Integrating INQUERY with an RDBMS to Support Text Retrieval

Vasanthakumar S. R.,*James P. Callan, and W. Bruce Croft

Department of Computer Science
University of Massachusetts, Amherst, MA 01003, USA
vasant@cs.umass.edu

Abstract

Information is a combination of structured data and unstructured data. Traditionally, relational database management systems (RDBMS) have been designed to handle structured data. IR systems can handle text (unstructured data) very well but are not designed to handle structured data. With present day information being a combination of structured and unstructured data, there is an increasing demand for an IR-DBMS system that incorporates features of both IR and DBMSs. We discuss a framework that incorporates powerful text retrieval in relational database management systems. An extended SQL with probabilistic operators for text retrieval is defined. This paper also discusses an implementation of the probabilistic operators in SQL.

1 Introduction

The state of the art is that much information, especially multi-media, is represented as a combination of both structured and unstructured data. Structured data comprises data types like integer, real, fixed-length string; unstructured data comprises text, images, audio *etc*. Structured data has been efficiently stored and retrieved using relational database management systems (RDBMS). Text, an unstructured component of information, has been traditionally stored and retrieved using Information Retrieval (IR) systems. RDBMSs use exact matching to retrieve data. while IR systems use approximate matching. IR systems are not suitable for structured data and RDBMSs are not suitable for unstructured data. RDBMSs have the additional advantage of addressing the issues of concurrency, recovery, security and integrity, while most IR systems don't. The gap between structured and unstructured components in data has been recently narrowed (*e.g.,* medical information systems, pharmaceutical systems) and has demanded a system that incorporates the features of both RDB and IR systems.

Our goal is to add powerful *text* retrieval capabilities to an RDBMS using the relational framework and SQL. Regular boolean operators are used on the non-text attributes and probabilistic operators

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This research was supported in part by Center for Intelligent Information Retrieval and Digital Equiment Corporation.

are used on the text attributes. The probabilistic answers are converted into boolean values and later combined with the results of the non-text component of the query. The final result set is ranked on the probability (belief) that a record is relevant to the text query.

We have an existing information retrieval system INQUERY [2]. We are experimenting on implementing the same retrieval strategy in a relational database management system, DEC Rdb. Such an implementation will add powerful text retrieval capabilities to an RDBMS, facilitating the construction of IR systems that have all the features we get from an RDBMS (concurrency, recovery, *etc.*) [1]. This paper discusses the issues, our experiences and status.

2 Integrating IR and RDBM Systems

Integrating IR and RDBMS could be viewed at different levels:

- a loosely-coupled IR/RDBMS system, and
- a tightly-coupled IR/RDBMS system.

2.1 Loosely-coupled IR/RDBM system

A loosely-coupled IR/RDBMS system can be viewed in different ways:

- IR system as an application of RDBMS,
- A Hybrid of IR and RDBM systems, and
- Using RDBMS for storing IR data structures.

2.1.1 IR System as an Application of RDBMS

We could build an IR system as a RDB application without any major modification to the existing RDBMS [6]. These applications are based on "exact matching", and query evaluation is "boolean" in nature. Probabilistic evaluation of queries is very effective for *text retrieval*. Blair [1] uses the concept of probability for ranking the records. The inability of such systems to handle fuzzy queries results in an IR system with poor retrieval performance (low *precision* and *recall*). Also, IR data structures tend to vary in size greatly, and thus the application would be inefficient.

2.1.2 A Hybrid Approach

A hybrid IR/DB system utilizes both an IR and DB system. An *embedded full integration* is proposed by Gu *et al.* [5]. This approach proposes the use of two distinct systems, an IR system (INQUERY) and RDBMS (Sybase). The inverted lists for the *text* fields in the RDBMS tables are stored in INQUERY. An extended SQL (ESQL) is proposed which has both boolean and IR operations. A form-based IR interface is provided for the end users and the user's intention is interpreted into a program described by a query language called ESQL which is an extension of SQL. The ESQL program is then translated to a standard SQL program and an INQUERY query by a *parser and interpreter*. The INQUERY query is sent to *ProcINQUERY* - an INQUERY version which can be invoked as a procedure, and output the information about ranked textual data into Sybase. The SQL query is then sent to Sybase which searches the corresponding data based on the outputs of the ProcINQUERY. The disadvantages of such an approach are that we use two different systems, and we lack flexibility in combining IR and boolean parts of the query. This motivates us to develop an RDBMS system which does not make use of any IR system, but instead, stores all the IR data structures in the RDBMS and implements all the IR operators in SQL itself. Section 3 explains our approach to achieve the above mentioned goal.

2.1.3 Using RDBMS for storing IR data structures

Information retrieval systems index unstructured text into an *inverted index* or *inverted file* [8]. For each term a separate index is constructed that stores the record identifiers, or document identifiers, for *all* the records containing that term. With an inverted index, the record set corresponding to a given query formulation is easily determined. The identifiers for all retrieved items can be obtained by extracting from the inverted index the list of record identifiers corresponding to each query term and combining these record identifiers appropriately. For supporting probabilistic retrieval the term statistics are also stored along with record identifiers. In order to support complex query operators like *phrase* or *proximity*, the locations of each occurrence of the term in a record are also stored (*proximity* information). The number of tuples of an inverted file is huge when compared to the number of documents or records they represent. Thus the number of tuples in an inverted file will be the number of unique terms in the documents times the average number of terms in a document. As stated by Blair [1], any discussion of database management system implementation must address the controversial issue of processing speed. Traditional beliefs tend to hold that relational systems trade flexibility of query and database structuring for reduced processing speeds. In order to overcome this bottleneck, we sacrificed some flexibility and reduced the number of tuples to the number of unique terms in the collection or documents database. The term statistics and the proximity information are stored in a *binary object* or *blob*. We used the INQUERY information retrieval system and DEC Rdb RDBMS for this experiment. The following paragraphs discuss the lessons learned from this implementation.

Rdb was used to store all of INQUERY's file-based data structures (inverted file, *db* file, and term dictionary). The inverted list file contained term ids and their inverted lists. The *db* file contained document indices necessary for providing user interface functions in the API. In this implementation, Rdb did not know the internal structure of the inverted list and the *db* file. The encoded inverted list and *db* file information was stored in Rdb as blobs. An SQL-based interface was used between INQUERY and Rdb. There was an overhead in this implementation for INQUERY to decode the blob and extract the required information. Other than the query language, all the other features like concurrency control, recovery, *etc.* , (refer Section 1) of an RDBMS were exploited without sacrificing performance during document indexing and query evaluation.

2.1.4 Results of *Blob* Implementation

The results from the *blob* implementation are compared with the *keyfile* implementation of INQUERY in Table 2. *Keyfile* is a B-tree package which is used in INQUERY to store all its data structures. The experiments were done on a DEC Alpha running OpenVMS with 80 Mbytes of main memory. DEC Rdb V 6.0 was used. Several test databases with different characteristics were used to analyze the performance of the keyfile and Rdb versions of INQUERY. Table 1 shows the characteristics of the test databases used. Table 2 shows that the elapsed time of the Rdb version is in the same order of magnitude as that of the keyfile version and we get all the advantages of using an RDBMS.

2.2 Tightly-coupled IR/RDBM system

Different approaches have been proposed for a tighter integration of IR and RDBM systems [4, 7, 1, 9]. All these methods suggest implementing a new system incorporating the proposed theories. Schek *et al.* [9] propose an extension to the relational model allowing Non First Normal Form (NF^2) relations. They propose extensions to relational algebra with emphasis on new "nest" and "unnest" operations,

Table 1: Characteristics of test databases

Attribute	*Database*		
	cacm	*arman*	*wsj89*
Raw data	3 Mbytes	10 Mbytes	39 Mbytes
Number of documents	3204	628	12380
Number of unique words	5942	31838	68058
Total number of words	383182	907668	5451898
Number of transactions	112599	388066	2606670
Number of queries	50	50	50
Average number of words per query	7	94	94

Table 2: Resources used by INQUERY v1.6 on different test collections

Performance Metric	*Collection*					
	cacm		*arman*		*wsj89*	
	Keyfile	*Rdb*	*Keyfile*	*Rdb*	*Keyfile*	*Rdb*
Buffered I/O count	98	154	124	226	125	233
Peak working set size	4896	31072	6496	32120	17616	40960
Direct I/O count	345	406	1813	937	12595	3544
Peak page file size	19040	81760	20656	85072	329286	106496
Page faults	314	3174	424	4239	1193	21704
Charge CPU time (seconds)	9	12	20	66	111	163
Elapsed time (seconds)	15	22	40	85	221	246

which transform between first normal form relations and NF^2 ones. This allows the attribute domains to be sets and sets of sets, suitable for IR (*e.g.,* list of words as a single attribute).

Fuhr [4] proposes a probabilistic relational model which combines relational algebra with probabilistic retrieval. He proposes a special join operator implementing probabilistic retrieval. This model retrieves not only documents but also any kind of objects. Further, probabilistic retrieval provides implicit ranking of these objects. Fuhr argues that with independence assumptions, the relational model is a special case of this probabilistic relational model.

The above approaches demand a new design of the DBMS. This is expensive and would satisfy only IR requirements. Instead we propose a method in which the probabilistic retrieval can be done in the existing relational framework and also suggest ways to implement special join operations using SQL. Section 3 explains how IR data structures can be stored in an RDBMS so that SQL can be used on them to support probabilistic operators and special joins.

3 Adding Retrieval Capabilities to RDBMS

The two main issues that must be addressed in order to add IR capabilities to an RDBMS are the storage of IR data structures and query language support for IR operators. We observe from Section 2.1.3 that the blob implementation to store IR data structures in RDBMS is quite effective. Since our framework proposes to implement IR operators using SQL, we have a requirement that the data structures be accessible through SQL, obviously a table. If IR data structures are stored as regular tables, then it leads to poor data storage and retrieval performance.. To solve this problem, *Cooperative indexing* [3] can be used for efficient storage and retrieval. In this approach, the IR components of the system define what is extracted from documents (text attributes) along with the related index structure, and the database system provides efficient access to the index. The cooperative index can be accessed, like any regular table, through SQL. Our main focus here is to provide support for IR operators in the query language and a method to evaluate such complex queries.

3.1 Retrieval Model

Our text retrieval framework is based upon a type of Bayes net called a *document retrieval inference network* [10, 2] (which is used in INQUERY). The inference net has two components *i.e.,* the document network and the query network. The document network represents the content of the text and the query network represents the need for information. This framework creates a document network for the *text* attributes, creates a query network for the *text* component of the query, and uses the network to retrieve records that satisfy the *text* query. The result from the text and the non-text query components are combined to obtain the final result.

The document network is created automatically by mapping *text* attribute onto content representation nodes, and storing the nodes in an inverted file for efficient retrieval. For each term a separate index is constructed that stores the record identifiers, term statistics and term position information for *all* the records identified by the term. This information is stored in a relational table, say INV_LIST (TERM, DOC_ID, TF, MAX_TF, PROX), where TF is the term frequency, MAX_TF is the maximum term frequency and PROX is the position information.

3.2 Extending SQL to Support IR Operators

Text retrieval is based on partial matching and inference and thus returns scores (beliefs) as answers. These beliefs represent the relevance of a particular document (record) to the query. The traditional SQL operators are not suitable for handling beliefs since SQL operators are boolean in nature. Thus, additional text handling operators need to be added to SQL, as well as methods to combine the results from such operators with the traditional boolean operators. An extended SQL (ESQL) is defined as follows to support *text* retrieval. The ESQL will have a non-text component and a text component. The non-text component uses the regular *WHERE* conditions and operators of SQL. The following probabilistic operators [10, 2] are supported in the text component:

PAND: Probabilistic ("fuzzy") *and* of the terms in the scope of the operator.

POR: Probabilistic *or* of the terms in the scope of the operator.

PNOT: Probabilistic *negation* of the term in the scope of the operator.

PSUM: Value is the mean of the beliefs in the arguments.

PWSUM: Value is the mean of the weighted beliefs in the arguments.

Here it is assumed that all the probabilistic operators are localized to a subtree of an ESQL query. An example ESQL query on a table DOCUMENTS (DOC_ID, DATE_PUBLISHED, AUTHOR, TEXT) to get records about "operating system design" and published after "04/30/90" will be:

Example 1:

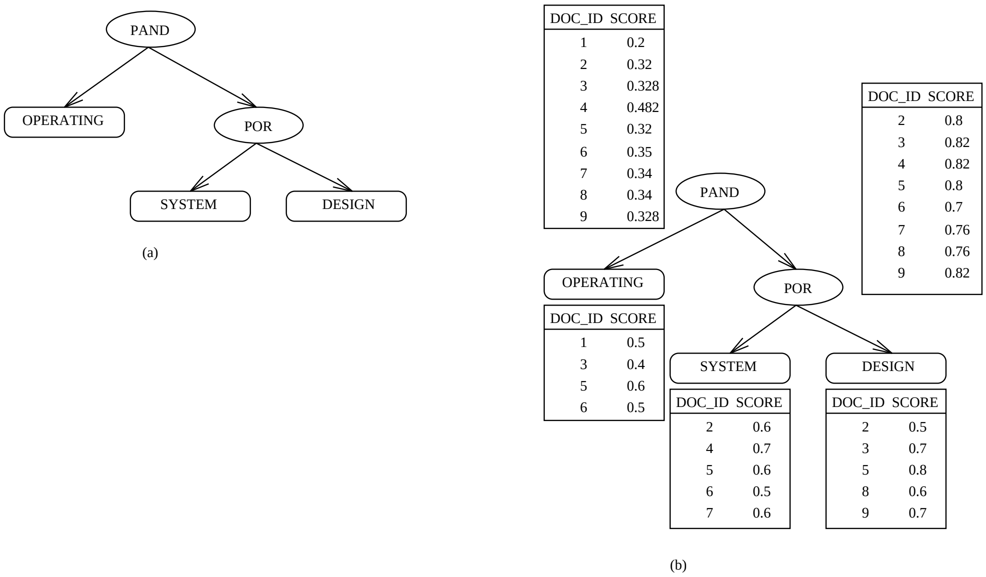

Figure 1: (a) Text query tree (b) Belief lists at different nodes

```
SELECT DOC_ID
  FROM DOCUMENTS
  WHERE DATE_PUBLISHED > '04/30/1990'
    AND TEXT_QUERY( TEXT CONTAINS 'operating'
                    PAND ( TEXT CONTAINS 'systems'
                           POR TEXT CONTAINS 'design') );
```

The query tree for the text component of the query in Example 1 is shown in Figure 1(a).

3.3 Query Evaluation

An ESQL parser is used to divide the query into *text* and *non-text* components. The non-text component is evaluated using regular SQL statements. The text component is evaluated using SQL statements with external functions. External functions are used to support IR operators. The result set from such an evaluation has record identifiers and belief scores. The result set is sorted in the descending order of belief scores. A threshold is applied to the result set for the text component. The threshold can be either the top n records or records greater than a specific threshold (say 0.4). Finally, the result set from the non-text component is used as a filter to generate the final result set.

A query network is created from the *text* component of the user query. In this section we show how the *text* component of the query can be evaluated using SQL and later combined with the non-text component. The query evaluation can be term-at-a-time or record-at-a-time.

3.4 Term-at-a-Time Processing

In term-at-a-time processing, each node in the query tree is evaluated for all documents or records. We evaluate the tree bottom up, as follows.

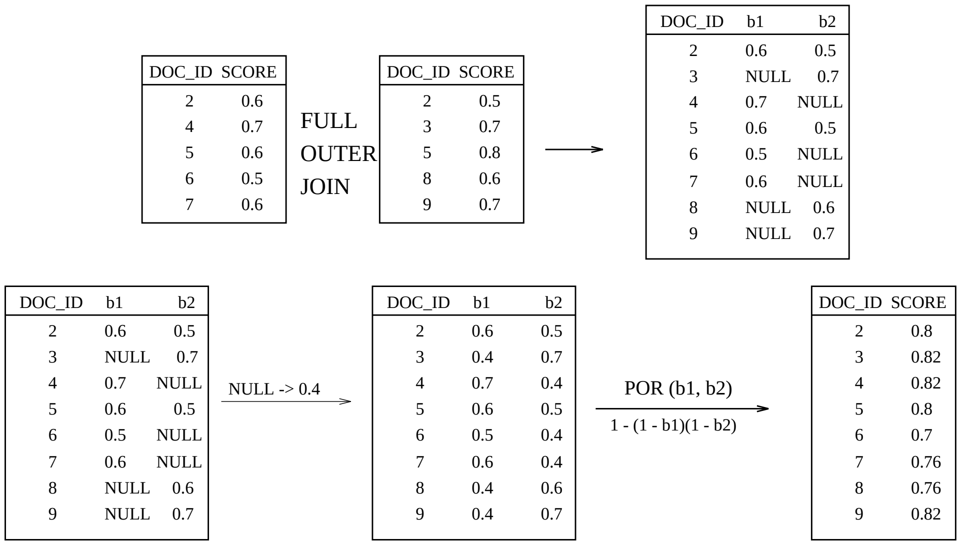

Figure 2: Special Join operation

3.4.1 Generating belief lists at the leaf nodes

Belief lists are generated for each leaf node. A belief list is a list of record identifiers and associated belief values at a given node, as well as default beliefs and weights. Node belief scores are calculated [10, 2] and normalized using the statistics stored in the inverted list (INV_LIST table). Belief lists can easily be generated from INV_LIST in SQL. External functions [12] are used to calculate the belief scores. The belief lists at the leaf nodes for Example 1 are shown in Figure 1(b).

3.4.2 Evaluating probabilistic operators

The probabilistic operators in the query tree are evaluated with a bottom-up strategy. Each operator is evaluated by executing a special join operation (different for different operators) on the belief lists of its children. A special join is achieved in two steps:

- A *full outer join* [12] of the two belief lists of the children is done, replacing all the NULLs with a default belief value, such as 0.4.
- Combine the two belief values for each record using the formula for each operator [10]:
  1. POR: $1 - (1 - b_1)(1 - b_2)$
  2. PAND: $b_1 * b_2$
  3. PNOT: $1 - b_1$
  4. PSUM: $\left(\frac{b_1 + b_2}{2}\right)$
  5. PWSUM: $\left(\frac{(w_1 b_1 + w_2 b_2) w}{w_1 + w_2}\right)$ where w_1 and w_2 are weight associated with the child nodes.

  This is also implemented using external functions.

The output of the special join is again another belief list. By evaluating all the nodes, bottom-up, we will finally have a belief list at the root, which is a list of record identifiers and belief values. The special join operation for POR node in Example 1 is shown in Figure 2.

3.4.3 Generating the final result set

The non-text component of the ESQL query is later applied as a filter on the result set of the previous step to obtain the final result set. The belief values in the belief list are used to rank order the result set. The `ORDER BY` clause in SQL can be used to rank order the records. The number of records in the result set is restricted by either using a *threshold* on the belief value or by using the *top n* records. If threshold is used, then a `WHERE` clause like `BEL > 0.3` can be used, where `BEL` is the belief for this document. If the top n strategy is used, then a condition like `LIMIT TO n ROWS` can be used.

The SQL statement for evaluating the text component of the ESQL query of Example 1 is as follows:

```
SELECT DOC_ID, (1 - (COALESCE(T1.B1, 0.4) * COALESCE(T2.B2, 0.4)))
  FROM ((SELECT DOC_ID, BEL(TF, MAX_TF, DOC_FREQ)
           FROM INV_LIST
           WHERE TERM = 'system' ) AS T1 (DOC_ID, B1)
        FULL OUTER JOIN
        (SELECT DOC_ID, BEL(TF, MAX_TF, DOC_FREQ)
           FROM INV_LIST
           WHERE TERM = 'design' ) AS T2 (DOC_ID, B2)
       ) AS T4 (DOC_ID, B4);
```

Here `BEL()` is an external function which calculates the belief score for a record.

3.5 Record-at-a-Time Processing

In contrast to term-at-a-time processing, where each query node is evaluated for all the records, in this method the entire query tree is evaluated for each record. This can be better because it avoids the expensive special joins. The non-text query is used as a filter to obtain the record set on which the text query is evaluated. For each record in the filtered record set, we do the following:

Step 1: Calculate the belief value for all the leaf nodes (query terms) for this record. The belief value is calculated from the inverted list (INV_LIST) as discussed in Section 3.4.

Step 2: Evaluate the entire query tree for this record. All the probabilistic query operators are implemented as external functions [12]. These external functions take two belief values as their arguments and return another belief value. Thus these external functions can be nested. Since nesting can be done, the entire query is easily implemented. If b1, b2 and b3 are the belief scores for a specific record (say `DOC_ID = ID_1`) at the leaf nodes for Example 1, the text component is evaluated as shown in the SQL statement below. Here `PAND` and `POR` are external functions. It should be noted that b1, b2, and b3 are themselves SQL statements which calculate belief scores from the term statistics stored in INV_LIST.

Step 3: A threshold is applied on the final belief b (*e.g.,* b $>$ 0.5) to convert the probability into a boolean result similar to the method of Gu *et al.* [5].

```
SELECT DOC_ID FROM DOCUMENTS
       WHERE DOC_ID = ID_1
       AND PAND (b1, POR (b2, b3)) > 0.5
       AND DATE_PUBLISHED > '04/30/1990';
```

The result set is ranked in the descending order of belief using the `ORDER BY` SQL clause to obtain the final result. The SQL statement for the entire ESQL query is:

```
SELECT DOCUMENTS.DOC_ID,
  PAND ((SELECT COALESCE (BEL(TF, MAX_TF, DOC_FREQ), 0.4)
           FROM INV_LIST
           WHERE TERM = 'operating'
           AND DOCUMENTS.DOCID = INV_LIST.DOCID
        ), POR ((SELECT COALESCE (BEL(TF, MAX_TF, DOC_FREQ), 0.4)
                   FROM INV_LIST
                   WHERE TERM = 'system'
                   AND DOCUMENTS.DOCID = INV_LIST.DOCID
                ), (SELECT COALESCE (BEL(TF, MAX_TF, DOC_FREQ), 0.4)
                      FROM INV_LIST
                      WHERE TERM = 'design'
                      AND DOCUMENTS.DOCID = INV_LIST.DOCID
                    ))) AS BEL
  FROM DOCUMENTS;
```

This SQL statement would generate a table of `DOC_ID` and `BEL` for all the documents in the DOCUMENTS table. It should also be noted here that the `DOC_FREQ` in the above SQL statement is again an SQL statement like

```
SELECT COUNT(*) FROM INV_LIST WHERE TERM = 'operating';
```

Even though both the *term-at-a-time* and *record-at-a-time* approaches return the same result set, the latter has an advantage in speed since there are no JOIN operations, which tend to be expensive. More optimization can be added in Step 2, by choosing a small set of records to evaluate the query on, depending on the operators in the query.

3.6 Evaluating PROXIMITY Operators

PROXIMITY operators are those which rely on the the relative positions of the terms in a document. Some examples of PROXIMITY operators [2] are

P#*n*: A match occurs whenever all of the arguments are found, in order, with fewer than *n* words separating adjacent arguments. For example A P#3 B matches "A B", "A c B" and "A c c B".

PHRASE: Value is a function of the beliefs returned by the P#3 and PSUM operators. The intent is to rely upon full phrase occurrences when they are present, and to rely upon individual words when full phrases are rare or absent.

Evaluating proximity operators is much more complicated than evaluating the simple operators explained in earlier sections. These operators require *proximity lists* for evaluation. A proximity list contains statistical and proximity (term position) information by document for a particular term. The proximity lists should be instantiated at the term nodes of the proximity operator nodes and propagated upwards. The proximity lists are converted into belief lists before being propogated to simple operators. Proximity lists are transformed into belief values using the information in the list, and are combined using weighting or scoring functions. Belief lists may be computed from proximity lists but the reverse derivation is not possible. Creating, merging, and transforming proximity lists can all be implemented partly as external functions and partly in SQL.

4 Conclusion

An RDBMS can handle text more efficiently by storing inverted lists of the text fields in cooperative indexes, and SQL can be used to support IR operators. An extended SQL can be defined with additional IR operators. A pre-processor can be designed to transform the ESQL query into the corresponding SQL query. Performance completely depends on how efficiently cooperative indexing is implemented. More efficient implementations can be done by modifying the SQL engine to support the probabilistic operators. In this paper, we have assumed that there exists only one text field, but there can be any number of text fields, with one cooperative index for each text field. The corresponding cooperative index should be selected during ESQL processing. With such a system, both structured and unstructured data can be handled efficiently and effectively without designing a totally new system. We are presently looking at allowing probabilistic operators anywhere in the query without the restrictions that they occur together, and the impact of such a design on precision and recall.

References

[1] David. C. Blair. An Extended Relational Retrieval Model. *Information Processing and Management*, 24(3):349–371, 1988.

[2] J. P. Callan, W. B. Croft, and S. M. Harding. The INQUERY retrieval system. In *Proceedings of the Third International Conference on Database and Expert Systems Applications*, 78–83, Valencia, Spain, 1992. Springer-Verlag.

[3] Samuel DeFazio, Amjad Daoud, Lisa Ann Smith, Jagadishan Srinivasan, Bruce Croft, and Jamie Callan Integrating IR and RDBMS using cooperative indexing In *Proceedings of SIGIR 95*, ACM, July1995.

[4] N. Fuhr. A probabilistic model for the integration of IR and databases. In *Proceedings of SIGIR 93*, 309–317, ACM, June 1993.

[5] Junzhong Gu, Ulrich Thiel, and Jian Zhao. Efficient Retrieval of Complex Objects: Query Processing in a Hybrid DB and IR System. In *GESELLSCHAFT FUR MATHEMATIK UND DATENVERARBEITUNG MBH.*

[6] L. A. Macleod and R. G. Crawford. Document Retrieval as a Database Application. In D. K. Harman, editor, *Information Technology: Research and Development*, 2:43–60, 1983.

[7] A. Motro. VAGUE: A User Interface to Relational Database that Permits Vague Queries. In *ACM Transactions on Office Information Systems*, Vol 6, No 3, 187–214. July 1988.

[8] G. Salton. *Automatic Text Processing.* Addison-Wesley Publishing Company, 1989.

[9] H. J. Schek and P. Pistor. Data Structures for an Integrated Database Management and Information System. *Proceedings of the Eighth International Conference on Very Large Data Bases*, 197–206. 1982.

[10] Howard R. Turtle and W. Bruce Croft. Efficient probabilistic inference for text retrieval. In *RIAO 3 Conference Proceedings*, 644–661, Barcelona, Spain, April 1991.

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

[12] The DEC Rdb Version 6.0 Documentation Kit, Digital Equipment Corporation, 1995.