Implementation of Extended Indexes in POSTGRES

Paul M. Aoki[1]

Computer Science Division, Department of EECS
University of California
Berkeley, CA 94720-0001

Abstract

The vaunted ‘‘Spartan simplicity’’ of the relational model can limit the usefulness of the relational database management system (RDBMS) for non-traditional applications. For example, there is no natural way to model a keyword index for bibliographic informational retrieval (IR), or any other index whose key values are *computed* from column values, in a standard RDBMS. The extended indexing proposed in [LYNC88a] is intended to support applications that require indexes on computed values. This paper reports on an implementation of this type of indexing using the POSTGRES extensible database management system [STON86a], focusing on two issues: general problems, and the features in POSTGRES that proved helpful in the solution of these problems.

1. Introduction

Standard RDBMS technology does not adequately meet the needs of certain non-traditional applications. Consequently, some new approaches have been proposed to increase DBMS expressive power and efficiency in these cases. One such approach, the extensible database management system (exemplified by EXODUS [CARE86] and POSTGRES [STON86a]), allows the user to define new operators, types and access methods without significant modification of the underlying database system. However, some extensions do not fit precisely into the extensibility models provided by these systems since they involve changes more complex or more fundamental than the addition of a new access method.

This paper is a case study of the implementation of one such extension, extended indexing, in POSTGRES. Section 2 describes extended indexing as it was originally proposed in [LYNC88a], including a discussion of its advantages over other solutions and some implementation difficulties that it presents. Section 3 gives an overview of the extensibility features of POSTGRES. Section 4 provides details of an implementation of this type of indexing under POSTGRES, such as the modifications made to the original proposal and extensions made to POSTGRES beyond those normally supported by the system. Finally, Section 5 summarizes the experience gained from the implementation.

2. Relational Systems for Information Retrieval

Information retrieval and database management systems[2], as divergent as they often are as research areas, still share a great deal at the practitioner’s level. IR applications, such as online library catalogs, are frequently written on top of commercial DBMS software.

There are two common choices of platforms on which to build production IR systems — inverted-file systems such as ADABAS [SOFT82] are commonly used for textual database applications, whereas relational systems are less often used. Inverted-file and relational systems differ in the *data model* presented to the user/programmer. Inverted-file systems store collections of records in an ordered data structure that is visible to the user. To benefit from the ordered structure (i.e., speed up database queries), the user must generate code or queries that make specific use of its properties. Relational systems present collections of records as tables (*relations*) in which records form the rows and record fields form the columns. Records are accessed using queries that are independent of the physical storage or access method actually used.

Relational systems have advantages over inverted-file systems. The *data independence* described above is one of the most commonly-cited advantages of relational systems. Users need not (and often cannot) formulate their queries with the aim of optimizing the use of the underlying physical storage structure; instead, the DBMS attempts to hide the storage structure and perform all query optimization. This reduces programmer effort greatly, since (1) the computer (rather than the programmer) searches for the best method of performing the query, and (2) the application itself need not contain

[2] Readers unfamiliar with DBMS concepts, particularly those of relational systems, should consult one of the standard introductions to the subject (e.g., [DATE90]). More experienced readers may find [STON88] useful, as it provides a selection of papers that tend to focus on the kind of implementation issues to which we allude in this paper.

[1] Author’s present address: USS *Merrill* (DD 976), FPO San Francisco, CA 96672-1214.

assumptions about the physical storage of the data and so need not be rewritten when the storage method is changed. Relational systems are typically implemented on top of what closely resembles (and sometimes is) an inverted-file system, with user interface and query optimization modules layered over the base query processing and data storage/retrieval module. Consequently, there is no reason why a RDBMS with a well-tuned query optimizer should perform significantly worse than an inverted-file system. If this last point makes it appear as if a relational interface is useless overhead, note that the user interface associated with information retrieval systems often resemble relational query languages in their use of logic predicates and that the code needed to turn these queries into inverted-file access calls is essentially a specialized query optimizer.

In DBMS terminology, an *index* is a physical storage structure used to speed access to the data records. For example, one might extract the values of a particular field from each record in a table (i.e., the contents of an entire column in a relation) and store them in a lookup table along with *pointers* that describe the location of the record from which any particular value came. Searching the lookup table (which is smaller than the base table and arranged for quick access to a given field value) and following the pointers is generally faster than performing an item-by-item search of the base table.

The arrangement of the field values in the index is critical and amounts to a precomputed partition of the data records. One may think of B-trees [BAYE72] as a precomputed record ordering, hash indexes (e.g., linear hashing indexes [LITW80]) as precomputed record hash-partitioning, and so on. In most systems, this precomputed information is based on partitioning information and operator/access method relationships that are hardwired into the query processor. In addition, the precomputed information can only be taken directly from the column values. By this we mean that one can build an index over the column ''emp.salary'' but not over values *computed from* the column, such as ''taxable_income(emp.salary)''. This limits the usefulness of indexes to certain applications.

2.1. Extended Indexing

Extended indexing as defined in [STON86b] is one approach by which a user can add new index access methods to a DBMS. It is particularly helpful when used with a system in which new data types and operators can be defined and used with the new access methods (existing access methods usually suffice for the basic data types, such as integers). When a new access method is defined, the access method and the operators usable with it must be associated with an ordering/partitioning *class*. The class information is used by the query optimizer to match the operators used in the query with access methods that can be used for efficient access to a column. Take the example of an abstract data type (ADT), BOX, that corresponds to two-dimensional boxes. One might construct a set of binary Boolean operators that compare boxes by their areas in a manner analogous to the integer comparison operators **< <= = => >**. Since the box-area comparison operators define an ordering on BOX columns, the box-area operators could then be associated with a new ''box-area-operations'' class. Finally, by associating the B-tree access method with the ''box-area-operations'' class, B-trees could then be used as secondary indexes over BOX columns. That is, the query optimizer can consider a B-tree index when it sees a query that uses the ''box-area-operations'' operators because of the class association between the index and operators. Of course, the B-tree code must be flexible enough to handle arbitrary comparison functions and data types.

The point here is that all of the class information is stored in system catalogs (relations reserved for internal use) rather than being hardwired. That is, the names and properties of the different types, operators and access methods are placed in relations rather than in the DBMS code. Because new information can be added easily to the catalogs, this approach not only allows the user to define and use new types, operators and access methods, it can allow their definition and use on the fly (i.e., without recompilation of the system). As will be seen, POSTGRES permits this.

As defined above, extended indexing still limits the user to access methods and operators that order or partition tuples based only the index key values. For example, whenever the ''box-area-operations'' operators are used, the areas of the BOX operands are recomputed with every comparison because the box area is not stored anywhere. It would be better to compute the area of a given box only once, in effect caching the values for later queries to use. Extended indexing as defined in [LYNC88a] does this, allowing indexing on *functions* of column values as well as on the column values themselves.

For an example where this is useful, consider the BOX scenario described above. If an index exists over the (computed) areas of each BOX in a column, the query ''find the BOXes with area less than 5'' can be executed as a scan of a small part of the box area index instead of a sequential scan of the entire relation.

As a more realistic example, consider the problem of bibliographic information retrieval using a RDBMS. Bibliographic searches typically involve predicates on keywords which are extracted from titles, abstracts or body text. There are many ways in which these searches can be implemented, and [LYNC88a] surveys many of the alternatives (e.g., pattern-matching operators, set-valued columns, and nested relations). We describe three methods (chosen more for their illustrative value than their practical value) below, comparing their strengths and weaknesses.

Consider a relation *reports* that includes a unique report-identifier field, *id*, as well as a text field, *abstract*, from which subject keywords are extracted. The user of a vanilla RDBMS can extract keywords from the base relation and store each of them in individual tuples of a second base relation, *keywords*, thereby creating an ad hoc keyword ''index'' that can be accessed with queries

reports	id	title	abstract	text
	1	Implementation..	The vaunted..	Standard..
	2	Extended..	A number..	Information..

keywords	id	keywords
	1	database
	1	POSTGRES
	1	bibliographic
	2	bibliographic
	2	relational

```
select  title
from    reports, keywords
where   reports.id = keywords.id and keywords.word = "database"
```

Figure 1. A solution to the keywords problem in a standard RDBMS.

```
select  title
from    reports
where   reports.abstract ?? "database"
```

Figure 2. A solution involving user-defined operators.

```
create index   keywords
on             report
ordering       keyword-ops
operator       extract

select  title
from    reports
where   extract(reports.abstract) = "database"
```

Figure 3. A solution using the syntax from [LYNC88a].

such as that shown in Figure 1. A number of efficient access paths can be generated by a query optimizer for keyword-based retrievals by indexing the *reports* and *keywords* relations on the *id* column and indexing the *keywords* relation on the *word* column. Although the keyword computation cost is only paid once, as desired, this approach has several obvious problems. First, the query is clumsy. Second, joining the two *id* columns and index-scanning the inner relation is less efficient than using a true keyword index. More complex queries compound this problem, since n+1 joins are required to evaluate the conjunction of n keywords in a SQL query. Finally, the ‘‘index’’ relation *keywords* is not updated when the base relation is updated because it is part of a user schema rather than a true index.

The query clumsiness problem is easy to solve. If the RDBMS allows user-defined functions and operators, the user can construct keyword extraction and containment functions and then run them over the base relation. In this case, we can eliminate the pseudo-index *keywords* from the schema described in the previous example. The example shown in Figure 2 assumes that the operator ‘‘??’’ has been defined as keyword extraction and containment — i.e., extract the keywords from the left-hand (text) argument and check to see if the right-hand (keyword) argument is one of them. The query in Figure 2 is certainly simpler than the non-intuitive join shown in Figure 1. Furthermore, there is no problem with update inconsistencies. However, in practice, keyword-extraction algorithms are fairly complicated and tend to be executed on large text objects. Recomputing the keyword extraction function for every tuple in every query would be prohibitively expensive and wasteful in an on-line library catalog system on the scale of the University of California’s MELVYL [DLA87], which contains over 13,000,000 records.

Extended indexing combines the best features of both approaches, since simple queries can be used for efficient retrieval and an update to the base relation results in correctly-updated indexes. In this approach, a secondary index is constructed from the return values of a function applied to each of the column values and then accessed by queries which appear *to the user* to be on the results of the function. The functional results may be either a system type (e.g., ‘‘Keyword’’ in the case we are describing), or a *list* of items of a system type (e.g., ‘‘KeywordList’’ would be the appropriate return-type of the keyword extraction function since any given text item may contain multiple keywords). Sample queries for creating a functional index and using it are shown in Figure 3. The ‘‘=’’ operator is keyword equality and the ‘‘extract’’ function is just the keyword extraction function. The equality operator, which compares Keywords to Keywords, is a member of the operator class ‘‘keyword-ops’’ that is used to order the tuples of the *keywords* index.

3. Extensibility in POSTGRES

The POSTGRES database management system [KEMN91, STON86a] allows the user to extend the system in many different ways. The user may easily implement and insert into the system new abstract data types, functions, operators, and access methods. For example, one can define a BOX type usable in the columns of any base relation, a ‘‘box_equality’’ function, a box-equality operator ‘‘=’’ that calls the function ‘‘box_equality’’, and a R-tree access method [GUTT84] for efficient access to tuples containing BOXes. Operators and access methods are assigned to classes when they are defined and are matched together at runtime by the table-driven query optimizer and query processor. Operator names can be overloaded, argument and return-value types being used in the parser to determine which function to apply (e.g., resolve whether the operator ‘‘<’’ means ‘‘integer_equality’’ or ‘‘box_equality’’).

These extensibility features have obvious applications to information retrieval research. The user-defined access methods, operators and functions and the table-driven query optimizer and processor give POSTGRES potential as a testbed for information retrieval research, especially for work aimed at linking IR and relational systems (e.g., [LYNC87, LYNC88a, LYNC88b]). The flexibility gained by being able to experiment with new storage and retrieval methods without changing the application interface (both code and user queries) is a powerful argument for choosing a relational system over one that exposes its internal structure to the user and programmer. The user-defined ADTs, operators and functions are ideally suited for supporting multimedia information retrieval research. For example, POSTGRES applications have been storing and displaying large text objects (news articles) and images (X Window System bitmaps) since 1987.

Unlike users of other extensible systems (e.g., EXODUS [CARE86]), the POSTGRES user need not stop the system and recompile it every time new code is added. In POSTGRES, the query optimizer and query processor are table-driven and all user-defined extensions used in a query may be loaded into a POSTGRES server process at runtime. That is, the types and functions need only be defined in the system catalogs before use; the user does not have to change the POSTGRES executable. Dynamically-loaded routines incur a small one-time link-and-load cost per server process when they are first invoked. The ability to add extensions by simply checking them into the catalogs has several obvious advantages:

- Binary images only include the extensions that the user has actually used.
- The user can do this alone, without the intervention of a database implementor (DBI) who manages the construction of server executables.
- The debugging turnaround time is potentially much lower. Rather than a

  think —> recompile extension —> rebuild server —> run

  debugging cycle, there is a

  think —> recompile extension —> run

  cycle. This is advantageous if rebuilding the server is a lengthy process[3].

Note that POSTGRES has been designed so that a DBI can easily incorporate new code (such as a new access method) directly into the DBMS should this prove to be desirable. Both dynamically-loaded and statically-loaded code must be registered in the catalogs in order for the parser, query optimizer and query processor to use them, so whether or not a piece of object code is compiled-in or not is invisible to the user. In general, then, the POSTGRES approach gains flexibility and ease of debugging during development (due to the dynamic loader) while retaining the ability to compile production code into the server.

In upcoming sections, one should note how the two major features — table-driven extensibility and dynamic loading — affected the implementation. In general, the fact that POSTGRES already supports the form of extended indexing described in [STON86b] made matters much easier.

4. The Implementation

The prototype implementation of extended indexing in POSTGRES consisted of three stages:

1. type-function-operator definition
2. access method implementation
3. modification of POSTGRES internals

The implementation reported here was performed on the ‘‘1.0 Beta’’ release of POSTGRES, written in C and Franz LISP Opus 43.[4] However, the ideas presented here do not rely on any particular internal features of that release.

There are two points worth noting immediately. First, the interface described for extended indexing is a general mechanism and is not restricted to the particular access method and datatypes used as the example. Second, while the primary goal was to make things work, another (only slightly less important) goal was to minimize the number of changes required to the base POSTGRES system. Both goals took precedence over making things ‘‘pretty.’’ The effects of this will be seen below.

---

[3] For example, the 1987 POSTGRES server prototype took an hour to link and load, not counting the time required to generate the individual object-code modules. This is no longer true, as the code has been completely rewritten and reorganized since, but it represents an extreme case.

[4] Franz LISP is a trademark of Franz, Inc.

4.1. Type/Function/Operator Definition

The initial implementation of the keyword data type and functions was trivial. Because this code was to be used to debug the access method code and modifications to the base POSTGRES system, it was kept as simple as possible. However, there were still a few non-obvious aspects to the implementation. First, the Keyword and KeywordList types were defined with the same internal representation (i.e., an object of type ‘‘Keyword’’ was stored as a single-element list). The motivation for this will be explained below. Second, the keyword extraction function was defined to always return a list. This is a simplification of the proposal for extended indexing in [LYNC88a], which permits the extraction function to return no elements (null), single elements, or lists of two or more elements, thereby forcing all functions which operate on the result of an extraction function to deal with all three types. Simply having the extraction function return a list of zero or more elements appears to be semantically equivalent and involves less work for implementors.

4.2. Access Method Implementation

Because the B-tree provides all of the functionality required for keyword-based extended indexing, a ‘‘functional B-tree’’ (henceforth referred to as ‘‘FB-tree’’) access method can be created from it. The following section demonstrates how this particular access method was implemented in POSTGRES.

We will first deal with the syntactic, or user-interface, issues. Figure 4 shows examples of the syntax used in this implementation for the definition and use of a keyword FB-tree index. The top query in Figure 4 is an example of a POSTQUEL ‘‘define index’’ query that happens to define a keyword FB-tree index; the query includes no extensions to the basic POSTQUEL syntax, which has been constructed to allow this kind of flexibility. The first three lines simply name the index, its base relation, the column over which the index is being constructed, and the operator class associated with the column. The following four lines list some parameters specific to keyword FB-trees. The ‘‘with’’ clause of the ‘‘define index’’ statement, used in POSTQUEL for access-method-specific parameters, here lists the names of a number of functions that the access method needs to know about in order to operate on lists of keywords. The bottom query in Figure 4 shows a sample retrieval query. Like the retrieval query in Figure 3, the ‘‘kw_extract’’ function returns a list of keywords; unlike the query in Figure 3, the ‘‘@=’’ operator corresponds to keyword *containment*.

How are these queries different from that shown in Figure 3, SQL vs. POSTQUEL syntax aside? The main difference in the index definition query is the list of function specifications, ‘‘extract = kw_extract, ...’’. These functions are used by the FB-tree access method to perform certain actions needed to maintain the index, such as form (‘‘listify = ...’’) and tear apart (‘‘element = ...’’) lists of the base type. Similar functions are also needed in [LYNC88a] but are assumed to be built into the system or defined in a separate query; hence, this is simply a syntactic difference, chosen for ease of implementation, and not an inherent part of either POSTGRES or extended indexing. The main difference in the retrieval query is the use of the keyword containment operator, ‘‘@=’’. The original proposal uses a keyword equality operator for the index qualification clause and requires the implementor to modify the parser to understand lists. Since the parser normally relies on type-checking to resolve operator overloading, forcing the parser to resolve overloading in this situation would require additional information or some kind of arbitrary restrictions. For example, the parser would not be able to determine which of two operators to use if (1) they had the same name and (2) one took a KeywordList and a Keyword as arguments and the other took two Keywords. The modified proposal uses a list-to-keyword operator and thus does not present any type inconsistencies to the parser. Furthermore, the original proposal makes comparison of a list of items to a single item equivalent to the logical OR of comparing each element of the list to the single item. Using a different operator means that the query processor does not have to be modified to incorporate this change.

We now turn to the issues involved in coding. The critical insight in the POSTGRES implementation of FB-trees is that they can be implemented by adding a

```
define index  keywords
on            reports
using         fbtree ( abstract keyword-ops )
with          ( extract = kw_extract,      /* keyword extraction */
                count = kwlist_count,      /* list length */
                element = kwlist_element, /* list access */
                listify = kw_listify )     /* keyword -> list */

retrieve      (reports.title)
where         kw_extract(reports.abstract) @= "database"
```

Figure 4. A sample solution in POSTGRES requiring no extensions to POSTQUEL.

layer between the table-driven query processor and the generic B-tree access method. The system catalogs, and thus the table-driven query processor, see the FB-tree index as an index on the column values themselves that is only usable when one argument to a specific set of operators is a specific function (e.g., ''kw_extract''). The FB-tree access method does all of the translation necessary for this to happen, which is why so little POSTGRES code has to be modified. When an update or insertion occurs, the query processor calls the access method insertion routine with the new index tuple as an argument. The FB-tree insertion routine constructs a number of single-keyword index tuples from the index tuple it receives from the query processor and inserts them into a standard B-tree. Retrievals with keyword keys correspond to simple B-tree scans. The functions needed to break down and reconstruct the various index tuples are found through lookups in the system catalogs as described below. In short, all of the magic is hidden from both the query processor and B-tree code by this in-between code, so neither the query processor nor the base access method need to be changed at all.

Consider the following specific examples, again using our keyword example. The most complicated operation is tuple insertion (update). For each heap tuple inserted into the base relation, the query processor forms a B-tree index tuple that includes the text field from the heap tuple (i.e., the abstract text). The FB-tree code takes the index tuple, pulls out the field being indexed (the abstract), looks up the extraction function in the system catalogs, and then calls the extraction function with the indexed field (abstract) as its argument. The extraction function returns a list of keywords contained in the abstract; the FB-tree code forms individual B-tree tuples for each of the keywords, and these keyword tuples are inserted into the index relation. The query processor thinks it has inserted a B-tree tuple containing the abstract text, but what ends up in the index relation are multiple B-tree tuples containing the keywords. Index definition is simply a matter of running this insertion procedure over all of the tuples currently in the heap relation. Because the insertion process does all of the precomputation (in this case, precomputation of the keyword extraction function), retrieval queries are now completely trivial. A probe of the functional index with a given search key (keyword) is simply a B-tree probe with that search key.

4.3. Modifications of POSTGRES Internals

POSTGRES provides a great deal of support for user extensions, but this kind of indexing requires modification to the query optimizer and catalog information. No substantial changes were made, but in some places identifying what to change and how to change it required more than casual knowledge of the system's implementation.

4.3.1. System Catalog Modifications

Three slight changes were made to the catalogs as defined in the base system.

First, two fields in the `INDEX` catalog relation, which normally contains information on the schema of each index relation, were modified to contain information relating to the extraction function.

Second, the class and type inconsistencies mentioned in the previous section had to be resolved. A relatively clean solution is to use a single representation for a given type and list-of-type so that the same functions can operate on them. That is, a Keyword is simply a singleton KeywordList. Thus, the same operator can be used in sequential scans (where the keyword containment function is called upon to compare a KeywordList and a Keyword) and also index scans (where the same function is called on two Keywords). For this to work, the ''@='' operator must be a member of an operator class that orders index elements (Keyword/Keyword comparisons) and yet work as a restriction operator in both types of queries (KeywordList/Keyword comparisons for sequential scans and Keyword/Keyword comparisons for index scans). Hence, the operator argument type declarations in the `OPERATOR` catalog relation will be inconsistent with the actual application of the operator, but no *code* needs to be changed to deal with this inconsistency, unlike [LYNC88a], where the parser and query processor must be modified to understand the special list-to-singleton comparison semantics.

Third, a new ''system'' relation, `FUNCINDEX`, was added to store the lists of functions that defined the properties of the functional index. This relation is keyed on the index relation object ID and contains the object IDs of the functions listed in the index definition. In principle this information could simply be part of the `INDEX` relation, but it is not needed for most indexes. In addition, if so implemented, a change to the existing POSTGRES system catalogs would be required.

4.3.2. Other Modifications

The changes to the POSTGRES query optimizer [FONG86] were minimal. All that was added was the ability to recognize function clauses and consider the appropriate index scans. This meant adding routines to find qualification clauses of the appropriate form, perform `FUNCINDEX` catalog lookups, and change the format of the affected index scan qualifications. The net difference was about 40 lines of code.

Because the FB-tree implementation is invisible to the query processor, no changes to the query processor were required to support them.

5. Conclusions

There are some obvious, if cosmetic, areas for continued work: the code requires a tremendous amount of cleaning-up, a nicer syntax could be devised for the index definition statement, a large database could be loaded into POSTGRES and the relative performance of extended user-defined access methods with varying query optimizer options and queries could be evaluated, and so on.

The following key points can be picked out from the experience described above:

- The extended indexing defined in [LYNC88a] has been implemented.

  The ability to produce indexes over functions of column values has use beyond bibliographic information retrieval. Applications might include image processing, computer vision, geographic information systems, or any other field where queries commonly require the repeated computation of expensive functions.

- A few modifications to the original proposal in [LYNC88a] greatly simplified this implementation without loss of expressive power.

  Implementors should be prepared to make simplifying changes, such as requiring extraction functions to return lists and having comparison functions operate on both lists and singletons. In some cases, there aren't any desirable implementation choices, and the implementor is left to choose one kludge or another. For example, in this implementation, the system type (Keyword) was given the data representation of a singleton list (KeywordList) in preference over performing arbitrary modifications to the parser and query language.

- The extensibility features in POSTGRES proved to be useful and flexible in the course of this implementation.

  This implementation effort did not call for access method debugging, since the underlying B-tree code was already debugged. Nevertheless, dynamic loading proved very useful for the access method/ADT integration. Initially, dynamic loading was convenient because the time to recompile and load the access method and ADT code was insignificant compared to the time to recompile the server. On the other hand, once debugging became a matter of finding logic bugs instead of core-dumping problems and the server had to be run several times to track down a particular error, having the access method compiled-in proved to be more useful since dynamically-loaded routines cannot be easily traced (they are not added to the executable's symbol table, which is used by the debugger). Hence, both dynamic and static loading were useful at different times.

It is a tribute to the system design that an extension that was totally unforeseen at project inception (indexing over the results of functions of column values) could be incorporated with trivial changes to the base system code and without changes to the query language. A large factor in this is, of course, the table-driven query processor architecture. For example, because the access method internals are hidden from the query processor and vice versa, a simple piece of ''glue'' code sufficed to implement the FB-tree interface on top of the B-tree code.

The ease with which the code was incorporated into the base POSTGRES system can be summarized by the fact that the actual implementation (coding) phase took two weekends. The leverage that extensible database management systems provide in rapid prototyping of major changes, even changes that change the assumptions under which the system was originally built, cannot be overemphasized. This leverage can be of particular use to information retrieval researchers looking for a testbed for their innovations.

References

[BAYE72] R. Bayer and E. McCreight, ''Organization and Maintenance of Large Ordered Indexes'', *Acta Inf. 1*, 3 (1972).

[CARE86] M. J. Carey, D. Frank, M. Muralkrishna, D. J. DeWitt, G. Graefe, J. E. Richardson and E. J. Shekita, ''The Architecture of the EXODUS Extensible DBMS'', *Proc. 1986 Int. Wksp. on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

[DLA87] DLA, *MELVYL Online Catalog Reference Manual*, Division of Library Automation, Univ. of California, Oakland, CA, 1987.

[DATE90] C. J. Date, *An Introduction to Database Systems, Volume I (5th Ed.)*, Addison Wesley, Reading, MA, 1990.

[FONG86] Z. Fong, ''The Design and Implementation of the POSTGRES Query Optimizer'', M.S. Report, Univ. of California, Berkeley, CA, Aug. 1986.

[GUTT84] A. Guttman, ''R-Trees: A Dynamic Index Structure for Spatial Searching'', *Proc. 1984 ACM-SIGMOD Conf. on Management of Data*, Boston, MA, June 1984.

[KEMN91] G. Kemnitz, editor. ''The POSTGRES Reference Manual, Version 2.1'', UCB/ERL Tech. Rep. M91/10, Univ. of California, Berkeley, CA, Feb. 1991.

[LITW80] W. Litwin, ''Linear Hashing: A New Tool for File and Table Addressing'', *Proc. 6th VLDB Conf.*, Montreal, Canada, Sep. 1980.

[LYNC87] C. A. Lynch, *Extending Relational Database Management Systems for Information Retrieval Applications*, Ph.D. Thesis, Univ. of California, Berkeley, CA, May 1987.

[LYNC88a] C. A. Lynch and M. Stonebraker, ''Extended User-Defined Indexing with Application to Textual Databases'', *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[LYNC88b] C. A. Lynch, ''Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values'', *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[SOFT82] Software AG, *ADABAS Introduction Manual*, Software AG of North America, Reston, VA, 1982.

[STON86a] M. R. Stonebraker and L. A. Rowe, ''The Design of POSTGRES'', *Proc. 1986 ACM-SIGMOD Conf. on Management of Data*, Washington, DC, June 1986.

[STON86b] M. R. Stonebraker, ''Inclusion of New Types in Relational Data Base Systems'', *Proc. 2nd IEEE Data Engineering Conf.*, Los Angeles, CA, Feb. 1986.

[STON88] M. Stonebraker, ed., *Readings in Database Systems*, Morgan Kauffman, San Mateo, CA, 1988.