DESIGN AND PERFORMANCE EVALUATION OF A MAIN MEMORY RELATIONAL DATABASE SYSTEM

by

Tobin Jon Lehman

Computer Sciences Technical Report #656

August 1986

DESIGN AND PERFORMANCE EVALUATION OF A MAIN MEMORY RELATIONAL DATABASE SYSTEM

by

TOBIN JON LEHMAN

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1986

Abstract

Most previous work in the area of main memory database systems has focused on the problem of developing techniques that work well with a very large buffer pool. This dissertation addresses the problems of database architecture design, query processing, concurrency control, and recovery for a *memory resident* relational database, an environment with a very different set of costs and priorities. An architecture for a memory-resident database system is presented, along with a discussion of the differences between memory-resident database systems and conventional disk-based database systems. Index structures are then studied for a memory-resident database environment. The T Tree, a new index structure designed for use in this environment, is introduced and compared with several existing index structures: Arrays, AVL Trees, B Trees, Extendible Hashing, Linear Hashing, Modified Linear Hashing and Chained Bucket Hashing. The T Tree is shown to perform well in a memory-resident environment. Several of the index structures are then used to examine relational join and projection algorithms for a main memory database environment. Experimental results show that a Merge Join algorithm that uses a T Tree index is usually the best method, and that a simple Hash Join algorithm is usually second best.

Recovering a memory-resident database is different from recovering a disk-oriented database, so a different approach is taken in this dissertation. Existing proposals for memory-resident database recovery treat the database as a single entity, so recovery and checkpoint operations are applied to the entire database. A new design is proposed that allows logging, checkpointing and recovery to be done at the relation or index level, providing a form of *demand recovery*. After a crash transactions declare the relations that must be restored before they can run, with undemanded relations being recovered by a background task. Finally, the cost issues for concurrency control are different for a memory-resident database system. Locking is more costly on a per database access basis, so it must be made more efficient. Multiple granularity locking is desirable, but it would be too expensive if several levels of locks needed checking for every database reference. An algorithm is presented that uses locks with a dynamic level of granularity, with locks being escalated or de-escalated in size to meet the system's concurrency requirements.

Table of Contents

List of Figures

List of Graphs

List of Tables

Chapter 1

Introduction

Today, medium to high-end computer systems typically have memory capacities in the range of 16 to 128 megabytes, and it is projected that chip densities will continue their current trend of doubling every year for the foreseeable future [Fishe 86]. As a result, it is expected that main memory sizes of a gigabyte or more will be feasible and perhaps even fairly common within the next decade. Some researchers believe that many applications with memory requirements that currently exceed the range of today's memory technology will thus become memory-resident applications in the not-too-distant future, and that the database systems area is certain to be affected in some way by these trends [Garci 83]. By switching to a memory-resident database, a relational database system gains significant potential performance improvements due to the lack of disk interaction and the avoidance of disk-oriented algorithms. For those applications having storage requirements that continue to exceed memory capacities, there may still be often-referenced relations that will fit in memory, in which case it may pay to partition the database into memory-resident and disk-resident portions and then use memory-specific techniques for the memory-resident portion (much like IMS Fastpath and IMS [Date 81]).

In addition to traditional database applications, there are a number of emerging applications needing relational model support for which main memory sizes will almost certainly be sufficient — applications that wish to be able to store and access relational data mostly because the relational model and its associated operations provide an attractive abstraction for their needs. Horwitz and Teitelbaum have proposed using relational storage for program information in language-based editors, as adding relations and relational operations to attribute grammars provides a nice mechanism for specifying and building such systems [Horwi 85]. Linton has also proposed the use of a database system as the basis for constructing program development environments [Linto 84]. Snodgrass has shown that the relational model provides a good basis for the development of performance monitoring tools and their interfaces [Snodg 84]. Finally, Warren (and others) have addressed the relationship between Prolog and relational database systems [Warre 81] and have shown that having efficient algorithms for relational operations in main memory could be useful for processing queries in future logic programming language implementations.

Previous studies of how large amounts of memory will affect the design of database management systems have focused mostly on how to make use of a large buffer pool [DeWit 84], [Elhar 84], [DeWit 85], [Shapi 86]. However, a number of researchers have recently begun exploring various aspects of memory-resident designs [Lelan 85], [Amman 85], [Salem 86], [Eich 86], [Hagma 86]. It is possible to use a large amount of memory simply to create a large buffer pool capable of holding the entire database, and this approach has several advantages. In particular, no major programming work is needed, since changing the buffer pool size involves simply changing a constant in the DBMS, and the speed of the overall system will increase since fewer disk accesses will be required. However, a buffer-oriented database system has a number of inherent handicaps. For example, every reference to the database must go through the buffer manager, causing the buffer manager to first check its page list to see if the page is in memory, then perform the address calculation, and finally, if necessary, pin the page to guarantee that it is not swapped out prematurely. Thus, it seems that there is much to be gained by redesigning a database system to avoid contact with the disks and the buffer manager to the maximum extent possible.

1.1. New Possibilities

A memory-resident database architecture presents new possibilities that were not available with a large-buffer-oriented database architecture. For example, memory-resident database systems do not require disk accesses to read database information, so it is possible to assume that there will be no process switches caused by disk accesses during normal transaction processing. (And, as will be seen in Chapter 5, transactions can run asynchronously with disk-oriented recovery operations, thus avoiding I/O-triggered process switches in the recovery component as well.) Therefore, process scheduling policies can include the possibility of variable-length time-slices so that processes will be interrupted only at well-known, acceptable points in their execution. Such an approach can combine benefits from both preemptive and non-preemptive scheduling policies.

The direct addressing capability of a memory-resident database architecture allows tuple pointers to be used where copies of field values or tuples are needed in disk-resident designs. Small fixed-size tuple pointers can make index structures more compact and their algorithms easier to implement. Temporary relations, consisting only of tuple pointers, can also be much smaller. Finally, tuple pointers stored in fields of tuples (*i.e.* links) can enhance the performance of pre-computed joins.

1.2. New Problems

This new found wealth of possibilities is not without its problems. With the new memory-resident database environment come different cost formulas, increased emphasis on minimizing cost, and increased vulnerability to system crashes. The memory-resident approach requires new directions in database management system design — the algorithms and data structures for query processing, concurrency control, and recovery must all be structured to stress the efficient use of CPU cycles and memory rather than disk accesses and disk storage. In a memory-resident database system, CPU cost is the most important cost component for query execution, so overall execution path length must be minimized.

Index structures for disk-based database systems are designed to minimize disk accesses, so index node sizes usually correspond to the size of a disk page. Updates applied to these large, page-size nodes often require large amounts of data movement for index entry reorganization. If, on the other hand, a directory is used to minimize data movement cost on a page, then the page search cost is increased because of an added indirection through the directory. Algorithm descriptions for most hash-based index structures are usually concerned only with finding the desired disk page, and the performance of such algorithms is usually measured in terms of disk accesses. Locating a particular record within a page is considered secondary; Litwin's implementation of Linear Hashing, for example, employs a simple linear search [Litwi 85]. Index structures for a memory-resident database system must be designed to support efficient search and update operations, so search routines should not involve unnecessary computation and update operations should involve as little data movement as possible.

Query optimizers in disk-based database systems usually use disk I/O as the main cost parameter in their cost model, and model CPU cost in only general terms, if at all. In fact, in a well-known disk-based database system prototype, R^*, only recently has CPU cost been shown to be an important cost parameter that must be modeled in detail [Macke 86]. In memory-resident database systems, query execution costs are based only on CPU execution time. Query optimizers must be fitted with new cost functions appropriate for memory-resident database systems, and alternative query processing algorithms must be analyzed in a memory-resident database environment to determine their computation costs.

Recovering data after a system crash is more complicated in memory-resident database systems, as the entire database resides in volatile memory. Only data that is kept in stable memory can be recovered after a power loss, so a stable copy of all committed data must be kept at all times. (Making large random-access memories stable is not currently cost effective, so we assume that stable storage comprises mostly disk memory.) Logging all database modifications to stable storage does protect the database from system crashes, but recovery time can be very large unless the amount of log data is kept small, so frequent checkpoints of the database are needed. Current checkpointing schemes write to disk all data updated since the last checkpoint to disk, thereby causing a large amount of disk activity on a regular basis. Similarly, for crash recovery, most schemes reload the entire database and recover from the log in one operation. If the database is very large, on the order of several gigabytes, then transactions waiting to run after a system crash may have to wait a long time if they are not allowed to start until after the entire database has been restored.

The cost of concurrency control mechanisms used in disk-based database systems would not change significantly if they were used in a memory-resident database environment, as most concurrency control operations are already done in main memory. However, the *relative* costs of concurrency control and database accesses would change significantly. For example, the cost of a lock operation, involving a lock table lookup, and the cost of a database access, involving an index lookup, will be similar in a main memory environment. For maximum sharing, most database systems allow record-level locking, but this presents problems for transactions that need to access large portions of the database. Some systems opt for hierarchical locking protocols that allow locks to be set at any of several granule sizes. Unfortunately, using hierarchical locks can involve setting several locks for each database reference, thereby causing a situation where the locking cost is potentially much greater than the cost of a database reference. A well-known cost metric for concurrency control is that the cost of synchronization should not exceed its benefits [Gray 78]. Concurrency control mechanisms must be re-examined in a memory-resident environment and their costs must be reduced.

1.3. Thesis Overview

This dissertation addresses many of the design and performance issues associated with the design of a main memory database system. Five main areas are addressed: basic data architecture, index structures, query processing, recovery and concurrency control. Each is examined in light of the unique problems of a memory-resident database environment. It will be seen that many algorithms designed for disk-resident database systems are not appropriate for memory-resident database systems, and that even some algorithms designed for memory-resident database systems lack the necessary qualities to provide the *high performance* that one would expect of such systems.

In optimizing a program, it is often possible to replace an operation that is done in a loop with a slightly more expensive operation that is done only once before the loop and amortize the cost of the more expensive operation over the number of times the loop is executed. In a database system, similar optimizations exist. Chapter 2 describes uses of links, direct addressing, and compiled code to eliminate various levels of computation or interpretation, thus moving repeated operations out of the "loop" and replacing them with operations that cost a little more up front, but are less expensive overall. Chapter 2 also presents a software architecture for a main memory database management system (MM-DBMS), describing the structure of memory *partitions*, the basic building block of relations and index structures; the structure of records in a relation; and the contents of database catalogs that describe the contents of the database.

Chapter 3 presents a description of the T Tree, an index structure that has been designed specifically for use in a main memory environment. Also presented are the results of a set of experiments that were conducted on several different types of index structures in a main memory environment: AVL Trees, B Trees, Extendible Hashing, Linear Hashing, Modified Linear Hashing, Chained Bucket Hashing, and T Trees. In previous work on balanced trees or hashing for main memory, theoretical results such as asymptotic retrieval and update complexity have been the main concerns. Practical concerns such as constant factors have often been considered unimportant. The experimental results of Chapter 3 show that constant factors are indeed very important in a memory-resident database environment, and that the T Tree and Modified Linear Hashing index structures both perform well in a main memory environment.

In Chapter 4, several index structures from Chapter 3 are used in experiments that measure the performance of several relational join and project algorithms. In the join algorithm experiments, three nested loop join algorithms (Hash Join, T Tree Join and Pure Nested Loops Join) and two merge join algorithms (T Tree Merge and Array Sort Merge) are tested. The results of the join algorithm experiments show that, with a few exceptions, the T Tree Merge Join algorithm is preferred when the proper indices exist, and, when the proper indices are not available, the Hash Join algorithm is the best choice. In the project algorithm experiments, two algorithms are tested: a Sort-Scan algorithm and a Hashing algorithm; Hashing is shown to be the preferred method.

Chapter 5 describes several traditional disk-oriented recovery methods and argues that they are inappropriate for a memory-resident database system. It then describes several recent proposals for recovery in memory-resident database systems, and argues that they, too, fall short of the requirements of a high-performance database system. Finally, a new recovery mechanism based on partition-level logging, partition-level checkpointing, and partition-level demand-recovery is proposed. An analysis is provided that indicates this new proposal should perform very well in a memory-resident database environment.

Chapter 6 discusses concurrency control for a memory-resident database system and shows that the cost issues are different from those of a disk-resident database system. A new concurrency control algorithm based on locking is introduced. The algorithm uses two different locking granule sizes, depending on the concurrency needs of the system. It attempts to reduce overall locking cost by using expensive, fine granularity locks only when a high level of data sharing is needed; less expensive, coarse granularity locks are used otherwise.

Finally, Chapter 7 summarizes the dissertation and suggests topics for future work in the area of memory-resident database systems.

Chapter 2

Main Memory DBMS Architecture

Memory-resident database systems do not require buffer management, hence, relations in memory-resident database systems can be anchored at fixed locations in memory for the duration of their existence. Except in situations where relations experience repeated growth and decay in large amounts, tuples will stay in well-known locations inside their relations for long periods of time, allowing them to be referenced directly by memory address. This direct addressing affects all database operations that reference individual database entities. The cost of referencing a tuple *via* a tuple pointer is inexpensive in main memory, as it is only the cost of a memory indirection, so tuple pointers can be used wherever copies of tuples or tuple field values would otherwise be used. The small, fixed-size nature of tuple pointers and the ability to reference tuples randomly affects many areas in a memory-resident database architecture.

In this chapter the architecture of a Main Memory Database Management System (MM-DBMS) is presented. The MM-DBMS design is motivated by the desire to exploit all possible uses of direct addressing and memory-residence.

2.1. Advantages of Direct Addressing and Memory-Residence

Using tuple pointers in place of actual field values provides several benefits for index structures. First, a single tuple pointer provides an index with access to both the attribute value of a tuple and the tuple itself, thus reducing index size. Second, this eliminates the complexity of dealing with long fields, variable length fields, compression techniques, and calculating storage requirements for the index. Third, moving pointers tends to be cheaper than moving the (usually longer) attribute values when updates necessitate index operations. Finally, since a single tuple pointer provides access to any field in the tuple, multi-attribute indices need less in the way of special mechanisms.

Using tuple pointers also benefits temporary relations. Temporary relations holding query results do not need to hold copies of the tuples participating in the query answer set; they can simply hold pointers to them. Tuple fields taking part in the query answer set can be identified by a descriptor that is used to extract them when the query results are presented[1]. (An example of a descriptor is shown in Section 2.4.3.) As a result, temporary relations require much less space when tuple pointers are used than when whole values are used, and their space requirements are easier to calculate; of course, the number of result tuples in a query answer set cannot be known in advance, but the space requirements for individual result tuples can be known, as they are composed of fixed-size tuple pointers. Also, temporary relations composed of tuple pointers are less costly to create, as less data needs to be copied.

Just as query results can be stored as tuple pointers in a temporary relation, they can also be stored as tuple pointers in a regular relation. When an expensive join operation is expected to be performed often, it may be preferable to instead perform the join operation once and store the results (as tuple pointers) in one of the relations involved in the join operation. The stored tuple pointer represents a *link* between the tuple holding the tuple pointer and the tuple being pointed to. Once links have been stored in a relation, the corresponding join results can be retrieved by simply scanning

[1]If the results of a query are to be saved for use beyond the end of the query transaction, then actual values must be copied, as tuple pointers can be rendered invalid by the delete operations of subsequent transactions.

the relation holding the links. (An example of links is presented in Section 2.4.5.)

Another advantage of memory-residence is related to scheduling. Since disks are not used to read database information during normal transaction processing, and transactions are never interrupted because of disk I/O, transactions can be given partial control over when they relinquish the CPU. It is possible for a transaction to tell the CPU that it should not be preempted during a particular (short) operation, such as during a modification to a system data structure while the transaction is executing in a critical section. This ability allows the database system to use preemptive scheduling and still be able to avoid the convoy problem (as described in [Blasg 79]).

2.2. Problems with Direct Addressing

Direct addressing of tuples provides many benefits, but it also creates some problems. The more database entities there are that reference a tuple directly, the greater the update cost when that tuple moves, as all pointers to that tuple must be updated with the tuple's new location. Fortunately, tuple movement should be rare, caused only by relation reorganization due to a large number of deletes or by a tuple's variable-length field overflowing the available space in a partition. It is probable that in the event of every tuple in a relation being relocated due to relation reorganization, the relation's indices would simply be rebuilt. However, tracking down unidirectional links could be an expensive task. Fortunately, there is a solution to this problem, and it is presented in section 2.4.4.

2.3. Environmental and Parametric Assumptions

Before presenting the MM-DBMS design, we state the underlying assumptions of the environment.

- The computer's physical memory is large enough to hold the entire database and supporting data structures (catalogs, indices, etc). The majority of the random access memory is volatile, but a small portion (on the order of a few megabytes) can be made stable and very reliable. (This is considered to be a reasonable assumption given current memory technology [Goodm 86].)
- Only a single processor is used for the main tasks involved in query processing. However, the algorithms presented in this dissertation are intended not to preclude future extensions to a multiprocessor environment. Chapter 5 proposes the use of an independent recovery processor to offload some of the recovery operations from the main processor, but this processor does not take part in regular query processing tasks.
- The computer is equipped with memory mapping hardware that maps logical addresses to physical addresses.
- The hardware supports the abstraction of segments that are composed of fixed-size pages (referred to here as *partitions*), and it provides access rights checking, bounds checking, and address mapping. This is similar to the paged segments of Multics [Peter 82], but without paging (in the disk sense).

2.4. Memory-Resident Data Organization

In this section we describe the structure of segments and partitions, which are the basic units of addressing and memory allocation used to manage all database objects; the structure of relations, temporary relations, and indices; and the database catalogs that describe the contents of relations and indices.

2.4.1. Segments and Partitions

Direct addressability of database entities implies that close physical proximity of tuples is not important for retrieval performance (indeed, the tuples of a relation *could* be scattered across all of memory). However, other components of the database system do require tuples (and index components) to be organized in a clustered, modular design. For memory management reasons, relations and indices each have their own separate memory space, which allows them to be created, reorganized, or destroyed without affecting other relations or indices. And, as will be seen in Chapter 5, checkpoint operations write to disk portions of relations or indices; having tuples arranged in a non-clustered organization would make the checkpoint procedure overly complex.

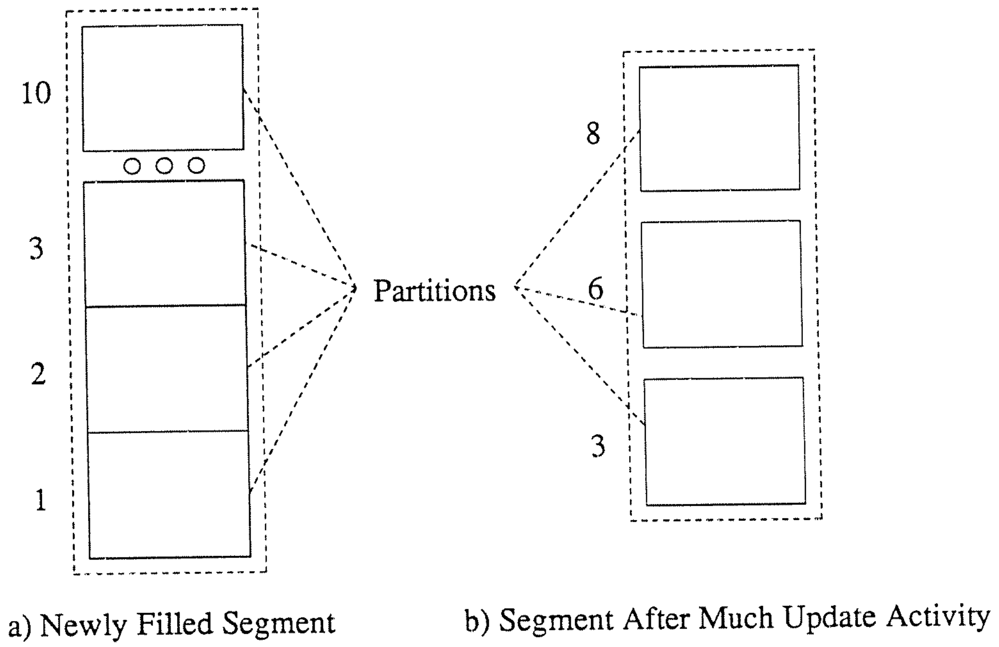


Figure 2.1 — Segment Structure

Every database object (relation, index, or system data structure) is stored in its own logical segment. Segments are composed of fixed-size partitions, which are the unit of memory allocation for the underlying memory mapping hardware. (We use the word partition rather than page to avoid any preconceived notions about the uses of a partition.) Partitions represent a complete unit of storage; database entities are stored in partitions and do not cross partition boundaries.[2] Partitions are also used as the unit of transfer to disk in checkpoint operations. Optimal partition size is determined by many factors: storage efficiency, underlying partition table support, and checkpointing efficiency. Thus, picking an optimal size involves dealing with a list of tradeoffs. A large partition size could lead to excessive internal memory fragmentation, or it could cause too much clean data to be written in a checkpoint operation. On the other hand, a small partition size could cause a large number of partitions to be needed for an object, thereby creating unnecessary space overhead for the catalogs that hold partition information and for the underlying memory mapping hardware.

Partitions are addressed as logical offsets from the beginning of their segment (Figure 2.1). After creation, a segment would comprise a set of contiguous partitions (Figure 2.1a). As entities in the segment are deleted and possibly reorganized, some partitions might be released, thereby creating "holes" in the segment. Logically contiguous partitions are not needed to prevent invalid memory addresses from being detected, as they are caught by underlying memory mapping hardware using a segment's partition table. Creating a new partition for a segment involves creating a

[2]For simplicity we assume that segments, and therefore partitions, contain only one database object. Although a more complex scheme allowing multiple objects to be contained in a single partition is possible, it would not significantly change any of the algorithms presented in this dissertation.

new entry in the segment's partition table and allocating physical memory for the new partition. The relation catalog maintains an entry for each partition in a relation, so new valid partition addresses can be created using this list.

2.4.2. Relations

As mentioned in the previous section, relation partitions are self-contained units of storage that hold tuples of relations, and tuples do not cross partition boundaries. A relation partition holds control information, a tuple list that grows bottom up, and a string space heap that grows top down (Figure 2.2). The control information describes the state of the partition, including a count of the amount of free space in the string space heap, a pointer to a list of free tuple slots, a count of used tuple slots, a count of used string space bytes, and a latch control block that restricts access to a partition

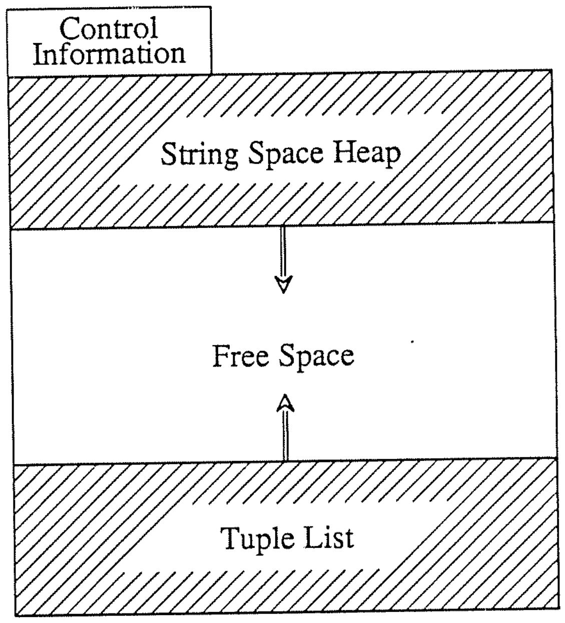


Figure 2.2 — Relation Partition Structure

during update operations. (More details of the latch mechanism will be presented Chapter 6.)

Since memory addresses are used as tuple identifiers, tuples should change locations as infrequently as possible. Therefore, tuples are made fixed length so that newly inserted tuples can easily fill holes left by previously deleted tuples, thus avoiding the need for reorganization of the tuple list to reclaim variable-size blocks of memory. Variable-length strings are stored separately in the string space heap, and offsets of strings are stored in the fixed-length tuples. Although the next available free tuple slot is used to hold a newly inserted tuple during normal processing, it may be necessary to allocate a specific free slot during recovery; therefore, the list of free tuple slots is managed as a doubly linked list. Tuples contain control information describing their tuple slot status (valid tuple or deleted tuple) and possible null field values. Since there must be room for a status flag and two pointers for maintaining the doubly-linked free list, tuples have a minimum size requirement equal to the size of the flag plus the size of the two list pointers. Tuple length and field offset information do not need to be stored with the individual tuples, as this information is constant for a given relation. It is therefore stored only once, in the relation's catalog entry.

Since holes caused by deletes are allowed in the tuple space of relation partitions, direct traversal would require additional structure to link valid tuples in a chain. Instead, direct traversal of a relation partition is not allowed; all access to a relation is done though an index and all relations are required to have at least one index.

2.4.3. Temporary Relations

Temporary relations hold intermediate and final query results. Most query results are composed of information that already exists in the database, but some queries compute new values — queries containing aggregate operations, for example. When there are no computed values, temporary relations comprise a list of tuple pointers and a result descriptor that names the fields belonging in the temporary relation. (Each result tuple may hold several tuple pointers, each corresponding to a separate tuple with fields participating in the final result tuple.) The descriptor takes the place of projection; field extraction is not needed. (This form of temporary relation has a structure similar to that of the "markings" of the Frame Memory secondary memory management system described in[March 81].)

When a temporary relation does contain computed values, its structure appears almost identical to that of a regular relation, except for the added descriptor. Only in rare cases will temporary relation partitions contain string spaces, as that would require a query that used an aggregate operation to create new variable length strings. Unlike regular relations, a temporary relation can be traversed directly because there are no holes in the tuple list; however, it is also possible to have indices on a temporary relation.

As an example of a typical temporary relation, suppose the Employee and Department relations of Figure 2.3 were joined on their Department Id fields (Query 2.1). Each resulting tuple in the temporary list would contain a pair of tuple pointers, one pointing to an Employee tuple and one pointing to a Department tuple, and the result descriptor would list the fields in each relation that appear in the result.

Query 2.1: Retrieve the Employee name, Employee age, and Department name for all employees under age 65.

Employee Relation				
Ptr	*Name*	*Id*	*Age*	*Dept_id*
124	Dave	23	24	459
105	Suzan	12	27	459
137	Yaman	44	54	411
110	Jane	40	47	411
102	Cindy	22	22	409
133	Toby	49	69	455

Department Relation		
Ptr	*Name*	*Id*
243	Toy	459
201	Shoe	409
213	Linen	411
287	Paint	455

Employee Indices	
Name	*Id*
102	105
124	102
110	124
105	110
133	137
137	133

Result Relation
(124, 243)
(105, 243)
(137, 213)
(110, 213)
(102, 201)

Result Descriptor
Emp.Name
Emp.Age
Dept.Name

Figure 2.3 — Relation and Index Design

2.4.4. Indices

Unlike relation partitions, index partitions have no database entities that are referenced from outside their segment. Index partitions are simply memory heaps from which index nodes, index tables and other miscellaneous index components are allocated; thus, reorganization of an index segment affects only itself. This places a constraint on index structure design, as each index structure component must fit in a single partition. Fortunately, this is not a problem for any of the main memory index structures considered here.

Recall that indices hold only tuple pointers, as tuple pointers provide access to both the heads and fields of tuples. Figure 2.3 shows an example of two indices built for the Employee relation. (The indices are shown as sorted tables for simplicity.) If the tuple pointers used in index structures actually pointed to beginnings of tuples, then each compare operation in the index routines would require an offset calculation to locate the key field. Instead, the tuple pointers used in indices point directly to the key field (or to the first key field in the case of a multi-attribute index), thereby eliminating this offset calculation to locate the key field, while necessitating a negative offset calculation to locate the head of each tuple retrieved in the search. However, we assume that the number of tuples referenced during search operations will be greater than the number of tuples retrieved, on the average. Even in the case of multi-attribute indices, avoiding the cost of computing the offset of the first field is a significant savings, as most compares involve *only* the first field — other fields are compared only when the value of the first field is equal to the search value.

Indices can be keyed on many different types and formats of data. For example, the tuple pointers of an index could directly reference integers, floating point numbers, fixed-length bit or character strings, or fixed-length user-defined values; they could reference tuple fields that hold offsets to variable-length bit strings, variable-length character strings, or variable-length user defined values; or they could reference fields that contain tuple pointers that in turn reference one of the above-mentioned types of values or even other tuple pointers! If a compare routine had to determine which compare method to use for each compare, it could easily spend more time deciding on the proper compare method than doing the compare. Instead of using a general purpose compare routine, each index has a compiled compare routine associated with it that is dynamically linked with the general index code when the index is used. The compare routine for an index is compiled and stored in the index's catalog entry when the index is created, and the general index code contains a jump to the index-specific compare routine (specified at runtime) instead of a general compare routine. Some computer architectures support this type of operation directly, such as the IBM 370 architecture's branch and link instruction [IBM]. This compiled compare routine approach also allows users to supply their own types and compare routines, thereby making the type structure of the database extensible [Stone 86]. Similar compare routines can be generated for join and project operations.

2.4.5. Links

A link is a tuple pointer that is stored as a field value in a tuple.[3] It essentially joins a pair of tuples — the tuple that holds the tuple pointer, and the tuple that is pointed to. Links provide a method for pre-computing join operations and storing the results in the original relations.[4] For example, Figure 2.4 shows how the relations of Figure 2.3 would be joined with a uni-directional link from the Employee relation to the Department relation.

Only single tuple pointers are stored in fields, so only one-to-one and many-to-one relationships are supported. (Figure 2.4 shows a many-to-one relationship between the Employee relation and the Department relation.) Many-to-many relationships are

[3]The idea of links is not new — they are used even in disk systems. However, because of their random access nature, they are much more useful when used in a main memory environment.

[4]Space for links in a relation is allocated statically when the relation is created. A dynamic scheme would have significant potential performance implications.

Employee Relation				
Ptr	*Name*	*Id*	*Age*	*Dept_Link*
124	Dave	23	24	243
105	Suzan	12	27	243
137	Yaman	44	54	213
110	Jane	43	47	213
102	Cindy	22	22	201

Department Relation		
Ptr	*Name*	*Id*
243	Toy	459
201	Shoe	409
213	Linen	411
287	Paint	455

Figure 2.4 — Link Example

possible, but storing multiple tuple pointers in a tuple (*i.e.* a string of tuple pointers) would be difficult; complex situations would arise if this string of tuple pointers grew to be larger than a single partition.

Links are to be maintained by the database system, and they will always consist of valid tuple addresses or null values. When a tuple that is referenced by a link is relocated, that link will be found and updated to point to the tuple's new location; if the tuple is deleted, the link is given a null value. There is an interesting relationship between links and foreign keys. Foreign keys, as defined by Date [Date 85], are attributes that uniquely identify tuples in other relations (and tuple pointers certainly qualify as unique identifiers). Relational referential integrity states that a foreign key must point to a valid tuple or it must be null. Thus, links appear to be an excellent mechanism for maintaining referential integrity, and could be extended to provide the sort of support that Date proposes [Date 85].

Since links are uni-directional because of the one-to-one / many-to-one relationship restriction, it could be difficult to locate all of the holders of links for a tuple when the tuple moves. Fortunately, a solution to this problem exists. It is possible to create indices on link fields of relations, using tuple pointers as actual key values, thus providing access to the holders of links for a given link value (*i.e.* tuple address). Of course, it is also necessary to maintain a list of all relations participating in link relationships in order to find these indices. This can be done by recording the pairs of relations that participate in link relationships in a Link Catalog. (And, for performance reasons, the Link Catalog itself is indexed both on holders of links and those referenced by links.) As an example of how the links for a particular tuple can be found, suppose tuple T_a of relation R_a moves. First, the Link Catalog is searched (*via* the index) to find all relations that hold links to relation R_a. Then, for each such relation, the link index corresponding to relation R_a is searched for tuple T_a's address; the appropriate link field of each tuple found is updated with tuple T_a's new address.

2.4.6. Database Catalogs

Database catalogs provide information about the structure of the database including relation-specific, attribute-specific and index-specific information. A general description of the functions of database catalogs can be found in [Date 85]. In addition to the usual catalog functions, the catalogs of our memory-resident database system provide information specific to its memory-resident aspects: link catalog entries contain information about pairs of relations that participate in link relationships; relation catalog entries contain partition disk location information; and index catalog entries hold index compare routine code information. (These items are explained in more detail in later chapters.)

Chapter 3

Index Structures

3.1. Main Memory Index Structures

Index structures designed for main memory are different from those designed for disk-based systems. The primary goals for a disk-oriented index structure are to minimize the number of disk accesses and to minimize disk space. A main-memory-oriented index structure is contained in main memory, hence there are no disk accesses to minimize. Thus, the primary goals of a main-memory-index structure are to reduce overall computation time while using as little memory as possible.

There are many data structures available for consideration as main-memory-index structures. There are two main types: those that preserve some natural ordering in the data and those that randomize the data. The index structures studied here are: arrays, AVL Trees, and B Trees, for the order-preserving class[5]; and Chained Bucket Hashing, Linear Hashing and Extendible Hashing, for the randomizing class[6]. We describe briefly each algorithm in turn. Then, having considered these algorithms, we introduce a new algorithm called the T Tree. The T Tree is an order-preserving tree structure

[5]Radix structures provide an excellent method for maintaining ordered data *for some special types of data* [Knuth 73, Willa 84], but since they are not general enough to be used for all data they are not considered here.

[6]There were several dynamic hashing algorithms to choose from. Extendible Hashing and Linear Hashing were chosen because of their popularity and the fact that most other methods are based on variants of these methods. Methods using spiral storage [Kjell 84] or partial expansions [Larso 80] require more data reorganization [Mulli 84], so they would not do as well in a dynamic main memory environment.

designed specifically for use in main memory.

3.1.1. Existing Index Structures

Arrays are used as index structures in IBM's OBE project [Amman 85]. They use minimal space, providing that the size is known in advance or that growth is not a problem (e.g., that one can somehow use the underlying mapping hardware of virtual memory to make the array grow gracefully). The biggest drawback of the array is that the amount of data movement is $O(N)$ for each update, so it appears impractical for anything but a read-only environment.

AVL Trees [Aho 74] are used as indices in the AT&T Bell Laboratories Silicon Database Machine [Lelan 85]. The AVL Tree was designed as an internal memory data structure (Figure 3.1). It uses a binary tree search, which is very fast since the binary search is *intrinsic* to the tree structure — no arithmetic calculations are needed. Updates always affect a leaf node and may result in an unbalanced tree, so the tree is kept balanced by rotation operations. The AVL Tree has one major disadvantage, which is its poor storage utilization. Each tree node holds only one data item, so there are two pointers and some control information for every data item.

B Trees [Comer 79] are well suited for disk use since they are broad, shallow trees and require few node accesses to retrieve a value (Figure 3.1). Most database systems use a variant of the B Tree, the B+ Tree, which keeps all of the actual data in the leaves of the tree. For main memory use, however, the B Tree is preferable to the B+ Tree because there is no real performance advantage to keeping all of the data in the leaves, and doing so requires more space. B Trees are good for memory use since their storage utilization is good (the pointer to data ratio is small, as leaf nodes hold only data items and they comprise a large percentage of the tree); searching is reason-

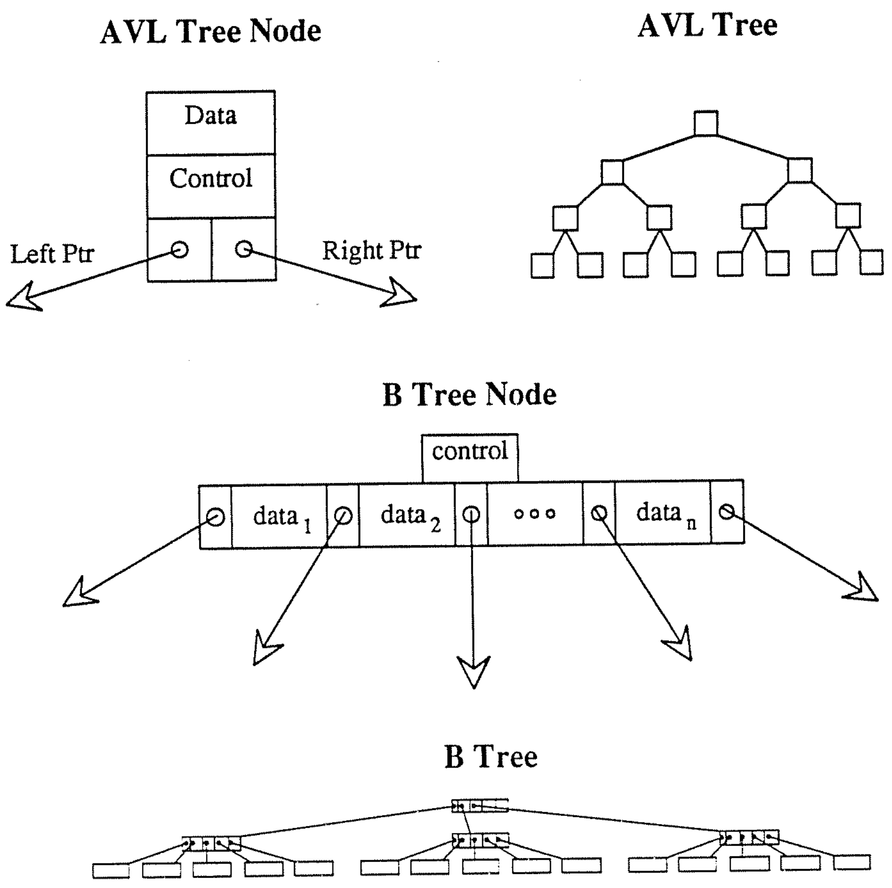


Figure 3.1— Tree Structured Indices

ably quick (a small number of nodes are searched with a binary search); and updating is fast (data movement usually involves only one node).

Chained Bucket Hashing [Knuth 73] is a static structure used both in memory and on disk (Figure 3.2). It is very fast because it is a static structure — it never has to reorganize its data. But, this static quality also carries a disadvantage; because it is static, it may have very poor behavior in a dynamic environment because the size of the hash table must be known or guessed at before the table is filled. If the estimated size is too small, then performance can be poor; if the estimated size is too large, then much space is wasted. At best, there is some space wasted, since each data item has a pointer associated with it.[7]

Extendible Hashing [Fagin 79] employs a dynamic hash table that grows with the data, so the table size does not need to be known in advance (Figure 3.2). A hash node contains several items and splits into two nodes when an overflow occurs. The directory grows in powers of two, doubling whenever a node overflows and has reached the maximum depth for a particular directory size. One problem with Extendible Hashing is that *any* node can cause the directory to split, so the directory can grow to be very large if the hash function is not sufficiently random.

Linear Hashing [Litwi 80] also uses a dynamic hash table, but it is quite different from Extendible Hashing (Figure 3.2). A Linear Hash table grows linearly, as it splits nodes in predefined linear order — as opposed to Extendible Hashing, which splits the nodes that overflow. The decision to split a node and extend the directory in a controlled fashion can be based on criteria other than overflowing nodes, providing several advantages over the uncontrolled splitting of Extendible Hashing. First, the buckets can be ordered sequentially, allowing the bucket address to be calculated from a base address — no directory is needed. Second, the event that triggers a node split can be based on storage utilization, keeping the storage cost constant for a given number of elements.

[7] By grouping several items in a node, the data to pointer ratio could be reduced, but the extra data space might be wasted. Our tests showed that the best table size (*i.e.* the table size that provided near optimal storage utilization *and* performance) for a given number of elements has an average of only two items per hash value.

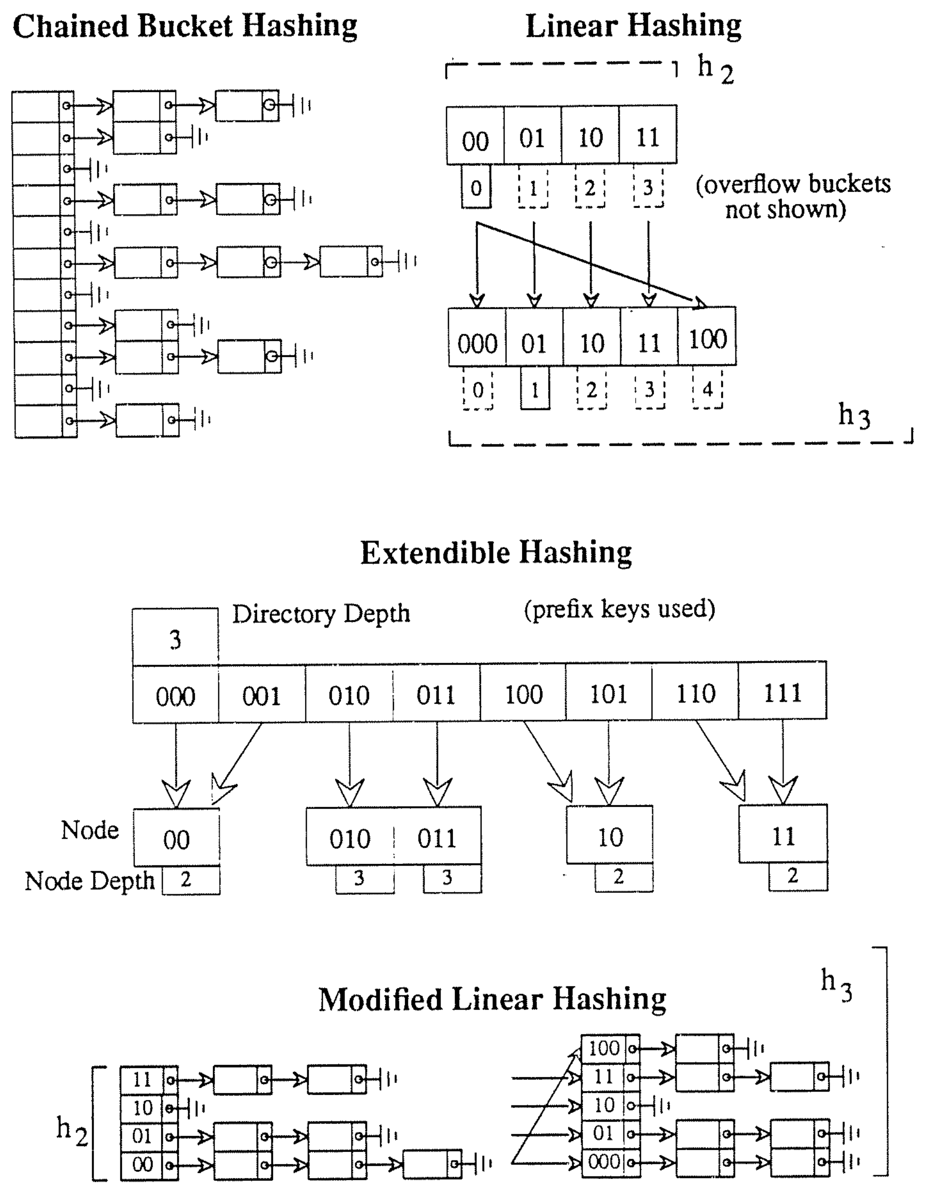


Figure 3.2 — Hashing-based Indices

Modified Linear Hashing is oriented more towards main memory than the regular version mentioned above (Figure 3.2). By using larger contiguous nodes rather than a directory, normal Linear Hashing can waste space with empty nodes when a node corresponding to a hash entry has no data items. Also, unless a clever scheme can be worked out with the underlying virtual memory mapping mechanism, the contiguous nodes have to be copied into a larger memory block when the hash table grows. Modified Linear Hashing uses a directory much like Extendible Hashing, except that it grows linearly, and it uses chained single-item nodes that are allocated from a general memory pool. (Unlike Chained Bucket Hashing, there are typically many items per hash value, so multiple item nodes could be used here to increase storage utilization.) The splitting criteria is based on performance, *i.e.* the average length of the hash chains, rather than storage utilization. Monitoring average hash chain length provides more direct control over the average search and update times than monitoring storage utilization. (Litwin's description of Linear Hashing does not exclude this style of Linear Hashing, but his description is generally targeted more towards disk applications [Litwi 80].)

3.1.2. The T Tree

The T Tree, a new data structure, evolved from AVL Trees and B Trees. The T Tree is a binary tree with many elements in a node. Figure 3.3 shows a T Tree and a node of a T Tree, called a T Node. Since the T Tree is a binary tree, it retains the intrinsic binary search nature of the AVL Tree. Also, because a T node contains many elements, the T Tree has the good update and storage characteristics of the B Tree. Data movement is required for insertion and deletion, but it is usually needed only within a single node. Rebalancing is done using rotations similar to those of the AVL Tree, but it is done much less often than in an AVL Tree due to the possibility of intra-node data movement.

To aid in our discussion of T Trees, we begin by introducing some helpful terminology. There are three different types of T-nodes. A T-node that has two subtrees is

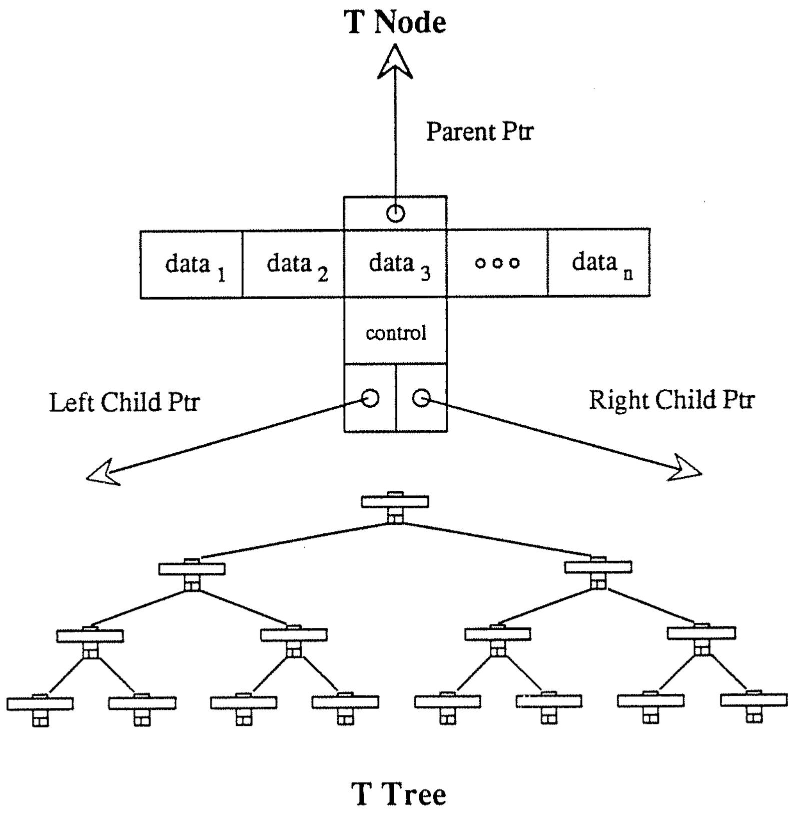


Figure 3.3 — The T Tree

called an *internal node.* A T-node that has one NIL child pointer and one non-NIL child pointer is called a *half-leaf node.* A T-node that has two NIL child pointers is called a *leaf node.* For each internal node A, there is a corresponding leaf (or half-leaf) that holds the data value that is the predecessor to the minimum value in A, and there is also a leaf (or half-leaf) that holds the successor to the maximum value in A. The predecessor value is called the *greatest lower bound* of the internal node A, and the successor value is called the *least upper bound* of A, as shown in Figure 3.4. For a node N and a value X, if X lies between the minimum element of N and the maximum element of N (inclusive), then we say that node N *bounds* the value X. Since the data in a T-node is kept in sorted order, its leftmost element is the smallest element in the

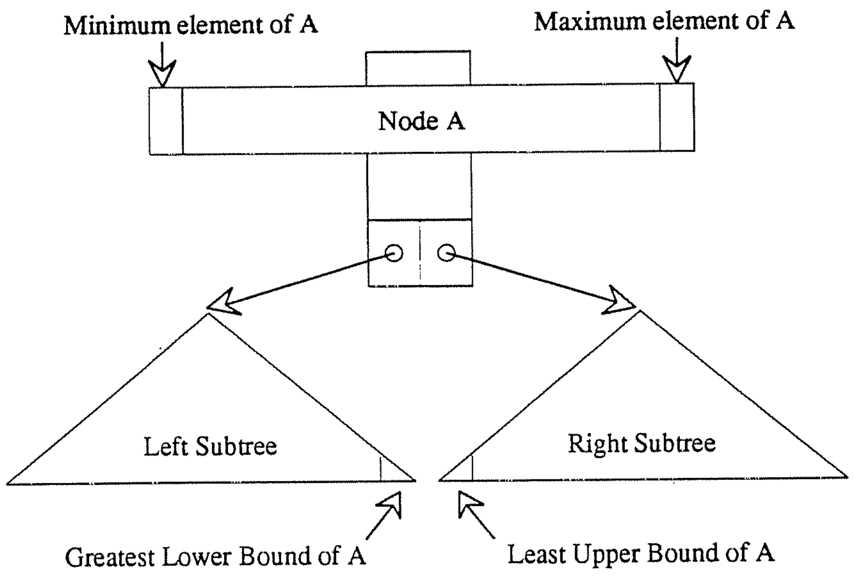


Figure 3.4 — The Bounds of a T-node

node and its rightmost element is the largest.

Associated with a T Tree is a minimum count and a maximum count. Internal nodes keep their occupancy (*i.e.* the number of data items in the node) in this range. The minimum and maximum counts will usually differ by just a small amount, on the order of one or two items, which turns out to be enough to significantly reduce the need for tree rotations. With a mix of inserts and deletes, this little bit of extra room reduces the amount of data passed down to leaves due to insert overflows, and it also reduces the amount of data taken from leaves due to delete underflows. Thus, having flexibility in the occupancy of internal nodes allows storage utilization and insert/delete time to be traded off to some extent. Leaf nodes and half-leaf nodes have an occupancy ranging from zero to the maximum count.

3.1.2.1. Search and Update Operations for T Trees

Search Algorithm

Searching in a T Tree is similar to searching in a binary tree. The main difference is that comparisons are made with the minimum and maximum values of the node rather than a single value as in a binary tree node. The algorithm works as follows:

(1) Start at the root of the tree.

(2) If the search value is less than the minimum value of the node, then search down the subtree pointed to by the left-child pointer. Else, if the search value is greater than the maximum value of the node, then search down the subtree pointed to by the right-child pointer. Else, search the current node.

The search fails when a node is searched and the item is not found or when a node that bounds the search value cannot be found.

Insert Algorithm

The insert operation begins with a search to locate the bounding node. If the insert operation results in the creation of a leaf node, then the tree is checked for balance. Insertion works as follows, although we postpone our discussion of the rebalancing rotations until Section 3.1.2.2:

(1) Search for the bounding node.

(2) If a node is found, then check for room for another entry. If the insert value will fit, then insert it into this node and stop. Else, remove the minimum element from the node, insert the original insert value, and make the minimum element the new insert value. Proceed from here directly to the leaf containing the greatest lower bound for the node holding the original insert value. The

minimum element (the new insert value) will be inserted into this leaf, becoming the new greatest lower bound value for the node holding the insert value.

(3) If the search exhausts the tree and no node bounds the insert value, then insert the value into the last node on the search path (which is a leaf or a half-leaf). If the insert value fits, then it becomes the new minimum or maximum value for the node. Otherwise, create a new leaf (so the insert value becomes the first element in the new leaf).

(4) If a new leaf was added, then check the tree for balance by following the path from the leaf to the root. For each node in the search path (going from leaf to root), if the two subtrees of a node differ in height by more than one level, then a rotation must be performed (see Section 3.2.2). Once one rotation has been done, the tree is rebalanced and processing stops.

A design note is in order here: When an internal node overflows, and thus its minimum value is removed and passed down to a leaf, the insert into the leaf requires no data movement because the value becomes the leaf's rightmost entry. If instead the maximum value had been removed from the internal node, it would have to be inserted as the leftmost entry in the leaf, requiring intra-node data movement. Hence, removing the minimum value instead of the maximum value avoids this data movement.

Delete Algorithm

The deletion algorithm is similar to the insertion algorithm in the sense that the element to be deleted is searched for, the operation is performed, and then rebalancing is done if necessary. The algorithm works as follows:

(1) Search for the node that bounds the delete value. Search for the delete value within this node, reporting an error and stopping if it is not found.

(2) (A) If the delete will not cause an underflow (*i.e.* if the node has more than the minimum allowable number of entries prior to the delete), then simply delete the value and stop.

(B) Else, if this is an internal node, then delete the value and take the greatest lower bound of this node from a leaf or half-leaf to bring this node's element count back up to the minimum.

(C) Else, this is a leaf or a half-leaf, so delete the element. (Leaves are permitted to underflow, and half-leaves are handled in step (3).)

(3) If the node is a half-leaf and can be merged with a leaf (*i.e.* the combined total number of elements of the half-leaf and the leaf are less than or equal to the maximum amount), then coalesce the two nodes into one node (a leaf), discard the other node, and proceed to step (5); otherwise, proceed to step (4).

(4) If the current node (a leaf or half-leaf) is not empty, then stop. Else, free the node and proceed to step (5) to rebalance the tree.

(5) For every node along the path from the leaf up to the root, if the two subtrees of the node differ in height by more than one, perform a rotation operation (see Section 3.2.2). Since a rotation at one node may create an imbalance for a node higher up in the tree in the case of deletion, balance-checking for deletion must examine all of the nodes on the search path until a node of even balance is discovered.

3.1.2.2. Rebalancing a T Tree

The rebalancing operations for a T Tree are similar to those for an AVL tree [Aho 74]. A T Tree's balance is checked whenever a leaf is added or deleted, as indicated in the descriptions of the insertion and deletion algorithms. The search path is

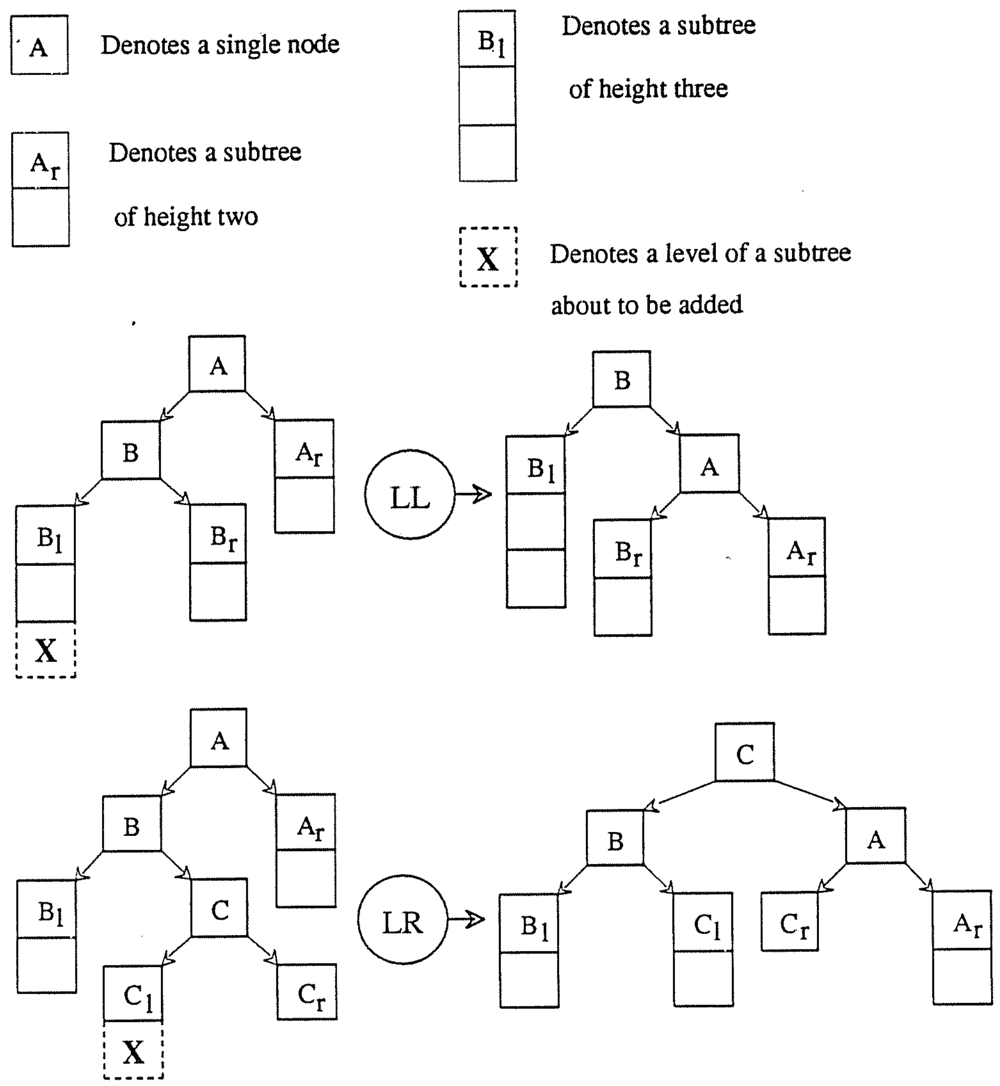


Figure 3.5 — T Tree Rebalancing Operations

checked from the leaf to the root — for each node in the path, if the node's two subtrees differ in height by more than one level, a rotation operation is needed. In the case of an insertion, at most one rotation is needed to rebalance the tree, so processing stops after one rotation. In the case of a deletion, a rotation on one node may trigger an imbalance for a node higher up in the tree, so processing continues after a rotation until an evenly balanced node is found. (When a rotation operation rebalances a subtree, the longer side of the subtree is made shorter to match the rest of the subtree. In the case of insert, this brings the subtree back into the same height as before the insert, so only one rotation is needed. In the case of delete, the subtree height is reduced, so it may trigger further rotations.) Figure 3.5 shows the simple LL rotation and the more complex LR rotation for the case of an insert. These are two of the four types of rotations used to rebalance an AVL tree or a T Tree. The algorithms for the RR and RL rotations are symmetrical to the LL and LR rotations, respectively, so they are not shown.[8] The rebalancing rotations for deletion are identical to those for insertion, except that the cause of the imbalance in the tree is that a subtree has grown shorter rather than longer and that more than one rotation may be needed to balance the tree.

The T Tree requires one special-case rotation that the AVL tree does not have. When an LR or RL rotation is done and the node C is a leaf (and both nodes A and B are half-leaves), a regular rotation would move C into an internal node position, as shown in Figure 3.6. If C has only one item, which is always the case during an insert, then C would *never* get to hold more than a single item as an internal node (unless it is later rotated back into a leaf position) because items are always inserted *between* the lower and upper bounds of an internal node. Since this would be detrimental to storage utilization, a special rotation operation first moves values from B to C so that, after the rotation, C is a full internal node. This special case rotation is shown in Figure 3.6.

[8] The names of the rotations (LL, RL, LR, and RR) are derived from the child of the node that causes the imbalance. In the LL rotation in Figure 3.5, the Left Left child of A is longer; in the LR rotation, the Left Right child is longer.

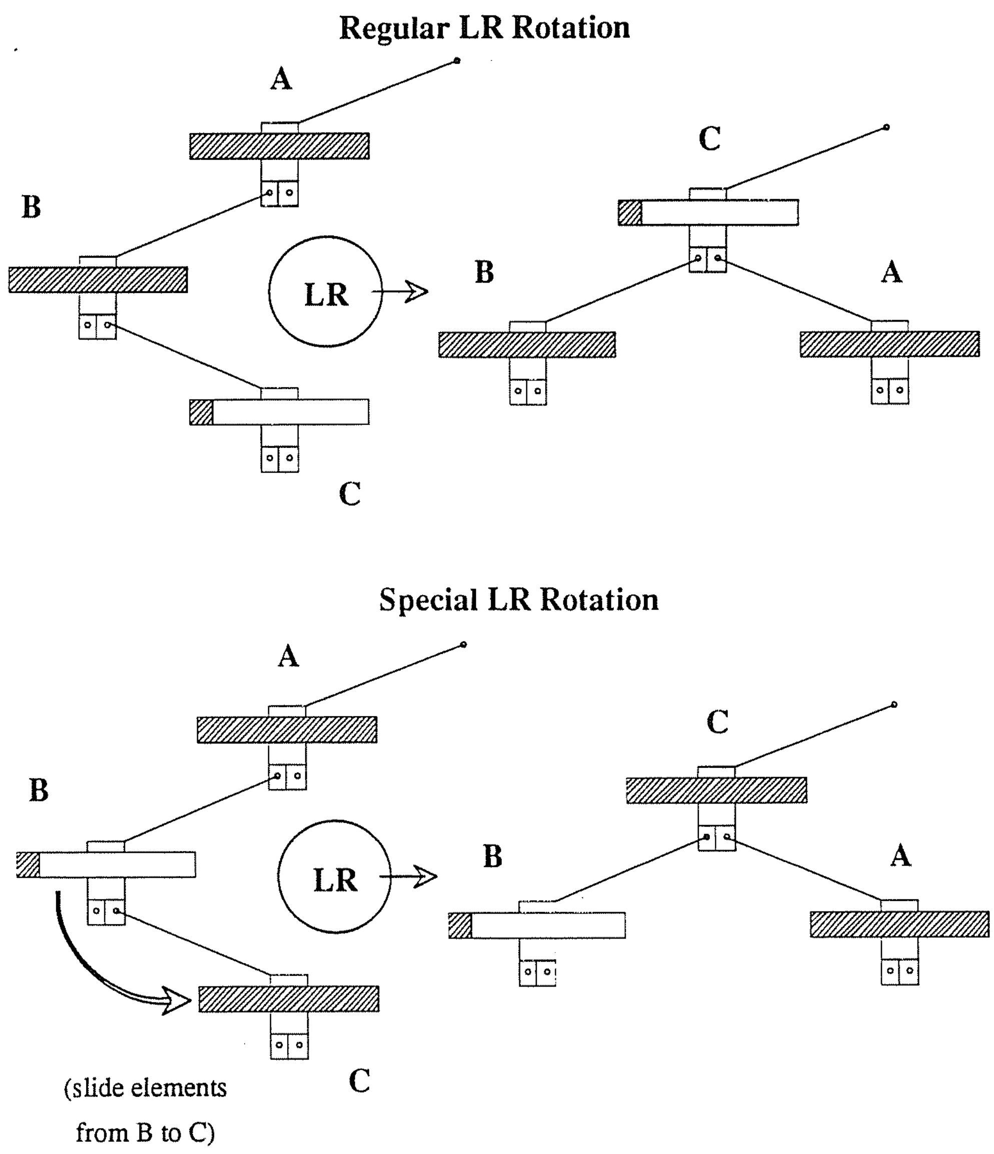


Figure 3.6 — Special T Tree Rebalancing Operations

3.1.2.3. Additional Properties of the T Tree

T-nodes are equipped with child and parent pointers so that they can be traversed in either direction. From any node in a T Tree, all of the other nodes can be retrieved — in order. This property is useful when using the T Tree to perform a merge join, as described in Chapter 4. This property also allows the T Tree to store and retrieve duplicates with no extra work. Duplicate values (if allowed) are simply stored in logically adjacent locations. The update and rotation operations work as before. To retrieve a set of tuple pointers corresponding to a single value, one simply searches for the value, marks the spot where the first such item is found, and then scans left and then right from that spot.

3.1.3. Modifications to the T Tree

Observations on the T Tree algorithms have motivated the development of two modifications to the T Tree index structure and its algorithms. One modification involves a change in the search algorithm, and the other involves a significant change in the structure and its search and update algorithms.

3.1.3.1. A Different Search Method

The motivation for the alternative search method comes from the observation that a T Tree search operation often performs two comparisons per T Node. During the search for a bounding T Node, the search operation compares the search key and the minimum element of each node in the search path. When the search key is less than the minimum element of a T Node, the search proceeds down the left subtree, otherwise an additional compare involving the maximum element is needed to decide between searching the T Node or proceeding down the right subtree. When the number of elements in a single node is small compared to the number of elements in the tree, half of the compares performed on the minimum values of T Nodes will have *less than* results. Thus, half of the nodes in the search path will require one compare operation and the other half will require two compare operations.

Ideally, it would be preferable to perform only one compare per node, as performing two compares per node appears to "waste" a compare. Consider a T Tree comprising three T Tree nodes labeled A, B, and C, and a search value of 21 (Figure 3.7). Comparing the search value (21) and maximum value of node B (31) shows that node B's right subtree cannot contain the search value, and that either node B or node B's left subtree *might* hold the search value. Testing node B's minimum value (15) does not provide any more information than testing A's maximum value (13), as either test shows that B's left subtree, node A, does not hold the search value and that node B is the only possible bounding node; hence, node B is searched (and the search value is found).

Generalizing this example, one can draw one of three possible conclusions after both the minimum amd maximum values of a T Node have been tested.

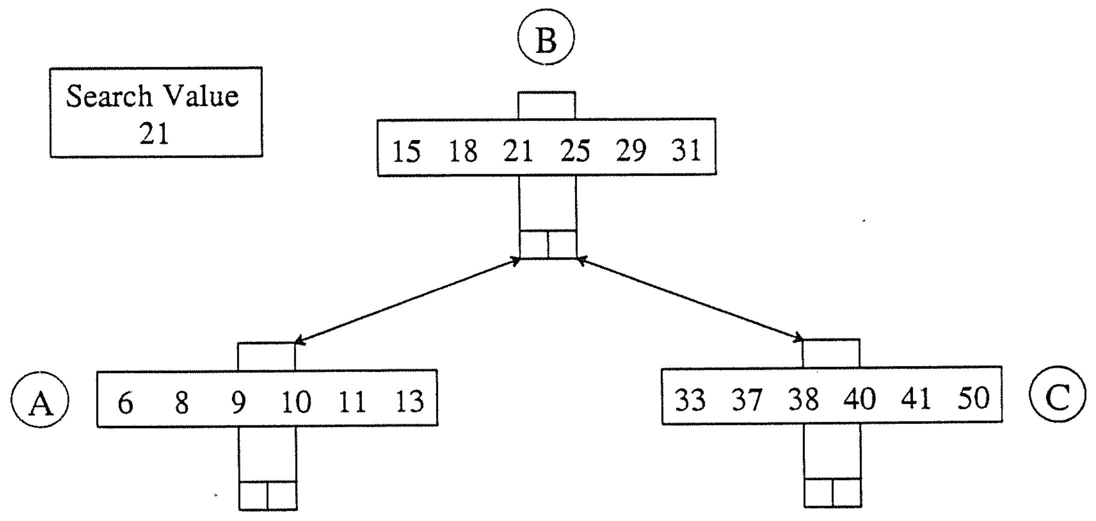


Figure 3.7 — Testing Adjacent Nodes

(1) The left subtree may contain the search value
(search value $<$ min value)

(2) The right subtree may contain the search value
(search value $>$ max value)

(3) The current node may hold the search value and should be searched
(min value $\leq$ search value $\leq$ max value)

With these three possibilities, one knows exactly which set of nodes can contain the search value. When only the maximum value of a T node is tested, one can draw one of two possible conclusions:

(1) The current node *or* the left subtree may contain the search value
(search value $\leq$ max value)

(2) The right subtree may contain the search value
(search value $>$ max value)

Testing only the maximum value makes the decision process more difficult because in the case of a less than or equal compare result (case 1), it is not clear whether the current node should be searched or the left subtree should be traversed. It would be inefficient to search the current node each time case (1) arose, since only one node can contain the search value (assuming for now that only unique values are stored in the index). How, then, does one figure out which node to search when only one compare is used? Looking at the example in Figure 3.7 again, we see that node B wasn't searched until node A's maximum value was tested and found to be less than the search value. In general, the node that needs to be searched is the last node that had a maximum value greater than the search value. Figure 3.8 shows how a search might pass through a tree, marking nodes where the search value was less than the

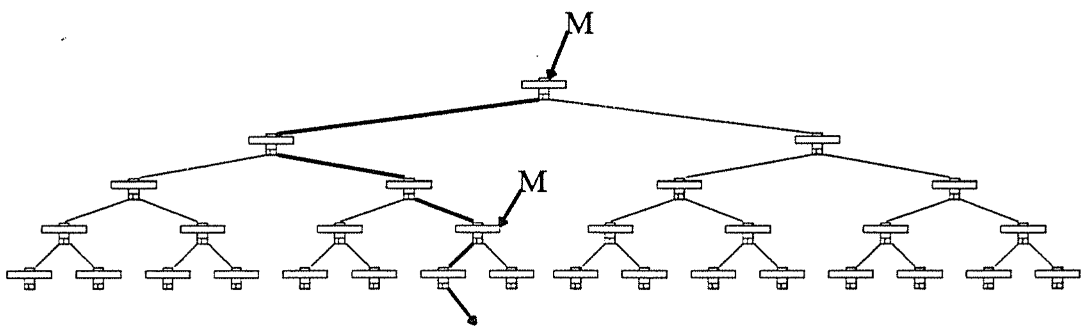


Figure 3.8 — Marking Left Turns in the Search Path

maximum value of a node.

The alternative search method works as follows:

(1) Start at the root of the tree.

(2) If the search value is greater than the maximum value, then search the right subtree, otherwise mark this node and search the left subtree.

(3) When the search reaches a leaf, the leaf and the last marked node are remembered. (It is possible for the leaf and the last marked node to be the same node; this will occur whenever the search value is less than or equal to the maximum value of the leaf.)

(4) The last marked node is searched with a binary search.

This alternative search method possesses an interesting quality — the search always terminates with pointers to the two important nodes for the insert and delete routines. The marked node is the bounding node for the insert/delete value, and the leaf (the last node touched in the search) is the overflow or underflow node for the bounding node. Using a node's maximum value for the compare with the search value is responsible for this, as using its minimum value would result in the tree search terminating with a different (and useless) leaf node.

Referencing a T node's maximum value could be more expensive than referencing the node's minimum value (which is in a well-known place) if the offset of the maximum value had to be computed from the count of items in the node. Instead, a copy of the maximum value of a T node is kept in the rightmost position of the node's data space, so it will be in a well-known location that can be computed at compile time. When a T node is full, its maximum value is in the rightmost location by default. Otherwise, operations that modify the contents of a node always update the node's rightmost maximum value copy with its true maximum value.

The alternative T Tree search operation always ends at a leaf because it does not determine which T node is the bounding node until it reaches a leaf. It may not be obvious at first that the alternative search method performs fewer compares than the regular search method, since the regular search method stops when it finds a bounding node and the alternative search method always proceeds to a leaf node. However, because approximately 1/2 of the nodes of a balanced binary tree are leaf nodes, the regular search method traverses, on the average, the top N - 1 levels of an N level tree. Thus, the regular search method and the alternative search method visit almost the same number of nodes during a tree search, but the alternative search method performs one compare per node visited, and the regular search method performs, on the average, one and one half compares per node visited. Therefore, the alternative search method performs far fewer compares than the regular search method. (This statement is supported by the compare count measurements presented in Section 3.2.6.)

3.1.3.2. The T+ Tree

The second T Tree modification leads to a new tree structure, the T+ Tree (Figure 3.9). The T+ Tree is to the T Tree as the B+ Tree is to the B Tree. As in a B+ Tree, the data in a T+ Tree is kept only in the leaves of the tree, and the upper nodes hold only copies of data (actually copies of tuple pointers); the minimum values of leaf nodes are stored in the upper nodes and are used to guide searches through the upper nodes to the leaf nodes. Since the upper nodes simply provide a search path to the leaf nodes, only a single value is needed in each upper node. Hence, the internal nodes of a T+ Tree actually compose an AVL Tree. The leaves of a T+ Tree are doubly linked for traversal in either direction.

Searching a T+ Tree begins with a tree search through the upper nodes to locate the leaf that bounds the search value. This is similar to an AVL Tree search except that the search always goes down to the leaves, so the compare procedure of a node is simpler — a less than compare result means one takes the left subtree, otherwise one takes the right subtree. When the search reaches a leaf node (a data node), that node is

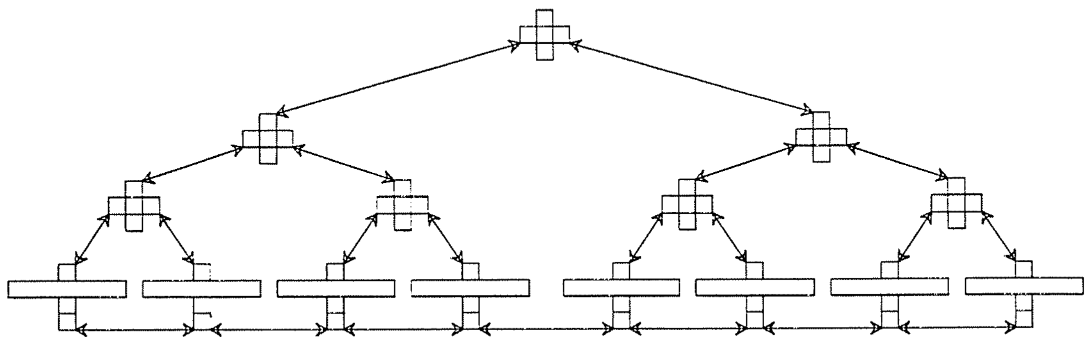


Figure 3.9 — The T+ Tree

searched with a binary search.

The T+ Tree algorithms are still in the development stages. T+ Tree insert and delete operations on leaf nodes appear to be similar to those of the B+ Tree, and its insert and delete operations on internal nodes appear to be identical to those of the AVL Tree. The main interesting feature of the T+ Tree is that at most two leaf nodes appear to be affected by any one update, thus better isolating the effects of updates than the T Tree; isolation of updates is useful for locking and recovery purposes. (This is explained in more detail in Chapters 5 and 6.)

3.2. Index Performance Tests

Each of the index structures described in Section 3.1 took part in the index performance study. The index tests simulated the operations of a normal database management system so that the index structures would be compared in a realistic environment. Each algorithm implementation was done in "main memory style", that is, the indices held pointers to records instead of keys or actual records. To supply the indices with data, a random number generator filled a test relation with unique 4 byte integers that were used as key values for the indices and, since the values could be read out of the relation, the search and delete operations were always performed on existing elements, hence they were always successful. The index size chosen was 30,000 elements, as this was the largest number of elements that most of the implementations could support for the amount of memory available in the test environment (see Section 3.2.3).[9]

[9] Extendible Hashing still ran out of memory in some cases. This was due in part to an inefficient storage allocator and in part to the algorithm itself (as will be explained more fully in Section 3.2.4).

Both the regular T Tree search method and the alternative T Tree search method were tested. To our surprise, the alternative T Tree search method showed slightly *slower* search times then the regular method. Therefore, the regular search method was used for the tests. We provide a partial explanation for the search results in the test result validation section (Section 3.2.6).

3.2.1. Index Test Description

Tests were run in the following order: each data structures was built, then searched, then subjected to three query mixes, then searched with range queries, then scanned, and then half of its values were deleted. We describe each test in more detail below.

Insert 30,000 elements — To measure the insert cost, 30,000 elements were inserted into each index structure. The inserts were each separate (as opposed to a special block insert). The index structures allowed unique values only, so each insert operation involved a search to ensure that the item was not already there.

Search for 30,000 random elements — To measure the retrieval speed of the indices, each index was searched for 30,000 different elements, with each element requiring a new search.

Query mix — Each index structure was tested in a "normal" update environment by performing a mix of inserts, searches, and deletes. Three query mixes were used: the first query mix was composed of 80 percent searches, 10 percent inserts and 10 percent deletes; the second was composed of 60 percent searches, 20 percent inserts and 20 percent deletes; and the third was composed of 40 percent searches, 30 percent inserts and 30 percent deletes. To keep the index structures at a constant size the search, insert, and delete operations were interspersed and the percentages of inserts and deletes were kept equal, thereby making the results easier to evaluate.

Query mix(1) — 24,000 searches, 3000 inserts, 3000 deletes
Query mix(2) — 18,000 searches, 6000 inserts, 6000 deletes
Query mix(3) — 12,000 searches, 9000 inserts, 9000 deletes

Range queries — Order-preserving data structures often need to supply a list of values corresponding to a given range. Three range queries were tested, each one testing different amounts of searching and scanning. The lower and upper bounds of a range known to be 10, 100, or 1000 elements apart were given to each index. The index would search for the lower bound value, then scan the elements in order until it found or passed the upper bound value. Note that these tests are inappropriate for the hashing algorithms because they do not store values in logical order.

Range query 10 — 30,000 queries, retrieving 10 elements per query
Range query 100 — 3,000 queries, retrieving 100 elements per query
Range query 1000 — 300 queries, retrieving 1000 elements per query

Sequential scan — To test the scanning speed of each data structure, each item in the index was read. For the array and tree data structures, the values were read in logical order. For the hashing structures, the values were read in physical order.

Delete 15,000 elements — To get a more realistic delete cost, only half of the index structure was deleted. By deleting only half of the index, the cost of a single delete can be calculated as roughly 1/15,000th of the total time because the data structures were not greatly affected by having half of their elements removed. A T Tree, for example, would become only one level smaller, and a B Tree would probably not even change height. (Removing all of the elements would have given a false impression of the delete cost since the cost per item deleted would be much less for the second half of the elements than the first half.)

Storage costs — The storage cost of each of the data structures after building the index with 30,000 elements is sometimes different from the storage cost after the query mixes, even though the number of elements is the same, so each storage cost value was collected twice — once before the query mixes and once after. The storage cost is measured as the total number of bytes needed to store the given number of elements. The storage cost computations assume that byte (as opposed to word) alignment is permissible, and that the smallest unit of allocation is a byte. (The pointer size for both data and nodes is four bytes.)

3.2.2. Reducing the Number of Test Parameters

Many of the index structures have several variable parameters and, in order to keep the number of tests reasonable, some of these parameters had to be held constant. Preliminary tests on the index structures gave us some indications of reasonable values for the parameters. For each structure, the number of variable parameters was reduced to one, node size (except for Modified Linear Hashing, where the average hash chain length was varied). The restriction to one similar variable allowed the execution times of the index structures to be compared side by side in the same graphs. A discussion of the parameter reduction process follows.

The B Tree has a variable node size, a variable leaf size, and two possible node/leaf search methods: linear and binary search. Preliminary tests showed that varying the leaf and node sizes separately did not significantly affect the results, so they were varied together at the same size. Linear search on small nodes (4 and 6 elements) was 5 percent faster than the best binary search (on any size node), but small nodes cause the tree to use much more space. Since storage and performance are both important, the binary search was used. The T Tree has a variable node size and two possible node-searching methods: linear search and binary search. For the same reasons as for the B Tree, binary search was used for the tests.

Extendible Hashing has a variable node size and three different possible node search techniques: linear search, binary search, and hash search. A hash search in the node would speed up the search time, but it would introduce problems in handling collision resolution. Separate chaining would waste at least half of the node's space on pointers, and open addressing would cause difficulties with delete by wasting either time or space to fill in holes left by deleted items [Knuth 73]. A binary search would have required keeping the node sorted, and therefore would have added data movement cost for insertion and deletion. Linear search seemed to be the simplest and quickest method, so this was used for the tests.

Linear Hashing has a variable node size, a variable overflow bucket size, a variable storage utilization, and three possible node/bucket search techniques: linear search, binary search and hash search. The overflow bucket size could range from a fraction of the node size to several times the node size. The storage, insert and delete costs were found to be better for larger buckets, while the search and query mix costs were better for small buckets. An overflow bucket that was half the size of the node gave reasonable overall performance. The storage utilization threshold parameter (the number of data bytes used divided by the total number of data bytes currently available) could range anywhere from a few percent to close to 100 percent. It is used to trigger a change in directory size — when the storage utilization goes up above the threshold value (usually because of an excess of overflow buckets), then the directory expands. Similarly, when the storage utilization falls below the threshold value (due to too many empty data slots), the directory shrinks. A lower storage utilization threshold parameter value implies that less time needs to be spent on data reorganization, since more space can be wasted; less reorganization cost in turn means higher performance. A value of 70 percent seemed to give the best overall ratio of performance to storage cost. Lastly, for the same reasons as stated for Extendible Hashing, a linear search was used to search the nodes and the overflow buckets.

Chained Bucket Hashing has only one parameter — table size. The preliminary tests showed that a table size equal to half the number of elements gives the best ratio of performance to storage cost. Modified Linear Hashing is very similar to Chained Bucket Hashing, except that its hash table is dynamic. It uses its only parameter, the average length of the hash chains, to determine when to split the next hash chain. When the number of elements increases to the point that the average chain length value $(\frac{\textit{Size Of Table}}{\textit{Number Of Data Elements}})$ is greater than the threshold value, the directory expands, and when this value is less then the threshold value, the directory shrinks. The final set of values chosen for the tests appears in Table 3.1.

Data Structure	*Parameters*
Array	no parameters
AVL Tree	no parameters
B Tree	binary search, variable node size
T Tree	binary search, minimum node size set at two less than the maximum node size, variable (maximum) node size
Chained Bucket Hashing	table size fixed at 50% of element count (at 15,000)
Extendible Hashing	linear search, variable node size
Linear Hashing	linear search, overflow bucket size of half the node size, 70% storage utilization factor, variable node size
Modified Linear Hashing	linear search, variable average hash chain length

Table 3.1 — Parameters Chosen for the Tests

3.2.3. Implementation Details

Each algorithm was implemented in the C programming language and run in single-user mode on a VAX 11/750 with 2 megabytes of real memory (with no virtual memory involved). The timing measurements were taken using "getrusage", a timing facility available in BSD UNIX 4.2. The execution times reflect that of a main memory computer with a processing power of about 0.5 MIPS. Some details about the implementation of the algorithms that might help the reader to more fully understand the results follow:

- A standard hash function was used in all of the hashing implementations.

  (key * P) *mod* table size (where P is a large prime)

  In the cases where the table size was a power of two, the *mod* operation was done by stripping off bits with a bit mask operation.

- Since there was no virtual memory mechanism available, the Linear and Modified Linear Hashing directories had to be reallocated from memory when the table size grew or shrunk. To amortize the cost of this reallocation over several increases in directory size, new directories were allocated with an additional 1000 bytes for future growth. The memory for the array, however, was constant, being allocated once for the duration of the tests.

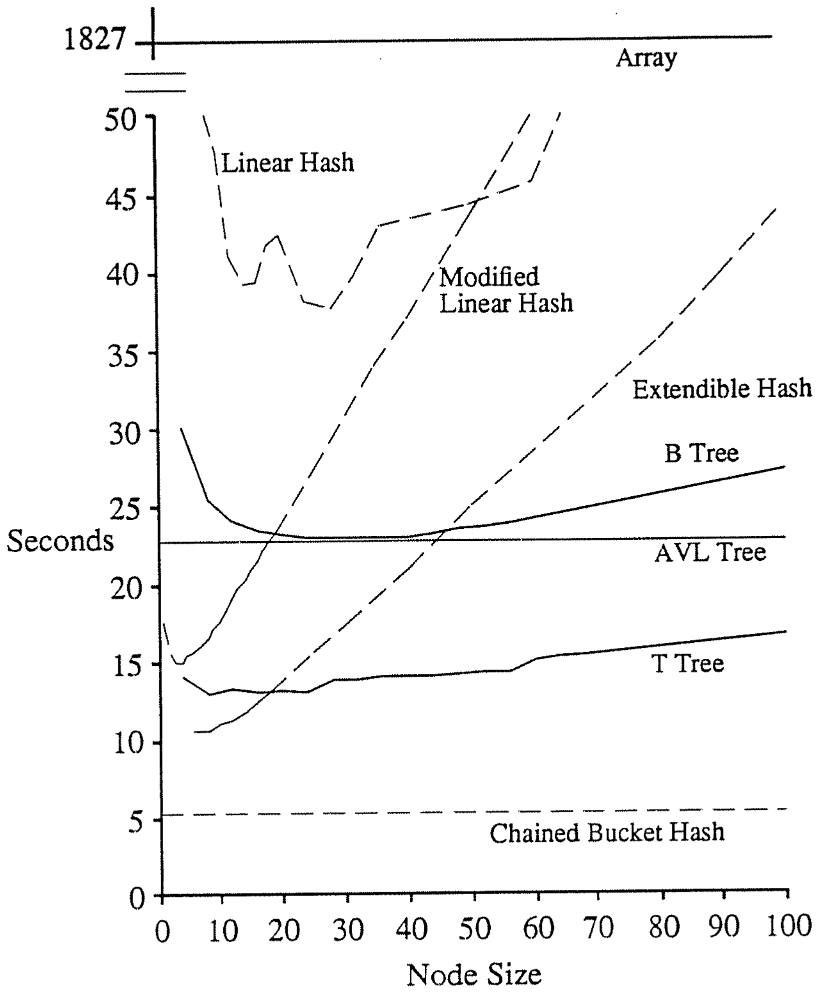
- Linear Hashing used an extra array of pointers to point to the last overflow bucket in each chain. This helped the insertion and deletion performance while sacrificing only a small amount in storage. (This array is figured into the the storage costs in Graph 3.7 in the test results.)

3.2.4. The Test Results

Graphs 3.1 through 3.6 show the execution times of the index structures for a representative subset of the tests, while Graph 3.7 shows the storage costs. The graphs use solid lines to represent the order-preserving structures and dashed lines to represent the hashing structures. Note that the different node sizes on the X-axis represent the average chain length parameter for Modified Linear Hashing rather than its node size, as Modified Linear Hashing used single-element nodes for these experiments. Also note that the array index structure had large execution times in all of the tests involving updates because of its $O(N)$ data movement. The array is therefore omitted from further discussion involving updates.[10] The reader should keep in mind that the lowest execution time or storage point for each line in the graphs is more important than the overall shape of the line; we are interested in the one node size that provides the best overall ratio of performance to storage cost. The shape of each line does have its use, however, as it shows how sensitive each index structure is to different node sizes.

INSERT 30,000 ELEMENTS — The results of this test are shown in Graph 3.1. The cost of a series of inserts depends on the search time and various operations particular to the individual data structure, such as the amount of data moved, the number of hash function calls, the number of tree rebalance operations, and the number of memory allocation operations. Looking at the AVL Tree search time *versus* the B Tree search time (Graph 3.2), it would seem that the AVL Tree should have the better

[10] It should be pointed out, however, that if the array were used as an index, a practical way of *building* it would be to do a bulk insert into the array as a heap and then quicksort it afterwards. During the preliminary tests for query processing (Chapter 4), it was discovered that an array index can be filled and then sorted in approximately half the time it takes to build a T Tree (with node size 30).



Graph 3.1 — Insert Cost for 30,000 Elements

insert time, but it has about the same insert time for B Tree Node sizes of 20 to 40 because it makes more memory allocation calls (one per data item inserted) and uses a costly rotation operation to rebalance the tree after insertions. Inserting a value into the B Tree, on the other hand, requires fewer memory allocation calls, some intra-node data movement and, much less frequently, some inter-node data movement (when a node splits into two nodes).[11] The T Tree has a search time close to that of the AVL Tree, but it also has update characteristics similar to those of the B Tree; the T Tree makes fewer memory allocation calls than the AVL Tree and usually relies on intra-node data movement rather than tree rotations to keep the tree balanced. Hence, the T Tree has the best insertion times of the order-preserving index structures.

In each hashing structure, the search cost is independent of the number of elements in the table, as there is a fixed cost to jump to the right node and then search the set of elements at that node. Also, hashing structures require little data movement because the new item is added to the end of the heap of elements in the node. All four hashing methods have approximately the same search cost for small nodes (see Graph 3.2), and they all insert a data item in about the same way (by appending to the end of the list), yet their insert costs are very different. The difference between the hashing schemes is in the amount of work needed to resize the directory. Linear Hashing splits nodes in order, possibly splitting nodes that are not full. The node splitting criteria is based on both the number and the distribution of elements in the hash table.[12] This means that table growth (node splitting) is allowed only when the main nodes are close to full, so a nonuniform distribution causes some of the main nodes to remain underfilled while some of the other nodes acquire long overflow chains due to the lack of reorganization. As shown in Graph 3.1, different node sizes yield different distributions for Linear Hashing, which have a direct effect on the search portion of the insert

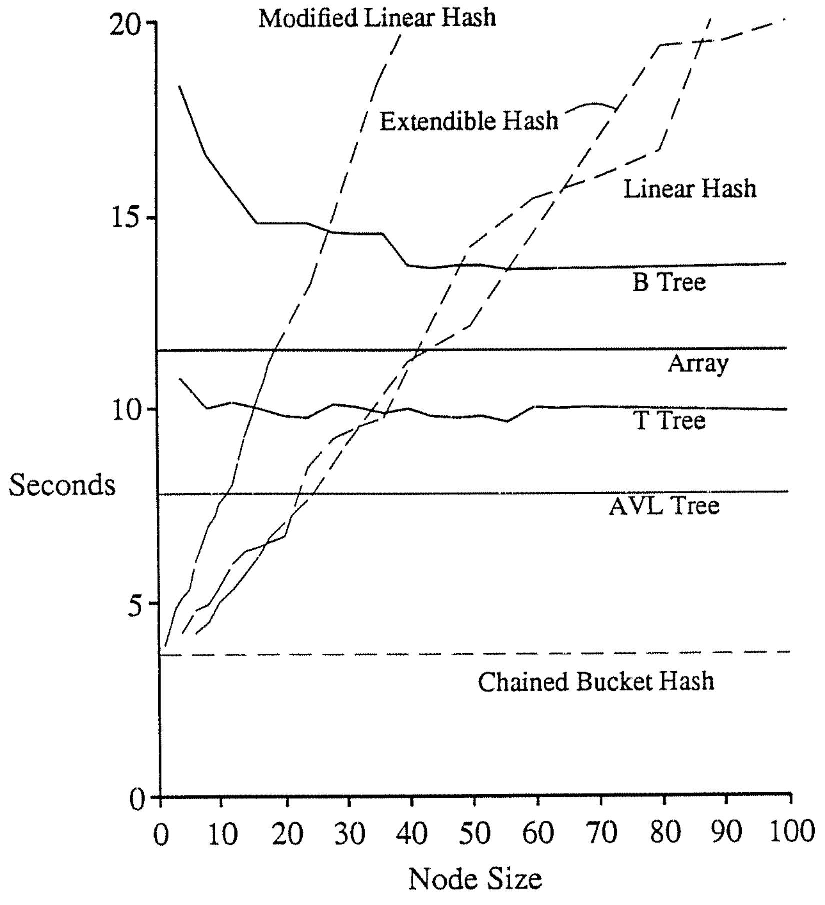
[11] Memory allocation and tree rotation costs exceeds this data movement cost for the node sizes used here due to the existence of the MOVC3 block move instruction on the VAX 11/750. This move instruction executes much faster than a hand optimized assembler routine of similar function.

[12] This is referred to as *load control* in [Litwi 80], and it is based on the ratio of data elements to total space allocated.

cost. Modified Linear Hashing, on the other hand, uses a node splitting criteria that is based solely on the total number of elements in the table, so the average length of the overflow chains is better controlled. It still uses the extra data reorganization of the Linear Hashing algorithm, but its more closely monitored overflow chain lengths result in a significantly decreased insert cost. Finally, Extendible Hashing requires that a node splits only when it overflows, so it does the least amount of reorganizing of data and thus has the fastest index insertion time of the dynamic hashing methods.

SEARCH — Graph 3.2 shows the search costs. The array uses a pure binary search. The overhead of the arithmetic calculation and movement of pointers is noticeable when compared to the "hardwired" binary search of a binary tree. The array binary search must *compute* the location of the next value in the array that is to be compared with the search value. In contrast, the AVL Tree search requires no arithmetic calculations — it follows a pointer to the location of the next compare value. The T Tree does the majority of its search in a manner similar to that of the AVL Tree; then, when it locates the correct node, it switches to a binary search of that node. Thus, the search cost of the T Tree search is slightly greater than the AVL Tree search cost, as some time is lost in binary searching the final node. The B Tree search time is the worst of the four order-preserving structures, because it requires several binary searches, one for each node in the search path.

The hashing schemes have a fixed cost for the hash function computation plus the cost of a linear search of the node and any associated overflow buckets. For the smallest node sizes, all four hashing methods are basically equivalent. The differences lie in the search times as the nodes get larger. Linear Hashing and Extendible Hashing are just about the same, as they both search multiple-item nodes. Modified Linear Hashing searches a linked list of single-item nodes, so each data reference requires traversing a

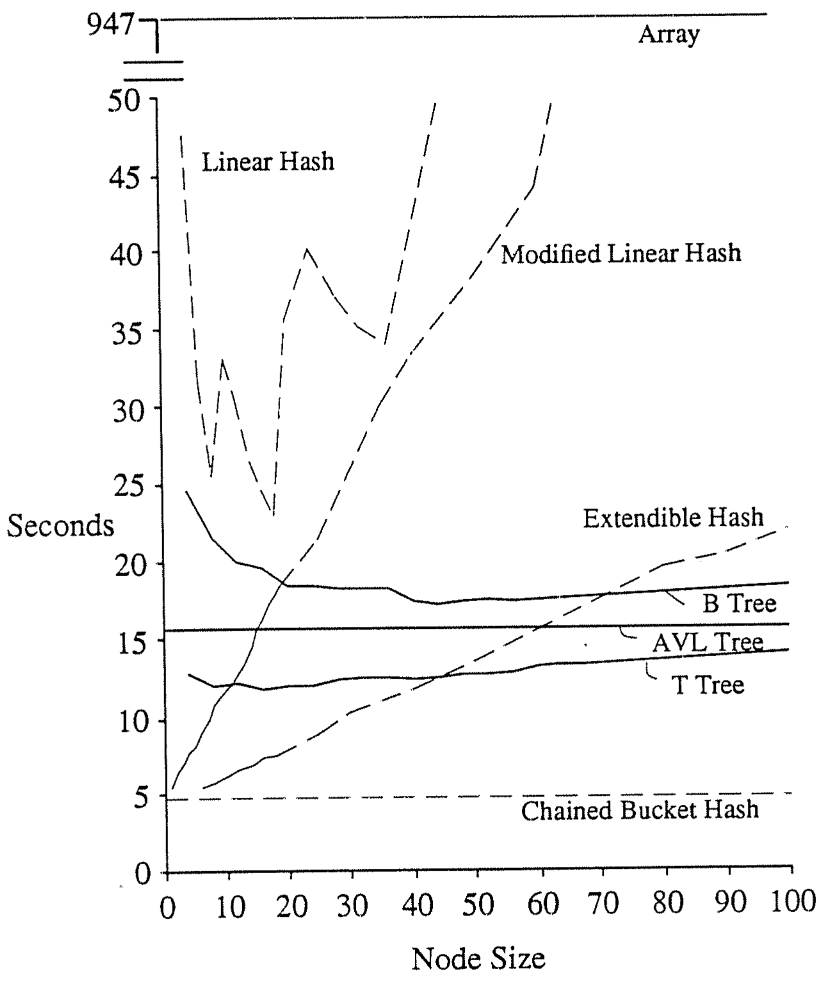


Graph 3.2 — Search Cost for 30,000 Elements

pointer. This overhead is noticeable when the chain becomes long. (Recall that "Node Size" is really average chain length for Modified Linear Hashing.)

QUERY MIXES — Graph 3.3 shows the "main event" of the query tests. This test is most important, as it shows the index structures in a normal working environment. The query mix of 60 percent searches, 20 percent inserts and 20 percent deletes was representative of the three query mix graphs, so it is the only one shown; index structures with faster search times did somewhat better in the 80 percent search query mix and those structures with better update characteristics did somewhat better in the

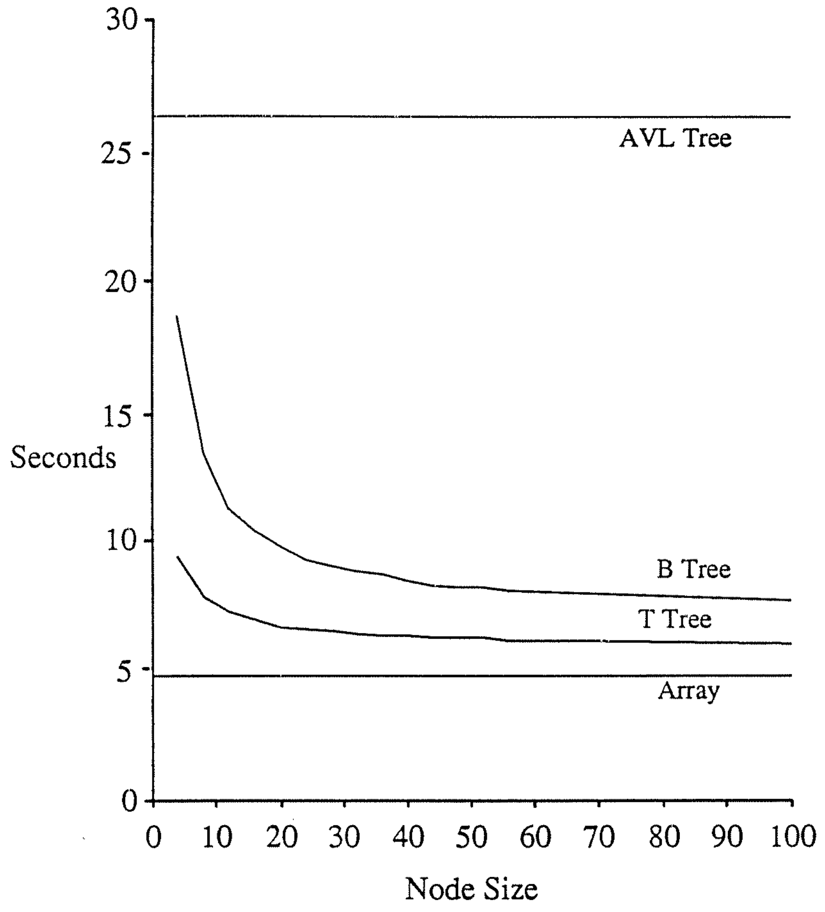
40 percent search query mix. The T Tree performs better than the AVL Tree and the B Tree here because of its better combined search/update capability. The AVL tree is faster than the B Tree because it is able to search faster than the B Tree, but their execution times are fairly close because of the B Tree's better update capability. For the smallest node sizes, Modified Linear Hashing, Extendible Hashing, and Chained Bucket Hashing are all basically equivalent. They have similar search costs, and when the need to resize the directory is not present, they all have the same update cost.



Graph 3.3 — Query Mix with 60% Searches

Linear Hashing, on the other hand, still reorganized its data in an attempt to maintain a particular storage utilization factor, so it had a slower query mix execution time as a result.

RANGE QUERIES — The results of the range query tests are shown in Graph 3.4. The range query returning 100 elements for each of 3,000 queries was representative of the range query tests, so it is the only one shown here. A range query is composed of two parts: the search for the lower bound value, and then the scan through the

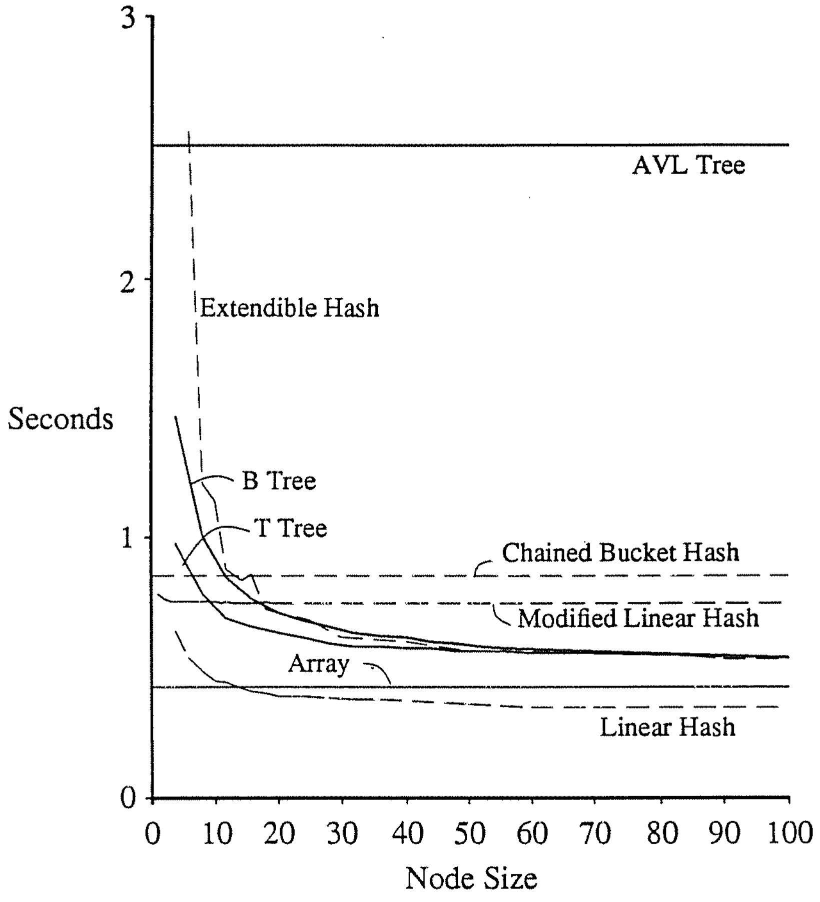


Graph 3.4 — Range Query

data structure until the upper bound value of the range is found or surpassed. When few elements per query are returned, the range query times are similar to the search times. As the number of elements returned per query increases, the index structure with the best scanning speed becomes dominant. The array has the best scanning speed, as its elements are contiguous, while the T Tree is close because it has many logically contiguous elements in each node. (This is only true for leaf nodes in the case of the B Tree.) The slower search speed of the B Tree causes it to be third, and the slow scanning speed of the AVL Tree (since each data reference requires traversing a pointer) makes it the slowest of the four. Note that the hash tables do not preserve a logical ordering of the values, so they were not tested for range query performance.

SEQUENTIAL SCAN — The results of the sequential scan tests are given in Graph 3.5. For most of the structures a scan is relatively fast because the structures contain segments of contiguous elements that can be traversed quickly. The main exception is the AVL tree, because the cost of walking a binary tree in order is exacerbated by the fact that every data item is in a different node. In general, those index structures using linked lists had slower scan times than those structures that have segments of contiguous elements. (There is one anomaly here, in that Linear Hashing seems to have beaten the array. After examining the scan routine code for both index structures to justify this difference in execution time, we found no tangible reason to account for this anomaly. However, since the difference in the execution times is only a tenth of a second, we attribute it to the error margin in the timing facility.)

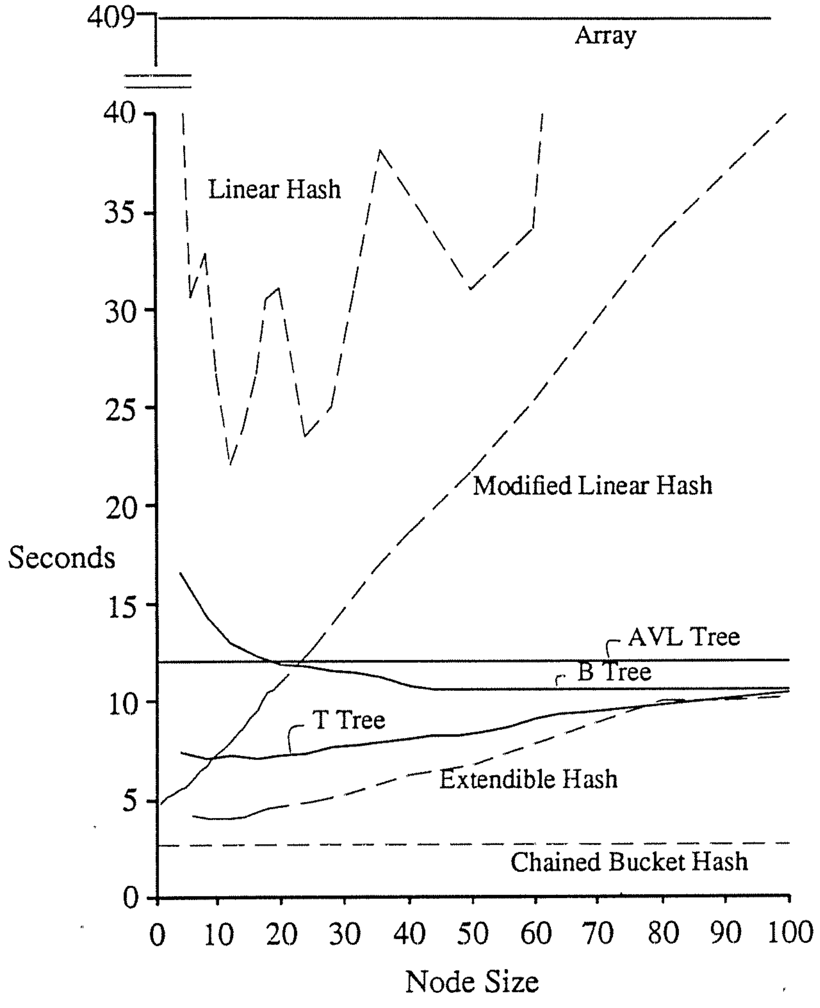
DELETE — Graph 3.6 shows the execution times of the index structures for deleting half of each data structure. Chained Bucket Hashing needs to do little work to remove an item from the hash table, so it has the fastest execution time once again. Extendible Hashing rarely decreases its hash table size, so delete is fast, but space may



Graph 3.5 — Index Scan

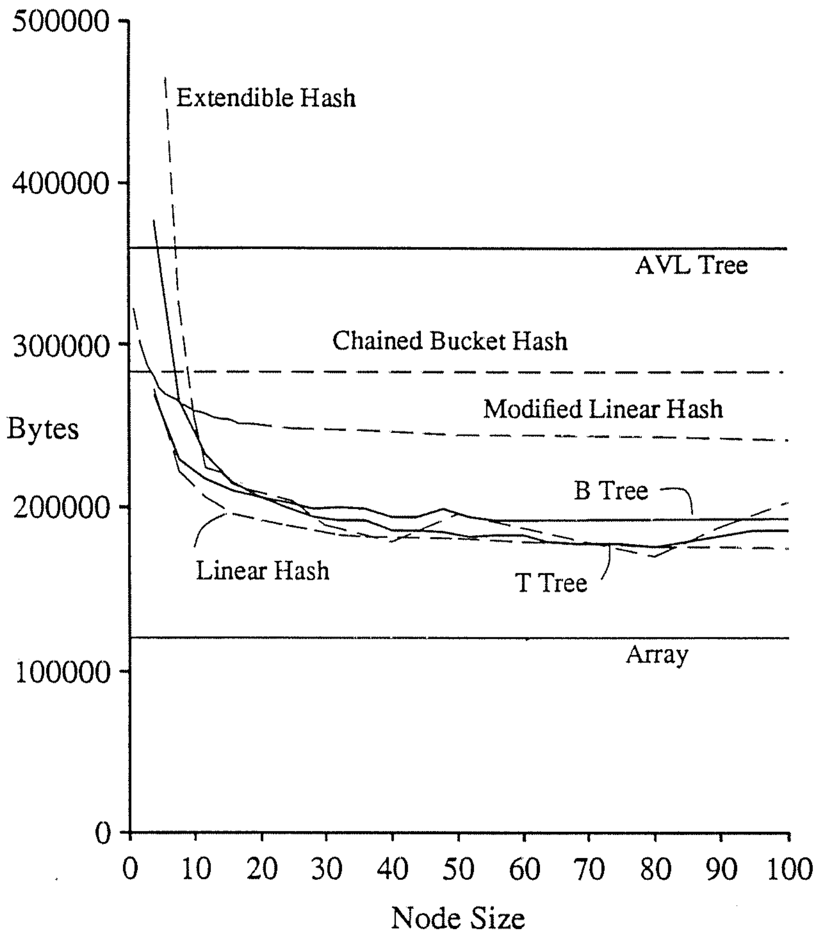
be wasted by its large directory and many partially filled nodes. At the shortest possible average chain length, Modified Linear Hashing is almost as fast as Chained Bucket Hashing; the difference in the two times is the amount of time devoted to reducing the size of the Modified Linear Hashing directory. The T Tree is faster than the AVL Tree and B Tree for the same reasons as before, while the B Tree is a little faster than the AVL Tree for medium size nodes because less work is required to keep it balanced.

STORAGE COST — The storage costs for Linear Hashing, Extendible Hashing, B Trees and T Trees were 5 percent more after the query mixes than their storage costs just after they were created. The storage costs for the index structures after the query mixes are shown in Graph 3.7. The array uses the minimum amount of storage, so we discuss the storage costs of the other algorithms as a ratio of their storage cost to the array storage cost. First, we consider the fixed values: the AVL Tree storage factor is 3 because of the two node pointers it needs for each data item, and Chained Bucket Hashing has a storage factor of 2.3 because it has one pointer for each data item and



Graph 3.6 — Delete

part of the table remains unused (because the hash function is not perfectly uniform). Modified Linear Hashing is similar to Chained Bucket Hashing for an average hash chain length of 2, but, as its hash chains grow longer, the number of empty slots in the table decreases and eventually the table becomes completely full. Finally, Linear Hashing, B Trees, Extendible Hashing and T Trees all had about equal storage factors of 1.5 for medium to large size nodes. Extendible Hashing tends to use the largest amount of storage for small nodes (6, 8, 10 and 12). This is because a small node size



Graph 3.7 — Storage Costs

increases the probability that some nodes will get more values than others, causing the directory to double repeatedly and thus use large amounts of storage. As its nodes get larger, the probability of this happening becomes lower. (Node sizes of 2 and 4 elements used so much memory that they could not be used in our test environment.)

3.2.5. Test Observations

An important thing to notice about the hash-based indices is that, while the Extendible Hashing and Modified Linear Hashing index structures had very good performance for small nodes, they also had high storage costs for small nodes. The storage utilization for Modified Linear Hashing, however, can probably be improved by using multiple-item nodes, thereby reducing its pointer to data item ratio. Extendible Hashing, besides its poor storage utilization, has another problem — storing duplicate values. If a page were to overflow with duplicate values, the directory could grow infinitely large and still not be able to resolve the problem; there is no normal overflow method in Extendible Hashing as there is in Linear Hashing. If duplicate values were allowed, then nodes would need a special overflow pointer and a special test to recognize them, while processing regular values in the normal fashion. As for the other two hash-based methods: Chained Bucket Hashing has fast execution times, but it has fairly high storage costs, and it is only a static structure; and finally, Linear Hashing is just too slow to use in main memory.

Looking at the order-preserving index structures, AVL Trees have good search execution times and fair update execution times, but they have high storage costs. Arrays have good search and scan execution times and low storage costs, but any update activity at all causes them to have execution times *orders of magnitude* higher than the other index structures. AVL Trees and arrays, therefore, do not have sufficiently good performance/storage characteristics for consideration as main memory indices. T Trees and B Trees do not have the storage problems of dynamic hashing methods; they have low storage costs for those node sizes that lead to good performance. The T Tree seems to be the best of choice for an order-preserving index structure, as it performed well throughout all of the tests.

3.2.6. Validating Experimental Results

A major problem with this type of comparison is the issue of fairness. Though we did our best to code each index algorithm equally well, the results are close in some cases, and it is possible that constant factors could come into play and alter the results. Also, some algorithms may have had small advantages because of the test environment. For example, the data used for the tests was generated by a random number generator, so the hashing schemes may have been aided by having a fairly uniform distribution of key values. Also, because unique key values were used and searches and deletes were always successful, those index structures using linear searches (*i.e.*, the hashing structures) needed to search only half of the list, on the average, to find the desired element.

To ensure that the tests were *coded* as fairly as possible, the code was instrumented with counters to record the number of occurrences of various operations (Table 3.2). These counters did not affect the execution times because they were compiled out of the code when the timing tests were run. Using the counters, it was possible to observe the algorithms in action, making sure that the expected amount of work was being done. These counters also present their own measures of algorithm performance.

For simplicity, only a subset of the general measurements will be presented here, as most of the index-specific operations are actually reflected in them; for example,

General Measurements	Number of Primary Compares Number of Secondary Compares Amount of Memory Moved Number of Move Calls Number of Memory Allocation Calls Amount of Memory Allocated Amount of Memory Freed
Tree Specific	Number of Node Searches Number of Single Tree Rotations Number of Double Tree Rotations Number of Tree Node Splits Number of Tree Node Merges
Hash Specific	Number of Hash Calls Number of Directory Splits Number of Directory Merges Number of Hash Node Splits Number of Hash Node Merges

Table 3.2 — Index Operations Measured

node splits and merges mainly involve data movement, so their cost is reflected in the data movement measurement. The measurements from two of the index tests are presented here, as they are fairly representative of the rest of the tests. The measurements presented are from the index search experiment and from the query mix experiment involving 60 percent searches. Also, since explanations for index performance have already been provided in the test results section, they are not duplicated here. Instead, we simply highlight the interesting anomalies in the measurement figures.

Measurements from the Search Operation

The main search measurements of interest are primary and secondary compare counts. Table 3.3 presents these results, including results for the alternative T Tree search algorithm. The primary compare count represents the number of times a search

Name	Node Size	Primary Compares	Secondary Compares	Execution Time
Array	n/a	417,242	645,000	11.54
AVL Tree	n/a	424,953	651,140	7.86
B Tree	52	424,465	670,406	13.74
CB Hash	n/a	59,875	59,875	3.70
Ext. Hash	6	84,295	84,295	4.22
Lin. Hash	8	130,293	130,293	4.95
ML Hash	2	77,262	77,262	3.94
T Tree	52	527,947	618,874	9.8
Alt T Tree	52	431,253	522,180	10.0

Table 3.3 — Measurements of Search Operation

key and a field value were compared, with the compare results being stored in a register. The secondary compare count represents the number of times the compare result was tested. (A primary compare was always followed by at least one secondary compare.) The array and tree binary search operations perform one or two secondary compares for each primary compare, as they test for *less than* or *less than* and *greater than.* On the average, a secondary compare is performed one and one half times per primary compare; approximately half of the time the search value will be less than the index value, requiring one test, and the other half of the time the search value will be greater than the index value, requiring two tests. The search procedures for the hash-based algorithms test only for equality, so the secondary compare counts are equal to the primary compare counts.

It is initially surprising to see the compare figures and then the resulting execution times; the T Tree performs many more primary compares than the other tree structures, yet it has the second fastest search time. As mentioned in the search performance results discussion Section 3.2.5, the reason for this has to do with the amount of work each index structure does *between* compares, and to a lesser degree, with the number of secondary compares. When compares are simple (as they are here), their cost does not play a dominant role in the search cost formula. In fact, since these index experiments used simple integers as index key values, the secondary compare cost was close to the primary compare cost. The more significant cost consideration, and probably the least obvious, is the computation required to get from one compare to the next. The AVL Tree and the T Tree contain pointers to the values to be compared, whereas the array and B Tree index structures must *compute* the locations of their compare values.

The most significant anomaly in the compare figures, however, is the performance of the alternative T Tree search method. The number of primary and secondary compares for the alternative method are less than the regular method, the same binary tree structure is used for both methods, and yet the regular T Tree search is slightly faster! The explanation for this is found in the total number of instructions executed for both search operations. The alternative search method always searches from the root to a leaf, whereas the regular search method stops when it finds a bounding node. For a tree size of about 600 nodes ($\frac{30{,}000}{52}$), the tree height is roughly 10 levels (for a node size of 52 elements). The regular T Tree search method traverses about 9 levels of the tree, the number of levels used to hold approximately half of the elements in the tree, and it performs one primary compare for each level and one primary compare every other level. The alternative search method traverses all 10 levels, making one compare at each level and marking a node at every other level. (Recall that the alternative search method marks a node when the search value is less than or equal to the maximum value in the node.) When the final instruction counts are tallied, the costs of the two search methods are very close. However, when data comparison costs are higher than those used here (recall that simple integers were used as data in our experiments), the search times will be different. The regular T Tree search method is slower than the alternative T Tree search method when more expensive compares are performed. This argument can be extended to the T+ Tree. The T+ Tree should have search times comparable to that of the T Tree using the alternative search method; in fact, they should be slightly faster since the T+ Tree search does not need to mark any nodes.

Name	Node Size	Primary Compares	Secondary Compares	k Bytes Moved	Move Calls	Mem. Calls	Exec. Time
Array	n/a	423,424	651,117	712,871	17,319	n/a	947.97
AVL Tree	n/a	415,642	633,246	n/a	n/a	11,993	15.64
B Tree	52	427,412	671,238	855	12,193	79	17.60
CB Hash	n/a	57,297	57,297	n/a	n/a	10,397	4.70
Ext. Hash	6	74,408	74,408	131	3	1,495	5.56
Lin. Hash	8	143,406	143,406	7,705	205	4,687	25.54
ML Hash	2	82,669	82,669	0	0	11,320	5.54
T Tree	52	533,156	623,954	1,053	11,929	110	12.65
Alt T Tree	52	441,804	532,610	1,053	11,934	125	12.80

Table 3.4 — Measurements of 60% Query Mix Operation

Measurements from a Query Mix Test

The measurements of the query mix involving 60 percent searches, 20 percent inserts and 20 percent deletes (Table 3.4) show results similar to the search measurements. The general measurements give an indication of the work being done by the index structures in an update environment: compare counts (search cost), data movement, and memory allocation activity. Memory allocation cost in our environment was relatively independent of the size of the memory block requested — independent tests showed that the memory allocation time increased roughly 10 percent for an order of magnitude increase in block size, and thus the *number* of memory allocation calls, rather than the amount of memory allocated, provides the best indication of the memory allocation costs. As for data movement, most large computers have a block move instruction that can move data quickly. However, there is a startup cost for each move, as the source address of the memory block, the destination address of the memory block, and the number of bytes to be moved have to be loaded into registers. Data movement costs are thus made up of two parts, the initial cost per copy operation, and the additional cost per byte copied.

In general, the measurements show a relatively straightforward breakdown of the index computing costs. The one exception is the AVL Tree, which had a significant number of tree rotation operations (4,658). (Tree rotation operations are specific to AVL Trees and T Trees, so they are not presented in the general measurement table.) By comparison, the T Tree needed only one rotation for the 12,000 inserts and deletes that were performed. The combined costs of the AVL Tree balance checking and rotation operations are larger than the cost of moving a block of data — the method used by the B Tree (and the main method used by the T Tree) to maintain a balanced tree.

An interesting figure is the amount of data moved by Linear Hashing. Its criteria for deciding to split or merge nodes is based on both storage utilization and distribution. The alternating insert and delete operations of the query mix experiments tend to change the distribution of values in an index structure; thus, Linear Hashing was constantly splitting or coalescing nodes to compensate for the changing distribution of values. In contrast, Modified Linear Hashing did not require any reorganization. Its node split/merge decision criteria is based only on the number of elements in the table, and the number of elements stayed relatively stable during the query mix experiments, so no reorganization was needed. Its only major cost is that of memory allocation. If the relative cost of memory allocation were to increase, Modified Linear Hashing might need to be redesigned to make fewer calls for allocating and freeing hash nodes.

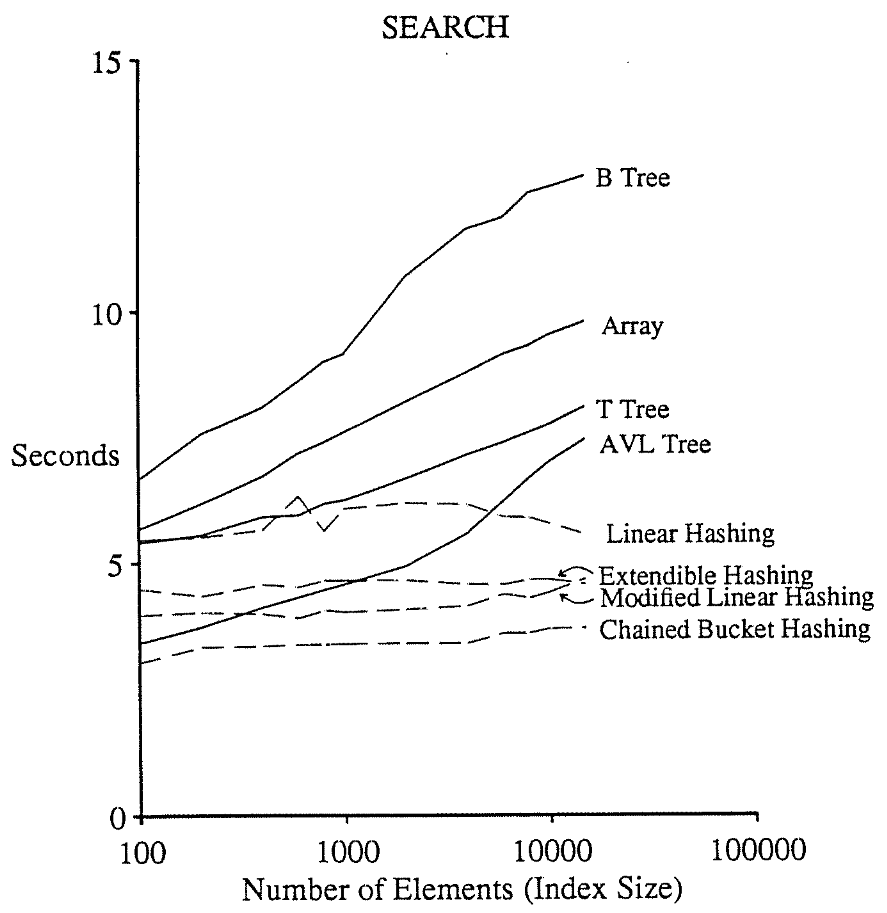
Except for the compare count anomaly discussed previously, the execution times of the tree structures appear to reflect the number of operations done. The data movement costs for T Trees and B Trees are about the same for similar size nodes, but since a B Tree node is typically 69 percent full and a T Tree node is typically 75 percent full, the T Tree moves more data. When the average number of elements in a node is equal, T Trees and B Trees move roughly the same amount of data. B Trees require more move calls, however, as extra move operations are generated during a node split or merge. As for the T+ Tree, it would be likely to have update characteristics similar to that of a B Tree and similar storage characteristics in its leaf nodes.

3.2.7. How Do These Performance Figures Scale?

Since we anticipate using database systems with large memories that can potentially store large relations, it is worth considering how larger index sizes will affect the performance of these index structures. If our index experiments could have been conducted using a much larger main memory, and thus larger index sizes, it appears that the relative execution times of the hashing structures and the order-preserving structures would differ significantly. The hash-based index structures should have roughly the same search and update performance for any index size, as they all have $O(1)$ search and update complexity; the number of hash table entries for a given hash value should be constant regardless of the index size. The order-preserving structures, on the other hand, have search and update complexity of $O(Log\ N)$, so their execution times will increase as the number of elements stored in them increases. Also, their *relative* execution times should vary because their cost formula parameter values change with their size. For example, consider how the relative search costs vary for T Trees, B Trees and AVL Trees for different index sizes. A T Tree search is composed of two search phases: the tree search for the bounding node and a binary search of the bounding node. Recall that, on a per compare basis, the node binary search is more expensive than the tree search because of the computations required in the binary search. As a T Tree increases in size, the number of compares done during the tree search increases, whereas the number of compares done during the node binary search remains constant. Thus, for increasing T Tree sizes, the percentage of compares done in the less costly manner (*i.e.* the tree search) also increases.

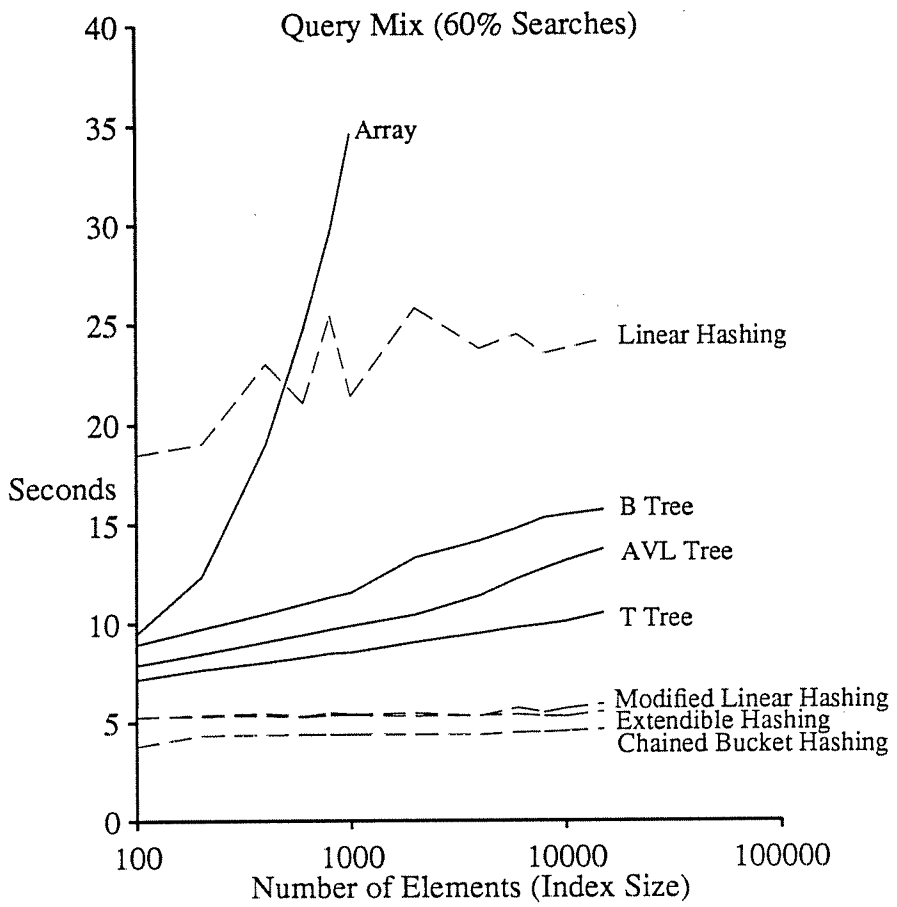
For very small T Tree sizes, most of the compares are done during the node binary search, so the T Tree search times should be close to that of the Array. As the size of a T Tree increases, more compares are performed during the tree search phase, so its search times should become increasingly less than those of the array and approach those of the AVL Tree. The B Tree, on the other hand, has the opposite behavior with respect to the array. For small tree sizes, the B Tree search time is bounded by the array search time, as they both use binary searches, but the B Tree search involves extra overhead for traversing nodes and initiating several binary searches. As its size increases, it has to perform more binary searches, and thus its search time will tend to diverge from the Array search time because each extra binary search requires a small amount of additional processing overhead. Therefore, for both large and small index sizes, the T Tree should continue to provide better search performance than the Array or the B Tree.

To show how the search performance actually varies with different index sizes, each index structure was built over a range of sizes and searched 30,000 times (Graph 3.8). For this graph, the node sizes were fixed at the values shown in Table 3.3.



Graph 3.8 — Search Performance for Different Index Sizes

Theoretically, when plotted on a log scale as in Graph 3.8, the Array and AVL Tree search times should appear as straight lines, and the slope of the array search times should be steeper than the slope of the AVL Tree search times. This is partially true, as the lines in Graph 3.8 show this behavior for index sizes of 100 to about 3,000 elements; however, Graph 3.8 also shows a rise in the AVL Tree search time for larger index sizes. It is possible that this is due to the balance of the AVL Tree. Even though it is a "balanced" structure, it can become fairly lopsided and cause search times to vary. (Although subtrees of the same parent can differ in height by only one level, a



Graph 3.9 — Update Performance for Different Index Sizes

large tree can have leaf nodes at many different levels.) In contrast, the Array is always "balanced" with respect to its binary search. A similar phenomenon occurs for the hash-based structures. Even though their search times should be relatively constant regardless of the index size, it is possible that the index structures that use chains of small hash nodes (Modified Linear Hashing and Chained Bucket Hashing) will create some long chains of elements because of a slightly non-uniform hash function, and thus make their search times slightly slower. Extendible Hashing does not have this problem because the number of elements corresponding to a hash value is bounded by its node size.

Index size affects index update performance in a similar manner. Graph 3.9 shows the query mix performance of the index structures for a range of index sizes. For this query mix experiment, the index structure node sizes were fixed at the values shown in Table 3.3, as in the search experiment above. Since search performance has a major impact on update performance, growth trends similar to those seen in the search graph (Graph 3.8) are seen here. The main exception to this is the array. Since it has a linear data movement cost, its update times increase linearly rather than logarithmically. It appears that the T Tree will remain the dominant order-preserving structure because of its good search performance and insertion/deletion characteristics.

3.3. Summary

In this chapter a new main memory index structure, the T Tree, was introduced and compared against AVL Trees, simple arrays, B Trees, Chained Bucket Hashing, Extendible Hashing, Linear Hashing and Modified Linear Hashing. The test results indicate that Modified Linear Hashing is the best index structure for storing non-ordered data and that the T Tree is the best index structure for storing ordered data. When both types of data are used in a database, a mix of two different index structures provides the best overall storage and performance. Also, having observed these index structures for various index sizes, it appears that the T Tree and Modified Linear Hashing will be the preferred index methods for *all* index sizes. For unordered data, Modified Linear Hashing gives excellent performance for exact match queries. When used as a temporary structure where the size of the index is known in advance, the table size can be chosen initially to fit the application, thereby removing its reorganization overhead and allowing Modified Linear Hashing to behave like Chained Bucket

Hashing. For ordered data, the T Tree provides excellent overall performance for a mix of searches, inserts, and deletes, and it does so at a relatively low cost in storage space. The T+ Tree, although not tested explicitly, appears likely to have search times similar to that of the T Tree and slightly slower insert and delete times. Its ability to isolate the effects of updates to one or two data nodes might be useful with respect to locking and recovery operations (see Chapters 5 and 6).

The machine independent measurements presented in Section 3.2.6 indicate that the index execution times presented are reasonably accurate; they also provide some insight into where the most work is being done in the various index structures. The amount of computation required *between* data comparisons is an important consideration, as it is responsible for much of the differences in constant factors among the index structures. Also, it appears that the relative performance results for the index structures will scale to very large index sizes, as the importance of constant factors does not lessen with increases in index size.

Chapter 4

Query Processing

The direct addressability of data in a memory-resident database has a significant impact on query processing. Interaction with buffer managers and the concern for the clustered order of tuples are removed, thus creating a different query processing environment with a new set of costs for selection, join, and projection operations. This chapter presents results from experiments conducted on old and new algorithms for these query processing operations. The purpose of these experiments are to determine which algorithms perform best in a main memory environment.

As in the index studies in Chapter 3, all of the tests reported here were run on a PDP VAX 11/750 running with two megabytes of real memory (as opposed to virtual memory). Each of the algorithms was implemented in the C programming language, and every effort was made to ensure that the quality of the implementations was uniform across the algorithms. The validity of the execution times reported here was verified by recording and examining the number of comparisons, the amount of data movement, the number of hash function calls, and other miscellaneous operations to ensure that the algorithms were doing what they were supposed to (*i.e.*, neither more nor less). These counters were compiled out of the code when the final performance tests were run, so the execution times presented here reflect the running times of the actual operations with very little time spent in overhead (e.g., driver) routines. Timing was done using a routine similar to the 'getrusage' facility of Unix.[13]

[13]Unix is a trademark of AT&T Bell Laboratories.

From the results of the index study in Chapter 3, it is clear that the two index structures of choice are the T Tree, for ordered data, and Modified Linear Hashing, for nonordered data. Chained Bucket Hashing and Modified Linear Hashing have similar search and update performance when the table size can be set to the correct size in advance. Since Chained Bucket Hashing routines are easier to write and modify, they are used in place of Modified Linear Hashing routines. Thus, T Trees and Chained Bucket Hashing are used here to test possible join and project algorithms in order to determine the best algorithm in each category.

4.1. Join

Previous join studies involving large memories have been based on the large buffer pool assumption [Shapi 86], [DeWit 84], [DeWit 85]. (Others have studied hash joins as well in a normal disk environment [Babb 79], [Valdu 84], [Bratb 84], but their results are less applicable here.) Three main join methods were studied in [DeWit 85]: Nested Loops with a hashed index, Sort Merge [Blasg 77], and three hashing methods, Simple Hash, Hybrid Hash and GRACE Hash [DeWit 84]. The results showed that when both relations fit in memory, the three hash algorithms become equivalent, and the nested loops join with a hash index was found to perform just as well as the other hash algorithms (and outperformed Sort Merge). They also studied the use of semijoin processing with bit vectors to reduce the number of disk accesses involved in the join, but this semijoin pass is redundant when the relations are entirely memory-resident. The variety of join relation compositions (e.g., sizes, join selectivities, and join column value distributions) used in their study was small, and may not completely reflect all possibilities (performance-wise).

In this study, we examine the performance of a number of candidate join methods for the MM-DBMS. We use a wide selection of relation compositions so as to evaluate the algorithms under a wide variety of possible conditions.

4.1.1. Relation Generation

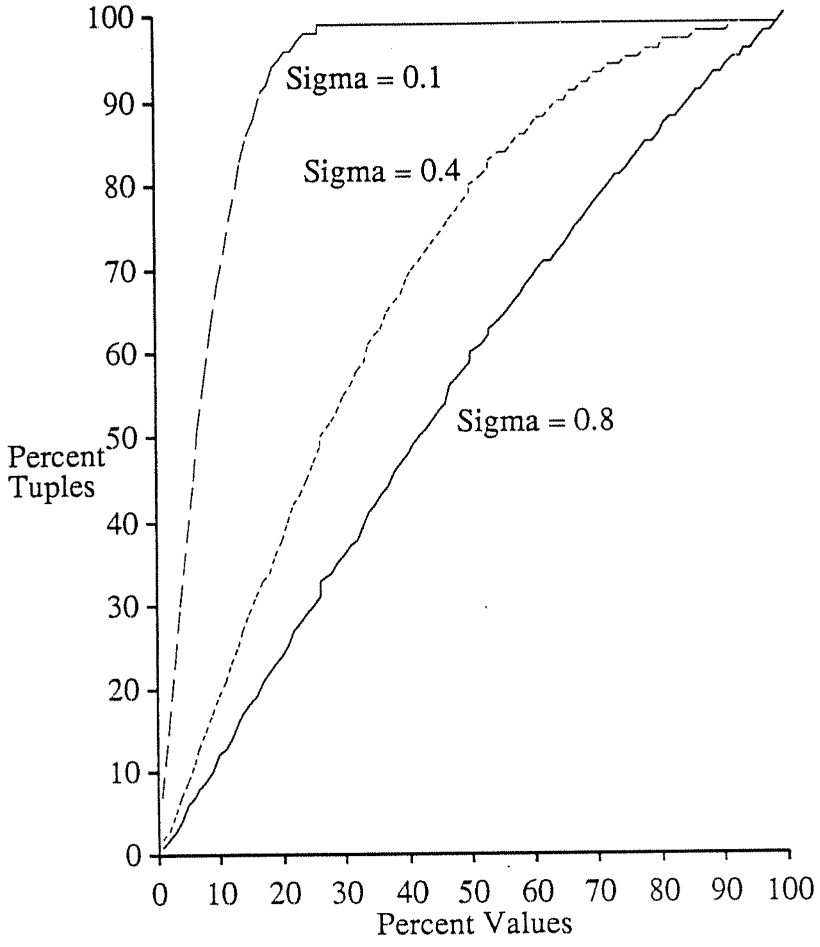
In order to support our intent to experiment with a variety of relation compositions, we constructed our test relations so that we could vary several parameters. The variable parameters were:

(1) The relation cardinality (|R|)

(2) The number of join column duplicate values (as a percentage of |R|) and their distribution.

(3) The semijoin selectivity (the number of values in the larger relation that participate in the join, expressed as a percentage of the larger relation).

In order to get a variable semijoin selectivity, the smaller relation was built with a specified number of values from the larger relation. To get a variable number of duplicates, a specified number of unique values were generated (either from a random number generator or from the larger relation), and then the number of occurrences of each of these values was determined using a random sampling procedure based on a truncated normal distribution with a variable standard deviation. Graph 4.1 shows the three duplicate distributions used for the experiments — a skewed distribution (where the standard deviation was 0.1), a moderately skewed distribution (the 0.4 curve in the graph), and a near-uniform distribution (the 0.8 curve in the graph). The results for the 0.4 and 0.8 cases were similar, so results are given here only for the two extreme cases.



Graph 4.1 — Distribution of Duplicate Values

4.1.2. The Join Algorithms

For memory-resident databases, all of the hash-based algorithms tested in [DeWit 85] were found to perform equally well. Therefore, the hash-based nested loops algorithm is the only hash-based algorithm that we examine here. For our experiments, we implemented and measured the performance of a total of five join algorithms: Nested Loops, a simple main-memory version of a nested loops join with no index; Hash Join and Tree Join, two variants of the nested loops join that use indices;

and Sort Merge and Tree Merge, two variants of the sort-merge join method of [Blasg 77]. We briefly describe each of these methods in turn. Recall that relations are always accessed *via* an index; unless otherwise specified, an array index was used to scan the relations in our experiments.

The pure Nested Loops join is an $O(N^2)$ algorithm. It uses one relation as the outer relation, scanning each of its tuples once. For each outer tuple, it then scans the entire inner relation looking for tuples with a matching join column value. The Hash Join and Tree Join algorithms are similar, but they each use an index to limit the number of tuples that have to be scanned in the inner relation. The Hash Join builds a Chain Bucket Hash index on the join column of the inner relation, and then it uses this index to find matching tuples during the join. The Tree Join uses an existing T Tree index on the inner relation to find matching tuples. We do not include the possibility of building a T Tree on the inner relation for the join because it turns out to be a viable alternative only if the T Tree already exists as a regular index — if the cost to build the tree is included, a Tree Join will *always* cost more than a Hash Join, as a T Tree costs more to build and a hash table is faster for single value retrieval (as shown in Graph 4.2).

The merge join algorithm [Blasg 77] was implemented using two index structures, an array index and a T Tree index. For the Sort Merge algorithm tested here, array indices were built on each relation and then they were sorted. The sort was done using quicksort with an insertion sort for subarrays of ten elements or less.[14] For the Tree Merge experiments, the T Tree indices were built on the join columns of each

[14] A preliminary test was run to determine the optimal subarray size for switching from quicksort to insertion sort; the optimal subarray size was 10.

relation, and then a merge join was performed using these indices. However, the T Tree construction times are not reported in the experiments — it turns out that the T Merge algorithm is only a viable alternative if the indices already exist. Preliminary tests showed that the arrays can be built and sorted in 60 percent of the time to build the trees and also that the array can be scanned in about 60 percent of the time it takes to scan a tree.

4.1.3. Join Experiments

The join algorithms were each tested with a variety of relation compositions in order to determine their relative performance. Six experiments were performed in all, and they are summarized below. In the description of the experiments, |R1| denotes the cardinality of R1, the outer relation, and |R2| denotes the cardinality of R2, the inner relation:

(1) *Vary Cardinality*: Vary the sizes of the relations with |R1| = |R2|, 0% duplicates, and a semijoin selectivity of 100%.

(2) *Vary Inner Cardinality*: Vary the size of R2 (|R2| = 1-100% of |R1|) with |R1| = 30,000, 0% duplicates, and a semijoin selectivity of 100%.

(3) *Vary Outer Cardinality*: Vary the size of R1 (|R1| = 1-100% of |R2|) with |R2| = 30,000, 0% duplicates, and a semijoin selectivity of 100%.

(4) *Vary Duplicate Percentage (skewed)*: Vary the duplicate percentage of both relations from 0-100% with |R1| = |R2| = 20,000, a semijoin selectivity of 100%, and a skewed duplicate distribution.

(5) *Vary Duplicate Percentage (uniform)*: Vary the duplicate percentage of both relations from 0-100% with |R1| = |R2| = 20,000, a semijoin selectivity of 100%, and a uniform duplicate distribution.

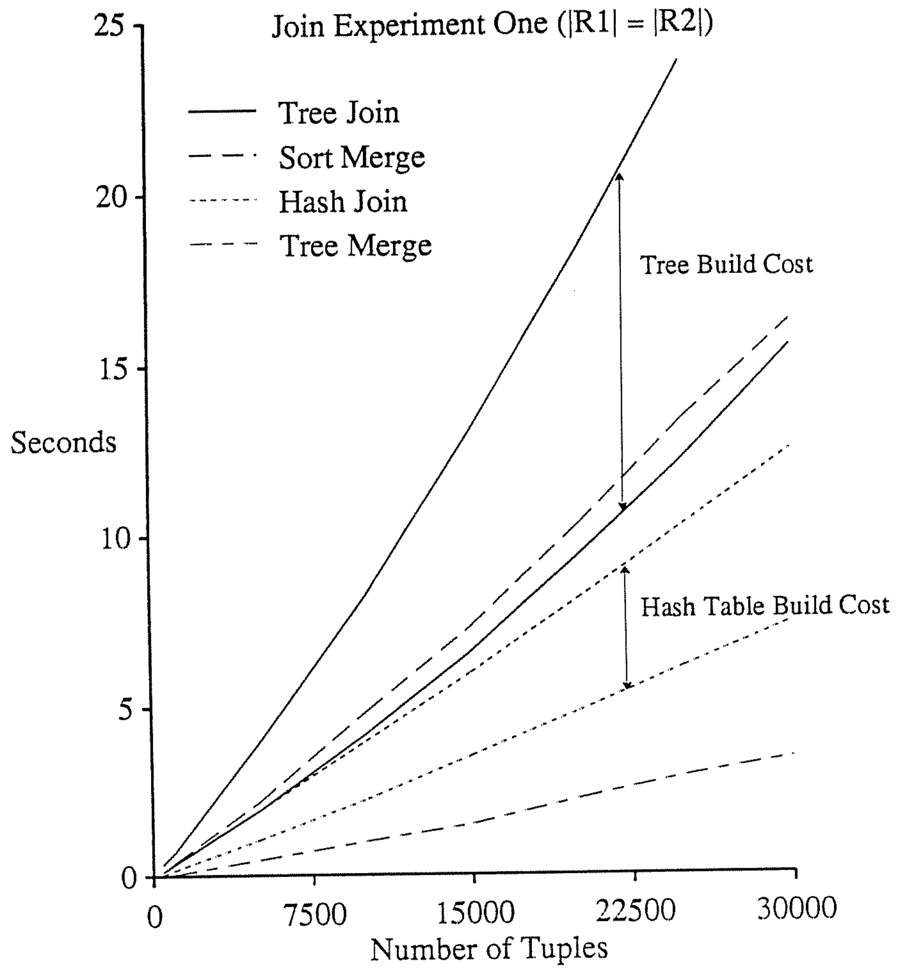
(6) *Vary Semijoin Selectivity*: Vary the semijoin selectivity from 1-100% with |R1| = |R2| = 30,000 and a duplicate percentage of 50% with a uniform duplicate distribution.

4.1.4. Join Experiments Results

We present the results of each of the join experiments in this section. The results for the Nested Loops algorithm will be presented separately at the end of the section, as its performance was typically two orders of magnitude worse than that of the other join methods.

Experiment One — Vary Cardinality

Graph 4.2 shows the performance of the join methods for relations with equal cardinalities. The relations are joined on keys (*i.e.*, no duplicates) with a semijoin selectivity of 100% (*i.e.*, all tuples participate in the join). If both indices are available, then a Tree Merge gives the best performance. It does the least amount of work, as the T Tree indices are assumed to exist, and scanning them in order limits the number of comparisons required to perform the join. The amount of work done is proportional to |R1| + |R2| * 2, as each element in R1 is referenced once and each element in R2 is referenced twice. (The presence of duplicates would increase the number of times that the elements in R2 are referenced.) If it is not that case that both indices are available, it is best to do a Hash Join. It turns out that, in this case, it is actually faster to build and use a hash table on the inner relation than to use an existing T Tree index, if the hash index does not already exist. A Hash table has a fixed cost to look up a value that is independent of the index size. The amount of work done in a Hash Join is proportional to |R1| + (|R1| * k), where k is a fixed lookup cost for each search of the inner relation, whereas the amount of work done in a Tree Join is proportional to |R1| + (|R1|



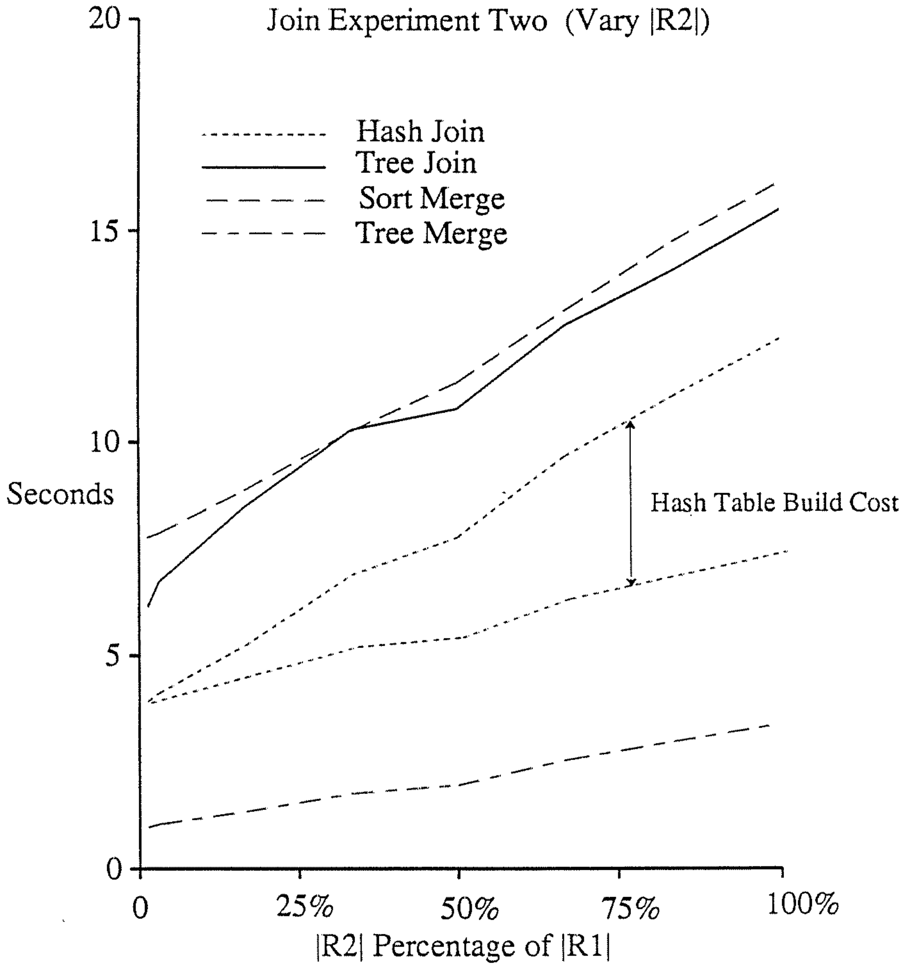
Graph 4.2 — Vary Cardinality

$* Log_2 |R2|)$[15]. (The value of k is typically much smaller than $Log_2 |R2|$.) Finally, the Sort Merge algorithm has the worst performance of the algorithms in this experiment, as the cost of building and sorting the arrays for use in the merge phase is too high, roughly $(|R1| * Log_2 |R1|) + (|R2| * Log_2 |R2|) + (|R1| + |R2|)$.

[15]In both formulas, the first term (|R1|) is the cost of scanning the outer relation; the second term is the cost of looking up |R1| values in the inner relation.

Experiment Two — Vary Inner Cardinality

Graph 4.3 shows the performance of the join methods as R2's cardinality is varied from 1-100% of the cardinality of R1. In this experiment, R1's cardinality is fixed at 30,000, the join columns were again keys (*i.e.*, no duplicates), and the semijoin selectivity was again 100%. The results obtained here are similar to those of Experiment One, with Tree Merge performing the best if T Tree indices exist on both join columns,

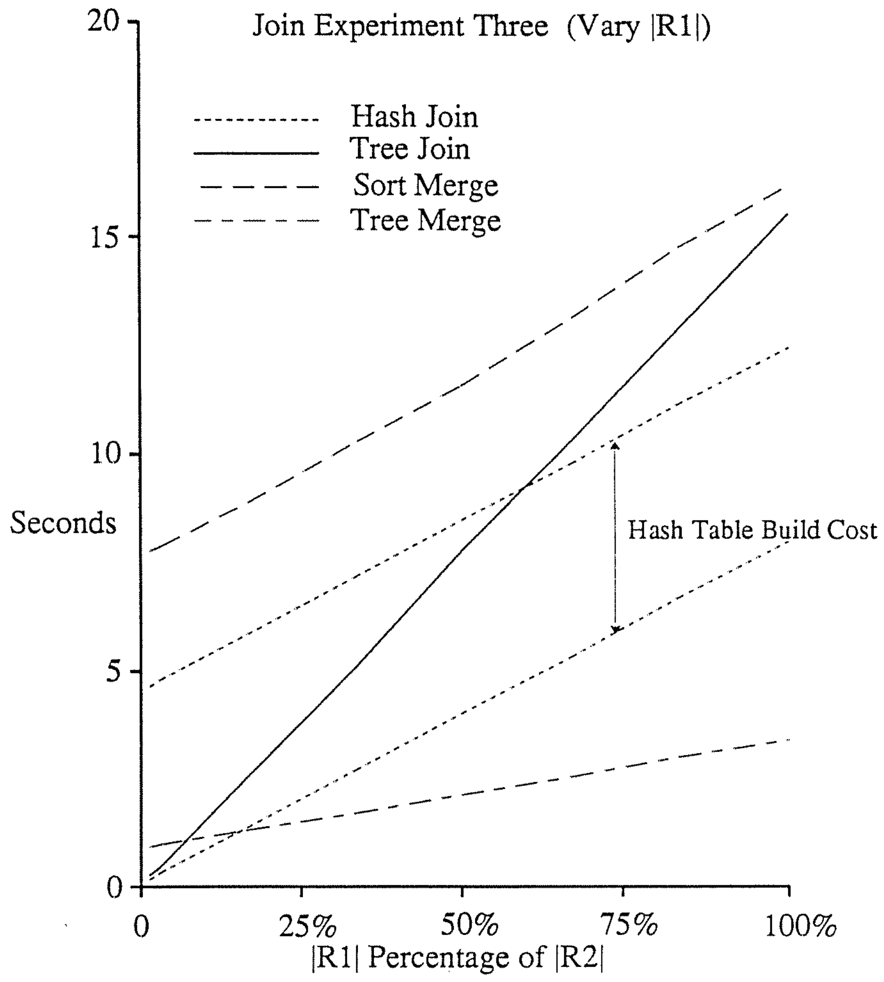


Graph 4.3 — Vary Inner Cardinality

and Hash Join performing the best otherwise. In this experiment, each of the the index joins were basically doing |R1| searches of an index of (increasing) cardinality |R2|.

Experiment Three — Vary Outer Cardinality

The parameters of Experiment Three were identical to those of Experiment Two except that |R1| was varied instead of |R2|. The results of this experiment are shown in

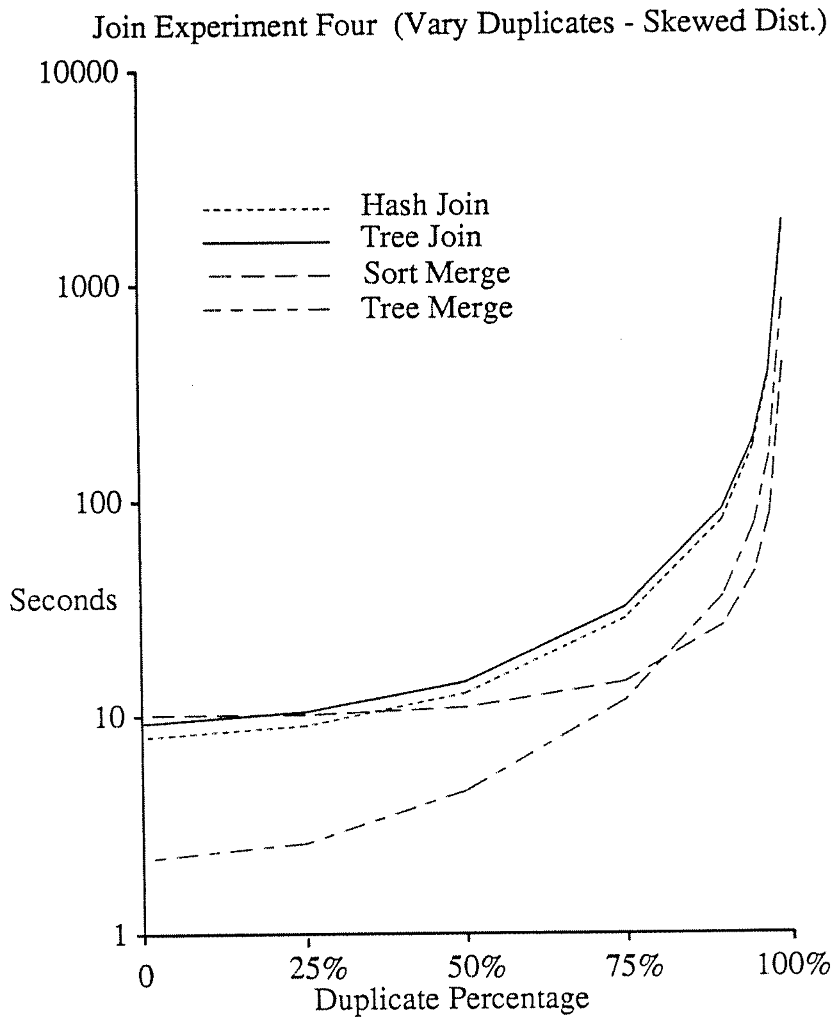


Graph 4.4 — Vary Outer Cardinality

Graph 4.4. The Tree Merge, Hash Join, and Sort Merge algorithms perform much the same as they did in Experiment Two. In this case, however, the Tree Join and the Hash Join with an existing index outperform the others for small values of |R1|, beating even the Tree Merge algorithm for the smallest |R1| values. This is intuitive, as the T Tree Merge Join behaves like a sequential scan when R1 contains few tuples, whereas the Tree Join or the Hash Join require only a few index lookups.

Experiment Four — Vary Duplicate Percentage (skewed)

For Experiment Four, |R1| and |R2| were fixed at 20,000, the semijoin selectivity was kept at 100%, and the duplicate percentage for both relations was varied from 1 to 100%. (Only 20,000 tuples were used in experiments Four and Five because the execution times tended to grow very large as the duplicate percentage increased.) The results of this experiment are shown in Graph 4.5. The duplicate distribution was skewed, so there were many duplicates for some values and few or none for others. Also, the duplicate percentages of the two relations were different in this experiment as a result of the relation construction procedure that was used. In order to achieve 100 percent semijoin selectivity, the values for R2 were chosen from R1, and R1 already contained a non-uniform distribution of duplicates. Therefore, the number of duplicates in R2 is greater than that of R1. The duplicate percentages in Graph 4.5 refer to R1.



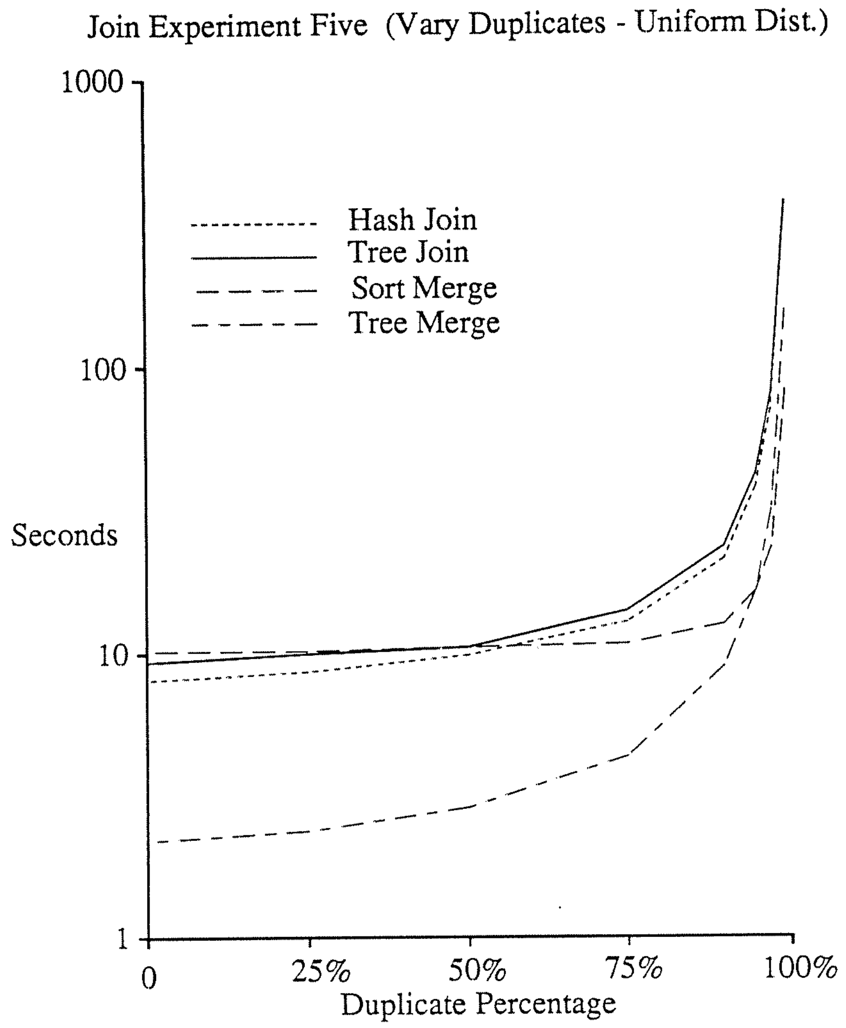
Graph 4.5 — Vary Duplicate Percentage (skewed)

When the number of duplicates becomes significant, the number of matching tuples (and hence result tuples) is large, resulting in many more tuples being scanned. Graph 4.5 shows that the Sort Merge method is the most efficient of the algorithms for scanning large numbers of tuples — once the skewed duplicate percentage reaches about 80 percent, the cost of building and sorting the arrays is overcome by the efficiency of scanning the relations *via* the arrays, so it beats even Tree Merge in this case. Although the number of comparisons is the same, as both Tree Merge and Sort

Merge use the same Merge Join algorithm, the array index can be scanned faster than the T Tree index because the array index holds a list of contiguous elements whereas the T Tree holds nodes of contiguous elements joined by pointers. The experiment results from Chapter 3 show that the array can be scanned in about 2/3 the time it takes to scan a T Tree. The Index Join methods are less efficient for processing large numbers of elements for each join value, so they begin to lose to Sort Merge even before the skewed duplicate percentage reaches 40 percent.

Experiment Five — Vary Duplicate Percentage (uniform)

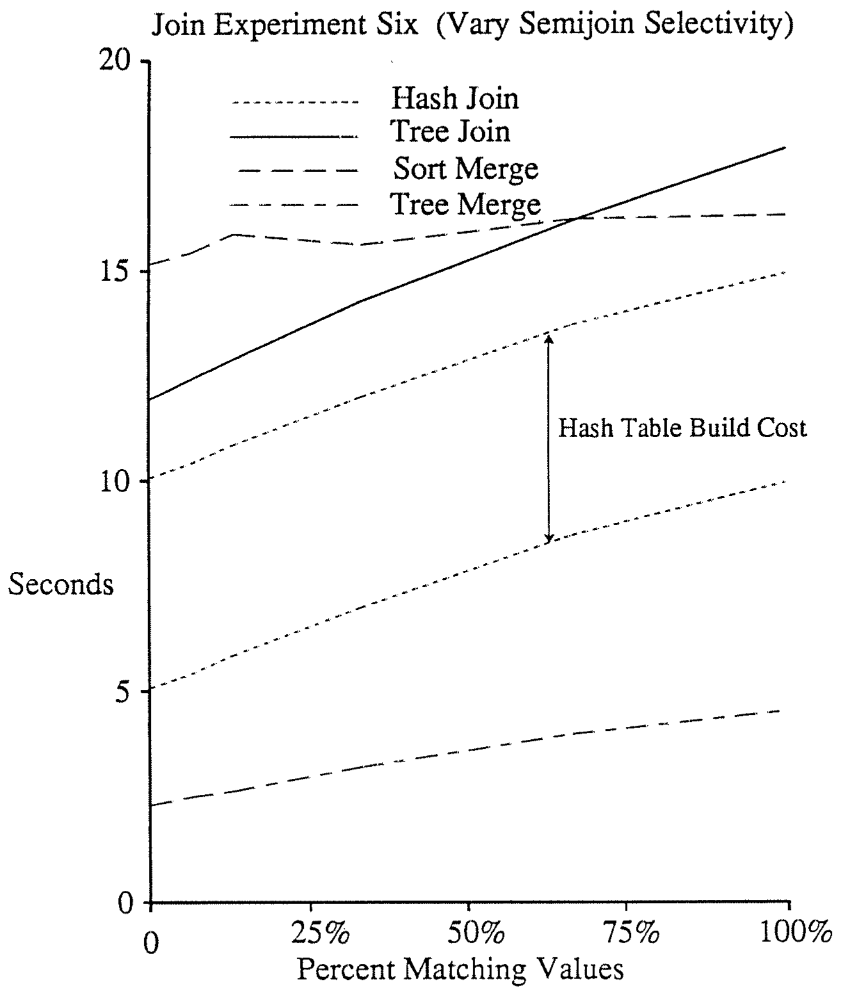
Experiment Five is identical to Experiment Four except that the distribution of duplicates was uniform. The results of Experiment Five are shown in Graph 4.6. The duplicate percentages of R1 and R2 are the same here, because R2 was created with a uniform distribution of R1 values. Here, the Tree Merge algorithm remained the best method until the duplicate percentage exceeded about 97 percent because the output of the join was much lower for most duplicate percentages. When the duplicate percentages were low (0-60 percent), the join algorithms had behavior similar to that of the earlier experiments. Once the duplicate percentage became high enough to cause a high output join (at about 97 percent), Sort Merge again became the fastest join method.



Graph 4.6 — Vary Duplicate Percentage (uniform)

Experiment Six — Vary Semijoin Selectivity

In the previous experiments, the semijoin selectivity was held constant at 100%. In Experiment Six, however, it was varied, and the results of this test are shown in Graph 4.7. For this experiment, |R1| = |R2| = 30,000 elements, the duplicate percentage was fixed at 50% in each relation with a uniform distribution (so there were roughly two occurrences of each join column value in each relation), and the semijoin



Graph 4.7 — Vary Semijoin Selectivity

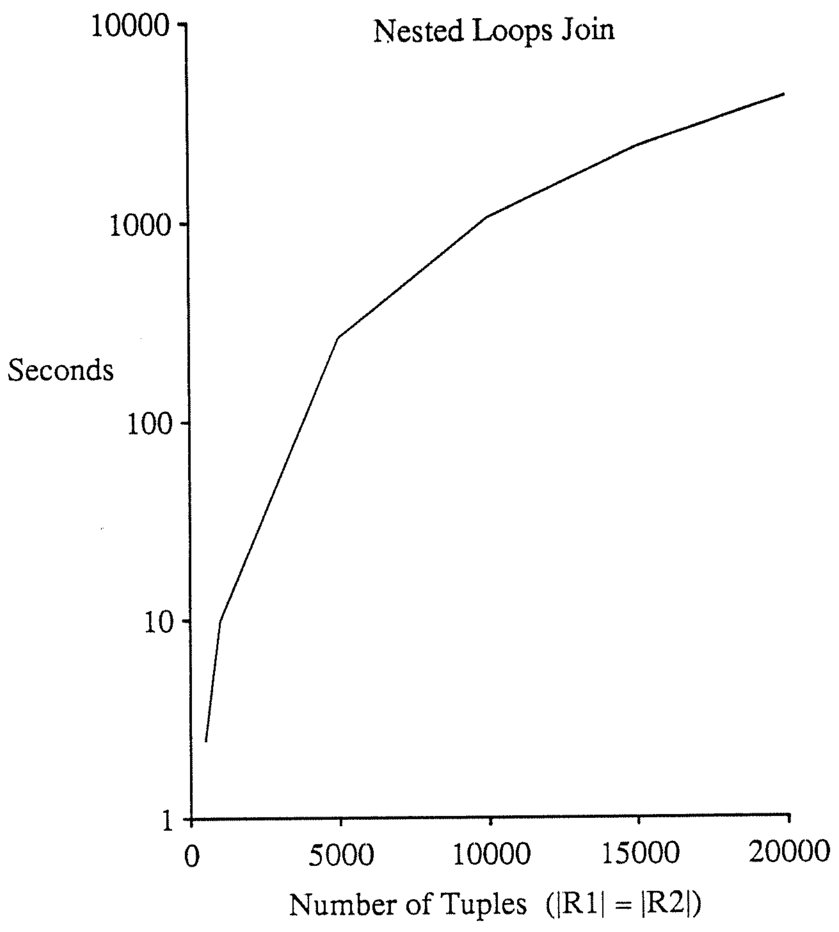
selectivity was varied from 1-100%. The Tree Join was affected the most by the increase in matching values, and a brief description of the search procedure will explain why: When the T Tree is searched for a set of tuples with a single value, the search stops at any tuple with that value, and the tree is then scanned in both directions from that position (since the list of tuples for a given value is logically contiguous in the tree). If the initial search does not find any tuples matching the search value, then the scan phase is bypassed and the search returns unsuccessfully. When the percentage of matching values is low, then most of the searches are unsuccessful and the total cost is much lower than when the majority of searches are successful. A similar case can be made for the Hash Join in that unsuccessful searches sometimes require less work than successful ones — an unsuccessful search may scan an empty hash chain instead of a full one. The increase in the Tree Merge execution time in Graph 4.7 was due mostly to the extra data comparisons and the extra overhead of recording the increasing number of matching tuples. Sort Merge is less affected by the increase in matching tuples because the sorting time overshadows the time required to perform the actual merge join.

4.1.5. Join Experiment Results Summary

If the proper pair of tree indices is present, the Tree Merge join method was found to perform the best in almost all of the situations tested. It turned out never to be advantageous to build the T Tree indices for this join method, however, as it would then be slower then the other methods. In situations where one of the two relations is missing a join column index, the Hash Join method was found to be the best choice. There are only two exceptions to these statements:

(1) If a T Tree index or a hash index exists on the larger relation and the smaller relation size is very small (on the order of 5 to 10 percent of the larger relation size), then a Hash Join or Tree Join is preferred over the T Tree Merge Join. When both hash and tree indices exist, the Hash Join is preferred over the Tree Join.

(2) When the semijoin selectivity and the duplicate percentage are both high, the Sort Merge join method should be used, particularly if the duplicate distribution is highly skewed. A Tree Merge join is also satisfactory is this case, but



Graph 4.8 — Nested Loops Join

the required T Tree indices may not be present. If the indices must be built, then the Tree Merge join will be more costly than the Hash Join for duplicate percentages less then 60 percent in the skewed case and 80 percent in the uniform case.

It should be mentioned that only equijoins were tested. Non-equijoins other than "not equals" can make use of ordering of the data, so the Tree Join should be useful for such (<, ≤, >, ≥) joins.

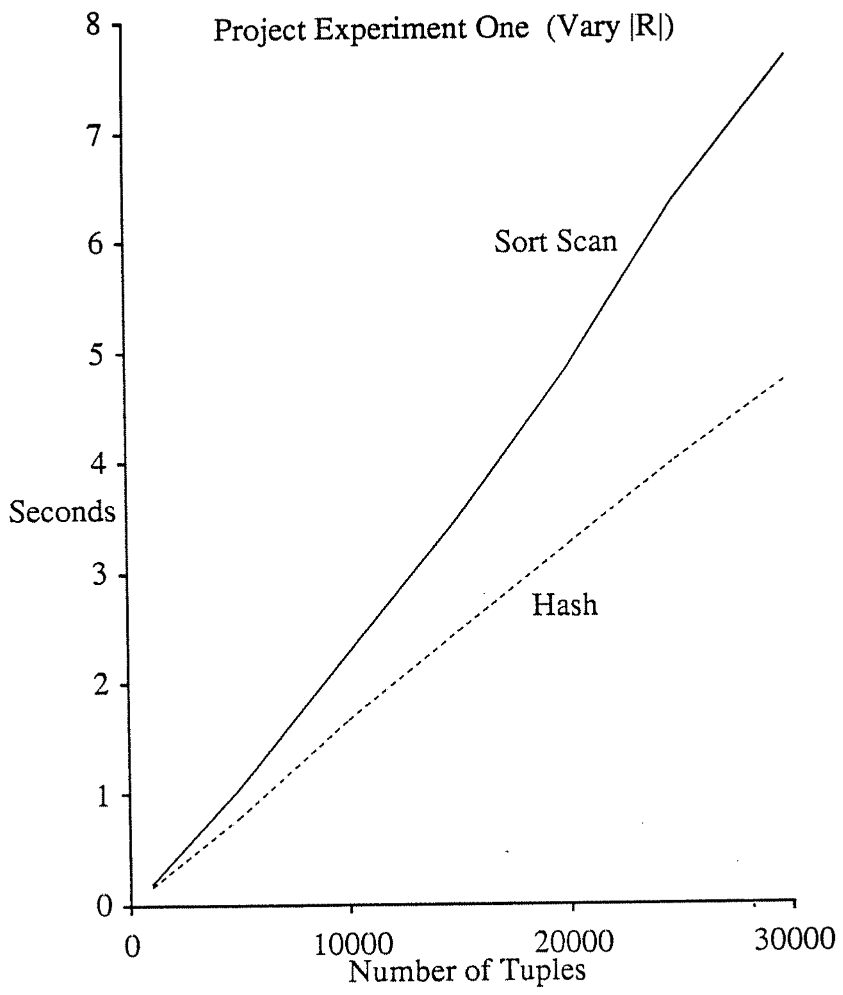
As mentioned earlier, we also tested the Nested Loops Join method. Due to the fact that its performance was usually several orders of magnitude worse than the other join methods, we were unable to present them on the same graphs. Graph 4.8 shows the cost of the Nested Loops Join for a portion of Experiment One, with |R1| = |R2| varied from 1,000 to 20,000. It is clear that, unless one plans to generate full cross products on a regular basis, Nested Loops Join should simply never be considered as a practical join method for a main memory DBMS.

The precomputed join described in Section 2.4.5 was not tested along with the other join methods. Intuitively, it would beat each of the join methods in every case, because the joining tuples have already been paired. The tuple pointers for the result relation can simply be extracted from a single relation. However, the links for this method must be maintained, so the benefits of their potential fast retrieval performance must be weighed against the potential overhead of managing them in an update environment.

4.2. Projection

In our discussion of the MM-DBMS in Chapter 2, we explained that much of the work of the projection phase of a query is implicitly done by specifying the attributes in the form of result descriptors. Thus, the only step requiring any significant processing is the final operation of removing duplicates. For duplicate elimination, we tested two candidate methods: Sort Scan [Bitto 83], based on the combination of quicksort and insertion sort described previously and Hashing [DeWit 84]. Again, we implemented both methods and compared their performance.

In these experiments, the composition of the relation to be projected was varied in ways similar to those of the join experiments — both the relation cardinality and its



Graph 4.9 — Vary Cardinality

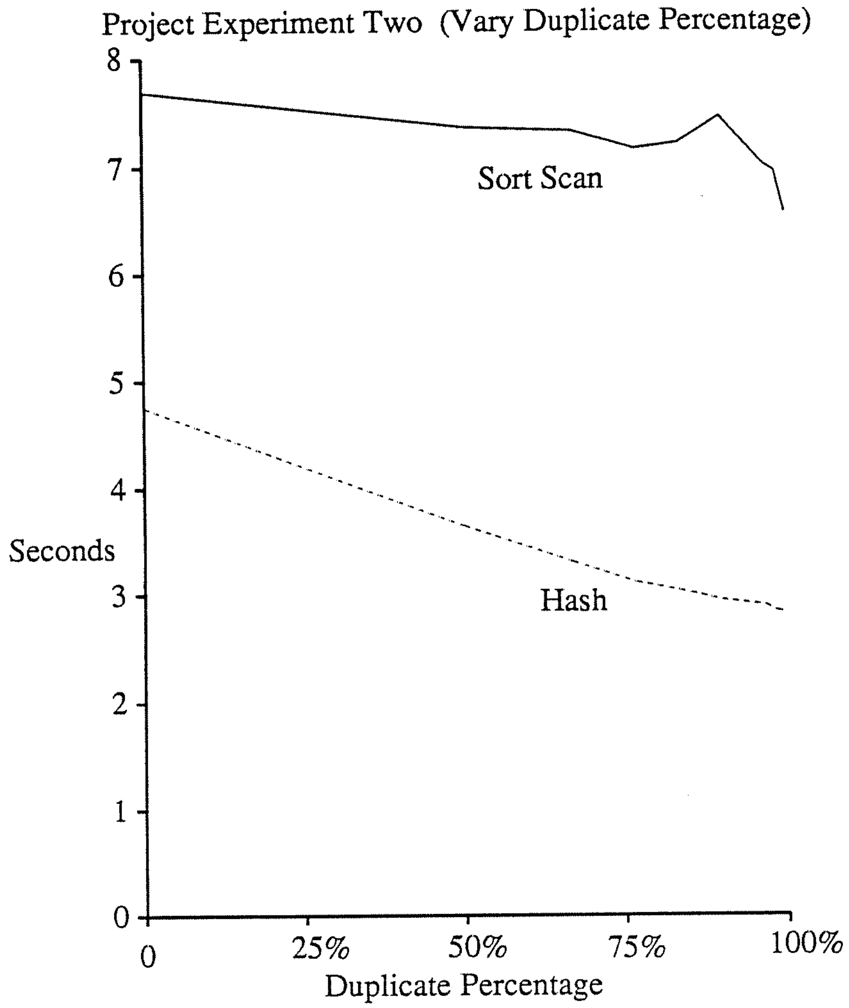
duplicate percentage were varied. Since preliminary tests showed that the distribution of duplicates had no significant effect on the results, we do not vary the distribution in the experiments presented here.

Graph 4.9 shows the performance of the two duplicate elimination algorithms for relations of various sizes. For this experiment, no duplicates were actually introduced in the relation, so the starting size of the relation and its final size were the same. The insertion overhead in the hash table is linear for all values of |R| (since the hash table size was always chosen to be $\frac{|R|}{2}$, while the cost for sorting goes as $O(|R| \; Log \; |R|)$. As the number of tuples becomes large, this sorting cost dominates the performance of the Sort Scan method. In addition, these experiments were performed using single column relations — the number of comparisons required is much higher in the sort process, and this cost would only be exacerbated, making the sort scan method relatively worse, if more columns participated in the projection. Thus, the Hashing method is the clear winner in this experiment.

Graph 4.10 shows the experiment results for a relation with 30,000 elements but a varying number of duplicates. As the number of duplicates increases, the hash table stores fewer elements (since duplicates are discarded as they are encountered). The Hashing method is thus able to run faster than it would with all elements in the result, as it has shorter chains of elements to process for each hash value. Sorting, on the other hand, realizes no such advantage, as it must still sort the entire list before eliminating tuples during the scan phase. The large number of duplicates does affect the sort to some degree, however, because the final insertion sort has less work to do when there are many duplicates — with many equal values, the subarray from quicksort is often already sorted by the time it is passed to the insertion sort.

4.3. Summary

In this chapter, we have examined query processing algorithms for a main memory database management system. We addressed the problem of processing the relational join and project operations in the MM-DBMS architecture. A number of candidate algorithms were implemented for each operation, and their performance was experimentally compared. For joins, when a precomputed join does not exist, a T Tree



Graph 4.10 — Vary Duplicate Percentage

based merge join offers good performance if both indices exist, and hashing tends to offer the best performance otherwise. A main memory variant of the sort merge algorithm was found to perform well for high output joins. Finally, it was shown that hashing is the dominant algorithm for processing projections in main memory.

Many aspects of query processing change when a memory-resident database is used. The issue of clustering and projection for size reduction is removed from consideration, thereby simplifying the choice of algorithms. Projection may still be needed to reduce the number of duplicate entries in a temporary result, but it will never be needed to reduce the size of result tuples, as tuples are *never* copied, they are only pointed to. And, since temporary results hold only tuple pointers, it is likely that they will be less expensive to create.

In light of these results, query optimization in an MM-DBMS should be simpler than in conventional database systems, as cost formulas will be less complicated [Selin 79]. There are three possible access paths for selection, hash lookup, tree lookup, or sequential scan through an unrelated index; three main join methods, precomputed join, Tree Merge join, and Hash Join; and one method for eliminating duplicates, Hashing. Moreover, the choice of which algorithm to use is simplified because there is a more definite ordering of preference: a hash lookup (exact match only) is always faster than a tree lookup, which in turn is always faster than a sequential scan; a precomputed join is always faster than the other join methods; and a Tree Merge join is nearly always preferred when the T Tree indices already exist.

Chapter 5

Recovery

The issue of making data recoverable has been studied extensively for disk-based database systems [Astra 76], [Stone 76], [Gray 78], [Verho 78], [Linds 79], [Lamps 79b], [Kohle 81], [Gray 81], [Haerd 83], [Reute 84], [Elhar 84], [Schwa 84]. However, a number of new and interesting problems appear when the primary copy of the database is placed in volatile storage. With a memory-resident database, the *entire* database is in jeopardy rather than just a small portion as in a disk-based database. In this chapter, the problems related to the recovery of memory-resident database systems are discussed. Several existing disk-oriented recovery methods are described briefly and their shortcomings for use in a memory-resident database system are listed. Then, several recent papers on large buffer or memory-resident database systems are discussed and their shortcomings are also listed. Finally, a new proposal for a memory-resident database system recovery component is presented and shown to perform well in a high-performance transaction processing environment.

5.1. Traditional Recovery Methods

In [Reute 84], Reuter compares twelve recovery algorithms. The four best algorithms are described in detail and have also appeared in the literature as [Astra 76], [Linds 79], [Elhar 84], [Reute 80]. Each algorithm focuses on a different aspect of recovery such as efficient UNDO processing, efficient logging, or minimal processing during recovery. A brief description of each algorithm follows.

The first recovery algorithm of interest is the one used in System R, a research prototype developed at IBM. System R uses write-ahead logging in combination with shadow pages [Astra 76], [Lorie 77], [Gray 81]. Commit processing is done with write-ahead logging (WAL), and checkpoints are taken when the database system is in an RSS level action consistent state. A checkpoint installs the current copy of the database as the shadow copy. It updates page tables and allocation maps and flushes them to disk with the dirty pages in the buffer pool. The cost of the page table manipulation could be significant, as it is possible that the tables may have to be read in from the disk before they can be modified and written back to disk. Checkpoints of the database are taken at regular intervals and at operator command. Reflection on this design by the implementors suggests that shadowing is a very expensive process and that logging alone, although not available as an option in their system, would probably be sufficient [Chamb 81], [Gray 81].

The algorithm proposed in "Notes on Distributed Databases " by Lindsay et. al. [Linds 79] attempts to minimize normal operating costs for recovery. Entry-level log records are written to the log using a write-ahead-log protocol. Updated pages are not propagated to the database atomically upon commit; instead, they are propagated separately as they are swapped out of the buffer to make room for other pages. Since dirty uncommitted pages can also be flushed to disk, UNDO information, as well as REDO information, is logged. Checkpoints are taken periodically; a checkpoint operation records the Log Sequence Numbers of the current set of pages in the buffer pool so that, during crash recovery, the recovery manager can read this list and know which pages need to be recovered. As an optimization to bound recovery time, any page that stays in the buffer pool for a specified number of checkpoint intervals is written to the database disk.

The Database Cache [Elhar 84] was designed to use a large amount of memory as a disk cache. The large buffer allows the Database Cache to hold all of a transaction's pages until commit, thereby keeping UNDO information out of the permanent log and dirty uncommitted data out of the permanent database. The effects of an uncommitted transaction can be undone quickly; since all updated copies of pages are kept in memory, they can simply be thrown away. Committing a transaction is also fast because, although the Database Cache forces all of the transaction's dirty pages to disk upon commit, it schedules all of the disk writes at once and is able to make use of chained disk writes, which are significantly faster than separate disk writes [Reute 84]. Recovering the database after a crash is straightforward, as the only action required before allowing normal transaction processing to resume is to read the contents of the safe (a stable copy of the database cache) back into the memory cache — an operation done at disk transfer speeds. Some drawbacks of the Database Cache are that two-phase page-level locks are required and that regular write-ahead-logging is needed for transactions that are very long. Page-level locks can potentially restrict concurrency by allowing transactions to lock too much data. In the case of a long transaction, the cache can't hold all of its pages so some of them must be written to disk, thus requiring the use of UNDO log records and write-ahead-logging protocols. Also, although some logging cost is reduced by eliminating the writing of UNDO information to disk during *normal* processing, the Database Cache algorithm uses page-level REDO log entries rather than record-level log entries, thus incurring a write cost that is much more than the potential minimum cost, as writing entire pages to disk typically entails transferring one to two orders of magnitude more data than writing log records.

Finally, there is the TWIST algorithm devised by Reuter [Reute 80]. It was designed for fast UNDO processing and fast recovery. It uses a shadow page scheme, allocating *two* disk pages, located physically close to each other, for every database page, thus avoiding the de-clustering problem of the System R shadow mechanism.

Like the Database Cache, undoing a transaction is simple, as it only involves discarding the updated memory pages, but, also like the Database Cache, it writes whole pages and assumes two-phase page-level locking.

In [Reute 84], Reuter analyzes twelve algorithms and concludes that the algorithm presented in [Linds 79] (the second algorithm described earlier) is the best general algorithm. Reuter describes it as a *non-atomic* (updates are propagated to the database over several actions), *steal* (dirty pages are allowed to be written to the database before commit to make room for other pages), *non-force* (updates are not flushed to the database upon commit), *fuzzy dump* (the stable disk state of the database does not correspond to any instantaneous state of the actual database) algorithm. This algorithm is the most efficient because it does the least amount of logging (entry-level versus page-level) and it checkpoints a minimal amount of information — it writes out only those pages that remain in the buffer during several checkpoint intervals. In contrast, System R regularly flushes *any* dirty pages in the buffer along with page tables and allocation maps that may need to be read in and modified first. The other algorithms, Database Cache and TWIST, perform a transaction-oriented checkpoint after every transaction by writing out entire pages that hold dirty data, thus incurring a large I/O cost when there is a large amount of update activity. With Lindsay's algorithm, a page write is amortized over several transactions, and a log record, rather than a page, is written for each value updated.

Unfortunately, although the traditional algorithms would *function* correctly for a memory-resident database system, they would probably not *perform* satisfactorily. The Database Cache and TWIST algorithms write out too much data after each transaction. Although System R does not force its shadow pages after every transaction, it still writes out all dirty pages frequently, thus incurring a smaller but similar cost as the other shadow schemes. Lindsay's algorithm, although the best of the four mentioned, is not quite appropriate for a memory-resident system. Its list of pages that reside in the "buffer" over several checkpoint intervals would consist of all the pages in the database. Also, after a crash, the entire database would need to be reloaded and recovered from the log before transaction processing could begin. The logging, checkpointing, and recovery algorithms for a memory-resident database system should be designed to log and checkpoint the database efficiently. Current work on recovery algorithms for large memory database systems has addressed these issues.

5.2. Previous and Current Work on Main Memory Recovery

There has been some previous work [IBM 79], [DeWit 84] on main memory database systems, but much of the research in this area is current [Lelan 85], [Amman 85], [Eich 86], [Hagma 86], [Salem 86]. IMS FASTPATH [IBM 79], [IBM 84] uses a main store database that is treated differently from the rest of the disk-oriented IMS database. IMS FASTPATH was the first system to use the idea of *commit groups* to reduce the amount of time needed to flush log records to disk during the commit process. Log records are not flushed immediately upon commit, instead, they are collected in a buffer with other committed records so that the cost of writing the log page to disk can be amortized over several transactions. IMS FASTPATH performs no writes to the database during normal processing — the stable copy of the database is refreshed with a transaction-consistent version of the memory copy of the database during system shutdown, or some other infrequent checkpoint [Haerd 83].

DeWitt et. al. [DeWit 84] point out the possibility of using stable memory, as well as commit groups, to speed up the commit process by reducing the amount of time a transaction must wait until its log records are safely flushed to stable storage. They also describe a method, based on keeping multiple versions of data, to get action-consistent checkpoints without quiescing the system during a checkpoint operation. A memory-resident table holds an entry for every page in the database along with an entry for a possible shadow page. Transactions that update pages participating in the checkpoint create new memory images of those pages and enter them in the table, thereby leaving a shadow copy of each page behind for use by the checkpoint process. The collection of checkpointed pages is written to disk as a temporary log and held there while the memory copies of the checkpointed pages are written to the disk in-place. Once all the checkpointed pages have been written to disk, the temporary disk copy and the memory copies of the checkpointed pages are released. Checkpoints are taken as often as possible to reduce the amount of log information.

Hagmann [Hagma 86] sketches a method for fast recovery of medium-size main memory databases (on the order of 100 megabytes) that uses fast, frequent checkpoints to keep the log size small. Recognizing that checkpointing the entire large memory database can be costly if done a page at a time, Hagmann suggests writing the entire database to disk at the fastest possible rate. Reading the entire database as fast and as often as possible will cause the backup copy of the database to be almost up to date with the memory copy. The checkpoint process performs no synchronization with concurrent transactions, so the checkpoint copy of the database does not represent a consistent view of the database at *any* level. To support this type of fuzzy dump, the UNDO and REDO log information must be in physical before-image or after-image form, and every entity must be locked using two-phase locking to maintain a serializable schedule. Larger objects such as queues or index structures would need to be locked in their entirety, thus restricting concurrency [Schwa 84]. Also, logging would need to be done at the page or object level, which would require much more log data to be written than if logging were done at the record level.

Eich [Eich 86] outlines a recovery method that uses main memory shadow pages, pre-committed transactions (as in [DeWit 84]), a random access log, automatic checkpointing, and a recovery processor. Transactions do not update the main memory database in-place; instead they create new versions of any pages that will be modified. Undoing the effects of a transaction only involves releasing the newly created pages, and committing a transaction involves installing the newly created pages in the database. Mechanisms to allow transactions to see their own updates and to coordinate the page tables for relations are not discussed, but it appears that they would increase the complexity and overhead of the recovery mechanism during normal processing, as every database reference would need to use the page table mechanism to get the correct version of the page. To get transaction consistent checkpoints of the entire database, Eich proposes an automatic checkpointing mechanism that runs on a recovery processor. It waits for the database system to become quiescent, then blocks out other transactions while it writes the database to disk. Between checkpoints, the recovery processor reads the log and applies the log records to the disk version of the database, attempting to keep the disk version of the database up to date with the memory version. There are two design points in this proposal that appear questionable. The first is that a high performance main memory database system seems unlikely ever to be quiescent except during a crash or system shutdown, thus causing checkpoints to occur infrequently and long streams of log information to accumulate. The second point assumes that one can apply updates to a disk-resident database at the same rate that one can apply updates to a memory-resident database. This might work for transactions that use a large amount of computation to produce relatively few updates to the database, but low computation, update-intensive operations are likely to proceed much faster in a memory-resident database system than in a disk-resident database system.

IBM's OBE project [Amman 85] uses a memory-resident design, but it is based on virtual memory rather than physical memory. OBE uses the direct addressability features of a memory-resident database architecture, but it reads relations and indices from disk *via* the virtual memory mechanism. OBE is designed to minimize the number of disk accesses performed during a transaction; every relation participating in a transaction is read once at the start of the transaction, and every relation that is updated is written once, in its entirety, at the end. OBE keeps shadow copies of relations; the new copy of a relation is installed atomically during commit. The cost of flushing entire relations and indices at the end of every transaction appears high, and it would probably be prohibitive in a high-performance database system environment.

Salem and Garcia-Molina [Salem 86] propose a hardware logging device (HALO) that monitors the main CPU, intercepts word-level writes to the database, and logs them before passing them on to the database system. HALO contains stable random access memory, so it does not need to synchronize with disks before allowing transactions to commit. Another hardware-oriented proposal is made by Leland and Roome [Lelan 85]. They propose using a large memory (1 to 2 gigabytes) that is made completely stable through the use of battery-backup memory. They do not comment on the expense and reliability of this large stable memory.

Although not necessarily intended for use in a memory-resident database system, an algorithm for "on the fly" consistent checkpointing is presented by Pu in [Pu 85]. Pu states that the database does not need to be quiesced to capture a consistent checkpoint image; instead, transactions that do not interfere with the checkpoint process's consistent view of the database are allowed to run. During a checkpoint operation, the database is colored “read” or “not read yet”. A transaction can continue processing as long as it modifies only information that has already been read by the checkpoint process or hasn’t been read yet. Transactions that have written “read” data must wait before writing “not read yet” data until after the checkpoint process has read it. Transactions that have modified “not read yet” data and desire to write “read” data are aborted, as they would cause the checkpoint image to be inconsistent.

5.3. A New Recovery Proposal

Previous work in main memory database recovery has produced several useful ideas, but these individual ideas have not yet been combined into one design. And, along with the useful ideas that have been generated so far, there are still several aspects of memory-resident database recovery that need better, more efficient algorithms. We discuss the useful ideas for memory-resident recovery, point out what is still needed, and then describe a design that meets those needs.

5.3.1. Motivation

The logging process is the most visible component of the recovery mechanism, hence it is no surprise that it has received the most attention thus far. It is important that the main processor do as little work as possible to accomplish logging activities, as its CPU cycles are the most important resource in a memory-resident database system. And, in a high-performance system, transaction response time is important, so the time requirement for the commit phase must be small. Several methods for reducing CPU cost and improving response time in logging have been proposed. First, the requirement that a transaction cannot commit until its log records have been flushed to disk is removed by the use of stable random access memory, as once the log records are entered there, the transaction can commit. Second, a recovery processor can reduce the amount of time the main CPU spends on logging activities. Transactions need to do little more than write their log records to stable storage, as the recovery processor is responsible for safely transporting the log records to disk.

Less attention has been focused on producing efficient checkpoint algorithms. Checkpoints of the database are needed to limit the amount of log information, as that bounds the time needed to recover the database after a system crash. Checkpoint operations write parts of the memory copy of a database to disk, thus making the log information for those parts of memory redundant. One proposal for a checkpoint operation suggests writing the entire database to disk as often as possible. Better checkpoint algorithms write only memory pages that have been modified since the last checkpoint to disk, thus avoiding a considerable amount of disk activity if the modified portion of the database is a small percentage of the whole database. If checkpoint operations could be more efficient, then they would write to disk only those pages of the database that had acquired a certain threshold amount of log information. (This threshold amount is determined by the time one is willing to spend to recover the database.)

Like existing checkpoint proposals, post-crash recovery proposals for memory-resident database systems suggest loading memory with a checkpoint version of the *entire* database; then log information is used to bring it up to its state just before the crash. Loading a large database into memory from disk can be a time consuming operation. Transactions may needlessly wait for the entire database to be loaded, as they often reference only a small portion of the database. Unfortunately, transactions cannot proceed as soon as their information is loaded from a checkpoint image because they must wait until their required information is also restored to its most recent state from the log. Ideally, after a system crash, the recovery mechanism should allow transactions to run as soon as possible, recovering *on demand* those portions of the database that are needed by the transactions waiting to run.

5.3.2. Overview of Proposal

The proposed design for a memory-resident database system recovery component uses two independent processors, a main processor and a recovery processor; stable memory comprising two different log components, a Stable Log Buffer and a Stable Log Tail; and disk memory to hold both a checkpoint copy of the database and log information. The two processors run independently and communicate through a buffer area in the Stable Log Buffer.

The main CPU performs regular transaction processing — its only logging function is to write a transaction's log records to the Stable Log Buffer. The recovery manager, running on the recovery CPU, reads records in the Stable Log Buffer that belong to committed transactions and places them into bins (called *partition bins*) in the Stable Log Tail according to their partition address. (Recall that a partition is the fixed-size unit of memory allocation used to hold portions of relations and indices.) Each partition having outstanding log information is represented in the Stable Log Tail by such a partition bin. Each partition bin holds REDO log records and other miscellaneous log information pertaining to its partition. Partitions having outstanding log information are referred to as *active*.

As partition bins become full, they are written out to the log disk; log pages for a given partition are chained together. Grouping log records according to their corresponding partitions allows the recovery manager to keep track of the update activity on individual partitions. When a partition has accumulated a specified threshold count of log records, it is marked to be checkpointed, and thus the cost of checkpoint operations are amortized over several update operations. Grouping the log records also has another significant advantage — it allows partitions to be recovered independently.

After a crash, the recovery manager restores the database system catalogs and then signals the transaction manager to begin processing. Each transaction, at the outset, declares the set of relations and indices that it will be using. The transaction manager checks to make sure these objects are available, and, if not, initiates recovery transactions to restore them on a per partition basis. Transactions that already have their required information in memory are allowed to run.

In the next sections the supporting hardware for the recovery mechanism is presented. Then, details of the recovery algorithms, regular logging, checkpointing, recovery, and archive logging are described. Finally, a simple performance analysis of the logging, checkpointing, and recovery methods is provided to show that these algorithms provide reasonable performance in a memory-resident database system.

5.3.3. Description of Hardware Organization

The architecture for the recovery mechanism is composed of a recovery CPU, several megabytes of reliable stable memory and a set of disks. The recovery CPU has access to all of the stable memory, and the main CPU has access to at least part of the stable memory. The two CPUs *could* both address all of memory (both nonvolatile and volatile), or they could share only the address space of the Stable Log Buffer. To allow more flexibility in the actual hardware design, the two CPUs are required to share only the address space of the Stable Log Buffer, using it as a communication buffer along with its other uses. The disks are divided into two groups and used for different purposes. One set of disks holds checkpoint information and the other set of disks holds log information. The recovery CPU manages the log disks, and both the recovery CPU and the main CPU manage the checkpoint disks. Figure 5.1 shows the basic layout of the system.

The two processors have logically different functions. The main CPU is responsible for transaction processing, while the recovery CPU manages logging, checkpointing operations, and archive storage. Although the two sets of tasks could be done by a single processor, it is apparent that there is a large amount of parallelism possible. Indeed, although only two processors are mentioned, each CPU could even perhaps be a multiprocessor which further exploits parallelism in its own environment. In

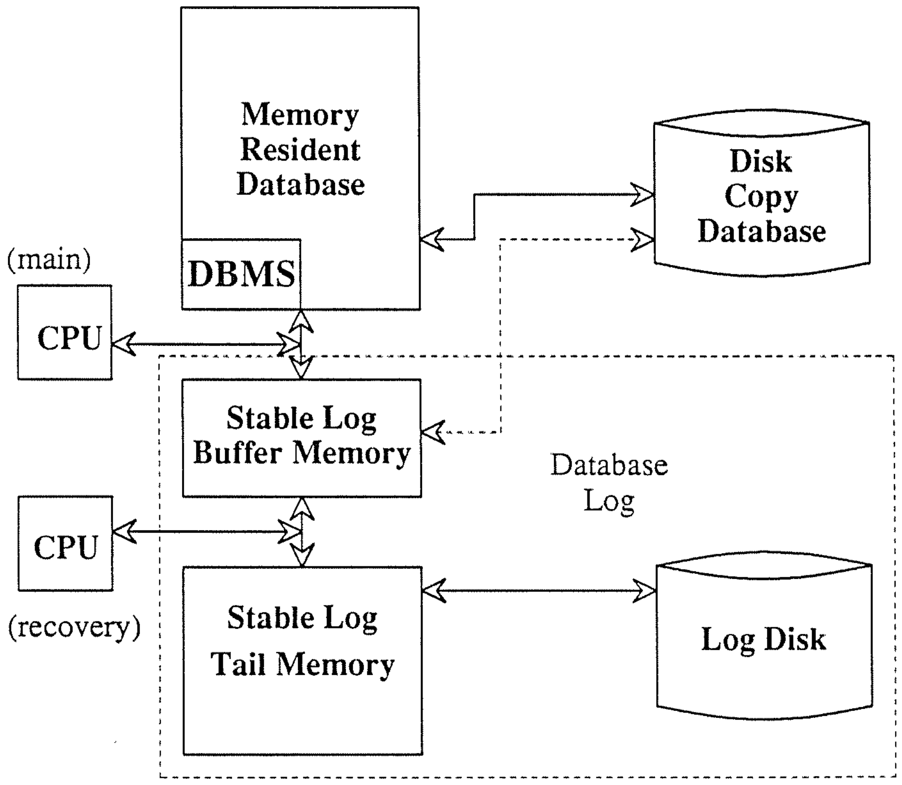


Figure 5.1 — Recovery Mechanism Architecture

discussing the design, however, we refer to only two CPUs.

5.3.4. Regular Logging

The normal logging procedure is composed of three tasks. First, transactions create both REDO and UNDO log records; the REDO log records are placed into the Stable Log Buffer, and the UNDO records are placed into a volatile UNDO space. Second, the recovery manager (running on the recovery CPU) reads log records belonging to committed transactions from the Stable Log Buffer and places them into partition bins in the Stable Log Tail. Last, the partition bin pages of REDO log records in the Stable Log Tail are written to disk when they become full. These three steps are discussed in more detail below.

5.3.4.1. Writing Log Records

Transactions write log records to two places: REDO log records are placed in the Stable Log Buffer (SLB) and UNDO log records are placed in the volatile UNDO space. REDO log records are kept in stable memory so that transactions can commit instantly — they do not need to wait until the REDO log records are flushed to disk. UNDO log records are *not* kept in stable memory because they are not needed after a transaction commits — the memory-resident database system does not allow modified uncommitted data to be written to the stable disk database. A log record corresponds to an entity in a partition: a relation tuple or an index structure component. An entity is referenced by its memory address (Segment Number, Partition Number, and Partition Offset).

Both the volatile UNDO space and the Stable Log Buffer (SLB) are managed as a set of fixed-size pages. These pages are allocated to transactions on a demand basis, and a given page will be dedicated to a single transaction during its lifetime. As a

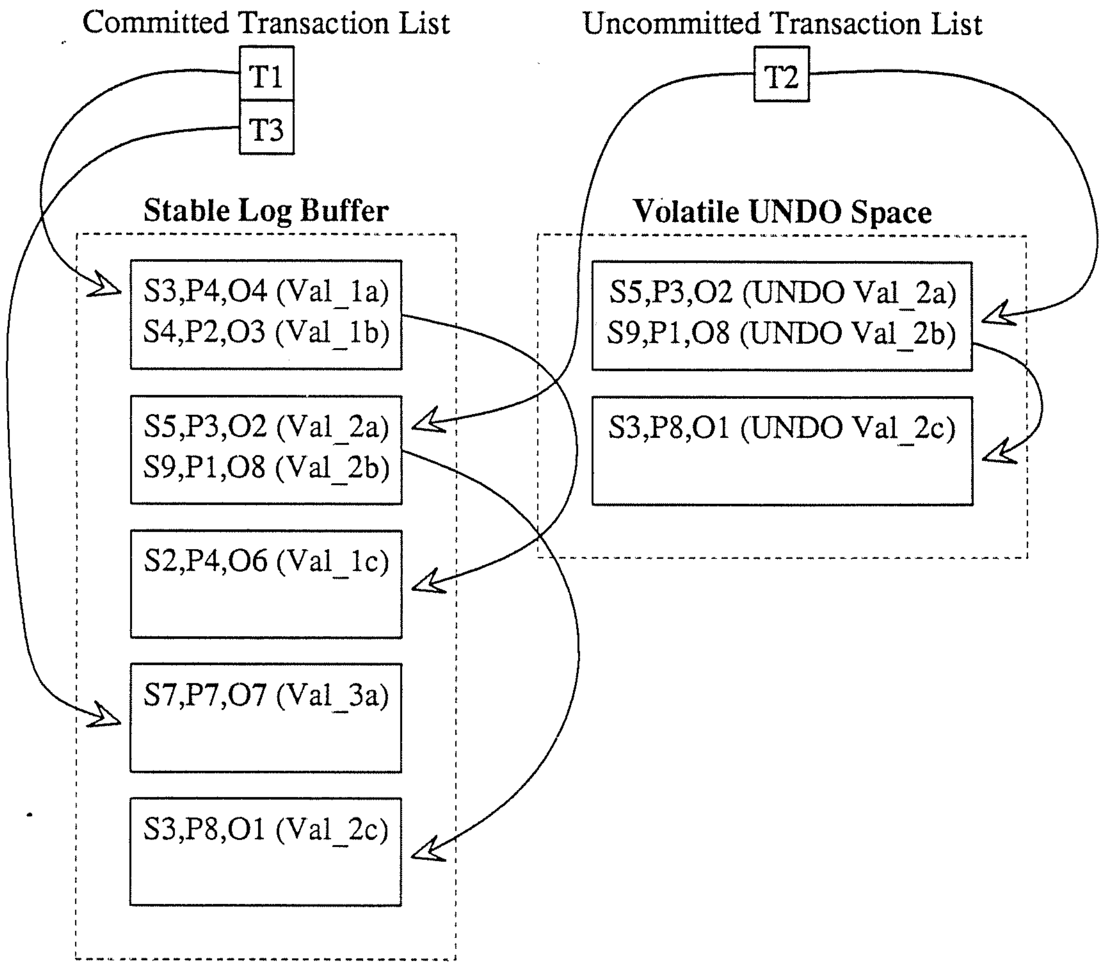


Figure 5.2 — Writing Log Records

result, critical sections are used only for page allocation — they are not a part of the log writing process itself. The chains of log pages for a transaction will appear on one of two lists, the committed transaction list or the uncommitted transaction list. When a transaction commits, its REDO log page chain is removed from the uncommitted list and appended to the committed transaction list, and its UNDO log page chain is discarded (Figure 5.2). The committed transaction list is maintained in commit order so that the log records can be sent to disk in committed order.

Log Record Format

Log records have different formats depending on the type of database entity they correspond to (relation tuples or index components) and the type of operation they represent. All log records have three main parts:

TAG	Bin Index	Operation

TAG refers to the type of log record, *bin index* is the index into the partition bin table where the log record will be relocated, and *operation* is the REDO operation for the entity. The bin index is a direct index into the partition bin table, and it is used to locate the proper partition bin log page for an entity's log record. Partitions maintain their partition bin index entries in their control information, so the bin index entries are easily located given the address of any of their entities. (The next section discusses this in more detail.)

Relation log records specify REDO values for specific entities, so in one sense they are *value* log records. However, they may also specify operations that entail updating the string space in a partition, which is managed as a heap and is not locked in two-phase manner, so relation log records are really *operation* log records on a partition. Index log records specify REDO operations to index components (T Tree nodes, hash nodes, or Modified Linear Hash tables). A single update operation may affect several index components, so a log record must be written for each updated index component. To maintain serializability and allow for the ability to undo transactions, index components and relation tuples are locked with two-phase locks [Eswar 76] that are held until transaction commit.

5.3.4.2. Grouping Log Records by Partition

The main purpose of the Stable Log Tail (SLT) is to provide a stable storage environment where log records can be grouped according to their corresponding partitions. The recovery manager uses this for several things:

(1) The log records corresponding to a partition are collected in page units and written to disk. The log pages for a partition are linked, thus allowing all the pages of a particular partition to be located easily during recovery.

(2) The number of log records for each partition is recorded, so the checkpoint mechanism can write to disk those partitions having a specified amount of log

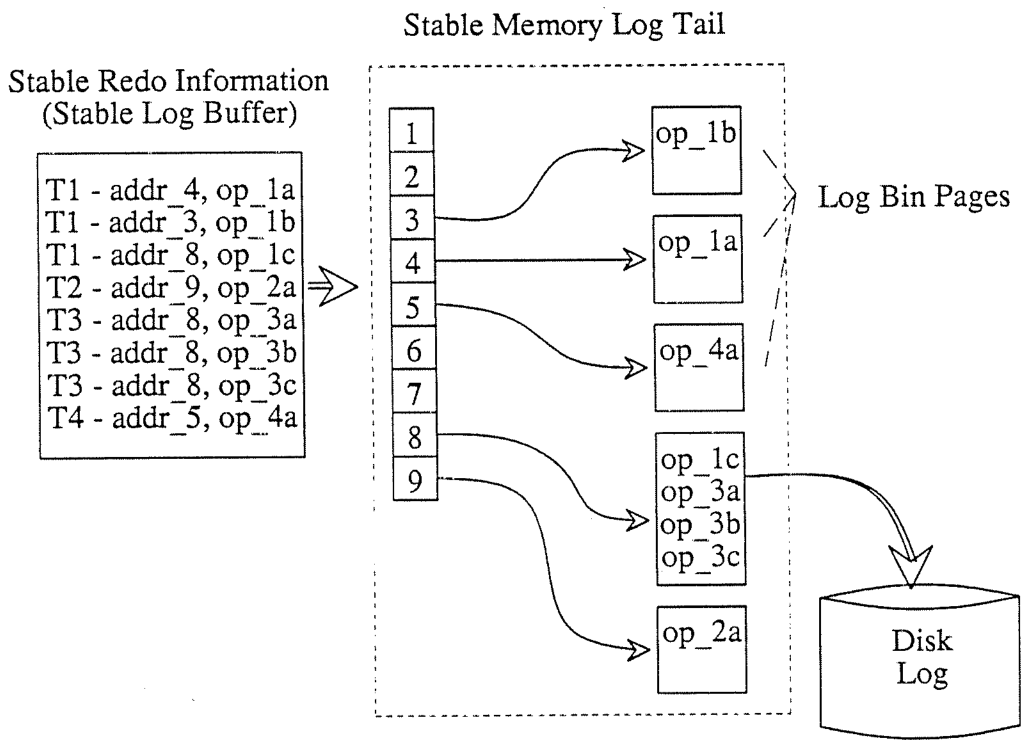


Figure 5.3 — Regular Transaction Logging

information, and thus amortizing the cost of a checkpoint operation over many update operations. Once a partition has been checkpointed, its corresponding log information is no longer needed for memory recovery.

(3) Redundant address information may be stripped from the log records, thereby condensing the log information.

Log records are read from the Stable Log Buffer and placed into partition bins in the Stable Log Tail (Figure 5.3). Each log record is read, its bin index field is used to calculate the memory address of the log page of the record's partition bin, and the log record is copied into the log page. There are two main possibilities for the log bin table: there could be an entry in the table for every existing partition in the database, or there could be an entry for every active partition. (Recall that an active partition is one that has been updated since its last checkpoint, so it has outstanding log information.) Since the bin index should not be sparse, bin index numbers must be allocated and freed, like fixed blocks of memory. If every partition were represented in the log bin table, then the bin index numbers would be allocated and de-allocated infrequently — only as often as partitions are allocated and de-allocated. However, if only the *active* partitions were represented in the log bin table, then the bin index numbers would be allocated when partitions are activated and reclaimed when they are de-activated, thereby causing the bin index number resource manager to be activated more frequently. For simplicity in design, we assume that each partition has a small permanent entry in the partition bin table. This requires an information block in the Stable Log Tail for each partition in the database, but only active partitions require the much larger log page buffer.

Each partition has an information block in its log bin containing the following entries:

- Partition Address(Segment Number, Partition Number)
- Update Count
- LSN of First Log Page
- Log Page Directory

The *Partition Address* is attached to each page of log records that is written to disk. The entry serves as a consistency check during recovery so that the recovery manager can be assured of having the correct page. It also allows the log pages of a partition to be located when the log is used for archive recovery.

The *Update Count* and the *LSN of First Log Page* are monitors used to trigger a checkpoint operation for a partition. The update count reflects the number of log records that have accumulated for a partition. When the update count exceeds a predetermined threshold, the partition is marked for a checkpoint operation. The Log Sequence Number of the first log page of a partition shows when the partition's first log page was written; it is the address of the oldest of the partition's log pages. When a partition is infrequently updated, it will have few log pages and they will be spread out over the entire log space. The available log space remains constant, and it is reused over time. The log space holding currently active log information is referred to as the *log window*. The log window is a fixed amount of log disk space that moves forward through the total disk space as new log pages are written to it, thus some active log information may fall off the end. To allow log space to be reused, partitions having old log information that is falling off the end of the log window are checkpointed. (There will actually be a grace period between when the checkpoint is triggered and when the log space really needs to be reused.)

If the log space were infinite, all partition checkpoints would be triggered by the *update count*. However, since the log space is finite, infrequently updated partitions will have to be checkpointed before they accumulate a sufficient number of updates.

In this case, we say that those partitions were checkpointed because of *age*. This works as follows: The recovery manager maintains an ordered list of the first log pages of active partitions. Whenever the log window advances, due to a log page being written, this *First LSN* list is checked for any partition whose first log page extends beyond the log window boundary. When a partition becomes active it is placed on the First LSN list, and when it is checkpointed it is removed from the list. The head of the list holds the oldest partition, so only a single test on this list is necessary when checking for the possibility of generating a checkpoint for age.

The *Log Page Directory* holds pointers to a number of log pages for a given partition (Figure 5.4). During recovery, REDO log records must be applied in the order that they were originally written. If log pages were chained from the most recently written to the least recently written, the reverse of the order *needed*, then log records could not begin to be applied until the last page was read (which is the first page that would be needed). A directory allows the log pages to be read in the order that they are needed during recovery. The size of the directory is chosen to be equal to the average number of log pages for an active partition, so that, during recovery, it should often be the case that the log pages will be able to be read in order. This allows the log records of one page to be used while the log records of the next page are being read from the log disk. If fewer than N (the directory size) log pages have been written, the directory holds all of the log pages for the partition (Figure 5.4(a)). When more than N log pages have been written, the directory is stored in every N^{th} log page.

5.3.4.3. Writing Log Records to the Log Disk

When the log records of a partition fill up a log page, the records are written to the log disk. The recovery CPU issues a disk write request for that page and allocates

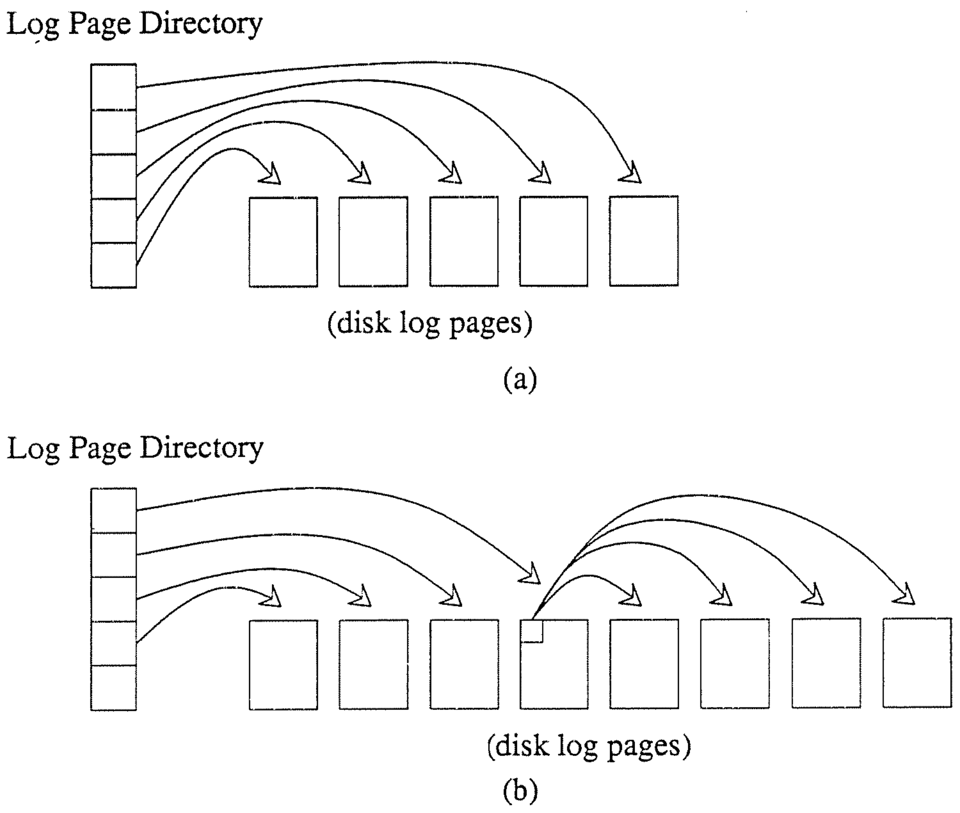


Figure 5.4 — Log Page Directories

another page to take its place. (The memory holding the old page is released after the disk write has successfully finished.) The recovery CPU can issue a disk write request with little effort because it is a dedicated processor, it is using real memory, and it is running a single thread of execution. All it needs to do is append a write request to the disk device queue that points to the memory page to be written.

5.3.5. Regular Checkpointing

As explained earlier, the main purpose of a checkpoint operation is to bound the log space for partitions by writing to disk those partitions that have a predefined number of log records. Its secondary purpose is to reclaim the log space of partitions that have to be checkpointed because of age. When the recovery manager (running in the recovery CPU) determines that a partition should be checkpointed, either due to update count or age, it tells the main CPU that the partition is ready for a checkpoint *via* a communication buffer in the Stable Log Buffer. The recovery manager enters the partition's address in the buffer along with a flag that represents the status of the checkpoint for that partition; initially this flag is in the *request* state, it changes to the *in-progress* state while the checkpoint is running, and it finally reaches the *finished* state after the checkpoint transaction commits. A finished state entry is a signal to the recovery CPU to flush the remaining log information for the partition from the Stable Log Tail to the disk log.

After a partition has been checkpointed, though its log information is no longer needed for memory recovery, it cannot be discarded because the information is still needed in the archive log to recover from media failure. If the partition has any log records remaining in the Stable Log Tail, they are flushed to the log disk. If only a partial page of log records is present, the log records are copied to a buffer where they are combined with other log records to create a full page of log information, thereby saving log space and disk transfer time by writing only full or mostly full pages to the log.

Partition checkpoint images could be kept in well-known locations on the checkpoint disks, but that would require a disk seek to a partition's checkpoint image location for each checkpoint. Instead, the checkpoint images are simply written to the first available location on the checkpoint disks and a partition's checkpoint image location (in the relation catalog) is updated after each checkpoint. Therefore, for performance reasons, the disks holding the partition checkpoint images are organized in a pseudo-circular queue. Frequently updated partitions will periodically get written to new checkpoint disk locations, but read-only or infrequently updated partitions may stay in one location for a long time. (We use a pseudo-circular queue rather than a real circular queue so that partitions that are rarely checkpointed don't move and get "jumped" over as the head of the queue passes by.) The checkpoint disk space should be large enough so that there will be sufficiently many free locations, or "holes" available to hold new checkpoint images.

A map of the disks' storage space is kept in the system catalogs to allow transactions to find the next available location for writing a partition. New checkpoint copies of partitions *never* overwrite old copies. Instead, the new checkpoint copy is written to a new location (the head of the queue), and installed atomically upon commit of the checkpoint transaction. The relation catalog contains the disk locations of these checkpoint copies so that they can be located and used to recover partitions after a crash.

The steps of the checkpoint procedure are as follows:

(1) The recovery CPU issues a checkpoint request containing a partition address and a status flag in the Stable Log Buffer.

(2) The transaction manager, running on the main CPU, checks the checkpoint request queue in the Stable Log Buffer between transactions. For each partition checkpoint request that it finds, it starts a checkpoint transaction to read the specified partition from the database and write it to the checkpoint disk, and it also sets the checkpoint status flag to *in-progress*.

(3) The checkpoint transaction sets a read lock on the partition's relation and waits until it is granted. Notice that a single read lock on a relation is sufficient to ensure that its relation partitions and its index partitions are all in a *transaction consistent* state; thus, only committed data is checkpointed. (Finer granularity locks such as partition-level locks would possibly allow more sharing during a

checkpoint operation, but checkpoint operations are relatively short and, as shown in Chapter 6, partition-level locks are problematic.)

(4) When the read lock on the partition's relation is granted, the checkpoint transaction allocates a block of memory large enough to hold the partition, copies the partition into that memory, and releases the read lock. Relation locks are held just long enough to copy a partition at memory speeds, so checkpoint transactions will cause minimal interference with normal transactions. The checkpoint transaction then locates a free area on the checkpoint disk to hold the partition. (Since multiple checkpoint transactions may be executing in parallel, a write latch on the disk allocation map is required for this.)

(5) The updates to the disk allocation map and the partition's catalog entry are logged before the partition is actually written to disk. Checkpoints to catalog partitions are done in a manner similar to regular partitions, except that their disk locations are duplicated in stable memory and in the disk log, since catalog information must be in a well-known place so that it can be found easily during recovery.

(6) The partition is written to the checkpoint disk and the checkpoint transaction commits. The memory buffer holding the checkpoint copy is released, the new disk location for the partition is installed in its catalog entry, and the status of the checkpoint operation is changed to *finished*.

(7) The recovery manager, on seeing the finished state of the checkpoint operation, flushes the partition's remaining log information from the Stable Log Buffer to the log disk.

5.3.6. Post-Crash Memory Recovery

Since the primary copy of the database is memory-resident, the only way a transaction can run is if the information it needs is in main memory. Restoring the memory copy of the database entails restoring the catalogs and their indices, and then using the information in the catalogs to restore the rest of the database on a demand basis. The information needed to restore the catalogs is a list of catalog partition addresses, and it is kept in a well-known location; it is stored twice, in the Stable Log Buffer and in the Stable Log Tail, and it is periodically written to the log disk. Once the catalogs and their indices have been restored, regular transaction processing can begin.

During its initialization phase, a transaction declares the set of relations and indices that it will need. The transaction manager checks the relation catalog for these entries to see if they are memory-resident. If they are not, it initiates a set of recovery transactions to recover them, one per partition. A relation catalog entry contains a list of partition descriptors that make up the relation, so the recovery transaction knows which partitions to recover. Each descriptor gives the disk location of that partition along with its current status (memory-resident or disk-resident).

A recovery transaction for a partition reads the partition's checkpoint copy from the checkpoint disk and issues a request to the recovery CPU to read the partition's log records and place them in the Stable Log Buffer. Once the partition and the log records are available, the log records are applied to the partition to bring it up to its state preceding the crash. (Actually, log records can be applied in parallel with reading log pages if the pages can be read in the correct order.) Then, between regular transactions, a system transaction passes through the catalogs issuing recovery transactions at a lower priority for partitions that have not yet been recovered and that have not been requested by regular transactions.

Reading in the Log Records

The operations specified by log records must be applied in the same order that they were originally performed. A single linked list of log pages would force the recovery manager to read every log page before it could even begin to apply the log records. Recall that a *directory* of log pages is used here, and since the directory size is chosen to be equal to the average number of log pages for a partition, it should be possible in many cases to schedule the log page reads in the order that they were originally written. Thus, the log records from one page can be applied to the partition while the next page of log records are being read. When the number of log pages exceeds the directory size, it is possible to get to the first log page after $\left\lceil \frac{Number\ of\ Log\ Pages}{Number\ of\ Directory\ Entries} \right\rceil$ page reads.

Relation log records represent operations to update a field, insert a tuple, or delete a tuple in a partition. (More complicated issues involving changes to relation schema information are beyond the scope of this discussion.) Index log records represent *partition-specific* operations on index components. Recall from Chapter 3 that a single index update may involve several different actions to be applied to one or more index partitions. For example, a T Tree rotation operation can modify up to 5 different T nodes, thus generating 5 different log records. Thus, a given log record always affects exactly one partition.

5.3.7. Archive Logging

The disk copy of the database is basically the archive copy for the *primary* memory copy, but the disk copy also requires an archive copy (probably on tape or optical disk) in case of disk media failure. Protecting the log disks and database checkpoint disks comes under the well-known area of traditional archive recovery, for which many algorithms are known [Haerd 83], so it is not discussed here.

5.4. Performance Analysis of the Recovery Proposal

To get some idea of how this recovery mechanism will perform, we examine the performance of three main operations of the recovery component: logging, checkpointing, and recovery. First, the logging capacity of the recovery mechanism is calculated to determine the maximum rate at which it can process log records. Second, the frequency of checkpoint transactions is calculated for various rates, and the overhead imposed by checkpointing transactions is calculated as a percentage of the total number of transactions that are running. Finally, the speed of post-crash recovery is calculated for an individual partition, and the performance of partition-level recovery is demonstrated by comparing it with database-level recovery.

5.4.1. Logging, Checkpointing, and Recovery Parameters

Before the analysis, we introduce the parameters that are used throughout the performance analysis section.

We use the following convention for subscripted variables:

Letter	Represents
I	*Instructions*
N	*Number* (a count of something)
S	*Size*
R	*Rate*
T	*Time*
P	*Processing Power*
%	*percentage*

In determining the number of instructions required to perform the operations (Table 5.2), the instruction set of the VAX 11/750 was used as a general model. Complicated microcoded instructions such as the block move instruction are represented as multiple instructions. A "generic" instruction is assumed to execute in two microseconds and a memory reference executes in one microsecond. For each operation, the number of instructions was estimated (conservatively) based on a high-level pseudo-code version of its algorithm.

The disk numbers are based on an imaginary disk drive that is close in design to the Fujitsu Eagle disk drive. It uses two read/write heads per surface, so it has relatively low seek times. The transfer rate for a track of data is double the transfer rate for individual pages; partitions are written in whole tracks, whereas log pages are written individually (requiring separate disk operations). To achieve the maximum transfer rate possible for writing log pages, the disk sectors of the log disk are interleaved; logically adjacent sectors are physically one sector apart. After writing one page, a disk needs a small amount of "think time" to set up for the next page write — more time, we assume, than the time it takes to travel from the end of one sector to the beginning of the next physically adjacent sector. Therefore, by logically interleaving the sectors, the disk has the time of one full sector (1 millisecond in our case) to reset for the next page write. We also use different seek times for the checkpoint disks and the log disks. The seek time for a partition read is an average seek time, as a partition can be anywhere on the disk in relationship to the disk head during the recovery process. However, even though the log pages for a partition will be spread out over the log space, each page will be relatively close to its sibling, so the seek times between log pages should be somewhat less than the average seek time.

Name	Explanation	Units
I_{record_lookup}	Read one log record and calculate index to proper the log bin	Instructions/Record
I_{copy_start}	Startup cost of copying a string of bytes	Instructions/Copy
I_{copy_add}	Additional cost per byte of copying a string of bytes	Instructions/Byte
I_{write_Init}	Cost of initiating a disk write of a full log bin page	Instructions/Page Write
I_{alloc_page}	Cost of allocating a new log bin page and releasing the old one	Instructions/Page Write
I_{page_update}	Cost of updating the log bin page information	Instructions/Record
I_{page_check}	Cost of checking the existence of a log bin page	Instructions/Log Record
$I_{process\_LSN}$	Cost of maintaining the LSN count and checking for possible checkpoints	Instructions/Page Write
$I_{checkpoint}$	Cost of signaling the main CPU to start a checkpoint transaction	Instructions/Checkpoint
I_{record_sort}	(Calculated) Total cost of the record sorting process	Instructions/Record
I_{page_write}	(Calculated) Total cost of writing a page from the SLT to the log disk	Instructions/Page
S_{log_record}	Average Size of a log record	Bytes/Record
S_{page}	Size of a Log Page	Bytes/Page
$S_{partition}$	Size of a Partition	Bytes/Partition
S_{log_window}	Size of the Log Window	Pages
$N_{updates}$	The number of log records that a partition can accumulate before a checkpoint is triggered	Log Records/Partition
N_{log_disks}	The number of dedicated log disks	None
N_{LSN_Dir}	The length of the log page directory for a partition bin	None
T_{part_read}	Time to read one partition (a track)	Seconds
T_{log_read}	Time to read one log page	Seconds
T_{part_seek}	Time required to seek to a partition	Seconds
T_{log_seek}	Average time to seek to a log disk page during recovery	Seconds
R_{bytes_logged}	(Calculated) Rate of the Logging Component	Bytes/Second
R_{disk}	Transfer Rate of a single disk drive	Bytes/Second
$R_{checkpoint}$	(Calculated) The Checkpoint Frequency	Checkpoints/Second
$P_{recovery}$	MIPS power of the recovery CPU	Instructions/Second
P_{main}	MIPS power of the main CPU	Instructions/Second

Table 5.1 — Parameter Explanations

I_{record_lookup}	10 Instructions
I_{copy_start}	2 Instructions / Copy
I_{copy_add}	0.125 Instructions / Byte
I_{write_init}	40 Instructions
I_{page_alloc}	40 Instructions
I_{page_update}	6 Instructions
I_{page_check}	8 Instructions
$I_{process\_LSN}$	40 Instructions
I_{check_pt}	40 Instructions
S_{log_record}	20 Bytes
S_{log_page}	2000 Bytes
S_{disk_sector}	2000 Bytes
$S_{partition}$	32,000 Bytes
S_{disk_track}	32,000 Bytes
$N_{updates}$	1000
N_{lsn_dir}	10
$T_{ave\_disk\_seek}$	0.018 Seconds
$T_{ave\_log\_disk\_seek}$	0.012 Seconds
$T_{partition\_transfer}$	0.015 Seconds
$T_{page\_transfer}$	0.001 Seconds
$T_{latency}$	0.0075 Seconds
R_{disk}	2,130,000 Bytes / Second (Track)
R_{disk}	1,065,000 Bytes / Second (Page)
$P_{recovery}$	500,000 Instructions / Second
P_{main}	500,000 Instructions / Second

Table 5.2 — Recovery Parameters

In chosing sizes for log pages and partitions we were presented with a list of tradeoffs. For example, log page size represents a subtle tradeoff between the space required to hold the pages in the Stable Log Buffer and the frequency of page writes and page allocations. Partition size affects the number of entries in the Stable Log Tail (larger partitions mean fewer partition entries), the cost and efficiency of checkpoints (larger partitions might cause a larger percentage of non-updated data to be written during a checkpoint operation), and the overhead of managing partitions (small partitions mean maintaining more entries per relation). The sizes for log pages and partitions were chosen from the middle of a range of possible values; given the specifications and database reference patterns of a particular database system, it might be possible to pick better page size and partition size values.

5.4.2. Logging Capacity

The logging rate of a recovery mechanism must be greater than the rate at which the main CPU can generate log records, or else the recovery mechanism will be the performance bottleneck of the system. We estimate the logging capacity of the proposed design using a simple analytical model.

During normal processing the recovery CPU spends most of its time moving log records from the Stable Log Buffer into partition bins in the Stable Log Tail. It spends a smaller portion of its time initiating disk write requests for full pages of log records and an even smaller portion of its time notifying the main CPU of partitions that must be checkpointed. The process of reading records from the Stable Log Buffer and moving them into the Stable Log Tail is composed of many steps. The recovery CPU locates a log record, reads its bin index, and uses that bin index to compute the address of the log page that will hold the log record (I_{record_lookup} represents this cost); the log page is then checked for existence and available space (I_{page_check} represents this cost); and finally the log record is copied into the log page. A byte copy has a combined cost of an initialization phase and a copy phase. The initialization phase involves loading registers with the address of the log record (source), the address of the correct location in the log page (destination), and the length of the log record. Once the registers are loaded, the byte copy can proceed at memory speeds. Hence, the log record copy cost is shown by $I_{copy\_start} + (I_{copy\_add} * S_{log\_record})$. Last, the log page space count and the partition update count must be updated (and I_{page_update} represents this cost). For every page's worth of log records $\frac{S_{log\_record}}{S_{page}}$, a new memory page must be allocated and the old page, after it is written to disk, is released (using I_{page_alloc} instructions). The cost of the first phase of recovery processing, then, is the "sorting" process moving a single log record from the Stable Log Buffer to its correct location in the Stable Log Tail:

$$I_{record\_sort} = I_{record\_lookup} + I_{page\_check} + I_{copy\_start} + (I_{copy\_add} * S_{log\_record}) + I_{page\_update} + \frac{I_{page\_alloc} * S_{log\_record}}{S_{page}}$$

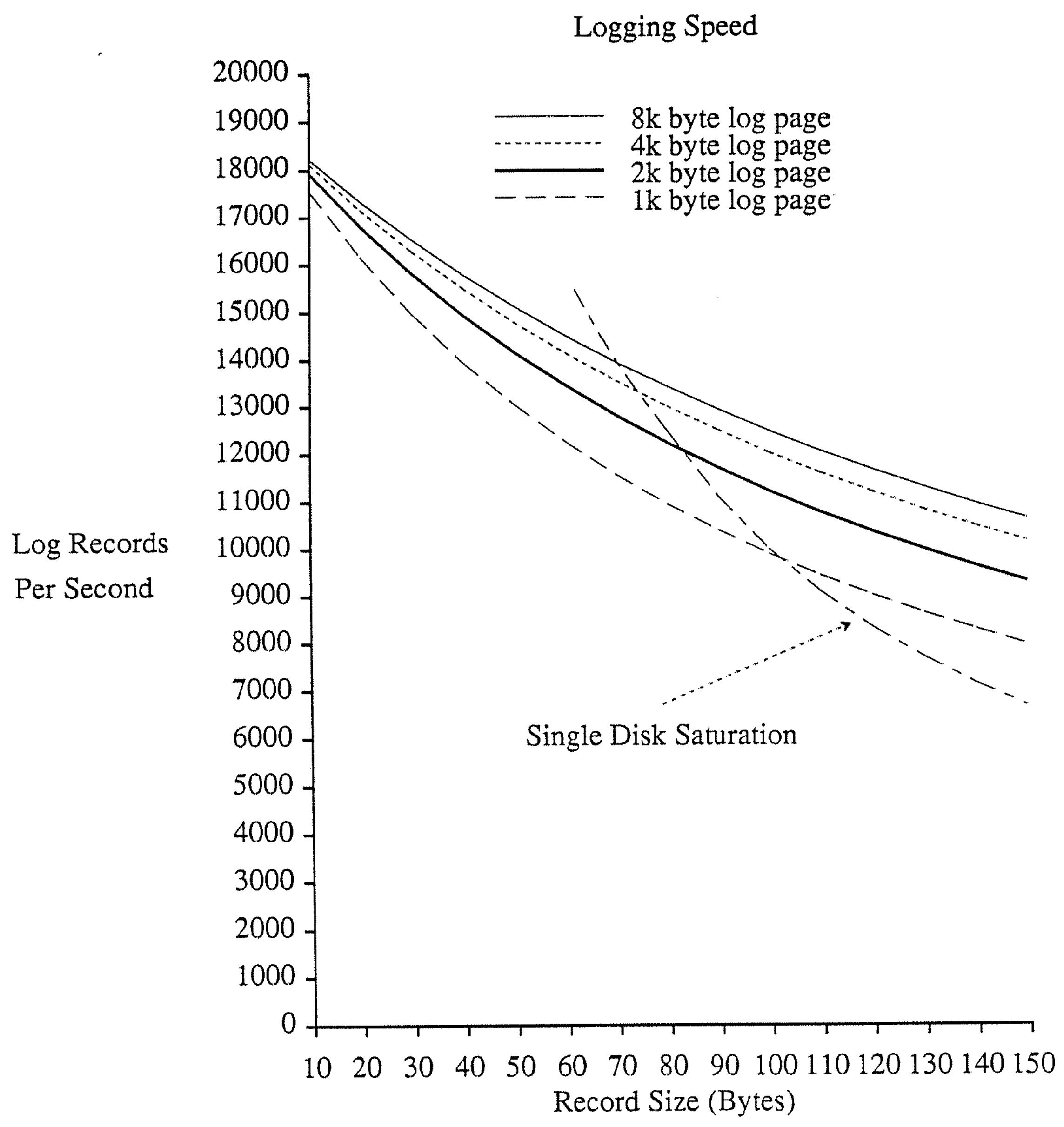
The second phase of the logging process entails writing the partition bin log pages to disk as they become full. Every time a log page is written, several operations are done. A disk-write request is entered into a queue of disk requests (I_{write_init}) and the current Log Sequence Number is incremented, advancing the log window boundary ($I_{process\_LSN}$). Whenever the log window boundary changes, the *First LSN* list of log pages is checked for a partition whose first log page extends beyond the log window boundary (there is never more than one). Also, the update count for the partition whose log page is being written is checked for exceeding the update count threshold. If either the update count exceeds the threshold value, or some partition's first log page extends beyond the log window boundary, the recovery manager enters a flag in the Stable Log Buffer that signals the main CPU to initiate a transaction to checkpoint the appropriate partition ($I_{checkpoint}$). A checkpoint operation is initiated once every time the number of pages needed to hold $N_{updates}$ log records have been written. After a partition has been checkpointed, the checkpoint transaction signals the recovery manager that the operation is complete; the log page in the Stable Log Tail is then flushed, and the Stable Log Tail space is reclaimed (the page de-allocation was accounted for in the record sort equation). Thus:

$$I_{page\_write} = I_{write\_init} + I_{process\_LSN} + \frac{I_{check\_pt}}{N_{update} * \frac{S_{record}}{S_{page}}}$$

The maximum logging rate of the recovery mechanism is expressed in log bytes per second. The recovery processor executes a number of instructions to move a log record from the Stable Log Buffer to a partition bin in the Stable Log Tail (I_{record_sort}), and it executes a number of instructions to write a partition bin log page to disk (I_{page_write}). If we combine those instruction costs in terms of instructions per byte and divide that into the processing power of the system (instructions per second) then we get the speed of the logging component in bytes per second:

$$R_{bytes\_logged} = \frac{P_{recovery}}{\frac{I_{Record\_Sort}}{S_{Record}} + \frac{I_{Page\_Write}}{S_{Page}}}$$

The predominant cost of the logging operation is the per record sorting cost. Although the instruction count of a record sort operation is about one third the instruction count of a page write, it occurs much more frequently (by a factor of a hundred or so for 2 kilobyte pages and 20 byte log records). The log record sorting cost increases as the record size increases, as it affects both the log record copy cost and the frequency of allocating new pages. The page size also affects this cost, as larger pages require fewer disk writes and fewer page allocation operations. Graph 5.1 shows the logging speed in log records per second (*i.e.* $\frac{R_{bytes\_logged}}{S_{log\_record}}$) for various log record and disk page sizes.



Graph 5.1 — Logging Speed

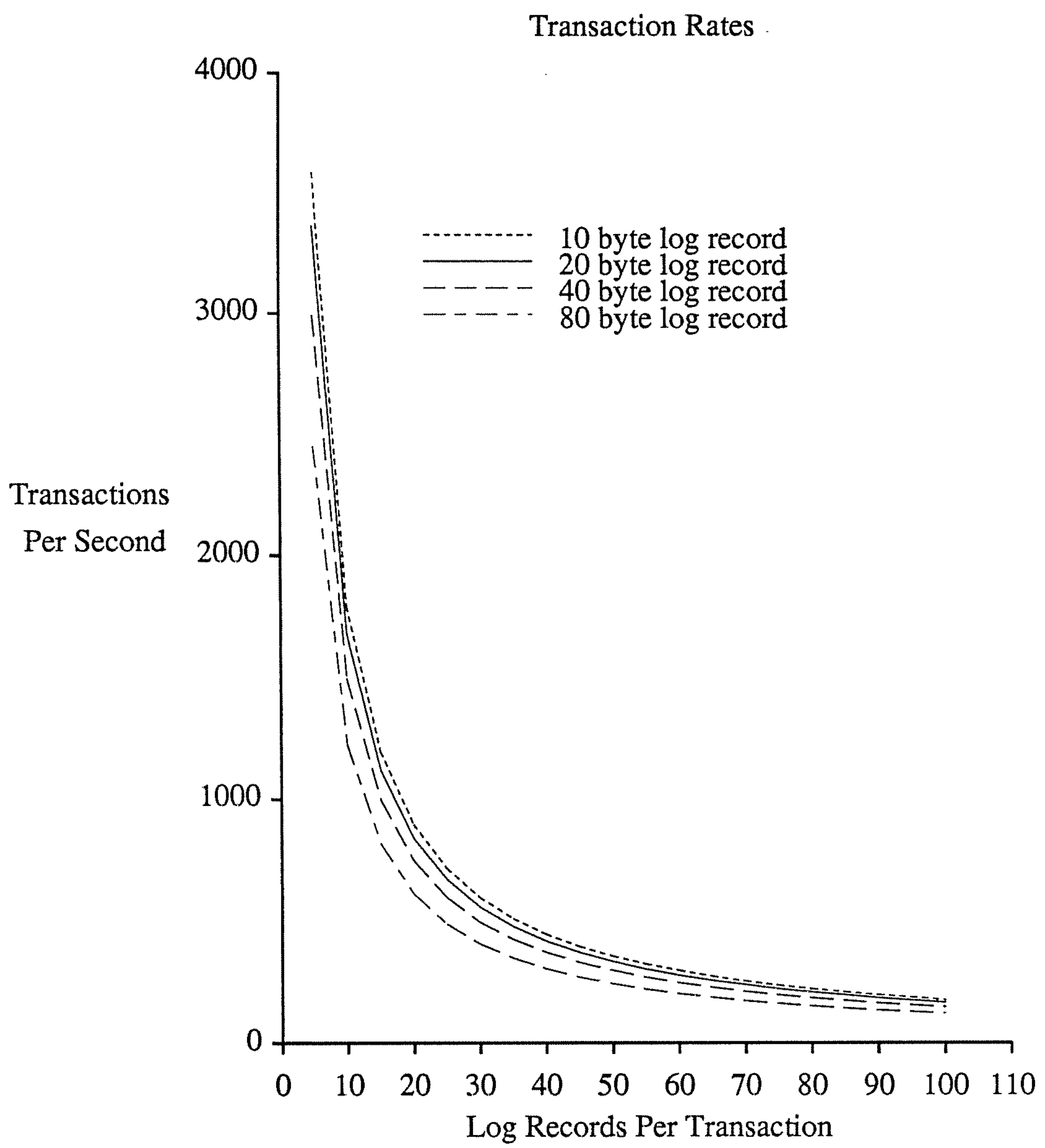
For small record sizes the logging rate is CPU-bound, but larger log records cause the total number of bytes being produced to reach the transfer rate of the log disk, hence the logging rate becomes I/O-bound for larger records. The dotted line on the graph shows the values of record logging rate and log record size at which a single disk operating at about a 1 megabyte transfer rate would become saturated. When the log disk transfer rate is the bottleneck, multiple disks can be used to make the logging rate once again CPU-bound.

The number of log records generated by a transaction is application-dependent. It can range from a few log records over thousands of instructions (for computation-intensive transactions) to a few records over hundreds of instructions (for Gray's debit/credit transactions [Gray 85]) to one log record over only tens of instructions (for update intensive transactions). Graph 5.2 shows the various maximum transaction rates that can be supported by the logging mechanism as the number of log records generated by a transaction is varied.

Typical log records should be small, as common operations such as index operations, numerical field updates, and delete operations all generate log records that are 8 to 20 bytes in size. Larger log records are generated by other operations, updates to long fields or insertions of whole tuples, for example, but they are less likely to be used frequently. Gray's notion of a typical debit/credit transaction is one that writes approximately four log records. Using these figures, our logging mechanism can accommodate transaction rates of up to 4,000 transactions per second! Judging from these results, it appears safe to conclude that under most realistic conditions, our logging mechanism will not be the performance bottleneck of the system.

5.4.3. Checkpointing Overhead

The recovery processor does little work for checkpointing. When it notices that an object should be checkpointed, it simply signals the main CPU that the task should be done. The real overhead lies with the main CPU, as it performs the real work of a checkpoint operation. It is responsible for locking the object, copying it to a side



Graph 5.2 — Transaction Rates

buffer and releasing the lock on the primary copy of the partition, locating a disk track for the partition, logging the updates to the disk map and catalog information, scheduling the disk write, and finally, committing the checkpoint transaction.

The *frequency* of checkpoint transactions is of interest because it shows the percentage of all transactions that are devoted to regular transaction processing *versus* the percentage that are devoted to checkpointing partitions. The frequency of checkpoint transactions is determined by the logging rate, the update count for each partition, the number of active partitions, the distribution of updates over the active partitions, and the size of the log window. (Recall that the log window is the active portion of the reusable log.) For a given log window size, an active partition may reside in the log window long enough to accumulate enough updates to trigger a checkpoint, or it may reach the end of the log window and be checkpointed so that its log space can be reclaimed. The number of checkpoints triggered by update count depends on the number of active partitions for a particular log window and the distribution of log records over those partitions. Given an infinite log window, checkpoints of all active partitions would eventually be triggered by update count, in which case the checkpoint rate would be:

$$R_{checkpoint} = \frac{R_{records\_logged}}{N_{update}}$$

This is the best possible scenario, as the cost of each checkpoint operation is amortized over N_{update} update operations.

Since log space is finite, there will be some active partitions that do not accumulate N_{update} updates, so they will be checkpointed because of age instead. This leads to a higher number of checkpoint operations, as the cost of checkpoints triggered by age are amortized over fewer than N_{update} update operations. Thus, the worst case occurs when each active partition accumulates only one page of log records before it is checkpointed. In this case:

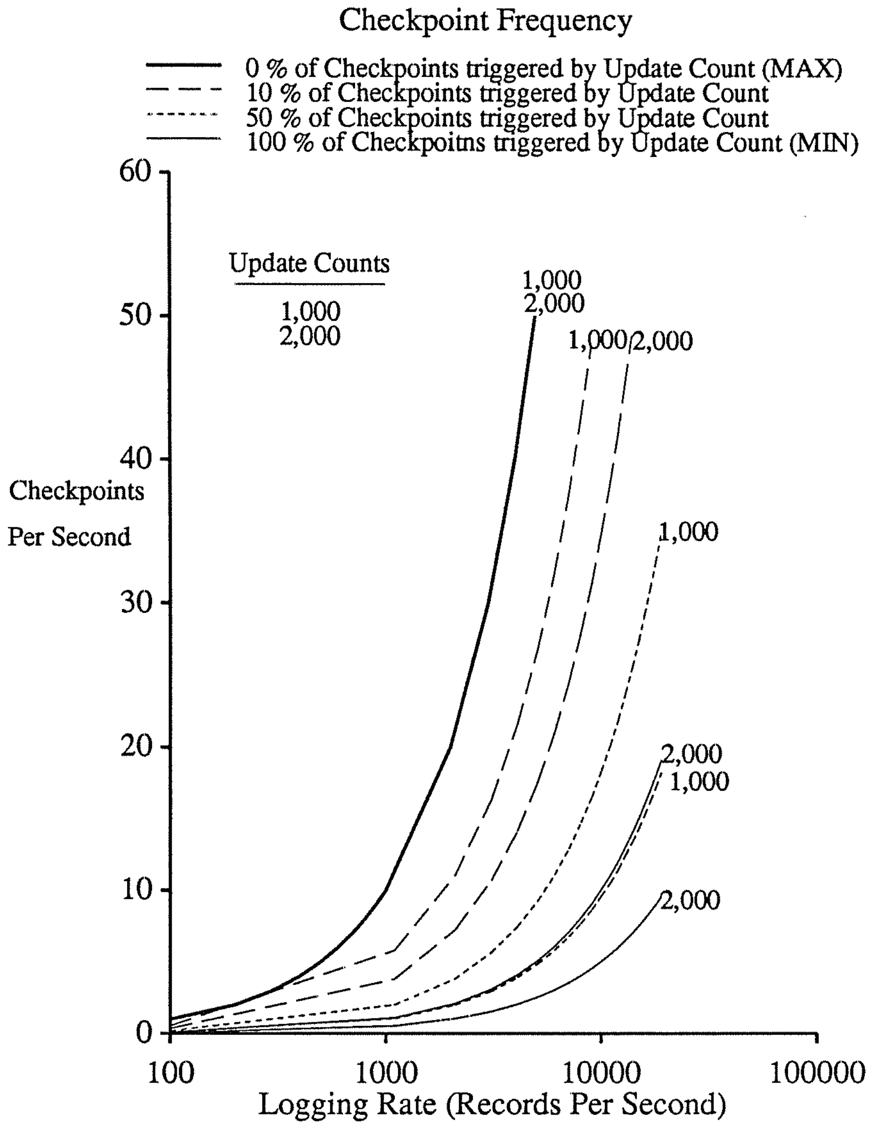
$$R_{checkpoint} = R_{records\_logged} * \frac{S_{log\_record}}{S_{page}}$$

It is not likely that the best or worst case will ever occur; instead, there will be some percentage of each type of checkpoint operation. For a given number of active partitions, a large log window simply provides a better opportunity for a partition to accumulate N_{update} log records than a small window. Thus, for performance reasons, there is a minimum log window size for a given number of active partitions — there should be at least enough pages in the low window to hold N_{update} log records for every active partition. The only limitations on maximum size are related to how much disk space is affordable.

To calculate an average case checkpoint frequency, it is necessary to use a mix of checkpoints triggered by age and triggered by update count. For a given percentage of partitions that are checkpointed because of age, there are a range of possible checkpoint frequencies, as those partitions could have anywhere from a single page of updates to almost N_{update} updates. For comparison purposes we will always assume the worst case — a partition checkpointed because of age has accumulated only one page of log records. Thus, the equation to determine the checkpoint frequency rate for given percentages of partitions being checkpointed because of update count and age is:

$$R_{checkpoint} = \frac{R_{records\_logged}}{\%_{update\_count} * N_{update} + \%_{age} * \frac{S_{page}}{S_{log\_record}}}$$

Rather than try to choose actual numbers for the log window size, the number of active partitions, and the distribution of log records, we simply examine some different mixes of checkpoint trigger percentages. Graph 5.3 shows the checkpoint frequencies for various update counts and trigger percentages as the logging rate is varied. The log window size and the number of active partitions determine the number of checkpoints,



Graph 5.3 — Possible Checkpoint Frequencies

but the logging rate determines how frequently they will happen; the logging rate determines how fast pages of log records go into the log window, and thus, how fast they reach the end of the log window, possibly triggering checkpoint operations. For an update count of 1,000, if the log window size is large enough to allow 50 percent of the active partitions to accumulate N_{update} log records, then the checkpoint frequency will be fairly low. Assuming an average transaction writes about 10 log records, this would indicate that checkpoint transactions generated at this frequency would compose only one percent of the total transaction load. An average number of log records less than 10 simply decreases the percentage of checkpoint transactions.

A larger update count causes fewer checkpoint operations to occur for a given trigger percentage, but it also increases the suggested minimum size of the log window. A similar effect is seen when the log page size is doubled or the record size is halved, as either one increases the number of log records that an active partition must accumulate before it can have a checkpoint triggered by age. (Recall that an active partition's log information is not even entered into the disk log space until it has accumulated a full page of log records.)

If a single disk is used for holding checkpoint images, then the transfer rate of the single disk puts an upper bound on the possible frequency of checkpoint transactions. Assuming a 15 millisecond partition write time and a 5 millisecond average seek time between writes to find a vacant checkpoint image location, a single disk drive would place an upper bound of 50 checkpoint transactions per second. However, given a realistic environment, it does not appear that this bound will have any effect whatsoever. Notice that checkpoint transactions provide an interesting form of load control, as an increase in the number of checkpoint transactions causes more disk and CPU resources to be used for checkpoint transactions; thus, less work gets done by normal transactions, and less log information is generated.

5.4.4. Post-Crash Partition Recovery Speed

The purpose of the partition-level recovery algorithm is to allow transactions to begin processing as soon as their data is restored. Transactions issue requests for certain relations to be recovered, and they are able to proceed when their requested relations are available; they do not wait for the *entire* database to be restored. An upper bound on the time to recover a relation is the sum of its partition recovery times. A partition's recovery time is determined by the time to read its checkpoint image from the checkpoint disk, the time to read all of its log pages, and the time to apply those log pages to its checkpoint image. A partition's checkpoint image and its log pages may be read in parallel, since they are on different disks. Also, provided that the log page directory was chosen to be large enough to hold entries for all of the log pages for the partition, the log pages can be read in the order that they were originally written, thus allowing the log records from one log page to be applied to the partition in parallel with the reading of other log pages. (Assuming, of course, that the time to apply a page of log records to a partition is less than the time to read a log page.) In this case, the time to recover a partition is the time to read its log pages plus the time to apply the updates from the last log page. (The last page of updates cannot be done in parallel with any disk page reads for this partition, as it is the last page.) Under these assumptions:

$$T_{partition\_recovery} = MAX(T_{read\_checkpoint\_copy}, (T_{read\_log\_page} * N_{log\_page})) + T_{apply\_updates}$$

The time to read a partition checkpoint copy is determined by the seek time of the checkpoint disk (T_{part_seek}), plus the average disk latency ($T_{latency}$), plus the read transfer time of a partition (T_{part_read}). Thus:

$$T_{read\_checkpoint\_copy} = T_{part\_seek} + T_{latency} + T_{part\_read}$$

Similarly, the time to read a log page is:

$$T_{read\_log\_page} = T_{log\_seek} + T_{latency} + T_{log\_read}$$

The time required to apply REDO operations to a partition depends on the types of

REDO operations; in the interest of generality (and simplicity) we pick an average distribution of insert, update, and delete operations. It seems likely that updates will be more common than inserts and deletes, so we pick a distribution of 60 percent update, 20 percent delete, and 20 percent insert operations. For the instruction costs of these operations, we assume that insert operations are the most expensive operations, since they require allocation of space and copying of data; deletes are less expensive, since they require only deallocation of space; and updates are least expensive, since they require only a modification of a value. Thus, estimating the work that each operation would have to do by either modifying a relation partition or an index partition (from Chapter 3), we assign instruction costs to these operations (Table 5.3).

C_{insert}	60
C_{delete}	40
C_{update}	20

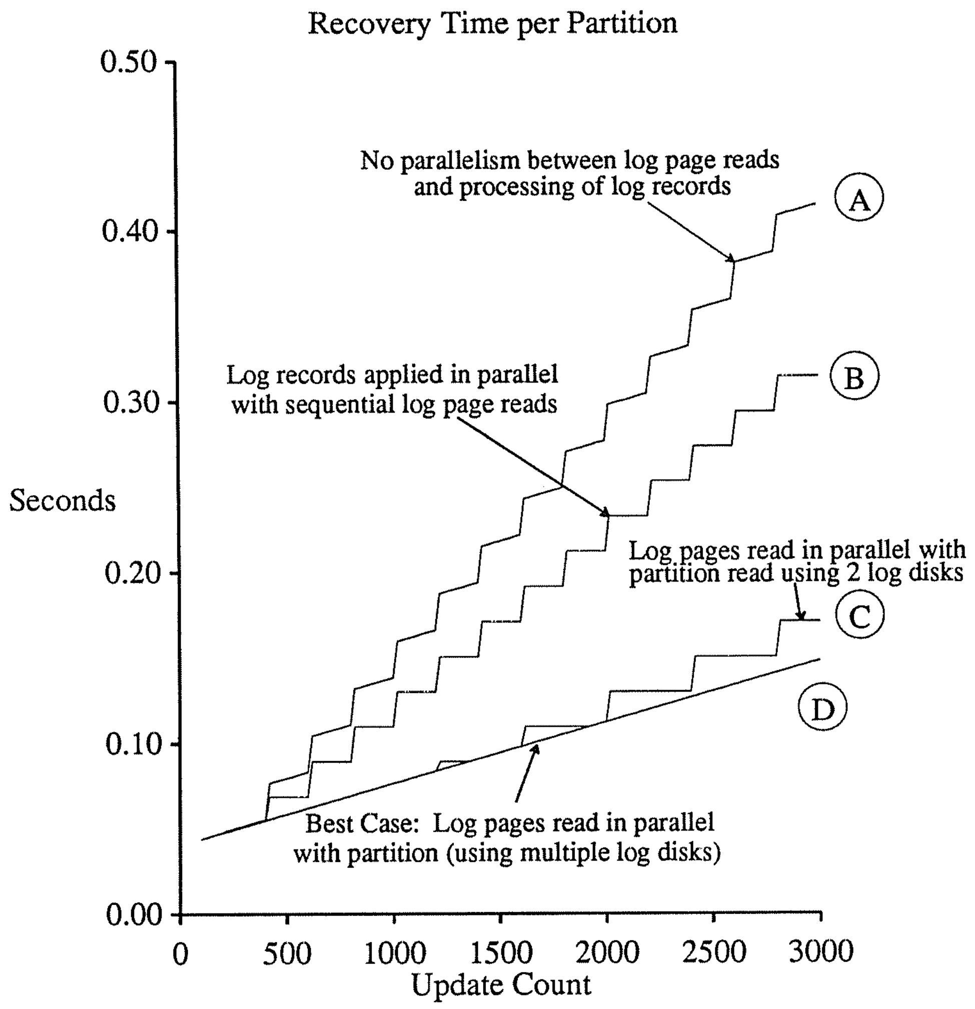
Table 5.3 — Redo Operation Instruction Costs

The time required to apply partition updates is determined by the number of REDO operations, the cost of an average REDO operation, and the speed of the processor.

$$T_{apply\_updates} = \frac{N_{update} * (0.6C_{update} + 0.2C_{delete} + 0.2C_{insert})}{P_{recovery}}$$

As mentioned earlier, the recovery process can have various amounts of parallelism; log records can be read in parallel with reading the partition checkpoint image, the REDO operations specified by the log records can be applied to the checkpoint image while log records on later log pages are being read, and finally, log pages can even be read in parallel if more than one log disk is present. The minimum recovery time for a partition, given that it has log information, is the time required to read a checkpoint image plus the time required to apply the log record updates. (They cannot be done in parallel.)

Graph 5.4 shows the possible recovery times for a partition, using various amounts of parallelism. In all cases the checkpoint image is assumed to be read in parallel with the reading of the log information. Also, on the average, active partitions will have about half of the maximum possible amount of log information (represented



Graph 5.4 — Partition Recovery Time

by N_{update}). Hence, the update count values used in generating the graph were half of what is shown on the X-axis. (We use the whole values on the X-axis to give a clearer indication of the recovery times for a partition for various update counts.) Line *A* shows the time needed to recover a partition when the log records are not processed in parallel with the reading of subsequent log pages. Line *B* shows the improvement that one can get by applying log records in parallel with reading log pages, using the partition bin log page directory. Line *C* shows the same method used for line *B* except that multiple log disks were assumed. In this situation, two log disks were sufficient for most update counts, as they were able to read the log information about as fast as the checkpoint image was read. Line *D* represents the best case; the log information is read in parallel with the partition checkpoint image. In recovering multiple partitions, it may be possible to take advantage of further parallelism by overlapping the reading of the checkpoint images with log record processing, but we do not consider that case here.

5.4.4.1. Comparison with Complete Reloading

The main alternative to partition-level recovery is database-level recovery. (An interesting point is that database-level recovery is a special case of partition-level recovery — using one very large partition.) Partition-level recovery is cost effective only if it provides benefits over database-level recovery. We compare the two methods in our assumed environment. First, the parameters of the analysis must be stated (Table 5.4). Next we state our assumptions about the database-level recovery model:

- The entire database can be read into memory at disk transfer rates.
- The amount of log information is determined by the logging rate and the amount of time required for a checkpoint operation, which in turn is determined by the

Number of Log Disks	2
Number of Checkpoint Disks	1
Update Count	1,000
Page Size	2,000 bytes
Log Record Size	20 bytes
Partition Size	32,000 bytes
Logging Rate	1,000 records per second

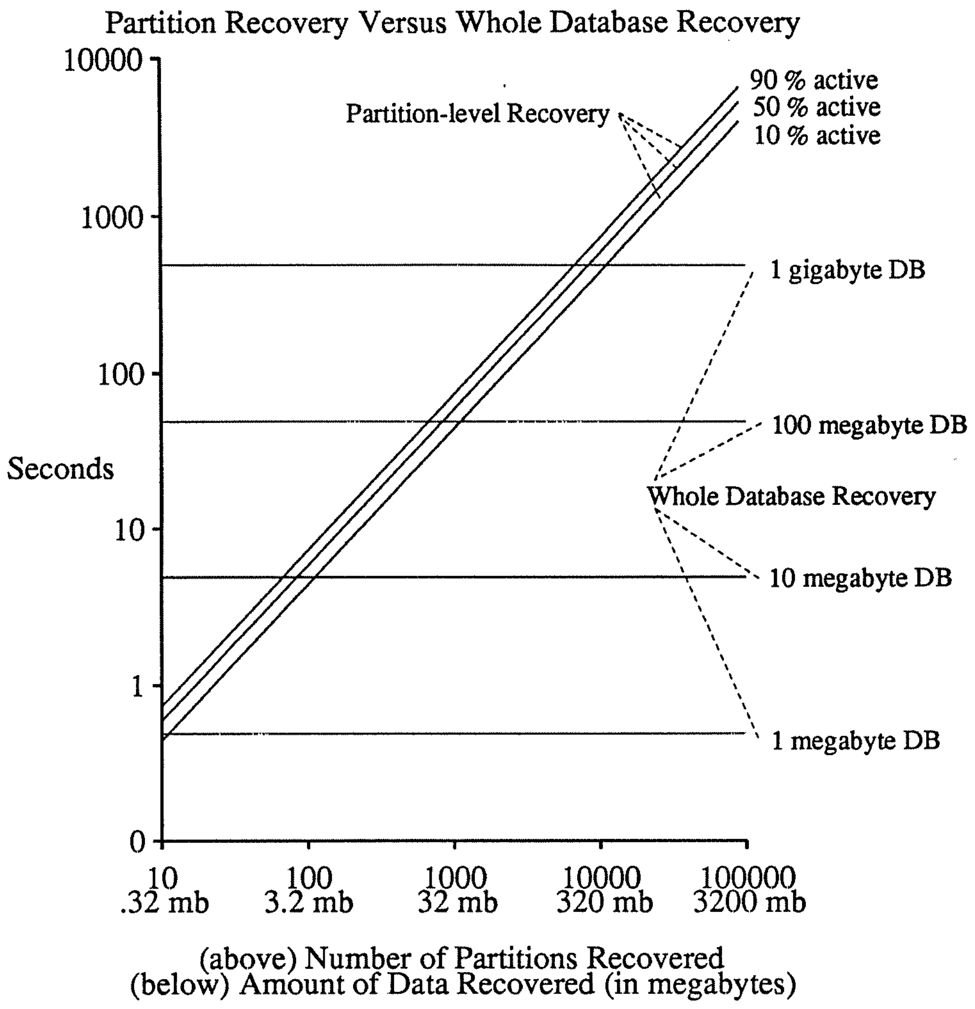
Table 5.4 — Parameters for Complete Reloading

size of the database. A checkpoint operation writes the database to disk at the disk transfer rate. We assume, for generality, that crashes occur midway through the checkpoint operation, so the amount of log information that must be processed is half of the maximum amount.

- The log comprises entry-level log records, as opposed to page-level log records. (Entry-level log records can be one to two orders of magnitude smaller than page-level log records.)
- The cost of applying log records to the checkpoint image of the database is the same as the cost used in the previous section.

Graph 5.5 shows the recovery costs for both partition-level and database-level recovery. The database-level recovery times are constant for a given database size and logging rate, thus they are represented as horizontal lines. Partition-level recovery times vary with the number of partitions that must be recovered. For this comparison, both types of recovery are using all of the system resources — there are no other parallel activities.

If the entire database *must* be reloaded before transaction processing can begin, then database-level recovery is best, as it can recover the entire database roughly three



Graph 5.5 — Comparison with Database-level Recovery

times faster than partition-level recovery. Database-level recovery takes about 500 seconds to recover a 100 megabyte database, whereas partition-level recovery takes about 500 seconds to recover 32 megabytes of data. However, a typical transaction is likely to need only one or two relations before it can begin processing. Using partition-level recovery, such transactions could have their required information restored quickly — in the time it takes to restore one or two relations, as opposed to the time it takes to restore the entire database.

5.4.5. Summary

Previous work in recovery, both for traditional disk-based systems and for memory-resident systems, has addressed issues of logging, checkpointing and recovering a database. Many designs have been proposed, but none of them have completely satisfied the needs of a high-performance memory-resident database system. Such a system needs a fast efficient logging mechanism that can assimilate log records as fast or faster then they can be produced. It needs efficient checkpoint operations that can amortize the cost of a checkpoint operation over many updates to the database. Finally, the recovery mechanism should be geared toward allowing transactions to run as soon as possible after a crash.

A design has been presented that meets these three criteria. With the use of stable memory and a recovery processor, the logging mechanism will not be the performance bottleneck. Using a PDP VAX 11/750 as a model, it was estimated that the recovery CPU, with proper disk support, can assimilate log records much faster than the main CPU could produce them. Checkpoint operations for partitions are triggered when the partitions have received a significant number of updates, and thus the cost of a checkpoint operation is amortized over many updates. The worst case, where checkpoint operations are generated over only a log page's worth of log records, is assumed to be unlikely, especially if the log window size is made large enough. Recovery of data in our design is oriented toward transaction response time. After a crash, relations that are requested by transactions are recovered first so that the transactions can begin processing. Relations that are not requested are recovered on a low priority basis.

Partition-level recovery will also be beneficial in the event of a partial memory failure. When part of the main memory fails, the information in that part of memory is lost. Existing recovery methods would either have to recover the entire database, or else filter out the required information from the checkpoint copy and the log in order to recover the lost portion of the database. In the case of a partial memory loss with this proposal, the lost partitions can be restored independently once the bad memory has been mapped out of the logical space.

Chapter 6

Concurrency Control

Although there are many algorithms for concurrency control [Berns 81], they can be divided into three main categories: serial validation, timestamping, and locking. Several performance studies have addressed the relative merits of these methods [Carey 84], [Agraw 85a], [Agraw 85b]. These studies make the assumption that the concurrency control costs are roughly the same for each algorithm (an assumption supported by an earlier paper [Carey 83]), and their experiments examine the performance of the various methods under a variety of assumptions regarding workload characteristics, database characteristics, and available system resources (e.g. CPUs and disks). Their performance results indicate that locking methods provide the best performance for most realistic environments. In a memory-resident database system, the relative costs of serial validation, timestamping and locking would remain about the same, as most modern database systems already keep their concurrency control mechanisms in main memory [Gray 82]. Judging from these results, it seems likely that locking will also be the preferred method for memory-resident database systems.

A locking-based concurrency control method sets its locks on some unit of the database; that portion is referred to as a *granule.* The optimal granule size for a database system is somewhat application-dependent, as transactions that access large amounts of the database can lock their data most efficiently by using large granularity locks, and transactions that access small amounts of the database can maintain a high degree of sharing by using small granularity locks. In their paper on locking granularity, Ries and Stonebraker conclude that, when a mix of such transactions exists, a two level lock hierarchy, using relation locks and tuple locks, seems to be best [Ries 79]. Such a hierarchical locking mechanism allows transactions accessing large amounts of data to set relation-level locks and transactions accessing small amounts of data to set tuple-level locks.

Hierarchical locking is considered acceptable in a disk-based system because the locking cost is much smaller than the cost of a disk-page reference. However, the cost of a database reference in a memory-resident system is the cost of an index lookup or, in some cases, just the cost of a simple memory reference. Since the cost of setting a lock involves a table lookup and the allocation of a lock control structure, the cost of a database access and the cost of setting a lock in a memory-resident database system are likely to be quite similar. If two locks need to be set for each database reference due to hierarchical locking, the locking cost might well exceed the database reference cost. In this chapter a new locking-based concurrency control algorithm is presented that provides the benefits of a hierarchical approach without the cost of setting multiple locks for each database reference.

6.1. Motivation

A hierarchical locking approach is desirable, as it provides a way for a transaction to lock the appropriate amount of data. However, its locking mechanisms must be changed to reduce their locking cost. One way to minimize locking cost would be to reduce the lookup time required to locate a database entity's lock control block. One possible approach would be to store pointers inside database entities that point to their lock control blocks. This would save time, as it would cost only a memory indirection to access the lock information, but storing these pointers for *every* database entity would use a large amount of memory that could otherwise be used to hold data.

(Recall that we view the memory as being large, but not free.) For most small database entities this extra space overhead would probably not be cost-effective, as they would not be locked most of the time, but it may be worthwhile for often-used data such as system catalog entries.

In a memory-resident database system, disk accesses do not interrupt normal transaction processing. This suggests that a CPU time slice can be lengthed so that more time is spent on transaction processing and less time is spent on process switching. Without disk interactions, most transactions will complete quickly and hold locks for only a short time, so sharing of data might seem to be less important. Indeed, given a uniprocessor running only short transactions, *complete serialization* would be possible, thus removing the need for concurrency control entirely [Salem 86]. However, most real database systems occasionally run long transactions, and some employ multiple processors, so some form of concurrency control is still needed.

In a multiprocessor system, a short transaction still may be able to lock an entire relation without significantly restricting the progress of any other transactions. If so, a transaction *could* begin by setting relation-level locks and then, if other transactions block on its relation locks to the point where concurrency is significantly reduced, its relation-level locks could be *de-escalated* into tuple-level locks. Thus, overall locking cost could be reduced by only using the more expensive tuple-level locks when they are needed. Some relations, like database catalogs that are accessed frequently, may stay at the tuple-level lock status.

6.2. Overview

The proposed locking algorithm uses two locking granule sizes, setting relation-level locks and tuple-level locks. Locking at the relation level is much cheaper than locking at the tuple level, so it is the preferred method when a fine granularity of sharing is not needed. When several transactions desire access to a relation that is locked with a relation-level lock, the relation lock is de-escalated into a set of tuple locks and then the higher cost for tuple-level locking is paid, but the level of sharing is increased. To allow the possibility of relation lock de-escalation, relation-specific tuple-level read sets, write sets, and read predicate lists of a transaction are kept in a control block so that they may be turned into tuple locks if the need to do so arises. When such fine granularity locks are no longer needed, tuple-level locks are escalated back into relation-level locks. Certain operations that require the use of an entire relation will be able to *force* lock escalation to the relation level and then disable lock de-escalation until they have completed.

When locking at the tuple level, the problem of locking phantom tuples exists [Eswar 76]. That is, problems can arise due to tuples that cannot be locked out with physical locks because they do not exist until shortly after the locks are set. Solutions for preventing phantoms range from locking the entire relation to locking a range of values that cover the tuples being read. Locking a range of values can be done with a physical lock, as is done in R* *via* locks on index pages [Linds 85], or it can be done with predicate locks, as in the *Precision Locks* algorithm proposed by Jordon *et. al.* in [Jorda 81]. Since appropriate indices are not always present, we propose a predicate lock scheme to handle the phantom problem. Tuple read and write locks are used for regular write/read and write/write synchronization, and a variant of Precision Locks is used for read/write synchronization.

6.3. Avoiding the Convoy Problem

A latch is a low level primitive that provides a cheap serialization mechanism with shared and exclusive lock modes, but no deadlock detection [Gray 78]. In disk-based database systems, latches are used mainly by resource managers to gain temporary exclusive access to shared data structures. In a memory-resident database system the entire database is a shared data structure, so latches are needed to serialize updates to everything. For example, a latch is needed to prevent readers from referencing a partition while it is being updated because the update operation might reorganize the partition's string space, and cause the tuples in the partition to be in a temporarily inconsistent state.

Latches should not be held by inactive processes, as this can lead to the convoy problem [Blasg 79]. The convoy problem is created when a preemptive scheduler preempts a process that is holding a critical resource in a critical section (*i.e.* a *latch*). All processes needing this critical resource block on the latch and wait for it to be released. When the holder of the latch is dispatched, it usually rerequests the latch shortly thereafter (after 1,000 instructions or so) and, since scheduling is fair, the process must wait until its turn to get the latch again. Once created, convoys tend to persist for a very long time [Gray 78]. To prevent the convoy problem, a process holding a latch can also implicitly hold a "latch" on its processor, thus avoiding the possibility of being descheduled while holding an important shared resource in a critical section. In many systems, having a transaction disable a process switch would require an expensive kernel call, but it can be done inexpensively by having the database system share a data structure with the kernel. Setting a flag in this data structure will allow a transaction to run uninterrupted while it executes the operation requiring the latch. This is similar to the "open addressing" design of the Xerox open operating system for a single-user machine [Lamps 79a].

6.4. Lock Structure

Relations keep a small amount of static lock information. The relation *static lock control block* resides in a well-known location in the first partition of a relation segment, so it is easily found given any tuple address in that relation. A relation's static lock control information block contains its current locking status (tuple-level or relation-level) and a pointer to a list of transaction lock control blocks (allocated dynamically) that are kept by each transaction holding a lock on this relation (Figure 6.1). Each transaction lock control block holds a lock mode (share or exclusive), a list of tuple pointers corresponding to its read set, a list of tuple pointers corresponding to its write set, a list of predicates corresponding to its read set, its transaction id, and list

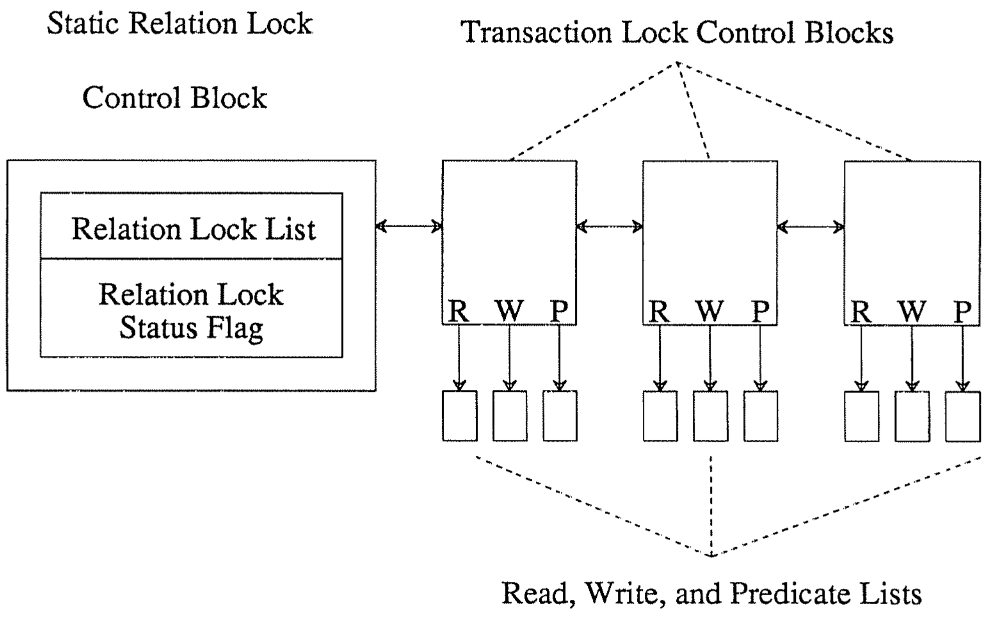

Static Relation Lock Control Block
Transaction Lock Control Blocks
Relation Lock List
Relation Lock Status Flag
R W P
R W P
R W P
Read, Write, and Predicate Lists

Figure 6.1 — Relation Lock Control Block Structure

of other transactions blocking on any of its lock(s) (Figure 6.2). Transactions *always* create lock control blocks for each relation they reference because, regardless of a relation's lock status value, transactions always maintain a read set, a write set, and a list of predicates specific to that relation.

The locking status flag for a relation is kept in the relation's static lock control block. As a transaction runs, it checks this locking status flag before each tuple reference. A transaction sets the appropriate lock according to the lock status flag value. Since the locking status for a relation may change during the course of a transaction, it must be checked before each potential locking operation. Some types of transactions, however, will force the use of a relation lock; they would not need to test the lock status flag. Examples of such transactions are those used to create indices, perform relation scans, and checkpoint partitions.

Relation Locks

A single lock at the relation level locks both a relation and all of its associated indices. Partition-level and index-level latches are not needed when a relation lock is present, as there is no possibility of conflicts among readers and writers or writers and writers. Each relation lock set by a transaction is represented by a physical lock control block that is attached to the relation's static lock control block (as described earlier).

Tuple Locks

Tuple locks are managed differently than relation locks. They have entries in a lock table that hold their tuple address, their lock mode, and the transaction id of their transaction. There is no need to keep a list of blocking transactions for *each* tuple lock of a transaction, since they will all be released at the same time. Instead, the list of

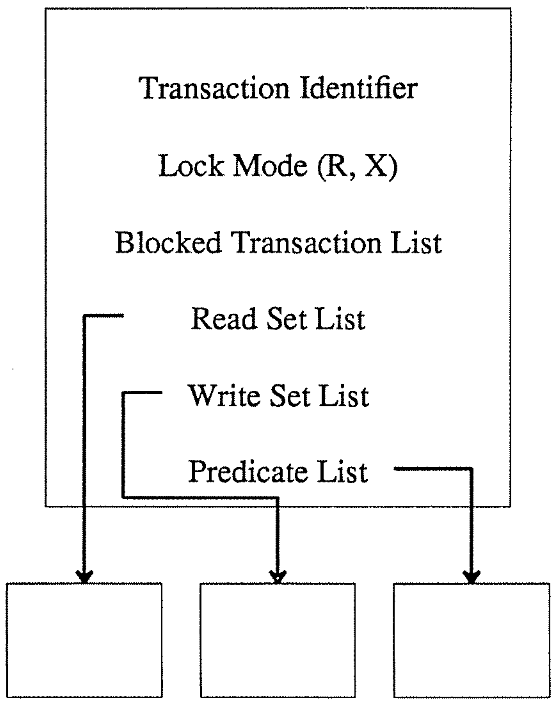


Figure 6.2 — Transaction Lock Control Block Structure

blocking transactions can be combined and kept in a transaction lock control block. There are several hidden costs that are a part of tuple-level locking: When tuple locks are used, latches for read and update operations on index structures and partitions are required. Each tuple read or write operation must set a partition latch and, if an indexed field is updated or the index is used in a search operation, an index latch must also be set. Also, writers of tuples in a relation must check all outstanding read predicates for that relation — a potentially time-consuming task.

Predicate Locks

When locking at the tuple level, the problem of phantom tuples may appear [Eswar 76]. Precision Locks, a form of predicate locks, have been proposed as a method for dealing with phantoms [Jorda 81]. Precision Locks do not try to detect intersection of read and/or write predicates, as this problem is recursively unsolvable for general predicates [Eswar 76]. Instead, writers of values simply compare each new value to be written against all outstanding read predicates, and readers compare their read predicates against a list of all outstanding updated tuples.

Our method uses regular shared and exclusive locks for write/write and write/read synchronization, and it requires readers to post their predicates to keep out writers of phantoms (read/write synchronization). In other words, our method uses tuple locks to prevent readers and writers from seeing or modifying the uncommitted data of other writers, and it posts read predicates to allow readers to keep out writers of phantom tuples. This is similar to the Precision Lock mechanism in the sense that it tests a writer's values against a list of read predicates. The difference between the methods is that the Precision Locks algorithm tests a requesting reader's predicate against a list of outstanding updates for write/read synchronization, whereas our method uses simple shared and exclusive locks.

Read predicates are created when transactions are compiled. The predicates for a transaction are attached to its relation lock control block (Figure 6.1). Each node in the predicate list holds a small portion of compiled code that tests potential write values against the predicate contained in the compiled code. If a writer blocks on a read predicate lock, then it is entered in the transaction lock control block of the transaction holding the predicate lock.

Index Component Locks

When a relation's lock status is at the tuple level, indices are latched at the index level rather than the partition level, as a partition-level latch would have to be checked for every index node touched in a search operation, slowing it down considerably. Thus, for performance reasons, the entire index is latched during updates. (Since single update operations are relatively fast, this should not significantly inhibit concurrency.) Index-node locks are used to ensure that the order of updates applied to index nodes corresponds to transaction commit order (and thus, transaction REDO order). Only write operations need to observe index node locks, however, as the read/write synchronization for tuples is done in the relation itself. Except *during* update operations, an index is sure to be in a consistent state, so an index search can still proceed, using tuple pointers in the index only as search path information. When locking is done at the relation level, the updated index nodes have their addresses entered in an index node write list just as modified tuples would have their addresses entered in the tuple write list.

For T Trees, pointers to lock control blocks are kept in the actual nodes, and, for Modified Linear Hashing, the lock control block pointers are kept in a separate lock pointer array. One lock corresponds to a number of hash entries; the number is made to be a power of 2, so that checking the lock will be easy, as it will require just a mask operation on the address of the hash entry. Any nodes that participate in a T Tree rotation operation or a Hash table split must also be locked. Locking all T Tree nodes participating in a tree rotation can be expensive, but rotations occur infrequently (as shown in Chapter 3). The proposed T+ Tree structure also requires locks, but only on the leaves of the tree. Updating the tree causes at most two nodes to be locked, possibly making it less costly to lock during updates than the T Tree.

6.5. Lock De-escalation

Typical transactions are assumed to be short, so most relations will not be referenced by more than one transaction at a time. There will, however, be some relations (for example, catalogs) that will be touched by every transaction and will usually need to be shared. Unless it is known in advance that a relation needs tuple-level locks, relations always start off with their lock status set at the relation level. Locks de-escalate in size when the relation-level lock held by one transaction is found to be significantly inhibiting the progress of other transactions. The decision to de-escalate is determined by two factors: the number of transactions waiting on a lock and the amount of time they have been waiting. Each time a relation lock acquires another waiting transaction, the lock manager increments a de-escalation count for that relation. Also, the lock manager periodically increments the de-escalation count for all relations with outstanding relation-level locks. When the de-escalate count for a relation lock reaches a specified threshold, the relation lock is de-escalated into tuple locks and predicate lists.

There are several ways that lock de-escalation could be accomplished. One approach would be to restart all transactions currently holding the relation-level locks, thus quiescing the relation, changing its lock status, and rerunning the transactions at the new locking granularity. However, restarting transactions could potentially waste a large amount of work, since long transactions would be the most likely ones to be aborted. Another way to de-escalate locks would be to wait until all active transactions finish, holding off new transactions until the relation became quiescent, and then set the locking status to tuple-level and allow transactions to continue. Unfortunately, it is not possible to predict when transactions holding the relation-level lock will finish, and concurrency may be significantly reduced in the meantime.

Instead of either waiting for or aborting active transactions in order to de-escalate relation locks, the lock manager de-escalates relation-level locks into tuple-level locks "on the fly" during normal transaction processing. This works as follows: Transactions save their tuple-level read and write sets and read predicates when running at relation-level lock status. These read/write sets and predicates are simply linked lists that are attached to the transaction's relation-level lock control block (Figure 6.1). Inserting into a transaction's read or write set is fast, as it is not shared with any other transaction, so latches are not needed. When de-escalation of a relation's lock(s) is required, the lock manager temporarily suspends all transactions currently holding locks on the relation, changes their relation locks into tuple locks using these lists, changes the lock granularity to tuple-level, and then allows the transactions to continue at the new locking granularity level. This process of changing lock granularity is obviously expensive, so it should not be done frequently; if a relation often needs de-escalated locks then it should probably always operate at the tuple lock granularity level. Note that the cost of de-escalation can be calculated from the number of read and write entries that have been posted by transactions holding relation-level locks on a particular relation. If a transaction holds locks on a significant percentage of the tuples in a relation, then it will probably not be cost effective to de-escalate that relation lock.

6.6. Lock Escalation

Since tuple-level locks are much more costly to use than relation-level locks, tuple locks are escalated into relation locks as soon as possible. Except for relations with a permanent tuple-level lock status, lock escalation for a relation will be done when there is only one transaction or only read-only transactions referencing it. However, there are times when the lock status must be temporarily forced to the relation level so that a relation-level operation can be done without locking each individual tuple. An important case of this sort is the checkpoint operation. Checkpoint transactions should not be kept waiting long, if possible, and they should not be starved by other transactions that might have their tuple-level locks granted while the checkpoint transaction waits for the relation-level lock to be granted. However, a checkpoint transaction should never cause deadlock, so it cannot prevent another transaction from accumulating more tuple locks if it already holds some. Therefore two constraints must hold:

(1) For a given relation, tuple-level locks should not be granted to transactions that don't already hold tuple locks if there are any transactions waiting for a forced-escalation relation-level lock.

(2) Tuple-level locks must be escalated to a relation-level lock as soon as possible, so the transaction waiting for the forced-escalation lock does not wait unnecessarily.

As mentioned previously, transactions create relation lock control blocks for each relation they reference, regardless of the locking status of those relations. If a transaction already has a relation control block for a relation, then it is allowed to ask for tuple locks in that relation. When it doesn't have a relation control block, then it must wait behind any other transactions requesting forced-escalation locks. The first transaction to request a forced-escalation lock is given a lock control block so that other transactions may be kept in its queue if they must wait for it. Subsequent transactions making forced-escalation lock requests also block on the first transaction to make such a request.

After escalation, tuple and index-node locks are discarded. Although these locks could be kept in case of another de-escalation, we assume that escalation and de-escalation will not happen often enough for such lock information to be reused. In fact, if a significant number of de-escalation and escalation operations occur for a particular relation, then it should probably be fixed at tuple-level locking status.

When a relation does not have any forced-escalation locks waiting on it, it is not necessary to *immediately* escalate its tuple locks into relation locks, but it appears that doing so will be less expensive than continuing to lock at the tuple level. The actual lock escalation operation should be relatively inexpensive, as it only involves releasing the tuple lock control blocks. Then, once the relation locks are restored, transactions no longer need to pay the cost of checking read predicates for write operations, or latching partitions or indices during read or write operations.

6.7. Deadlock Detection and Resolution

Methods for detecting and resolving deadlocks are well-known [Gray 78]. A waits-for graph can be constructed by reading the lists of transactions that hold locks on relations and their transaction blocked lists. Cycles of transactions waiting for transactions are looked for as opposed to cycles involving individual locks. Once deadlock has been detected, the deadlock resolution method preserving transactions that have the largest read and write sets should be used [Agraw].

6.8. Summary

A concurrency control algorithm based on multi-level lock granularities has been presented. The algorithm attempts to reduce the overall cost of locking by using fine granularity locks only when they are needed to increase the level of sharing in the system; otherwise, the less expensive, coarse granularity locks are used. Using only tuple-level locks would not be cost-effective, as without relation-level locks, transactions performing operations on whole relations would need to lock every tuple in the relation. Also, using only relation-level locks prevents needed sharing of data in many places; database catalogs, for instance, are referenced by every transaction and would become a concurrency bottleneck. It is possible that some relations could be locked with tuple locks and others could be locked with relation locks, but that assumes that relations always have a particular reference pattern. De-escalating locks allows tuple-level sharing of data when it is needed, but otherwise keeps the locking status at the relation-level where the cost of locking is less.

Chapter 7
Conclusion

A design for a memory-resident database system covering five main areas of database research has been presented. The five areas are: database software architecture, index structures, query processing technique, recovery, and concurrency control. Each area was examined in the light of the unique problems of a memory-resident database system, and new ways of exploiting the various aspects of memory-residence were addressed.

An important feature in the memory-resident database environment is the ability to address database entities directly. Direct addressability allows tuple pointers to be used anywhere values or whole tuples would otherwise be needed. In Chapter 2, the software architecture of the database was described, showing how relations, temporary relations, indices, links, and database catalogs can be designed to exploit the advantages of a memory-resident database environment.

Chapter 3 presented a new main memory index structure, the T Tree, and compared it against AVL Trees, simple arrays, B Trees, Chained Bucket Hashing, Extendible Hashing, Linear Hashing and Modified Linear Hashing. The T Tree and Modified Linear Hashing, both structures created or tuned to work in a main memory environment, were shown to be the best order-preserving and hash-based index structures (respectively), as they provided the best performance to storage cost ratios. Index structures for main memory have different formulas for determining search and update costs. Machine independent measurements indicated that index control structure processing cost is equally as important as the number of compares done during a search or update operation.

Chapter 4 examined algorithms for computing the relational join and project operations in a main memory database management system. Several join algorithms were tested: Hash Join, T Tree Join, Nested Loops Join, T Tree Merge Join, and Sort Merge Join. When T Tree indices were available, the T Tree Merge Join was the preferred method, except in a few special cases. A more general method, the Hash Join, showed reasonable performance in all cases, and did not require any extra support from existing indices. Precomputed joins (using links), although not implemented and tested, appear to be an excellent join method under certain circumstances. Their use can potentially reduce the execution time of a join operation by an order of magnitude. Finally, it was shown that hashing is the dominant algorithm for processing projections in main memory.

Chapter 5 presented previous work in recovery, both for traditional disk-based systems and for memory-resident systems, and argued that none of the existing designs completely satisfy the needs of a high-performance memory-resident database system. Such a system needs a fast, efficient logging mechanism that can assimilate log records as fast or faster than they can be produced, efficient checkpoint operations that can amortize the cost of a checkpoint over many updates to the database, and a recovery mechanism that is geared toward allowing transactions to run as soon as possible after a crash. A design was presented that meets these three criteria. With the use of stable memory and a recovery processor, the logging mechanism will not be the performance bottleneck. Checkpoint operations for partitions are triggered when the partitions have received a significant number of updates, so the cost of a checkpoint operation is amortized over many updates. Finally, recovery of data in our design is oriented toward transaction response time. After a crash, relations that are requested by transactions are recovered first so that the transactions can begin processing quickly.

In Chapter 6, a concurrency control algorithm based on two levels of lock granularities was presented. The algorithm attempts to reduce the overall cost of locking by using tuple-level locks only when they are needed to increase the level of sharing in the system; otherwise less expensive, relation-level locks are used. Latches are used throughout the database system to restrict access to partitions and indices while they are temporarily in inconsistent states. When locking is done at the tuple level, a form of predicate read locks are used to prevent phantom tuples from appearing.

A basic theme of this dissertation has been to reduce the amount of work done to accomplish various tasks. One method of reducing the amount of work is by saving results of computations and reusing them rather than recomputing them. For example, links can greatly reduce the cost of a join operation because they are computed once and used many times. Compare routines used in indices, join operations, project operations, and lock predicates are compiled once and used many times, thus removing the cost of interpreting the type of compare that needs to be performed. Finally, other work-reducing techniques involve amortizing the cost of an operation over many steps, thus paying less total cost, and using expensive operations only when necessary. The partition checkpoint operation is usually done only after the partition has accumulated a large number of updates, so the cost of the checkpoint operation is amortized over many updates to the partition. The locking method uses expensive tuple locks only when they are needed to increase the level of sharing in the database system, as otherwise the less expensive relation locks are used.

7.1. Future Work

The area of memory-resident database systems is relatively new, so there are many possible avenues for future work. In each of the areas presented in this dissertation there is much more to be done.

In the area of index structures, the T+ tree has been conceived, but its algorithms have not been specified in detail. Concurrency and recovery methods for main memory might work well with the T+ Tree, as the fast binary search trees like the AVL Tree and the T Tree tend to allow many nodes to be affected by updates to the tree, but concurrency and recovery algorithms like the ones presented here favor index structures like the T+ Tree that can restrict the effects of their updates to one or two nodes. The new hardware environment of a memory-resident database system presents another problem for index structures. When the disk, the slowest level of the memory hierarchy, is removed, the differences in speed between the other levels of memory may become more visible. The amount of interaction between the processor cache and the main memory may be important in a memory-resident database system, just as the amount of interaction between the memory and disk is important in a disk-resident database system. Just as disk-oriented index structures use the buffer pool to reduce disk accesses, perhaps memory-oriented index structures should use the processor cache more effectively to reduce memory traffic. Unfortunately, cache technology is not as stable as disk technology — cache line sizes typically vary from 4 bytes to 128 bytes, cache replacement algorithms range from direct to set-associative to fully associative, and cache sizes range from a few hundred bytes to several megabytes. Although processor cache use is important, it is not obvious how general algorithms can be developed with the current diversity of cache specifications.

Query processing has been studied extensively for disk-based systems and query optimization techniques are well known. In this dissertation we examined the costs of various select, project and join algorithms. We found the T Tree Merge Join and the Hash Join to be the preferred join methods most of the time, and we found hashing to be the preferred method for performing duplicate elimination. These findings need to be included in query optimization cost formulas that are oriented toward a memory-resident database system.

Concurrency control algorithms have been examined in a disk environment and judged in terms of such performance metrics as transaction throughput and response time. In a memory-resident environment, the cost of the actual concurrency control mechanism must be taken into account, as it can essentially double the cost of a transaction since the relative cost of database accesses are low. A new concurrency control algorithm has been presented that attempts to reduce the overall cost of locking. A new more detailed model for comparing concurrency control methods in main memory is needed to compare this newly proposed method with existing methods.

Finally, a simple analysis of our proposed recovery method gives a fair estimation of possible performance, but a very detailed analysis or simulation is needed to determine how the various logging, checkpointing, and recovery operations interact when all three operations are running in parallel. Also, it would be interesting to see exactly to what extent we can take advantage of possible added parallelism in the recovery process.

References

[Agraw 85a] R. Agrawal, M. Carey and M. Livny, Models for studying concurrency control Performance: Alternatives and Implications, *Proc. ACM SIGMOD Conf.*, May 1985.

[Agraw 85b] R. Agrawal and D. DeWitt, Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation, *ACM Trans. on Database Systems 10*,4 (December 1985).

[Agraw] R. Agrawal, M. Carey and L. McVoy, The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems, *IEEE Transactions on Software Engineering*, . (to appear).

[Aho 74] A. Aho, J. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.

[Amman 85] A. Ammann, M. Hanrahan and R. Krishnamurthy, Design of a Memory Resident DBMS, *Proc. IEEE COMPCON*, San Francisco, February 1985.

[Astra 76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade and V. Watson, System R: Relational Approach to Database Management, *ACM Trans. on Database Systems 1*,2 (June 1976), 97-137.

[Babb 79] E. Babb, Implementing a Relational Database by Means of Specialized Hardware, *ACM Trans. on Database Systems 4*,1 (March 1979), 1-29.

[Berns 81] P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys 13*,2 (June 1981), 185-221.

[Bitto 83] D. Bitton, H. Boral, D. J. DeWitt and W. Wilkinson, Parallel Algorithms for the execution of Relational Database Operations, *ACM Trans. on Database Systems 8*,3 (September 1983), 324.

[Blasg 77] M. Blasgen and K. Eswaran, Storage and Access in Relational Databases, *IBM Systems Journal 16*,4 (1977).

[Blasg 79] M. Blasgen, J. Gray, M. Mitoma and T. Price, The Convoy Phenomenon, *Operating Systems Review 13*,2 (April 1979), 20-25.

[Bratb 84] K. Bratbergsengen, Hashing Methods and Relational Algebra Operations, *Proc. 10th Conf. Very Large Data Bases*, Singapore, August 1984, 323.

[Carey 83] M. Carey, An Abstract Model of Database Concurrency Control Algorithms, *Proc. ACM SIGMOD Conf.*, May 1983, 97-107.

[Carey 84] M. J. Carey and M. R. Stonebraker, The Performance of Concurrency Control Algorithms for Database Management Systems, *Proc. 10th Conf. Very Large Data Bases*, 1984, 107-117.

[Chamb 81] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolo, P. G. Selinger, M. Schkolnik, D. R. Slutz, I. L. Traiger, B. W. Wade and R. A. Yost, A History and Evaluation of System R, *Comm. of the ACM 24*,10 (Oct. 1981), 632-646.

[Comer 79] D. Comer, The Ubiquitous B-Tree, *Computing Surveys 11*,2 (June 1979).

[Date 81] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley, 1981. (3rd Ed.).

[Date 85] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley, 1985. (4th Ed.).

[DeWit 84] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, Implementation Techniques for Main Memory Database Systems, *Proc. ACM SIGMOD Conf.*, June 1984, 1-8.

[DeWit 85] D. J. DeWitt and R. Gerber, Multiprocessor Hash-Based Join Algorithms, *Proc. 11th Conf. Very Large Data Bases*, Stockholm, Sweden, August 1985.

[Eich 86] M. Eich, MMDB Recovery, Southern Methodist Univ. Dept. of Computer Sciences TR # 86-CSE-11, March 1986.

[Elhar 84] K. Elhardt and R. Bayer, A Database Cache for High Performance and Fast Restart in Database Systems, *ACM Trans. on Database Systems 9*,4 (December 1984), 503-526.

[Eswar 76] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, The Notions of Consistency and Predicate Locks in a Database System, *Comm. of the ACM 19*,11 (Nov. 1976).

[Fagin 79] R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, Extendible Hashing : A fast access method for dynamic files, *ACM Trans. on Database Systems 4*,3 (Sept. 1979), 315-344.

[Fishe 86] M. Fishetti, Technology '86: Solid State, *IEEE Spectrum 23*,1 (January 1986).

[Garci 83] H. Garcia-Molina, R. J. Lipton and J. Valdes, A Massive Memory Machine, Princeton Univ. EECS Dept TR # 315, July 1983.

[Goodm 86] J. Goodman, Personal communication, July 1986.

[Gray 78] J. N. Gray, Notes on Database Operating Systems, in *Operating Systems: An Advanced Course*, vol. 60, Springer-Verlag, New York, 1978, 393-481.

[Gray 81] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu and I. Traiger, The Recovery Manager of System R, *ACM Computing Surveys 13*,2 (June 1981).

[Gray 82] J. Gray, personal communication to David DeWitt, April 1982.

[Gray 85] J. Gray, D. Gawlick, J. Good, P. Homan and H. Sammer, One Thousand Transactions Per Second, *Proc. IEEE COMPCON*, San Francisco, February 1985.

[Haerd 83] T. Haerder and A. Reuter, Principles of Transaction-Oriented Database Recovery, *ACM Computing Surveys 15*,4 (December 1983).

[Hagma 86] R. Hagmann, A Crash Recovery Scheme for a Memory Resident Database System, *IEEE Transactions on Computers*, 1986. (to appear).

[Horwi 85] S. Horwitz and T. Teitelbaum, Relations and Attributes: A Symbiotic Basis for Editing Environments, *Proc. of the ACM SIGPLAN Conf. on Language Issues in Programming Environments*, Seattle, WA, June

1985.

[IBM 79] IBM, *IMS Version 1 Release 15 Fast Path Feature Description and Design Guide*, IBM World Trade Systems Centers (G320-5775), 1979.

[IBM 84] IBM, *Guide to IMS/VS V1 R3 Data Entry Database (DEDB) Facility*, IBM International Systems Centers (GG24-1633-0), 1984.

[IBM] IBM, IBM 370 Principles of Operation.

[Jorda 81] J. Jordan, J. Banerjee and R. Batman, Precision Locks, *Proc. ACM SIGMOD Conf.*, May 1981.

[Kjell 84] P. Kjellberg and T. Zahle, Cascade Hashing, *Proc. 10th Conf. Very Large Data Bases*, Singapore, August 1984, 481-492.

[Knuth 73] D. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., 1973.

[Kohle 81] W. Kohler, A survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems, *ACM Computing Surveys 13*,2 (June 1981).

[Lamps 79a] B. W. Lampson, An Open Operating System for a Single-User Machine, Proc. of 7th Symp. Operating Systems Principles, Pacific Grove, Calif., December 1979.

[Lamps 79b] B. Lampson and H. Sturgis, Crash Recovery in a Distributed Data Storage System, XEROX Research Report, Palo Alto, California, 1979.

[Larso 80] P. Larson, Linear Hashing with Partial Expansions, *Proc. 6th Conf. Very Large Data Bases*, Montreal, Canada, October 1980, 224-231.

[Lelan 85] M. Leland and W. Roome, The Silicon Database Machine, *Proc. 4th Int. Workshop on Database Machines*, Grand Bahama Island, March 1985.

[Linds 79] B. Lindsay, P. Selinger, C. Galtieri, J. Gray, R. Lorie, T. Price, F. Putzolu, I. Traiger and B. Wade, Notes on Distributed Databases, IBM Research Report RJ 2571, San Jose, California, 1979.

[Linds 85] B. Lindsay, Personal Communication, November 1985.

[Linto 84] M. Linton, Implementing Relational Views of Programs, *Proc. of the ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[Litwi 80] W. Litwin, Linear Hashing : A New Tool For File and Table Addressing, *Proc. 6th Conf. Very Large Data Bases*, Montreal, Canada, October 1980.

[Litwi 85] W. Litwin, Personal communication, 1985.

[Lorie 77] R. Lorie, Physical Integrity in a Large Segmented Database, *ACM Trans. on Database Systems* 2,1 (March 1977), 91-104.

[Macke 86] L. Mackert and G. Lohman, R* Optimizer Validation and Performance Evaluation for Local Queries, *Proc. ACM SIGMOD Conf.*, May 1986.

[March 81] S. March, D. Severance and M. Wilens, Frame Memory: A Storage Architecture to Support Rapid Design and Implementation of Efficient Databases, *ACM Trans. on Database Systems 6*,3 (September 1981).

[Mulli 84] J. Mullin, Unified Dynamic Hashing, *Proc. 10th Conf. Very Large Data Bases*, Singapore, August 1984, 473-480.

[Peter 82] J. Peterson and A. Silberschatz, *Operating Systems Concepts*, Addison-Wesley, Reading, Mass., 1982.

[Pu 85] C. Pu, On-the-Fly, Incremental, Consistent Reading of Entire Databases, *Proc. 11th Conf. Very Large Data Bases*, Stockholm, Sweden, August 1985, 369-375.

[Reute 80] A. Reuter, A Fast Transaction-oriented Logging Scheme for UNDO Recovery, *IEEE Trans. Software Eng. SE-6*(July 1980).

[Reute 84] A. Reuter, Performance Analysis of Recovery Techniques, *ACM Trans. on Database Systems 9*,4 (December 1984), 526-559.

[Ries 79] D. Ries and M. Stonebraker, Locking Granularity Revisited, *ACM Trans. Database Systems 4*,2 (June 1979).

[Salem 86] K. Salem and H. Garcia-Molina, Crash Recovery Mechanisms for Main Storage Database Systems, Princeton Univ. Comp. Sci. Dept. Tech. Rep. CS-TR-034086, April 1986.

[Schwa 84] P. Schwarz, Transactions on Typed Objects, Ph.D. Dissertation (Carnegie-Mellon University), December 1984.

[Selin 79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, Access Path Selection in a Relational Database Management System, *Proc. ACM SIGMOD Conf.*, June 1979.

[Shapi 86] L. D. Shapiro, Join Processing in Database Systems with Large Main Memories, *ACM Trans. on Database Systems*, 1986. (to appear).

[Snodg 84] R. Snodgrass, Monitoring in a Software Development Environment: A Relational Approach, *Proc. of the ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[Stone 76] M. Stonebraker, E. Wong, G. D. Held and P. Kreps, The Design and Implementation of INGRES, *ACM Trans. on Database Systems 1*,3 (Sept. 1976), 189-222.

[Stone 86] M. Stonebraker, Inclusion of New Types in Relational Database Systems, *Proc. 2nd Data Eng. Conf.*, Los Angeles, California, February 1986.

[Valdu 84] P. Valduriez and G. Gardarin, Join and Semijoin Algorithms for a Muiltiprocessor Database Machine, *Trans. on Database Systems 9*,1 (March 1984), 133.

[Verho 78] J. Verhofstad, Recovery Techniques for Database Systems, *ACM Computing Surveys 10*,2 (June 1978).

[Warre 81] D. H. D. Warren, Efficient Processing of Interactive Relational Database Queries Expressed in Logic, *Proc. 7th Conf. Very Large Data Bases*, Cannes, Fance, September, 1981.

[Willa 84] D. E. Willard, New TRIE Data Structures Which Support Very Fast Search Operations, *Journal of Computer and Systems Sciences 28*,3 (June 1984), 379-394.