EXTENDED USER-DEFINED INDEXING WITH APPLICATION TO TEXTUAL DATABASES

by

Clifford A. Lynch and Michael Stonebraker

Memorandum No. UCB/ERL M88/18

10 March 1988

EXTENDED USER-DEFINED INDEXING WITH APPLICATION TO TEXTUAL DATABASES

by

Clifford A. Lynch and Michael Stonebraker

Memorandum No. UCB/ERL M88/18

10 March 1988

TITLE PAGE

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research supported by Defense Advanced Research Projects Agency Contract N00039-84-C-0089, Task 8.

EXTENDED USER-DEFINED INDEXING WITH APPLICATION TO TEXTUAL DATABASES

by

Clifford A. Lynch and Michael Stonebraker

Memorandum No. UCB/ERL M88/18

10 March 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research supported by Defense Advanced Research Projects Agency Contract N00039-84-C-0089, Task 8.

Extended User-Defined Indexing with Application to Textual Databases

Clifford A. Lynch
Division of Library Automation
Office of the President and Universitywide Services
University of California
Berkeley, CA 94720

Michael Stonebraker
Department of Electrical Engineering & Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

Abstract

A number of application-specific searching mechanisms, including keyword searching in textual databases, can be implemented naturally in a relational DBMS using abstract datatypes and user-defined operators. For query efficiency these operators and abstract datatypes must be supported by indices. A new indexing scheme is proposed which allows a large class of query predicates to be evaluated using indices, including many key operators for textual databases. The indexing scheme also significantly reduces the space required to store indexed textual data in a relational database system.

1. Introduction

Information retrieval systems such as online library catalogs, citation retrieval systems, and full-text database systems are usually implemented using either inverted-file database systems or special-purpose software that does not use a DBMS [Lynch 1987]. Substantial problems arise when attempts are made to use a modern relational DBMS such as INGRES or DB2 to support these applications. Examples of such problems are excessive disk space consumption and overly complex and expensive queries. This paper explores the application of user-defined operators and abstract datatypes as a means of effectively implementing information retrieval (IR) applications using relational DBMS technology.

An abstract datatype (ADT) is an encapsulated data structure that is accompanied by a set of user-defined operators with which to manipulate the ADT. The internal implementation is concealed from its users, who manipulate the ADT using its associated operators. User-defined operators can also be defined for existing (built-in) datatypes, and thus serve as an extensibility mechanism in their own right. In the early 1980s several efforts were made to incorporate ADTs into relational database systems, including ADT-INGRES [Ong et al. 1984] and RAD [Osborn & Heaven 1986]. Current research DBMSs such as POSTGRES [Stonebraker & Rowe 1985], EXODUS [Carey & DeWitt 1985, Carey et al. 1986a,1986b] and STARBURST [Schwarz et al. 1986, Lindsay et al. 1987] clearly view system extensibility as a major goal and include mechanisms to simplify the incorporation of new operators and datatypes.

In order to make user-defined operators practical for large, high-volume database applications, however, appropriate secondary index structures must provide fast access paths for query evaluation. Previous work on indexing support for ADTs and user-defined operators [Stonebraker et al. 1983a, Stonebraker 1986] showed how a variety of operators and accompanying indices could be included in a relational DBMS. This paper demonstrates that the previous indexing proposals must be generalized in order to overcome the problems inherent in using relational DBMS technology for IR applications.

Section 2 of the paper begins by defining keyword searching, a common search technique for textual data of all types. This is done in the context of a database design for an online library catalog. The space utilization and query complexity problems that arise with a standard RDBMS are illustrated. Section 3 reviews previous proposals for addressing these problems and argues the case for user-defined operators as the appropriate method for incorporating keyword searching into an RDBMS. Previous proposals for indexing such operators are examined in Section 4. Section 5 develops a new extended indexing proposal and compares the resulting performance and query utilization against a typical inverted-file DBMS, INGRES, and DB2. As a byproduct of this analysis we show that the storage system used in INGRES provides major benefits for bibliographic databases compared to that of DB2. Section 6 surveys related user-defined operators for IR applications that can be supported effectively by the proposed indexing scheme. Section 7 concludes the paper by describing a further generalization of indexing that can be used to support an even larger class of user-defined operators.

2. Database Definition and Queries for Keyterm Searching

A relational database corresponding to a typical online library catalog would include a relation BOOKS:

```
CREATE TABLE BOOKS
    (BOOK-ID INTEGER,
    TITLE LONG VARCHAR,
    other columns);
```

and a relation containing keywords from the TITLE column of books in the BOOKS relation:

```
CREATE TABLE TITLE-KEYWORDS
    (TITLE-KEYWORD VARCHAR,
    BOOK-ID INTEGER);
```

An online catalog would include similar relations for keywords extracted from other columns, such as subject headings or cataloger's notes appearing in BOOKS.

The keywords appearing in the TITLE-KEYWORDS relation that corresponds to a TITLE column value in the BOOKS relation are not simply all of the words in the TITLE column value. The precise algorithm for deriving keywords is very application-dependent. The value in the TITLE column of BOOKS will be in mixed case, and, therefore, the keywords appearing in TITLE-KEYWORDS will be converted to all uppercase (or lowercase) to permit case-insensitive searching. Words containing punctuation, such as "data-base" may generate multiple keywords (e.g., "DATA", "BASE", and "DATABASE"). Variant spellings may be accommodated by generating multiple keywords (thus the word "colour" in a title generates keywords "COLOR" and "COLOUR"). Abbreviations may be expanded (e.g., "U.S." in a title generates keywords "UNITED" and "STATES"). Some words are suppressed because they are too common to be useful for retrieval purposes (e.g., articles such as "THE" or "A").

A typical large online catalog might have 4 million tuples in the BOOKS relation and 20 million tuples in the TITLE-KEYWORDS relation. For performance reasons, indices would be created on BOOKS(BOOK-ID) and TITLE-KEYWORDS(TITLE-KEYWORD).

The database requires considerable disk space for redundant information. Specifically, the DBMS does not understand the semantics of keywords because keywords are derived from title by an application program external to the DBMS. These derived values appear once in the TITLE-KEYWORDS table proper and again in the secondary index to this relation (at least under DB2). In addition, the need to create multiple tables because a book title can have many keywords creates overhead through the BOOK-ID columns necessary to relate those tables to one another and the need to index BOOKS on BOOK-ID.

Space utilization is not a problem unique to relational databases. It also arises in inverted-file systems commonly used for information retrieval applications. An inverted-file implementation of the example database would consist of book records containing a title and all the extracted title keyterms. The title keywords would be extracted into a B-tree index, with each unique title keyword appearing in the B-tree accompanied by a list of pointers to all records containing that keyword. An inverted-file system has no built-in understanding of keywords, and thus precomputed keywords must be stored in both data and indices. Computer Corporation of America's Model 204 inverted-file DBMS includes an interesting but ultimately unsatisfactory attempt to ameliorate this problem. Model 204 allows fields in records to be defined as standard keys (both indexed and stored in the data records) or as *invisible* keys [CCA 1986]. Invisible keys are indexed and then removed from the data records. While this reduces space utilization, the reduction is accomplished at the expense of logical database integrity and consistency. There is no way to update invisible key entries in indices to reflect changes to the data records that they came from since the DBMS has no means of computing invisible key values from the remaining fields of the data record. In addition, query evaluation strategies are severely constrained because predicates involving invisible keys can be resolved only through reference to indices and not by direct examination of records selected by other predicates.

However, a relational system consumes substantially more space than an inverted-file system since BOOK-ID values connecting the multiple relations must be stored as well. The need for indices on BOOK-ID to provide adequate retrieval performance further increases storage overhead. Techniques such as invisible keys will be no more satisfactory in a relational system than they are in an inverted-file system like Model 204.

The amount of space consumed is dependent on the specifics of table storage and index data structures. It is interesting to contrast INGRES and DB2 in this regard. INGRES has the ability to construct primary or secondary indices, whereas DB2 offers only secondary indices and clustering [Selinger et al. 1979, Stonebraker et al. 1976]. Since the relations BOOKS and TITLE-KEYWORDS each have only a single index, the INGRES storage scheme allows significant space savings by allowing these tables also to serve directly as indices. In the case of INGRES, we assume that BOOKS is a hash table on a primary key of BOOK-ID and that TITLE-KEYWORDS is a B-tree on primary key of TITLE-KEYWORD. For DB2, we assume that TITLE-KEYWORDS is clustered on TITLE-KEYWORD, and that secondary indices exist on BOOKS(BOOK-ID) and TITLE-KEYWORDS(TITLE-KEYWORD).

DB2 does manage index storage more efficiently than INGRES. In particular, DB2 stores each value only once in an index, followed by a list of tuple IDs (TIDs) identifying rows containing that value. INGRES repeats the index value once for each tuple containing it by storing a (value,TID) entry. In a bibliographic database, some keyword values will appear tens of thousands of times. Implementation of a compressed storage scheme for index pages in INGRES using differential encoding techniques would be advantageous in bibliographic retrieval applications.

A typical user query against an online catalog is "find all books containing the words 'american' and 'history' anywhere in the book's title." This translates into the SQL query:

```
SELECT BOOK-ID,TITLE,other columns
    FROM BOOKS, TITLE-KEYWORDS TK1, TITLE-KEYWORDS TK2
    WHERE BOOKS.BOOK-ID = TK1.BOOK-ID
      AND TK1.BOOK-ID = TK2.BOOK-ID
      AND TK1.TITLE-KEYWORD = "AMERICAN"
      AND TK2.TITLE-KEYWORD = "HISTORY";
```

This is a reasonably complex query involving 3 joins. In general, a user query involving n keywords translates to an SQL query involving $n+1$ joins. These joins make the queries expensive, particularly when more than 2 or 3 keywords are specified.

3. Previous Proposals for Improving Bibliographic Databases with RDBMSs

A few researchers have previously examined the difficulties in using standard RDBMSs for bibliographic and information retrieval applications. [Macleod & Crawford 1983] survey this work. Papers such as [Crawford 1981, Macleod & Crawford 1983, Schek 1981] discuss some of the problems in handling keywords within the relational model and recognize that in a standard relational system separate relations for keywords are required, and consequently that keyword queries will require joins. These papers offer few proposals for resolving the problems that they identify. [Macleod 1979] suggests some cosmetic extensions using macros to simplify query formulation and some extended string-matching operators that are akin to more elaborate versions of the SQL LIKE operator. Such extended string-matching operators have also been proposed in other contexts such as document processing [Stonebraker et al. 1986]. [Schek 1981] sketches a proposal to enhance an RDBMS with a series of operators that pattern match on text fields and thus allow the searching of keywords that are appropriately encoded within the text fields (or any other substring). This approach has been refined and implemented in the AIM-II system [Dadam et al. 1986]. These proposals are not satisfactory solutions for keyword searching for the following reasons:

- Proposals for pattern-matching operators are of little use unless indices can be defined to permit their rapid evaluation. However, pattern-matching facilities are so general that the only feasible type of index structure will be similar to that described in [Schek 1981]. Such a structure requires a very large index on arbitrary string fragments and slow, complex access method algorithms that match fragment patterns by selecting candidate tuples through computations on the index and then examining the tuples. Space requirements and performance from such an index will be unacceptable in a large database.

- The extraction of keywords is a sufficiently complex, algorithmically oriented process that it is unlikely to be expressed through any reasonable set of pattern-matching operators. At best, enormously complex patterns will be required which will be computationally expensive.

- Proposals to add built-in operators specifically to match fields that contain a keyword do not make sense since, as previously discussed, keyword extraction is highly application-dependent. It is not feasible to develop a standard keyword matching operator that will meet the needs of textual applications.

Set-valued relations [Zaniolo 1983] offer a way to avoid joins. The BOOKS relation might be redefined (using Zaniolo's GEM notation for sets, adapted for SQL) as:

```
CREATE TABLE BOOKS
    (BOOK-ID INTEGER,
    TITLE LONG VARCHAR,
    TITLE-KEYWORDS {VARCHAR},
    other columns);
```

and a query for books by title keyword "history" in the set-valued relation would be specified as

```
SELECT * FROM BOOKS WHERE "HISTORY" IN TITLE-KEYWORDS;
```

However, the availability of sets does not eliminate the need to store keyterms redundantly both in the relation proper and again in the index. Additionally, proposals for set-valued relations do not speak to an indexing strategy for members of a set comprising a column value and have not been generalized to permit set elements that are ADTs. The indexing proposal presented here can be readily extended to work for an RDBMS that has been enriched to include sets as a datatype, and complements set-valued relations well.

Nested relations [Dadam et al. 1986] can be viewed as a generalization of set-valued relations. They could be used to provide much the same effect as set-valued relations: the title keywords for each title could be defined as a single-column relation. Nested relations share with set-valued relations a high storage overhead due to the need to redundantly store the keywords in the relation and in an index, and again proposals for nested relations do not fully address the indexing issue. Finally, a nested relation implementation of a large bibliographic database would give rise to a database containing millions of relations; this is likely to be quite cumbersome.

4. Extended Secondary Indices, User-Defined Operators, and Abstract Datatypes

Keyword derivation is a rather ad-hoc, database- and application-specific process, best implemented by the developer of a particular application using procedures written in a programming language. By its nature, keyword extraction is not a database primitive. The natural and appropriate tools for this type of application-specific extension within a DBMS are abstract datatypes and user-defined operators. However, to be practical, user-defined operators must be accompanied by secondary indices. Previous proposals reviewed below do not provide the necessary indexing capability and must be generalized.

[Stonebraker et al. 1983a] (and subsequently [Stonebraker 1986], which greatly extended, simplified, and generalized the proposal from the original paper) developed a detailed scheme for defining ADTs and user-defined operators in database systems. Perhaps the most important contribution of these two papers is their recognition that ADTs and accompanying operators must be supported by secondary indices to be viable in many real-world contexts. Without the performance such indices provide, ADTs have limited utility as practical tools for building production applications. Thus, a facility called *extended secondary indices* was also proposed, which provides the following capabilities:

- The ability to create indices on ADT columns with existing operators.
- The ability to create indices on ADT columns to support new user-defined operators.
- The ability to create indices on non-ADT columns (e.g., existing built-in datatypes) to support new user-defined operators.

The proposed facility can be summarized as follows. Note that the proposal of [Stonebraker 1986] has been recast from QUEL to SQL and some of the terminology has been changed here.

1. ADTs are registered with the DBMS; the definition includes the specification of a pair of functions to convert the ADT to and from character form, which are used to support input and output of the ADT.

2. New operators can be registered with the DBMS. The main case considered is binary infix operators, where one defines the datatypes of the left- and right-hand operands and the operator's result, the operator's precedence, and the name of a function that implements the operator.

3. Restricted classes of Boolean-valued binary operators, in which both arguments have the same datatype, may be supported through B-tree indices using the B-tree access method built into the DBMS. The classes of operators that can be supported through the B-tree access method are those that can play the same role as the usual comparison operators with respect to the datatype upon which they operate. To construct a B-tree consisting of instances of a given datatype, it is necessary to have an operator that provides an ordering on that datatype analogous to the $\leq$ operator on numeric or character datatypes. This B-tree can be searched for entries satisfying operators analogous to any of the operators $\{\leq, \geq, =, >, <\}$ using this comparison operator. Other restricted classes of operators can be supported through other access methods which may be included in the DBMS. The specific restrictions are access-method-dependent. In this paper we will consider only B-tree indices.

A user-defined *operator class* is established for B-trees by providing a name for the class and supplying a list of user-defined operator names, and specifying the correspondence between the user-defined operators and the standard B-tree operators $\{<, >, =, \leq, \geq\}$. (See [Stonebraker 1986] for details.) Any built-in datatype that can be ordered using the usual comparison operators (e.g., integers or strings) is assumed to have an associated default ordering class consisting of the standard comparison operators.

A B-tree index to support a specific ordering operator class can be created through the SQL statement

CREATE INDEX *index-name* ON *table* (*column*) ORDERING *operator-class-name*

The analog to $\leq$ in *operator-class-name* is used to place the values that appear in *column* into a B-tree structure. Subsequently, predicates of the form (*column relop value*) can be supported through this B-tree index when *relop* is an operator that is a member of the user-defined ordering operator class specified in the CREATE INDEX statement. The ORDERING clause is compatible with current query language usage in that, if it is omitted, the built-in ordering operator class is used when *column* contains a built-in datatype known to the DBMS, such as integer or character string.

Two approaches to formulating keywords with user-defined operators are possible. Neither approach allows useful indexing to support the operators under the proposal of [Stonebraker 1986]. The first approach uses an ADT for sets of strings. Define a unary operator, KEYWORDS, on strings returning a set-of-strings ADT containing all the keywords from the input string. Define CONTAINS as a Boolean-valued binary operator with one ADT set-of-strings operand and one string operand. CONTAINS is true if the string operand is a member of the set specified by the set-of-strings operand. Using these operands a user query such as "find all books with the word 'history' in the title" can be formulated as:

```
SELECT * FROM BOOKS WHERE KEYWORDS(TITLE) CONTAINS "HISTORY";
```

The indexing proposal of [Stonebraker 1986] does not allow rapid evaluation of this query for two reasons. The two operand datatypes of the CONTAINS operator are not identical, and thus CONTAINS cannot be a member of an operator class. Additionally, even if CONTAINS could be indexed somehow, the presence of the unary operator KEYWORDS in the WHERE clause of the query prohibits the use of an index to evaluate the predicate. The first objection can be overcome by redefining CONTAINS as an operator on pairs of sets-of-strings (where A CONTAINS B is true if every member of B is a member of A). However, this more general CONTAINS operator cannot be indexed using [Stonebraker 1986] because it is not analogous to one of the ordering operators $\{=, >, <, \leq, \geq\}$ in any operator class.

The second approach defines a Boolean-valued binary operator on strings, CONTAINS-KEYWORD. A CONTAINS-KEYWORD B is true if B is a keyword contained in the string A. With this approach, the user request for all books with the word 'history' in the title becomes the SQL query

```
SELECT * FROM BOOKS WHERE TITLE CONTAINS-KEYWORD "HISTORY";
```

Again, the indexing proposal of [Stonebraker 1986] provides no help in evaluating this query. The problem is that the CONTAINS-KEYWORD operator is not analogous to any of the comparison operators and thus cannot be a member of an operator class.

5. Generalized Extended Secondary Indices

The keyterm searching problem is an instance of a general retrieval problem that seems likely to arise in a wide range of applications. One has a table with a column C of datatype *D1*, and a unary operator U which takes an argument of type *D1* and returns a datatype *D2* or set-of-*D2*. There is a B-tree operator class on *D2*, and an index is required to evaluate predicates of the form $(U(C)\ opr\ v)$, where *opr* is a member of the operator class and v is a (constant) value of datatype *D2*. The following extensions to the scheme described in [Stonebraker 1986] add the functionality necessary to create indices in support of this class of predicates.

5.1 List Datatypes

A new set of datatypes called LISTs is defined. It is proposed that these be built-in, rather than user-defined, datatypes for the following reasons:

- Building in lists allows the DBMS to extend automatically most built-in or user-defined operators on other datatypes to lists of these datatypes. The inheritence technique used to extend these operators is described below.

- The DBMS will need to understand the semantics of lists in order to implement the extensions to indexing discussed below. If lists are to be user-defined datatypes, an ad-hoc parameter passing mechanism will have to be defined to support indexing.

Almost everything discussed below can be accomplished with lists as a user-defined rather than a built-in datatype, at the expense of less attractive syntax and more effort for the user in explicitly extending operators to lists.

A list is a set of zero or more instances of a specific datatype; the datatype may be a built-in datatype or a user-defined ADT. Thus, there are datatypes LIST-OF-INTEGER, LIST-OF-CHARACTER, LIST-OF-REAL, etc. The syntax for defining a list encloses its elements with braces, for example, {1, 2, 3, 4} or {'a,' 'b,' 'c,' 'd'}. Lists of lists (of a specific type) are permitted.

Built-in or user-defined operators on instances of a given datatype extend to lists of that datatype as follows. Assume that $\{x_i\}$ and $\{y_j\}$ are lists of the appropriate datatype.

- A unary operator (in functional notation) F applied to a list $\{x_i\}$ returns a new list $\{F(x_i)\}$.

- Any binary-valued operator *OPR* where both arguments are of the same datatype permits a list of that datatype for either or both arguments, and $\{x_i\} OPR \{y_j\}$ returns a new list with $i * j$ entries $\{x_i \ OPR \ y_j\}$. If either argument list is empty, the result is an empty list.

- The exception to the preceeding rule is that lists of Boolean values are not permitted. While Boolean-valued operators extend to permit lists as arguments, they continue to return Boolean values. A value of true is returned if the operator returns true for any $\{x_i \ OPR \ y_j\}$ and false otherwise. These rules define the operator =, when extended to lists, to have the semantics that $\{x_i\} = z$ is true if and only if $x_i = z$ for some i ; $\{x_i\} = \{y_j\}$ is true if $x_i = y_j$ for some pair (i, j). This exception permits built-in comparison operators to extend to lists gracefully. An alternative way of formulating the same requirement would be to allow lists of Boolean values, and to say that when a predicate evaluates to a list of Boolean values it is considered true if the list *contains* the value true, and false only if *all* entries in the list of Boolean values have the value false.

Through these rules all built-in arithmetic operators extend immediately to lists of integers or reals; all comparison operators $\{=, >, <, \geq, \leq\}$ extend to lists of any built-in datatypes on which they are defined (such as integers and strings). Because of the rule for extending Boolean-valued binary operators above, if L is a list of integers, for example, $(L > X)$ and $(L < X)$ may both be true. Thus the standard comparison operators do not form a B-tree operator class on LIST-OF-INTEGERS since they do not create an ordering on these lists.

5.2 Indexing

We proposed to extend the CREATE INDEX statement to permit another parameter OPERATOR *operator-name* in addition to the ORDERING parameter of [Stonebraker 1986].

The semantics of

CREATE INDEX *index-name* ON *table* (*column-name*) ORDERING *operator-class-name*
OPERATOR *operator-name*

are as follows:

1. The *operator-name* operator must be a unary operator that has an argument datatype equal to the datatype of the column being indexed. It can return any datatype including list of a datatype.

2. If *operator-name* returns an ADT, the ORDERING parameter must also be supplied to define an ordering operator class for that ADT type that will be used to build the index. This is necessary because the DBMS must know how to order the ADT to build the index. A nonstandard (user-defined) ordering operator class can be employed to construct an index on a built-in datatype returned by *operator-name* by specifying the ORDERING parameter to identify the nonstandard ordering operating class. If *operator-name* returns a built-in datatype, the ORDERING parameter may be omitted, and the standard built-in ordering operator class for that datatype will be used by default to build the index if such a class exists. (If the DMBS does not know how to order the datatype returned by *operator-name* and is not instructed how to do so through specification of an ordering operator class through the ORDERING parameter of the CREATE INDEX statement, the attempt to construct the index is terminated with an error indication.)

3. As the index is created, each value in *column-name* is passed to the operator *operator-name*. If the operator returns a single value, that value (along with the TID of the relevant row) is placed in the index. If the operator returns a *list*, then entries are placed in the index for each *element* of the list, along with the TID of the relevant row. If null values are allowed in the relation, it will be desirable to allow the operator to return zero values (indicating that nothing is to be stored in the index for the given tuple) or a null value for the returned datatype, depending on the specific application.

The choice of processing here, depending on whether the operator returns a list, is the only point where DBMS *must* understand the semantics of lists. If lists are provided as user-defined rather than built-in datatypes, some ad-hoc method must be used to allow lists to be passed back from the operator.

4. An index built through this construct can be used to resolve predicates of the form

(*operator-name*(*column-name*) *relop value*)

where *relop* is any operator in the B-tree operator class used to build the index (either the default ordering operator class or one explicitly specified through the ORDERING parameter). Resolving the predicate is accomplished simply by looking up *value* in the B-tree index that has been created on *operator-name*(*column-name*) using the ordering operator class. This proposal is upwardly compatible to the proposal in [Stonebraker 1986]. If no OPERATOR parameter is specified in the CREATE INDEX statement, then the index can be used to resolve predicates of the form (*column-name relop value*) where *relop* is a member of the ordering operator class specified in the ORDERING parameter. Note that the two predicate types (*column-name relop value*) and (*operator-name* (*column-name*) *relop value*) cannot be supported through the same index.

SQL also permits the creation of indices using multiple columns through the syntax

CREATE INDEX *index-name* ON *table* (*column1,column2,...,columnk*)

Specifying k-ary operators rather than unary operators in the OPERATOR keyword of the extended CREATE INDEX statement permits a straightforward accommodation of this more general form of index construction, thus allowing the construction of an index that can be used to quickly evaluate predicates of the form

(*operator-name*(*column1,column2,...,columnk*) *relop value*).

5.3 Applications to Keyterm Searching

One operator needs to be defined to extract all keywords from a string:

```
DEFINE OPERATOR TOKEN=KEYWORDS,
          ARGUMENT1=CHARACTER,
          RESULT=LIST-OF-CHARACTER
```

The table TITLE-KEYWORDS then can be eliminated, along with its associated index. In its place, an additional index on the BOOKS relation can be built:

```
CREATE INDEX TITLE-KEYWORDS ON BOOKS(TITLE) OPERATOR KEYWORDS;
```

Using this operator, a search for all books with titles containing a specified keyword (for example, "DATABASE") can be formulated as

```
SELECT * FROM BOOKS WHERE KEYWORDS(TITLE) = "DATABASE";
```

Here the operator = is being extended as discussed above to permit a LIST-OF-CHARACTER datatype on the left and a CHARACTER datatype on the right. Similarly, all books with titles containing keywords beginning with the prefix "COMPUT" ("COMPUTERS", "COMPUTING", etc.) can be requested by

```
SELECT * FROM BOOKS WHERE KEYWORDS(TITLE) LIKE "COMPUT%";
```

5.4 Effects on Query Processing Costs

Comparisons will be made among four environments: an inverted-file system such as ADABAS [Software AG 1982] (which is commonly used in real bibliographic retrieval systems today, and thus provides a performance baseline for other implementations); standard INGRES; standard DB2; and a relational system incorporating the extensions proposed in this paper. The results of this analysis are summarized in Table 1. In the analysis, we assume that the database consists of the two tables defined at the beginning of Section 2 for the INGRES and DB2 cases, and that in the extended relational case the database consists of a single relation with secondary index as defined in Section 5.3.

We will compare the number of reads necessary to evaluate queries. A hash table lookup is assumed to be 1 read; a B-tree lookup is assumed to be 3 reads (the effects of caching index blocks in the buffer pool are ignored). Storage pages are assumed to be 4K. We assume that a TID or an inverted-file record number is 4 bytes, and thus about 1000 TIDs fit on a storage page. We assume that title keywords average 9 characters in length, and that the integer values for BOOK-IDs require 4 bytes. We assume that about 300 tuples from the TITLE-KEYWORDS relation fit on a storage page since each tuple averages 14 bytes including a length count for the variable-length keyword.

Consider a single keyword query, such as "find all books with the word 'packet' in the title." This translates into an SQL query:

```
SELECT BOOK-ID,TITLE,other columns FROM BOOKS,TITLE-KEYWORDS WHERE
    BOOKS.BOOK-ID = TITLE-KEYWORDS.BOOK-ID AND
        TITLE-KEYWORDS.TITLE-KEYWORD = "PACKET";
```

in standard DB2 or INGRES, and into the query

```
SELECT BOOK-ID,TITLE,other columns FROM BOOKS WHERE KEYWORDS(TITLE) = "PACKET";
```

in the extended relational system.

Assume that there are n books containing the keyword "packet" in the title. For the inverted-file system, the query requires one index lookup (3 reads), $n/1000$ reads to obtain the inverted list, and n reads to actually fetch the records, for a total of $3 + n/1000 + n$ reads. For INGRES, one index lookup on TITLE-KEYWORDS is required (3 reads), followed by $n/300$ page reads to obtain all of the tuples and BOOK-IDs; n hash table lookups are then required against the BOOKS relation to obtain the actual records, for a total of $3 + n/300 + n$ reads. If differental encoding is used to store the TITLE-KEYWORDS relation in unextended INGRES, query cost is equivalent to that of the inverted-file system.

For DB2, the situation is much worse. One index lookup (3 reads) and $n/300$ reads of tuples in TITLE-KEYWORDS are needed to obtain BOOK-IDs. Each of the n BOOK-IDs must then be looked up (at 3 reads per lookup) in the BOOK-ID index to BOOKS. After each BOOK-ID is looked up, the corresponding tuple from BOOKS must be read. The total cost is $4n + n/300 + 3$ reads.

With the proposed extension, DB2 requires only $3 + n/1000 + n$ reads, as does INGRES with differential encoding on the secondary index built through the extended indexing mechanism. Without differential encoding, the performance of INGRES with the proposed extension is unaltered.

Consider a two-keyword query, such as "find all books with the words 'computer' and 'art' appearing in the title." This turns into an SQL query like

```
SELECT BOOK-ID,TITLE,other columns FROM BOOKS WHERE
   KEYWORDS(TITLE) = "ART" AND KEYWORDS(TITLE) = "HISTORY';
```

in the extended RDBMS. The query for the standard DBMS is identical in structure to the example query at the beginning of the paper. Assume that there are 12,000 books that have a title containing the keyword COMPUTER and 17,000 that have a title containing the keyword ART, and assume that there are 40 books where the keywords COMPUTER and ART both appear in the title. In analyzing standard INGRES and DB2 we will assume (optimistically) that the query planner chooses the most selective predicate as its access path and that there is sufficient memory to maintain one part of the join in memory.

In an inverted-file system like ADABAS, this query would require two index lookups (one for each keyword) and the reading into memory of two record pointer lists, one of 12 pages and one of 17 pages. These two lists of pointers would be intersected to find the records containing both keywords, and the 40 resulting record pointers would be used to read 40 records. The total is 75 reads.

INGRES will perform one lookup and 40 page reads to load the tuples in TITLE-KEYWORDS satisfying TITLE-KEYWORD="COMPUTER" into memory, and then perform an index lookup and 56 page reads to run through the tuples in TITLE-KEYWORDS satisfying TITLE-KEYWORD="ART", matching each against the incore tuples from the first predicate. This will result in 40 tuples, each of which has to be read from BOOKS for a total of 142 reads. If differential encoding is used to store TITLE-KEYWORDS, then the performance is equivalent to the inverted-file system in terms of I/O as long as the tuples satisfying the most selective predicate can be maintained entirely in memory. It is worth noting, however, that the processing done by INGRES to resolve the join will be much more CPU-intensive than the pointer list intersection performed by the inverted-file system. In addition, if the smallest set of tuples cannot be maintained in memory, the I/O cost for INGRES without differential encoding becomes 1927 reads; with differential encoding it is 271 reads.

Again, the situation with DB2 is much worse. Basically the same processing logic is followed, but it requires 160 reads instead of 40 to obtain the resultant tuples from BOOKS. In addition, an extra read is required after each index lookup to start the sequential scan of tuples in TITLE-KEYWORDS. Thus, DB2 will require 264 reads.

For the proposed extended relational database, this query would require one index lookup (3 reads) followed by reading 12 index pages that identify 12,000 rows in BOOKS (assuming that the extended RDBMS selects the optimal access path). These rows would be read and scanned to resolve the index. The total is 12,015 reads. The reason that this performs badly, however, is that the evaluation strategy for the Boolean AND is not appropriate. If the DBMS knew the strategy of looking up the other predicate involved in the AND, first intersecting the TID lists and then reading the TIDs resulting from the TID list computation, the performance would be identical to that of the inverted-file implementation. (See [Lynch 1987] for a discussion of this query processing strategy.)

In general, if there are two keywords, the first identifying x books and the second y books ($x \leq y$), and there are z books containing both keywords, then the inverted-file system takes 6 + $x/1000 + y/1000 + z$ reads; INGRES requires $6 + x/300 + y/300 + z$; and the extended relational system requires $3 + x/1000 + x$ reads. With appropriate query processing strategies, the extended relational system requires $min\{6 + x/1000 + y/1000 + z, 3 + x/1000 + x\}$ reads.

5.5 Effects on Space Utilization

Assume that the database contains 4 million books, that the average title is 45 characters long, and that the average title keyword is 9 characters long; we assume 5 keywords per title on average. Assume further that about 1 million unique title keywords occur in the database. There are 20 million occurrences of title keyterms. (These values are consistent with actual observed figures for bibliographic databases of this size, such as the University of California's MELVYL®online catalog [Lynch 1987].) We analyze the space required in order to provide an index on title keyterms. The results are summarized in Table 2.

The inverted-file system will store every title keyterm occurrence in its record in the data records (20 million * 9 bytes), the unique title keywords in the index (1 million * 9 bytes), and 20 million pointers in the index. This totals 215MB.

Standard INGRES will require 24 million BOOK-IDs to connect the TITLE-KEYWORDS and BOOKS relations (20 million in TITLE-KEYWORDS and 4 million in BOOKS), plus one copy of the title keywords (20 million * 9 bytes, or about 1 million * 9 bytes if differential encoding is used). This totals 105MB if differential encoding is used in the TITLE-KEYWORD relation and 276MB if differential encoding is not used.

Standard DB2 will store an extra (differentially encoded) copy of the keywords in an index (89MB) and an index for BOOKS on BOOK-ID (8 bytes * 4 million, or 32MB), for a total of 226MB if the TITLE-KEYWORD relation is stored with differential encoding, and 397MB if differential encoding is not used.

The extended relational system will store 1 million * 9 bytes of keywords and 20 million * 4 bytes of pointers (assuming a differential encoded index) for a total of only 89MB.

6. Other Applications of User-Defined Operators and Generalized Extended Secondary Indexing

The same need for lists of values derived from columns appears in many other contexts in bibliographic databases. In this section we consider a few of these situations.

6.1 Searchable vs. Displayable Forms

Typically, users want to search independently of case and without regard to most punctuation, accent marks, and special characters. In addition, when specifiying full titles or subject headings, users want to search independently of the presence or absence of a leading article. Thus, a second copy of each field in a bibliographic record is normally maintained which has been converted to a suitable form for matching against search criteria entered by the user at a terminal, along with the "full" field suitable for display to the user as a search result. These two forms are called *searchable* and *displayable* fields respectively. The precise conversion process from displayable to searchable form is complex and varies from system to system, but is similar to keyword extraction. The searchable form of the field is derivable from the displayable form (which must be retained in the database) and its only purpose is to serve as an access path into the database.

In a standard RDBMS one would construct the BOOKS table as

```
CREATE TABLE BOOKS
    (BOOK-ID INTEGER,
    DISPLAYABLE-TITLE LONG VARCHAR,
    SEARCHABLE-TITLE LONG VARCHAR,
    other columns);
```

with an index on the SEARCHABLE-TITLE column. By defining a unary operator SEARCHABLE that returns the searchable form of the string that is passed as the function argument, this table could be simplified by eliminating the SEARCHABLE-TITLE column and creating an index:

```
DEFINE INDEX SEARCHABLE-TITLES ON BOOKS(DISPLAYABLE-TITLE) OPERATOR SEARCHABLE;
```

Using this new operator, one can search for books containing a given title through a query such as

```
SELECT * FROM BOOKS WHERE SEARCHABLE(DISPLAYABLE-TITLE) = "THE WINDS OF WAR";
```

6.2 Personal Name Indexing

(Personal) author names provide an interesting example of a rather different keyword extraction algorithm. A tuple for a book usually will contain a full author name, such as JOHN JACOB ASTOR, as an additional field. A user can specify many forms of a name that should match this author name, such as ASTOR; ASTOR,J.; ASTOR,J.J.; ASTOR,JOHN; ASTOR,JOHN JACOB; or ASTOR,J. JACOB.

To support this type of access, a series of "name keywords" are extracted from each name in the database using a rather complex algorithm [DLA 1987]. Each keyword that is not from the last name is prefixed with a character that cannot occur in a name (the symbol @ is used in the example below); these keywords denote initials, first names, and middle names. The number of prefix characters gives the "type" of the extracted keyword (e.g., one for first initial or first name, three for middle name, etc.). Essentially, these special characters are used to avoid requiring separate indices on first name, first initial, middle name, first and middle initials, etc. For example, the name above might produce the name keywords: ASTOR,@J,@JOHN,@@JJ,@@@JACOB. Clearly, this personal name keyword extraction can be implemented by a user-defined operator, say NAMEKT, where NAMEKT("JOHN JACOB ASTOR") = {"ASTOR", "@J", "@JOHN", "@@JJ", "@@@JACOB"}.

When various forms of the name are encountered by the user interface, the interface generates name keywords as follows:

Name entered by user	Name keywords generated
ASTOR,J	ASTOR,@J
ASTOR,JJ	ASTOR,@@JJ
ASTOR,JOHN	ASTOR,@JOHN
ASTOR,JOHN JACOB	ASTOR,@JOHN,@@@JACOB

Searches for personal names result in predicates of the form (NAMEKT(AUTHOR) = *name-keyword*) being included in the query constructed by the user interface, with one predicate for each name keyword generated by the user interface. For example, for the end-user query

FIND AUTHOR ASTOR, J

the user interface will generate two name keywords for "ASTOR" and "@J" and construct the query

```
SELECT * FROM BOOKS WHERE
   NAMEKT(AUTHOR) = "ASTOR" AND NAMEKT(AUTHOR) = "@J";
```

7. Compatible Operators and Generalized Secondary Indices

Even with the extensions proposed earlier in this paper, many operators cannot be indexed. The most general type of predicate that can be supported by indices thus far has the form (*operator*(*column*) *relop value*), where *operator* is a unary function. Additionally, *relop* in the predicate above can only be one of five possibilities $\{ = , \ < , \ > , \ \leq, \ \geq \}$ for B-trees. This section defines *compatible* operators and describes how they can be supported through B-tree indices. This is a general extension that allows a much larger class of predicates to be supported by a single extended secondary index.

Consider a B-tree index on a column C of datatype D constructed using an operator class $OC = \{=, <, >, \leq, \geq\}$. Predicates are resolved using this index as follows in a standard RDBMS:

- For ($C <$ *value*) or ($C \leq$ *value*), find the smallest key in the B-tree index. Proceed sequentially through the pages of the B-tree from this smallest key, testing each new key encountered to ensure that (*key* $<$ *value*) or (*key* $\leq$ *value*) remains true. If this relationship is true, then the index entry for the new key specifies a TID satisfying the predicate. If the relationship is false, stop the index scan; all relevant TIDs have been found.

- For ($C =$ *value*), look up *value* in the B-tree through a standard comparison search. If a match occurs, proceed sequentially as for $<$ and $\leq$ until a key is encountered such that (*key* $=$ *value*) is false.

- For ($C \geq$ *value*), look up *value* in the B-tree through a standard comparison search. If value does not exist in the B-tree, locate the smallest key $>$ *value*. All keys from that point are considered to match the predicate.

- For ($C >$ *value*), look up the smallest key in the B-tree that is greater than *value* through a standard comparison search. All keys from that point are considered to match the predicate.

This search mechanism can be generalized to allow many additional Boolean-valued binary operators (with both arguments of datatype D) to be supported using this B-tree index on C. Let @ be such an operator. Predicates of the form

(C @ *value*)

can be resolved using the B-tree access method and the index for C can be built using the class OC, if the access method is provided with a unary operator F on datatypes D (returning datatype D) that provides a scan start point, and if the operators F and @ together have the properties

C1: if $X < F(Y)$, then $(X$ @ $Y)$ is false.
C2: if $Y > F(X)$ and $(Y$ @ $X)$ is true, then for all Z such that
$F(X) < Z < Y$, $(Z$ @ $X)$ is true.
C3: if $Y > F(X)$ and $(Y$ @ $X)$ is false, then for all Z such that
$Y < Z$, $(Z$ @ $X)$ is false.

Intuitively, these axioms require that the computed start point $F(A)$ is either the first key satisfying the predicate $(C$ @ $A)$ or that the smallest key $> F(A)$ is the first key satisfying the predicate, and that the values in the index satisfying the predicate appear as a single contiguous block of values starting at $F(A)$ (or the first key greater than $F(A)$) in the ordering defined by the operator ordering class OC used to build the B-tree.

An operator that satisfies the axioms C1–C3 for an operator ordering class OC is said to be ***compatible with the operator class*** OC. A mechanism is required to indicate that an operator is compatible with an ordering class (and consequently any B-tree built using this operating class can be used to resolve predicates involving the compatible operator). As part of this mechanism, the unary function F that supplies the scan start point to the B-tree access method must be provided. The following simple directive is proposed:

operator COMPATIBLE WITH *ordering-class* START *function type*

where *operator* is the name of the compatible operator, *ordering class* is the name of an ordering operator class (on some datatype D), and *function* is the name of a unary operator on datatype D returning a datatype D that supplies the scan start point for the access method. *Type* is either "EQUAL" or "GREATER" and indicates whether the scan start point is the first value in the index *equal* to the value returned by *function*, or the smallest value in the index *greater* than the value returned by *function*. As with a standard B-tree scan, processing stops when the first key in the B-tree after the start point fails to satisfy the predicate being evaluated.

Compatible operators have a wide range of uses. For example, consider the ordering operator class consisting of the standard built-in operators on strings, and suppose we have a column C of datatype string where all strings are at least four characters. The same B-tree used to index the column C can also be used to support a set of compatible operators $\{@ =,\ @ <,\ @ \geq,\ @ \leq,\ @ >\}$ that compare the four-character initial stems of strings in C defined as follows:

s_1 @= s_2 if SUBSTR(s_1,1,4) = SUBSTR(s_2,1,4)
s_1 @< s_2 if SUBSTR(s_1,1,4) < SUBSTR(s_2,1,4)
s_1 @> s_2 if SUBSTR(s_2,1,4) > SUBSTR(s_2,1,4)
s_1 @≤ s_2 if SUBSTR(s_1,1,4) ≤ SUBSTR(s_2,1,4)
s_1 @≥ s_2 if SUBSTR(s_1,1,4) ≥ SUBSTR(s_2,1,4)

If we let $-\infty$ denote the smallest possible value for a string, then the START functions are as follows:

Operator	START Function Name	Definition
@=	F(s_1)	SUBSTR(s_1,1,4)
@<	G(s_1)	$-\infty$ (constant)
@≤	G(s_1)	$-\infty$ (constant)
@≥	F(s_1)	SUBSTR(s_1,1,4)
@>	F(s_1)	SUBSTR(s_1,1,4)

If the standard string comparison operators are assigned the ordering operator class name STRINGCOMP, then the compatible operators defined above would be specified as

```
@=  COMPATIBLE WITH STRINGCOMP START F EQUAL;
@<  COMPATIBLE WITH STRINGCOMP START G EQUAL;
@≤  COMPATIBLE WITH STRINGCOMP START G EQUAL;
@≥  COMPATIBLE WITH STRINGCOMP START F EQUAL;
@>  COMPATIBLE WITH STRINGCOMP START F GREATER;
```

As a second example, consider ADTs consisting of two component integer vectors (written $[x, y]$), with the ordering operator class given by

$[a, b] \leq [c, d]$ if $a < c$ or if $a = c$ and $b \leq d$.

The class of operators comparing second components of vectors when the first components are equal is compatible with this ordering class. These operators include

$[a, b]$ @< $[c, d]$ if $a = c$ and $b < d$
$[a, b]$ @> $[c, d]$ if $a = c$ and $b > d$

The scan start point determination functions would be

$$F1([a,b]) = [a,-\infty] \quad \text{for } @<$$
$$F2([a,b]) = [a,b] \quad \text{for } @> \quad .$$

And, if the ordering operator class is called VECTORD, the compatibility definitions would be

```
@< COMPATIBLE WITH VECTORD START F1 EQUAL;
@> COMPATIBLE WITH VECTORD START F2 GREATER;
```

Compatible operators are also useful in textual applications where they can index operators used in word adjacency searching (e.g., "find books with the words 'american' and 'history' in the title separated by not more than two other words." See [Lynch 1987] for details).

8. Conclusions

The extensions to indexing for user-defined operators given in this paper enable a large class of user-defined operators to be supported by indices. These extensions are essential for the efficient support of large text-oriented databases by RDBMSs and additionally offer great space savings for textual databases. The extensions seem to fit well with proposals for extended relational systems that allow set-valued columns. When complemented by proper query optimization methods [Lynch 1987], they also provide substantial gains in query execution performance. The compatible operator construction should be particularly useful in applications involving vector ADTs. In general, we believe that these constructions should be useful in a wide range of applications outside the realm of textual databases.

References

[CCA 1986] Computer Corporation of America. *Model 204 System Overview* (Cambridge, MA: Computer Corporation of America, 1986).

[Carey & DeWitt 1985] Carey, Michael J. and DeWitt, David J. "Extensible Database Systems," in *Proceedings, 1st International Workshop on Expert Data Bases*, Kiowa, SC, October 1984.

[Carey et al. 1986a] Carey, Michael J.; DeWitt, David J.; Richardson, Joel E.; and Shekita, Eugene J. "Object and File Management in the EXODUS Extensible Database Management System," in *Proceedings, 12th International Conference On Very Large Databases*, Kyoto, Japan, August 1986, pp. 91–100.

[Carey et al. 1986b] Carey, Michael J.; DeWitt, David J.; Frank, Daniel; Goetz, Graefe; Richardson, Joel E.; Shekita, Eugene J.; and Muralikrishna, M. "The Architecture of the EXODUS Extensible DBMS," in *Proceedings, 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Crawford 1981] Crawford, Robert G. "The Relational Model in Information Retrieval," *Journal of the American Society for Information Science* 32 (1981), pp. 51–64.

[Dadam et al. 1987] Dadam, P.; Duespert, K.; Anderson, F.; Blanken, H.; Erbe, R.; Guenauer, J.; Lum, V.; Pistor, P.; and Walch, G. "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies," in *Proceedings, SIGMOD '86*, pp. 356–364.

[DLA 1987] Division of Library Automation. *MELVYL Online Catalog Reference Manual* (Berkeley, CA: Division of Library Automation, University of California, 1987).

[Lindsay et al. 1987] Lindsay, Bruce; McPherson, John; and Pirahesh, Hamid. "A Data Management Extension Architecture," in *Proceedings, SIGMOD '87*, San Francisco, CA, May 1987, pp. 220–226.

[Lynch 1987] Lynch, Clifford A. *Extending Relational Database Management Systems for Information Retrieval Applications*, Ph.D. Thesis (Berkeley, CA: Department of Electrical Engineering and Computer Sciences, University of California, Berkeley 1987).

[Macleod 1979] Macleod, Ian A. "SEQUEL as a Language for Document Retrieval," *Journal of the American Society for Information Science* **30** (1975), pp. 243–247.

[Macleod & Crawford 1983] Macleod, Ian A. and Crawford, Robert G. "Document Retrieval as a Database Application," *Information Technology: Research and Development* **2** (1983), pp. 43–60.

[Ong et al. 1984] Ong, J.; Fogg, D.; and Stonebraker, M. "Implementation of Data Abstraction in the Relational Database System INGRES," *SIGMOD Record* **14:1** (March 1984), pp. 1–14.

[Osborn & Heaven 1986] Osborn, Sylvia L. and Heaven, T.E. "The Design of a Relational Database System with Abstract Datatypes for Domains," *ACM Transactions on Database Management* **11:3** (September 1986), pp. 357–373.

[Schek 1981] Schek, H-J. "Methods for the Administration of Textual Data in Database Systems," *Information Retrieval Research*, Oddy, R.N.; Robertson, S.E.; Van Rijsbergen, C.J.; and Williams, P.W. (eds.) (London, England: Butterworths, 1981), pp. 218–235.

[Schwarz et al. 1986] Schwarz, P.; Chang, W.; Freytag, J.C.; Lohman, G.; McPherson, J.; Mohan, C.; and Pirahesh, H. "Extensibility in the STARBURST Database System," in *Proceedings, 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986, pp. 85-92.

[Selinger et al. 1979] Selinger, Patricia Griffiths; Astrahan, M.M.; Chamberlin, D.D.; Lorie, R.A.; and Price, T.G. "Access Path Selection in a Relational Database Management System," in *Proceedings, SIGMOD '79* (1979), pp. 23–34.

[Software AG 1982] Software AG of North America, Inc. *ADABAS Introduction Manual* (Reston, VA: Software AG of North America, Inc., October 1982).

[Stonebraker 1986] Stonebraker, Michael. "The Inclusion of New Types in Relational Data Base Systems," in *Proceedings, 2nd International Conference on Data Base Engineering*, Los Angeles, CA, February 1986.

[Macleod 1979] Macleod, Ian A. "SEQUEL as a Language for Document Retrieval," *Journal of the American Society for Information Science* **30** (1975), pp. 243–247.

[Macleod & Crawford 1983] Macleod, Ian A. and Crawford, Robert G. "Document Retrieval as a Database Application," *Information Technology: Research and Development* **2** (1983), pp. 43–60.

[Ong et al. 1984] Ong, J.; Fogg, D.; and Stonebraker, M. "Implementation of Data Abstraction in the Relational Database System INGRES," *SIGMOD Record* **14:1** (March 1984), pp. 1–14.

[Osborn & Heaven 1986] Osborn, Sylvia L. and Heaven, T.E. "The Design of a Relational Database System with Abstract Datatypes for Domains," *ACM Transactions on Database Management* **11:3** (September 1986), pp. 357–373.

[Schek 1981] Schek, H-J. "Methods for the Administration of Textual Data in Database Systems," *Information Retrieval Research*, Oddy, R.N.; Robertson, S.E.; Van Rijsbergen, C.J.; and Williams, P.W. (eds.) (London, England: Butterworths, 1981), pp. 218–235.

[Schwarz et al. 1986] Schwarz, P.; Chang, W.; Freytag, J.C.; Lohman, G.; McPherson, J.; Mohan, C.; and Pirahesh, H. "Extensibility in the STARBURST Database System," in *Proceedings, 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986, pp. 85–92.

[Selinger et al. 1979] Selinger, Patricia Griffiths; Astrahan, M.M.; Chamberlin, D.D.; Lorie, R.A.; and Price, T.G. "Access Path Selection in a Relational Database Management System," in *Proceedings, SIGMOD '79* (1979), pp. 23–34.

[Software AG 1982] Software AG of North America, Inc. *ADABAS Introduction Manual* (Reston, VA: Software AG of North America, Inc., October 1982).

[Stonebraker 1986] Stonebraker, Michael. "The Inclusion of New Types in Relational Data Base Systems," in *Proceedings, 2nd International Conference on Data Base Engineering*, Los Angeles, CA, February 1986.

[Stonebraker et al. 1976] Stonebraker, Michael; Kreps, Peter; Wong, Eugene; and Held, Gerald. "The Design and Implementation of INGRES," *ACM Transactions on Database Systems*, **1:3** (September 1976), pp 189–222. Also in *The INGRES Papers: Anatomy of a Relational Database System*, Stonebraker, Michael (ed.) (Reading, MA: Addison-Wesley Publishing Co., 1986), pp. 5–45.

[Stonebraker et al. 1983a] Stonebraker, Michael; Rubenstein, Brad; and Guttman, Antonin. "Application of Abstract Data Types and Abstract Indices to CAD Data Bases," in *Proceedings, Engineering Applications Stream of 1983 Data Base Week*, San Jose, CA, May 1983. Also in *The INGRES Papers: Anatomy of a Relational Database System*, Stonebraker, Michael (ed.) (Reading, MA: Addison-Wesley Publishing Co., 1986), pp. 317–333.

[Stonebraker et al. 1983b] Stonebraker, Michael; Stettner, Heidi; Kalash, Joseph; Guttman, Antonin; and Lynn, Nadene. "Document Processing in a Relational Database," *ACM Transactions on Office Information Systems* **1:2** (April 1983). Also in *The INGRES Papers: Anatomy of a Relational Database System*, Stonebraker, Michael (ed.) (Reading, MA: Addison-Wesley Publishing Co., 1986), pp. 357–375.

[Stonebraker & Rowe 1985] Stonebraker, Michael and Rowe, Lawrence A. "The Design of POSTGRES," in *Proceedings, SIGMOD '86*, Washington, DC, May 1986, pp. 340–355.

[Zaniolo 1983] Zaniolo, Carlo. "The Database Language GEM," in *Proceedings, SIGMOD '83*, pp. 207–218.