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

Several important database reorganization techniques move tuples in a table from one location to another in a single pass. For example, the Mariposa distributed database system frequently moves or copies tables between sites. However, moving a table generally invalidates the pointers contained in its secondary indices. Because index reconstruction is extremely resource-intensive, table movement has been considered a expensive operation. In this paper, we present simple, efficient mechanisms for translating index pointers. We also demonstrate their effectiveness using performance measurements of an implementation in Mariposa. Use of these mechanisms will enable parallel and distributed systems like Mariposa to move tables more freely, providing many more options for performance-enhancing reorganizations of the database.

1. Introduction

A variety of important 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 certain types of space reclamation. However, the simple process of moving tuples has a side effect with serious performance implications: because secondary indices in most systems contain tuple pointers that contain physical elements,[1] moving the table invalidates the links between the indices and the underlying table. Rebuilding an entire set of secondary indices from scratch can be expensive, making reorganization a lengthy, heavyweight process. Although this process can be accelerated using parallel sorting and bulk-loading algorithms [PEAR91], parallelism makes the process even more resource-intensive. Since secondary access methods are critical to performance, this expense has been an unavoidable tax.

---

* 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.

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

Reducing the expense of data movement has become an important factor in the design of the Mariposa distributed database management system [STON94b]. The Mariposa model assumes that there are many database sites with widely-varying network connectivity and processing power. Many, if not most, database sites are workstation-class machines; Mariposa algorithms must therefore work well on commodity desktop hardware as well as centrally-administered, massively-parallel servers. Furthermore, the Mariposa design includes an economic model for management of query processing and storage space [STON94a]. A lightweight primitive for moving and copying tables between sites is critical to the system's ability to perform automatic tuning and load balancing using this economic model.

In this paper, we explore the tradeoffs involved in preserving index structures when the underlying tuples shift beneath them. The remainder of this introduction provides more precise definitions 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 Mariposa.

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 the distributed database context, T may be on another machine. (We assume that S has a known number of pages, $|S|$, and a known cardinality, $\|S\|$. These values can be approximate.) Even if we do not reorder the tuples while copying, it still may not be possible to map the contents of a source page into a single target page (or a fixed number of target pages that can be determined *a priori*). 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. For example, when moving data from a Microsoft Windows NT machine based on an Intel x86 processor to a UNIX machine based on a Hewlett-Packard PA-RISC processor, the data manager must alter the tuples to ensure that four-byte integers align on four-byte boundaries. The resulting changes in tuple size may cause pages to overflow in a way that is entirely dependent on the contents 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 copying a table to a different disk partition, changing a table's page size, and compacting pages to reclaim storage space. In Mariposa, the latter operation becomes very desirable after the vacuum cleaner [STON87] runs.
- *Media interchange*. Different storage devices may have different page sizes that are visible to the data manager, either for performance or functional reasons.

We observe that a limited amount of preprocessing by the source site and the transmission of part of the index structure can save a much greater amount of effort at the target site. This becomes possible if the I/O complexity of preprocessing and moving the index is substantially less than the I/O complexity of rebuilding it. For example, if moving the index requires a single scan of the index and rebuilding requires a multi-pass $O(n \log n)$ sort, there is great potential savings. If we are in fact moving the table across a network, we must also consider the relative costs of network I/O and local processing.

Note that we do not necessarily want to preserve the existing index structure. Instead, we want to take advantage of the information stored in the existing index structure in order to save some of the work involved in building a new one from scratch. That is, instead of modifying the existing index, we are essentially *recycling* the materials (e.g., clustering/ordering, base table page pointers, etc.).

Recycling has at least two interesting subproblems. These correspond to the different types of nodes found in secondary index structures, which include:

- *Index nodes that refer to base table tuples*. For consistency, we will call these *leaf* nodes for all data structures, even those that are not trees. In essence, we are attempting to avoid redoing the most expensive steps of a bulk-load process at the target site. For example, this would be the sorting step in the case of a B^+-tree index. The decision problem is to determine whether it is possible and cost-effective to preserve clustering for the given access method. The main implementation problem is that of efficient *TID translation*.
- *Index nodes that refer exclusively to other index nodes*. Such *internal* nodes are very different from leaf nodes. They are generally far fewer in number, their organization has a far more profound effect on search efficiency (because they are consulted early in a search) and they can be reorganized independently of the base table tuples. The decision problem is to determine whether it is possible and cost-effective to preserve the internal structure. In this paper we will not discuss techniques for recycling the internal nodes of an index.

In Section 2, we discuss the options for moving the leaf level of an index. In Section 3, we present the details of our implementation in Mariposa and the results of our experiments. Finally, in Section 4, we discuss future directions and conclusions.

2. Processing Leaf Nodes

In this section we describe when and how we can recycle the leaf nodes of an index structure. We first discuss some of the sufficient conditions for us to recycle indices. We then turn to implementation mechanisms for actually doing so.

All of the techniques described below assume the following model for moving tables. First, we copy S into T without reordering the tuples. While copying, we extract some information which will allow us to determine where a given S tuple can be found in T. Second, we copy the leaf pages of a given index on S into the leaf level of an equivalent index on T. As we copy index tuples, we translate the TIDs that point to S tuples into TIDs that point to T tuples. Finally, we move T and its (partially constructed) indices to the target site. (Alternatively, individual pages of these files can be moved as we complete translation.) The target site then builds the internal levels of the indices, completing the movement process.

2.1. When Can We Preserve Leaf Nodes?

In the most general terms, it can be cost-effective to preserve the leaf level of an index if we can easily apply splitting/merging criteria to a collection of tuples. For example, say a target index page t has overflowed because the page size of T is smaller than that of S. We now wish to share some of its tuples with another page; we might also simply choose to split t. If we must perform an expensive calculation to split t's tuples, or search a large portion of the original index to find a suitable page on which to place overflow tuples, then this process may be prohibitively expensive.

Two common properties can help determine whether we can profitably recycle a given access method. First, if the access method has the notion of equivalent and/or sibling leaf nodes, it is easier to find a node with which to share index tuples. Second, we need some kind of easily computable clustering function so that we can make local decisions when splitting/merging. The need for clustering to be computable rather than implicit in the index structure may not appear to be necessary, inasmuch as we are free to reorganize leaf pages before splitting them or after merging them; only the tuples on the base table pages are kept in order. However, we wish to avoid performing random searching probes of the source index structure and random insertions into the target index structure; both of these lead to poor I/O behavior and generally increase the cost of processing the index.

The common access methods vary in the ease with which they can be recycled. The B^+-tree is an example of an access method that is easy to process. The key ordering and side-links make it easy to find nodes with which to exchange index tuples without damaging the index clustering, and the ordering function is trivial to compute. The algorithm for processing the leaf levels of a B^+-tree is therefore trivial: one simply descends the S index to the least key on the leaf level and reads the leaf pages in side-link order, filling the T index pages. Dynamic hashing access methods that use external overflow techniques, such as linear hashing [LITW80], should also be amenable to recycling. Overflow chains make it easy to grow or shrink buckets. By contrast, naive splitting/merging of unordered tree structures such as R-trees is easy but intelligent splitting/merging is much more difficult [BECK90].[2]

2.2. TID Translation

In the remainder of the paper, we will use the terms *translation* and *mapping* to mean the same thing: conversion of *S* TIDs into valid *T* TIDs within the target index pages.

Translating Individual References

One obvious solution is a simple TID mapping table, but a mapping table consisting of `old-TID` → `new-TID` entries is useless because of its size. For 60-byte tuples and 12-byte TID mapping entries, the mapping table is 20% of the base table size! The mapping table will generally not fit in main memory, slowing translation unacceptably — TID translation in an unclustered index will perform random probes of the mapping table and cause heavy paging.

TID translation mechanisms at the reference granularity have some similarity to *pointer swizzling*, or translation between object reference formats as stored in secondary and primary memory, in object-oriented databases. When primary memory pointers are OIDs, the "how" (as opposed to the "when") part of swizzling is known as the *OID mapping* problem. The mechanisms used include segmented mapping tables (e.g., ObServer [HORN87]), hash tables (e.g., Itasca [EICK95]) and B-trees (e.g., GemStone [MAIE87]). Simple data structures full of `OID` → `address` entries work in this environment because programs frequently exhibit locality of reference and have small reference sets. This means that the portion of the mapping structure kept in memory will be small and well-utilized. However, when moving an index, we know we are processing all references contained in the index in a short period of time without locality guarantees. Pointer-mapping techniques will not perform well in this bulk-translation environment.

Translating Only Page Numbers

Since the main problem with simple TID-mapping structures is the size of the mapping table, an obvious option is to change the mapping granularity in some way. If we constrain the problem, we can reduce the size of the mapping table by storing only per-page information instead of per-TID information. For example, assume for the moment that (1) we are using physical `{page,offset}` TIDs, (2) tuples cannot change size and cannot be reordered, and (3) old

---

[2] Fortunately, in some cases we can impose an inexpensive linear ordering that clusters the data (e.g., least Hilbert value clustering [KAME94]), which makes the R-tree more closely resemble the B^+-tree in terms of our ability to make local clustering decisions.

pages map directly to a fixed number of new pages (e.g., using a constant expansion or contraction factor). In this case, we can use a translation table 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.

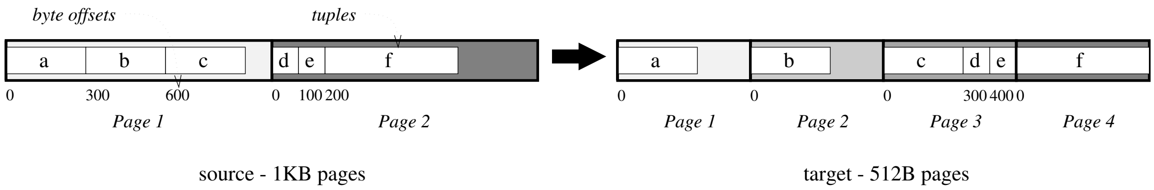
Aside from simple but impractical solutions such as the one just described, there exist at least two proposed TID translation algorithms. Both use page number translation tables that fit in main memory. However, as we will see, both algorithms also fail to translate the source byte offset into a target byte offset without high cost.

Like the simple proposal just described, the original Mariposa design [STON93] uses a simple page number translation table. However, the Mariposa design does assume that tuples can change size in unpredictable ways. This means that an arithmetic expression can no longer be used to calculate byte offsets. Instead, the translation algorithm extracts the source page number from the TID and maps it into a set of one or more eligible target pages. It then searches each of the target page(s) for the desired tuple. (The means of matching the source and target tuples is not specified in the paper, but index keys can be used.) When the desired tuple is located, the page number and byte offset returned by the search routine must be the final target TID.

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(s) do not form a primary or candidate key. However, since real B^+-tree implementations nearly always enforce key uniqueness (through addition of system-generated unique identifiers, if necessary), the latter point is not a serious problem. Finally, the translation table does not provide us with a unique page within the target base table, only a set of potential pages. This makes the search for the matching tuple much more inefficient.

Sun *et al.* [SUN94] address the TID translation problem for the special case of B^+-trees. Like the Mariposa design, they use a page-level translation table and therefore cannot compute the target byte offset without great effort. However, their page-mapping solution is superior to the original Mariposa solution because it accurately maps a source TID to the correct target page.

Figure 1 shows how the byte-offset technique works. We show how the base table is reorganized, how the translation table is created, how the translation table is used, and how the translation table can be made more compact. Values shown in Times-Roman are page numbers or byte offsets that are valid for S. Values shown in ***bold italic*** are page numbers or byte offsets that



(a) Changes in base table page layout.

	source page *s*	offsets from source page *s* valid on target page ***t*** — start	end	target page ***t***	
	1	[0]	[0]	***1***	
	1	[1]	[300]	***2***	
	1	[301]	[600]	***3***	
{2,[0]} →	2	[0]	[100]	***3***	→ ***{3,[?]}***
	2	[101]	[200]	***4***	

= derived (implicit) value

(b) Byte offset translation table.

source page *s*	first target page ***t***	source offset of last tuple from source page *s* on target page ***t,t+1,t+2,...***
1	***1***	[0],[300],[600]
2	***3***	[100],[200]

[x] = byte offset 'x'
roman = value valid at source
bold italic = value valid at target

(c) Compact byte offset translation table.

Figure 1. Example using byte offsets.

are valid for T.

In Figure 1(a), we see that the source machine has 1KB pages, whereas the target machine has 1/2 KB pages. Furthermore, tuples are being packed on target pages in order to fit them into the minimal number of pages possible without reordering; note that the tuples from source pages 1 and 2 have been mixed in target page 3.

Figure 1(b) shows the contents of the translation table corresponding to Figure 1(a) and an example of how the translation table is used. The translation table is loaded when S is copied into T and contains an entry corresponding to a source page s and a target page t iff any tuple from s has been placed on t. In fact, because tuples are not reordered when they are packed on the target pages, we know that a contiguous set of one or more tuples from s has been placed on t. Therefore, in addition to the source and target page numbers, the table contains a range of byte offsets. If the page number from a source TID matches the page number s of some translation table entry and the TID's byte offset falls within the matching range of byte offsets, we know that the tuple corresponding to that TID has been placed on t.

Consider the tuple d in Figure 1(a). Tuple d has source TID `{2,[0]}`. When we encounter an index tuple containing `{2,[0]}`, we examine the translation table entries corresponding to source page 2. There are two such entries, but the source byte offset `[0]` matches the byte offset range in the fourth row of the translation table. We therefore know that d is located on page 3 of T. Notice, however, that there is no way for us to determine that d starts at byte offset 300 of target page 3. The offset must be determined by searching the target page using the key.

The translation table in Figure 1(b) is highly redundant and can be converted into a much more compact form. All of the table cells shown in gray can be derived from other values within the table. Eliminating these cells results in the representation shown in Figure 1(c). If page numbers are 32 bits and byte offsets are 16 bits, we need at least $8|S| + 2(|S| - 1) + 2|T| \approx 10|S| + 2|T|$ bytes to construct the table. In fact, [SUN94] uses a hash table instead of an array and therefore requires more memory.

We note again that the final, key-based translation can be very costly for the two algorithms just described. We can attempt to reduce this cost in two ways: we can modify the query processing engine to handle partially-valid TIDs, or we can implement algorithms that perform this translation efficiently. We discuss each of these options in turn.

If the table being moved is not very active and we are unlikely to use the table or its indices in the near future, it may make sense to leave the byte offsets untranslated. If the index is ever traversed on the target site, the query processing engine can detect the invalid byte offsets. The page number will be valid, so the query processor can simply search the page for the desired tuple instead of accessing it directly. We can even have the database update the TIDs in an index as it dereferences them and determines the correct offsets. For this kind of lazy translation to be desirable, we must make several assumptions. First, we must assume that it is acceptable to slow down traversal of recycled indices (base table pages must now be searched instead of being accessed with the byte offset). Second, we must assume that it is possible to turn index reads into index writes (such may not be the case if the index is on an archival medium, such as a WORM optical drive). Third, we must assume that it will be considered worthwhile to modify the query processing engine in this way. This special case falls into the index scan code and it may not be desirable to slow down all index scans to support this functionality. Finally, we must assume that it is worthwhile to recycle the indices of a table that will not be accessed very frequently in the first place.

Alternatively, we might define the problem in terms of finding a more efficient way to perform a join of the entire index with its underlying base table. One might think of applying one of the many TID-join techniques to make this more efficient. Such techniques make the process of dereferencing many TIDs more I/O-efficient by reordering the references. For example, one might try to adapt ideas from hybrid join [CHEN91]. However, hybrid join requires several sorting steps (something we are trying to avoid because of its expense, especially since we join the entirety of both tables). Another alternative, the nested-block join algorithm, does not scale well. If $|S|$ is large, the probability of having a large number of TIDs with duplicate page numbers on any given index page is rather low. Unless an index page (or set of index pages) has a large proportion of duplicate page numbers, we must still fault in many random base table pages.

Translating Page Numbers and Slots

Notice that we have 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 a few relational systems (e.g., NonStop SQL) use primary keys as TIDs.

In fact, most database systems use a particular kind of TID instead of the physical TIDs discussed in [SUN94]. 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.[3] 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. This scheme is discussed in more detail elsewhere [GRAY93, p. 755]. Systems using this scheme include a wide range of relational and object-relational (e.g., Rdb/VMS [HOBB91, p. 79], nearly all IBM relational systems [MOHA93], POSTGRES [STON91], Illustra) as well as object-oriented (e.g., ObServer [HORN87], ESM [CARE88]) data managers.

The main advantage of the slotted page scheme is that the added level of indirection allows the system to reorganize the storage of tuples within a page without updating all of the TIDs that point to those tuples; this is generally held to outweigh the added space and time overhead of the indirection. Furthermore, unlike tuples, item IDs have the critical property of being fixed-size.

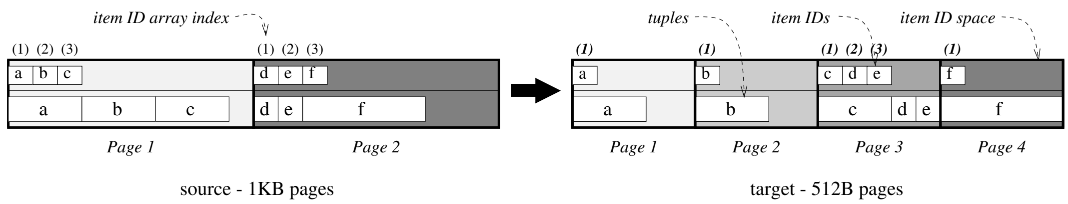
[3] Or, in some systems, segments (groups of pages).

As we will see, if we can figure out how to combine the item ID arrays of several pages, we can calculate the position of a given tuple's item ID on the target page given only its original TID and some amount of additional per-page information.

Figure 2 demonstrates our proposed method for creating translation tables. Just as in Figure 1, we 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. Times-Roman and ***bold italic*** indicate values valid for S and T, respectively.

In Figure 2(a), the shifting of the base table tuples due to page size changes and page compaction is the same as in Figure 1(a). Note that each page now contains an array of item IDs; for simplicity, we depict this array as being stored in a separate part of the page from the tuples. When a tuple is copied to a page, its item ID is copied to the same page.

Figure 2(b) shows how the translation table is constructed and used. Our translation table is similar to the byte offset translation table in several ways. First, the translation table is loaded the same way and contains an entry for source page s and target page t iff any tuple from s has



(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	(1)	(2)	***3***	***(2)***	***(3)***	→ ***{3,(2)}***
	2	(3)	(3)	***4***	***(1)***	***(1)***	

= derived (implicit) 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.

been placed on t. Second, if more than one tuple from s has been copied to t, those tuples are contiguous. Finally, each entry contains a range of item ID array indices that are valid on t. However, there are also several important differences. Unlike byte offsets, item ID array indices form continuous sequences. Suppose that t contains three tuples from s, and that these tuples have source TIDs {s,4}, {s,5} and {s,6}. If {s,4} corresponds to {t,2}, then {s,5} and {s,6} must correspond to {t,3} and {t,4}. Therefore, if our translation table records the first and last array indices in each of these sequences for both the source and the target files, we can translate any array index that falls within a given sequence by simple interpolation.

Recall the example in Figure 1(b), in which we could not recover the byte offset of tuple d. In Figure 2(b) we show how we can 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)}`.

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.[4]

Figure 2(c) shows how we can make the translation table more compact. After using the same redundancy-reducing techniques applied in Figure 1(c), the translation table is relatively small. Each page number is 32 bits and each item ID array index is 16 bits, so the overhead is at worst $10|S| + 2(|S| - 1) + 2|T| \approx 12|S| + 2|T|$ bytes. The nominal overhead of the mapping table is therefore only 2 bytes more per source page than in [SUN94]; in fact, because we use an array and do not have to store the source page number (the hash table key), our structure is actually $8|S| + 2|T|$ bytes (i.e., $2|S|$ bytes smaller than theirs). In general, for typical page sizes, the table

[4] 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.

will be two to three orders of magnitude smaller than the base table. This should easily fit in main memory.

2.3. Implementation Issues

There are several interesting issues that arise in the implementation of these array-based mapping tables. For the most part, these fall into the realm of future work.

2.3.1. Parallelism

Index recycling is "embarrassingly parallel." We can partition the index pages into contiguous sections (as defined by the clustering/ordering imposed by the access method) and translate each section in parallel. The only cost is that the last page in each section might not be as fully packed as it might have been if the entire index had been translated as one section. Furthermore, there is no locking contention for the base table pages, the index leaf pages, or the sections of the mapping table. By contrast, the mapping hash table of the offset method must have conventional mutual exclusion.

2.3.2. Compression

Instead of sending leaf pages, we can send projections of the leaf pages. For example, we can send only the TIDs for each index tuple, gathering the keys from the base table pages and constructing the index leaf pages at the target site. If the base table is well-clustered then this gathering process has good I/O behavior and greatly reduces the amount of data transmitted (we are no longer sending the index tuple header and key). For example, for our experimental indices, this would reduce the amount of transmitted data by a factor of 6. If the data is poorly-clustered then this has the same I/O cost as the Sun *et al.* method on unclustered data. In addition, we can reduce the amount of data transmitted by another factor of 2 by observing that many tables will only need 16 bits for page numbers and 8 bits for item ID indices. It costs nothing to determine whether this additional compression can be applied because it will be known immediately after we build the translation table. (Note that we cannot know this before we start translating because, in general, we do not know *a priori* how many target pages will be required.)

Another interesting option is the application of system compression utilities to the transmitted data stream. We have observed that our database files have similar average-case compression characteristics to ordinary files; 3:1 and 4:1 compression ratios are common. This becomes very attractive if the source and target site have fast processors and poor network connectivity; in a high-speed LAN environment, however, compression can easily make the transmission process slower by similar factors.

2.3.3. Concurrency Control and Recovery

In the techniques as described, recovery is trivial. 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.

One problem with naive old-master/new-master techniques is that they are not highly concurrent. However, known techniques used to improve concurrency of index-building (e.g., [MOHA92, SRIN92]) and other forms of reorganization [SOCK93] can be applied. For example, base table updates can be saved into side files during the translation process and then applied to both the source and target tables.

An additional possibility involves keeping reference counts for the source base table pages in our translation table. When we make our translation pass on the base table, we obtain share locks on each page s and initialize the reference count for s to the number of tuples it contains. Then, when we process the index leaf pages, each mapping operation involving a pointer to s decreases its reference count; when the reference count reaches zero, s is unlocked and can then be modified by other processes. As in the last paragraph, however, we must have some method (not necessarily a side file) for tracking these updates so that they can be applied to the target copy.

3. Performance Analysis

We have implemented several alternative algorithms for building indices, recycling indices, and translating TIDs (i.e., routines to translate base table pages and process B^+-tree pages accordingly). This section describes our implementation and the experiments performed.

3.1. Implementation in Mariposa

A small number of important changes were needed to support the experiments described below. First, we modified the Mariposa storage system to support several required abstractions. We then reimplemented some existing routines to provide credible base cases for our performance comparisons. Finally, we implemented the additional functionality need to perform the TID translation.

The Mariposa storage system code is based on the POSTGRES storage system (e.g., the buffer manager, storage device manager and access methods). However, the POSTGRES code had to be modified. For example, all of the original POSTGRES, storage system code used compile-time constants to determine the size of disk pages. In order to support building base table and index pages of varying sizes, the Mariposa buffer manager and storage manager now supports variable-size buffers. We had to make another set of modifications because the POSTGRES, disk page structures were not uniformly self-describing. For example, each of the POSTGRES access methods defined its own special page format for its internal metadata pages. These special page formats ignored the existing page structure conventions; if followed, these conventions would allow access-method-independent routines to determine (for example) the size of the page. Consequently, Mariposa access method code now uses the correct page structure.

We spent a fair amount of effort implementing and tuning a B^+-tree bulk-load routine to replace the POSTGRES 4.2 insertion-load routine. 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 and uses large private buffers to accelerate its sequential I/O. A credible sort/build bulk-load is important as a base case for our tests, inasmuch as insertion-load is not used in practice. The reason for this is fairly clear — 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 insertion-load routine and takes under six minutes using our bulk-load routine.

Finally, 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 POSTGRES TIDs into the index tuples. The algorithms are implemented as described in the respective papers, with the following exception: we do not need the MAP array described in [SUN94] because POSTGRES page numbers are already logical page numbers within a file instead of physical disk block addresses. The target site therefore does not need to perform this step.

3.2. Experimental Environment

Our experimental environment consists of DECstation 3000/300 desktop workstations. These machines have a single 150MHz DECchip 21064 Alpha AXP processor with a 256KB L2 cache and are rated at 66 SPECint92. All machines were configured with Digital UNIX 2.1 or 3.2, 64MB of main memory[5] and a single internal RZ26/26L disk drive.

[5] Mariposa servers were configured with default buffer pools (512KB shared, 512KB per-server unshared). These are small, but 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.

Efforts were made to enforce strict "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 startup transient (opening tables, etc.) is larger than in the steady state. Further, since Mariposa tables are ordinary files, the file system cache was flushed between experiments by reading and writing large files.

Tables are stored as ordinary UNIX files. All base tables were organized as primary heaps but varied in cardinality (10^4, 10^5 and 10^6 tuples). Tuple size was fixed, resulting in approximate base table sizes of 1MB, 10MB and 100MB, respectively. Index tables were one-third the size of the corresponding base tables. We built indices using four-byte integer primary keys in both clustered (sorted) and unclustered (uniform random) heap distributions.

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.

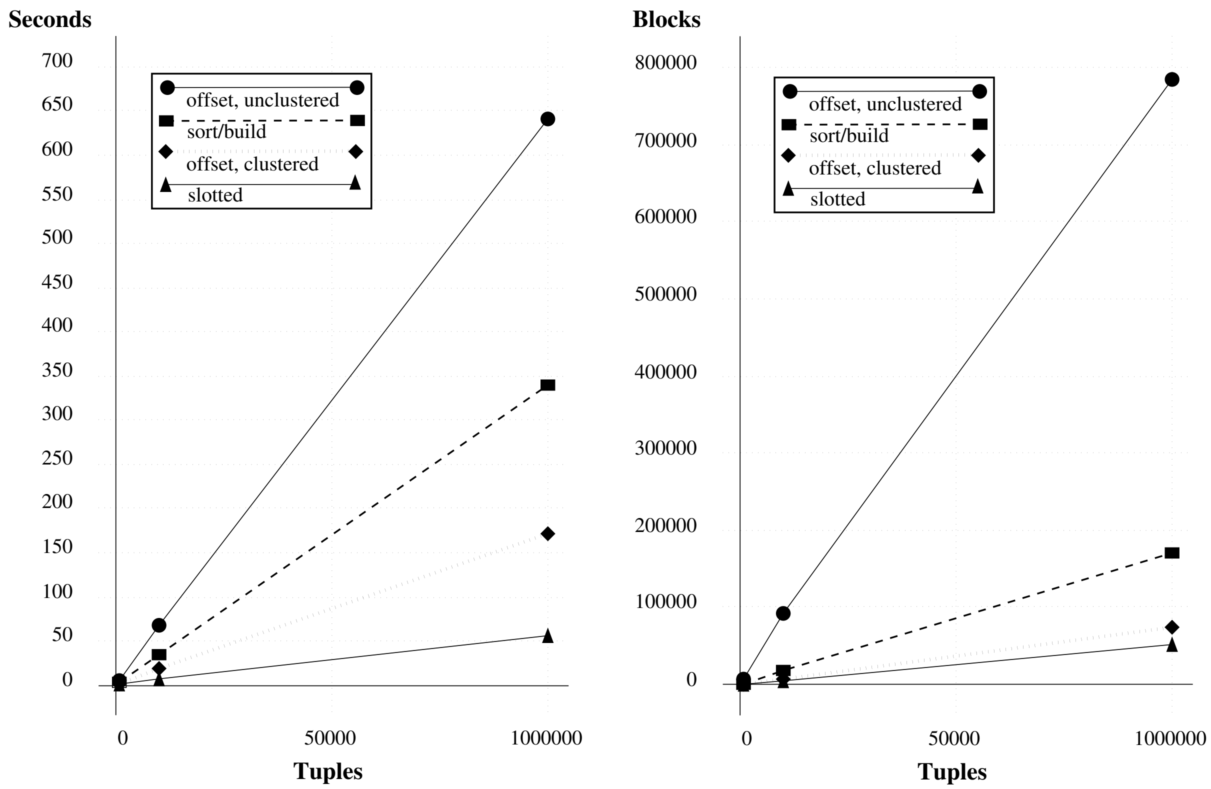
3.3. Experimental Results

We performed a large number of experiments, the most significant of which we report here. Motivated by the analytic results of [SUN94], we measured the performance of our algorithms using some of the same parameters used in their study. The figures 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 our base case.

The next two figures 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.

The parameters we varied include the file size (characterized in the figures by the table cardinality), the target page size and whether the source heap tuples were clustered on the indexed column or not. 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. Merge fan-in was fixed at seven, a typical value according to [SUN94], for all experiments.

Figure 3 shows how the various algorithms scale up with increasing file size. Predictably, the elapsed time and the number of I/Os increases in a linear fashion. Even the sort/build algorithm appears to scale linearly, since $n \log n$ growth is difficult to distinguish from linear growth when n is large. However, as hoped, index recycling provides significant time and I/O-cost savings over sort/build.

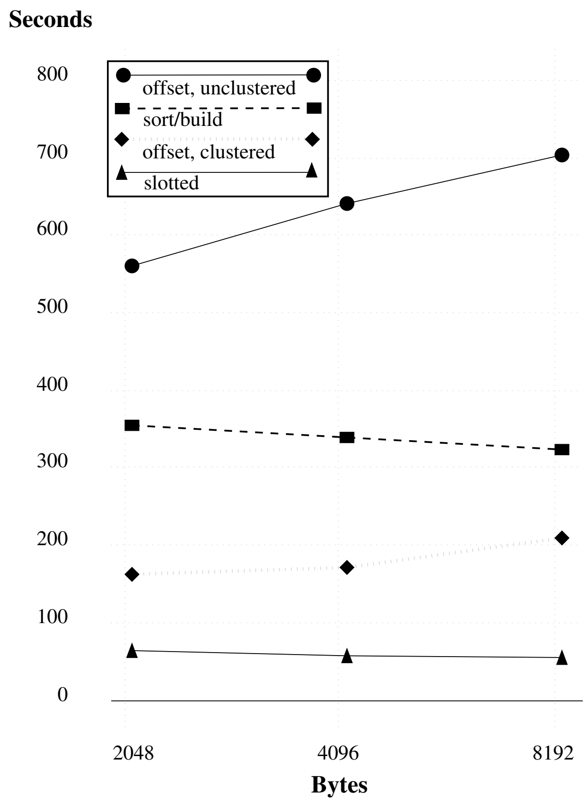


(a) Elapsed time. (b) Number of block I/Os.

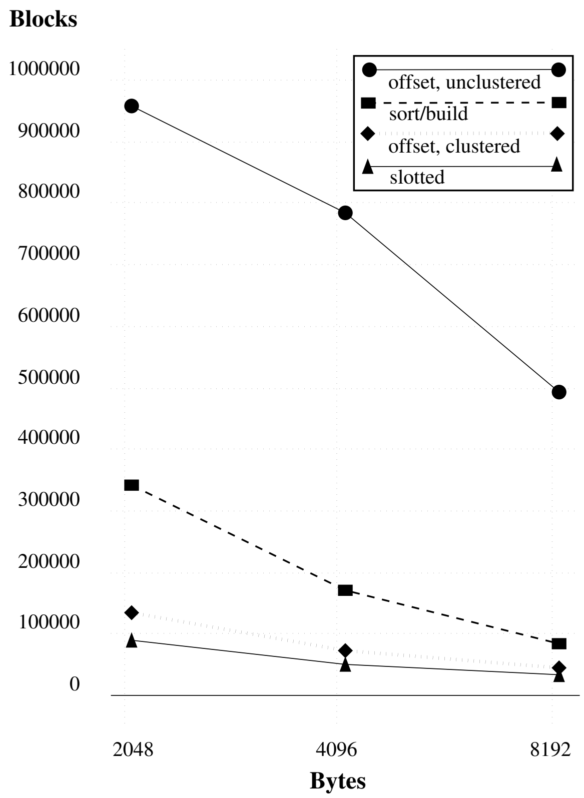
Figure 3. Effects of increasing file size (4KB pages).

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 end-to-end 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. This is true for Figure 4 as well. 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. 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 [SUN94] for sort/build and



(a) Elapsed time.



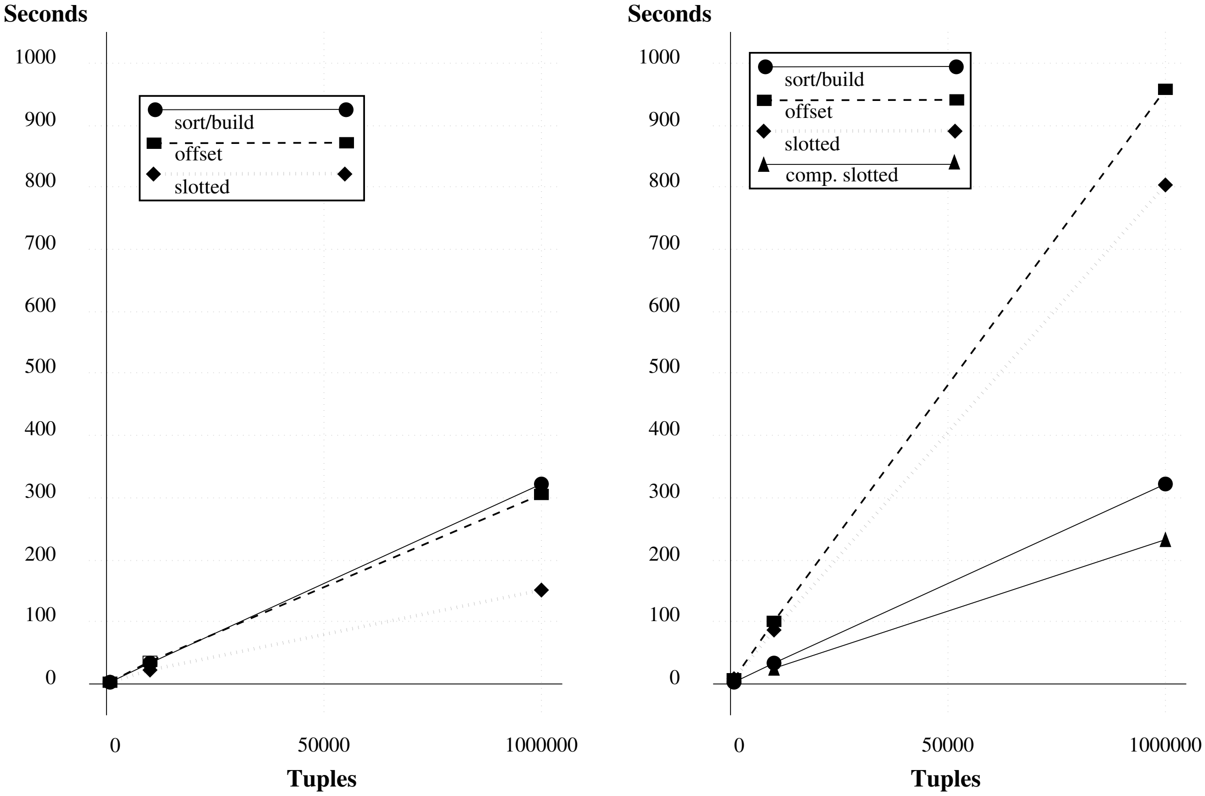
(b) Number of block I/Os.

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

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.

File Size, MB (tuples)	**Delay by Network Type,** s.			
	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 1. Representative network transmission delays for index files.



(a) MAN case.

(b) WAN case.

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

Our final set of results show how network bandwidth limitations affects the relative performance of the algorithms. The performance analysis in [SUN94] was limited to the Ethernet LAN case. Table 1 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 in mid-1995. Figure 5 shows the effects of these delays on the relative performance of the various algorithms. Algorithms that translate and transmit the leaf pages work well in the bandwidth-rich MAN case (and, by extension, the LAN case). As bandwidth becomes more scarce, as in the WAN case, sending the leaf pages of the index takes far more time than simply rebuilding the index.

The WAN result is not encouraging, but we can make index recycling marginally cheaper than sort/build by compressing the index (i.e., sending only the TIDs). However, as previously discussed, such compression can only be applied effectively in the clustered case.

Under the characteristics described in this section, index recycling appears to be practical in local area (10-100 Mb/s) as well as metropolitan area networks (≈ 1 Mb/s). However, below 100 Kb/s the scheme uses too much network bandwidth to be competitive.

4. 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. Our techniques do not require modification of the query execution engine, novel index/heap join methods or prohibitively expensive translation steps. Furthermore, unlike [SUN94], we have demonstrated the performance benefits by implementation and measurement and have provided evidence that our techniques may have wider applicability than LANs.

We are investigating several new directions in data movement algorithms. In terms of making movement more efficient, the following questions should be answered:

- 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?
- Are there convincing arguments for recycling internal nodes?
- What performance improvements are possible when recycling access methods in which reclustering can be extremely expensive (e.g., R-trees without an ordering heuristic)?
- The Mariposa design in [STON94b] proposes a *Mariposa canonical representation* (MCR) for tables and indices. MCR is an architecture- and page-size-neutral format on which all sites can perform minimal processing; this format minimizes (1) the number of formats sites must be able to send/receive and (2) the amount of translation required to receive a table. (2)

is critical for sites such as tertiary storage file servers which house mostly-cold data and have low processing power relative to the amount of data they store. What are the costs and benefits of moving tables using MCR?

In support of our general goal of finding non-quiescing (on-line) data movement algorithms, we are studying concurrency control and recovery issues. As previously mentioned, variations of the techniques used during on-line index building can be employed.

References

[BECK90] N. Beckmann, H. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. 1990 ACM-SIGMOD Conf. on Management of Data*, Atlantic City, NJ, June 1990.

[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.

[CHEN91] J. Cheng, D. Haderle, R. Hedges, B. R. Iyer, T. Messinger, C. Mohan and Y. Wang, "An Efficient Hybrid Join Algorithm: A DB2 Prototype", *Proc. 7th IEEE Int. Conf. on Data Eng.*, Kobe, Japan, Apr. 1991, 171-180.

[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.

[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.

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

[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.

[KAME94] I. Kamel and C. Faloutsos, "Hilbert R-tree: An Improved R-tree Using Fractals", CS-TR-3032 , Univ. of Maryland Institute for Advanced Computer Studies, Dept. of Computer Science, Univ. of Maryland, Feb. 1994.

[LITW80] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing", *Proc. 6th VLDB Conf.*, Montreal, Quebec, Oct. 1980, 212-223.

[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 (editor), MIT Press, 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.

[SOCK93] G. H. Sockut and B. R. Iyer, “Reorganizing Databases Concurrently with Usage: A Survey”, Document Nr. TR 03.488, IBM Santa Teresa Laboratory, San Jose, CA, June 1993.

[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.

[STON91] M. Stonebraker and G. Kemnitz, “The POSTGRES Next-Generation Database Management System”, *Comm. of the ACM 34*, 10 (Oct. 1991), 78-92.

[STON93] 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.

[STON94a] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer and C. Staelin, “An Economic Paradigm for Query Processing and Data Migration in Mariposa”, *Proc. 3rd Int. Conf. on Parallel and Dist. Info. Sys.*, Austin, TX, Sep. 1994, 58-67.

[STON94b] 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.