Recycling Secondary Index Structures*

Paul M. Aoki

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720-1776
aoki@CS.Berkeley.EDU

Abstract

Many database reorganization techniques move tuples in a table from one location to another in a single pass. For example, distributed database systems move or copy tables between sites to optimize data placement. However, such systems typically drop and then rebuild the secondary indices defined over the table being moved. There are two primary reasons for this. First, moving a table invalidates any physical tuple pointers contained in its secondary indices (e.g., in the leaves of a tree). Second, changes in tuple or page size can cause index tuples on the remote site to be repacked onto pages in a way that degrades the clustering imposed by the structure (e.g., in the upper levels of an R-tree). The cost of rebuilding secondary indices is largely why table movement has been considered a expensive operation. This, in turn, means that data layout optimization has been considered expensive as well. In this paper, we present a simple, efficient mechanism for translating index pointers as well as an approach to preserving internal index clustering. By exploiting the structure of the original index, we can recycle its important properties and produce a usable index on the remote site without the expense of building one from scratch. We also demonstrate the effectiveness of these mechanisms using performance measurements of an implementation in the Mariposa distributed data manager.

1. Introduction

A variety of database reorganization techniques move tuples in a table from one location to another in a single pass. Examples of such techniques include movement of tables between sites in a distributed database and tablespace defragmentation in a single-site database. Unfortunately, reorganization comes at a cost. Because secondary indices usually contain (completely or par-

* This research was sponsored in part by the Army Research Office under contract DAAH04-94-G-0223, the Advanced Research Projects Agency under contract DABT63-92-C-0007, the National Science Foundation under grant IRI-9107455 and Microsoft Corp.

tially) physical tuple pointers into the underlying table,[1] moving a table invalidates its indices. Rebuilding indices from scratch has two major cost components. First, rebuilding is extremely time-consuming and resource-intensive. Time requirements can be reduced using parallel sorting and bulk-loading algorithms [PEAR91], but resource requirements cannot. Second, proper indexing is critical to good performance and the table is likely to be useless while the table is being reindexed. If we can reduce the cost of reindexing a table, we can afford to reorganize more often and improve the overall performance of the database.

In this paper, we explore the tradeoffs involved in reconstructing index structures when the underlying table is moved. The remainder of this introduction provides a more precise definition of the problem, its applications and our cost and benefit metrics. In the rest of the paper we discuss several implementation options, including some previously suggested in the literature, and present a comparative performance analysis based on an implementation of these options in the Mariposa distributed data manager [STON94].

Our study focuses on the class of reorganization operations in which a base table is copied from a *source table S* to a *target table T*. (In a single-site database, *T* may be on another disk partition; in a distributed database, *T* may be on another machine.) We assume that the copying operation can alter the layout of tuples on pages (e.g., because the page size or tuple size changes, or through defragmentation) but that the order of valid tuples cannot be changed. This kind of copying operation arises in many situations, such as:

- *Architecture interchange*. Computer architectures impose varying restrictions on the size and memory alignment of native data types. These restrictions cause changes in tuple size that may cause pages to overflow or underflow in a way that is entirely dependent on the schema and contents (e.g., in the case of variable-length columns) of the tuples.
- *Reorganization*. Even within a single database, we may wish to copy a table without altering the order of the tuples. Such situations include moving a table to a different disk partition, changing a table's page size, defragmenting pages within a tablespace, and certain types of garbage collection (such as that performed by the POSTGRES vacuum cleaner [STON87]).
- *Media interchange*. Different storage devices may have different page sizes that are visible to the data manager.

Our basic idea for making reindexing more efficient is that it is often cheaper to transmit a modified version of an index than to rebuild the index from scratch. Note that we do not necessarily want to preserve the exact structure of the source index. Instead, we *recycle* the materials

[1] There are a few important exceptions, such as Tandem's NonStop SQL, which use primary key tuple pointers.

(e.g., clustering/ordering, tuple pointers, etc.) in the source index that are computationally expensive.

Index recycling has two main subproblems. First, we must decide how we can best adapt the *structure* of the source index. That is, given an index over S, we wish to know what (sub)set of index pages and other information we should send to the target machine to facilitate the efficient production of an index over T. This adaptation process must, of course, be less costly than rebuilding the index over T and should not significantly degrade index retrieval performance relative to a rebuilt index. Second, we must be able to translate the tuple pointers, or *tuple identifiers* (TIDs), in a source index into valid TIDs in the corresponding target index.

The remainder of the paper is organized as follows. In Section 2, we discuss previous solutions to the problems of reindexing and TID translation. In Section 3, we give our algorithms for these problems. In Section 4, we present the details of our implementation in Mariposa and the results of our experiments. In Section 5, we discuss possible directions for future work. Finally, Section 6 provides our conclusions.

2. Related Work

While one can find a great deal of related research on index construction and the translation of various kinds of pointers, there has been relatively little published on the reconstruction of indices based on other, similar indices. Below, we discuss the literature on recycling index structure and index pointers. In particular, we try to draw out the shortcomings in the existing work that we address here.

2.1. Recycling Index Structure

General techniques for the splitting and merging of index nodes are certainly relevant; these are the tools with which we manipulate existing index structures. Index bulk-load algorithms are also clearly relevant. However, there appears to be little work on reconstruction of indices based on the content and properties of existing indices. Sun *et al.* [SUN94] present an algorithm for "reestablishing" B^+-tree indices in a distributed environment. Their algorithm makes no use of the overall tree structure; instead, it is a variation on the standard sort/build technique for bulk-loading B-trees. The source machine transmits the (sorted) leaf tuples of the source index to the target machine, allowing the target machine to build its index bottom-up without the usual sorting step. This approach obviously applies only to ordered data structures such as the B^+-tree or B-tree.

2.2. Recycling Pointers

The problem of pointer translation has been more extensively studied than index structure translation. Previous work falls into two broad categories: translation of pointers by keeping per-pointer information, and translation of pointers using per-page information. We examine each in turn.

Approaches Using Per-TID Information

One obvious way to map TIDs is to use an array consisting of `old-TID` → `new-TID` entries. In general, such a array will not fit in physical memory; translating the TIDs in a large unclustered index will have poor locality and therefore poor virtual memory behavior.

A more sophisticated set of options is suggested by the fact that the TID translation problem has some similarities to *pointer swizzling*, or translation between object reference formats as stored in secondary and main memory, in object-oriented databases. When main memory pointers are OIDs, the "how" (as opposed to the "when") part of swizzling is known as the *OID mapping* problem. OID mapping mechanisms are generally more complex than arrays and include segmented mapping tables (e.g., ObServer [HORN87]), hash tables (e.g., Itasca [EICK95]) and B-trees (e.g., GemStone [MAIE87]). Mapping data structures that contain `OID` → `address` entries work in the OODBMS environment because the database clusters and caches the mapping structure. However, when moving an index, we know we are processing all pointers contained in the index in a short period of time without locality guarantees. OID mapping techniques will not perform well in this bulk-translation environment.

Since the main problem with simple mapping structures is their size, an obvious alternative is to change the mapping granularity. We now turn to TID translation algorithms that use page-level bookkeeping.

Approaches Using Per-Page Information

There are three main groups of proposed algorithms for translating TIDs while keeping only per-page information. These algorithm classes vary in the amount of information they store and the precision of their TID translations. However, all of them assume the use of physical `{page,offset}` TIDs. First, there have been a number of proposals that exploit (unrealistic) assumptions to provide precise translation of source TIDs to target TIDs. Second, the original Mariposa design [STON93b] provides an algorithm for mapping a source TID to a small range of potential target pages. Finally, Sun *et al.* give an algorithm that maps a source TID into the correct target page.

Under appropriate assumptions, we can map source TIDs directly into target TIDs. Assume for the moment that (1) the system uses byte offset TIDs, (2) tuples cannot change size and cannot be reordered, and (3) source pages map directly to a fixed number of target pages (e.g., using a constant expansion or contraction factor) irrespective of fill factor. In this case, we can use the constant expansion/contraction factor to map source page numbers to target page numbers and then use a simple arithmetic formula to map the source offsets to target offsets. This solution achieves our goal of storing only page-level mapping information, but the assumptions violate the conditions we stated in Section 1. [SUN94] and [STON93b] both contain a number of strawman solutions similar to that just described.

Unlike the above strawman proposal, Mariposa does assume that tuples can change size. Instead of using a constant expansion/contraction factor, the original Mariposa design proposes a simple page number translation table. For example, the table might record the fact that tuples on source page 14 have been placed on target pages 66, 67 and 68. However, because tuple size changes are data-dependent, an arithmetic expression can no longer be used to calculate precise byte offsets within the target pages. Instead, the translation algorithm extracts the key from the index tuple. It then extracts the source page number from the source TID and maps it into the set of one or more eligible target pages. The final target TID is obtained by searching each of the target page(s) for the desired tuple using the index key.

The original Mariposa approach works for any access method but has several important shortcomings. First, if the base table is not clustered on the indexed column(s), searching the base table pages to complete the TID translation will result in many random page faults. Second, this strategy fails if the indexed column is not unique. Finally, the translation table only provides a range of possible target pages on which the tuple might be found, which makes the process of translating the target byte offset much less efficient than if the translation table provided the correct page.

Finally, Sun *et al.* also address the TID translation problem. Like the Mariposa design, they use a page-level translation table. However, their page-mapping solution is superior to the original Mariposa solution because it accurately maps a source TID to the correct target page. The basic idea is to record, for each source page s, the byte offset of the first tuple from s that falls on a given target page t. Recall that tuples in S are copied into T in order. Hence, given the byte offset within a source TID, one can determine the exact t to which the corresponding target TID should point. A more detailed critique of the algorithm may be found in [AOKI95]; the main observation here is the fact that this method still does not allow the computation of the byte offset of the target TID. Like the Mariposa algorithm, this algorithm requires that the database fault in and search t in order to translate the byte offset.

3. Toward More General Index Recycling Algorithms

Having described the techniques proposed in the literature for supporting operations similar to those found in index recycling, we now present our own methods for performing those operations. Again, we structure our discussion in terms of recycling index structure and recycling index pointers.

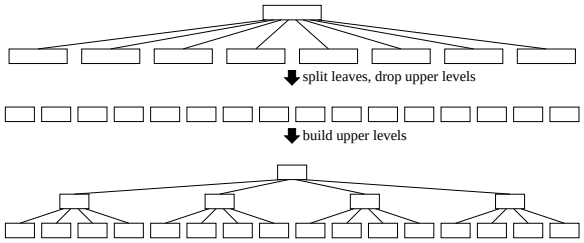
3.1. Recycling Index Structure

In this paper, we discuss algorithms for recycling certain types of hierarchical index structures (i.e., trees). Variations of the algorithms described can be applied to other structures (e.g., hash tables), but we restrict our discussion to those types of index for which we have implementations and benchmarks.

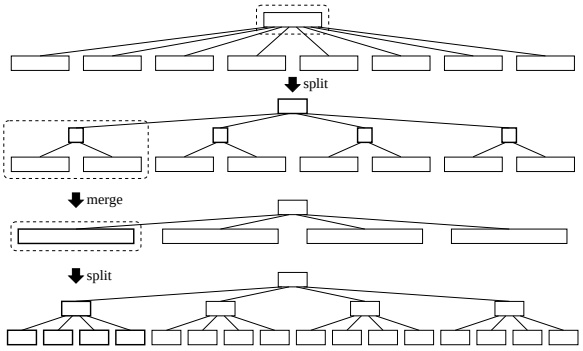
There are two general types of hierarchical index structures: *ordered* (e.g., B^+-trees, Hilbert R-trees [KAME94]) and *unordered* (e.g., R-trees). Generally speaking, a "bottom-up" recycling strategy similar to that described by Sun *et al.* should work for nearly any ordered structure. However, a bottom-up strategy may not necessarily work well for an unordered structure. In this section, we propose a taxonomy of techniques for recycling either ordered or unordered trees.

Figure 1 shows our generic strategies for constructing hierarchical structures. For pictorial clarity, the strategies are depicted as modifying the tree in-place, but it should be understood that changes are actually being reflected in the target index.

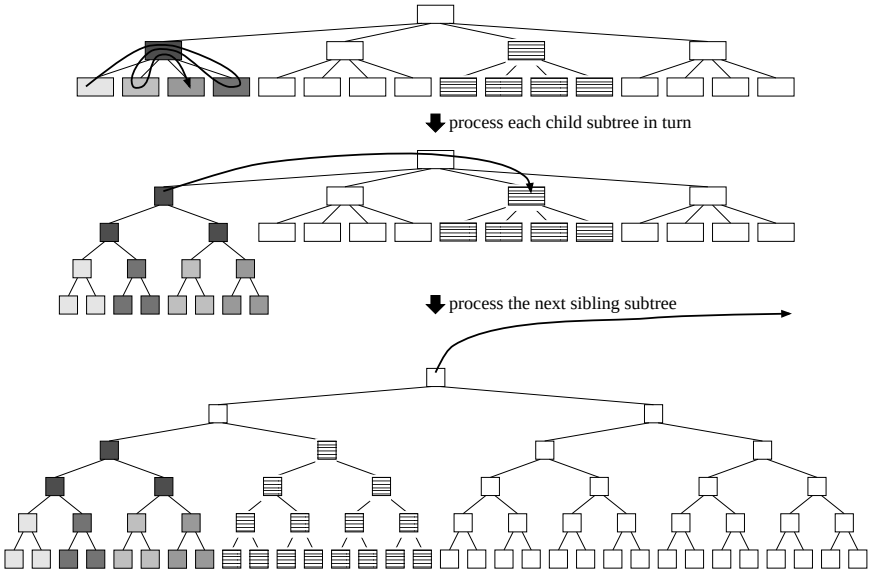
- Ordered structures such as the B^+-tree should use the *bottom-up* strategy shown in Figure 1(a). In this strategy, the tuples from the leaf pages of the source index are simply repacked into leaf pages in the target index and the remainder of the target index built bottom-up. The structure (e.g., fill factor, height) of the upper levels of an ordered tree is independent of the leaf key values, so the concern for such trees should be building a short, well-balanced target tree rather than attempting to reproduce any properties of the source tree.
- To index an unordered structure that supports an abstract node split/merge interface, one could use the *top-down* strategy in Figure 1(b). By recursively merging underfull nodes and splitting overfull nodes, this strategy can achieve a desired fill factor.
- The *serpentine* strategy of Figure 1(c) can be applied to unordered structures in which heuristic orderings can reasonably be applied in a very local context (e.g., within the keys stored in a single node). For example, while traversing an R-tree, we can "order" the bounding box keys contained in each internal node using the same metric by which we would split the page (or one that is very similar). We then visit the children of that node in the sorted order instead



(a) Bottom-up.



(b) Top-down.



(c) Serpentine.

Figure 1. Strategies for recycling index structure.

of the (non-)order in which they are actually stored. For classic R-trees, this means ordering the keys by their "proximity" to the split seed values. Because structures such as R-trees do not normally order keys within a page, the goal of this serpentine traversal is to assure that nodes that wind up sharing keys due to splitting/merging are at least neighbors according to this heuristic ordering rather than random keys on the same page.

We will revisit this taxonomy when we discuss their empirical effectiveness on R-trees in Section 4.3.

3.2. Recycling Pointers

In Section 2.2, we discussed what amount to several kinds of TIDs. Just as one can have physical, logical or physiological logging, one can have *physical* TIDs (e.g., relative byte addresses of the form `{page,offset}`), *logical* TIDs (e.g., primary key addresses of the form `{key}`), and any number of hybrid *physiological* TIDs (e.g., `{page,key}`). This is discussed in more detail in [GRAY93, p. 760]. All are used in one system or another. For example, object systems (e.g., POMS [COCK84]) often use relative byte addresses, whereas relational systems (e.g., NonStop SQL) sometimes use primary keys as TIDs.

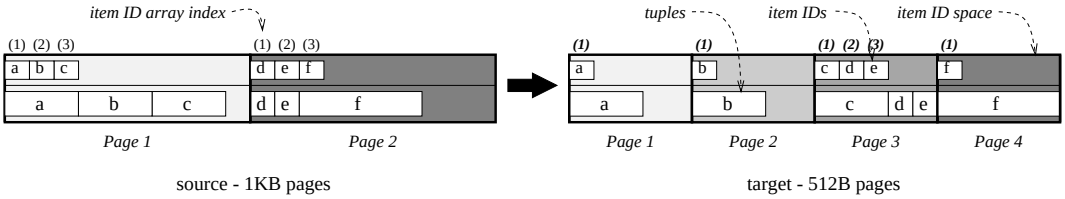
In fact, most database systems use a particular kind of physiological TID instead of the physical TIDs discussed by Sun *et al.* These systems use *slotted pages*; that is, they store an array of *item identifiers* (also known as *slots* or *line arrays*) at a known location on each disk page.[2] Item IDs contain the byte offset within the page of each tuple on that page; TIDs, therefore, are of the form `{page,index}` where `index` is the array index of the item ID that contains the byte offset of the desired tuple on `page`. Although `index` is an index into a physical array, it is immutable (as long as the tuple does not move to a different page) and is therefore a logical identifier within the page. A wide range of data managers use slotted pages, including relational and object-relational databases (e.g., nearly all IBM relational systems [MOHA93], Illustra [ILLU95], Oracle Rdb [HOBB91, p. 79]) as well as object data managers[3] (e.g., ESM [CARE88], O_2 [DEUX90], Papyrus [CONN93], SHORE [CARE94]). The advantages of this scheme are discussed in more detail elsewhere [GRAY93, p. 755].

Item IDs have a critical property: unlike tuples, item IDs are fixed-size. As we will see, the ability to combine the item ID arrays of several pages means that we can calculate the position of a given tuple's item ID on the target page using only its original TID and a small amount of additional per-page information.

Figure 2 demonstrates our proposed method for creating the kind of translation table just described. We will show how the base table pages are copied from *S* to *T*, how the translation table entries are constructed and used, and how the translation table can be made more compact.

[2] Or, in some systems, segments (groups of pages).

[3] The fact that an object data manager exposes logical OIDs to the user does not preclude its internal use of physiological TIDs. For example, SHORE does exactly that.



(a) Changes in base table layout.

source page *s*	array indices of items from *s* on target page ***t*** first	last	target page ***t***	array indices of items from *s* on target page ***t*** first	last
1	(1)	(1)	***1***	***(1)***	***(1)***
1	(2)	(2)	***2***	***(1)***	***(1)***
1	(3)	(3)	***3***	***(1)***	***(1)***
2	(1)	(2)	***3***	***(2)***	***(3)***
2	(3)	(3)	***4***	***(1)***	***(1)***

{2,(1)} → ***{3,(2)}***

= redundant value

(b) Slotted page translation table.

source page *s*	first target page ***t***	array index of first item from *s* on page ***t***	array index of last item from *s* on page ***t,t+1,...***
1	***1***	***(1)***	(1),(2),(3)
2	***3***	***(2)***	(2),(3)

(x) = item ID array index 'x'
roman = value valid at source
bold italic = value valid at target

(c) Compact slotted page translation table.

Figure 2. Example using slotted pages.

Throughout the figure, Times-Roman and ***bold italic*** indicate values valid for S and T, respectively.

In Figure 2(a), we see that the source machine has 1KB pages, whereas the target machine has 1/2 KB pages. Tuples *a* through *f* are being packed into target pages in order to fit them into the minimal number of pages possible without reordering; note that tuples from both source pages 1 and 2 have both been placed on target page 3. In addition to the data items, each page contains an array of item IDs. When a tuple is copied to a page, its item ID is copied to the same page. For pictorial clarity, we depict this array as being stored in a separate part of the page from the tuples.

Figure 2(b) shows how the translation table is constructed and used. We load the translation table incrementally as we copy the base table tuples from source page s to target page t; the table contains an entry for source page s and target page t iff any tuple from s has been copied into t. In that respect, our translation table is similar to the byte offset translation table from [SUN94]. The key difference is that each entry in our translation table stores the *range* of item ID array indices corresponding to the tuples that have been copied from s into t. Recall that we never

reorder tuples, implying that any tuples copied from s to t must be in the same order on both s and t. Furthermore, unlike byte offsets, item ID array indices form *continuous sequences*.[4] Hence, we can translate any array index by simple interpolation.

Figure 2(b) also demonstrates how to recover d's item ID array index. Here, d has source TID `{2,(1)}`. First, we examine the translation table entries that correspond to source page 2. We then find the entry such that our source index falls between the "first" and "last" source array index values in the second and third columns. Our array index, `(2)`, falls in the range `(1),(2)`. This means that we need the fourth row of the table. This row indicates that `{2,(1)}` maps to `{3,(2)}` and that `{2,(2)}` maps to `{3,(3)}`. The target TID for d is therefore `{3,(2)}`. By contrast, the mechanism described by Sun *et al.* cannot recover the full TID for item d without searching page 3.

Finally, Figures 2(b) and (c) indicate how we can make the translation table much more compact. First, we can eliminate the shaded values in Figure 2(b) that can be inferred from other available information. Second, we do not have to store the source page number shown in Figure 2(c) because it is implicit in the array address. Let $|S|$ and $|T|$ represent the number of pages in S and T, respectively. If we assume that each page number is 32 bits and each item ID array index is 16 bits, the table in Figure 2(c) will require $6|S| + 2(|S| - 1) + 2|T| \approx 8|S| + 2|T|$ bytes. This is $2|S|$ bytes smaller than the Sun *et al.* translation table. In general, for typical page sizes, the table will be two to three orders of magnitude smaller than the base table. This should easily fit in main memory.

Note that the preceding discussion assumes that we assign logical sequence numbers $(0 \cdots n)$ to the pages of a table. If not provided by the operating system or the database, such sequence numbers can be assigned on-the-fly while moving the base table.

4. Performance Analysis

In the previous section we described new algorithms for recycling index structure and translating TIDs. This section describes our implementation of these algorithms and the experiments we have performed to demonstrate their effectiveness at reducing the expense of reindexing

[4] Note that the ordering and spacing of the item ID arrays must be preserved so that an index into the source page's array can be used to index into a set of item IDs that may be spread over several target pages. In practice, item ID arrays have gaps corresponding to item IDs for deleted tuples, but preserving these gaps is not a problem because gaps will eventually be reused when new tuples are inserted on a page. Note also that the slotted page indirection means that the physical location of the tuple corresponding to a given item ID does not matter as long as it is still on the same page as its item ID. Hence, our original constraint that tuples are not reordered can be relaxed slightly.

In fact, most slotted page implementations allow a tuple to be replaced by a forwarding TID. The query processing engine will follow such forwarding pointers, which makes possible the relocation of tuples between pages. However, a high proportion of forwarding pointers greatly degrades performance by adding yet another level of indirection and every effort is made to avoid such relocation.

moved tables.

4.1. Implementation in Mariposa

Implementing the algorithms described above in Mariposa was relatively straightforward. First, we modified the storage system to support several required abstractions (e.g., variable buffer sizes). Second, we implemented the additional functionality need to perform the TID translation. That is, we implemented the Sun *et al.* byte offset TID translation routines as well as our own slotted page TID translation routines. We did not actually convert Mariposa into a system using byte offset TIDs; instead, we simulated their algorithm by performing all of the steps required and then placing Mariposa TIDs into the index tuples. Finally, we reimplemented some existing Mariposa utilities to provide credible base cases for our performance comparisons.

The additional code required to implement our recycling routines is extremely small. In fact, each access method requires only 500 lines of C. The byte offset and slotted page TID translation routines (common to all access methods) are under 800 lines.

We spent a fair amount of effort implementing and tuning a B^+-tree bulk-load routine to replace the existing insertion-load routine.[5] A credible sort/build bulk-load is important as a base case for our tests — in the experimental environment described below, building an index on 100MB of uniform random base table data takes over 3.5 hours using the POSTGRES 4.2 insertion-load routine and takes under six minutes using our bulk-load routine.

4.2. Experimental Environment

Our experimental hardware environment consisted of DECstation 3000/300 Alpha AXP workstations rated at 66 SPECint92. All machines were configured with Digital UNIX 3.2, 64MB of main memory and RZ26L disk drives.

We conducted measurements using several different data sets. All base tables and indices were stored as ordinary UNIX files and all base tables were organized as primary heaps. Table 1 summarizes the basic parameters of the data used in our study. The data sets for the B^+-tree tests were synthetic, with integer keys generated uniformly at random. The data sets for the R-tree tests consisted of geographic data obtained from the U. S. Geological Survey. The small data set was taken from the regional version of the Sequoia 2000 benchmark [STON93a] and contained 60,000 points and 80,000 polygons extracted from the USGS GNIS [USGS95] and Land Use

[5] Our bulk-load routine uses the standard technique of extracting `{key,TID}` pairs from the base table, sorting the pairs into index leaf pages and then building the rest of the tree bottom-up. Our external sorting routine follows the recent trend [DEWI91, GRAE92, NYBE94] toward quicksort-based run generation.

Index Type	Base Table (Heap)			
	Cardinality (tuples)	**Size** (bytes)	**Distributions** clustered	unclustered
B^+-tree	10^4	10^6	sorted	random
	10^5	10^7	sorted	random
	10^6	10^8	sorted	random
R-tree	1.4×10^5	1.3×10^7	Hilbert (H_{22})	alphabetic
	1.4×10^6	1.6×10^8	Hilbert (H_{25})	alphabetic

Table 1. Benchmark data sets.

and Land Cover [USGS86] databases for the state of California. The large data set consisted of the contents of GNIS for the entire continental United States and contained nearly 1,400,000 points with their associated place names. Each B^+-tree and R-tree data set was loaded into a base table in two different orders. Depending on the index type, the load order of the data in the base table can strongly affect the build time and final structure of the indices built over it. The base tables used in the B^+-tree studies were loaded in both numerically sorted and random key order, whereas the base tables used in the R-tree studies were loaded in both *least Hilbert value* order [KAME93] and the alphabetic order in which the USGS distributes the data.

The following conventions apply to all I/O measurements described hereafter. Page access counts include both reads and writes. Counts are measured in the file system routines below the buffer manager (i.e., they measure the number of file system requests made by the buffer manager and bulk I/O routines) and therefore do not necessarily correspond to physical I/Os because of file system caching.[6]

[6] Mariposa servers were configured with default buffer pools (512KB shared, 512KB per-server unshared). However, this does not affect the results as much as might be expected; because the server buffer manager does not support either asynchronous (read-ahead or write-behind) or multi-page I/O requests, utilities such as the sort/build routine perform their own bulk I/O using private buffers. In addition, efforts were made to enforce "cold cache" conditions. Server buffer pools were completely flushed between experiments, forcing out even metadata pages. This errs on the side of conservatism because it means that the query startup latency (opening tables, etc.) is larger than in the steady state. The file system cache was flushed between experiments by reading and writing large files.

4.3. Experimental Results

The goals of our experimental measurements were as follows:

- Demonstrate that recycling an index can be substantially faster than rebuilding an index.
- Explore the performance tradeoffs (if any) between recycling and rebuilding in terms of our main experimental variables (table size, page size, data clustering).
- Determine the kind of network bandwidth conditions under which recycling (which inherently involves the transmission of more data than rebuilding) becomes impractical.

These experiments were conducted using ordered trees (i.e., B^+-trees)

In addition to the access method-independent goals, we had the following goal specific to unordered trees:

- Measure the reindexing speedup and retrieval degradation caused by our various heuristics for recycling unordered trees.

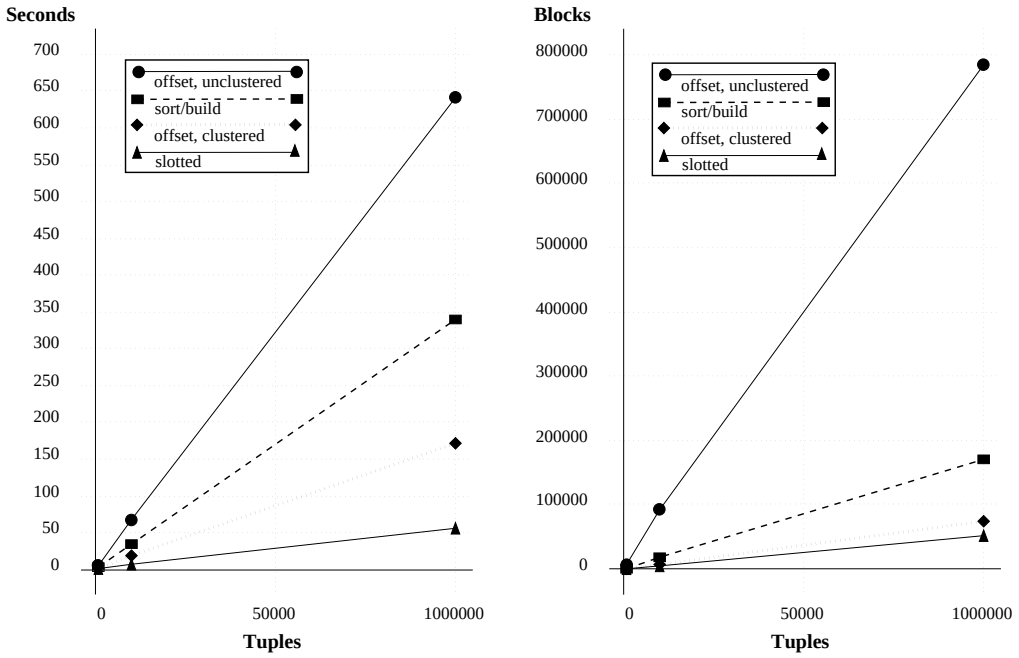
Experiments on Ordered Trees

We performed experiments consisting of all combinations of three table sizes, three target page sizes, and two types of data clustering for a total of 18 experiments. We report only representative data here.[7] Motivated by the analytic results of Sun *et al.*, we measured the performance of our algorithms using some of the same parameters used in their study. For example, merge fan-in for the sorting routine was fixed at their "typical value" of seven for all experiments. Figures 3, 4 and 5 show comparisons between the Sun *et al.* byte **offset** translation mechanism, our own **slotted** page translation mechanism, and the standard **sort/build** mechanism. The latter forms the base case against which we compare the mapping algorithms.

Figures 3 and 4 show the difference in performance between the three mechanisms under different parameters. In these figures, we do not include the cost of reformatting or transmitting the base table over the network because these costs are the same for all three. In addition, we ignore the transmission delay incurred by sending the index over the network; this cost obviously varies widely depending on the type of network used and will be considered in Figure 5.

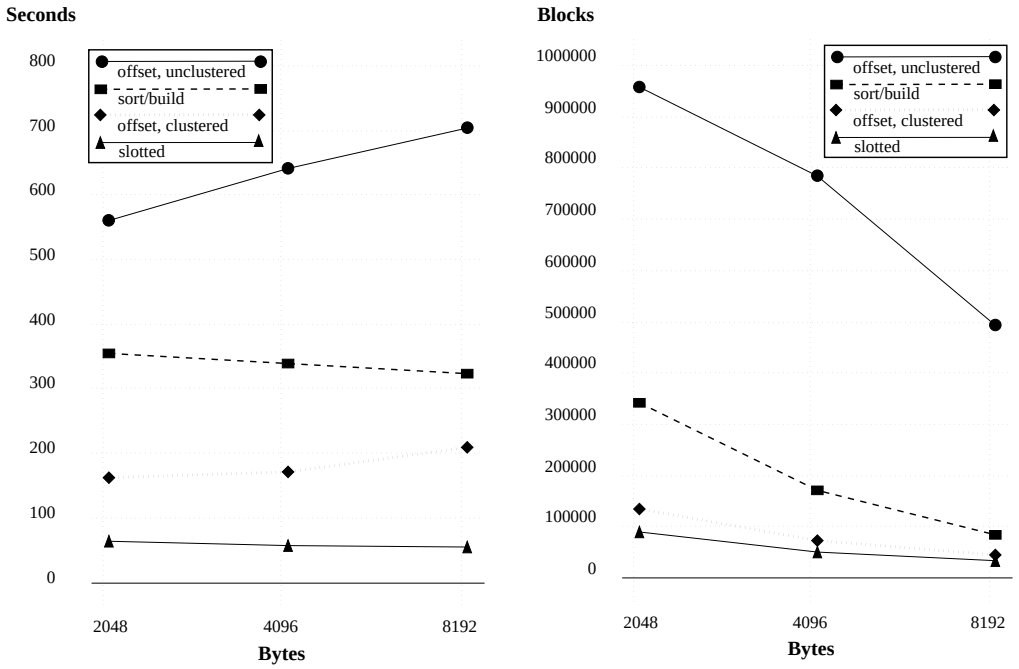
Figure 3 shows how the various algorithms scale up with increasing file size with page size fixed at 4KB. The measurements here are representative of data collected for 2KB and 8KB pages as well. Predictably, the elapsed time and the number of I/Os increases in a linear fashion.

[7] We performed additional experiments that varied the amount of buffer space available to the sort/build routine, but varying this parameter did not make an appreciable difference because very large I/O units provided diminishing returns.



(a) Elapsed time. (b) Number of block I/Os.

Figure 3. Effects of increasing file size (4KB pages).



(a) Elapsed time. (b) Number of block I/Os.

Figure 4. Effects of increasing page size (10^6 tuples).

However, as hoped, index recycling provides significant time and I/O-cost savings over sort/build.

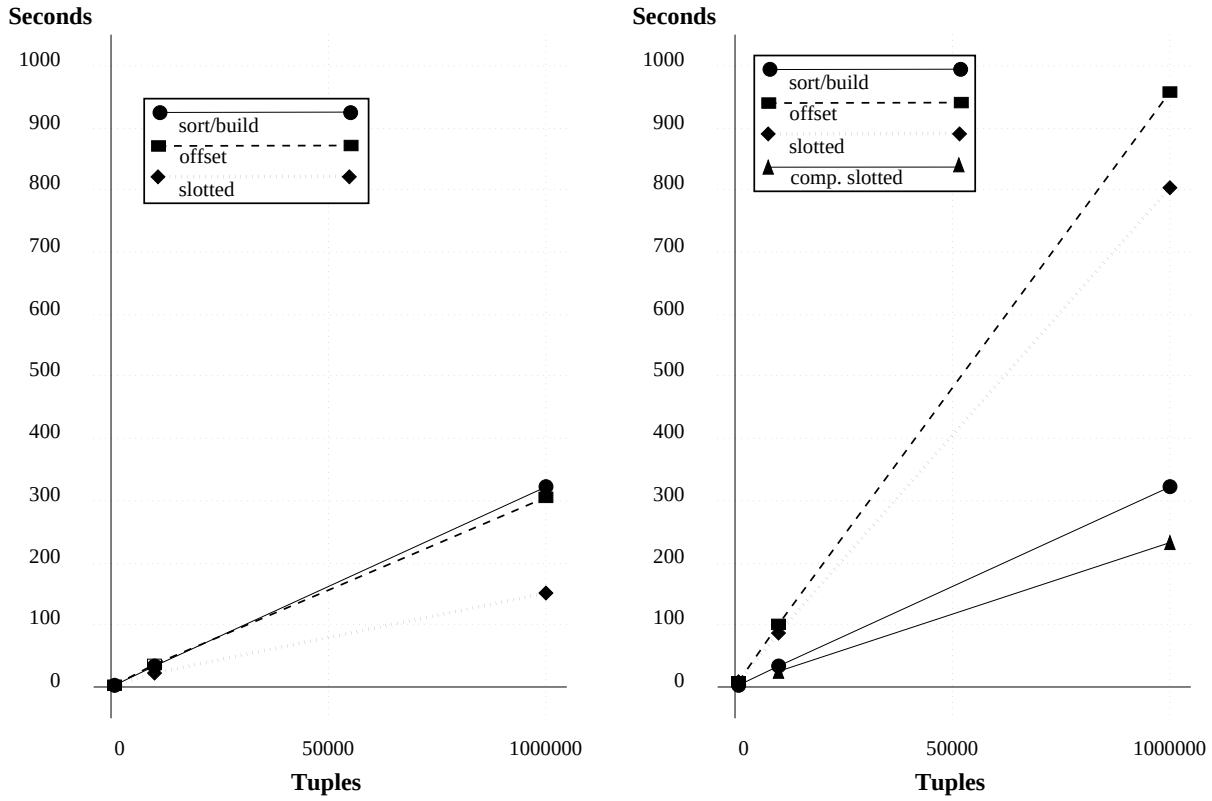
Observe that clustering has varying effects on the different algorithms. We do not differentiate between the clustered and unclustered cases for sort/build and slotted-page translation; the performance of these algorithms is not sensitive to clustering because they only perform one sequential scan of the base table. Because the offset algorithm must randomly probe the base table many times, it is very sensitive to base table clustering. As predicted in Section 2, translating byte offsets by searching base table pages is extremely time-consuming.

Figure 4 shows the effect of target page size on our algorithms with table cardinality fixed at 10^6. Again, this data represents runs made for 10^4 and 10^5 tuples as well. In each plot, we show the performance of reformatting 8KB pages into 2KB, 4KB and 8KB pages. The plots of the sort/build and slotted page algorithms resemble the predictions of Sun *et al.* for sort/build and their own algorithm. Both sort/build and the slotted page recycling algorithm do slightly better as target page size increases because the I/O that goes through the buffer manager is mostly sequential (which is more efficient with larger I/O units). Also, as predicted, sort/build benefits slightly more than index recycling from the increased page size. By contrast, the byte-offset algorithm degrades as page size increases, particularly in the unclustered case. The number of page faults does go down as the page size increases because the probability of the next heap tuple being on the same page as the current heap tuple increases. However, misses are still more common than not, and doubling the page size increases the disk wait (miss penalty). The increased page size therefore makes the byte-offset algorithm less efficient.

Our final set of ordered tree results show how network bandwidth limitations affects the relative performance of the algorithms. The performance analysis of Sun *et al.* was limited to the Ethernet LAN case. Table 2 shows mean network transmission delays obtained by repeated measurement on representative local area (Berkeley), metropolitan area (BARRNet) and wide area (MCINet/AlterNet) networks. These were obtained by simple measurement at various times; they represent neither the best case nor the worst case, but simply indicate the kind of bandwidth available in the US networks at the time. Figure 5 shows the effects of adding these delays to the relative performance of the various algorithms. Algorithms that translate and transmit the leaf pages work well in the high-bandwidth MAN case (and, by extension, the LAN case). As bandwidth decreases, as in the WAN case, sending the leaf pages of the index takes far more time than simply rebuilding the index.

File Size MB (tuples)	Delay by Network Type, sec.			
	FDDI LAN	Ethernet LAN	Regional MAN	National WAN
0.3 (10^4)	0.28	0.39	1.66	6.78
3 (10^5)	1.3	2.63	13.7	79.0
30 (10^6)	13.3	27.1	84.0	735

Table 2. Representative network transmission delays for index files.



(a) MAN case. (b) WAN case.

Figure 5. Total additional end-to-end costs (8KB pages, clustered).

The WAN result is not encouraging. It is possible to make index recycling marginally cheaper than sort/build by compressing the index (e.g., by sending only the TIDs). Reducing the

transmission delay due to index size and adding in the cost of an extra scan of the base table on the target machine results in the "comp. slotted" plot in Figure 5(b). However, we cannot always apply this technique (see [AOKI95]). Under the characteristics described in this section, then, index recycling appears to be practical in local area (10-100 Mb/s) as well as metropolitan area (≈ 1 Mb/s) networks. However, below 100 Kb/s the scheme uses too much network bandwidth to be competitive.

Experiments on Unordered Trees: Avoiding Index Degradation

In our final set of experiments, we benchmarked slotted page index recycling against the Mariposa R-tree build routine. We tested several variations on and combinations of the generic structural recycling methods described in Section 3.1. The methods are indicated by the letters **N** (none), **S** (sort), **P** (partition), **SP** (partition and sort) and **B** (build index). *None* is a naive bottom-up build. Because it packs index tuples from several different source leaf pages onto the same target leaf pages irrespective of whether the source pages are clustered or not, we would expect this heuristic to degrade the clustering of the source index. *Partition* is the same as None, except that the recycling routine avoids placing index tuples from different source pages onto the same target page. This has an obvious impact on the fill factor of the target pages but should not cause as many clustering problems. *Sort* is a serpentine traversal that uses the heuristic described in Section 3.1. *Sort and Partition* is a combination of the two heuristics just described. Finally, *Build* is the standard Mariposa R-tree build algorithm. We measured each of these five methods while copying a table from 2KB pages to 8KB pages and *vice versa*. We performed these ten experiments over both of the two different R-tree data sets and both clustered and unclustered data for a total of 40 experiments.

As previously mentioned, the goal of these experiments was to determine whether inexpensive recycling techniques could be applied without degrading the retrieval effectiveness of the target index. The metrics for our experiments were:

- *Reindexing Performance*: The delay from the initiation of the movement operation to the time when both the target table and its index are available.
- *Retrieval Performance*: Since we are concerned with the performance of actual index instances, we assess the "goodness" of an R-tree using the bounding box coverage metric of Kamel and Faloutsos [KAME93]:

$$P(\vec{q}) = \sum_{n=1}^{N} \prod_{i=1}^{D} (x_{i,n} + q_i)$$

where $\vec{x_n} = (x_{1,n}, \ldots, x_{D,n})$ and $\vec{q} = (q_1, \ldots, q_D)$ are D-dimensional node bounding boxes and

query boxes with side length $x_{i,n}$ and q_i, respectively, and $P(\vec{q})$ is the expected mean number of R-tree nodes visited while searching for $\vec{q}$.

Representative results of our experiments are shown in Figure 6. The figures on the left side correspond to a 4:1 increase in page size, whereas the figures on the right correspond to a 4:1 decrease. The top and bottom sets of figures show the table movement time (in seconds) and $P(\vec{q})$ for both clustered and unclustered versions of the larger R-tree data set.

Figures 6(a) and (b) break down the total table movement time into the time required to reformat the base table and the time to rebuild/recycle the index. As we can see, the index operations are far more expensive than the base table copy. Furthermore, the figures show that rebuilding an R-tree is extremely expensive — Figure 6(a) indicates that a 130MB R-tree build can consume the entire processing capability of a workstation for over 1.5 hours.

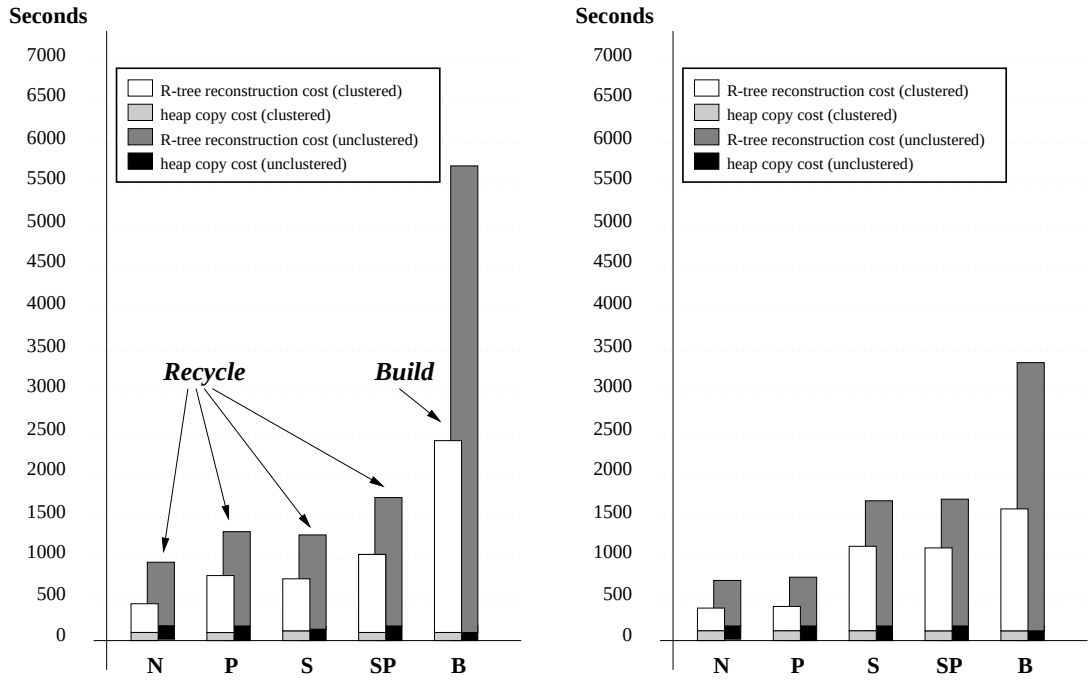
Figures 6(c) and (d) do demonstrate that building a new R-tree almost always produces a better R-tree than recycling. That is, the rebuilt R-tree generally has a smaller $P(\vec{q})$ than the recycled R-trees. However, the results of recycling are often not significantly worse than the result of rebuilding. In particular, the Partition heuristic achieves a fairly reasonable and consistent level of retrieval degradation across all levels of clustering and changes in page size. On the other hand, the Sort heuristic is very sensitive to some of our experimental parameters.

It appears that a simple bottom-up build, combined with the Partition heuristic to prevent the worst cases of declustering, is reasonably effective in producing a "good" index and is many times faster than rebuilding the index. Hence, index recycling appears to be an excellent way to put the table and an index "on line" quickly.

5. Future Work

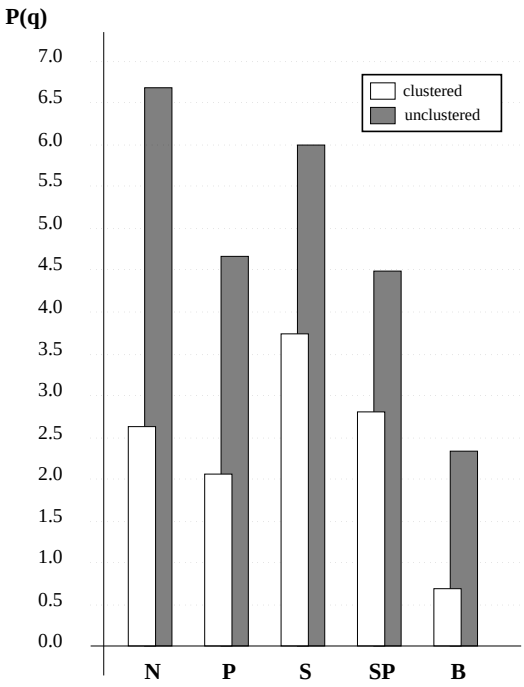
There are several interesting issues that arise in the implementation of index recycling. These are described in more detail in [AOKI95].

- *Lazy Translation*: What are the performance tradeoffs of lazy and eager translation, given that the table and its indices may not be heavily used before the next time they are moved?
- *Support for Additional Access Methods*: Intelligent support for index types other than basic trees will require careful thought. For example, it is not clear that recycling hash indices always makes sense.
- *Parallelism*: Index recycling appears to be "embarrassingly parallel"; a parallel implementation could reveal unforeseen bottlenecks.

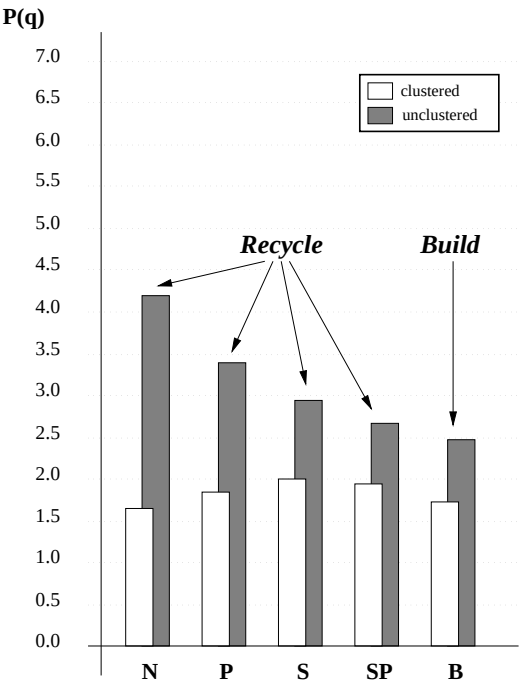


(a) Reindexing time, 2KB→8KB.

(b) Reindexing time, 8KB→2KB.



(c) Retrieval perf., 2KB→8KB.



(d) Retrieval perf., 8KB→2KB.

Figure 6. Reindexing performance vs. retrieval performance for R-trees.

- *Compression*: There are several options for compressing the transmitted index files, ranging from the application of lossless compression utilities to the careful use of semantic compression (e.g., transmitting only the TIDs of the leaf index tuples).
- *Concurrency Control and Recovery*: Because we use a technique similar to "old-master/new-master" rather than in-place translation, we never modify the source base table or index table as part of the translation process. However, naive techniques of this kind are not concurrent; there are a variety of ways to improve reorganization concurrency while retaining recoverability (e.g., [MOHA92, SRIN92]).

6. Conclusions

While the ideas proposed in [SUN94] are useful, the algorithms described are not a good fit for implementation in existing systems. By adapting their algorithms for use with slotted pages, we have produced practical techniques for recycling secondary indices. In addition, we have generalized the notion of recycling to unordered trees as well as ordered trees. Finally, we have empirically demonstrated the performance benefits by implementation and measurement and have provided evidence that our techniques may have wider applicability than LANs.

References

[AOKI95] P. M. Aoki, "Recycling Secondary Index Structures," Sequoia 2000 Tech. Rep. 95/66, Univ. of California, Berkeley, CA, July 1995.

[CARE88] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita and S. L. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview," CS Tech. Rep. 808, Univ. of Wisconsin, Madison, WI, Nov. 1988.

[CARE94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White and M. J. Zwilling, "Shoring Up Persistent Applications," *Proc. 1994 ACM-SIGMOD Conf. on Management of Data*, Minneapolis, MN, May 1994, 383-394.

[COCK84] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey and R. Morrison, "Persistent Object Management System," *Software—Practice & Experience 14*, 1 (Jan. 1984), 49-71.

[CONN93] T. Connors and M. Neimat, "The Papyrus Object Library," in *Persistent Object Systems* (Proc. 5th Int. Wksp. on Persistent Object Systems, Pisa, Italy, Sep. 1992), A. Albano and R. Morrison (ed.), Springer Verlag, Berlin, Germany, 1993, 198-215.

[DEWI91] D. J. DeWitt, J. F. Naughton and D. A. Schneider, "Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting," *Proc. 1st Int. Conf. on Parallel and Dist. Info. Sys.*, Miami Beach, FL, Dec. 1991, 280-291.

[DEUX90] O. Deux *et al.*, "The Story of O_2," *IEEE Trans. Knowledge and Data Eng. 2*, 1 (Mar. 1990), 91-108.

[EICK95] A. Eickler, C. A. Gerlhof and D. Kossmann, "A Performance Evaluation of OID Mapping Techniques," *Proc. 21st VLDB Conf.*, Zurich, Switzerland, Sep. 1995, 19-29.

[GRAE92] G. Graefe and S. S. Thakkar, “Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor,” *Software—Practice & Experience 22*, 7 (July 1992), 495-517.

[GRAY93] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1993.

[HOBB91] L. Hobbs and K. England, *Rdb/VMS: A Comprehensive Guide*, Digital Press, Bedford, MA, 1991. DEC Order Number EY-H873E-DP.

[HORN87] M. F. Hornick and S. B. Zdonik, “A Shared, Segmented Memory System for an Object-Oriented Database,” *Trans. on Office Info. Systems 5*, 1 (Jan. 1987), 70-95.

[ILLU95] Illustra Information Technologies, *Illustra Server User’s Guide, Version 3.2*, Illustra Information Technologies, Inc., Oakland, CA, Oct. 1995.

[KAME93] I. Kamel and C. Faloutsos, “On Packing R-trees,” *Proc. 2nd Int. Conf. on Information and Knowledge Management*, Arlington, VA, Nov. 1993, 490-499.

[KAME94] I. Kamel and C. Faloutsos, “Hilbert R-tree: An Improved R-tree Using Fractals,” *Proc. 20th VLDB Conf.*, Santiago, Chile, Sep. 1994, 500-509.

[MAIE87] D. Maier and J. Stein, “Development and Implementation of an Object-Oriented DBMS,” in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (ed.), MIT Press, Cambridge, MA, 1987, 355-392. Reprinted in: *Readings in Object-Oriented Database Systems*, S. B. Zdonik and D. Maier (eds.), Morgan Kaufmann, San Mateo, CA, 1990.

[MOHA92] C. Mohan and I. Narang, “Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates,” *Proc. 1992 ACM-SIGMOD Conf. on Management of Data*, San Diego, CA, June 1992, 361-370.

[MOHA93] C. Mohan, “IBM Relational DBMS Products: Features and Technologies,” *Proc. 1993 ACM-SIGMOD Conf. on Management of Data*, Washington, DC, May 1993, 445-448.

[NYBE94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray and D. Lomet, “AlphaSort: A RISC Machine Sort,” *Proc. 1994 ACM-SIGMOD Conf. on Management of Data*, Minneapolis, MN, May 1994, 233-242.

[PEAR91] C. Pearson, “Moving Data in Parallel,” *Digest of Papers, 36th IEEE Computer Society Int. Conf. (COMPCON Spring ’91)*, Feb. 1991, 100-104.

[SRIN92] V. Srinivasan, *On-Line Processing in Large-Scale Transaction Systems*, Ph.D. thesis, Univ. of Wisconsin, Madison, WI, Jan. 1992. Also available as CS Tech. Rep. 1071.

[STON87] M. Stonebraker, “The Design of the POSTGRES Storage System,” *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987, 289-300.

[STON93a] M. Stonebraker, J. Frew, K. Gardels and J. Meredith, “The Sequoia 2000 Storage Benchmark,” *Proc. 1993 ACM-SIGMOD Conf. on Management of Data*, Washington, DC, May 1993, 2-11.

[STON93b] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin and M. Olson, “Mariposa: A New Architecture for Distributed Data,” Sequoia 2000 Tech. Rep. 93/31, Univ. of California, Berkeley, CA, May 1993.

[STON94] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin and M. Olson, “Mariposa: A New Architecture for Distributed Data,” *Proc. 10th IEEE Int. Conf. on Data Eng.*, Houston, TX, Feb. 1994, 54-65.

[SUN94] W. Sun, W. Meng, C. Yu and W. Kim, “An Efficient Way to Reestablish B^+ Trees in a Distributed Environment,” *Information Sciences 77*, 3-4 (Mar. 1994), 227-251.

[USGS86] U. S. Geological Survey, “Land Use Land Cover Digital Data from 1:250,000 and 1:100,000-Scale Maps,” Data Users Guide 4, U. S. Geological Survey, U. S. Department of the Interior, Reston, VA, 1986.

[USGS95] U. S. Geological Survey, “Geographic Names Information System,” Data Users Guide 6 (4th printing, revised), U. S. Geological Survey, U. S. Department of the Interior, Reston, VA, 1995.