selected from the database vocabulary. The query tokens include high, medium, and low use words, where each word is assumed to have equal weight. The *TextRetrieval* program located all documents containing one or more of the query tokens, and produced a ranked answer set. Each retrieval experiment executed the same 10 queries, and started with an empty database buffer. For this set of queries, we measured the elapsed times with and without retrieving the *occlist* attribute of the DOCINDEX table.

One set of load and retrieval experiments operated with a single-segment index, called *TokenInd*, defined on the *token* attribute of the DOCINDEX table. The other set of experiments had a multi-segment index, termed TokenIdFreqInd, defined for the DOCINDEX table. This index includes the token, ID, and frequency attributes. The multi-segment index configuration was employed to study the performance implications of storing the data required for ranking as part of the key for each B-tree entry. For both sets of retrieval experiments, a single-segment index, *MaxFreqInd* was defined on the *maxfreq* attribute of the DOCUMENTS table.

3.1 Database Scaling

The load and retrieval experiments are focused on determining the scaling properties for the related transactions. For database systems, *batch scaleup* is usually defined as executing the same transaction on a database that grows in size by a factor of N [DEWIT92]. The batch scaleup for a transaction is said to be *linear* whenever the database size and processing time increase by a factor of N while holding the computing resources constant. For example, when the amount of database text doubles in size, the load transaction would scaleup linearly only if the processing time on the same computer system increased exactly by a factor of two.

When transactions use a B-tree index to retrieve data, one expects sublinear batch scaleup. This is because the number of B-tree levels equals the *log* of the number of keys. If we assume that doubling the amount of text increases the number of index entries approximately two fold, then the growth in levels for the corresponding B-tree index will be substantially less. Thus, one would expect to observe sublinear batch scaleup properties for our retrieval transactions.

3.2 Initial Experimental Findings

Summary statistics for the initial database experiments are given in the following table.

Table 2: Summary Statistics for Each Initial Case

Total Documents	30,000
Total Raw Text	156 Mbytes
Total Unique Words	199,433
Total Word Occurrences	7,639,928
Total Load Transactions	300
Total Retrieval Transactions	120

Figure 1 displays the load transaction times versus the amount of source text indexed under both single and multi-segment indices on the DOCINDEX table.

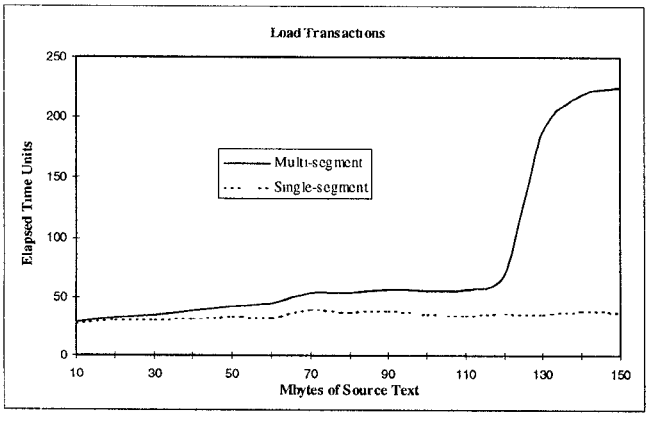

Figure 1. Initial Load Transaction Times.

Notice that in Figure 1 the transaction times for the single and multi-segment index cases are nearly identical for data sizes less than 30 Mbytes. When the amount of text exceeds 110 Mbytes in size, elapsed times for the multi-segment load transactions begin to increase much more rapidly. Observe that in both cases, the load transactions exhibit sublinear scaleup when the amount of source text loaded is less than 100 Mbytes. However, in the multi-segment case the load transaction exhibits an epoch of nonlinear behavior after the data size exceeds 110 Mbytes, and then returns to a more uniform growth pattern. This behavior can be explained by considering the structure of an Oracle Rdb B* tree [GRAY93].

As shown in Figure 2, the B* tree structure used by Oracle Rdb has duplicate nodes to handle multiple occurrences of the same token.

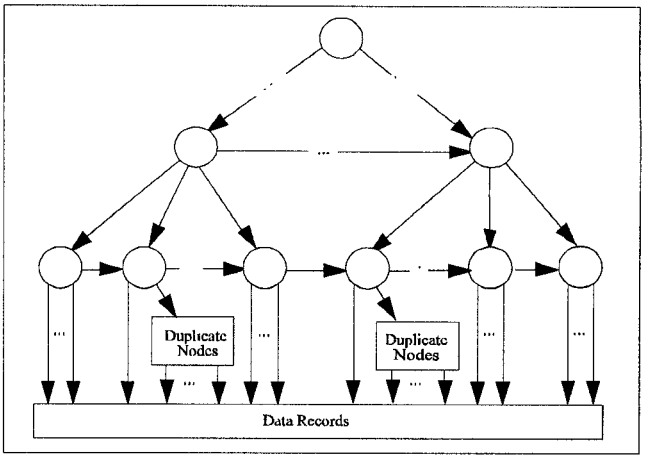

Figure 2. Logical Structure of an Oracle Rdb B* tree.

For the multi-segment case, the B* tree has many more interior and leaf nodes, and no duplicate nodes. Each of the load transactions accesses many of the related index pages. At some point, the amount of buffer space is exhausted, and performance degrades rapidly. After the buffer management algorithm adjusts to this situation, the growth in load transaction times seems to stabilize. However, by this point the load transaction times have increased by nearly an order of magnitude. In effect, this behavior occurs because as the B* tree grows, an increasing number of nodes must be flushed from the buffer per transaction. In the single-segment case, after the database reaches a certain size, most of the B* tree growth occurs in the duplicate nodes. As such, the database buffer pool can accommodate the interior and leaf nodes far longer than for the multi-segment case. Clearly, as the database grows, at some point the single-segment B* tree size must also exceed the amount of available buffer space. However, after the database reaches a certain size, the introduction of new vocabulary for most text sources tends to decrease significantly [CHAPMAN90]. Thus, we have reason to believe that the sublinear scaleability exhibited by the load transaction under single-segment indexing will hold for much larger databases.

The storage requirements for the experimental database are shown in Table 3. Notice that the total storage requirement for the database is roughly 3 times the source text size. As expected, the multi-segment index configuration requires more storage than single-segment index case. This follows since the multi-segment index entries are much larger in size than for the single-segment case.

Table 3: Initial Storage Requirements

Total Source (text)	156 Mbytes
DOCUMENTS (table)	86 Mbytes
MaxFreqInd (index)	0.5 Mbytes
DOCINDEX (table)	231 Mbytes
TokenInd (index)	101 Mbytes
TokenIdFreqInd (index)	160 Mbytes

Figure 3 shows the average elapsed time required for retrieval transactions at various database sizes.

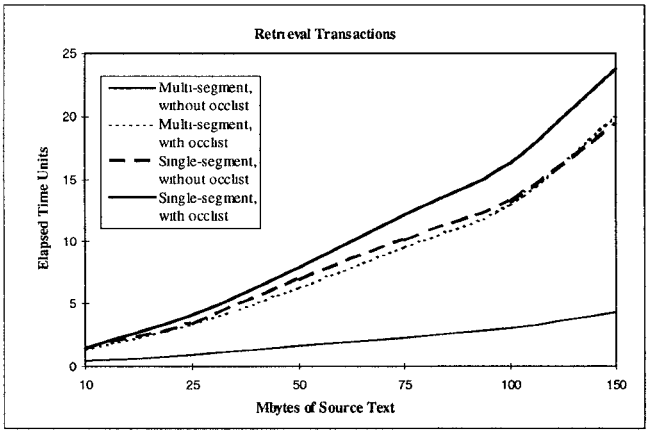

Figure 3. Initial Retrieval Transaction Times.

Without returning the *occlist* attribute, the retrieval transactions execute much faster in the multi-segment case. In fact, this is the only case where the retrieval transaction times exhibit the expected sublinear scaleup properties. The large difference in time among the cases is a consequence of accessing the DOCINDEX table to satisfy the query. That is, in the multi-segment case, the data attributes required for computing ranks; namely, token, document identifier, and token frequency reside in the B-tree. Thus, an index-only retrieval is sufficient. For the single-segment index, after the B-tree is searched the related records must be fetched from the DOCINDEX table.

Notice that when the *occlist* attribute, which resides only in the DOCINDEX table, is returned to the *TextRetrieval* program, we obtain relatively equivalent performance for the single and multi-segment cases. For this experiment, even with a multi-segment index, the DOCINDEX table must be accessed and the related performance penalty is significant.

3. 3 Discussion of Initial Results

Based on our initial experimentation we make the following observations:

- Using a physical table to store the text index does not yield scaleable performance for both load and retrieval transactions. As the database grows, load transaction times increases nonlinearly in the multi-segment case, and sublinearly when a single-segment index is employed. With multi-segment indexing, we observed the lowest retrieval transaction times and sublinear scaleability when the *occlist* attribute was not selected. As such, the best overall performance occurs with a single-segment index for loading, and multi-segment indexing for retrieval when the *occlist* attribute is not selected. Thus, our goal is to develop an indexing mechanism for Oracle Rdb that combines the best of these schemes, and is also capable of returning occurrence data to retrieval transactions with the minimum possible penalty.
- Being about three times the size of the source text, the amount of storage required for a table-based text index is high. By storing the text index structures entirely within a B-tree (analogous to the multi-segment case including the *occlist* attribute), we could eliminate the need for a physical DOCINDEX table and reduce storage costs significantly.

In the following section, we introduce the notion of *cooperative indexing*. The implementation that we propose for cooperative indexing contains a set of RDBMS extensions that are designed to enable highly scaleable load and retrieval transactions. These extensions also reduce the storage demand by eliminating the need for a physical text index table. As we shall see, however, the total storage required for our prototype remains larger than the space overhead for efficient file-based IR systems.

4 Cooperative Indexing

For simple data types such as integers and small strings, all aspects of indexing can easily be handled by the DBMS. This is not the case for content-based indexing of documents, images, video clips, and other complex forms of data. The essential reason is that complex data types have application-specific formats and indexing requirements. For the DBMS to effectively accommodate complex data objects, it is necessary to support application-specific indexing techniques. The approach we employ to satisfy this requirement is termed *cooperative indexing*. For cooperative indexing, an application and the DBMS share the responsibility for building and interpreting the index structures. In this scheme:

- Each index entry logically consists of a *key* along with application-defined *referent* and *data* attributes. The *referent* is a unique value by which the DBMS identifies the related database record. The *data* attributes contain application-specific indexing values.
- The application builds and interprets the index records, while the database system provides access to the index entries.

In effect, the application controls semantic content of the index, and the database system handles the physical storage of the related data structures. Notice that the cooperative indexing model assumes that the content for any index entry can be represented as a tuple. More specifically, a cooperative index is modeled as a table. The data elements for such tables define the structure of the related index entries.

To illustrate this concept, consider the DOCINDEX table described above. The attributes of this table could be mapped into a cooperative index as follows:

- *token* is the *key* attribute,
- *ID* is the *referent* attribute, and
- *frequency* and *occlist* are the *data* attributes.

For database updates, the IR indexing software parses the text, builds the index entries, and modifies the cooperative index. On retrieval, the IR query processing software obtains the index

entries for the given query terms, interprets the content, and determines the documents that satisfy the selection criteria. The resulting set of *IDs* is used to retrieve and possibly rank the matching documents.

Under the cooperative indexing model, the application software controls updating the index structures. Notice that this differs significantly from most DBMS provided indexing mechanisms. That is, when an index is defined on some database attribute, the DBMS updates the related structures as a side-effect of record addition, deletion, or modification. For cooperative indexing, the application assumes this responsibility.

The cooperative indexing scheme has the following important properties:

- *Data Integrity.* Updates to the index structures occur under transaction control. Thus, the level of data integrity provided by the cooperative indexing scheme is analogous to that associated with typical DBMS indexing mechanisms.
- *Data Concurrency.* Using a table to model an IR index requires two data structures. That is, inverted list entries are stored as rows in a table, and the DBMS maintains an index on one or more of the related table attributes. With cooperative indexing, the IR index resides in one data structure that is managed by the DBMS. This reduces the amount of locking during IR index updates and, therefore, enables the DBMS to provide increased levels of data concurrency.
- *Data Consistency.* The cooperative indexing scheme enables the application to select the desired level of data consistency. For example, an application may update a cooperative index at some point in time after textual content is added, deleted, or modified. In such cases the index becomes inconsistent, and full-text queries may not cover portions of the database. Some applications may require the IR indices to always be logically consistent with the associated database content. In this case, an application can elect to operate in a manner analogous to the typical DBMS indexing mechanisms. That is, modify database content and the cooperative index within the context of one update transaction. Notice that in either case, the index is updated under transaction control and database integrity is guaranteed by the DBMS.

In addition to the properties mentioned above, the cooperative indexing approach benefits from operating completely within a database framework. Thus, under this model of integration, IR applications can benefit fully from database system features such as integrated backup and recovery, security, replication, and so on.

4.1 RDBMS Extensions for Cooperative Indexing

Given the generality of our cooperative indexing model, there are many possible implementations. We have selected an approach that is based on the following assumptions:

- The DBMS must provide type-specific access methods for supporting cooperative indexing. That is to say, indexing based on typical B-tree or hashing schemes are incapable of handling content-based selection of complex objects such as documents, images, video clips, and so on.
- The native application programming interface for the DBMS must include support for cooperative indexing.

Working under these assumptions, we developed a prototype implementation of Oracle Rdb that has the following extensions:

- *New Sorted Access Method.* This new access method is modeled after a B+ tree [GRAY93]. That is, the *referent* and *data* attributes are stored in the B-tree. The primary goals of this access method are to more efficiently handle the occurrence lists associated with IR indexing technology, and eliminate the need to use physical tables for storing IR index structures.
- *INDEX ONLY Tables.* This feature permits the user to define a table that represents the structure of an index record. For such tables, the attributes are stored entirely within the physical data structures for the index. In effect, an INDEX ONLY table is a logical construct for defining and accessing index content with SQL statements.

It is worth noting that our implementation of cooperative indexing is very general. More specifically, the scope and content of each index entry is not restricted. Thus, we could design a cooperative index where the occurrence data for each key spans the entire database (much like typical inverted files). In the experiments discussed below, each index entry represents the token occurrences on a per document basis. The reason for selecting this level of granularity is to model data structures where the access method can directly support document deletion and updating. For example, using an INDEX ONLY table analogous to DOCINDEX, we could directly delete all entries having a given *ID* value.

5 Prototype Experiments

Our prototype implementation of cooperative indexing was evaluated using the extended version of Oracle Rdb discussed above. Unfortunately, the development of prototype software to support index delete and modification operations was beyond the scope of this effort. These features will be included in the production implementation of cooperative indexing for Oracle Rdb.

To ensure an equivalent evaluation of our extensions, the load and retrieval experiments described in Section 2 were employed. This time, however, we used the modified SORTED access method along with an INDEX ONLY table for indexing the text. The INDEX ONLY table definition for the database is described in Section 4.0. The results of these experiments along with a comparative performance analysis are presented below.

5.1 Prototype Experimental Findings

Summary statistics for the database experiments using the prototype version of Oracle Rdb are the same as those presented above in Table 2. Figure 4 displays the load transaction times versus the amount of source text for the prototype version of Oracle Rdb and the initial single-segment index experiment. Notice that after the source data size exceeds 60 Mbytes, the prototype load transaction times begin to increase rapidly. Then, after about 90 Mbytes of text is loaded, elapsed times for the prototype load transactions stabilize. This behavior is similar to what we observed in the initial experiments for the multi-segment load transactions. The major difference is that the elapsed time increased much less for the prototypes as compared to the multi-segment case. As with the multi-segment load

experiments, this behavior occurs when it is no longer possible to cache the entire index structure in database buffers. However, in this case the situation is somewhat different.

For a B+ tree, growth tends to occur mostly in the leaf nodes, and each load transaction accesses a large number of these nodes. The load transactions adds index entries in sorted order. As the transaction proceeds, pages are flushed from the buffer under a least recently used (LRU) replacement algorithm. After one load transaction completes, most of the B+ tree pages required to access entries that occur early in the alphabet have been flushed. In the following transaction, many of these pages must be reloaded from disk. Thus, as the B+ tree grows, more I/Os are required per load transaction. Fortunately, this cyclic sequential reference behavior is well understood [HAWTHORN79]. Also, there are known extensions to the LRU buffer replacement algorithm to address this behavior [GRAY93].

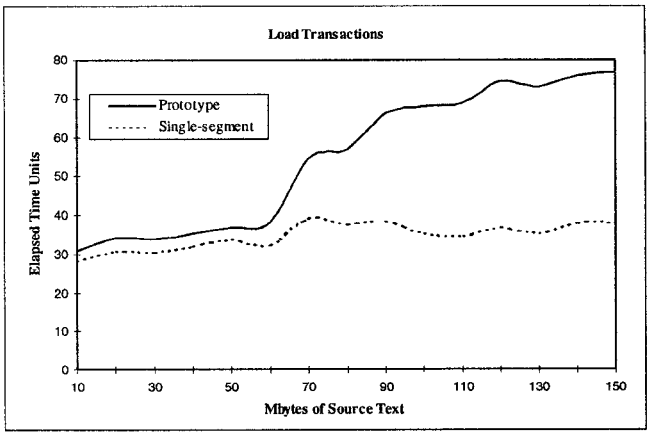

Figure 4 Load Transaction Times.

Notice that the prototype load transactions scale linearly or better for most regions of the curve shown in Figure 4. Generally speaking, this result is encouraging. With enhancements to the buffer replacement algorithm, we expect to see significant improvements for load transaction times. Also, there are other performance improvement opportunities for the prototype load transactions, the most important of which is I/O reduction. In Table 5, we show the average I/O per load transaction across the experiments. As shown in Table 5, on average the prototype generates significantly more database writes per load transaction than the initial single-segment case. This additional I/O is directly related to the large number of B+ tree pages that must be written per transaction. It is worth noting that adding the same amount of text index entries to the physical DOCINDEX table causes far fewer database pages to be modified. This is because writing records to a physical table typically appends data to the end of a contiguous storage area. However, adding the same amount of data to a B+ tree causes more discontiguous pages to be modified. As with the buffer replacement problem, there are known clustering techniques for reducing the amount of discontiguous allocations for related nodes in the B+ tree [GRAY93, TOMASIC93]. Also, the higher read rate for the prototype load transactions can be reduced by employing an enhanced buffer replacement algorithm as discussed above.

Table 5: Average Database I/O per Load Transaction

	Single-segment	Prototype
Average Reads	157	1816
Average Writes	3290	4420

For various source data sizes, Figure 5 shows the average elapsed time required for retrieval transactions from the prototype implementation and the initial multi-segment case.

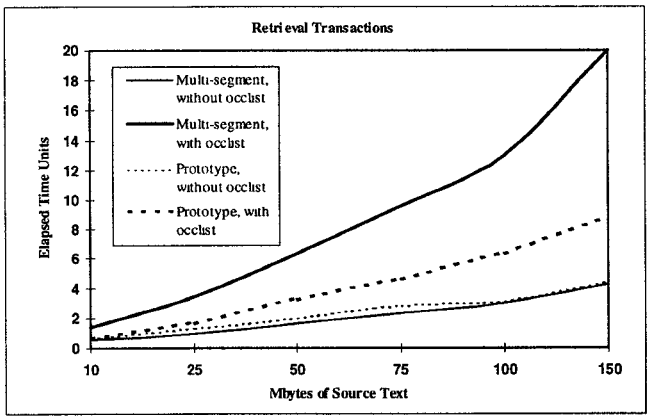

Figure 5. Retrieval Transaction Times..

From the curves in Figure 5 we find that when the *occlist* attribute is not selected, the prototype exhibits nearly identical retrieval transaction times as compared to the multi-segment case. When the *occlist* attribute is selected, the prototype retrieval transactions require much less time than for the multi-segment case. Notice that in all cases, the prototype retrieval transactions exhibit sublinear scaleup properties.

The storage requirements for the experimental database constructed with the prototype version of Oracle Rdb are shown in the following table.

Table 6: Prototype Database Storage Requirements

Total Source (text)	156 Mbytes
DOCUMENTS (table)	86 Mbytes
MaxFreqInd (index)	0.5 Mbytes
Cooperative (index)	200 Mbytes (estimate)

From Table 6, notice that the total storage requirement for our cooperative indexing scheme is roughly 2 times the source text size. This storage savings, as compared to the initial case, results from eliminating the need to redundantly store tokens in the DOCINDEX table.

Considering the results of our prototype collectively, we are encouraged. Although we did not achieve the desired performance for the load transactions, it is only because the necessary changes to Oracle Rdb are beyond the scope of our prototype efforts. In terms of retrieval transactions, we clearly demonstrated that the new access method exhibits highly

desirable scaleability properties. Also, the performance penalty for selecting occurrence data is much more reasonable than for the multi-segment case.

5. 2 Lessons Learned

In the process of building our Oracle Rdb prototype system, we made significant progress toward gaining the understanding required to effectively integrate IR and RDBMS technology. The most important lessons learned from this undertaking include:

- *The text workload differs considerably from that generated by traditional RDBMS operations.* Existing RDBMS technology has been highly optimized to handle tables and the related operations. The set of operations for text impose considerably different demands on the database system. For example, the B* tree indexing used by Oracle Rdb exhibits excellent performance properties for very large relational databases. However, we observed that this is not the case for text. The reason being one of scale. That is, documents generate about two orders of magnitude more index keys per byte of data. To obtain scaleable performance, access method enhancements are required. We have found that by using a B+ tree, it is possible to obtain scaleable performance. This is because the tree grows essentially at the leaf nodes. Thus, after the arrival of new vocabulary begins to fall off as the database grows, the number of B+ tree operations for indexing and retrieval increases at a sublinear rate.
- *Enhanced buffer management algorithms are essential.* The LRU buffer management scheme used by most RDBMS products must be enhanced to handle the cyclic sequential access patterns for index pages that result from a series of document load transactions. This behavior is generally handled by RDBMS products for operations on tables that require multiple scans (e.g., non-key join). However, this behavior is not typically associated with accessing RDBMS index structures.
- *Performance improvements will come with time.* Naively, one may claim that some IR-specific tree can be used to obtain highly desirable performance for text operations. This is simply not the case for RDBMS products. The interaction between RDBMS access methods, concurrency control, and recovery is highly complex [LOMET91]. It has taken years to understand this interaction, and only for a limited number of indexing methods such as B-trees and hashing. Thus, it is necessary to evolve the current RDBMS access methods to accommodate text and, consequently, we can expect to see continual performance improvement over time.

As one final observation, we mention that this work has been very challenging. Most of this challenge is directly related to developing RDBMS kernel level support for a significantly different indexing paradigm.

6 Conclusions

Commercial RDBMS products available today do not provide highly integrated support for IR technology. In addition to this, many of the existing IR systems do not efficiently support document *update* operations. We have developed a prototype version of Oracle Rdb that implements a generic framework, called cooperative indexing, that enables the effective integration on IR technology and scaleable performance. The performance studies that we conducted with an "untuned" prototype version of Oracle Rdb yielded scaleable performance for database load and retrieval transactions. Although these studies were conducted with a relatively small collection of text, we expect similar performance for much larger databases.

Historically, IR system design has concentrated heavily on reducing storage requirements. This work has yielded numerous techniques for building highly compressed index structures. The use of such data structures, however, makes the integration of IR and DBMS nearly impossible. The primary problem derives from the fact that any modification to the source text typically forces a complete rebuild of compressed index structures. In a dynamic database environment, the overhead of a complete index rebuild and the impact on concurrent data access are unacceptable. With the rapidly decreasing costs for magnetic disks, we claim that IR systems can handle "reasonable" increases in secondary storage demand in order to obtain the benefits of database functionality.

Thus far, we have not addressed the extension of database query languages to include the notion of uncertainty. The evolving SQL/MM standard [SQLMM] is woefully lacking in this regard. To properly export IR functionality in a RDBMS context, SQL and the database infrastructure must be appropriately extended. Given that our work has demonstrated the viability of physically integrating IR and RDBMS, we will now begin addressing the RDBMS extensions that are required to adequately support uncertainty in query processing.

Acknowledgments

We would like to thank Rabah Mediouni of Oracle for his assistance in building the test systems and conducting the performance experiments. Also, we would like to thank those who reviewed the paper for their helpful comments.

References

[BLAIR88] D.C. Blair. An Extended Relational Retrieval Model. IPM, Vol. 24, No. 3, pp. 349-371, 1988.

[CALLAN92] J.P. Callan, W.B. Croft, and S.M. Harding. The INQUERY Retrieval System, Proceedings, Third International Conference on Database and Expert Systems Applications, pp. 78-83, Valencia, Spain, Springer-Verlag, 1992.

[CHAPMAN90] D. Chapman and S. DeFazio. Statistical Characteristics of Legal Document Databases. Technical Report, S&PT, Mead Data Central, Dayton, Ohio, January, 1990.

[CROFT85] W.B. Croft and T.J. Parenty. A Comparison of a Network Structure and a Database System used for Information Retrieval. Information Systems, Vol. 10, No. 4, pp. 377-390, 1985.

[DEFAZIO93] S. DeFazio. Full-text Document Retrieval Benchmark. In Jim Gray , editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, Chapter 8. Morgan Kaufmann, Second Edition, 1993.

[FUHR90] N. Fuhr. A Probabilistic Framework for Vague Queries and Imprecise Information in Databases. Proceedings of VLDB 90, pp. 696-707, 1990.

[GRAY93] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc., 1993.

[HARPER92] D. J. Harper and A. D. M. Walker. ECLAIR: an Extensible Class Library for Information Retrieval. Computer Journal, Vol. 35, 3, 256-267, June 1992.

[HAWTHORN79] P. Hawthorn and M. Stonebraker. Performance Analysis of a Relational Database Management System. Proceedings ACM SIGMOD Conference, pp. 1-11, 1979.

[HUFFMAN52] D. Huffman. A Method for the Construction of Minimum-redundancy Codes. IRE Proceedings, Vol. 40, No. 9, pp. 1098-1100, 1952.

[LOMET91] D. Lomet. Grow and Post Trees: Role, Techniques, and Future Potential. *Advances in Spatial Databases, Lecture Notes in Computer Science*, No. 525, pp. 183-206, Berlin, 1991. Springer-Verlag.

[LYNCH88] C.A. Lynch and M. Stonebraker. Extended User-defined Indexing with Applications to Textual Databases. Proceedings VLDB, pp. 306-317, 1988.

[MCLEOD83] I.A. McLeod and R.G. Crawford. Document Retrieval as a Database Application. Information Technology: Research and Development, Vol. 2, pp. 43-60, 1983.

[ORACLERdb] *Oracle Rdb Version 6.1 Development Documentation Kit.* Oracle Corporation, 1995.

[SQLMM] P. Cotton. ISO Working Draft SQL Multimedia Application Packages (SQL/MM) - Part 2: Full-text. ISO/IEC SC21/WG3 N1679, SQL/MM SOU-004, March, 1994.

[TOMASIC93] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. Technical Report STAN-CS_TN-93-1, Stanford University, November, 1993.

[ZIPF49] G.K. Zipf. *Human Behavior and the Principle of Least Effort*, Addison-Wesley Press, 1949.