Integrating Structured Data and Text: A relational approach

David A. Grossman
Office of Information Technology
Washington, DC 20505
Email: dgrossm1@osf1.gmu.edu

Ophir Frieder*
Department of Computer Science
George Mason University
Fairfax, VA
Email: ophir@cs.gmu.edu

David O. Holmes
AT&T Global Information Systems
Rockville, Maryland 20850

David C. Roberts
Office of Information Technology
Washington, DC 20505
Email: droberts@seas.gwu.edu

Abstract

We integrate structured data and text using the unchanged, standard relational model. We started with the premise that a relational system could be used to implement an Information Retrieval (IR) system. After implementing a prototype to verify that premise, we then began to investigate the performance of a parallel relational database system for this application.

We also tested the effect of query reduction on accuracy and found that queries can be reduced prior to their implementation without incurring a significant loss in precision/recall. This reduction also serves to improve run-time performance. After comparing our results to a special purpose IR system, we conclude that the relational model offers scalable performance and includes the ability to integrate structured data and text in a portable fashion.

1 Introduction

Increasingly, applications integrate structured and unstructured data, responding to requests such as "Find articles containing *vehicle* and *sales* published in journals with over 5,000,000 subscribers." Identifying articles containing specified terms requires a search of unstructured data, while circulation data are often stored as structured data in a relational database. We have developed a prototype to integrate structured data and text using combined resources from government, industry, and academia. Key aspects of the described prototype are currently deployed to real users as part of a larger system at the Internal Revenue Service.

*This work supported in part by the National Science Foundation under contract number IRI-9357785 and The Telephone Connection.

Our approach uses a relational database system for both structured data and text. This protects the large investment in relational systems while avoiding the need to purchase special purpose text search systems and integrate them with relational systems. Additionally, because versions of major relational database systems optimized for parallel machines are available, developers who employ the described approach can achieve the benefits of parallel processing without special purpose software. While parallel implementations of relational database systems are common, parallel implementations of information retrieval (IR) systems are rare. Implementing information retrieval as a relational database application provides a portable, parallel means of implementing information retrieval algorithms.

We show that the use of a relational system for IR incurs greater overhead than special purpose IR systems or object-oriented database systems. We have observed an average overhead ratio of 1.45:1 (index file to document collection) for a full text application; special purpose systems typically have a 0.47:1 ratio. Clearly, for some applications, the overhead of the relational model is excessive; however, we have found that for many applications, the described approach is cost-effective. Additionally, with the constantly increasing availability of lower cost storage and increasing computational resources, the cost of storage is becoming a much smaller portion of total system cost.

To evaluate this approach, we implemented a prototype on a sequential database system and a parallel database machine with four processors. We implemented tests using the TIPSTER document collection and the queries associated with the colleciton. These are the only standard set of queries that are currently available for benchmarking information retrieval systems. [3]. We are encouraged by our performance measurements; using a Pentium 90Mhz processor, most queries on a standard 280 megabyte text collection of Wall Street Journal articles obtained a result within five seconds. On the same machine, we compared the relational approach to an information retrieval system that also accesses structured data: *Lotus Notes*. For the majority of the test queries, the relational approach provided superior performance to the special purpose commercial system.

On a parallel machine, we typically observed that processor workloads were within ten percent of each other. Thus, we believe that the use of parallel processing provides a scalable method to

achieve run-time performance.

The idea of using the relational model to implement traditional IR functionality is not new. Essentially, prior efforts have suggested changes to SQL which make these approaches unavailable to users of present database systems and reduce the portability of applications written for these systems. Additionally, a change to SQL removes the developer's ability to leverage investments in legacy applications. Some previous work using unchanged SQL has been reported, but this work focused solely on bibliographic document collections [6]; that work is a direct predecessor to these efforts.

2 A Working Example

Since we are concerned with integration of structured data and text, we illustrate this discussion with two documents similar to those found in the TIPSTER document collection used to test our queries [8].

The effectiveness of relational systems with structured data is well established so we did not test performance with structured data. Thus, our example includes relatively little structured data. We show only that we can integrate *some* structured data; it follows that substantially more complicated structured data can be integrated using conventional techniques.

The relations used to represent a document are shown below. **DOC** contains structured information about a document. **DOC_TERM** models a traditional inverted index found in most commercial information retrieval systems. **DOC_TERM_PROX** is used for proximity searches. **IDF** refers to an automatically assigned weight, the inverse document frequency, assigned to each term in the document collection. The standard definition of IDF is used [16]. **QUERY** contains all terms in a sample query. Finally, **STOP_TERM** contains frequently occurring terms excluded from the **DOC_TERM** and **DOC_TERM_PROX** relation on the basis that they are "noise" terms.

Section 4 provides more details for each of these relations as well as descriptions of their use.

The following two documents are similar were taken from the TIPSTER collection. They were modified slightly to improve clarity:

<DOC> <DOCNO> WSJ870323-0180 </DOCNO>
<HL> Italy's Commercial Vehicle Sales </HL>
<DD> 03/23/87 </DD>
<DATELINE> TURIN, Italy </DATELINE>
<TEXT>
Commercial-vehicle sales in Italy rose 11.4% in February from a year earlier, to 8,848 units, according to provisional figures from the Italian Association of Auto Makers. Sales for the Association are expected to rise an additional 2% in July.
</TEXT>
</DOC>

<DOC> <DOCNO> WSJ870323-0181 </DOCNO>
<HL> Ford Discontinues Taurus SHO Five-Speed Vehicle </HL>
<DD> 01/21/95 </DD>
<DATELINE> George, Atlanta </DATELINE>
<TEXT>
Ford Motor Company announced that beginning in 1996, the Taurus SHO will no longer include a five-speed vehicle.
</TEXT>
</DOC>

DOC:

doc_id	doc_name	date	dateline
1	WSJ870323-0180	3/23/87	Turin, Italy
2	WSJ870323-0181	1/21/95	Georgia, Atlanta

DOC_TERM:

doc_id	term	tf
1	commercial	1
1	vehicle	1
1	sales	2
1	italy	1
1	rose	1
1	11.4%	1
1	february	1
1	year	1
1	earlier	1
1	8,848	1
1	according	1
1	provisional	1
1	figures	1
1	italian	1
1	association	2
1	auto	1
1	makers	1
1	expected	1
1	additional	1
1	2%	1
1	July	1
...	...	...
2	vehicle	1
...	...	...

DOC_TERM_PROX

doc_id	term	offset
1	commercial	1
1	vehicle	2
1	sales	3
1	italy	4
1	rose	5
1	11.4%	6
1	february	7
1	year	8
1	earlier	9
1	8,848	10
1	according	11
1	provisional	12
1	figures	13
1	italian	14
1	association	15
1	auto	16
1	makers	17
1	sales	18
1	association	19
1	expected	20
1	rise	21
1	additional	22
1	2%	23
1	July	24
...	...	...
2	vehicle	14
...	...	...

IDF

term	idf
11.4%	2.9595
2%	1.5911
8848	4.3936
according	0.7782
additional	1.0792
association	1.0792
auto	1.2788
commercial	1.0000
earlier	0.6021
expected	0.6990
february	1.3222
figures	1.2553
italian	1.8451
italy	1.6721
July	1.0414
makers	1.3010
provisional	2.5172
rose	0.7782
sales	0.6990
vehicle	1.7709
year	0.0000

QUERY

term	tf
vehicle	1
sales	1

STOP_TERM

term
a
an
and
...
the
...

Both structured data and text are found in our example. **DOC** contains structured data such as *date* and *dateline*; **DOC_TERM** and **DOC_TERM_PROX** contain unstructured data. A query with both structured and unstructured data, "List all documents that contain the terms *vehicle* or *sales* written on January 21, 1995," is implemented with the following SQL:

Ex: 1
```
SELECT d.doc_id
  FROM DOC d, DOC_TERM t
 WHERE t.term IN ("vehicle","sales") AND
       d.date = "1/21/95" AND
       d.doc_id = t.doc_id
```

3 Information Retrieval and Relational Database Systems: A Historical Progression

Prior work in this area can be described in terms of extensions to the Structured Query Language (SQL) and the proposed use of user-defined operators.

3.1 Extensions to SQL

The use of relational systems for text processing began with some of the original implementations of the relational model. An unpublished manuscript written in 1975 by Blair discusses the use of the SEQUEL research language (a precursor to SQL) [5]. In this work, SEQUEL was used to perform boolean keyword retrievals such as "Find all documents that contain the word *vehicle*." The first published work in this area was by Macleod and Crawford [10]. This paper presented SEQUEL queries to perform keyword searches and described SEQUEL extensions that could accomplish relevance ranking of a set of documents to a query.

Extensions to the relational model to assist an IR application were proposed in Macleod [11]. A RELEVANCE function assigning a measure of relevance between a document and a query was discussed. The RELEVANCE function used an attribute with an appropriate weight for each term in the query and the document. In weighted retrieval, the higher the weight assigned to a given term, the more significant the document is to the query. For a discussion of weighted boolean

retrieval, see Salton [14]. The *tf* attribute found in our example DOC_TERM and QUERY relations could serve as a simplistically assigned weight.

Other work that requires extensions to the relational model includes work done with NF^2 relational extensions [7, 17, 12]. In these systems, relations may be nested so that the data are in non-first normal form. Previous work has shown that embedded relations may be used to model inverted indexes [7, 17]. Embedded relations were intended to reduce the syntactic complexity of some SQL queries that required normalized relations. More recently, [12] illustrates that embedded relations assist with representation of hierarchical constructs found in text. Additionally, a new simplified query language is proposed. Although the use of embedded relations provides an interesting solution to the problem of integrating structured data and text, non-standard extensions to the standard relational model are required, and it is not clear how query optimization of these nested relational queries is implemented.

3.2 User-defined Operators

Interest in text applications of strictly commercial relational database systems diminished when user-defined operators were proposed for "application specific" operations [4]. Such operators could be used to provide any function required by an application that was not provided within the database system. Stonebraker, et. al., examined the utility of user-defined operators for a text editing application [18]. A thesis described the use of user-defined operators for typical information retrieval functionality such as keyword searches and proximity searches [9]. Additionally, database optimizer enhancements for query optimization of these user-defined operators were analyzed. The process for adding the proposed user-defined operators, however, required access to the entire address space of the DBMS [18]; thus such an operator could impact database integrity and security. Additionally, an operator implemented at one site might not be implemented at another, thereby reducing application portability.

4 Advanced IR Functionality using unchanged SQL

Today, attention is once again being given to using *standard* SQL for IR processing. Thus, for the remainder of this paper, we describe more advanced IR processing using strictly unchanged standard SQL.

Prior work has included extensions in which multiple joins are required to perform a Boolean search [10]. Each of the queries we have developed are of fixed syntactic length and do not require an increasing number of joins. Although it is theoretically possible to execute a join of n relations, many implementations impose limits on the number of relations in a join that fall much below those necessary to execute this query [2, 13].

We now describe SQL queries using standard SQL that are of fixed syntactic length that compute Boolean searches, Proximity searches, and Relevance ranking. Each of our queries depends upon the structures found in our working example. Essentially, each query is developed to perform well because of the construction of a relation that models an inverted index. The DOC_TERM relation models an inverted index without proximity information. The DOC_TERM_PROX relation models an inverted index that contains proximity data.

4.1 Boolean Retrieval

The following query computes a Boolean AND using standard syntatically fixed SQL:

Ex: 2
```
SELECT d.doc_id
  FROM DOC_TERM d,QUERY q
 WHERE d.term = q.term
 GROUP BY d.doc_id
 HAVING COUNT(d.term) = (SELECT COUNT(*) FROM QUERY)
```

The query given in Example 2 works by eliminating all terms from DOC_TERM that are not found in QUERY. The WHERE clause filters all terms in DOC_TERM that are not found in QUERY. This ensures that we consider only documents that have the terms found in QUERY. For a document d_i that contains k terms $(t_1, t_2, \ldots, t_k)$ in QUERY, the following tuples are found in the result set:

RESULT SET:

doc_id	*term*
d_i	t_1
d_i	t_2
...	...
d_i	t_k

This result set is actually constructed for every document. The GROUP BY found in the query partitions the result set into separate subsets that are of the form found in the example above. Each group contains only terms that are found in the query. However, we wish to return only documents that contain all of the terms found in the query. The HAVING clause eliminates all groups with a cardinality different from that of QUERY. This ensures that all of the terms found in the query are found in a group that corresponds to a particular document.

To this point, we have assumed that QUERY does not contain duplicate terms and that DOC_TERM does not contain duplicate terms for the same document. This is somewhat unrealistic especially when considering the need for proximity searches. However, at least up to now, we assumed that the text preprocessor that created the relations removed duplicates found in documents or queries. In the next section, we present a proximity search method for multiple occurrences of the same term in a document. To account for the possibility of duplicates, it is necessary to modify the HAVING clause found in the Example 2 to: HAVING COUNT(DISTINCT DOC_TERM.term). The DISTINCT causes duplicate terms in a document to be removed. The new query is:

Ex: 3
```
SELECT d.doc_id
  FROM DOC_TERM d, QUERY q
 WHERE d.term = q.term
 GROUP BY d.doc_id
 HAVING COUNT(DISTINCT(d.term)) = (SELECT COUNT(*) FROM QUERY)
```

We note that a Boolean OR can be constructed by removing the HAVING clause in Example 2. Finally the SQL to compute a TAND (threshold AND which retrieves documents that contain at least k specified terms) condition is obtained by modifying the HAVING clause:

Ex: 4
```
SELECT d.doc_id
  FROM DOC_TERM d,QUERY q
 WHERE d.term = q.term
 GROUP BY d.doc_id
 HAVING COUNT(DISTINCT(d.term)) ≥ k
```

4.2 Proximity Searches

Several IR systems provide proximity searching as a basic capability [15]. A slight modification to DOC_TERM facilitates such searches. A proximity is a request for all documents that contain n terms within a *term window* of size *width*. A *term window* of size *width* begins at the *ith* term in the document and continues to the $(i + width - 1)$ term.

To implement proximity searches, DOC_TERM_PROX is used. This relation is similar to DOC_TERM with the addition of a new attribute, *offset*. DOC_TERM_PROX has a tuple for each term in the original text, excluding doc_terms filtered out by the preprocessor.

We again require the user to place search terms in QUERY. Once the relations DOC_TERM_PROX and QUERY are established, the query in Example 5 can be used to find the documents that contain a *term_window* of size *width* that includes all of the terms found in QUERY. We choose this definition as simpler forms of proximity searches (show all documents that contain *term1* and *term2* within two words of one another) can be viewed as a subset of this query. The *term_window* may be thought of as a sliding window starting at the first term in the document. The query insists that not only must all terms found in QUERY be contained in a given document, but at least one occurrence of each term in QUERY must fall within a term window of size *width*.

For document number one in our example, the terms "vehicle" and "italy" occur in positions two and four, respectively. Hence, a query requiring these terms within a window of size greater than or equal to three would retrieve this document. Specifying a *width* greater than the maximum document length eliminates all proximity constraints and results in a query equivalent to the query found in Example 2. The following query uses unchanged SQL and only a single join to perform the proximity search:

Ex: 5
```
SELECT a.doc_id
  FROM DOC_TERM_PROX a, DOC_TERM_PROX b
 WHERE a.term IN (SELECT q.term FROM QUERY q) AND
    b.term IN (SELECT q.term FROM QUERY q) AND
    a.doc_id = b.doc_id AND
    (b.offset - a.offset) BETWEEN 0 AND (width − 1)
 GROUP BY a.doc_id, a.term, a.offset
 HAVING COUNT(DISTINCT(b.term)) = (SELECT COUNT(*) FROM QUERY)
```

The query of Example 5 eliminates all terms from DOC_TERM_PROX that are not found in QUERY. This is done via the first two conditions in the WHERE clause. It is necessary to join DOC_TERM_PROX on itself since we must evaluate the distance between a given term in a document and all other terms. The third WHERE clause condition ensures that we do not compare the distance between terms found in distinct documents. For a document d_i that contains k terms $(t_1, t_2, \ldots, t_k)$ in corresponding term positions $(o_1, o_2, \ldots, o_k)$, the following tuples make the first three conditions in the WHERE clause evaluate TRUE:

a.doc_id	*a.term*	*a.offset*	*b.doc_id*	*b.term*	*b.offset*
d_i	t_1	o_1	d_i	t_1	o_1
d_i	t_1	o_1	d_i	t_2	o_2
d_i	t_1	o_1	d_i	t_k	o_k
d_i	t_2	o_2	d_i	t_1	o_1
d_i	t_2	o_2	d_i	t_2	o_2
d_i	t_2	o_2	d_i	t_k	o_k
d_i	t_k	o_k	d_i	t_1	o_1
d_i	t_k	o_k	d_i	t_2	o_2
d_i	t_k	o_k	d_i	t_k	o_k

The fourth condition in the WHERE clause removes all tuples not within the size of *width*. This condition enforces the proximity constraint. The rest of the query is similar to Example 2. GROUP BY partitions the result into sets corresponding to each *doc id, term* and *offset*. HAVING eliminates groups with cardinality not equivalent to the cardinality of QUERY. It is necessary to use DISTINCT in the HAVING clause to ensure that a term window does not contain repeated occurrences of any terms.

Consider a request to find all documents containing the terms "vehicle" and "sales" within a term window of size 4. First a QUERY relation is constructed, with a tuple for each of these two terms. Using DOC_TERM_PROX as above, we now consider execution of the query given

in Example 5. The first three conditions of the WHERE clause eliminate all terms not found in QUERY. These three conditions are TRUE for these tuples:

a.doc_id	*a.term*	*a.offset*	*b.doc_id*	*b.term*	*b.offset*
1	vehicle	2	1	vehicle	2
1	vehicle	2	1	sales	3
1	vehicle	2	1	sales	18
1	sales	3	1	vehicle	2
1	sales	3	1	sales	3
1	sales	3	1	sales	18
1	sales	18	1	vehicle	2
1	sales	18	1	sales	3
1	sales	18	1	sales	18
2	vehicle	14	2	vehicle	14

The fourth condition uses the difference between *a.offset* and *b.offset* and eliminates all tuples that result in either a negative difference or fall outside of the term window. The following tuples make the entire WHERE clause evaluate TRUE:

a.doc_id	*a.term*	*a.offset*	*b.doc_id*	*b.term*	*b.offset*
1	vehicle	2	1	vehicle	2
1	vehicle	2	1	sales	3
1	sales	3	1	sales	3
1	sales	18	1	sales	18
2	vehicle	14	2	vehicle	14

GROUP BY partitions the result set based on *document id, term*, and *offset*. Double lines illustrate this partitioning which fixes a starting point of a term window and then places all terms within the window in a particular group.

a.doc_id	*a.term*	*a.offset*	*b.doc_id*	*b.term*	*b.offset*
1	vehicle	2	1	vehicle	2
1	vehicle	2	1	sales	3
1	sales	3	1	sales	3
1	sales	18	1	sales	18
2	vehicle	14	2	vehicle	14

The HAVING removes all groups that do not contain a term window with a cardinality equal to that of QUERY (two for the example). In the example, only the group that starts at offset two in document one contains two tuples. DISTINCT ensures that duplicate terms within the term window are not counted. Hence, only document number one is returned by the query.

It should be noted that the execution of the WHERE clause and the GROUP BY may be intertwined, so the actual sequence of operations may not correspond to the steps described above. To explain the logic of the query, we have presented the query as if the entire WHERE clause were first evaluated, followed by GROUP BY and HAVING.

Syntactically, the query in Example 5 is complex, but it is important to note that it is invariant regardless of the number of query terms. Additionally, only a single join of DOC_TERM_PROX is required regardless of the size of QUERY.

4.3 Computing Relevance Using Unchanged SQL

Boolean and proximity searches are often used in commercial IR systems. Since the vector-space model [16] is widely used, we have chosen to implement unchanged SQL to rank documents based on the vector-space model.

Boolean searches simply retrieve all documents that match a particular condition. A relevance ranking algorithm ranks all documents that are retrieved so that they may be examined by the user in the order of their computed "relevance" to the query. For a sufficiently large answer set, it is unlikely that the user will be able to examine all of the documents returned. The vector space model is frequently used to compute a measure of relevance between a query and a document.

The vector space model works by representing each document and a query with a term occurrence vector. The cartesian distance between the query and each document vector is used to rank the documents. The idea is that those documents "closest" to the query will be the most relevant. The vector for each document is of size n. It contains an entry for each term in the entire document collection. Each component of the vector contains a weight computed for each term in the document collection. For each document, each term is assigned a weight based on how frequently it occurs in the entire collection and how often it appears in the document. The weight of a term in a document increases the more often it appears in a document, and the less often it appears in all other documents.

The key addition to the database schema required by relevance ranking is the IDF relation. This relation stores the inverse document frequency (idf) for each distinct term in the document

collection. This is computed as:

$$idf_t = \log \frac{N}{df_t}$$

where N is the number of documents in the collection and df_t is the number of documents that contain term t. More details on this definition may be found in [16]. The following query provides a vector space ranking of all documents for the query composed of the terms found in the QUERY relation. The query given below is a simple dot product computation. Other variations on the dot product including Dice, Cosine, and Jaccard can be computed with variations on this query.

Our example assumes the QUERY relation has been extended to include two attributes, *term* and *tf*, that are used to store the terms found in the query and the number of occurrences for each term.

Ex: 6
```
SELECT d.doc_id, SUM(q.tf * i.idf * d.tf * i.idf)
  FROM QUERY q, DOC_TERM d, IDF i
 WHERE q.term = i.term AND
       d.term = i.term
 GROUP BY d.doc_id
 ORDER BY 2 DESC
```

Once again, the query in Example 6 is of fixed length and need only be developed once. Practical experimentation shows that this query performs well on a variety of relational systems. Results presented below provide further details on performance of this query.

5 Sequential Results

To study the performance of our approach, we measured run time and accuracy using a relational database system: *Microsoft SQL Server V4.2* running on a 90Mhz Dell Pentium with 64 megabytes of RAM using Windows NT V3.1. We evaluated queries 150-200 over the Wall Street Journal portion (1987, 1988, and 1989) of the collection comprising over 280 megabytes of text. These queries were required for the TREC-3 conference. Our results were obtained using default system parameters and were meant to provide an initial overview of actual performance. Additional tuning could improve response time.

For all of our relational results we used a text preprocessor that accepts the SGML marked TIPSTER documents and outputs flat files in the form of our relations. The preprocessor removes stop terms, numbers, and special characters. Additionally, all upper case letters are translated to lower case. Subsequently, a DBMS vendor-supplied load utility was used to move the data from the flat files into the DBMS.

For comparison, we also implemented the queries using *Lotus Notes*. We chose Notes as our product for comparison because it offers integration of both structured data and text, and it uses the search engine found in *Topic* (from Verity Inc.) to improve search performance.

Performance of our approach is dramatically affected by the selectivity of query terms. Frequently occurring terms result in degraded performance due to the I/O required to obtain all occurrences. To measure the impact of term selectivity on the query, we use a measure of term selectivity to vary the original query. First, the terms in a query are sorted by their frequency across the entire document collection. Subsequently, query variations are developed by including only the x least frequent terms in the original query where x varies based on the size of the query. We refer to x as a *query threshold.* A threshold of ten percent when applied to an original query of one hundred terms results in the ten least frequent terms being applied to a reduced query.

Execution time is not the only performance measurement; the number of relevant documents retrieved is also important. A system that misses relevant documents is useless regardless of execution speed.

Our hypothesis is that query reduction based on varying thresholds improves run time performance without affecting accuracy. Our premise is that queries contain many terms which occur so frequently that they serve only to degrade run time performance and possibly degrade accuracy as well. Reducing the query to remove such terms should improve both run time and accuracy.

Two variations of DOC_TERM for SQL Server were implemented. We used full length terms as described in the preceding sections as well as numerical term identifiers. Term identifiers use numerical values to represent character terms. On average, the term length for our collection was six characters, so the use of a four byte term identifier instead of full-length terms reduces the length of each tuple by two bytes. It was our hypothesis that the use of term identifiers would

improve run time performance. However, maintenance of term identifiers is complicated since it requires the generation of a term dictionary that must be updated when new documents are added to the collection.

We tested two different inverted indices constructed by Lotus Notes. The product allows users to optionally choose stemming (the removal of frequently used prefixes and suffixes from a term) as a means of building an index. Intuitively, stemming should improve accuracy since queries searching for the term *running* will match documents that contain *run*. However, some stemming reduces accuracy since terms such as *sting* can be stemmed to meaningless syllables that may falsely match with non-related terms.

Run time results for SQL Server and Lotus Notes are presented in Figure 1. We tested query thresholds of 10%, 20%, 25%, 33%, 50%, and 100% for all of our experiments. A known bug in Lotus Notes made it impossible to implement queries that produce a large result set, so we were unable to obtain results for a threshold of 100. It can be seen that as the query threshold increases, performance for all systems degrades. Note, however, that once a threshold of 33% is reached, the rate of increase of the number of relevant documents retrieved significantly decreases (see Figure 2). Given the dramatic increase in run time (shown in Figure 1) when using a threshold greater than 33%, depending on the application, the merits of using high (greater than 33%) thresholds may be questionable.

For lower thresholds, few terms are found in the query and performance of SQL Server is superior to Lotus Notes. After a threshold of 50%, the SQL Server performance degrades dramatically. We suspect based on the fact that Lotus Notes uses a more efficient inverted index that Lotus Notes would provide better performance at thresholds higher than 50.

Overhead costs of modeling an inverted index as a relation increase dramatically as the term selectivity increases. A term that occurs many times in an inverted index requires more overhead than an infrequent term, but with the relational implementation the cost skyrockets. This is because the DOC_TERM relation is so large and a term that occurs many times in this relation requires large amounts of I/O to obtain that term. Hence, performance for the relational model is good only for query thresholds lower than 50%.

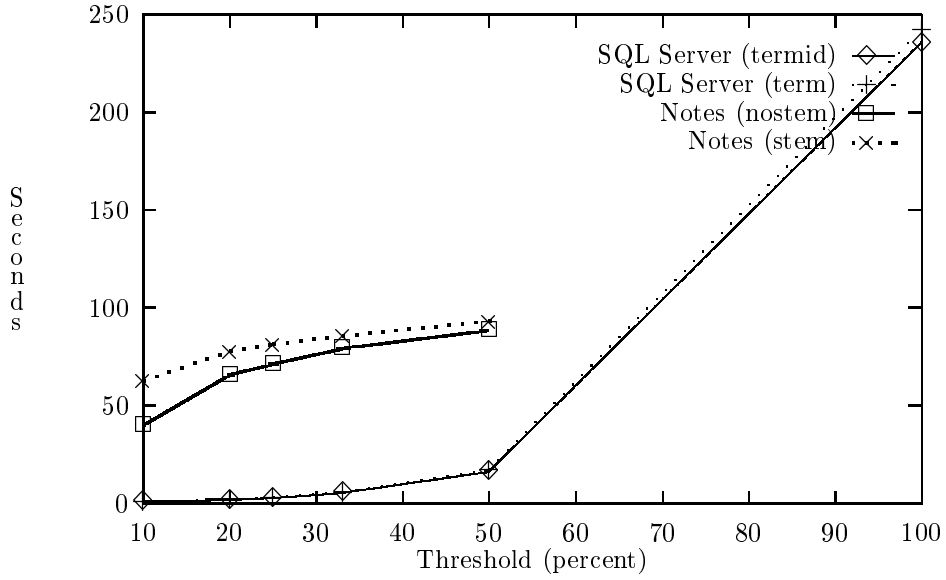

Figure 1: Average Response Time for Varying Query Thresholds

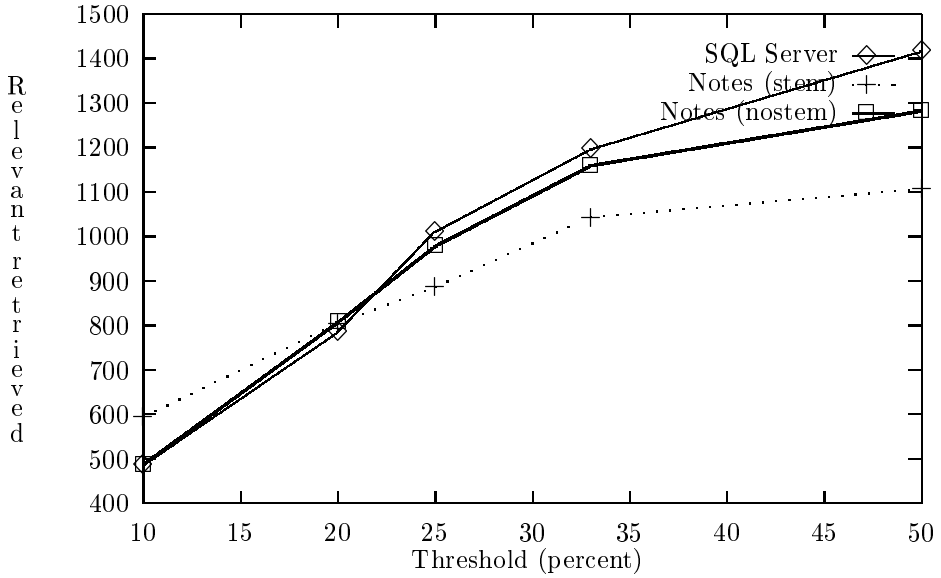

Figure 2: Number of Relevant Documents Retrieved

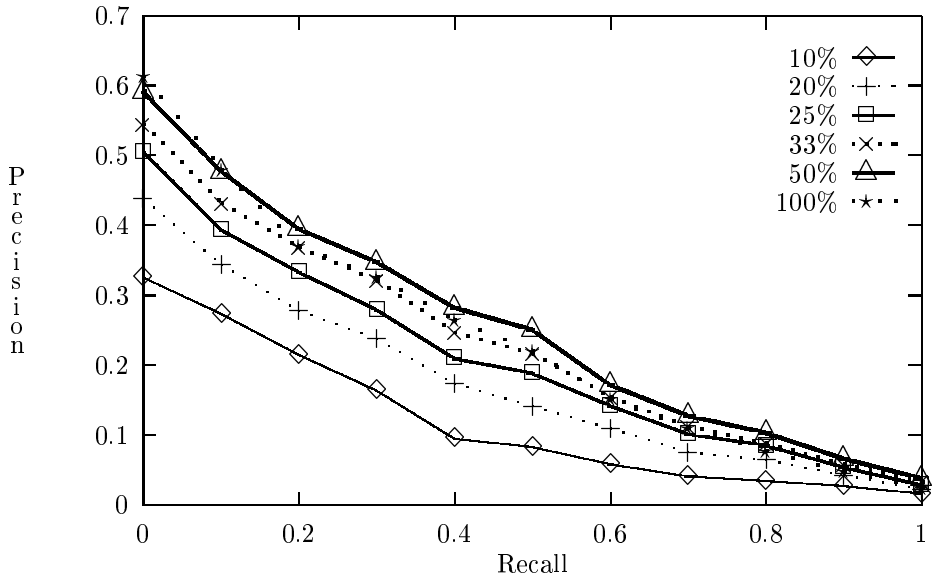

Figure 3: Precision/Recall for SQL Server for Varying Query Thresholds

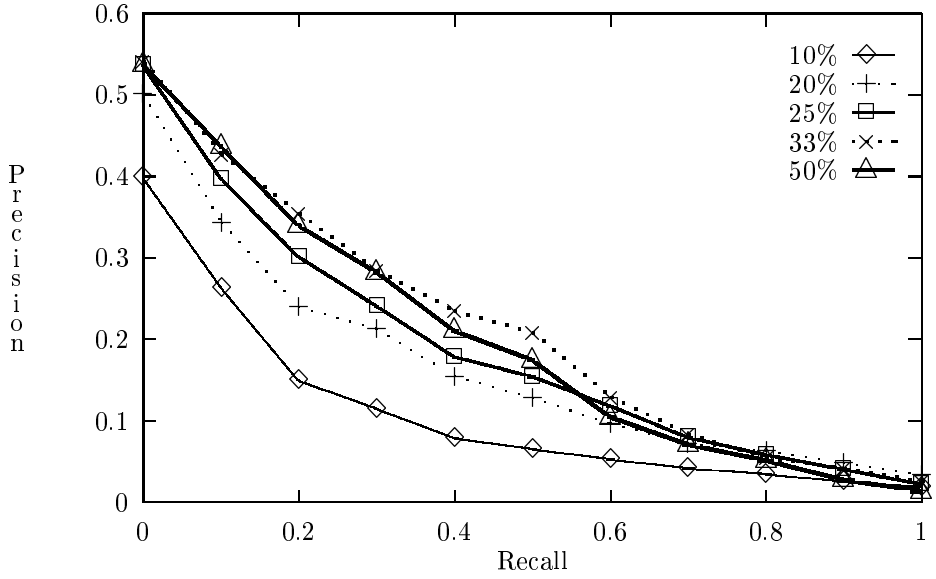

Figure 4: Precision/Recall for Lotus Notes for Varying Query Thresholds

Our evaluation of retrieval accuracy is summarized in Figures 2, 3, and 4. Figure 2 indicates the number of relevant documents obtained for all fifty queries. For low query thresholds, the number of relevant documents is low for all systems. As we increase the query threshold, the number of documents retrieved increases dramatically. This reflects the additional accuracy obtained by adding relatively infrequently occurring terms to the query. We believe that these terms assist in finding relevant documents as they are relatively infrequent and therefore are probably not noise words. As the threshold increases, it is reasonable to suspect that progressively less useful terms are added to the query and, interestingly, the number of relevant documents does not increase. Hence, despite dramatically increased I/O required to compute higher thresholds, they do not yield significantly more relevant documents.

In figures 3 and 4, we present results using standard information retrieval measures, precision and recall, to measure accuracy. *Precision* refers to the ratio of retrieved documents that are relevant, while *recall* refers to the ratio of relevant documents that are retrieved [14]. Since term identifiers result in the same accuracy as terms (the same documents are retrieved), accuracy results for only one implementation of SQL Server are provided in Figure 3. Figure 4 provides precision and recall values for the Notes experiment without stemming, as this was the superior result for Lotus Notes. Each line in these figures indicates a separate threshold. Again, for low thresholds, precision and recall are relatively low, as the infrequent terms are added, precision and recall increase, and finally, at high thresholds noise terms are added and precision and recall does not continue to increase.

We have noted that the relational approach yields good performance at thresholds below 33%. Now it can be seen that accuracy does not improve when thresholds are increased above 33%. Hence, the relational approach may be viable as our experiments have shown that it may not be necessary to implement thresholds above 33%.

6 Parallel Results

To study the applicability of parallel processing to our approach we measured run time using a four processor AT&T DBC-1012. The DBC-1012 is a commercial database machine that implements a

relational database system using multiple processors and I/O units. By spreading computational requirements over the number of processors, the goal is to provide a scalable approach to large relational database problems. Our hypothesis is that the SQL used to implement relevance ranking would result in a balanced workload across the processors.

Parallel Information Retrieval is not a mature technology. Some initial efforts are discussed in a special issue of *Information Processing and Management* [1]. The use of the relational model for relevance ranking makes it possible to use mature technology. As with the sequential approach, runtime performance is dramatically affected by the selectivity of the terms in the query. Running all fifty queries at a threshold of 50% required about an hour while running all fifty queries at a threshold of 100% required a day.

We implemented the same fifty queries used to obtain the sequential results on roughly two gigabytes of the TIPSTER collection. At present, this is the largest portion of the collection that contains known relevance results. After collecting results for each query, the sum of the CPU time and disk I/O for each of the four processors was computed. The highest sum is the largest factor in response time as the query cannot complete until all processing is completed. Figure 5 contains the maximum CPU obtained for all fifty queries.

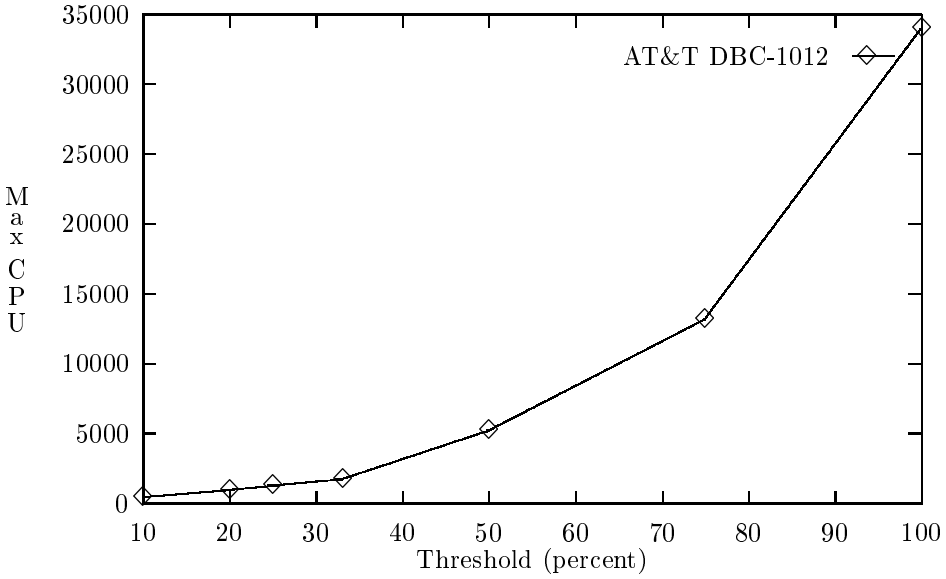

Figure 5: Max CPU Time for Varying Query Thresholds

Accuracy results for the two gigabyte collection are summarized in Figures 6 and 7. Figure 6 indicates the number of relevant documents obtained for all fifty queries at retrieval cutoffs of 1000 documents. As we increase the threshold from 10 to 20 a large number of relevant documents are found. This continues as we increase the threshold to 33. Once we move to 50, we again observe a dropoff in the number of relevant documents to a level even below a threshold of 20. Figure 7 gives precision and recall for a cutoff of 1000.

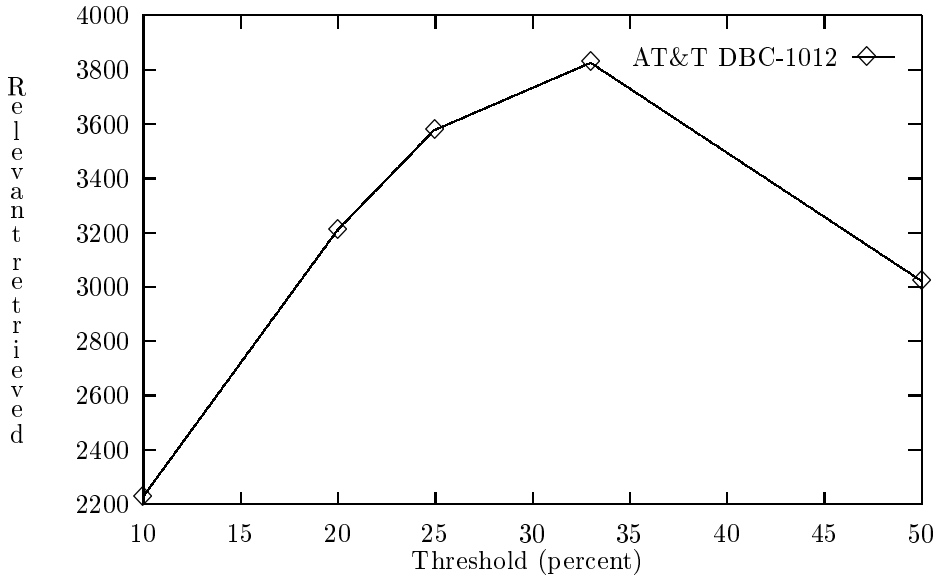

Figure 6: Number of Relevant Documents Retrieved (2GB)

Our hypothesis for the parallel approach was that the queries would run in a balanced fashion; that is, the workload for each processor would be approximately equal. This balance is critical if the approach is to be truly scalable. The table below indicates the amount of Processor Load Imbalance (PLIB) for CPU time and DISK I/O: $\left(\frac{max-min}{min}\right)$ measured at each query threshold. It can be seen that for all workloads, the processors are fifteen percent or less out of balance. Given that the workload is balanced evenly among the existing processors, if processors are added, response time will be reduced. Due to resource limitations, we were unable to empirically validate this scalability hypothesis.

Percent of Processor Imbalance :

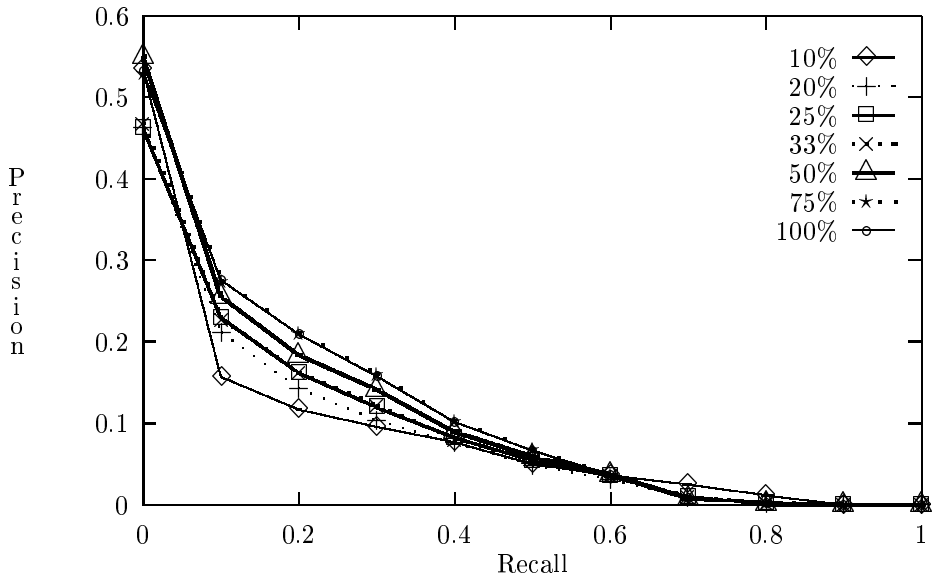

Figure 7: Precision/Recall for AT&T DBC-1012. (2 GB)

threshold	*CPU Time*	*Disk I/O)*
10	15.0	1.5
20	12.0	1.5
25	10.4	1.2
33	9.4	0.9
50	2.0	0.7

Finally, we note that data loading was improved when using the DBC-1012 because the DBC-1012 FASTLOAD utility makes use of all the processors and ensures that data are distributed to each processor in the fashion prescribed by indexes. We typically found that the FASTLOAD was able to load our largest relation at a speed of 857 rows per second. By comparison, in our experience, an Intel Pentium processor running Microsoft SQL Server on Windows NT typically loads data with the Bulk Copy facility at a rate of 381 rows per second.

7 Overhead

Typically, storage overhead has been viewed as the primary disadvantage of relational IR implementations. Given that the document identifier and the term must be replicated numerous times in DOC_TERM, it would appear that storage requirements would be substantial.

The observed storage overhead required by the SQL Server and the Lotus Notes implementations for the 280 megabyte data is given below:

Storage Overhead (Lotus Notes vs SQL Server)

System	Megabytes	Overhead Ratio
Lotus Notes (stemming)	133	0.48 : 1
Lotus Notes (no stemming)	130	0.46 : 1
SQL Server (termid)	246	0.88 : 1
SQL Server (terms)	377	1.35 : 1

The inverted index used by Lotus Notes is clearly the most efficient in terms of storage. However, the additional overhead for SQL Server may still be acceptable given the additional functionality provided by this approach. The use of term identifiers reduces storage overhead by 35%; however, surprisingly, response time does not dramatically improve. Therefore, the computational overhead of updating a separate term dictionary in order to assign term identifiers appears to be too large to justify the associated savings in storage.

The following table indicates the storage requirements for each of the relations used on the parallel machine. To simplify the data loading operation, and to allow us to easily run experiments on different portions of the collection, we constructed DOC_TERM relations for each of the nine different portions of the TIPSTER collection. To obtain final results, we implemented the same SQL given in Section 4.3 with a UNION to merge all of the results.

Since the DBC-1012 uses hash-based indices, there is no extra storage required for each index; rather, a fixed thirteen byte overhead is assigned for each tuple to maintain an internal hash identifier. The observed storage overhead for each of the nine sections is given below:

Storage Overhead (AT&T DBC-1012)

Section	DBMS (MB)	Original (MB)	Avg. Terms / Doc.	Overhead Ratio
AP (disk 1)	466	266	375	1.75 : 1
AP (disk 2)	441	248	370	1.77 : 1
DOE (disk 1)	400	190	89	2.10 : 1
FR (disk 1)	223	258	1017	0.86 : 1
FR (disk 2)	180	211	1073	0.85 : 1
WSJ (disk 1)	475	295	329	1.61 : 1
WSJ (disk 2)	334	255	377	1.31 : 1
ZIFF (disk 1)	347	251	412	1.38 : 1
ZIFF (disk 2)	260	188	394	1.38 : 1
Total	3126	2162	493	1.45 : 1

For the 2.1 gigabytes of text, the relational structures to implement it required 3.1 Gigabytes of storage for an overall storage ratio of 1.45:1. The key to the amount of relational storage is the number of terms in a document. As the number of terms increases, the likelihood for repetition increases. When a term is repeated in a document, the *cnt* attribute is updated, but a new tuple is not added to DOC_TERM. Hence, the portion of the collection with large documents (over 1000 terms per document), the Federal Register, had the lowest storage overhead ratios. The portion of the collection with the smallest documents was the Department of Energy Abstracts, and it had a storage overhead ratio of 2.1:1.

Although the storage overhead for a relational system is higher than for an inverted index, the trend in industry is towards dramatic reductions in the cost of disk space. Given the potential for reduced software development time, the ability to implement new functionality with standard SQL, and the peformance potential given by a parallel machine, this overhead may be acceptable for many applications.

8 Conclusions and Future Work

We described unchanged SQL that is capable of performing a variety of Boolean keyword searches, proximity searches, and relevance ranking. Each of the queries requires only a fixed number of joins regardless of the number of terms in the query. This makes it possible to consider the use of the relational model as a tool for IR even for queries with many terms.

We presented implementation details of an experimental prototype and compared it with a popular commercial product. We found that run time results were comparable for query thresh-

olds between 25 and 33 percent and that our approach provides additional functionality in that unchanged SQL may be used to integrate structured data and text.

The relational approach lends itself to parallel processing as relational DBMS have been implemented on many parallel machines. Our results indicate that the approach is scalable, and we believe it is feasible to implement our approach for large document collections (in the Terabyte range) using a parallel DBMS approach. We are currently working to test our approach for larger document collections.

We have measured the secondary storage overhead required to implement the relational approach and have found that it is less than 1.5:1, or roughly a factor of three as compared to traditional IR systems. Such an overhead may be acceptable given the functional advantages of our approach. A simplistic term identifier approach has yielded a 35% storage improvement. It is reasonable to expect that more sophisticated storage reduction techniques could be used to further reduce storage requirements.

Additionally, we have only shown term based Information Retrieval algorithms in SQL. Other approaches such as n-grams, thesauri, and passage-based retrieval can be implemented in a straightforward manner; however, in the latest results from the Text Retrieval and Evaluation Conference, it is not clear that these approaches improve precision/recall. In terms of run-time performance, additional processing results in substantially higher run-time. Therefore, we did not degrade run-time performance to obtain unsubstantiated improvements to the accuracy of our prototype. However, in the future, we will explore the relational implementation of approaches that have been shown to yield improvements to precision/recall.

References

[1] *Special Issue on Parallel Processing, Information Processing and Management, Volume 27, Number 4*, 1991.

[2] Sql/ds systems programming manual. 1992.

[3] *Proceedings of the Fourth Text REtrieval and Evaluation Conference*, 1995.

[4] M. Stonebraker Jeff Anton and Eric Hanson. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3):350–376, September 1987.

[5] D. Blair. Square (specifying queries as relational expressions) as a document retrieval language. Unpublished working paper, University of California, Berkeley., 1974.

[6] D. Blair. An extended relational retrieval model. *Information Processing and Management*, 1988.

[7] P. Goyal, B.C. Desai, and F. Sadri. Non-first normal form universal relations: An application to information retrieval systems. *Information Systems*, 12(1):49–55, 1987.

[8] Donna Harman. Overview of the third text retrieval conference. *Proceedings of the Third Text REtrieval Conference*, 1995.

[9] C. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. *Proceedings of the 14th VLDB Conference*, pages 306–317, 1988.

[10] I. Macleod. A relational approach to modular information retrieval systems design. *Proceedings of the ASIS Annual Meeting*, 15:83–85, 1978.

[11] I. Macleod. Sequel as a language for document retrieval. *Journal of the American Society for Information Science*, pages 243–249, September 1979.

[12] T. Niemi and K. Jarvelin. A straightforward nf2 relational interface with applications in information retrieval. *Information Processing and Management*, 31(2):215–231, 1995.

[13] *Informix-OnLine Dynamic Server, Performance Guide*, May 1995.

[14] G. Salton. Parallel text search methods. *Communications of the ACM*, pages 202–214, February 1988.

[15] G. Salton. *Automatic Text Processing.* Addison-Wesley, 1989.

[16] G. Salton, C.S. Yang, and A. Wong. A vector-space model for information retrieval. *Communications of the ACM*, 18, 1975.

[17] H.J. Schek and P.Pistor. Data structures for an integrated data base management and information retrieval system. In *Proceedings of the Eighth International Conference on Very Large Data Bases*, pages 197–207, September 1982.

[18] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and Antonin Guttman. Document processing in a relational database system. *ACM Transactions on Office Information Systems*, 1(2):143–158, April 1983.