

Concurrency Control and Performance Evaluation of B – Trees

Tanmoy Dutta

B.Tech. Indian Institute of Technology, Kharagpur, India
1988


A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science

September, 1989

The thesis "Concurrency Control and Performance Evaluation of B-trees" by Tanmoy Dutta
has been examined and approved by the following Examination Committee.

~~Dr. Harry H. Porter~~
~~Adjunct Assistant Professor~~
~~Thesis Research Advisor~~

Dr. David Maier
Professor

Dr. Michael Wolfe 
Associate Professor

Contents

Acknowledgement	vi
Abstract	vii
1. Prologue	1
1.1. Introduction	1
1.2. Concurrency Control and B-trees	4
1.3. Statement of Thesis	5
2. Concurrency Control in Database Systems	6
2.1. Preliminaries	6
2.2. Serializability and Nested Transactions	7
2.3. Locking Based Methods	9
2.3.1. Two-Phase Locking Protocols	10
2.3.1.1. Basic Two-Phase Locking	10
2.3.1.2. Conservative Two-Phase Locking	10
2.3.1.3. Strict Two-Phase Locking	11
2.3.1.4. Multigranularity Locking	11
2.3.1.5. Multiversion Two-Phase Locking	13
2.3.2. Non-Two-Phase Locking Protocols	13
2.3.3. The Effect of Granularity on Locking	15
2.4. Validation Based Methods	16
2.5. Timestamp-Ordering-Based Methods	18
2.6. Reservation Based Methods	18
2.7. Semantic Methods in Concurrency Control	19
2.7.1. Dynamic Determination Of Compatibility	19
2.7.2. Modular Concurrency Control Method	20
3. B-Trees in Database Systems	22
3.1. Introduction	22
3.2. Review of B-Trees and Their Variants	22
3.2.1. B^+ -Trees	23
3.2.2. B^* -trees	24
3.2.3. Prefix B^+ -trees	25
3.2.4. B_{link} -Trees	26
3.2.5. Hysterical B-Trees	26
3.3. Operations on B-Trees	27
3.4. Deferred versus Aggressive Splitting	28
3.5. Fringe Analysis	29
3.6. Performance Metrics of B-Trees	30

3.6.1.	The Average Storage Utilization	31
3.6.2.	The Expected Number of Splits per Insertion	35
3.6.2.1.	Deferred Splitting	35
3.6.2.2.	Aggressive Splitting	41
3.6.3.	The Average Cost of an Operation	42
3.6.3.1.	Search for a Key	42
3.6.3.2.	Insertion of a Key	43
3.6.3.3.	Deletion of a Key	43
3.7.	Other Optimizations in B^+ -Trees	44
3.7.1.	Partial Expansions of Nodes	44
3.7.1.1.	Expected Number of Full Expansions Per Insertion	44
3.7.1.2.	Average Storage Utilization	49
3.7.1.3.	Probability of a Partial Expansion	51
3.7.2.	Hashing at the Leaf Level	51
3.8.	Summary	53
4.	Performance of B^+-Trees	54
4.1.	Introduction	54
4.2.	Simulation Parameters and Workload Description	55
4.3.	Test Data Generation	57
4.3.1.	The Inverse Transform method	58
4.3.2.	The Acceptance Rejection Method	58
4.4.	Review of Probability Distributions	59
4.4.1.	The Uniform Distribution	59
4.4.2.	The Normal Distribution	60
4.4.3.	The Exponential Distribution	62
4.4.4.	The Gamma Distribution	63
4.4.5.	The Beta Distribution	65
4.4.6.	The Log-Normal Distribution	66
4.5.	Operation Execution Times	68
4.6.	The Average Storage Utilization	71
4.7.	The Average and the Total Insertion Time	74
5.	Concurrency Control of B-Trees	103
5.1.	Overview	103
5.2.	Log-Sequence Serializability versus Operational Serializability	105
5.3.	Concurrency Control Algorithms	107
5.3.1.	Concurrency Control of Bottom-Up B-Trees	109
5.3.1.1.	Type 1 Algorithms	109
5.3.1.1.1.	Samadi-Parr Algorithm	109
5.3.1.1.2.	Bayer and Schkolnick's Algorithms	110
5.3.1.1.3.	Ellis's Algorithm	112
5.3.1.1.4.	Kwong and Wood's Algorithm	112

5.3.1.2.	Type 2 Algorithms	113
5.3.1.2.1.	Ellis's Algorithm	114
5.3.1.2.2.	Miller and Snyder's Algorithm	116
5.3.1.2.3.	Lehman and Yao's Algorithm	117
5.3.1.2.4.	Kwong and Wood's Algorithm	117
5.3.1.2.5.	Sagiv's Algorithm	118
5.3.2.	Protocols for Top-Down Trees	119
5.3.2.1.	Algorithm One	119
5.3.2.2.	Algorithm Two	125
5.3.3.	Operation Specific Locking Protocols	128
5.3.3.1.	Biliri's Algorithm	128
5.3.3.2.	Algorithm Three	130
5.3.4.	Optimistic Concurrency Control	138
5.4.	Comparison of Algorithms	139
6.	Concurrency Control Performance	143
6.1.	Introduction	143
6.2.	Workload Model	144
6.3.	Implementation of Locks	145
6.4.	The Performance Metric	147
6.5.	Results	148
6.5.1.	Total Elapsed Time	148
6.5.2.	Effect of Read and Write Ratio	150
7.	Epilogue	156
7.1.	Summary and Conclusions	156
7.2.	Future Study	157
	Bibliography	159
	Biographical Note	165

List of Figures

2.1	Compatibility Matrix for Multigranularity Locking	12
2.2	Compatibility Matrix for Multiversion Locking protocol	13
2.3	The Effect of Granularity on Locking	15
3.1	Example of a 2-3-tree	23
3.2	Node splitting in B [*] -Trees	25
3.3	Example of Prefix B ⁺ -tree	25
3.4	Structure of B _{link} -trees	26
3.5	Parameter Notation	31
3.6	Effect of Minimum Fullness Factor on Storage Utilization	31
3.7	Effect of Slack Factor on Storage Utilization	35
4.1	The Uniform Distribution	59
4.2	The Normal Distribution	60
4.3	The Exponential Distribution	62
4.4	The Gamma Distribution	64
4.5	The Beta Distribution	65
4.6	The Log-normal Distribution	67
4.7	Insertion Time vs. Size of Tree for Uniform Distribution	77
4.8a	Insertion Time vs. Size of Tree for Normal Distribution ($\mu=1.0, \sigma=0.7$)	78
4.8b	Insertion Time vs. Size of Tree for Normal Distribution ($\mu=1.0, \sigma=1.0$)	79
4.9a	Insertion Time vs. Size of Tree for Exponential Distribution ($\mu=1.0$)	80
4.9b	Insertion Time vs. Size of Tree for Exponential Distribution ($\mu=0.5$)	81
4.10	Insertion Time vs. Size of Tree for Gamma Distribution	82
4.11	Insertion Time vs. Size of Tree for Beta Distribution	83
4.12	Insertion Time vs. Size of Tree for Log-normal Distribution	84
4.13	Deletion Time vs. Size of Tree for Uniform Distribution	85
4.14	Deletion Time vs. Size of Tree for Normal Distribution ($\mu=1.0, \sigma=1.0$)	86
4.15	Deletion Time vs. Size of Tree for Exponential Distribution	87
4.16	Deletion Time vs. Size of Tree for Gamma Distribution	88
4.17	Deletion Time vs. Size of Tree for Beta Distribution	89
4.18	Deletion Time vs. Size of Tree for Log-normal Distribution	90
4.19	Average Storage Utilization for Uniform Distribution	91
4.20	Average Storage Utilization for Normal Distribution	92
4.21	Average Storage Utilization for Exponential Distribution	93
4.22	Average Storage Utilization for Gamma Distribution	94
4.23	Average Storage Utilization for Beta Distribution	95
4.24	Average Storage Utilization for Log-normal Distribution	96
4.25	Insertion Times and Av. Insertion Time vs. Number of Keys for Uniform Distribution	97

4.26 Insertion Times and Av. Insertion Time vs. Number of Keys for Normal Distribution	98
4.27 Insertion Times and Av. Insertion Time vs. Number of Keys for Exponential Distribution	99
4.28 Insertion Times and Av. Insertion Time vs. Number of Keys for Gamma Distribution	100
4.29 Insertion Times and Av. Insertion Time vs. Number of Keys for Beta Distribution	101
4.30 Insertion Times and Av. Insertion Time vs. Number of Keys for Log-normal Distribution	102
5.1 The B ⁺ -tree before both processes start	103
5.2 Interleaving of steps of the two processes	103
5.3 The B ⁺ -tree after step 6	104
5.4 Compatibility Matrix for Bayer and Schkolnick's First Algorithm	110
5.5 Compatibility Matrix for Bayer and Schkolnick's Third Algorithm	111
5.6 Compatibility Matrix for Ellis's Type 2 Algorithm	114
5.7 Compatibility Matrix for Miller and Snyder's Algorithm	116
5.8 Compatibility Matrix for Kwong and Wood's Algorithm	117
5.9 Compatibility Matrix for Algorithm 1	119
5.10 Search procedure for Algorithm One	120
5.11 Insertion procedure for Algorithm One	121
5.12 Deletion procedure for Algorithm One	122
5.13 Compatibility Matrix for Biliris' Algorithm	129
5.14 Convertibility Matrix for Biliris' Algorithm	129
5.15 Compatibility Matrix for Algorithm Three	131
5.16 Convertibility Matrix for Algorithm Three	131
5.17 Search procedure for Algorithm Three	132
5.18 Insertion procedure for Algorithm Three	133
5.19 Deletion procedure for Algorithm Three	133
5.20 Comparison table for bottom-up type 1 algorithms	140
5.21 Comparison table for bottom-up type 2 algorithms	141
5.22 Comparison table for top-down and operation specific algorithms	142
6.1 Index of the protocols compared	148
6.2 Percentages of different operations.	150
6.3 Total Elapsed Time versus number of <i>short</i> transactions	152
6.4 Total Elapsed Time versus number of <i>medium</i> transactions	153
6.5 Total Elapsed Time versus number of <i>large</i> transactions	154
6.6 Effect of read-update ratio on Total Elapsed Time	155

Acknowledgement

I wish to express the thanks I owe to many people for providing emotional, intellectual and financial support during my stay at the Oregon Graduate Center. It was my privilege to have done this research under the guidance of Professor Harry Porter, Professor David Maier and Professor Michael Wolfe. Harry was a constant source of encouragement, innovative ideas, friendship and practicality. Without his persistence and support this thesis would never have been in this shape. Mike Wolfe suggested some necessary modifications and clarified certain concepts. I am grateful to my parents who constantly encouraged me from fifteen thousand miles away and to Rupa, whose letters were the only source of happiness in my life. I am also grateful to Professor Shakti Pramanik of the Michigan State University, Professor Goetz Graefe of the University of Colorado and my friends Pinaki, Meenakshi, Tom and Murali for providing some brilliant ideas and patiently editing my thesis. Most of all, I am ever obliged to the great Dave Maier. He not only taught me Computer Science – the science in it, but also taught me what is meant by a great professor, a great advisor and most importantly a great man – undoubtedly one of the best I have ever seen.

Abstract

The B-tree and its variants have been widely used as a data structure for indexing into large sets of records, especially when stored on secondary storage devices. The guaranteed small time of operations for these data-structures make them quite appealing for database applications. Previous research on B-trees has been concentrated on the case when the restructuring of the tree during an update is propagated bottom-up. Our research is on top-down restructuring of the tree.

We have compared the two approaches mathematically and verified the analysis by simulations. The behavior of the two approaches under different distributions of keys has also been discussed. Next, we looked at the various techniques for concurrency control of bottom-up B-trees and proposed two new concurrency control protocols for top-down B-trees. We have also studied concurrency control protocols which combine both the approaches and presented a new protocol. Further, we implemented the various protocols, proposed a metric of comparison and evaluated their performance. Our results show that, under certain circumstances, top-down B-trees perform better and are indeed a viable alternative of the bottom-up counterparts.

CHAPTER 1

Prologue

1.1. Introduction

Concurrency control is the activity of coordinating concurrent accesses to resources in an environment allowing more than one process. It is the synchronization of access to shared data while preserving the illusion that the user is executing alone on a dedicated system. With the growth of technology such environments are becoming common.

Various techniques of achieving concurrency control have been cited in the literature. Many metrics have been suggested to judge their performance. The most common of these metrics include the degree of concurrency supported, the complexity of the resulting logic, the protocol maintenance overhead, the total space overhead, the delay incurred in the response time, the amount of concurrent processing during restarts, the extent of system-induced transaction rollbacks, the overheads of deadlock detection, prevention or avoidance, the number of locks used, the degree of commutativity of the locks and many others.

In the context of centralized database systems, the problem of concurrency control is well understood and current research is directed towards performance analysis and optimizations, extensions to the mathematical theory [BeG81] and modifications of the algorithms for better performance in specialized applications. However, a major problem encountered in comparing concurrency control protocols is the variation in the assumptions underlying them. At times these assumptions are contradictory, making comparisons even more difficult. Moreover, many of the techniques are incorrect, unreliable or perform unacceptably in situations not accounted for in the assumptions. A survey of the various

concurrency control protocols is provided in Chapter 2.

Most of the concurrency control algorithms intended for database systems do not assume any inherent structuring of data. In that sense they are purely syntactical and provide less parallelism in the access patterns than can be extracted. The reason behind this is the requirement of *serializability*. A correct protocol should not allow a schedule that violates serializability. Checking whether a schedule of transactions is serializable or not is an *NP-Complete* problem and various heuristics are employed to attain serializability of a set of transactions. Different heuristics give rise to the different protocols for concurrency control. As a consequence of adopting heuristics, some possibly valid schedules have to be sacrificed also.

With semantic information about the structure of data, the data access pattern is better known and more concurrency can be achieved using this knowledge. In order to utilize the knowledge about the structure, the general purpose algorithms are often modified to such an extent that they bear little resemblance to the original versions. However, most of these special purpose algorithms are derived from a few general protocols and are modified according to the different properties of the data structure they exploit to achieve the extra parallelism.

The term *synchronization* is used to describe either of the two problems [Koh81]:

- (a) Specification and control of joint activity of co-operating sequential processes.
- (b) Synchronization of concurrent access to shared data by multiple processes.

The solutions to these two problems are related in the sense that access to shared data can be synchronized by controlling the processes which access the data. Thus, solutions of one often solve the other, too. Conventional operating systems solve the problem of synchronization by providing *semaphores*, *test and set* instructions, *locks* (e.g., spin), *event queues*, *monitors*, *critical regions*, etc. The basic requirements of all these solutions are *fair*

scheduling, termination and mutual exclusion. Though concurrency control algorithms provide concurrent access to the data both in operating systems and in database systems, solutions of synchronization problems in operating systems need not necessarily work in the case of database systems. The difference is that the operating system assumes that the resources are unchanged by use (like the CPU or a network channel), while a database system assumes that the resources can be modified through use. In fact, the problem in the case of database systems is strictly more difficult [BeG81].

Consider two processes P_1 and P_2 requiring concurrent access to the resources R_1 and R_2 . The following is a perfectly valid schedule in an operating system environment: P_1 uses R_1 , P_2 uses R_1 , P_2 uses R_2 , P_1 uses R_2 . However, in a database system, this schedule is not always valid. In a banking application for example, consider that P_2 debits one account (R_1) and then credits another (R_2). If P_2 checks both balances, it may see R_1 after it has been debited but before R_2 has been credited [CBT74].

Loosely speaking, another reason for the extra strictness in the case of database systems is the added requirement of *integrity constraints*. The *state* of a database system is defined to be the contents of all the data objects present in the system and is often represented as a tuple $\langle x_1, x_2, \dots, x_i \rangle$. A state is *consistent* if it conforms to the integrity constraints of the system. A *transaction* is a set of operations which take the database from one consistent state to another. However the constituent operations individually may not necessarily preserve state. In operating system problems, integrity constraints are not imposed on the resources (analogous to data items) accessed by the processes (analogous to transactions). Hence, the problem of synchronization in the case of operating systems is less difficult.

1.2. Concurrency Control and B-trees

B-trees occur very frequently in databases as a data structure for organizing and indexing large amounts of data. In a B-tree, data is organized into pages and the pages are stored on some direct access medium such as disk. The pages are linked together as a multiway tree satisfying certain constraints on the contents of these pages. B-trees were originally proposed by Bayer and McCreight [BaM72] for maintaining large ordered indices of dynamic random access files. Several variants of B-trees have been reported in the literature to cater to different needs. The variants can be broadly categorized into two classes — *structural* and *operational*. The structural variants have the same operation semantics but have a slightly modified data-structure. The operational variants on the other hand have different operation semantics preserving the basic data-structure. Our research is primarily on the operational variants of B-trees.

In view of the prominence of multi-processing and multi-user environments, accesses to these large quantities of data must be parallelized too. With the development of object-oriented systems, this need has been felt even more. Every object in an object-oriented system has a unique identifier which often serves as an index in the so-called *Object Table*. Fast access to the object table — which is maintained as a B-tree (or a trie) — is necessary (e.g. Gemstone, VBase). Moreover, the object table is the center of activity and all processes need access to it in some form. This provides a major motivation in the control of concurrent access to B-trees. Existing systems like Gemstone [MaS87] provide extremely rudimentary techniques or nothing at all (e.g., Volcano [Gra89]) in controlling concurrent accesses. This thesis explores the feasibility of the techniques reported in literature for controlling concurrent accesses to B-trees.

1.3. Statement of Thesis

In this thesis, we compare two operational variants of B-trees, namely — *top-down B-trees* and *bottom-up B-trees*. While bottom-up B-trees are established to be standard, top-down B-trees have earnest proponents (e.g., [Gra89]) and vocal detractors (e.g., [LeY81]). The proponents tout lower cost of operation while the detractors complain about loss of concurrency resulting from more frequent restructurings of the tree. Our thesis states that when the number of pages residing in main memory is bounded, top-down B-trees have comparable performance to their bottom-up counterparts. We attempt to prove our thesis in two steps. First, we perform probabilistic analysis of both the variants and show that the excess number of splits in the top-down case is only marginally more. To support our analysis and to justify our argument, we perform simulations under six different distribution of keys. The results of the simulation show that in all the cases the number of restructurings is only marginally more in the case of top-down trees. Next, we propose three concurrency control protocols for top-down trees and compare them with six other protocols for bottom-up trees. Our simulations show that the proposed protocols are comparable in performance.

The rest of the thesis is organized in the following way. In Chapter 2 we provide a short summary of the general techniques of concurrency control in database systems. In Chapter 3 we study some properties of B-trees. In Chapter 4 we study the performance of B-trees under different distribution of keys. In Chapter 5 we discuss protocols for concurrency control of B-trees and its variants. In Chapter 6 we discuss the performance of nine concurrency control algorithms and finally in Chapter 7 we summarize our conclusions.

CHAPTER 2

Concurrency Control in Database Systems

2.1. Preliminaries

The primary objective of this chapter is to survey the standard concepts of database concurrency control. Most of our definitions can be found in some form elsewhere, e.g., [BHG87].

A database is a set of data items $\{x, y, \dots, z\}$. Every operation on a data item is either a *read* or a *write*. A *transaction* is an ordered set of operations that preserves the integrity constraints of the database. In the most general case, transactions may be nested, i.e., composed of one or many *sub-transactions*. However, for the sake of simplicity, only non-nested transactions will be considered in this thesis.

DEFINITION 2.1 (e.g., [BHG87]): A transaction T_i is a *irreflexive partial order* given by $T_i = (\Sigma_i, <_i)$ where

- $\Sigma_i = \{r_i[x], w_i[x] \mid x \in \text{database}\} \cup \{a_i, c_i, s_i\}$,
- $(a_i \in T_i) \equiv \neg (c_i \in T_i)$,
- If e_i is either a_i or c_i then for all $p \in \Sigma_i$, $p <_i e_i$,
- For all $x \in \text{database}$, $s_i <_i o_i[x]$, where o_i is r_i or w_i ,
- If $r_i[x], w_i[x] \in T_i$ then either $r_i[x] <_i w_i[x]$ or vice versa.

□

The partial order $(<_i)$ is the *happens before* relation of Lamport [Lam78].

DEFINITION 2.2 (e.g., [Lom77]): An *action* is a collection of operations which takes the database from one state to another. An action is *atomic* if and only if the following are satisfied:

- The execution of the action progresses as if it is the only action,
- The action does not communicate with other actions running concurrently,
- There is no visible change in state until the action is over,
- The action is indivisible.

□

An action is said to be *recoverable* if the effects of the action can be removed from the database if necessary. In database systems, transactions are the units of atomicity. Thus transactions are synonymous to atomic actions. According to [HaR84], a transaction has the following four properties: *Failure Atomicity*, *Consistency*, *Isolation* (or synchronization atomicity), and *Durability* (guaranteed to survive a failure occurring after it has been committed).

DEFINITION 2.3: Two operations *conflict* if they operate on the same data element and either of them is a write.

□

DEFINITION 2.4 (e.g., [BHG87]): A *history* H_T of a set of transactions $T = \{T_1, \dots, T_n\}$ is a partial order on the operations $H_T = (\Gamma, <_H)$ such that:

(a) $\Gamma = \cup_i \Sigma_i$,

(b) $<_H \supseteq \cup_i <_i$,

(c) For two conflicting operations p, q in H_T , either $p <_H q$ or vice versa.

□

DEFINITION 2.5 (e.g., [BHG87]): A *committed projection* of a history, written as $C(H_T)$, is a subset of H_T , where the subset relation refers to the *prefix* property on the partial orders. The prefix consists of the transactions in H_T that are committed.

□

2.2. Serializability and Nested Transactions

In this section we briefly review the concepts involved in the synchronizing of transactions.

DEFINITION 2.6 A *serial execution* is one in which the transactions execute serially one after another.

□

We say that a set of transactions *preserves state* (precisely, preserves consistency of the state) if the execution of the transactions gives rise to a state which preserves the integrity constraints of the database. Thus, if the set of transactions $\{T_1, T_2, \dots, T_n\}$ executes serially (in any order) then, since each individual transaction preserves state, the

resulting state is also preserved. The particular order is called a *serial schedule*.

DEFINITION 2.7: Two schedules are *equivalent* if their histories are equivalent[†].
□

DEFINITION 2.8: A *serializable schedule* of a set of transactions is a temporal ordering among them that is equivalent to a serial schedule.
□

A serializable schedule preserves the state of the database. However the inverse is not necessarily true: there are non-serializable schedules that also preserve the state of the database.

DEFINITION 2.9: A *serialization graph* $SG(H)$ for the history H is a directed graph (V, E) where $V = \{ T_i \mid T_i \in H \text{ and } T_i \text{ is committed} \}$ and $(T_i, T_j) \in E [i \neq j]$ if one of T_i 's operation precedes and conflicts with one of T_j 's operations in H .
□

THEOREM 2.1: A history is serializable if and only if $SG(H)$ is acyclic.
Proof: See [BHG87].
□

In order to prove a history serializable, it is sufficient to prove that the corresponding serialization graph contains no cycles.

Transactions as defined above, are *flat* and the concept of atomicity is *uni-level*. The transaction is the *logical unit* (i.e., the unit which transforms the state) of a database system. Also, as mentioned previously, the transaction is the unit of *atomicity* and *recovery*. This unified model of transactions makes implementation simple but at the cost of parallelism. Intuitively, the unit of logical consistency should be as large as possible, while that of atomicity small, to achieve maximal parallelism. The unit of recovery can be anywhere in between. One would obviously not like to roll back (or undo on demand) a large logical unit

[†]Here, equivalence of two histories refers to equivalence of partial orders.

but would prefer to roll back a few of the atomic units. A transaction can thus be broken up into a set of logically independent sub-steps, where *independence* refers to the mutual exclusivity of the sets of data items on which each one of the sub-steps operate. Serialization can then be done over such *sub-transactions*, guaranteeing consistency of the database. Evidently the number of schedules in such a serialization scheme may be more than in the case of flat transactions and, in the process, more concurrency can be achieved.

Next, we discuss a few of the most common concurrency control methods.

2.3. Locking Based Methods

Locking based methods are by far the most popular of the concurrency control methods discussed in literature. They have been widely implemented, explored and researched.

On an operation request from a transaction, the transaction manager (scheduler) has three options [BSR83]:

- (a) Immediately schedule it.
- (b) Delay it.
- (c) Reject it.

An *aggressive* scheduler tries to schedule an operation as soon as possible in the process foregoing the opportunity to reorder the transactions in a later stage. An aggressive scheduler may have to reject the operations of one or more transactions, causing rollbacks or aborts. A *conservative* scheduler schedules an operation only when it is sure that the operation will not conflict with those of any other transaction. In other words, the scheduler tries to anticipate the future behavior of the transactions in order to prepare itself for operations which have not yet arrived.

Associated with the locking protocols is the problem of deadlock. Deadlocks can be avoided or resolved and a variety of methods have been proposed.

2.3.1. Two-Phase Locking Protocols

In two-phase locking protocols, a transaction passes through two phases. In the first phase it acquires all the locks and in the second phase it releases all the locks previously held. Once the transaction has unlocked any item, it can not acquire any more locks.

2.3.1.1. Basic Two-Phase Locking

In the Basic Two-Phase Locking protocol (2PL), before a transaction accesses a data item, the scheduler looks at the associated lock. If the no transaction holds the associated lock, then the scheduler obtains the lock on behalf of the transaction. If another transaction holds the lock, then the former transaction is made to wait until that transaction releases the lock. We write $wl_i[x]$ (respectively, $rl_i[x]$) to denote the operation of obtaining a write (respectively, read) lock on the data item x and $wu_i[x]$ (respectively, $ru_i[x]$) to denote the operation of releasing the lock.

Two locks $pl_i[x]$ and $ql_j[x]$ are said to conflict if $i \neq j$ and either of p or q is a write.

With these definitions, the rules of two-phase locking are:

- (a) When the scheduler receives a request for operation $p_i[x]$ it checks to see if $pl_i[x]$ conflicts with some $ql_j[x]$ which is already set. If there is a conflict then the scheduler delays T_i . Otherwise, the scheduler grants the lock.
- (b) The lock cannot be released until the operation completes, i.e., $p_i[x] < pu_i[x]$.
- (c) Once the transaction releases a lock it cannot obtain any more locks, i.e., for all x , y and o , $ol_i[x] < ou_i[y]$.

The correctness of the scheduler follows from the fact that whatever a 2PL scheduler allows is serializable.

2.3.1.2. Conservative Two-Phase Locking

Deadlocks appear in 2PL due to its aggressive attitude. In the Conservative Two-Phase Locking protocol (2CPL) we avoid this problem by *predeclaring* the read and the

write sets of the transactions. A transaction then obtains *all* its locks at one time. The scheduler periodically checks to see if there exists a transaction which can be granted all its locks and if so, that transaction is selected for execution. The correctness proof of the protocol follows directly from the proof of correctness of 2PL.

2.3.1.3. Strict Two-Phase Locking

The Strict Two-Phase Locking protocol (2SPL) is the most common variant of the 2PL protocol. It is used to avoid cascading aborts and to ensure recoverability. In 2SPL locks are released only after the transaction is committed. Thus, the protocol is a mixture of the basic 2PL protocol and the two-phase commit protocol.

THEOREM 2.4: *A 2SPL scheduler produces a strict history[†].*

Proof: Consider some history H in which $p_i[x] < q_j[x]$. By basic 2PL we have $pu_i[x] < ql_j[x] < q_j[x]$ and by 2SPL we have $c_i < pu_i[x]$, therefore $c_i < q_j[x]$. That is, H is strict.

□

2.3.1.4. Multigranularity Locking

The *granularity* of a data item refers to its relative size. Granularity of the data items does not affect correctness but may affect performance drastically. Coarse grain locking has lower locking overhead than fine grain locking but permits less parallelism. To achieve the best performance, we would like to allow transactions to obtain locks at the appropriate grain. Long transactions do well with coarser locks but short transactions do well with fine grain locks.

The scheduler, however, should not allow two transactions to obtain conflicting locks on data that overlap. Consider an object-oriented system in which each object may have shared sub-objects. When such a sub-object is locked, the locking information needs to be

[†] A history is strict if it avoids cascading aborts and is recoverable [BHG87].

propagated to the parent object. Moreover, the locking information about the parent object may need to be propagated to the sub-object too. For this, *intention locks* (of both read and write types) are used.

The rules of Multigranularity Locking (MGL) protocol are:

- (a) A transaction may not release an intention lock on an object x if it holds any lock on a sub-object of x .
- (b) To write an object x , T_i must obtain a write lock on some ancestor of x .
- (c) To read an object x , T_i must obtain a read lock or a write lock on some ancestor of x .
- (d) To set a read or intention-read lock on any sub-object, the transaction must obtain an intention-read or intention-write lock on its parent.
- (e) To set a write or intention-write lock on any sub-object, the scheduler must set an intention-write lock on its parents.

To check if two operations conflict or not, the compatibility matrix[†] given in Figure 2.1 is used. The distributed database system R^* uses this mechanism for concurrency control [BHG87]. The protocol is correct because if a transaction owns an explicit lock on a sub-object of the main object, then no other transaction owns a conflicting explicit or implicit lock on the sub-object.

	r	w	ir	iw	riw
r	y	n	y	n	n
w	n	n	n	n	n
ir	y	n	y	y	y
iw	n	n	y	y	n
riw	n	n	y	n	n

Figure 2.1: Compatibility Matrix for Multigranularity Locking

[†]A *compatibility matrix* is defined as a two dimensional matrix where the i^{th} element in the j^{th} row denotes whether the lock requested (row) conflicts with the existing lock(column).

2.3.1.5. Multiversion Two-Phase Locking

In some systems more than one version of the same data item can be maintained to enhance *read-write* concurrency. The idea is to provide that version which preserves serializability. It is shown in [Pap79] that more the number of versions kept, greater the amount of concurrency that can be achieved. To simplify the discussion, assume that only two versions of each data element are kept.

In Two-Version Two-Phase Locking (2V2PL) the scheduler has to maintain three types of locks, namely *read*, *write* and *certify*. The compatibility matrix of the three types of locks is given in Figure 2.2 [BHG87].

When a transaction T_i wants to get a write lock, the scheduler checks to see if a write or certify lock on the same data element is set. If set, the scheduler delays the operation. Otherwise the scheduler creates a new version and stamps it with the identification of T_i . To get a read lock, the scheduler checks to see if there exists a transaction already holding a certify lock. In that case the requesting transaction is delayed. If T_i already owns a write lock on x , the read lock is granted, else it is delayed. When the transaction commits, the locks held by the transaction are changed to certify locks by checking the compatibility of the locks held on the same data items by other transactions. Since only two versions of the data are kept, transactions which are older than the current one holding a certify lock on the same data item is forced to release its certify locks without any checking.

2.3.2. Non-Two-Phase Locking Protocols

Two phase locking protocols suffer from the two problems of deadlock and fewer allowable schedules. Various methods have been used to alleviate the locking protocols from these two problems. Most of the research has converged to the assumption of a priori knowledge about the item [SiK82][SiK83]. For example, the fact that the pages of a B-tree

	<i>Read</i>	<i>Write</i>	<i>Certify</i>
<i>Read</i>	<i>y</i>	<i>y</i>	<i>n</i>
<i>Write</i>	<i>y</i>	<i>n</i>	<i>n</i>
<i>Certify</i>	<i>n</i>	<i>n</i>	<i>n</i>

Figure 2.2: Compatibility Matrix for Multiversion Locking protocol.

are structured hierarchically can be used to achieve more parallelism than that achieved by the two-phase protocols.

Tree Locking (TL) is the simplest of all non-two-phase protocols and is the most widely used. The data items are assumed to be structured in the form of a tree. For simplicity, we assume that the only operations on the data-items (which correspond to the nodes of the tree) are writes. However, it is trivial to extend the protocol, which is presented next, to the case where reads and writes can be dealt with differently. Chapter 5 discusses special cases of such protocols at a greater depth.

Let $p_i[x]$ denote an operation on node x by transaction T_i . The protocol rules are:

- (a) To access x , it must be locked, i.e., $pl_i[x] < p_i[x]$,
- (b) The lock can be set only if not already set,
- (c) If x is not the root, then $pl_i[x]$ can only be set if $pl_i[y]$ is already set where y is a parent of x ,
- (d) No locks can be unlocked before the operation is done, i.e., $p_i[x] < pu_i[x]$, and
- (e) Once the scheduler releases a lock on behalf of T_i , it can no longer get the same lock.

Following these rules, we observe that the locks are obtained from the root in a *root-to-leaf* manner and that the locks in the parent can be released after locking the child. The protocol is analogous to the common optimization of pipelining (Tree Locking) compared to sequential (2PL) access of nodes from the root to the leaves. The protocol is discussed in

further detail in chapter 5.

2.3.3. The Effect of Granularity on Locking

Recall that the granularity of a lock refers to the relative size of the data-item that is being locked by the lock. Extensive research on the effect of the granularity of locking on system performance has been performed by [RiS77]. To discuss the effect of granularity on locks, we need to define some performance metrics. In any database system, the amount of work done in the system increases with the number of users in the system. However, this increase should not be linearly proportional to the number of users in the system. When there is resource contention, queues are formed for the data elements and system time is spent. These queues may be of two kinds, *resource-contention* queues and *data-contention* queues. The latter is due to lock conflicts while the former is due to resources such as CPU, memory and I/O channels. Although data objects are also resources, the two types of resources are distinguished, as the effect of locking is different in the two. Locking can also increase *thrashing* – a phenomenon which results in the system throughput falling abruptly. Thrashing is of two types. The first is called *RC-thrashing* – which is due to contention in the resources and *DC-thrashing* – which is due to data contention in the resources [BSR83].

Three factors contribute to the effect of granularity on locking:

Lock overhead: The finer the granularity, the more the overhead.

Data Contention: Though finer granularity decreases data-contention, it increases the overhead lock maintenance.

Resource Contention: Finer granularity may release many more transactions from the lock queues and they in turn may spend lot of time in the resource queues.

These three factors shape the granularity of the curves shown in the Figure 2.3. The graphs are extracted from [BHG87].

To explain the variation we note that, due to lock overhead and data contention, the curve falls initially. With the granularity of locks becoming finer, the number of locks a

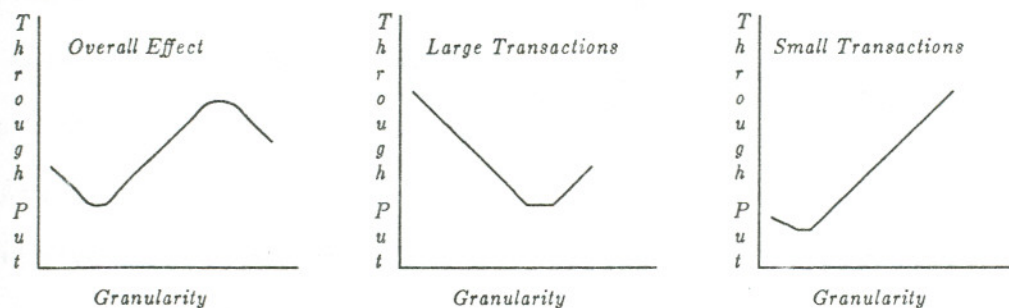


Figure 2.3: The Effect of Granularity on Locking

transaction acquires becomes nearly equal to one per data item accessed and the performance becomes insensitive to the granularity. The increase in throughput occurs due to lesser amount of resource contention. The final drop is due to resource contention, which results from the lock table becoming a hot spot.

2.4. Validation Based Methods

Though locking is by far the most common of the concurrency control protocols, such pessimistic approaches do not perform well in real time systems. Locking has the following major drawbacks [KuR81]:

- (a) The overhead of maintenance of locks.
- (b) The possibility of deadlocks.
- (c) The data contention due to the lock table.

We need locking only when actions conflict, but in practice we incur the overhead even in scheduling non-conflicting transactions.

Deadlock-free locking protocols appear unrealistic in database systems and their performance is questioned. Optimistic concurrency control methods (e.g., validation) view the problem from a different view point. Transactions are allowed to proceed freely. At a later

stage, if it is found that an operation of a transaction has conflicted with an operation of some other transaction, then one of the transactions is made to roll back. In fact, in cases such as *B-Tree management* where conflicts are rare, optimistic schemes perform better than the pessimistic ones. In all optimistic methods, e.g., *serial validation*, *parallel validation* and *certifiers*, a transaction has two or more phases. In the cases of serial validation and parallel validation there are three phases. Namely,

- (a) *Read*: For each transaction a *transaction buffer* is maintained in which the effects of the transaction are reflected. Later this image is used to update the main database.
- (b) *Validation*: After receiving the *end-of-transaction* message the transaction manager checks to see if the transaction had conflicted with some others, in which case appropriate measures are taken.
- (c) *Write*: This is the last phase in which the updates are posted to the main database.

In the validation phase, it must be ensured that the operations of a transaction should be directed to the same database state (as the read phase) to assure serializability. To satisfy this condition, each transaction is provided with an unique *transaction number* (TNC). The TNC can be assigned to a transaction at various points — at the end of the read phase or at the beginning of the read phase — depending upon which, some variations in the general method occur.

To ensure that two transactions are serializable, the following two conditions need to be enforced.

- (a) *No read dependency*: No transaction should read data written by a concurrently progressing transaction.
- (b) *No over-writing*: No transaction should over-write data written by a concurrently progressing transaction.

To meet these criteria various conditions are imposed, of which a transaction has to satisfy one or more. They can be broadly categorized into the following:

- (a) No time overlap at all.
- (b) No time overlap of the write phases.
- (c) No object set overlap of write sets.

(d) No object set overlaps of the read and write sets.

Further details on the protocols can be found in [BHG87].

2.5. Timestamp-Ordering-Based Methods

Unlike the previous mechanisms in which the serialization order is determined during transaction execution to provide maximal concurrency, in timestamp-ordering-based schemes the serialization order is determined a priori and transaction executions are made to obey that order. All transactions are assigned an unique *timestamp* when they start and conflicts are resolved strictly in that order. For a transaction T_i , let $ts(T_i)$ denote its timestamp. The following rule (called the TO Rule) is used to resolve conflicts:

If $p_i[x]$ and $q_j[x]$ are conflicting operations, then the scheduler processes $p_i[x]$ before $q_j[x]$ if and only if $ts(T_i) < ts(T_j)$.

An interesting analogy with time synchronization in distributed systems can be cited here. The difference between timestamp ordering and locking is analogous to the difference between Lamport's clocks and Virtual Time. Lamport's clocks are used to synchronize executions of concurrent events (analogous to TO) in contrast to Virtual Time where concurrent executions are unfolded using virtual time (analogous to locking).

2.6. Reservation Based Methods

Schedulers running this protocol are also known as *Serialization graph testing (SGT)* schedulers. The idea is to maintain a *serialization graph* of the history. When an operation request arrives, before scheduling the operation, the scheduler makes sure that the corresponding edge will not introduce a cycle. The serialization graph referred to here, though, is a bit different because it also contains nodes corresponding to the transactions which are not yet committed unlike the previous case where only the committed projection was considered. For convenience such a serialization graph is called a *Stored Serialization*

Graph (SSG).

The protocols are discussed in detail in [BHG87].

2.7. Semantic Methods in Concurrency Control

Semantic methods of concurrency control exploit the semantic information present in the transactions to exploit more parallelism than those provided by syntactic methods. The use of semantic information may drastically increase the performance of such algorithms due to the complex domain of interpretation of the various operations [Pap79]. A variety of techniques have been reported in the literature that use different types of semantic information. Some of the methods utilize information about the transaction by dividing them into *transaction classes* [Mol83]. Some others utilize the structure of the data. We discuss two of the most common protocols which use some semantic information.

2.7.1. Dynamic Determination Of Compatibility

In [BaR87] compatibility of the various operations are analyzed and transactions are allowed to proceed accordingly. Each object in the system has a manager associated to it, which synchronizes the operations on the data object by concurrent actions based on the compatibility of those operations. *Commutativity* is used as the basis for determining whether an operation can be allowed to execute concurrently with those in progress. Two operations are said to *commute* if the order of their operation does not affect the result, which is characterized by the state of the database after the operations. Such an approach has a threefold advantage. First, the results of executing such an operation hold irrespective of whether the other operations are aborted or committed. Second, the user can provide his own operations and a commutativity table which enhances the extendibility of the system. Third, as a consequence of the first property, along with serializability it is also

guaranteed that there will be no cascaded aborts.

One technique to determine commutativity dynamically is as follows. Each object is logically represented in terms of a *granularity graph*. The nodes of these represent the objects at a particular granularity and the edges represent the *composed-of* (or, the parts-of) relationship. Each operation affects both the vertices and the edges. The set of those vertices and edges which are affected is called the *affected set*. Two operations from different transactions commute if the affected sets are disjoint. The scheduler allows an operation if and only if it is compatible with the uncommitted operations already performed. To see the role of semantics, it is sufficient to note that the same operation may commute with some others while acting on one data set and may not be commutative while on acting on another. The scheduler works according to a standard locking-based scheduler except in that it dynamically determines the commutativity of the operations from the structure of the operand. The method is not deadlock free and deadlocks are resolved as in the case of standard lock-based concurrency control protocols.

2.7.2. Modular Concurrency Control Method

Based on the knowledge of the consistency constraints of the database, the database is partitioned into *atomic data sets*. Such a partition preserves consistency of the database in the sense that the consistency of the database can be preserved by a transaction without reference to the states of the other atomic data sets. The atomic data sets however need not be independent entities. Based on the local transaction semantic information, each transaction is independently decomposed into a partially ordered set of independent transactions. Each one of these sub-transactions must preserve the consistency of the atomic data sets. The postconditions of the parent transaction can also be similarly broken into a set of postconditions (Note that it is possible as the atomic data sets individually preserve

consistency.) which are to be satisfied by the sub-transactions to satisfy the overall postcondition.

Such a partitioning of both the transactions and the data space ensures that, as long as the elementary transactions are executed serializably with respect to each of the accessed atomic data sets, the database consistency constraints will be maintained and the post condition of the overall transaction will be satisfied. Note the post-condition can be the same as the integrity constraints on the data items in the database. Thus, once the database is properly decomposed, there is no need to serialize the entire transaction with respect to the entire database. The synchronization of the individual sub-transactions can be done by any of the standard methods discussed previously.

The following comments can be made from the last section. If 2PL is used to synchronize, then once the entire transaction acquires a lock, the scheduler cannot unlock any of the locks until the transaction commits. Also, since the sub-transactions by definition are smaller than the original one, the level of concurrency is greater in the case when the transactions are nested than in the case when transactions are not nested. The theoretical developments on the topic are complicated and beyond the scope of the thesis.

CHAPTER 3

B-Trees in Database Systems

3.1. Introduction

As mentioned previously, multiway search trees with certain constraints on the contents of the nodes provide a useful tool in indexing large quantities of data. The worst-case behavior of these search trees in their different forms has been studied extensively in [Knu73]. However, not many analytical or performance results for the average case performance are available. Yao [Yao79] introduced the concept of *Fringe Analysis* which is currently used as a tool in analysis of various properties of B-trees. It is complicated and does not appeal to intuition. This chapter deals in the experimental and analytical study of some properties of B-trees used in performance analysis. Unlike most of the citations in the literature, simple probabilistic reasoning will be used in the derivations we present. Some of our analyses follow those in [Wri85].

3.2. Review of B-Trees and Their Variants

A B-tree is a data structure used to index a file of records, each of which is identified by a key. The nodes of the tree that are farthest from the root are often called leaves.

DEFINITION 3.1: A B-tree of order d ($d > 0$) is a multiway tree in which:

- (a) The root has between 2 and $(2d+1)$ descendants.
- (b) Every other node has between $(d+1)$ and $(2d+1)$ descendants, and between d and $2d$ records (i.e., key and other data).
- (c) All leaves are at the same depth from the root and contain only records.

□

For a B-tree containing a maximum of $2k+1$ keys per node is said to be of order $\lceil k+1/2 \rceil$. All leaves are considered to be at *level 0* and the level increases by convention

from the leaves to the root. The height of a B-tree is the maximum path-length from the root to any leaf. A B-tree of order 1 is called a 2-3 tree. Figure 3.1 shows a 2-3 tree.

There are many variations on the B-tree data-structure [MaS80][Com72][Kus73], each having a specific purpose and having its own advantages and disadvantages over others. Next, we review B^+ -trees, B^* -trees, Prefix B^+ -trees, B_{link} -trees and Hysterical B-trees.

3.2.1. B^+ -Trees

To reduce the size of the nodes, the nodes of a B-tree usually contain only the keys of the records and pointers to the records rather than the records directly. More formally, a node of a B-tree is a tuple $\langle p_0, A_0, p_1, A_1, \dots, A_n, p_{n+1} \rangle$ where each A_i is a tuple $\langle K_i, P_i \rangle$. Each p_i is a pointer to another node in the B-tree, the K_i 's are the keys and P_i 's are pointers to the physical location of the data (also called *data record pointers*). The presence of the data record pointers decreases the number of A_i tuples that can be accommodated in a node (whose size is usually bounded by the disk page size) and consequently, increases the number of disk accesses needed to perform an indexed look-up. Moreover, when more information is also kept along with these pointers (e.g., buffer information in Volcano [Gra89]) the situation worsens.

This problem is circumvented in B^+ -trees, where these data record pointers are contained in only the leaf nodes. The internal nodes contain only keys and pointers to other nodes in the tree, acting merely as guides for a search to the appropriate leaf. Thus, the A_i 's in the nodes of a B^+ -tree are composed only of K_i 's. In B-trees, the number of disk accesses is dependent on the size of the data record pointers. In B^+ -trees, the pointer is not there and so accesses are faster. However, B^+ -trees may waste space over B-trees due to key replication in the interior nodes. In some implementations, the leaf nodes in the B^+ -trees are linked to the right siblings to facilitate sequential scanning.

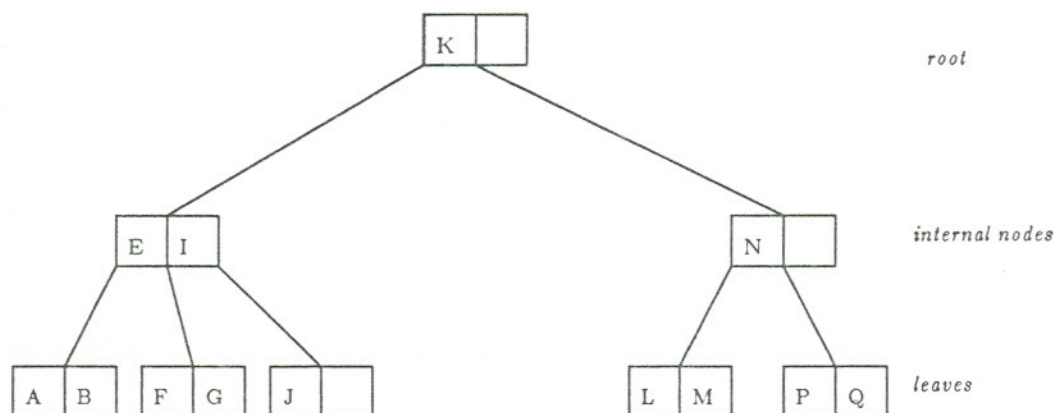


Figure 3.1: Example of a 2-3-tree

3.2.2. B^{*}-trees

During an insertion in a B-tree, if the targeted leaf overflows, the leaf is split into two. Half of the keys and data records pointers are moved from the original leaf to the newly formed leaf; the parent's pointers to the leaves changed appropriately and the change propagated bottom-up towards the root.

The underlying assumption is that the keys are uniformly distributed over the entire domain of key values and it is equally likely that a key will be inserted in any leaf node. But that is not always the case. In some data sets, the key values tend to cluster around a mean value and, in the process, some paths are prone to more splits.

This phenomenon is taken care of in B^{*}-trees [Knu73]. When a key is inserted in an already full node, instead of splitting the node, the right sibling is examined for space. If space is available, the keys are distributed equally among the two nodes and the pointers pointing to the children are shifted appropriately. If the right sibling cannot accommodate any more keys, the two nodes (the full one in which insertion is attempted and the right

sibling) are split into three nodes and the keys redistributed appropriately. Figure 3.2 illustrates key redistribution in B^* -trees.

B^* -trees have larger overheads of insertion and deletion but provide better storage utilization than the B^+ -trees [Knu71a].

3.2.3. Prefix B^+ -trees

In Prefix B^+ -trees, prefixes of the keys contained in the leaf nodes are stored in the internal nodes [Bay77]. Storing prefixes (which are also called separators) instead of full keys substantially saves space in the internal nodes and allows accommodation of more prefixes in the internal nodes. Increasing the order of the nodes potentially decreases the access times and enhances concurrency.

Another problem often faced in B^+ -trees is that of key duplication. As a key in a B^+ -tree may be duplicated in one or more of the internal nodes, a deletion of the key from a leaf does not guarantee the deletion of the key from the internal nodes. In the process of deletion special care must be taken on a future deletion operation of that key. The problem does not arise in B -trees as the keys are not duplicated there. In Prefix B^+ -trees the problem is reduced by not storing entire keys in the internal nodes. If prefixes of keys are taken only from the leaves, then the tree is called a *simple prefix B^+ -tree*. Figure 3.3 is an example of a Prefix B^+ -tree.

$$1 + \boxed{2d} + \boxed{2d} \longrightarrow \boxed{4d/3} + \boxed{4d/3} + \boxed{4d/3 + 1}$$

Figure 3.2: Node splitting in B^* -Trees

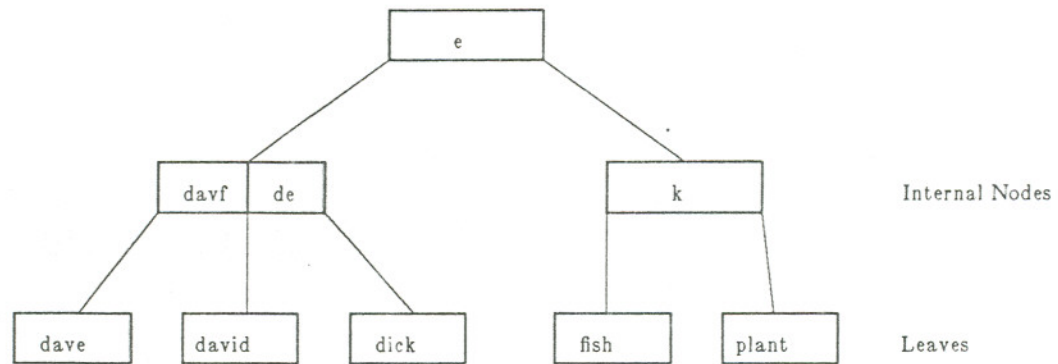


Figure 3.3: Example of Prefix B^+ -tree

3.2.4. B_{link} -Trees

B_{link} -trees [Yao79] are obtained from B^+ -trees by adding an additional key-pointer pair to each node. The extra key is called the *high value* of the node and equals the value of the largest key in the subtree rooted at that node. The extra pointer points to the node's right sibling. The rightmost node at any level has a high value of $+\infty$ and the corresponding pointer is nil. The B_{link} -tree data structure facilitates the traversal of nodes at the same level in key order, which is often required in sequential searches and partial match retrieval queries. Figure 3.4 illustrates the structure of a B_{link} -tree.

3.2.5. Hysterical B-Trees

Maier and Salveter [MaS80] observed that concurrent operations are drastically affected by restructuring the tree during insertions or by the merging of nodes during deletions (discussed later in Section 3.6.2.1). Moreover, m sequences of an insert followed by a delete of the same key into a nearly full B-tree of order d containing n nodes requires $O(m \log_d n)$ operations. To reduce the log factor from the complexity, they introduced the

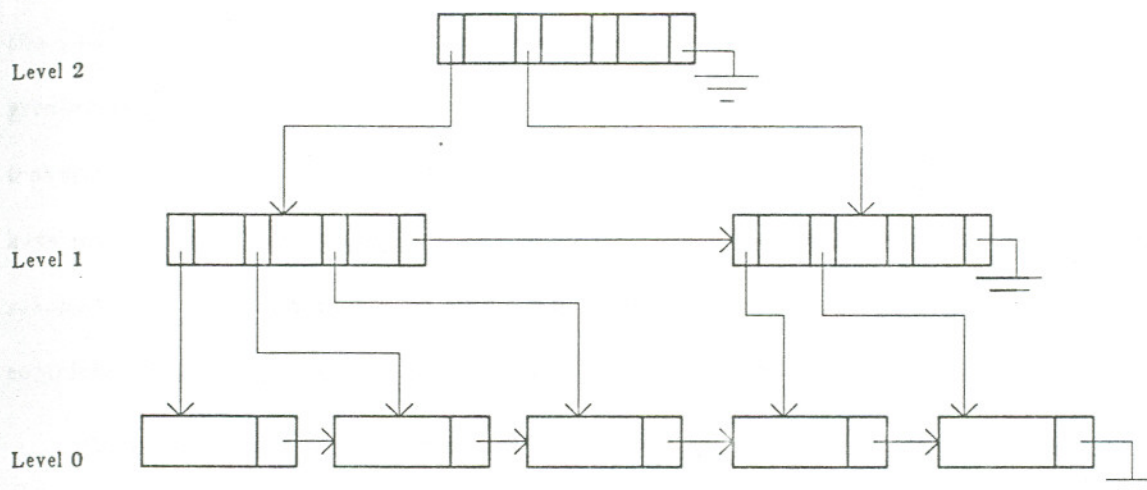


Figure 3.4: Structure of B_{link} -trees

concept of a *slack factor*, p .

During deletion, a node needs to be merged only when its size falls below $d-p$, instead of d . With appropriate choice of p , the log factor was shown to be eliminated. However, this change reduces the average storage utilization of the nodes. The effect of this slack factor on storage utilization and other performance metrics is analyzed in the later sections.

3.3. Operations on B-Trees

There are three main operations on a B-tree: *insert*, *delete* and *search* of a particular key value in the tree. The first two cause permanent changes in the structure of the B^+ -tree and are called *updaters*. Concurrent operations of insert and delete are thus not permissible on the same node. Searching for a key value causes no modification to the tree and may proceed at any node concurrently with other searches.

To search for a key in a B-tree, starting from the root, the path to the appropriate leaf is traversed by determining at each node which child to search next. For any node n , the $i+1^{\text{th}}$ child n_i of n is traversed if the search key value v is less than that of K_i and greater than that of K_{i-1} . If v is less than the key value K_0 then the 1^{st} child n_0 is traversed. If v is more than K_j then the child n_{j+1} is traversed, where $j+1$ is the number of keys in the node. A matching key may be found on the path from the root before a leaf is reached. In a B-tree, data record pointers are stored in the internal nodes, so the search is complete. In a B^+ -tree, the search must continue to the leaf.

To insert an element in a B^+ -tree, a search to the appropriate leaf is performed first. If the leaf can accommodate more keys (and the associated data record pointers), then the key is inserted there. Otherwise, the leaf is split into two and the keys in the original node distributed equally among them. This change in the leaf structure is reflected by the insertion of another key in the parent. If the parent cannot accommodate this extra key, the parent is also split. The structural change propagates bottom-up from leaf to root recursively.

To delete a key, a search is first performed to the node containing the key and the data record pointer (which must be a leaf in the case of a B^+ -tree) from which the key is physically deleted, if found. If this results in the node having fewer than d keys, then the leaf is merged with either sibling, the keys are redistributed and a similar structural change is propagated recursively upwards.

3.4. Deferred versus Aggressive Splitting

During insertion of keys in a B-tree, the structural change required to accommodate the extra key is propagated bottom-up from the leaf to root. A node is split if and only if it can accommodate no more keys. Guibas and Sedgewick [GuS78] proposed a scheme in

which, while traveling from the root to the leaf during the search phase of an insertion, full nodes (i.e., potential victims of future splits) are split preemptively when first encountered along the search path. Thus, splitting of nodes take place in a top-down fashion. We call this scheme *aggressive splitting*.

Several researchers claim that aggressive splitting does not perform well in situations where concurrent operations are allowed [LeY81]. The argument is that more nodes will be split than necessary and hence larger sub-trees need to be locked. If the transactions are of finite length, then a node that may not need to be split in the bottom-up case may need to be split in the aggressive case. No analytical or simulated performance results comparing aggressive and deferred splitting have been reported previously.

3.5. Fringe Analysis

To study the performance of B^+ -trees under probabilistic models of operation occurrence, certain analytical models have been reported in the literature; namely *Fringe Analysis* [Yao79][Bae86] and the *Generalized Overflow Technique* [Kus73].

A *tree-type* of height h is a B-tree of height h in which key values are ignored. Two trees that are structurally isomorphic, but have different keys in the nodes, are of identical tree-types. A *tree collection* is the finite collection C of all tree-types $\{ T_1, \dots, T_n \}$ of some fixed height h . Thus, considering only B-trees of a given order d , the set C is unique for each h . A collection C is closed if the type of a new tree formed by insertion of a key in any of the element trees of C , is also a member of C . In case an insertion results in the split of the root of a tree, only the fringe of the resulting tree is considered. Note that, closure is possible because the key values are ignored.

The *fringe* of a tree T is a disjoint set of one or more of its subtrees that are isomorphic to elements in some tree collection C . Thus, a tree may have more than one fringe

depending upon the collection C .

In Fringe Analysis, the composition of the fringe of a B^+ -tree is expressed in terms of such tree collections. The expected number of the different tree-types in the fringe of the tree being analyzed are estimated from the number of keys present in the tree. These expected numbers form a *Markov Chain* because an insertion into the B-tree changes the average number of trees in the fringe of at most two tree-types. Moreover, the change is closed – that is, the the fringe of a new tree formed by an update operation, does not contain any tree-type outside the collection C .

The order of the analysis is given by the height h of the trees in the collection. The greater the height, the more accurate the analysis, but the size of the collection grows exponentially with increase in height. Such analysis is intuitive in the case of 2-3 trees where there are only two tree-types of height 1 and the collection size for $h = 1$ is thus 2. In the case of B-trees of larger orders, the collection size can become unmanageable and hence appeals less to intuition.

An extensive mathematical treatment of fringe analysis can be found in [EZG82]. Some of their results regarding matrices are used below to calculate the performance metrics analytically.

3.6. Performance Metrics of B-Trees

Concurrency control in B-trees is different from concurrency control in general, because, in general concurrency control methods, the existence of a central scheduler synchronizes the processes. In the case of B-trees, the processes are independent and there is no scheduler process. Although in some commercial database systems B-tree nodes are treated as regular database objects for concurrency control and recovery, the practice is not standard. Thus, concurrency control in B-trees is asynchronous. Moreover, the structure of the

data may make increased concurrency possible. However, certain properties of B-trees are critical to database system performance and cannot be sacrificed to accommodate more concurrency. Thus, every concurrency control algorithm should try to preserve these properties. This section is dedicated to the study of several such properties that have a close relationship with the general performance of B-trees and the performance of concurrency control protocols for B-trees.

We begin with a probabilistic analysis of B-tree performance looking at storage utilization and run-time costs of operations. To simulate the validity of some of the derived expressions, several simulations were performed by inserting randomly chosen keys, into initially empty B-trees of order 25. Up to 50,000 keys were inserted into the trees. The individual simulations are discussed in more detail later when their results are presented.

Figure 3.5 lists the parameters used in our analysis and, unless otherwise stated, all trees are B-trees.

3.6.1. The Average Storage Utilization

DEFINITION 3.2: The *average storage utilization* u of a B-tree is the fraction of the total storage space of the tree that stores relevant data. More precisely,

$$u = \frac{r}{2nd} \quad (3.1)$$

□

For a fixed number of keys in the tree the expected average storage utilization is given by:

$$E[u] = \frac{r}{2d} E\left[\frac{1}{n}\right]$$

<i>Parameter</i>	<i>Meaning</i>
d	Order of the B-tree.
r	The number of keys in the tree.
n	The number of nodes in the tree.
u	The average storage utilization.
u_P	The average storage utilization with partial expansions.
f	The minimum fullness factor.
p	The slack factor of a node.
ρ	The number of records per page.
q	The probability of split of a node.
q_P	The probability of split of a node with partial expansion.
q_a	The probability of split of a node in aggressive case.
q_d	The probability of a merge at a node.
$A_{i,r}$	The expected number of nodes with size $d-p+i$ in a tree with r keys.
S	The average number of splits on an insert.
S_P	The average number of splits with partial expansions.
S_{P_L}	The average number of splits with partial expansions at leaves.
M	The average number of merges on a delete.
$C_{I/O}$	Cost of Unit I/O

Figure 3.5: Parameter Notation

In several concurrency control methods (e.g., optimistic methods), an increase in concurrency can be achieved at the cost of reduced average storage utilization. But a good dynamic data structure should have a high average storage utilization. To avoid comparing apples with oranges, it is important to determine the storage utilization resulting from the adoption of a particular concurrency control technique.

DEFINITION 3.3: The *minimum fullness factor* f is defined as the fraction of the total capacity of the nodes that, by definition, will always contain keys.

□

If, for example, every node must contain between 10 and 50 keys, the minimum fullness factor is 20%. The minimum fullness factor determines the worst-case storage

utilization of the tree excluding the root node.

The number of nodes in a B-tree of order d containing r keys (where d and r are constants), lies in the closed interval $[\frac{r}{2d}, \frac{r}{2df}]$. For large r , the effect of the root node, which may contain fewer than $2df$ keys, is negligible. Let $P(x)$ be any *probability density* function defined over that interval giving the probabilities of different n . Then, assuming a discrete distribution, we have:

$$E[\frac{1}{n}] = \sum_{\frac{r}{2d} \leq i \leq \frac{r}{2df}} \frac{P(n=i)}{i} \quad (3.2)$$

To calculate the probability, several assumptions are necessary. The simplest assumption is that the keys are uniformly and continuously distributed in the interval [Leu84]. This assumption replaces the summation with an integral and the expectation can be obtained as a function of f only. However this assumption is not always reasonable since keys are often distributed close to a mean value and the probability is discontinuous. The problem then reduces to the *restricted cell occupancy problem* where one has to place $r-2dfn$ keys (since each node must already contain $2df$ keys) in n cells (nodes) with the restriction that not more than $2d-2df = 2d(1-f)$ keys can be put in one cell.

A solution to this problem can be found in [Yao79](pp. 160) and the probability given is:

$$P(x=i) = \frac{\sum_{0 \leq k \leq i} (-1)^k \binom{i}{k} \binom{r-2dfi-rk+2dfk+n-k-1}{r-2dfi-rk-2dfk-k}}{\sum_{\frac{r}{2d} \leq l \leq \frac{r}{2df}} \left\{ \sum_{0 \leq k \leq l} (-1)^k \binom{l}{k} \binom{r-2dfk-rk+2dfk+n-k-1}{r-2dfk-rk-2dfk-k} \right\}} \quad (3.3)$$

The expected value of $1/n$ can be obtained from equation (3.2) and (3.3). However, the problem with this approach is that the right hand side of expression (3.3) cannot be expressed in closed form. Fringe Analysis techniques are employed for that reason.

Figure 3.6 illustrates the variation of the storage utilization with the minimum fullness factor f and a comparison between the simulated and the expected results. Each run consisted of inserting 50,000 keys (uniformly distributed) into an initially empty tree of order 50 and then measuring the average storage utilization. The number 50,000 was chosen because the storage utilization is fairly constant by that point. Keys were then deleted from the tree, in the order inserted and the tree was rebuilt using a different value of f . The simulation was repeated for different values of the minimum fullness factor between 0.1 to 0.9 in steps of 0.02. For each value of f , the simulation was repeated three times and the median value was plotted.

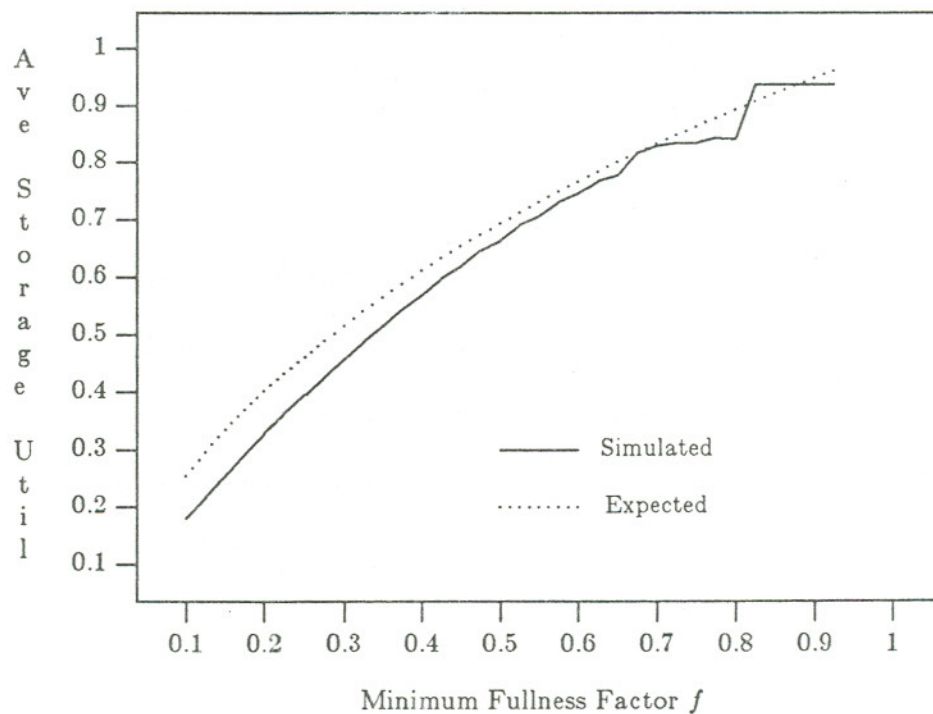


Figure 3.6: Effect of Minimum Fullness Factor on Storage Utilization

DEFINITION 3.4: The *slack factor* p of a node is defined as the number of keys less than the order of the node that are allowed before the node is merged with a sibling.

□

The slack factor and the minimum fullness factor are related as:

$$f = (d-p)/2d.$$

Increasing the slack factor increases the degree of concurrency but decreases storage utilization. Figure 3.7 illustrates the variation of the simulated average storage utilization with the slack factor. Here each run consisted of inserting 50,000 keys into an initially empty tree of order 10. Then, 30,000 keys were deleted from the tree, measuring the average storage utilization after each deletion and, in the end, averaging to get the final value. All keys were subsequently deleted from the tree and the experiment repeated for different values of the slack factor.

From both the graphs we conclude that increase in slack factor appreciably decreases the average storage utilization.

3.6.2. The Expected Number of Splits per Insertion

The greater the number of splits during an insertion and the greater the number of merges during a deletion, the lesser the amount of concurrency that can be achieved. Thus, a technique to achieve more concurrency should try to reduce the number of splits during insertions and the number of merges during deletion. In the following subsections we examine the number of splits that occur both when splitting is deferred and when splitting is done aggressively.

3.6.2.1. Deferred Splitting

We first consider the case when splitting is deferred. To calculate the expected number of splits due to the insertion of a key, it is necessary to calculate the probability of

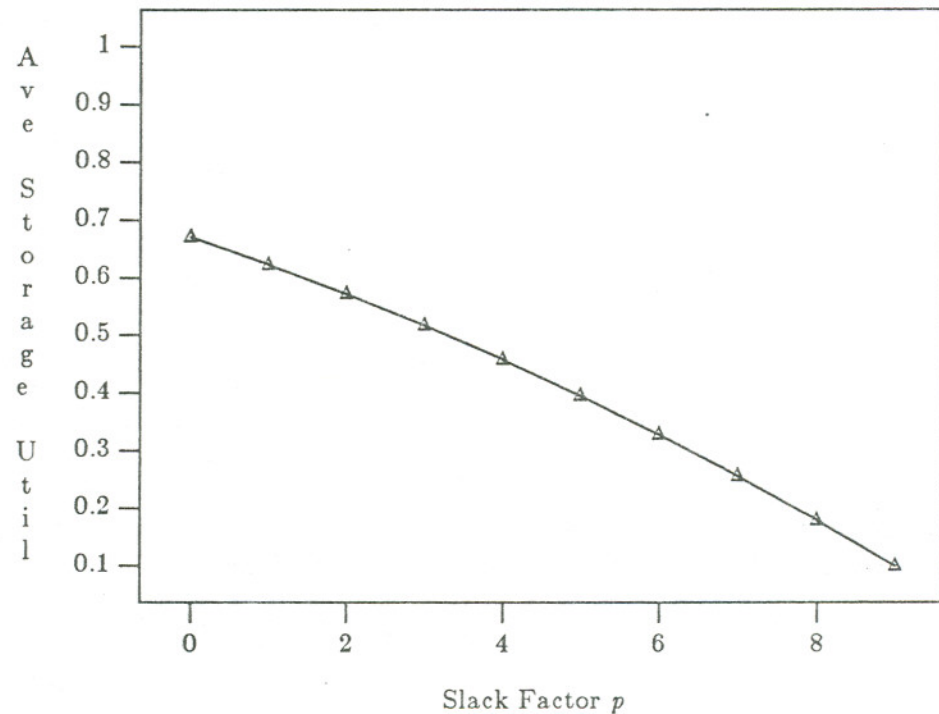


Figure 3.7: Effect of Slack Factor on Storage Utilization

a split of a particular node due to the insertion. Consider a situation when only insertions are done to a tree which contain nodes of size $d-p$ to $2d$. Let $A_{i,r}$ denote the expected number of leaves with size $(d-p+i)$ in a tree containing r keys. As a result of the insertion, the number of keys in the tree will increase from r to $r+1$. Moreover, assuming an uniform distribution, the key can be inserted in any of the existing $r+1$ intervals between between the existing keys. Thus, the probability that the insertion will be in a node already containing $d-p+i$ keys can be given as the product of the number of intervals in a node with $d-p+i$ keys and the expected number of such nodes, divided by the total number of possible intervals. That is

$$\frac{(d-p+i+1)A_{i,r}}{r+1}$$

For $i = 0, \dots, d+p-1$ the insertion will not cause a split in the node. But for $i = d+p$ the insertion will cause a decrease in the number of leaf nodes with size $2d$ by one and an increase in the number of leaf nodes of size d because the full node is split to form two half-filled nodes. Thus, the $A_{i,r}$'s form a *Markov Chain*.

In the case when there is a split on the insertion, the increase in the expected number of nodes of size d is given by:

$$A_{p,r+1} - A_{p,r}$$

This increase is equal to:

$$\begin{aligned} & 2(\text{probability that the insertion is done into a node of size } 2d) \\ & - 1(\text{probability that the insertion is done into a node of size } d) \\ & + 1(\text{probability that the insertion is done into a node of size } d-1) \end{aligned}$$

In the case when there is no split on the insertion, the increase in the expected number of nodes of size $d-p+i$ is given by:

$$A_{i,r+1} - A_{i,r}$$

This increase is equal to:

$$\begin{aligned} & 1(\text{probability that the insertion is done into a node of size } d-p+i-1) \\ & - 1(\text{probability that the insertion is done into a node of size } d-p+i) \end{aligned}$$

Thus, we get the following two recurrences:

$$\begin{aligned} A_{p,r+1} &= A_{p,r} + \frac{1}{r+1} (2A_{d+p,r}(2d+1) - A_{p,r}(d+1) + A_{p-1,r}(d)) \\ A_{i,r+1} &= A_{i,r} + \frac{1}{r+1} (A_{i-1,r}(d-p+i) - A_{i,r}(d-p+i+1)), \quad i = 1, 2, \dots, p-1, p+1, \dots, d+p \end{aligned}$$

Thus, if the column vector \mathbf{A} is expressed as:

$$\mathbf{A}(r) = [A_{0,r}, A_{1,r}, \dots, A_{d+p,r}]$$

we have the recurrences reducing to:

$$\mathbf{A}(r+1) = \left(I + \frac{D}{r+1}\right)\mathbf{A}(r) \quad (3.4)$$

where I is the identity matrix and D is a $(d+p+1)$ by $(d+p+1)$ matrix obtained from the coefficients of the recurrence relations.

$$D = \begin{bmatrix} -(d-p+1) & 0 & \cdot & \cdot & \cdot & \cdot & 0 \\ (d-p+1) & -(d-p+2) & \cdot & \cdot & \cdot & \cdot & 0 \\ 0 & (d-p+2) & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & d & -(d+1) & \cdot & 2(2d+1) \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\ 0 & 0 & \cdot & \cdot & \cdot & \cdot & 2d+1 \end{bmatrix} \quad (3.5)$$

Solutions to similar recurrences can be found in [Knu73] (pp. 680). Let

$$a_i = \lim_{r \rightarrow \infty} \frac{A_{i,r}}{r+1}, \quad i = 1, \dots, d+p$$

As in [Knu73], if \mathbf{a} is the column vector:

$$\mathbf{a} = [a_0, a_1, a_2, \dots, a_{d+p}]$$

then, it can be shown that (e.g., [Knu73], pp. 680) the roots of the characteristic polynomial of the matrix $D-I$ have negative real parts and that there exists a column vector \mathbf{a} such that:

$$(D - I)\mathbf{a} = 0$$

From which we get the following three recurrences:

$$\begin{aligned} a_i &= 0 && \text{for, } 0 \leq i < p \\ -(d+2)a_p + 2(2d+1)a_{d+p} &= 0 \\ -(d-p+i+2)a_i + (d-p+i)a_{i-1} &= 0 && \text{for, } p+1 < i \leq d+p \end{aligned}$$

Simplifying we get:

$$\begin{aligned} a_p &= \frac{2(2d+1)}{(d+2)} a_{d+p} \\ a_{i-1} &= \frac{(d-p+i+2)}{(d-p+i)} a_i && \text{for, } p+1 < i \leq d+p \end{aligned}$$

Substituting $p+j$ for i we have:

$$a_{p+j-1} = \frac{d+j+2}{d+j} a_{p+j} \quad \text{for, } 1 < j \leq d$$

Expressing the a_i 's in terms of a_{d+p} , we have:

$$a_{d+p-j} = \frac{(2d+1)(2d+2)}{(2d-j+1)(2d-j+2)} a_{d+p} \quad \text{for, } 0 \leq j \leq d \quad (3.6)$$

Summing the probabilities of an insertion in all possible node configurations to 1, we have:

$$\sum_{0 \leq i \leq d+p} (d-p+i+1) A_{i,r} = r+1$$

Dividing both sides by $r+1$, we have:

$$\sum_{0 \leq i \leq d+p} (d-p+i+1) \frac{A_{i,r}}{r+1} = 1$$

Taking limits ($r \rightarrow \infty$) we get:

$$\sum_{1 \leq i \leq d+p} (d-p+i+1) a_i = 1$$

Replacing i by $d+p-j$ and replacing the expression for a_{d+p-j} from equation (3.6), we get:

$$\sum_{0 \leq j \leq d} \frac{(2d+1)(2d+2)}{(2d-j+2)} a_{d+p} = 1$$

Finally by transposing:

$$a_{d+p} = \frac{1}{(2d+1)(2d+2) \sum_{0 \leq j \leq d} \frac{1}{2d-j+2}}$$

Denoting the Harmonic Numbers by $H(n)$ we have:

$$\sum_{0 \leq j \leq d} \frac{1}{2d-j+2} = H(2d+2) - H(d+1)$$

Thus we have:

$$a_{d+p} = \frac{1}{(2d+1)(2d+2)(H(2d+2) - H(d+1))}$$

Replacing the expression for a_{d+p-j} we have:

$$a_{d+p-j} = \frac{1}{(2d-j+1)(2d-j+2)(H(2d+2)-H(d+1))} \quad (3.7)$$

The probability of a split of a node is the same as the probability of an insertion into a node that is already full, i.e., the probability that the node contains $2d$ keys already.

Hence, the probability of split of leaf on an insertion q is given by,

$$\begin{aligned} q &= \frac{(2d+1)A_{d+p,r}}{r+1} \\ &= (2d+1)a_{d+p} \quad \text{in the limit } r \rightarrow \infty \\ &= \frac{1}{(2d+2)(H(2d+2)-H(d+1))} \quad (3.8) \end{aligned}$$

Assuming that the splitting of a node is independent of the splitting of its child, the probability of exactly j splits along a path is given by :

$$q^j(1-q)$$

since, along the path there will be exactly j consecutive splits and the following node will not be split. So the expected number of splits per insertion is given by:

$$\begin{aligned} E[S] &= \sum_{0 \leq j \leq \infty} jq^j(1-q) \\ &= \frac{q}{1-q} \end{aligned}$$

Note that the slack factor vanishes from the final expression of the probability of split of a node. This is true intuitively also. The slack factor becomes effective only when there are deletions. Since we are considering insertions only, therefore as more and more keys are inserted into the tree, the nodes which contain $d-p$ to d keys becomes fuller and fuller and in the asymptotic case when the number of nodes keys is infinite, there are no nodes which contains between $d-p$ and d keys. Moreover, the inverse relationship between the number of keys per node and the expected number of splits justifies the intuition that a increase in

node size will reduce the number of splits.

3.8.2.2. Aggressive Splitting

In the case of aggressive splitting, the probability of splitting any node is independent of which leaf the key is inserted into, because at the time of insertion in the leaf it is guaranteed that the insertion will not cause any split to propagate bottom up.

This unconditional probability of splitting q' of any node is given by:

$$q' = \frac{1}{d+p}$$

because a node can contain from $d-p$ to $2d$ keys. However, the statement is true in the transient case, that is, when the tree is not very large. Following the explanation in the case of deferred splitting, that there are no deletions involved, in the asymptotic case, there will be no node containing between $d-p$ and d keys. The unconditional probability of split of a node in the asymptotic case is then given by:

$$q' = \frac{1}{d}$$

The expected number of splits in this case can be similarly calculated to be,

$$\begin{aligned} E[S_A] &= \sum_{0 \leq j \leq \infty} j q'^j (1-q') \\ &= \frac{q'}{1-q'} \end{aligned}$$

Note that for constant p and d , the expected number of splits in the case of aggressive splitting is a strict upper bound for that of deferred case. Theorem 3.1 establishes the relationship between the number of splits in the case of aggressive and deferred splitting.

The probability of merging during deletions can be calculated accordingly. However, the exact analysis in the case of deletions, depend on the routing policy, that is, the condition under which re-distribution is done. The analysis is complicated and is omitted due to

that reason.

LEMMA 3.1: For all integral x , $H(2x) - H(x) > \ln 2$.

Proof: By induction on x .

□

THEOREM 3.1: *The expected number of splits on an insert is strictly more for aggressive splitting than for deferred splitting in the asymptotic case ($r \rightarrow \infty$).*

Proof: By simple algebra we have:

$$\frac{d}{2(d+1)} < \frac{1}{2} < \ln 2 = \ln \frac{2(d+1)}{d+1}$$

That is,

$$d < 2(d+1)(\ln(2d+2) - \ln(d+1))$$

By Lemma 3.1, we get:

$$d < 2(d+1)(H(2d+2) - H(d+1))$$

that is,

$$\frac{1}{d} > \frac{1}{(2d+2)(H(2d+2) - H(d+1))}$$

and finally,

$$q' > q$$

from which by componendo and dividendo, we get:

$$\frac{q'}{1-q'} > \frac{q}{1-q}$$

that is,

$$E[S_A] > E[S]$$

□

3.6.3. The Average Cost of an Operation

In this section we discuss briefly the cost of the various operations on B^+ -trees. By the cost of an operation, we mean the time overhead involved to execute the operation.

3.6.3.1. Search for a Key

The average cost of a search, C_{search} , is equal to the height of the tree times the cost of unit I/O. When the number of nodes in the tree is large and the average storage utilization has stabilized, each node in the tree except the root, contains on the average $2dE[u]$

keys. Thus if the root contains d keys on the average, (which is justified by the fact that it can contain between 1 and $2d$ keys and the key distribution is uniform) then the average height of the tree is given by:

$$\left(1 + \log_{2dE[u]+1} \frac{r}{d+1}\right)$$

where the average is taken over the lifetime of the tree *after* the average storage utilization stabilizes. Thus, assuming that a node is the unit of I/O and that the cost of unit I/O is $C_{I/O}$, the average cost of a search operation is:

$$C_{search} = \left(1 + \log_{2dE[u]+1} \frac{r}{d+1}\right) C_{I/O}$$

The average cost of a search for any key in a particular tree is the same since all leaves are at the same level.

3.6.3.2. Insertion of a Key

The average cost of insertion includes the cost of a full search for the key, which we assume is not in the tree. If the insertion does not cause a split in the leaf, then the cost is that of an extra I/O to write back the updated page. If there is a split, the cost also includes three extra writes. Thus, the total cost equals:

$$C_{insert} = C_{search} + ((1-q) + q(1+2E[S])) C_{I/O}$$

3.6.3.3. Deletion of a Key

The average cost of deleting a key can similarly be calculated to be the sum of the cost of a search added to that of the cost of a conditional merge.

$$C_{delete} = C_{search} + ((1-q_d) + q_d(1+2E[M])) C_{I/O}$$

3.7. Other Optimizations in B⁺-Trees

This section discusses the possibilities of various other optimizations in the B⁺-tree data structure to improve performance.

3.7.1. Partial Expansions of Nodes

Recall that, while inserting a key in the B⁺-tree, if the node into which the insertion is being done is full, the node is split into two and the split propagated bottom-up. This scheme is particularly useful if the nodes are of the same size as the unit of disk I/O, which is not always the case. If more than one page constitutes a node, then the following scheme is useful. During insertion, when there is an overflow from the node into which the insertion is being done, instead of splitting the node, an extra page is added to the node. This is called a *partial expansion* of the node. After passing through a sequence of one or many of such partial expansions, the node can actually be split (called a *full expansion*). This scheme provides better storage utilization but may need more page accesses during index look-ups.

Next, we derive some analytical results regarding B⁺-trees, when partial expansions are done.

3.7.1.1. Expected Number of Full Expansions Per Insertion

Consider a B⁺-tree of order d containing r keys, where r is a large number. Also, consider that each node of the tree spans multiple pages. If the page size is ρ records, then each node requires $2d/\rho$ pages. Assume that a node containing $2d$ is partially expanded, when full. In each of the partial expansions, assume that the page size increases by 1. Thus, nodes grow following the sequence:

$$\langle 2d + i\rho \rangle \quad \text{for} \quad 0 \leq i \leq k$$

where k (the maximum value of k is bound) such partial expansions occur before the node is

split into two. When the node is split into two, the two nodes formed contain at most $2d$ keys each. To satisfy this criterion we have:

$$2d + k\rho + 1 \leq 4d$$

Simplifying which, we get:

$$k \leq \frac{2d-1}{\rho}$$

The value of k has to be integral. Thus, assuming that the records do not span pages and that $d \gg 1$, we have the upper bound on the number of partial expansions given by:

$$\begin{aligned} k &= \left\lfloor \frac{2d-1}{\rho} \right\rfloor \\ &\simeq \frac{2d}{\rho} - 1 \end{aligned}$$

Thus, at most $\frac{2d}{\rho} - 1$ partial expansions can be given, after which the node has to be given a full expansion. When the split occurs, the keys in this full node containing $2d + \rho(\frac{2d}{\rho} - 1)$ ($= 2m - 1$, say) keys along with the key to be inserted, are distributed equally into each of the two new nodes. Thus, we have the relationship:

$$2m = 4d - \rho + 1$$

For simplicity, assume that m is odd. Then the number of keys in each of the newly formed nodes is given by: $(2m - 1 + 1)/2 = m$. Note that, the partial expansions can be given in terms of anything. That is, it is not mandatory that the partial increments should be in terms of pages. We choose a page to be the unit of expansion as it is realistic and considerably simplifies calculations.

The number of keys in a node can thus vary from m to $2m - 1$. Let $A_{i,r}$ denote (as before in section 3.7.2.1) the expected number of nodes of size $m + i$. Then, assuming a uniform distribution of keys, the probability that an insertion will occur in a node containing

$m+i$ keys is given by:

$$\frac{(m+i+1)A_{i,r}}{r+1}$$

We then get the two *Markov Chain Recurrences* as before:

$$\begin{aligned} A_{0,r+1} &= A_{0,r} + \frac{1}{r+1}(2A_{m-1,r} - A_{0,r}(m+1)) \\ A_{i,r+1} &= A_{i,r} + \frac{1}{r+1}(A_{i-1,r}(m+i) - A_{i,r}(m+i+1)) \quad i = 1, 2, \dots, m-1 \end{aligned}$$

The coefficient matrix of the system of recurrences, D , is given by:

$$D = \begin{bmatrix} -(m+1) & 0 & \cdot & \cdot & 2(2m) \\ (m+1) & -(m+2) & \cdot & \cdot & 0 \\ 0 & (m+2) & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot & 2m \end{bmatrix} \quad (3.9)$$

As in section 3.6.2.1, it can be shown that the characteristic polynomial of $D-I$ have negative real roots and therefore, there exists a column vector \mathbf{a} such that $(D-I)\mathbf{a} = 0$. Let \mathbf{a} be the column vector:

$$\mathbf{a} = [a_0, \dots, a_{m-1}]$$

Then,

$$(D - I)\mathbf{a} = 0$$

Solving the equation, we have:

$$a_0 = \frac{2(2m)}{(m+2)} a_{m-1}$$

and,

$$a_{j-1} = \frac{m+j+2}{m+j} a_j \quad \text{for } 1 \leq j \leq m-1$$

Simplifying them we have:

$$a_{m-2} = \frac{2m+1}{2m} a_{2m-1}$$

and,

$$a_{m-j} = \frac{(2m+1)(2m)}{(2m-j+1)(2m-j+2)} a_{m-1}$$

Replacing $m-j$ by i we have:

$$a_i = \frac{(2m)(2m+1)}{(m+i+1)(m+i+2)} a_{m-1} \quad (3.10)$$

Summing the probabilities of an insertion in all possible node configurations to 1, we have:

$$\sum_{0 \leq i \leq m-1} (m+i+1)A_{i,r} = r+1$$

Dividing both sides by $r+1$:

$$\sum_{0 \leq i \leq m-1} (m+i+1) \frac{A_{i,r}}{r+1} = 1$$

Taking limits ($r \rightarrow \infty$), we get:

$$\sum_{0 \leq i \leq m-1} (m+i+1)a_i = 1$$

Substituting the expression for a_i from equation (3.10), we get:

$$\sum_{0 \leq i \leq m-1} \frac{(2m+1)(2m)}{(m+i+2)} a_{m-i} = 1$$

Simplifying, we have:

$$(2m+1)(2m)a_{m-1} \sum_{0 \leq i \leq m-1} \frac{1}{m+i+2} = 1$$

Thus, by transposing:

$$a_{m-1} = \frac{1}{(2m)(2m+1) \sum_{0 \leq i \leq m-1} \frac{1}{m+i+2}}$$

Expressing the Harmonic numbers by $H(n)$, we have:

$$\sum_{i=0}^{m-1} \frac{1}{m+i+2} = H(2m+1) - H(m+1)$$

Hence, we have:

$$a_{m-1} = \frac{1}{(2m+1)(2m)(H(2m+1) - H(m+1))}$$

Replacing a_{m-1} in the expression of a_i in equation (3.10), we have:

$$a_i = \frac{1}{(m+i+1)(m+i+2)(H(2m+1) - H(m+1))}$$

Finally replacing the expression for m , we get:

$$a_i = \frac{1}{(2d - 0.5\rho + i + 1.5)(2d - 0.5\rho + i + 2.5)(H(4d - \rho + 2) - H(2d - 0.5\rho + 1.5))} \quad (3.11)$$

In this case also, the probability of a full expansion of a leaf is the same as the probability of an insertion into a leaf that is already maximally full, i.e., the probability that the leaf contains $2m-1$ keys already. Hence, the probability of split of leaf q_P on an insertion is given by,

$$\begin{aligned} q_P &= \frac{(2m)A_{m-1,r}}{r+1} \\ &= (2m)a_{m-1} \quad \text{in the limit } r \rightarrow \infty \\ &= \frac{1}{(2m+1)(H(2m+1) - H(m+1))} \end{aligned}$$

The probability of splitting of any node in the tree is approximately the same as that of the leaf, so the expected number of splits per insertion is given by:

$$\begin{aligned} E[S_P] &= \sum_{0 \leq j \leq \infty} j q_P^j (1 - q_P) \\ &= \frac{q_P}{1 - q_P} \end{aligned}$$

Note that, since $m \gg d$, the probability of a full expansion is much smaller compared to the case when partial expansions are not given.

In the case when partial expansions are given only at the leaf level, the expected number of splits per insertion is given by:

$$\begin{aligned} E[S_{P_L}] &= q_P \sum_{0 \leq j \leq \infty} j q^{j-1} (1-q) \\ &= \frac{q_P}{1-q} \end{aligned}$$

Note that, since $q_P \ll q$, the expected number of full splits is much less than in the case when no partial expansions are given. However, there are more page accesses because a node spans multiple pages.

3.7.1.2. Average Storage Utilization

The method to calculate the average storage utilization as presented in section 3.6.1, following the idea of [Leu84], is too simple to handle the case of partial expansions. Moreover, the trick employed in the analysis provided by [Leu84], was to replace a discrete summation by continuous summations. The technique worked because the intervals were small, but in this case, the intervals are large and the approximation cannot be done.

Consider the same B^+ -tree configuration of the previous section. The average storage utilization of the tree (u_P) is given by the ratio of the total number of keys in the tree and the number of nodes.

$$\begin{aligned} E[u_P] &= \frac{\frac{1}{2d} \sum_{0 \leq j \leq 2d-m} A_{j,r}(m+j+1) + \sum_{0 \leq i \leq k-1} \frac{1}{2d+i\rho} \sum_{0 \leq j \leq \rho} A_{2d+(i-1)\rho+j-m,r}(2d+j+(i-1)\rho+1)}{\sum_{0 \leq i \leq m-1} A_{i,r}(i+m+1)} \\ &= \frac{N_1 + N_2}{D_1} \quad (\text{say}) \end{aligned}$$

Where,

$$N_1 = \frac{1}{2d} \sum_{0 \leq j \leq 2d-m} A_{j,r}(m+j+1)$$

$$= \frac{r+1}{2d} \frac{H(2d+2)-H(2d-0.5\rho+1.5)}{H(4d-\rho+2)-H(2d-0.5\rho+1.5)}$$

and,

$$\begin{aligned} N_2 &= \sum_{1 \leq i \leq k} \frac{1}{2d+i\rho} \sum_{1 \leq j \leq \rho} A_{2d+(i-1)\rho+j-m, r}(2d+(i-1)\rho+j+1) \\ &= (r+1) \sum_{1 \leq i \leq k} \frac{1}{2d+i\rho} \sum_{1 \leq j \leq \rho} a_{2d+(i-1)\rho+j-m}(2d+(i-1)\rho+j+1) \\ &= \frac{(r+1) \sum_{1 \leq i \leq k} \frac{1}{2d+i\rho} \sum_{1 \leq j \leq \rho} \frac{1}{(2d+j+(i-1)\rho+2)}}{H(4d-\rho+2)-H(2d-0.5\rho+1.5)} \\ &= \frac{r+1}{H(4d-\rho+2)-H(2d-0.5\rho+1.5)} \sum_{1 \leq i \leq k} \frac{H(2d+i\rho+2)-H(2d+(i-1)\rho+1)}{2d+\rho i} \end{aligned}$$

and,

$$\begin{aligned} D_1 &= \sum_{0 \leq i \leq m-1} A_{i, r}(i+m+1) \\ &= (r+1) \sum_{0 \leq i \leq m-1} a_i(i+m+1) \\ &= \frac{(r+1)H(4d-\rho+2)-H(2d-0.5\rho+1.5)}{H(4d-\rho+2)-H(2d-0.5\rho+1.5)} \\ &= (r+1) \end{aligned}$$

Hence, we have the average storage utilization given by:

$$E[u_p] = \frac{\frac{1}{2d}(H(2d+2)-H(2d-0.5\rho+1.5)) + \sum_{1 \leq i \leq k} \frac{1}{2d+i\rho}(H(2d+i\rho+2)-H(2d+(i-1)\rho+1))}{H(4d-\rho+2)-H(2d-0.5\rho+1.5)}$$

If d is large, then the summation in the previous expression can be replaced by integration and after some algebraic manipulations we have:

$$E[u_p] = \frac{\frac{1}{2d} \ln \frac{2d+2}{2d-0.5\rho+1.5} + \frac{1}{\rho} \left[\ln \frac{4d+2}{2d+\rho+2} + 2 \ln \left(\frac{\ln(4d+2)-2}{\ln(4d+2)-2} \right) \right]}{\ln \frac{4d-\rho+2}{2d-0.5\rho+1.5}} \quad (3.12)$$

3.7.1.3. Probability of a Partial Expansion

Partial expansions are given to a node when the contents of the node are:

$$2d+i\rho \quad 0 \leq i \leq k-1$$

Thus, the probability of a partial expansion can be given by the summation:

$$\begin{aligned} q_{PE} &= \sum_{0 \leq i \leq k-1} \frac{(2d+i\rho+1)}{r+1} A_{2d+i\rho-m, r} \\ &= \sum_{0 \leq i \leq k-1} (2d+i\rho+1) a_{2d+i\rho-m} \\ &= \frac{1}{H(4d-\rho+2) - H(2d-0.5\rho+1.5)} \sum_{0 \leq i \leq k-1} \frac{1}{2d+i\rho+2} \end{aligned}$$

Note that, the ρ factor in front of the summation index, makes the sum not expressible in form of Harmonic Numbers. We conjecture that it is difficult to express the summation in a closed form. Also note that, large errors will be induced if the summation is expressed in form of integration, as ρ may be large.

3.7.2. Hashing at the Leaf Level

As discussed in the next chapter, the B-tree data structure is sensitive to the distribution of key values. In applications where this variation is not acceptable, hashing can be combined with the B-tree indexing structure [RaM89]. In this scheme, a leaf logically consists of a collection of buckets. During retrieval, instead of performing a binary or sequential search within the leaf, the address of the bucket holding the key can be computed by a hash function. With a proper choice of the hash function, a stable average storage utilization can be achieved, which does not vary with the distribution of key values.

Concurrency control performance can also be improved in this scheme. Instead of locking the entire leaf (which one has to do in the conventional B-tree configuration, to ensure safe shifting of keys within the leaf), each of the buckets can be individually locked. The granularity of locks, is reduced, and hence more concurrency can be achieved at the

leaf level.

However, the standard dangers of hashing are still present in this case. An improper choice of the hash function can drastically reduce storage utilization. One technique to solve this problem is to use a perfect hash function. But the best known method to get a perfect hash function takes at least linear time in the size of the keys. A recent paper discusses techniques to get perfect hash functions for such applications in greater detail [RaL89].

Another possible optimization is to use a dynamic hashing scheme in the leaf level. The concept of partial expansion works especially well in this context. The partial expansions of the leaves correspond to increase in local depth of the pages of the dynamic hashing scheme, and a split (i.e., a full expansion) of a leaf to an increase in global depth. Since the data within a logical leaf (which consists of one or many such hash buckets) is structured in the form of a hash table, more concurrency can be achieved than when the data is stored as a flat sequence of bytes.

However, when more than one process accesses this tree simultaneously, care has to be taken to ensure proper splitting of the leaves. Consider a logical leaf L consisting of the buckets $\langle B_1, B_2, \dots, B_p \rangle$. Assume that two processes P and Q are acting on this leaf simultaneously. P inserts the key into bucket B_1 and Q inserts the key into bucket B_2 . Both the processes find that the respective buckets are full and hence the leaf needs an expansion. Such a situation may lead to anomalies. In order to avoid such anomalies, an extra bucket, called the *overflow bucket* is associated with every logical leaf. When any process finds that the bucket into which it is supposed to insert a key, is full, the process inserts the key into the overflow bucket. When the overflow bucket gets filled up, only then the whole leaf is split.

3.8. Summary

In this chapter, we have discussed the different variants of the B-tree data-structure and have deduced expressions for the average storage utilization and the probability of split (during insertions) of two variants of B-trees. We have also discussed a few possible enhancements of the basic B-tree data-structure and deduced analytical results for one of them. In the next chapter, we verify the correctness of some of the results derived in this chapter by simulations.

CHAPTER 4

Performance of B^+ -Trees

4.1. Introduction

An interesting property of B^+ -trees is their sensitivity to the key values. Since B^+ -trees form an integral part of nearly all database systems, the effect of key distributions in different applications on the performance of B^+ -trees is an important aspect of the overall database performance.

Although it is hard to mathematically model data appearing in real life, mathematical modeling may be the only way to simulate a real life system. A scientific model is defined as an abstraction of some real life system which can be used for prediction and control. The model enables the analyst to determine how one or more changes in various aspects of the modeled system may affect the system performance as a whole. Special care must be taken to check that the model is indeed a true representative of the real life system. An important aspect of statistical modeling is that it is more concise than, for instance, a verbal description of the system. The model should encompass the entire domain of the parameter being modeled, including the boundary conditions, and at the same time general enough to be short and succinct. To embody completeness, verbosity should never be resorted to. Models may be of various types — *iconic*, *analog* or *symbolic*. Statistical models usually fall in the third category.

Computer simulations are replicable experiments. Simulations may be re-run with changes in selected parameters or conditions made by the investigator. In addition, computer simulations often allow one to induce correlation between random number sequences

to improve statistical analysis of the output of the simulation. In general, a negative correlation is desirable when the outputs of two replications are to be summed, whereas a positive correlation is preferred when the results are to be differenced, as in comparison of experiments.

In this chapter, we study the behavior of B^+ -trees under various distribution of keys. The key distributions are modeled by several probability density functions. The methods used to generate keys following such distributions are described and the variation of the simulated parameters is discussed.

4.2. Simulation Parameters and Workload Description

To examine the behavior of B^+ -trees, the parameters we measured include *the average storage utilization* and *the average time of operation*. The average storage utilization reflects the efficiency of the dynamic data structure and the average time of operation reflects the cost of performing operations on the data structure. These parameters were discussed in Chapter 3.

The simulations were done on a Sequent Symmetry machine with eight processors at times when the system load was minimal. All simulation runs were performed on trees stored entirely in main memory. The cost of an operation is determined by simulating the costs of I/O and measuring the elapsed CPU times. We assumed that the page size is 1K bytes. This assumption was motivated by the fact that 1K bytes is the typical page size in most of the existing commercial database systems (e.g., Oracle, Ingres). Moreover, the page size of the underlying Dynix operating System is also 1K bytes. Thus, a page size of 1K enhances aspects of the simulation considerably. We assumed that at any time, at most two pages can reside in main memory. The last assumption is motivated by the fact that about 80 percent of the operations on a B-tree are searches. A search does not necessitate a

writing back the page read in. So having two pages in the memory or otherwise does not affect the search performance. During an insertion or a deletion operation, only three pages need to be manipulated together at one time. Some careful reorganization of the algorithms enables all inserts and deletes to be done with only two pages in memory. No underlying buffer manager is assumed. The performance figures presented below might vary substantially in the presence of an intelligent buffer manager.

The time for an I/O operation from or to the secondary storage device is modeled in terms of two parts — the *seek time* and the *transfer time*. Seek time refers to the time taken to position the disk head on the appropriate track of the disk. Based on the actual seek times of the underlying hardware, the the simulated seek time is taken to be 15 milliseconds. The time to read or write a page is assumed to be 8 milliseconds. It is also assumed that a page is the unit of I/O and the size of every node in the B^+ -tree is bounded by the page size. When reading or writing a page that was read in the last disk access, seeks are avoided. We ignore any rotational latency in positioning the disk-head over the correct track. However, for non-sequential I/O (e.g., searching from the root to the leaves) seeking is necessary. Following the assumption that there is no buffer manager, an updated page is always written back before the search or update operation terminates. The CPU time is measured directly by *stop watches* throughout the program. The stop watches are implemented by the *getrusage()* system call which is provided by the operating system. There are certain problems with this system call. It is supposed to resolve at the order of microseconds, but in practice, for small computation steps, it resolves in the order of milliseconds. The resolution improves with the increase in time being measured.

4.3. Test Data Generation

Generation of good test data, as in any simulation, is crucial to the validity of the results. Because sampling from a particular distribution being modeled involves the use of random numbers, an efficient method to generate random numbers is necessary. The Dynix operating system provides a system call *random()* to generate such *pseudo-random numbers*, uniformly distributed in the range $[0, 2^{32}-1]$. We assume that the sequence of numbers generated by *random()* satisfies the randomness criteria as given in [Knu71b]. As the number of different numbers representable in a digital computer is fixed, to improve efficiency a deterministic method of generating random numbers is often used. Such numbers do not have all the properties of random numbers but are adequate for our use. Many applications (e.g., biological simulations), necessitate a non-uniform distribution of data.

A sequence of numbers following a particular distribution is called a *random variate*. Various techniques have been cited in literature to generate test data following different distributions. In the following sub-sections we discuss very briefly two general methods that have been employed to generate the test data following various non-uniform distributions.

Most of the techniques employed generate random variates in specific intervals. We have extended these methods to generate random variates which are integers lying within the range $[0, 2^{12}-1]$. Scaling a random sequence, does not affect the randomness of the sequence. That is, if $\langle X_i \rangle$ is a random sequence, then for constants β and c , $\langle [cX_i] + \beta \rangle$ is also a near-random sequence with the same normalized moments as the original one. The resulting sequence is called near-random because some errors are introduced in the process of taking the floor, detailed discussion of which can be found in [Knu71b].

4.3.1. The Inverse Transform method

Let X be a random variable with a cumulative probability distribution function (c.d.f.) $F_X(x)$. By definition then, $F_X(x)$ is non-decreasing and is bounded below by 0 and above by 1. Let the inverse $F_X^{-1}(y)$ of F be defined as follows:

$$F_X^{-1}(y) = \min \{x : F_X(x) \geq y\}, \quad 0 \leq y \leq 1$$

Then, if U is uniformly distributed over the interval $[0,1]$, $X = F_X^{-1}(U)$ has the cumulative distribution function $F_X(x)$.

Thus, given any probability density function (p.d.f.) and an uniform random variable (i.e., a random number generator), we can get a random variate having a c.d.f. as $F_X(x)$. The problem with this method lies in the fact that the c.d.f. must be expressed in a closed form and, moreover, the inverse of the c.d.f. should be expressible in a closed form[†]. When these conditions are not satisfied, this method is of little practical interest.

4.3.2. The Acceptance Rejection Method

In the *acceptance rejection* method, the c.d.f. is not required. The random variate can be generated directly from the p.d.f. Let X be the random variate to be generated following the probability density function $f_X(x)$. In this method, $f_X(x)$ is represented as

$$f_X(x) = Ch(x)g(x)$$

where $C \geq 1$, $h(x)$ is also a p.d.f., and $0 < g(x) \leq 1$. Next, a uniform random variate (a random number) U , and a random variate V satisfying the p.d.f. $h(x)$ are generated. If the inequality $U \leq g(Y)$ holds, then Y is accepted as the random deviate satisfying $f_X(x)$. Otherwise the process is repeated with new U and V .

[†] Note that by the definition of inverse, F_X^{-1} need not be one-to-one. But solving the equation $F_X(x) = U$ by standard numerical methods such as *root searching* is expensive.

4.4. Review of Probability Distributions

Numerous probability distributions exist in statistical theory. Since it is impossible to evaluate the performance of B⁺-trees under all possible distribution of keys, we consider in this sub-section brief descriptions of those distributions which we will use. See [Fel68] for more information on probability distributions.

4.4.1. The Uniform Distribution

The uniform distribution is often adopted as the working hypothesis in most of the scientific applications. It has been adopted as the "most realistic" distribution of keys in most of the benchmarks for database systems [BDT83]. Also, as mentioned above, computer generation of nearly all random variates by standard methods needs one or more uniformly distributed random variates. The uniform distribution occur in real life in scientific measurements. For example, the distribution of *round-off errors*, tabulated to a fixed number of decimal places, is usually uniform. Most of the analyses of B⁺-trees cited in literature, assume an uniform distribution of keys.

The p.d.f. of the continuous uniform distribution is given by:

$$p(x) = U_0(x) - U_1(x)$$

where $U_i(x)$ is the *unit function* of order i [Knu71b]. The graph of the p.d.f. of the uniform distribution is given in Figure 4.1. The random variates following this distribution are obtained directly from the system calls *srandom()*, *random()*, *setstate()* and *initstate()*. They are reduced to the interval $[0, 1]$ by dividing by the largest possible number of 32 binary digits.

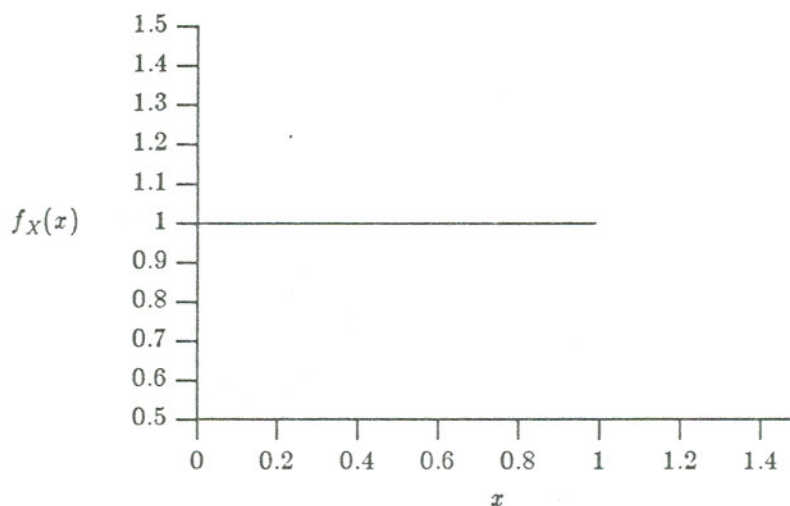


Figure 4.1: The Uniform Distribution

4.4.2. The Normal Distribution

The normal distribution originated in the early 17th century when it was used to study the variation of errors in measurements of planetary motions. Biological data is often normally distributed. For example, the stature of individuals (e.g., the arm length), the weight of the brain and the length of the forehead are all normally distributed.

The normal distribution has the p.d.f. given by:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The parameters μ and σ are the *mean* and the *standard deviation* respectively. A normal distribution with *mean* μ and *standard deviation* σ is denoted by $N(\mu, \sigma)$. The mean is the unnormalized first moment while the standard deviation is the second moment normalized by the square of the first moment. A plot of two normal distributions is given in Figure 4.2.

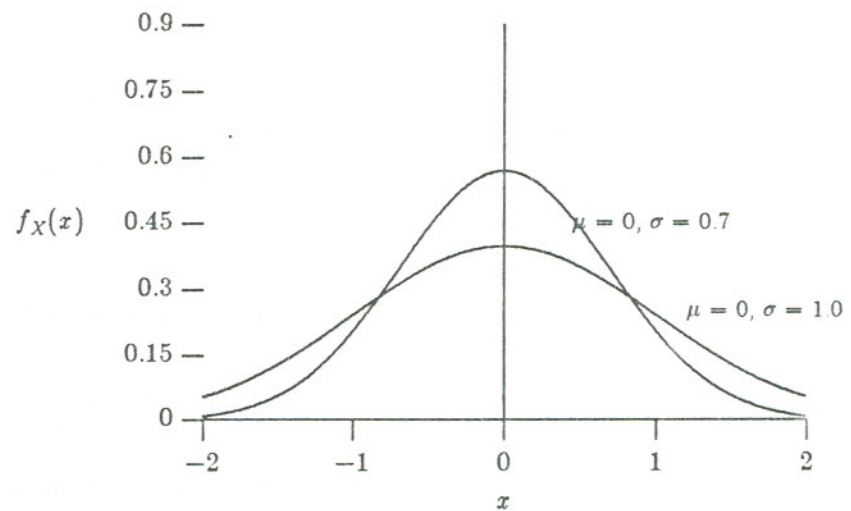


Figure 4.2: The Normal Distribution

Random variates following a normal distribution are generated by using following algorithm:

ALGORITHM 4.1

Input : Two independent uniform random variates.

Output : A random normal variate with mean μ and standard deviation σ .

Method :

Step 1: Generate two random numbers U_1 and U_2 in the interval $[0, 1]$.

Step 2: Compute $Z = (-2\ln U_1)^{\frac{1}{2}} \cos 2\pi U_2$

Step 3: Compute $Z' = \mu + \sigma Z$.

Step 4: Return Z'

□

4.4.3. The Exponential Distribution

Exponentially distributed data occur most frequently in applications measuring inter-arrival time. The mean time between failures of certain kinds of components, the time between successive electrical impulses received by measuring devices implanted in the spinal cord of mammals and the time between successive emissions of alpha particles from radioactive materials, all follow exponential distributions.

The p.d.f. of an exponential distribution is given by:

$$f_X(x) = \begin{cases} \mu e^{-\mu x} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

The parameter μ is called the mean of the distribution. A plot of the exponential distribution is given in Figure 4.3.

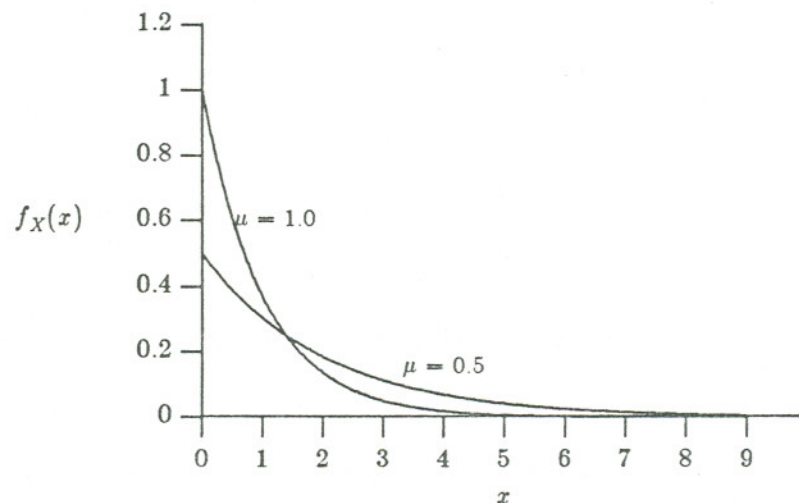


Figure 4.3: The Exponential Distribution

To generate random variates following an exponential distribution, the following algorithm is used:

ALGORITHM 4.2

Input : An independent uniform random variate.

Output : An exponential random variate with mean μ .

Method :

Step 1: Generate a random number U between $[0,1]$.

Step 2: Compute $Z = -\mu \ln U$.

Step 3: Return Z

□

4.4.4. The Gamma Distribution

Data that is gamma distributed occur most frequently in systems where the operation of a particular subpart is essential in the proper functioning of the entire system as a whole. As an example consider the following: In order to improve the reliability of the system, the system may be designed to carry $(r-1)$ spare components to be used in case the given component fails. When the original component fails, one of the remaining $(r-1)$ spare components is activated to take its place. The process continues until all the r components have failed. At this point the entire system fails. Assuming that the entire system can fail only if the single essential component fails, the lifetime of the entire system is the sum of the times until failure, X_1, X_2, \dots, X_r , of the r components. Assuming that each of the X_i 's are exponentially distributed, the time until the entire system fails, Y , given by $Y = X_1 + X_2 + \dots + X_r$, is gamma distributed. A more realistic example of the gamma distribution occurs in the measurement of average mean value of radioactivity within a sample of shale. An important property of the gamma distribution is that it can be used to approximate nearly all distributions in real life.

The p.d.f. of the Gamma distribution is given by:

$$f_X(x) = \begin{cases} \frac{x^{\alpha-1} e^{-x/\beta}}{\beta^\alpha \Gamma(\alpha)} & 0 \leq x < \infty, \alpha > 0, \beta > 0 \\ 0 & \text{otherwise} \end{cases}$$

A Gamma distribution with parameters α and β is often denoted by $G(\alpha, \beta)$. In effect $G(\alpha, \beta)$ is the sum of α exponential random variates, each with mean β . Figure 4.4 shows the gamma distribution for different values of α and β .

A random variate following a gamma distribution for integral α is generated by the following algorithm:

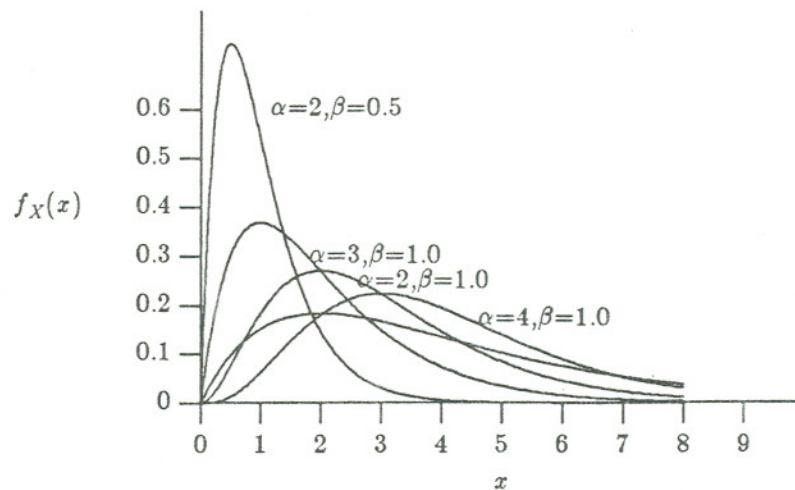


Figure 4.4: The Gamma Distribution

ALGORITHM 4.3

Input : α independent exponential random variates.

Output : A gamma random variate with parameters α and β .

Method :

Step 1: Generate α independent random exponential variates E_i with mean β .

Step 2: Compute $Z = \sum_{1 \leq i \leq \alpha} E_i$.

Step 3: Return Z

□

4.4.5. The Beta Distribution

The distributions discussed above (excepting the uniform distribution), are not applicable for describing the variation of a random variable whose range of possible values is bounded above and below. Examples of such distribution in real life occur in (i) the distance from one end of a steel bar of known length to the point where failure occurs when the bar is subjected to stress, (ii) the proportion of total farm product spoiled by a pest. In computers, the number of different representable numbers (integer or real) is finite. This is why the beta distribution is adopted in applications which necessitate non uniform data and where the standard approximations to a random variate are not applicable.

The p.d.f. of the beta distribution is given by:

$$f_X(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

and is often denoted by $B(\alpha, \beta)$. A plot of the beta distribution for different values of α and β is given in Figure 4.5:

The following algorithm is used to generate random variates following the Beta distribution:

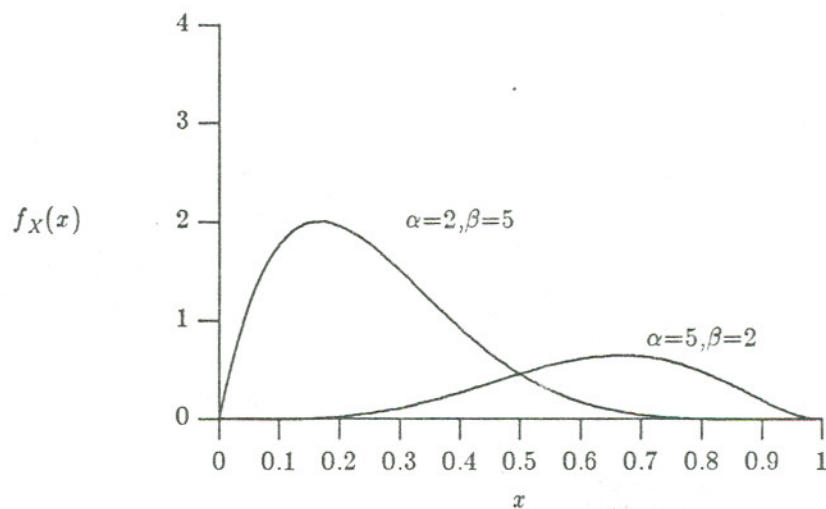


Figure 4.5: The Beta Distribution

ALGORITHM 4.4

Input : Two independent gamma variates.

Output : A beta random variate with parameters α and β .

Method :

Step 1: Generate Y_1 from $G(\alpha, 1)$.

Step 2: Generate Y_2 from $G(\beta, 1)$.

Step 3: Compute $X = Y_1 / (Y_1 + Y_2)$

Step 4: Return X

□

4.4.6. The Log-Normal Distribution

The log-normal distribution occurs most frequently in econometric theory, where the data values are non-negative. In fact, in the case where the data is non-negative, the log-normal distribution is the placeholder of the normal distribution. Data distributed log-normally occur in particle sizes in naturally occurring aggregates, length of words and sen-

tences, concentration of trace and non-trace elements in plants, lifetimes of mechanical and electrical systems and many others. The log-normal distribution has its own place in the theory of random variables. As proved in a recent paper [Rao89], the log-normal distribution violates the Central-Limit theorem.

The p.d.f. of the log-normal distribution is given by:

$$f_X(x) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma x}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}} & 0 \leq x < \infty \\ 0 & \text{otherwise} \end{cases}$$

Figure 4.6 shows plots of the log-normal distribution for different values of mean and standard deviation.

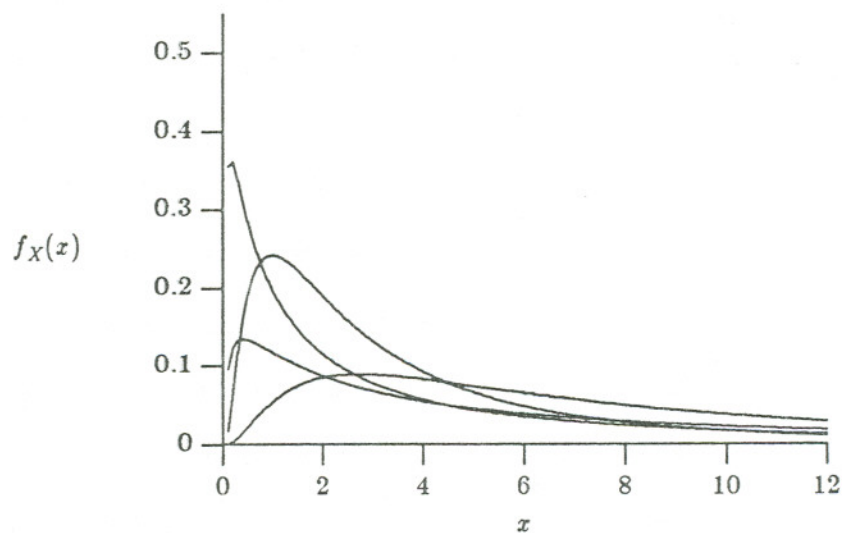


Figure 4.6: The Log-normal Distribution

A random variate following the log-normal distribution is generated by the following algorithm:

ALGORITHM 4.5

Input : A random normal variate.

Output : A log-normal random variate with mean μ and standard deviation σ .

Method :

Step 1: Generate a random variate Z from $N(\mu, \sigma)$.

Step 2: Compute $X = \mu + \sigma Z$

Step 3: Compute $Y = e^X$

Step 4: Return Y

□

4.5. Operation Execution Times

The variation of the time to insert or delete one key from a tree is discussed in this section. The graphs presented in this section are from simulations which are discussed in detail in the next paragraph. Figure 4.7 through Figure 4.12 plot the variation of the time to insert a key against the size of the tree. Figure 4.13 through Figure 4.18 plot the variation of the time to delete a key against the tree size. A node size of 50 keys is used for the simulation of insertion times and a node size of 20 for the simulation of deletion times. Each figure shows the results of a different key distribution.

The insertion process consisted of inserting up to 50,000 keys into an initially empty tree. The algorithms for operations on the B-trees are similar to those in [Gra89] and [KwW82]. A stopwatch is started before every insertion and the time to insert the key is determined after the insertion is complete. To avoid excessive details, the graph is divided into three segments. The first segment shows the insertion times when the tree is small. The topmost graph of every figure shows the variation in insertion time for trees with 0 to 1,000 keys. The second segment shows the insertion times when the tree contains 9,000 to

10,000 keys and is shown in the middle graph of every figure. The lowermost graph of every figure shows the variation in time when the tree contains 47,000 to 48,000 keys. Each graph shows results for two simulation runs with identical sets of keys. The dotted lines refer to deferred splitting, while the solid lines refer to aggressive splitting.

The graphs are more or less flat with occasional spikes. The spikes are caused by splits of pages. Larger spikes are caused by a greater number of splits while some small spikes (occurring in the case of deferred splitting) are caused by system induced errors in measurement. We identified this error by debugging the code. The graphs show equal time to insert a key regardless of whether it is at the beginning or the end of a page into which it is inserted. Inserting near the top necessitates shifting of more keys than when inserting near the bottom but the variation in time is transparent because of the available resolution of both the graph plotting program and the stop-watch that measured the time.

The graphs closely follow the analytical results derived in the previous chapter. In the case of deferred splitting, for example, the number of splits in inserting 3,000 keys, as shown on the three graphs in Figure 4.7 is 83. From the analytical results of the previous chapter we predict 86 (rounded to the nearest integer) splits. For this simulation, our analytical results are correct to an error of -3.4%. For the case of aggressive splitting, Figure 4.7 indicate 120 splits as compared to 125 calculated from the analytical results of the previous chapter — an error of -4%. The error is acceptable considering the fact that the analysis assumed an infinitely large tree.

From the graphs it can be inferred that the number of splits is smaller in the case when the key values are spread out as illustrated in the graphs in Figure 4.9 and Figure 4.8a and 4.8b. The number of splits in the case of a normal distribution with a larger standard deviation (Figure 4.8a) (and signifying a lower density around the mean value) is

smaller than that with a smaller standard deviation (Figure 4.8b, by 4). The two graphs corresponding to the exponential distribution (Figure 4.9a and 4.9b) with different means also illustrate the same result. Comparing Figure 4.7 with Figures 4.8a and 4.8b, we note that the number of splits in the case of Normal distribution is smaller than the number of splits in the case of Uniform distribution. The reason is that the average expected survival interval of a node is more in the case of the normal distribution. The exact mathematical proof of the fact is beyond the scope of this thesis.

Figures 4.13 through Figure 4.18 illustrate the variation in the deletion time with the size of the tree. To obtain the graphs, a tree of size 50,000 is built by inserting keys into an initially empty tree. The keys are stored in a file and are subsequently deleted from the tree in the order the keys are inserted. Again, each graph is divided into three phases as is in the case of insertions, with slight variation in the ranges. Note that the density of spikes is more in the case of deletions. One reason is that the order of the tree is smaller in the case of deletions and consequently merges or redistributions are more frequent. Another reason is the key replication in B^+ -trees and mathematical details of this is discussed in [KwW80a]. The number of larger spikes also is greater in the case of deletion because of the deletion algorithm which we use. When a node becomes less than half full, its neighbors are checked for space. The probability that a sibling that contains just above d keys is small and often keys are redistributed. Key redistribution is an expensive operation resulting in a larger spike. Some of the larger spikes are due to key redistribution while others are due to multiple merges. In our simulations, the cost of redistribution of one level is same as the cost of merging in two levels. Thus, it is not possible to distinguish merges from redistributions in the graphs. Moreover, the number of merges or key redistributions is appreciably less when the tree is large. The reason is that the the probability that a node is merged is low in a tree with a large number of nodes. The number of merges or redistributions of

more than one level is also low (i.e., there are fewer large spikes) because the nodes are more or less stable (i.e., do not contain near d or near $2d$ nodes). The graphs follow a more or less similar pattern with expected variations. For example, the number of large spikes, when the tree is large, is smaller in the case of the Normal distribution because the nodes on the paths which are traversed more are more stable and hence the merges do not propagate much.

Comparing the graphs of insertion times (Figure 4.7 through Figure 4.12) we note that the splits of nodes are distributed within the three segments more or less equally, whereas in the case of deletions there is a relatively high concentration of splits and merges when the tree is small (lowermost segment in the graphs in Figures 4.13 through Figure 4.18). This non-uniformity is due to a hysteresis effect in B^+ -trees resulting from key replications. As the tree size becomes smaller and smaller with more and more deletions, some keys stay in the internal nodes. To get rid of this situation, after a deletion is performed, such redundant keys are checked for in the next higher level. In case there exists some such keys, then redistribution in that level is forced, to get rid of them. Such redistributions occur only after an appreciable number of keys have been deleted from the tree and results in the dense spikes in the third segment of the graphs corresponding to the deletion times.

4.6. The Average Storage Utilization

Figures 4.19 through 4.24 illustrate the variation of the average storage utilization with the size of the tree. The graphs were obtained by inserting 40,000 keys, which followed the particular distributions, into an initially empty tree. A node size of 20 is assumed. The graphs show that in general the average storage utilization fluctuates wildly when the tree is small and stabilizes as the tree grows larger. With different distribution of keys the point of stabilization varies.

The storage utilization of a particular node is a saw tooth curve bounded below by 0.5 and above by 1.0. The initial fluctuation occurs because the number of nodes is small and the average storage utilization of the tree is a composition of a *few* such saw tooth curves. The resulting composition also grows more or less following a saw tooth pattern. Moreover, the storage utilization of the root is also a saw tooth curve bounded below at a lower value (the reciprocal of the maximum number of keys per node). The effect of the root also creates a fluctuation in the overall average storage utilization. As the tree grows, the effect of the root reduces and the composition becomes flatter, in the limiting case with infinitely many keys becoming absolutely flat.

Some interesting features of the B^+ -tree data-structure affecting the average storage utilization can be cited at this point. Like the different variations of the data-structure itself, there are many variants of the operations on the data-structure [Sag86][KwW80a]. For example, the branching condition that we used during the search (i.e., which child of a node to traverse next) has several alternatives. We decided to traverse the i^{th} child n_i of a node n if the search key value was *greater than or equal to* i^{th} key of the node n and less than the $(i+1)^{\text{th}}$ key. The insertion and the deletion algorithms were designed accordingly. Note that, using this condition, the tree degenerates (i.e., most nodes contain exactly the minimum number of keys) with keys appearing in a sorted order or when equal keys are inserted into the tree. Another alternative is to make the *greater than or equal* condition be simply *greater than*. However, in this case, the tree degenerates for reverse sorted keys. Moreover, the key maintenance overhead during insertion and deletion operations to avoid one key getting replicated more than once increases. The nature of the number of keys per node, i.e., even or odd, also affects the average storage utilization. Consider a tree with 11 keys per node. The minimum number of keys that should be in the node can be either 5 ($\lfloor 11/2 \rfloor$) or 6 ($\lceil 11/2 \rceil$). If the number of deletions is smaller than the number of insertions

then the average storage utilization stabilizes faster in the case when there are 6 keys per node.

Following the argument above, we note that the average storage utilization falls close to 0.5 when equal keys are inserted into the tree. It can be argued though that equal keys should not be inserted into the tree. However, the choice of whether or not equal keys should be inserted depends upon the requirements of the database system. We allow equal keys to be inserted into the tree to illustrate the difficulties encountered in doing so. Another problem may arise as a result of inserting equal keys. Consider a leaf of a B-tree of order 3 after some insertions whose keys are: $\langle 1, 3, 3, 3, 3 \rangle$. Now, let there be an insertion of key 2 in the tree. Consequently, this leaf has to be split. One possible resulting configuration after the split may be that the two nodes which are created contain keys as: $\langle 1, 2, 3 \rangle$ and $\langle 3, 3, 3 \rangle$, with another 3 passed up to the next higher level. If adequate care is not taken, then a search process in the future will not be able to find the 3 in the first node. To ensure that such a misdirection does not happen, the first node (e.g., $\langle 1, 2, 3 \rangle$) should not contain the 3 in it. In that case, if the order of the tree is large, appreciable amount of overhead has to be paid to perform such checks. Moreover, doing so may spoil the storage utilization of the first node.

Figures 4.19 through 4.24 illustrate the variation in the average storage utilization with the tree size. We observe from the figures that the average storage utilization stabilizes the fastest in the case of the normal distribution. The fluctuation after the initial stage is also of smaller amplitude. This result is counter-intuitive at a first glance. However, careful scrutiny reveals that though some paths are prone to more splits, the nodes in these paths have a more or less high storage utilization. The nodes in the other paths are traversed less. The storage utilizations of these nodes less frequently fall to the lower bound. The phenomena can be visualized as a composition of saw tooth curves, some of

which rise slowly and the others are faster-rising saw tooth curves (and so with smaller periods). It is the slow-rising curves that stabilize the average storage utilization.

The case of uniform distribution closely follows the analysis of the previous chapter. The average storage utilization in the case of the log-normal and the gamma distributions follow that of the normal distribution due to the similarity their p.d.f.'s as can be seen from the graphs.

The case when the keys are beta distributed requires special attention. It can be seen that the average storage utilization falls after a particular size of the tree (both for the case when the nodes are split aggressively and the case when the splitting is deferred) and stabilizes near a value of 0.53. As mentioned previously, the beta distribution is bounded on the X-axis by an upper and a lower limit. Moreover, the span on the X-axis is also small (smaller than the uniform distribution). As a result of this, when the random variates following the beta distribution are scaled to integer values, on many occasions equal keys are generated (see [Rub81]), resulting in the degeneracy of the tree and hence reduction in the average storage utilization.

From the graphs we can conclude that, in general, the average storage utilization of the tree is more affected by the key distribution than by the nature of splitting.

4.7. The Average and the Total Insertion Time

The average insertion time and the total insertion time are plotted in Figure 4.26 through Figure 4.31. Each graph shows the average time to insert one key (plotted along Y-axis) and the total cost to insert all the keys (also plotted along Y-axis) against the tree size (plotted along X-axis) both for the cases of deferred splitting and aggressive splitting. Up to 40,000 keys are inserted into an initially empty tree of order 10. The average time of inserting one key does not remain constant with the size of the tree. However, since the

majority of the insertions do not induce splits and since the difference in the number of splits for the different key distributions vary slightly, the graphs are more or less identical. Another reason behind this similarity is that the cost of insertion is largely dominated by a search to the appropriate leaf.

The time to insert is higher in the case of deferred splitting than that in the case of aggressive splitting because in order to insert a key in the case of a split, both up and down movement along the path from the root to the leaf is required. If there is no split, the cost of the two cases are the same. The above reasoning follows from our assumption that there can be at most two pages in main memory. However, if more pages can reside in main memory then, as shown in the previous chapter, the cost (and so the time overhead) will be more in the case of aggressive splitting (there will be more splits).

The total time for insertion grows more or less linearly with two kinks in the curve. The kinks occur at points where the height of the tree increases. The slope of the curve after the kinks is larger than the slope of the curve just before it. The slope of this curve (loosely speaking) indicates the average time to insert a key, which is also plotted in the same graph both for the case of aggressive and deferred splitting. The curves for the deferred and the aggressive cases are nearly the same in shape. The curves representing the average time of insertion of a key can be divided into three distinct phases. Each of these phases are characterized by a sharp rising time, which saturates to a constant. The first phase starts at the tree size of 0 and ends at a tree size of approximately 300. The second phase spans from a tree size of 300 to 4500 and the third phase spans the rest of the curve. When the height of the tree suddenly increases, the cost of insertion rises sharply as the number of keys in the tree is much smaller than what can be accommodated in a tree of that height. As more and more keys get inserted into a tree of a fixed height, the slope of the curve decreases. The kinks appear approximately at the points expected. For example

the second kink appears at a point corresponding to the tree size of 4,000 ($\simeq (21 \times 0.65)^3$) since, the average number of keys per node is $\simeq (21 \times 0.65)$ and the height of the tree is 3. An increase in the height of the tree is expected when the number of keys equal the product of the number of nodes and the average storage utilization.

Note that, even though when the keys are beta distributed, the average storage utilization falls below 0.55, it does not affect the the curve representing the average time of insertion (i.e., it is more or less identical to those for other distributions) because at the point of the last kink, the average storage utilization in the case of the beta distribution was also close to 0.62. Though not shown in the graphs to preserve symmetry, the next kink in the case when the keys are beta distributed occurs much earlier than in the other cases.

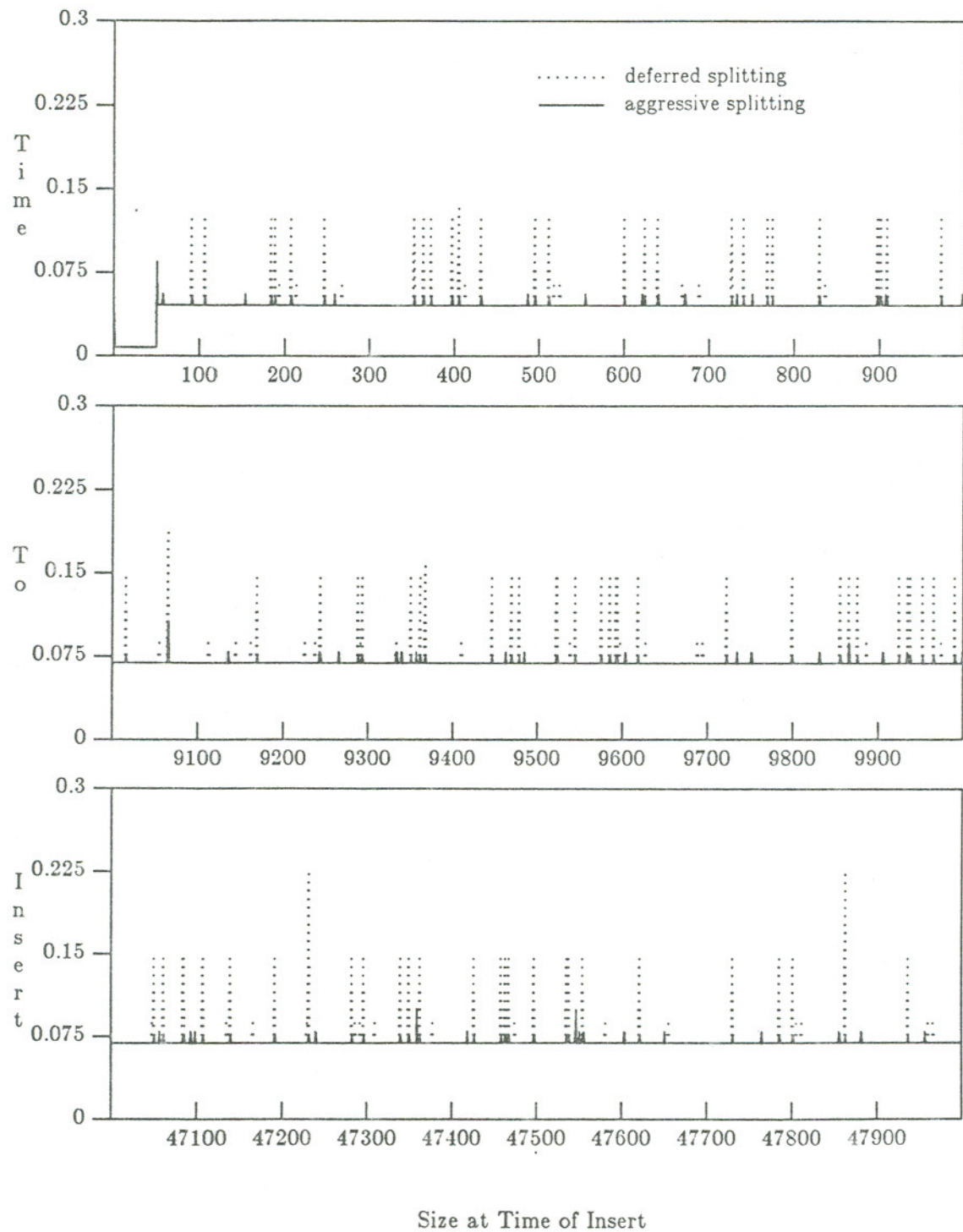


Figure 4.7: Insertion Time vs. Size of Tree for Uniform Distribution

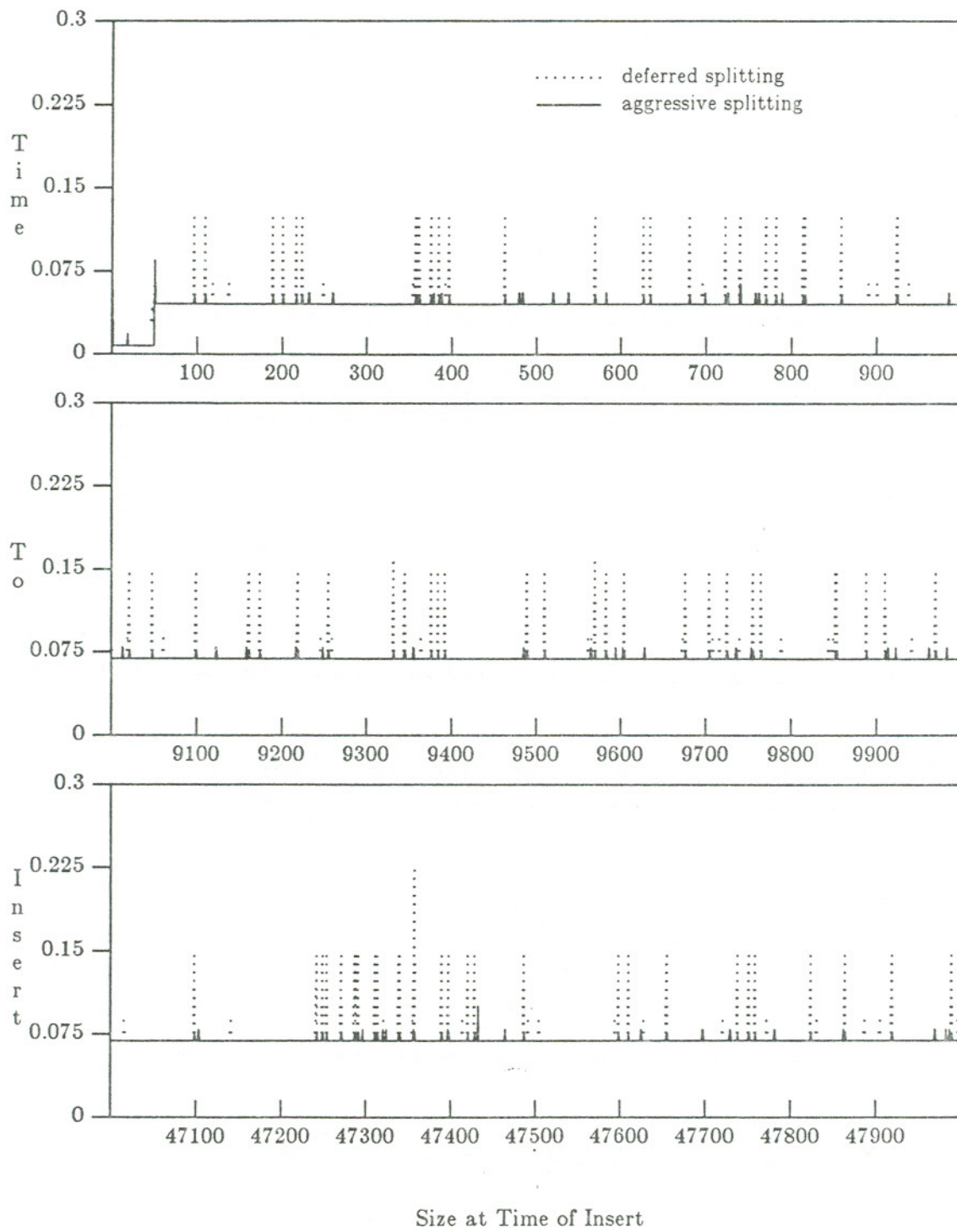


Figure 4.8a: Insertion Time vs. Size of Tree for Normal Distribution ($\mu=1.0, \sigma=0.7$)

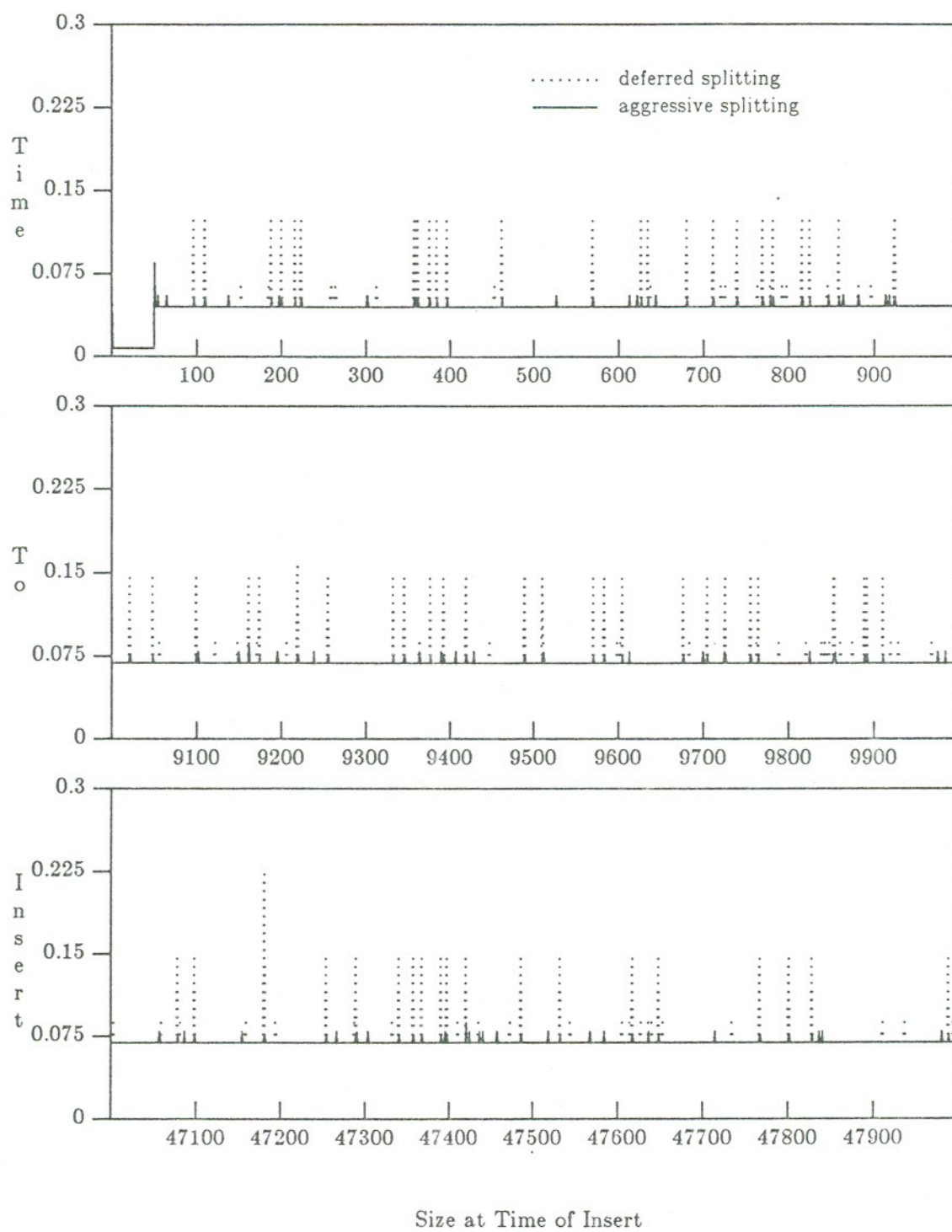


Figure 4.8b: Insertion Time vs. Size of Tree for Normal Distribution ($\mu=1.0$, $\sigma=1.0$)

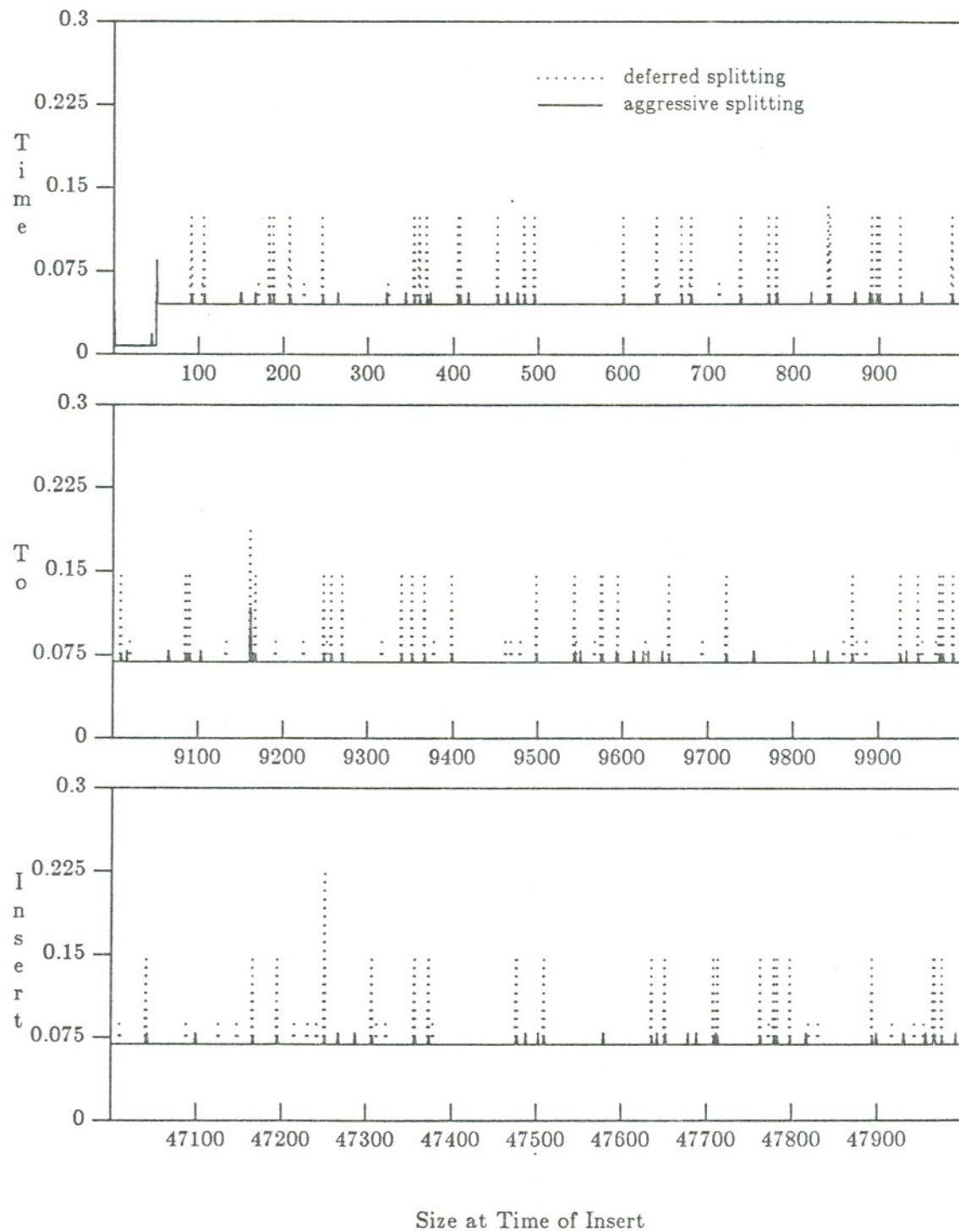


Figure 4.9a: Insertion Time vs. Size of Tree for Exponential Distribution ($\mu=1.0$)

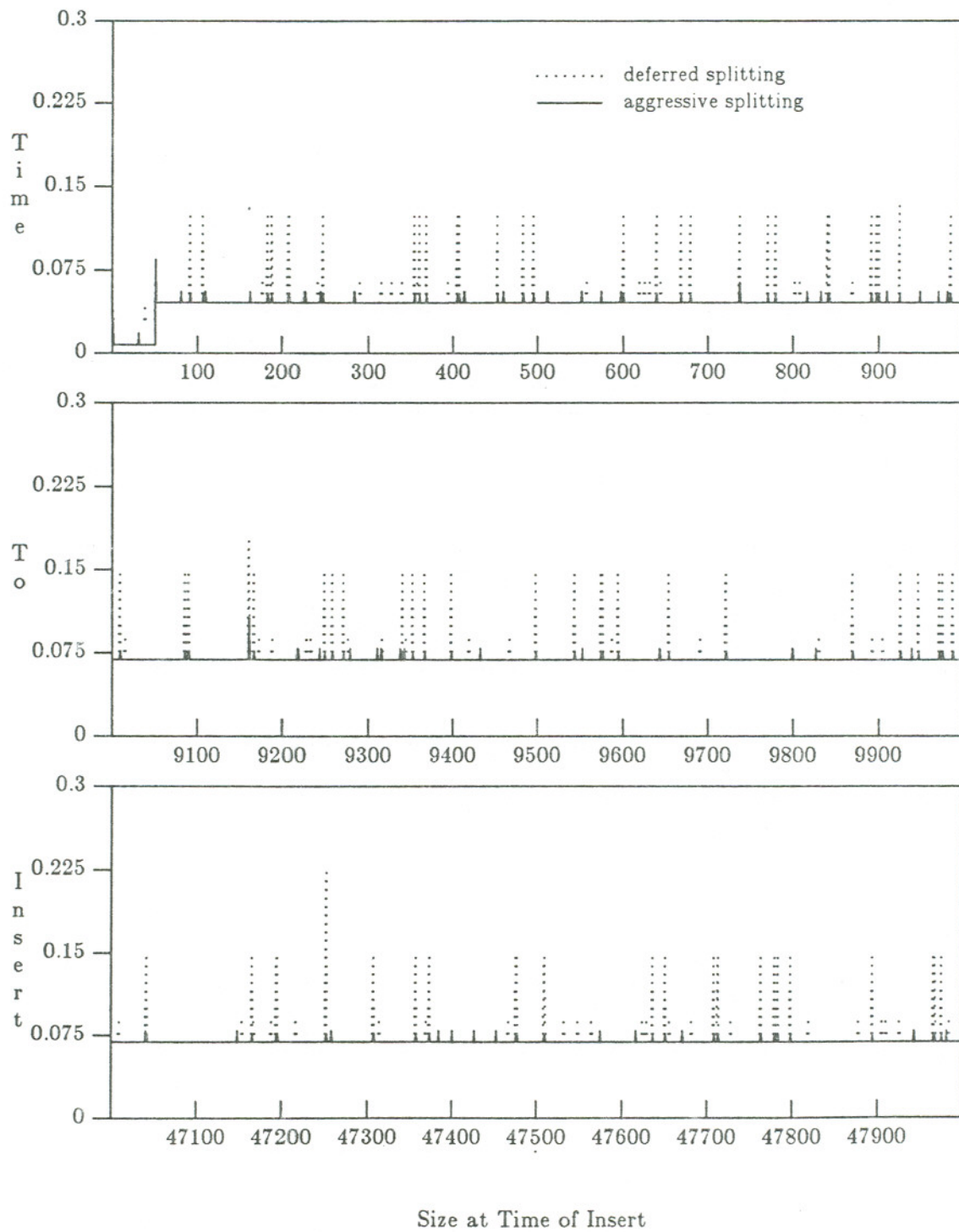


Figure 4.9b: Insertion Time vs. Size of Tree for Exponential Distribution ($\mu=0.5$)

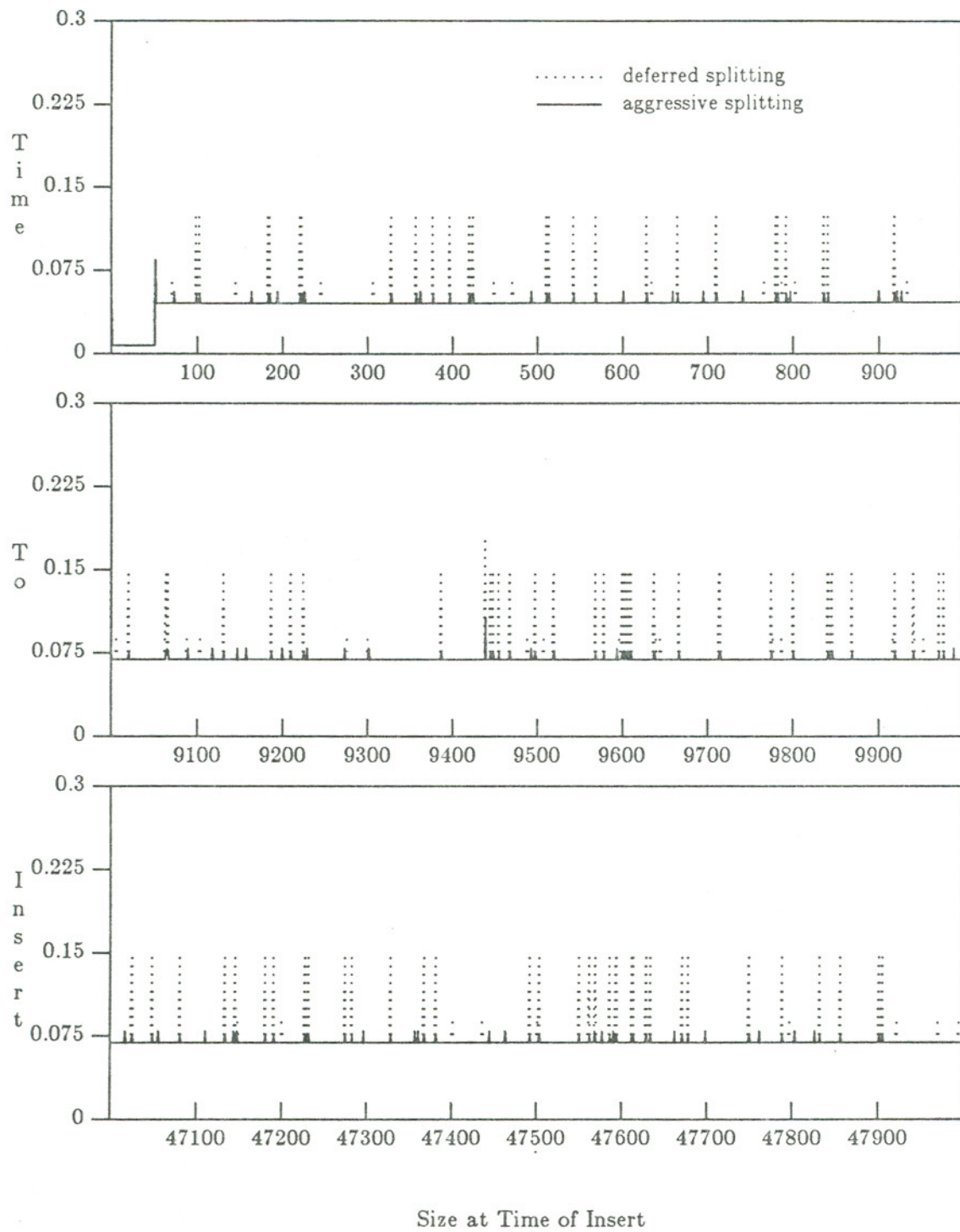


Figure 4.10: Insertion Time vs. Size of Tree for Gamma Distribution ($\alpha=2.0$, $\beta=1.0$)

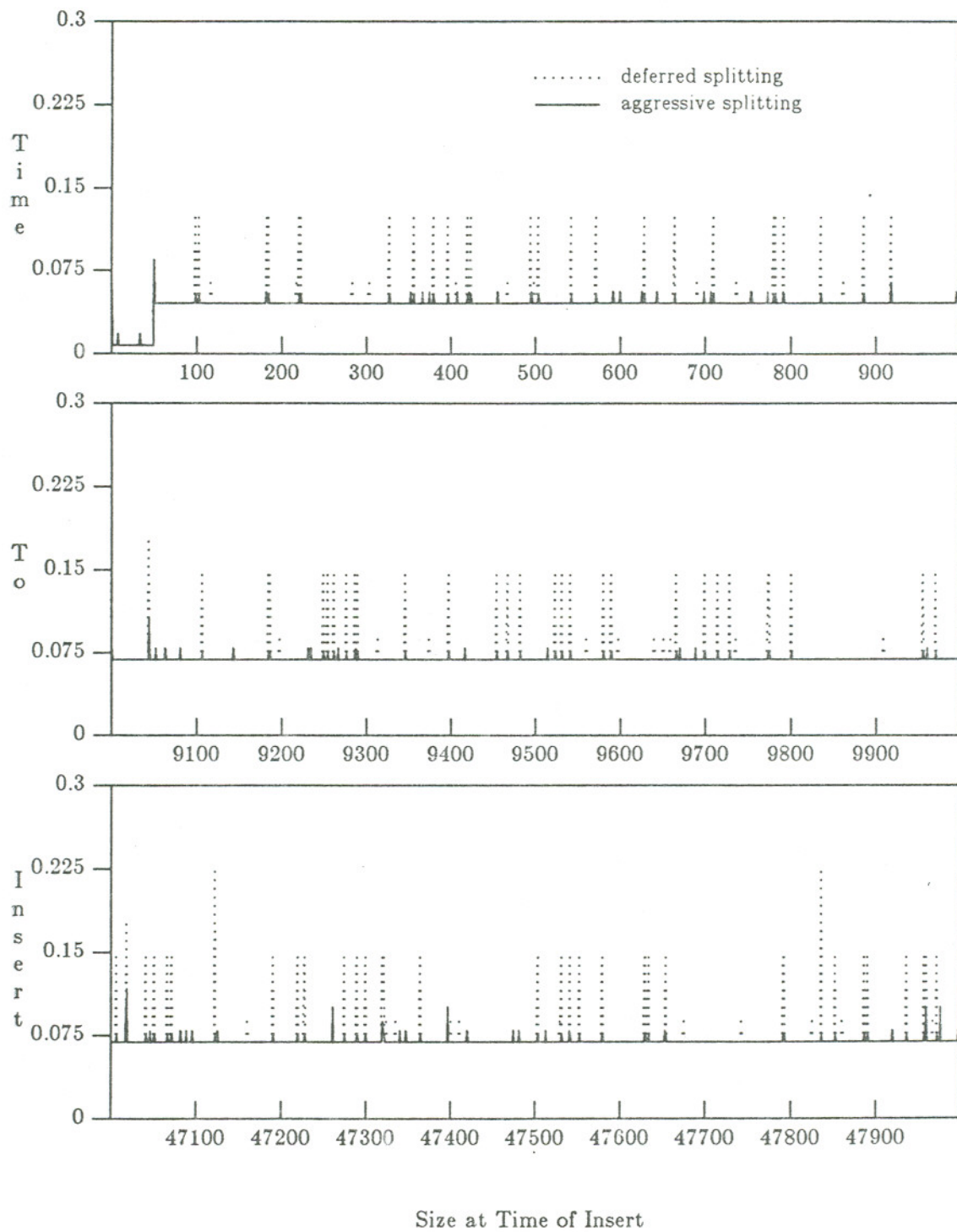


Figure 4.11: Insertion Time vs. Size of Tree for Beta Distribution ($\alpha=2.0$, $\beta=5.0$)

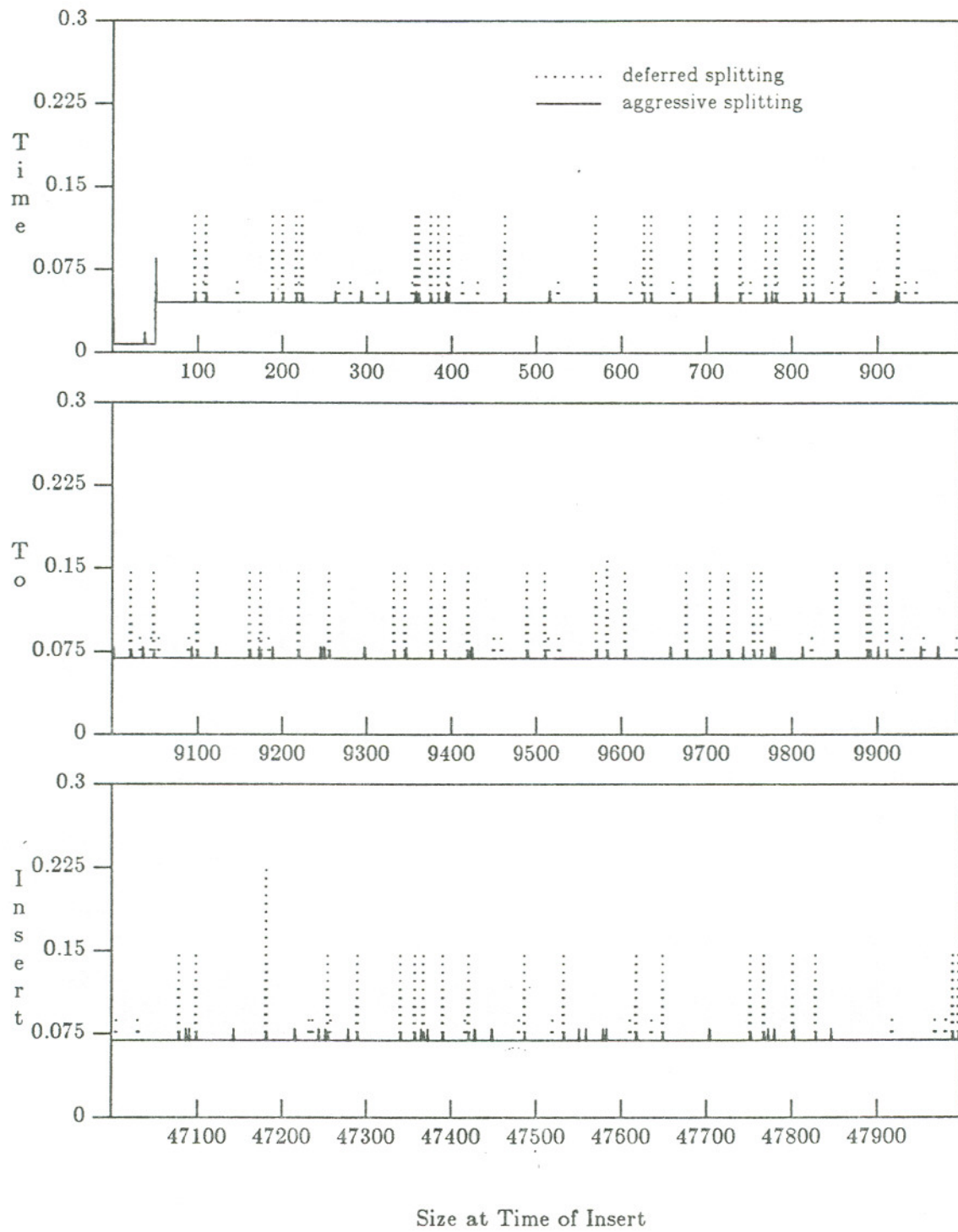


Figure 4.12: Insertion Time vs. Size of Tree for Log-normal Distribution ($\mu=1.0$, $\sigma=5.0$)

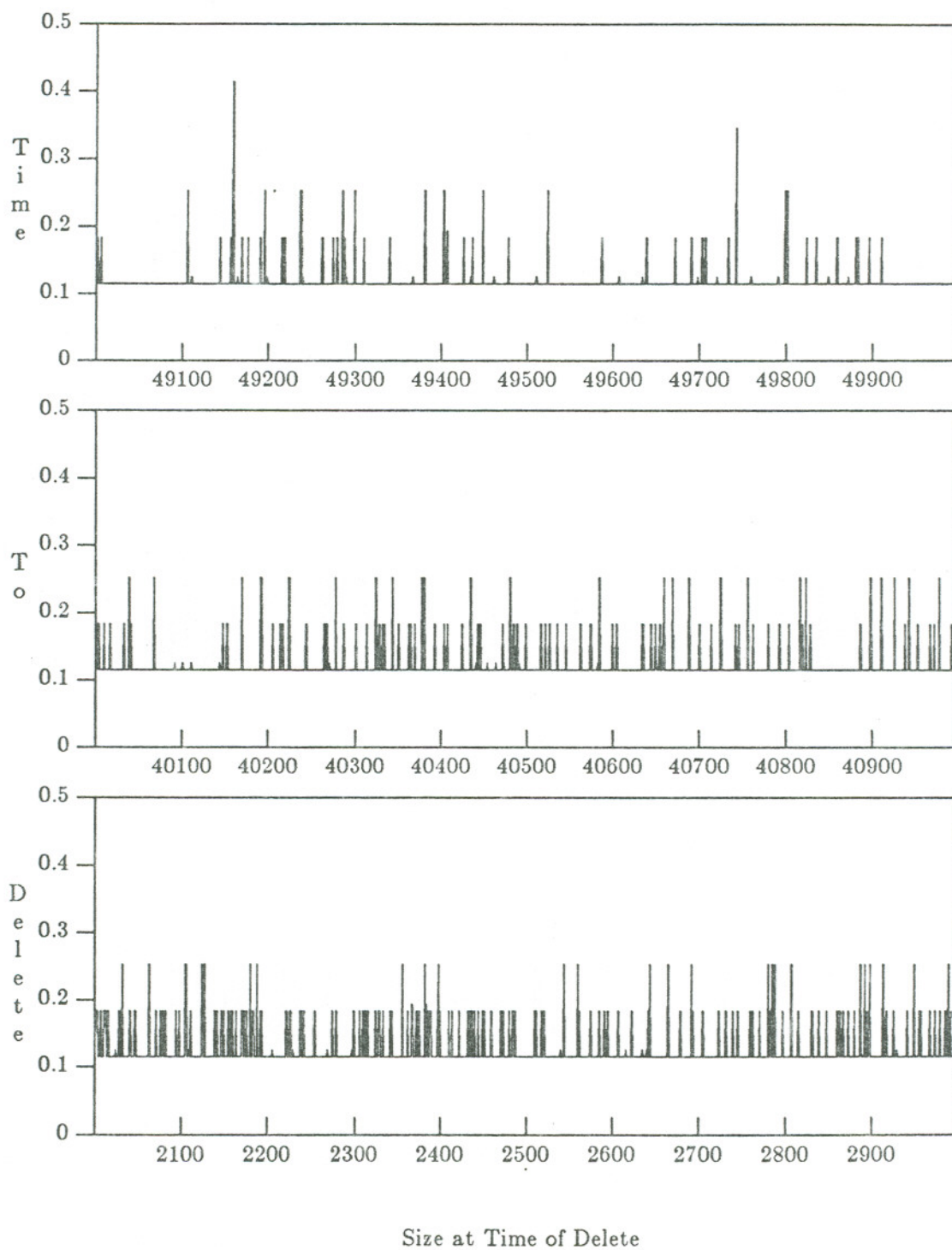


Figure 4.13: Deletion Time vs. Size of Tree for Uniform Distribution

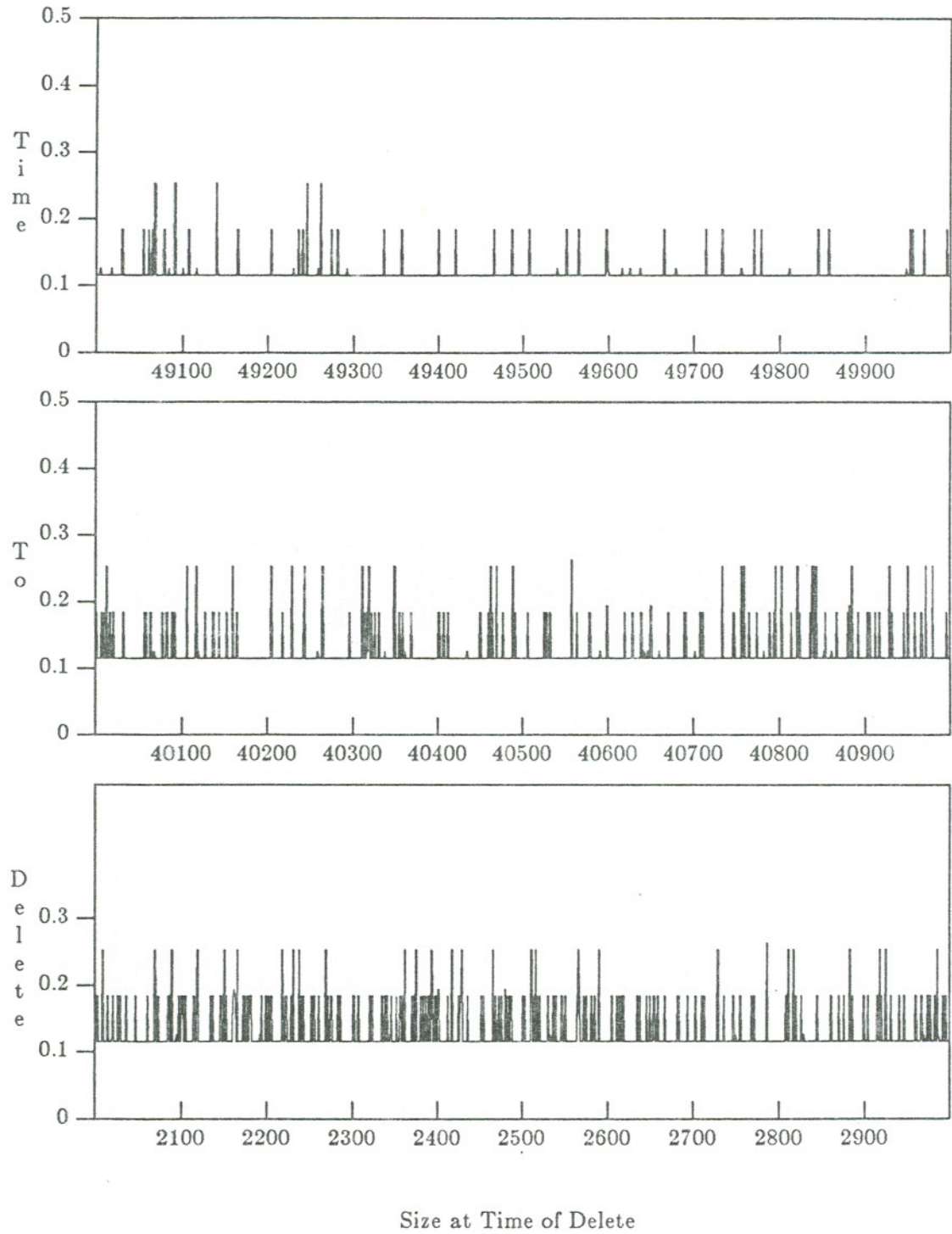


Figure 4.14: Deletion Time vs. Size of Tree for Normal Distribution ($\mu=1.0$, $\sigma=1.0$)

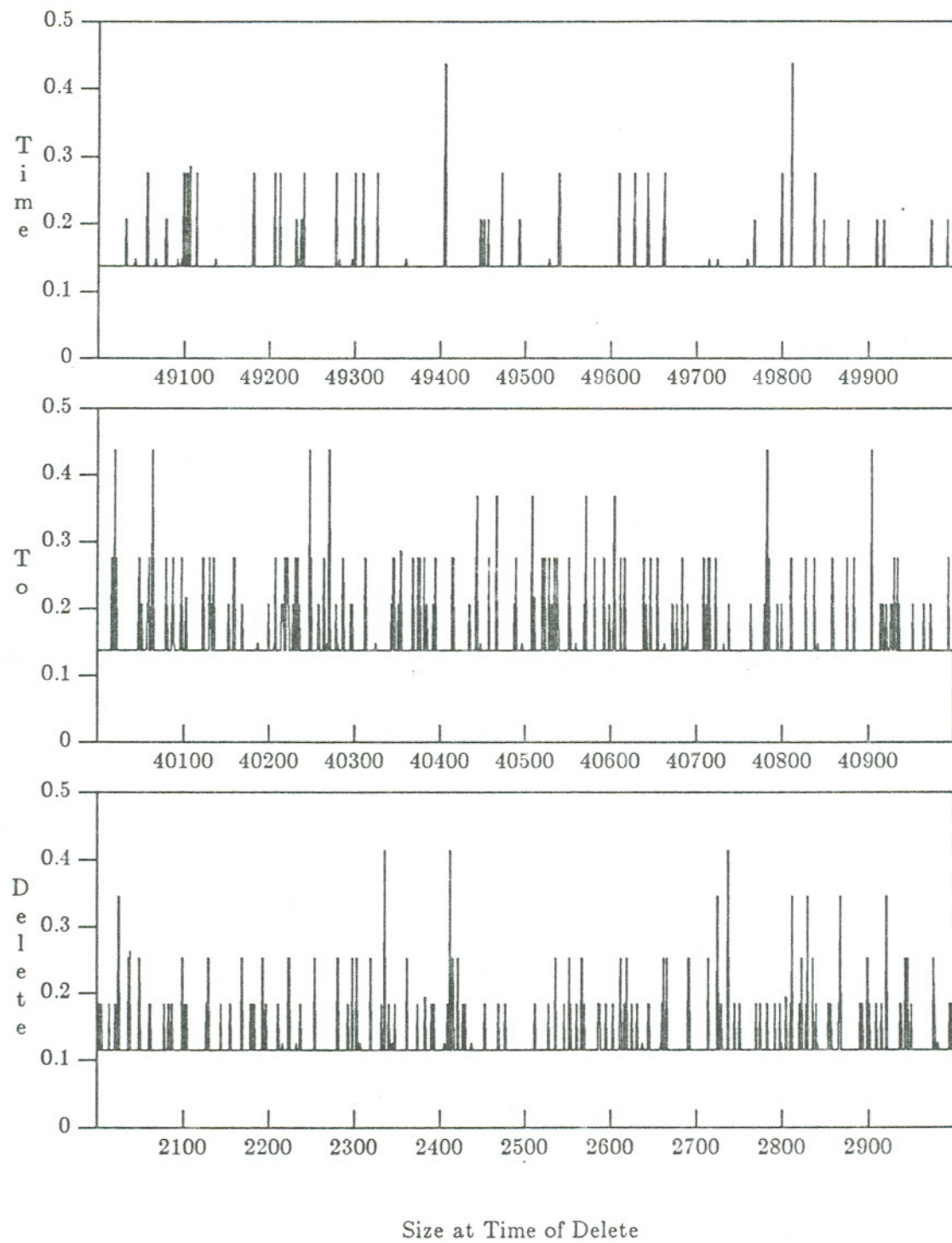


Figure 4.15: Deletion Time vs. Size of Tree for Exponential Distribution($\mu=1.0$)

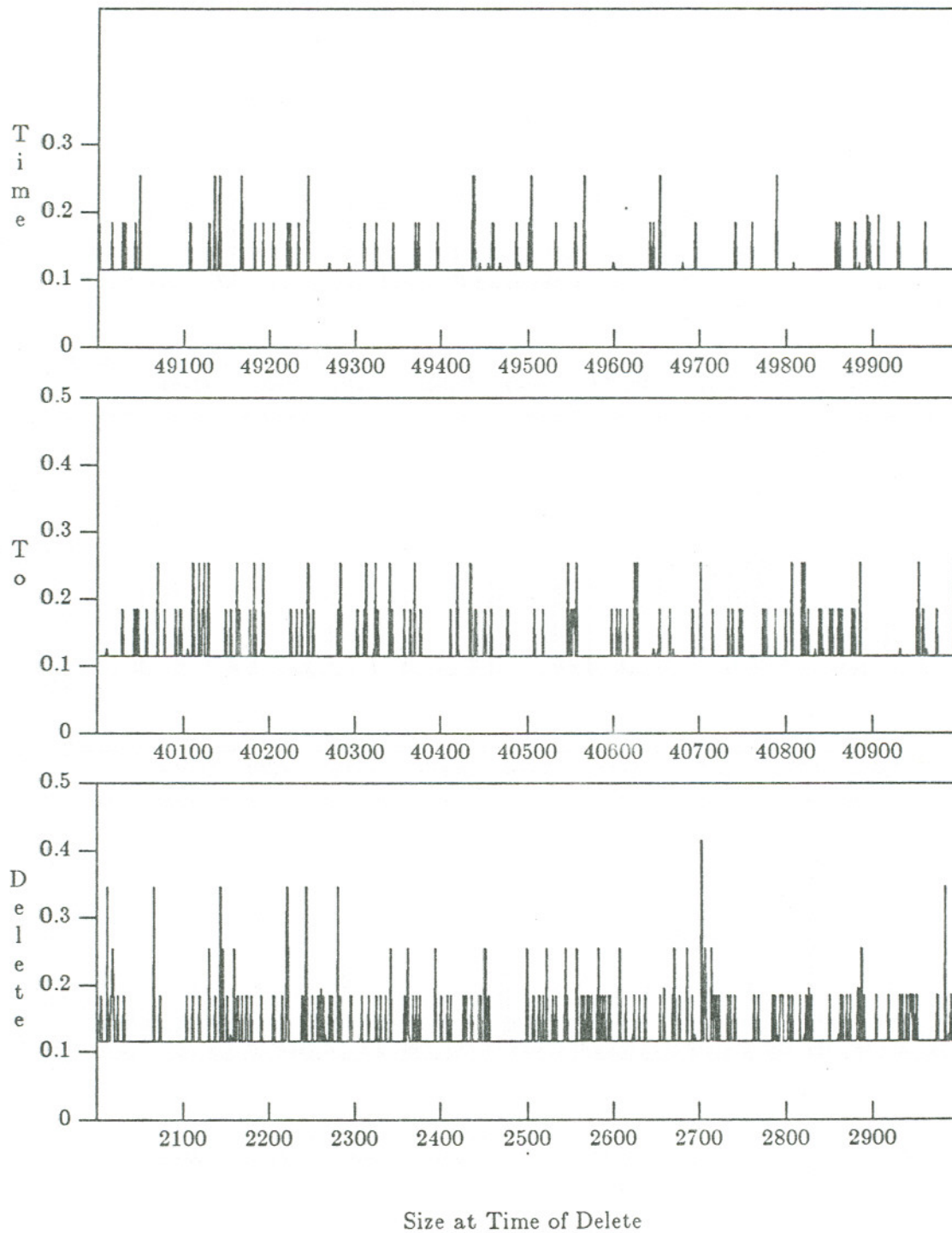


Figure 4.16: Deletion Time vs. Size of Tree for Gamma Distribution ($\alpha=2.0$, $\beta=1.0$)

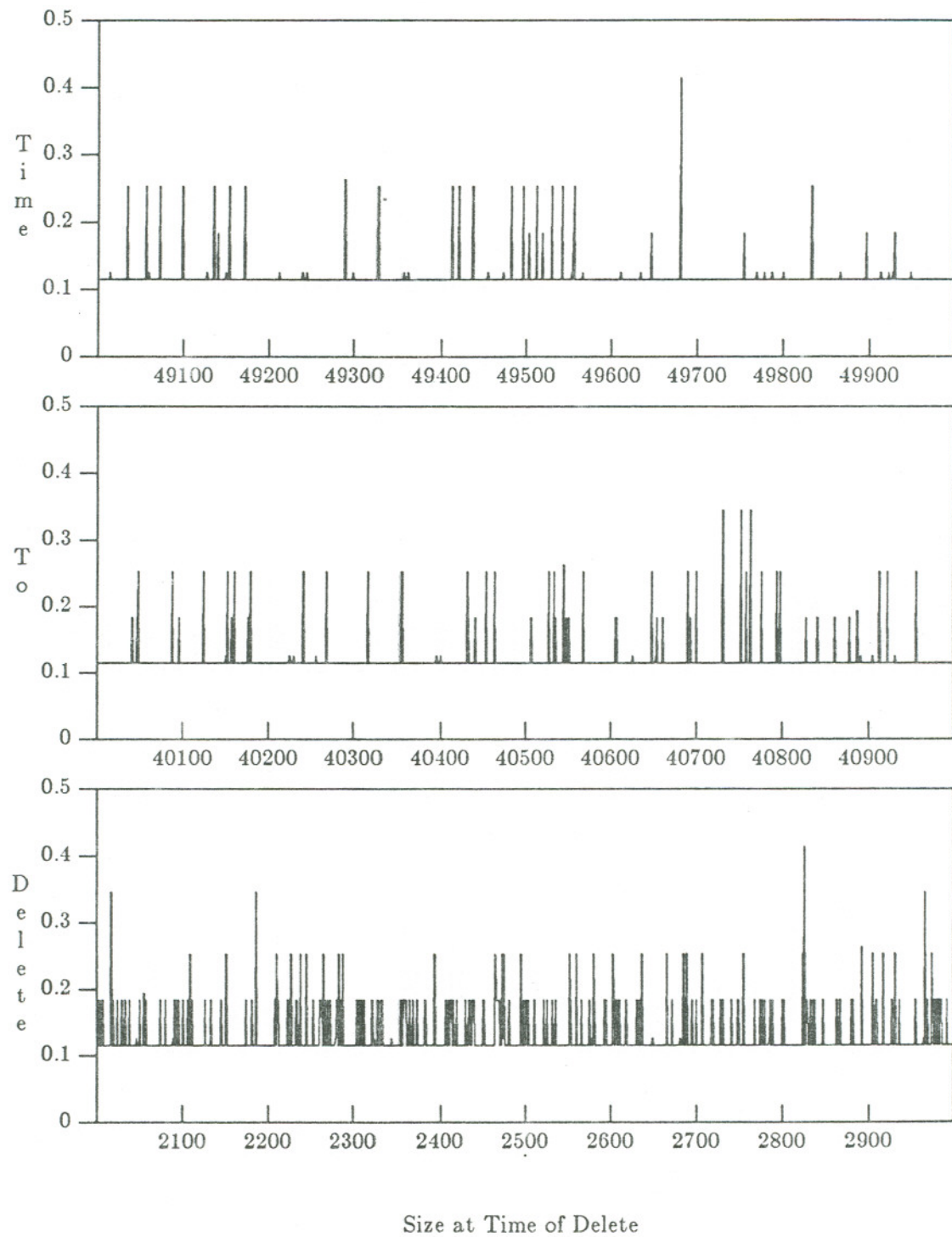


Figure 4.17: Deletion Time vs. Size of Tree for Beta Distribution ($\alpha=2.0$, $\beta=5.0$)

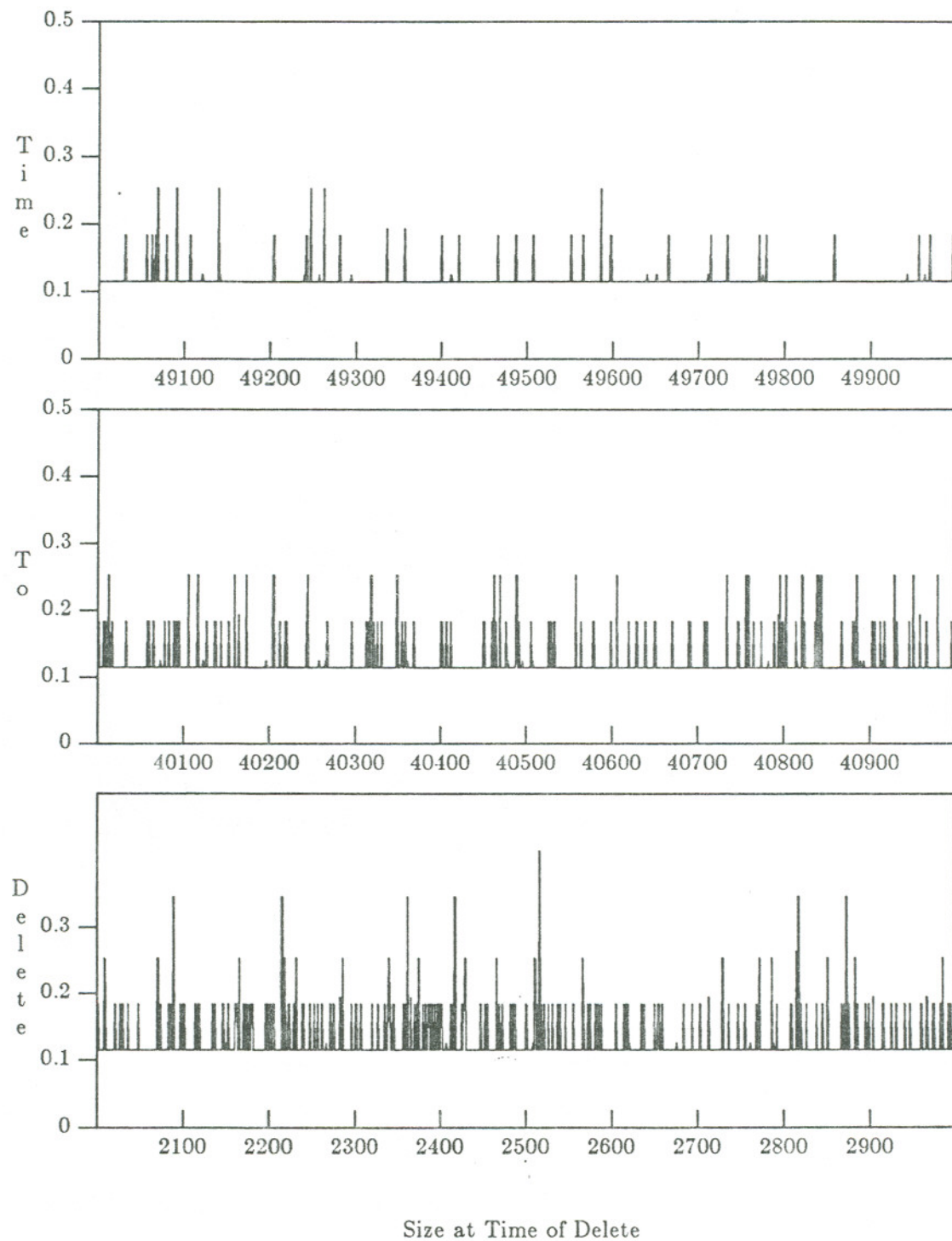


Figure 4.18: Deletion Time vs. Size of Tree for Log-normal Distribution ($\mu=1.0$, $\sigma=1.0$)

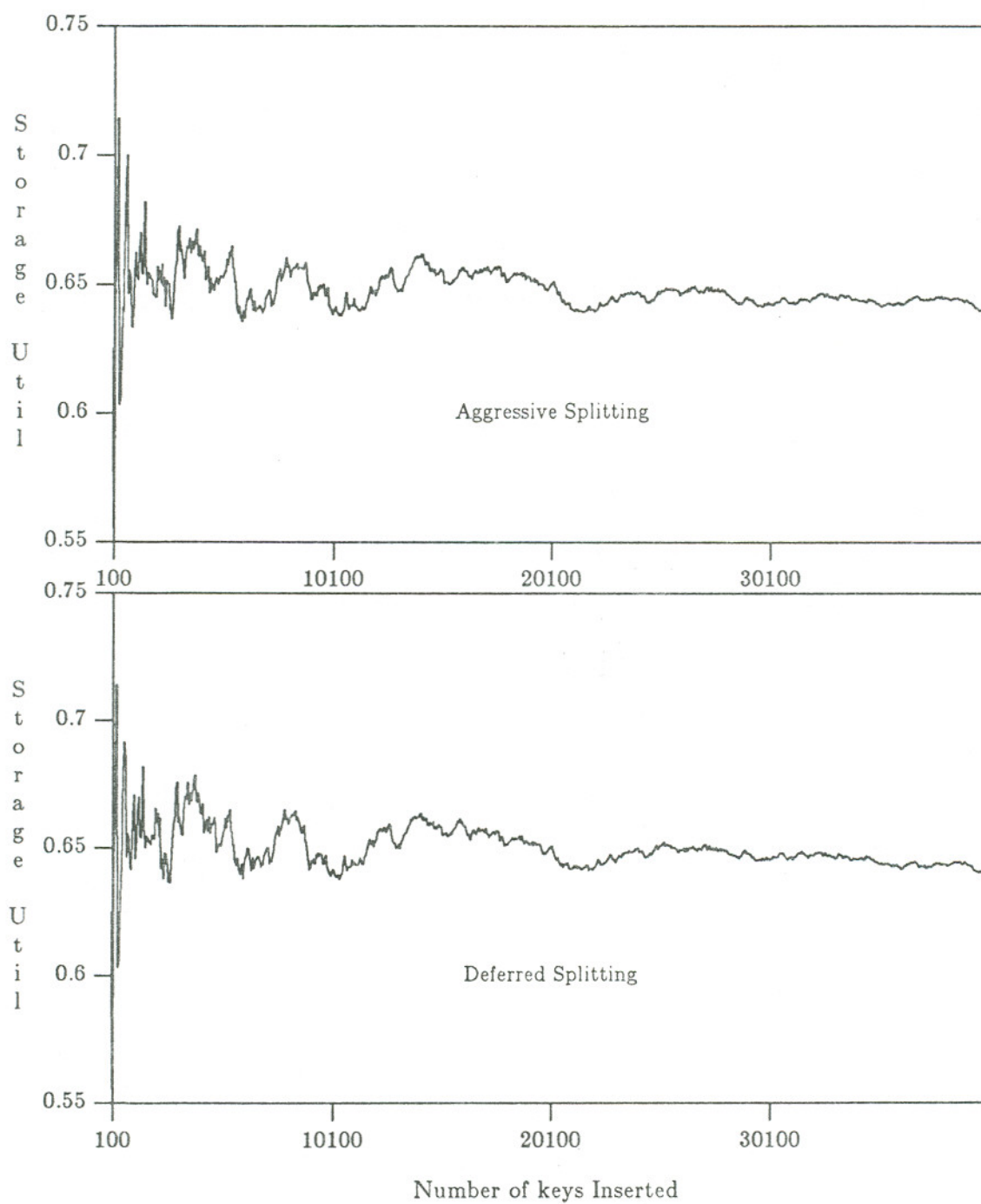


Figure 4.19: Average Storage Utilization for Uniform Distribution

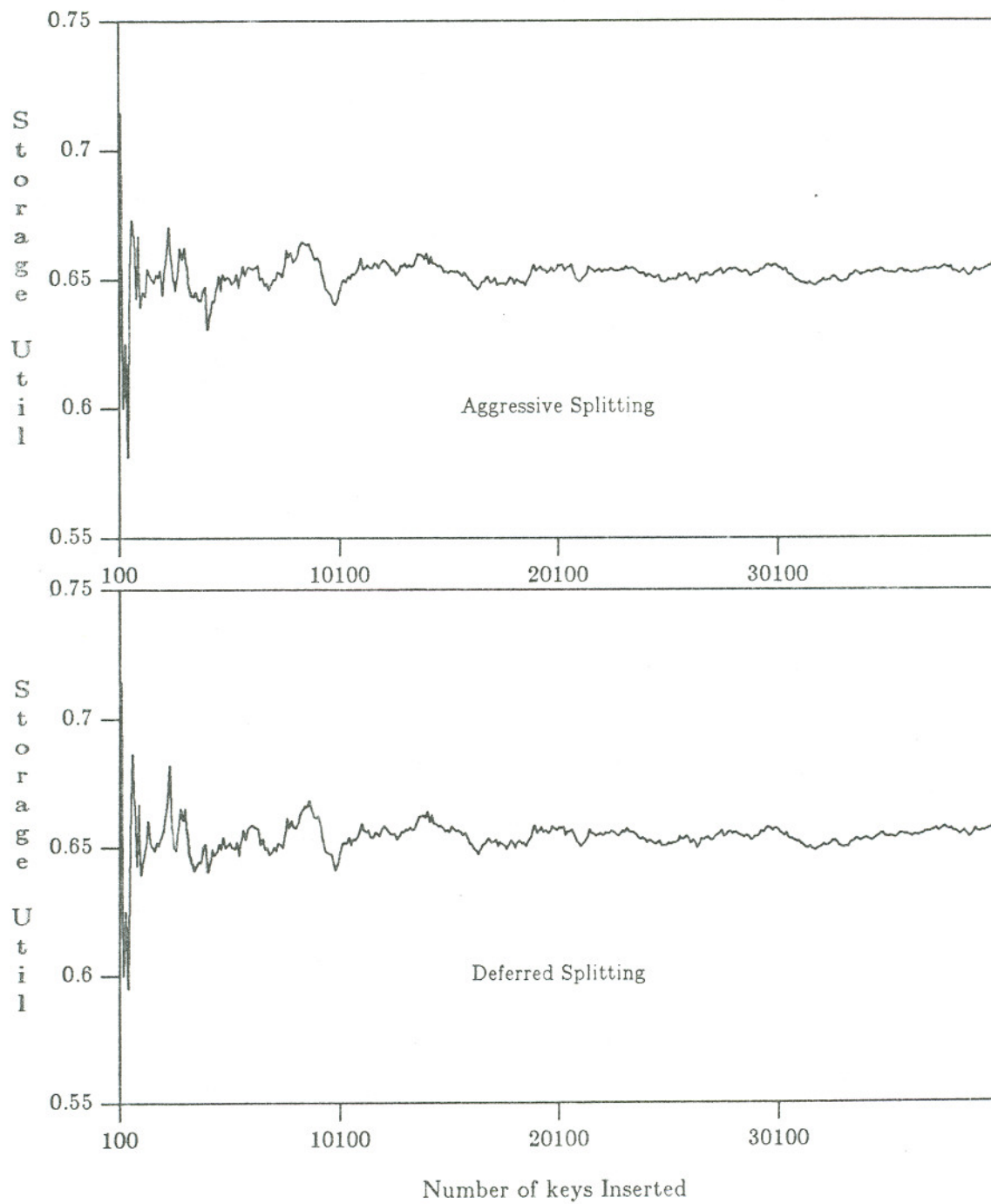


Figure 4.20: Average Storage Utilization for Normal Distribution ($\mu=1.0, \sigma=1.0$)

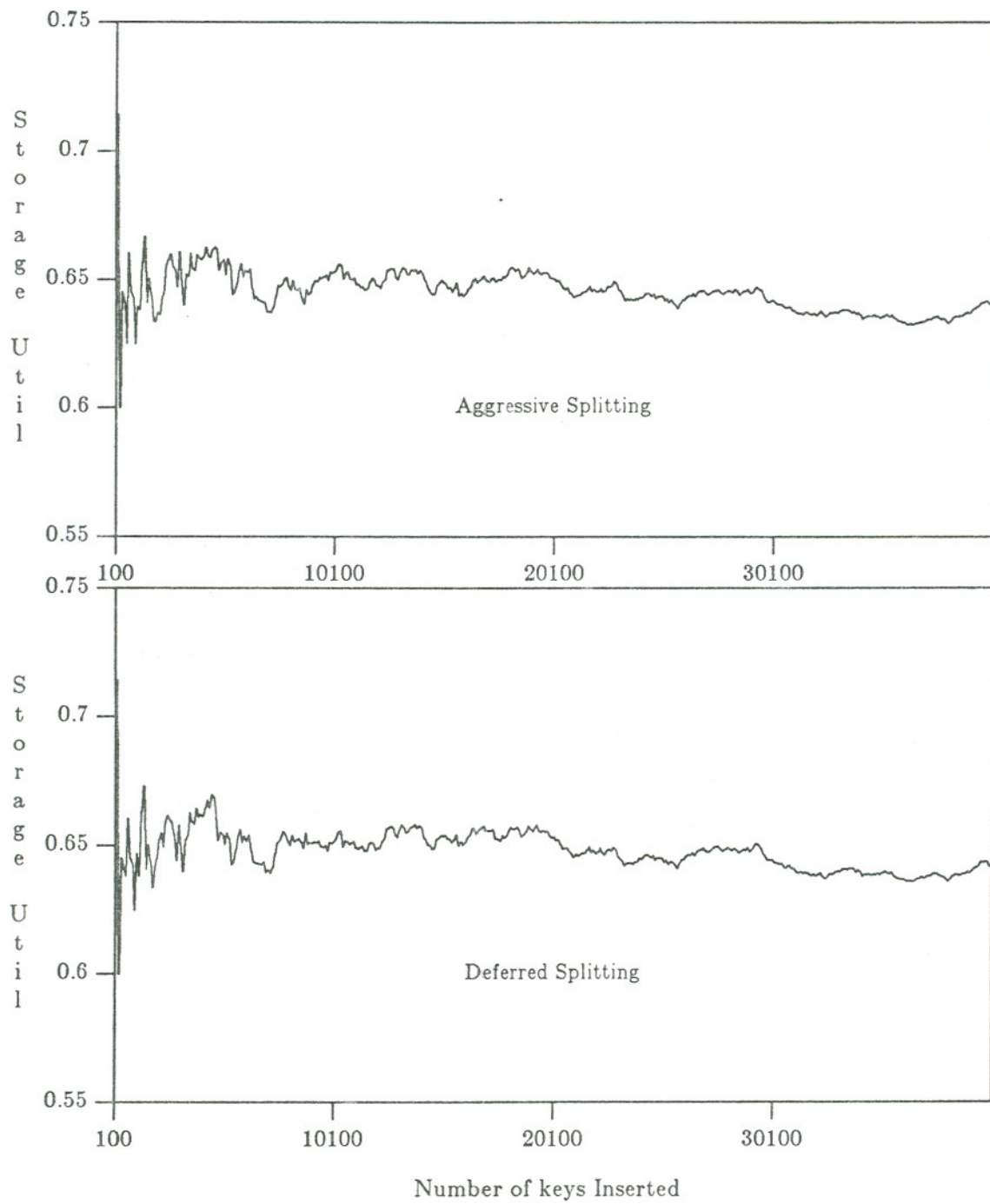


Figure 4.21: Average Storage Utilization for Exponential Distribution ($\mu=1.0$)

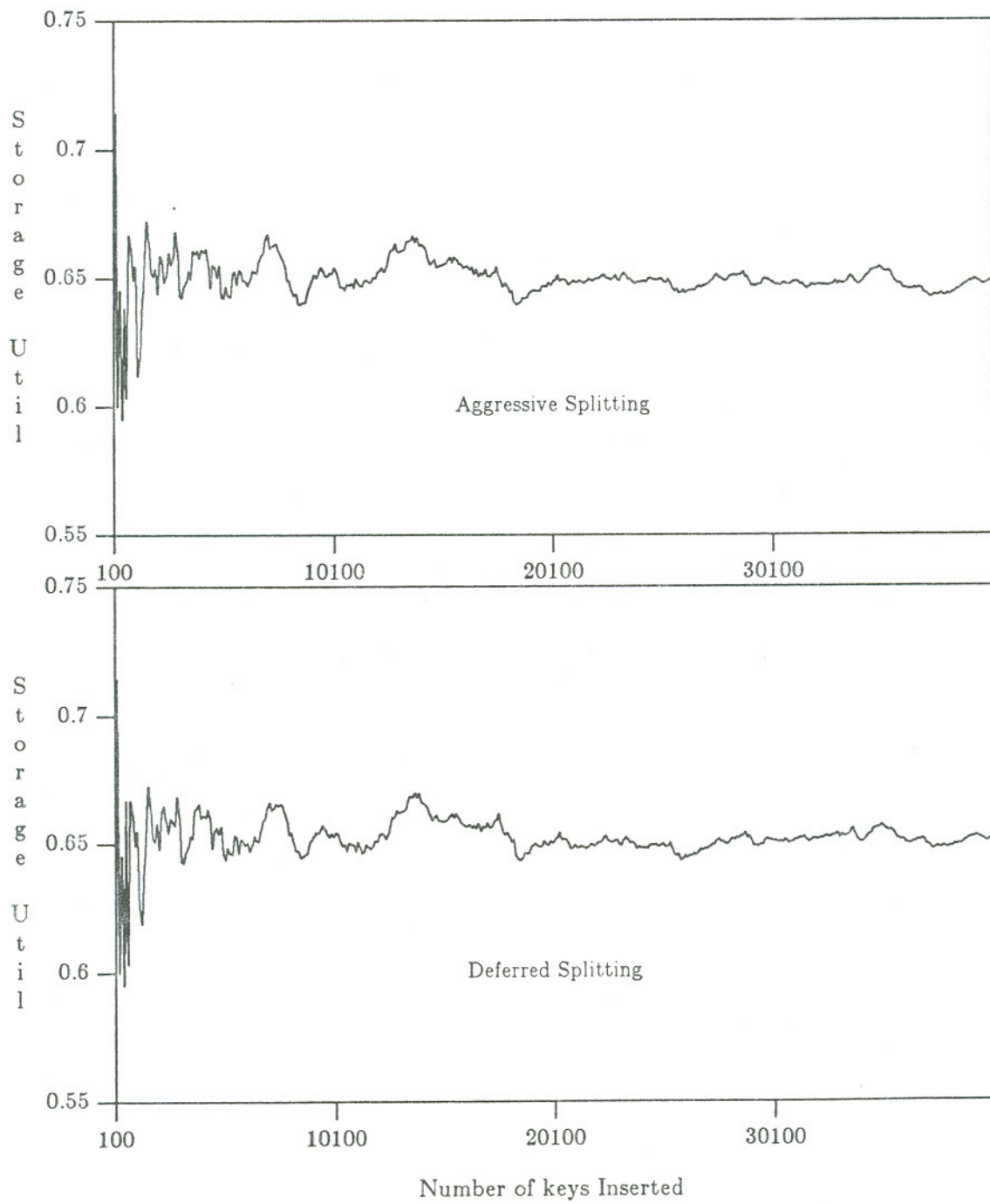


Figure 4.22: Average Storage Utilization for Gamma Distribution ($\alpha=2.0$, $\beta=1.0$)

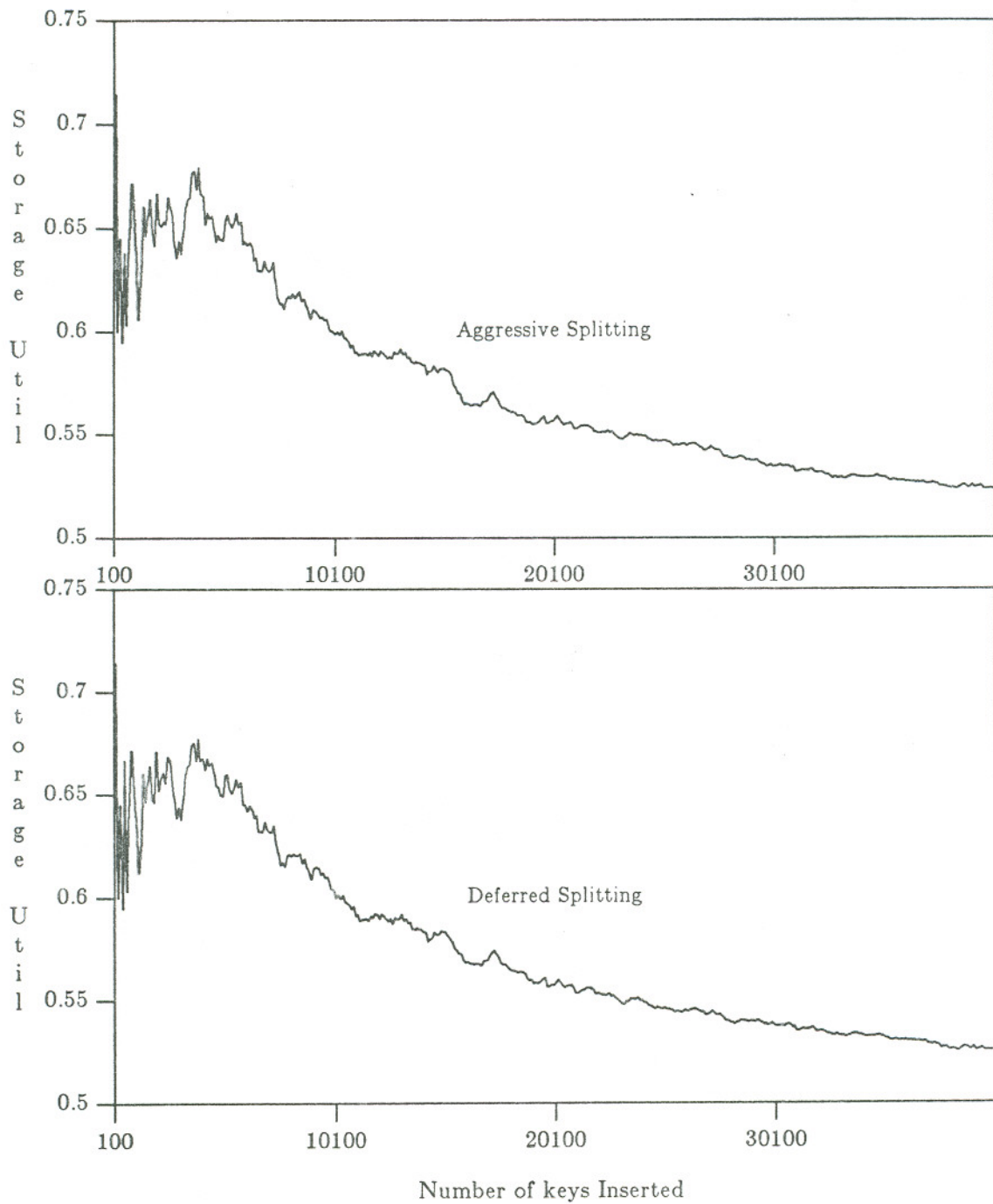


Figure 4.23: Average Storage Utilization for Beta Distribution ($\alpha=2.0$, $\beta=5.0$)

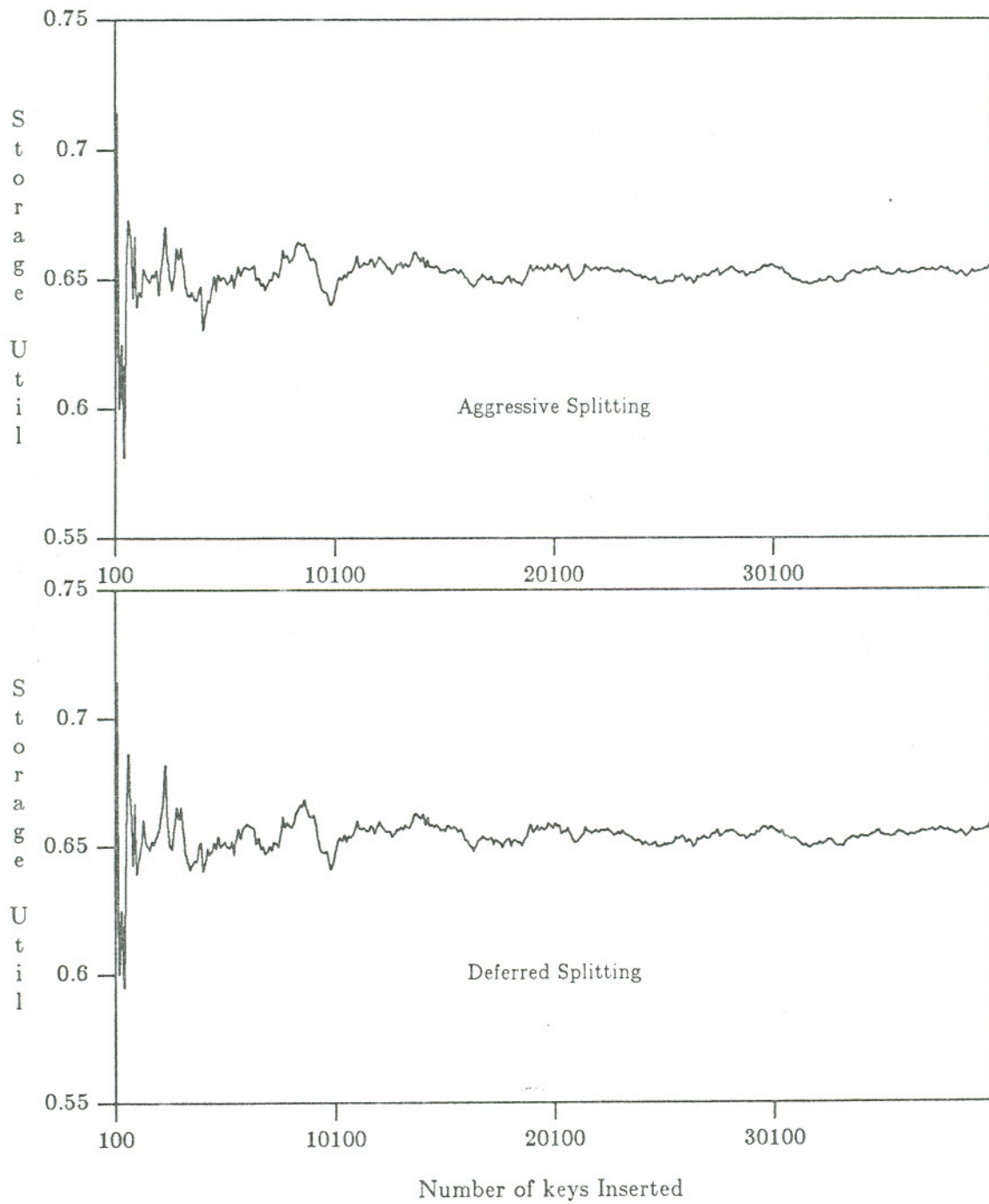


Figure 4.24: Average Storage Utilization for Log-normal Distribution ($\mu=1.0$, $\sigma=1.0$)

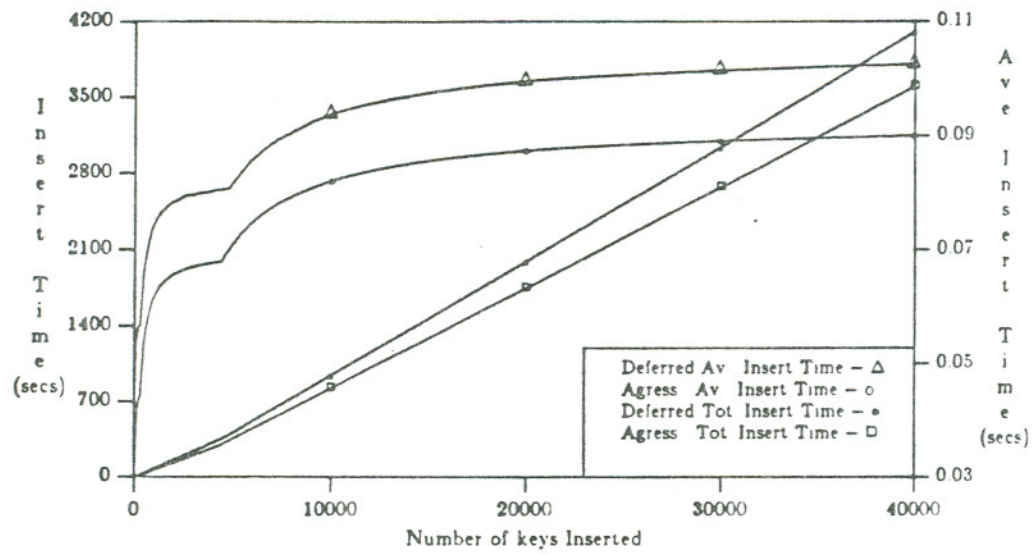


Figure 4.25: Insertion Times and Av. Insertion Time vs. Number of Keys for Uniform Distribution

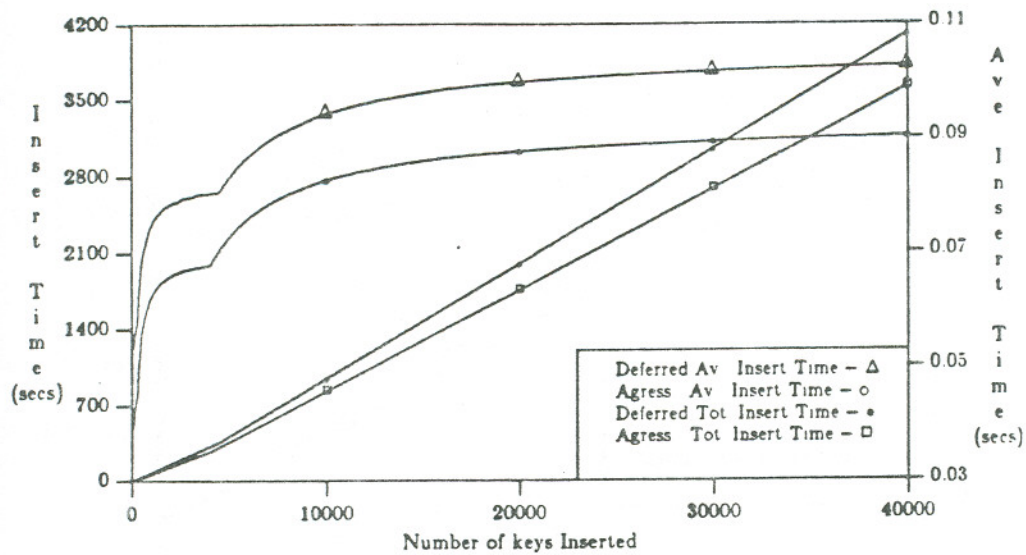


Figure 4.26: Insertion Times and Av. Insertion Time vs. Number of Keys for Normal Distribution

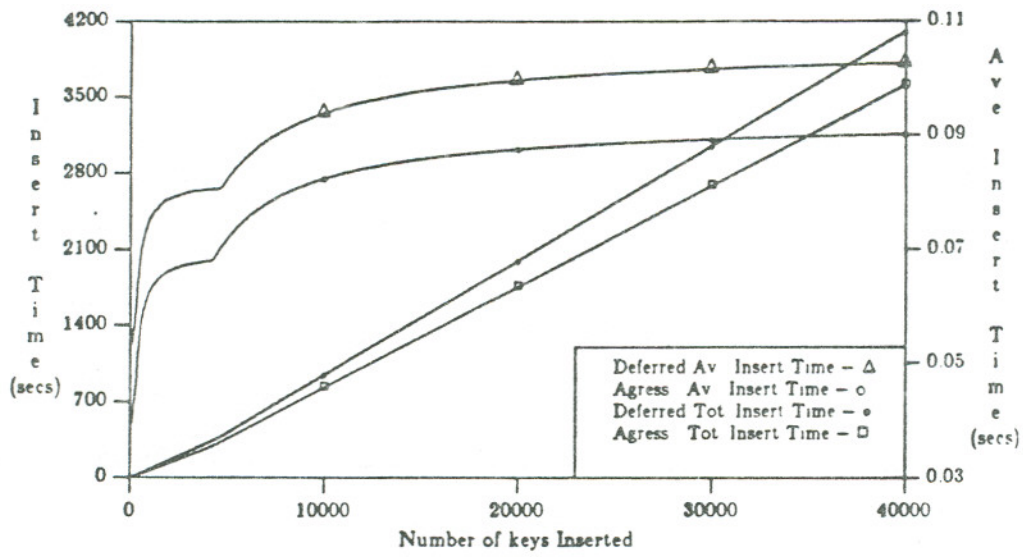


Figure 4.27: Insertion Times and Av. Insertion Time vs. Number of Keys for Exponential Distribution ($\mu=1.0$)

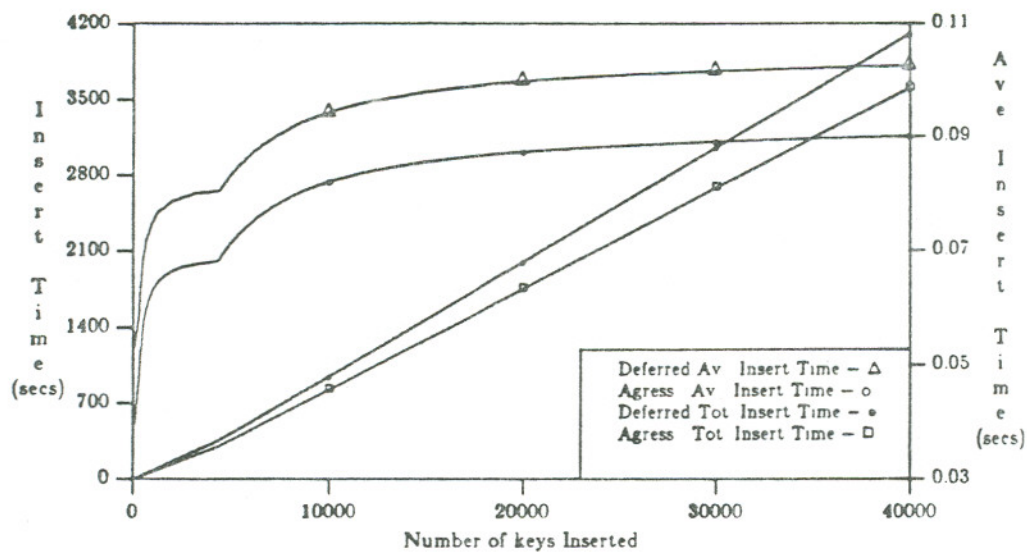


Figure 4.28: Insertion Times and Av. Insertion Time vs. Number of Keys for Gamma Distribution ($\alpha=2.0, \beta=1.0$)

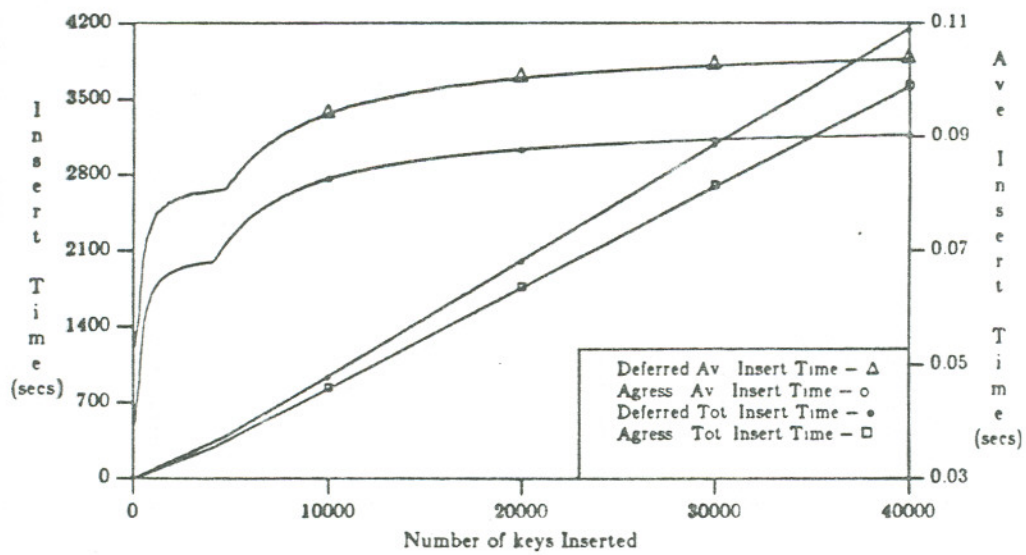


Figure 4.29: Insertion Times and Av. Insertion Time vs. Number of Keys for Beta Distribution ($\alpha=2.0$, $\beta=5.0$)

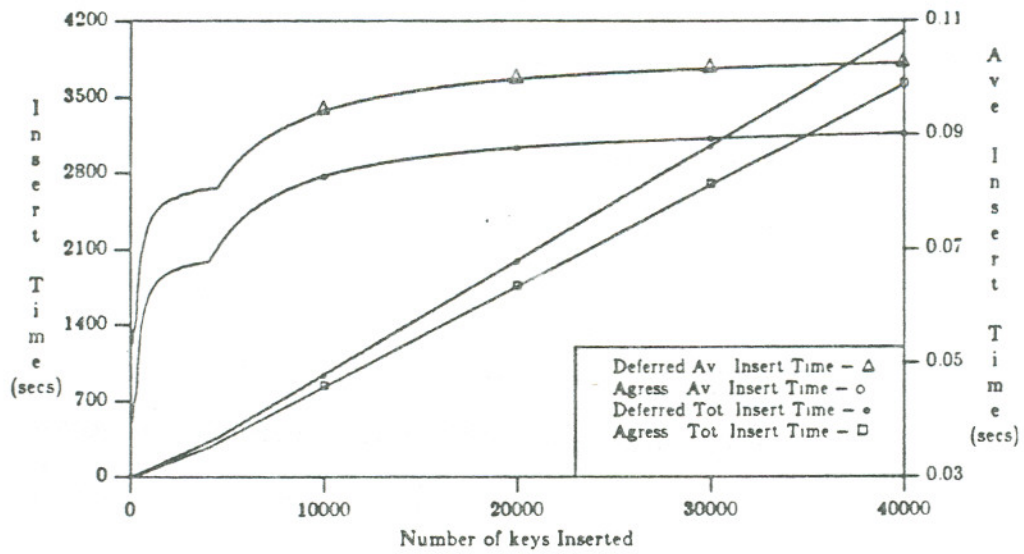


Figure 4.30: Insertion Times and Av. Insertion Time vs. Number of Keys for Log-normal Distribution ($\mu=1.0$, $\sigma=1.0$)

CHAPTER 5

Concurrency Control of B-Trees

5.1. Overview

As mentioned previously, the structure of data in B-trees can be exploited to provide enhanced concurrency. As an illustration of anomalies that can arise due to conflicts between processes concurrently operating on the same B-tree (or a variant of it), assume that two processes are acting concurrently on a B^+ -tree shown in Figure 5.1. One of the processes inserts key 35 and the other searches for key 36. If the steps of the two processes are interleaved in the manner given in Figure 5.2, then the search process makes a wrong conclusion about the existence of key 36 in the tree. Figure 5.3 shows the B^+ -tree after the execution of step 6. After the search process determines which leaf to examine, the insertion

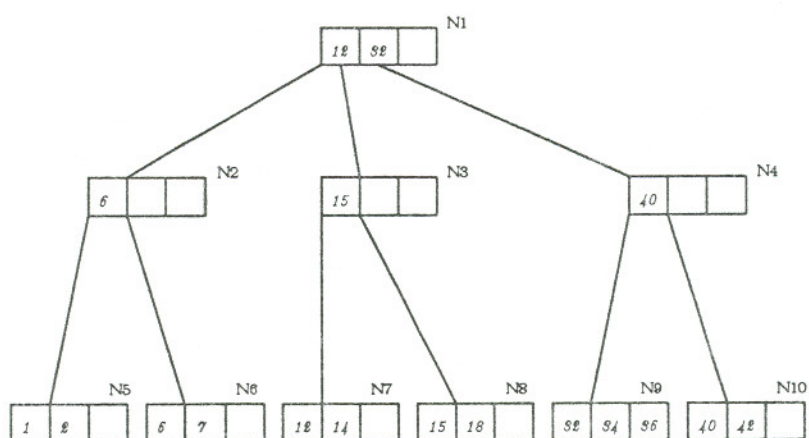
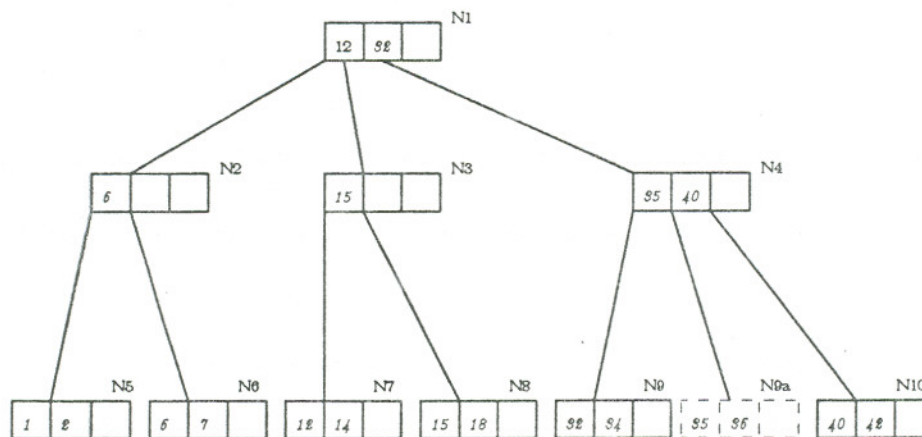


Figure 5.1: The B^+ -tree before both processes start

Step	Insert 35	Search 36
1	Determine which child of N1 to move to	
2	Determine which child of N4 to move to	
3		Determine which child of N1 to move to
4		Determine which child of N4 to move to
5	N9 is leaf and full	
6	Split N9, shift keys to N9a, insert 35	
7		N9 is leaf and does not contain 36

Figure 5.2: Interleaving of steps of the two processes

process — due to split of node N9 — moves key 36 elsewhere (to node N9a). Similar anomalies may be caused by conflicts between insertion and deletion processes and between deletion and search processes, necessitating the presence of concurrency control.

Figure 5.3: The B⁺-tree after step 6

The usual problems associated with any concurrency control protocol (deadlock, livelock, etc.) are present also in the case of B-trees. Moreover, since B-trees occur at a lower level of abstraction than the data itself, the concurrency control process is different. In conventional database systems, concurrency control is synchronous. That is, access to shared data is synchronized by a scheduler. In the case of B-trees, the process is asynchronous, in the sense that there is no physical scheduler present to schedule concurrent accesses to the same nodes. The enormity of the tree makes it impractical to have one scheduler to maintain livelock free access to thousands of nodes or to have a mini-scheduler for every node of the tree. Though the latter model has been proposed [KwW80b], it is yet to be implemented. There are other problems. The operations on the nodes of the tree are of a short duration and the processes are data intensive. This results in the lock table being a hot spot which drastically increases the response times. Consequently, the algorithms for different operations on B-trees need to be robust enough to ensure integrity of data.

In this chapter we survey the state of the art in concurrency control of B-trees and present three new concurrency control protocols both for traditional B^+ -trees and for variants of the B^+ -tree data structure.

5.2. Log-Sequence Serializability versus Operational Serializability

Recall from the discussions in Chapter 2 that a log is a sequence of operations performed on the database. A log is *serializable* if and only if it is equivalent to a serial execution. The execution of a set of transactions is correct if it is serializable. This view of correctness hides an important aspect of database systems [BGL83]: the semantic incoherency between the user and the system operations. What the user may see as a single operation may be a set of operations to the system. Such a semantic difference necessitates a more elaborate criterion of correctness [BGL83].

Log-sequence serializability is based on a totally syntactic criterion of preserving conflicts. In situations where semantic information about the abstract data structure is important, such a criterion of serializability is insufficient. The following example illustrates the difference. Consider the following C++ class structure:

```
class library {
    char (*arrayptr1)[];
    char (*arrayptr2)[];
    int  insert(arrayptr, key);
    int  search(arrayptr, key);
    int  delete(arrayptr, key);
    int  move(arrayptr1, arrayptr2);
    char* whereIs(key);
};
```

where the methods `insert`, `delete` and `search` insert, delete and search for the second argument in the array pointed to by the first argument respectively. Method `move` moves the contents of the array pointed to by `arrayptr1` to the one pointed to by `arrayptr2`. Method `whereIs` returns a pointer to the array that contains the second argument. Also let unequal elements be considered only. Let two processes T_1 and T_2 access an instance `aInst` of the class `library`. T_1 sends message `whereIs(x)` to `aInst` and gets the array name `arrayptr` where `x` resides. T_2 then sends the message `moveall(arrayptr1, arrayptr2)` to `aInst` and subsequently makes `arrayptr1` point to what `arrayptr2` points to. Unknowing of the message sent by T_2 , T_1 sends the message `delete(arrayptr1, x)` to `aInst`. Fortunately, due to proper sequencing, the deletion is successful. Though the sequence of operations is not log-sequence serializable, T_1 achieves what it intended. The anomaly is due to the semantics of the underlying data structure. If the first argument of the `delete` method was not a pointer to an array, but an array itself, then the execution would have been erroneous. Such sequences of operations which preserve a user-operation specific definition of correctness are called *operationally serializable*.

In our context, the log-sequences for an operation refer to that of the access and the writing of the nodes of the B-tree. As proved by an example in Section 5.3.3.2, log-sequence serializability is not satisfied by many of the B-tree concurrency control algorithms. However, at a higher level of abstraction, if we consider the correctness criterion to be operational serializability, then the algorithms are correct. To avoid confusions, we re-define correctness criterion for these protocols in Section 5.3.

5.3. Concurrency Control Algorithms

Next, we define some standard terminology in the theory of concurrency control of B-trees. Recall that trees in which the structural changes propagate from the root to the leaves, aggressively splitting or merging nodes, are called *top-down trees*. Trees in which the structural changes propagate from the leaves to the root employing deferred splitting or deferred merging of nodes, are called *bottom-up trees*. First, we consider concurrency control of bottom-up trees. Later, in Section 5.3.2, we consider concurrency control of top-down trees.

Consider a B-tree of order d as defined in Chapter 3.

DEFINITION 5.1: A node is *insertion safe* if it contains less than $2d$ keys.
□

DEFINITION 5.2: A node is *deletion safe* if it contains more than d keys.
□

A node is called *safe* where the operation is implicit.

A process performs operations on B-trees. A process that performs a search operation is called a *read-only* or *read* process and a process which performs an update operation is called an *updater* process. We define a partial ordering \sqsubseteq on the nodes of a tree such that, for any two nodes x and y , $x \sqsubseteq y$, if and only if x appears in the path from the root to y .

One process P is said to overtake another process Q , if for two nodes x and y , which both processes access in that order and $x \sqsubseteq y$ or $y \sqsubseteq x$, Q accesses x earlier in time than P but accesses y later in time than P . Read-only processes do not violate the structure of the tree even if processes overtake one another. However, the structure of the tree will not necessarily be preserved or the processes may not successfully complete their intended operations if updating processes overtake one another or other read-only processes.

DEFINITION 5.3: An updating process P for key k is *correct* when the following hold:

- a) If k is in the tree and there are no other deletion processes for k , then P must delete the key.
- b) If k is not in the tree and there are no other insertion processes for k , then P must insert the key.
- c) If k is in the tree and there are no other insertion processes for k , then P must delete the key.
- d) If k is not in the tree and there are no other deletion processes for k , then P must insert the key.
- e) At any time (even during an operation), if the tree is frozen, then an in-order traversal of the tree generates the nodes of the tree in sorted order (see [Knu73]).

□

If only one process at a time is allowed in the tree then it is guaranteed to be correct. However, allowing only one process at one time provides no concurrency.

DEFINITION 5.4: The *deepest safe node* for an updating process is defined to be the lowest (farthest from the root) safe node in the path from the root to the node (or leaf, in the case of B^+ -trees) into which the key is to be inserted or deleted.

□

DEFINITION 5.5: The *scope* of an updating process is the subtree rooted at the deepest safe node for the updating process.

□

Note that the effect of the update propagates from the node into which the insertion or the deletion is made, up to the deepest safe node and not above that. Also note that the last two definitions are applicable only to bottom-up B-trees.

5.3.1. Concurrency Control of Bottom-Up B-Trees

Concurrency control algorithms for bottom-up B-trees can be categorized into two broad classes, namely *type 1* and *type 2* [KwW82].

5.3.1.1. Type 1 Algorithms

In *type 1* algorithms, no updator is allowed in the scope of another updator. We discuss the most common type 1 algorithms that have been reported in literature.

5.3.1.1.1. Samadi-Parr Algorithm

The first algorithm for concurrent manipulation of B-trees was proposed by Samadi in 1976 [Sam76] and independently by Parr in 1977 [Par77]. In this algorithm, the authors use only one type of lock. The lock is incompatible with itself. The algorithm is easily extendible to B^+ -trees.

Each node can be locked exclusively by only one process and only two lock operations (*lock* and *unlock*) are available. The reader processes proceed down the tree in a *lock-step* manner. A child node is always locked before its parent is unlocked. Locking nodes in such a manner is also called *lock-coupling*. An updator process starts from the root and travels down the tree locking nodes one at a time and stacking the locked nodes. On reaching a safe node, the updator process unlocks all the locked nodes and pops them out of the stack. Thus, an updating process locks its scope exclusively and even reader processes cannot exist in that scope. Thus, the algorithm induces a total ordering on all readers sharing a common path and there is no possibility of overtaking.

To take care of the boundary condition, in which the root is split and the height of the tree increases, the existence of a dummy node d is assumed. For all nodes x , in the tree, $d \sqsubseteq x$. When the root is split by any process, the dummy node must have been the deepest

safe node for the process and there cannot be any processes waiting to lock the root. Hence, splitting the root does not divert any process along a wrong path. This implementation is particularly useful in a system like UNIX, where maintaining a lock table may be expensive.

5.3.1.1.2. Bayer and Schkolnick's Algorithms

Bayer and Schkolnick provide three algorithms [BaS77]. The first algorithm is similar to that of Samadi and Parr except that the authors used two types of locks (*read* and *exclusive*) instead of one.

Reader processes proceed down the tree, read-locking the nodes in a lock-step fashion according to the compatibility matrix given in Figure 5.4. Therefore, more than one reader process can co-exist at a particular node. Updator processes obtain exclusive locks while traveling down the tree, releasing all previously held locks when a safe node is reached. Thus, when the updator process reaches the node where the update takes place, the scope of the process is exclusively locked. Also, a total order is imposed on updator processes sharing a common path and updators drive away reader processes by exclusively locking their scopes. This algorithm is easily extendible to B^+ -trees.

In algorithm two, like the reader processes, the updator processes also read-lock the nodes on their way down to the appropriate node. However, the node into which the

	<i>read</i>	<i>excl</i>
<i>read</i>	<i>y</i>	<i>n</i>
<i>excl</i>	<i>n</i>	<i>n</i>

Figure 5.4: Compatibility Matrix for Bayer and Schkolnick's Algorithm 1

physical update is to take place is locked exclusively. If the leaf is not safe, all locks are released and the update process repeated with algorithm one. The algorithm depends on the fact that node splitting is relatively rare. For trees of low order, this algorithm is inefficient because the probability of split of a node is higher. Note that, if the updator process tried to traverse bottom-up from the leaf, exclusively locking the nodes up the tree in a lock step manner instead of repeating the entire process, deadlock could occur.

In algorithm three, three types of locks are used: *read*, *write* and *exclusive*. The compatibility matrix of the three types of locks is given in Figure 5.5. A write-lock held by a particular process can be converted into an exclusive lock held by the same process. Thus, write-locks are a kind of intention lock. Reader processes proceed down the tree, read-locking the nodes exactly as in algorithm one. Updator processes, instead of exclusively locking the scope, write-lock the scope on the way down to the appropriate leaf. The leaf, however is exclusively locked. Thus, no other updators, but many reader processes may co-exist in the scope of an updator. If the node, into which the update is physically made is unsafe, then the change is propagated bottom-up, converting the write-locks into exclusive locks on the way up. Note that the convertibility of write-locks into exclusive locks avoids the possibility of any deadlock, although there is no total ordering on all processes accessing

	<i>read</i>	<i>write</i>	<i>excl</i>
<i>read</i>	<i>y</i>	<i>y</i>	<i>n</i>
<i>write</i>	<i>y</i>	<i>n</i>	<i>n</i>
<i>excl</i>	<i>n</i>	<i>n</i>	<i>n</i>

Figure 5.5: Compatibility Matrix for Bayer and Schkolnick's Third Algorithm

a particular node.

5.3.1.1.3. Ellis's Algorithm

The algorithm provided by Ellis was originally for B^+ -trees and is a variant of the third algorithm of Bayer and Schkolnick [Ell80]. The proposed concurrency control protocol handles only the case of insertions and searches. Pure readers are allowed in the scope of an updatator both when it searches top-down to the appropriate leaf and when it restructures the tree bottom-up. During restructuring, the write-lock of the parent node of the node being updated is converted into an exclusive lock. This lock conversion into an exclusive lock disallows a search process to be misdirected since that can take place only at the parent of the node being currently updated. Moreover, since reader and updatator processes can co-exist at a node, the reading of a node and the update of the node are done in opposite order — a method proposed originally in [Lam77]. Thus, the atomicity of updates is the writing of a key-pointer pair.

In this algorithm, reading of a node is done from left to right and the shifting of keys done from right to left (in our notation, the key K_i being shifted before the corresponding pointer p_{i+1}) in such a way as to ensure that a search process will not be misdirected by another process simultaneously updating the node.

5.3.1.1.4. Kwong and Wood's Algorithm

Kwong and Wood [KwW82] improved Bayer and Schkolnick's and Ellis's algorithms by introducing *side-branching* — a technique analogous to the multi-version concurrency control schemes in general database systems. Their algorithm uses the three types of locks as in the third algorithm of Bayer and Schkolnick.

Consider the insertion of a key into a B-tree (or a B^+ -tree) that causes one or more splits. During bottom-up restructuring, the restructuring process is carried out in two passes. In the first pass, the process travels from the leaf along the appropriate path to the deepest safe node, creating and filling up new nodes from their old counterparts. This results in the formation of a separate sub-path which is called a *branch*. The branch formation goes on until the deepest safe node of the process is reached. The old nodes (i.e., the ones to be split) are otherwise kept intact. On reaching the deepest safe node, the branch which is thereby created is added atomically to the deepest safe node, which is exclusively locked by lock coupling as in the case of the third algorithm of Bayer and Schkolnick. In the second pass, the path from the deepest safe node to the appropriate leaf (which was traversed during the searching phase) is scanned once more and the filled nodes appropriately halved. Deletions are done analogously.

The protocol also employs the technique of reading and writing in opposite directions (as suggested by Lamport in [Lam77]) to enhance concurrency. Note that extra concurrency is provided over the third algorithm of Bayer and Schkolnick by not converting the write-locks of unsafe nodes during the first phase of restructuring. However, each process takes a longer time to complete.

5.3.1.2. Type 2 Algorithms

The basic idea of type two algorithms is to allow other updaters within the scope of an updater. This is achieved by locking only a bounded number of nodes within the scope of the process. In type 2 algorithms, the deepest safe nodes lose their significance. Concurrency control protocols for top-down B-trees also lock only a fixed number of nodes during updates, but they are discussed later. Next, we discuss the most common type 2 algorithms.

5.3.1.2.1. Ellis's Algorithm

Ellis [Ell80] provided an algorithm for concurrent searches and inserts into B-trees. She modifies the B-tree data structure slightly by changing the subtrees of leaves into a chain of leaves. The tree structure above the leaves is not changed.

Updaters proceed down the tree like normal readers. Since readers within the scope of the writers are not supposed to find the keys which are just being inserted, this does not violate the correctness of the protocol. However, the updaters must be able to find the keys inserted by other updaters within their scope. To achieve that goal, another type of lock called a *write'* lock was introduced. Figure 5.6 shows the lock compatibility matrix. Note that the write-locks alone were inadequate to guarantee correctness since they did not prevent concurrent insertions at the same positions. Moreover, if the same type of lock (i.e., write-lock) is used both during the insertion phase and during the bottom-up restructuring phase of an insertion, then there is a possibility of deadlock.

Pure readers and the reading phase of updaters use lock-coupling with read-locks on their way down the tree. On reaching the parent of the appropriate leaf (recall that the leaves are now chained into a linked list, the head of which is the parent), it write'-locks the

	<i>read</i>	<i>write</i>	<i>write'</i>	<i>excl</i>
<i>read</i>	<i>y</i>	<i>y</i>	<i>y</i>	<i>n</i>
<i>write</i>	<i>y</i>	<i>y</i>	<i>n</i>	<i>n</i>
<i>write'</i>	<i>y</i>	<i>y</i>	<i>n</i>	<i>n</i>
<i>excl</i>	<i>n</i>	<i>n</i>	<i>n</i>	<i>n</i>

Figure 5.6: Compatibility Matrix for Ellis's Type 2 Algorithm

node and makes the insertion. The process then releases all the locks it holds. Next, during the restructuring process, the writer travels bottom-up coupling write-locks with exclusive-locks and changing the contents of the exclusively locked node appropriately.

Concurrency is enhanced over the type 1 algorithms in three ways. First, other updators are allowed in the scope of an updator. Second, a concept analogous to pipelining, applied to the leaves, allows more than one updator at the parent of a leaf. Third, the total restructuring of the tree is delegated to a later stage. The last observation follows from the following example. Consider a node x immediately above the leaf level. By the definition of the tree structure, x will have a chain of children leaves. Consider two processes P and Q arriving at the node x for insertion of two keys and that the number of children of node x is $2d-1$. If both process arrive simultaneously, then both will be allowed to add a child without splitting x . After this, when another process R appears, it sees that the number of children of x is $2d+1$. It then splits x and makes the insertion in the appropriate child of x .

However, the protocol has some inherent drawbacks that have hindered its popularity. First, the leaves that are children of a particular internal node need to be traversed sequentially, resulting in increased access times. Second, the tree structure is destroyed at the leaf level, which is unacceptable in many database applications. Note that, the change in tree structure closely resembles overflow chaining in the ISAM file organization. It has been reported in the literature that there is an algorithm for deletions, provided that some modifications are made in the insertion algorithm [Sag86]. But, only one deletion process can be allowed to run in parallel with multiple reader and insertion processes.

5.3.1.2.2. Miller and Snyder's Algorithm

Three types of locks, namely, *a*-locks (access), *p*-locks (pioneer) and *f*-locks (follower) are used in an algorithm described by Miller and Snyder. The compatibility matrix of the locks is given in Figure 5.7.

The algorithm uses manipulation of a queue associated with every node. Since in our model of concurrency control we do not have queues associated with nodes, we shall avoid discussing the queue manipulation aspects of the protocol.

Reader processes search top-down the tree locking nodes with *a*-locks but without coupling the locks. That is, the processes release held locks before accessing another lock. Updater processes use lock-coupling with *p*-locks. As a consequence of lock-coupling, readers need not be driven away from the scope of the updater. To avoid updater processes from colliding at the same node, asymmetric locks are used. When two updaters (U_1, U_2) collide, one of them (U_1) will *p*-lock the common ancestor while the other (U_2) must wait. However U_1 might need to lock some other nodes, already held by U_2 . In that case *f*-locks are used, to avoid deadlocks.

	<i>a</i>	<i>p</i>	<i>f</i>
<i>a</i>	<i>y</i>	<i>n</i>	<i>n</i>
<i>p</i>	<i>n</i>	<i>n</i>	<i>n</i>
<i>f</i>	<i>n</i>	<i>y</i>	<i>n</i>

Figure 5.7: Compatibility Matrix for Miller and Snyder's Algorithm

This algorithm has an unacceptable drawback. For B-trees of large orders, the number of locks held by an updater process is often more than the number of locks that would have been necessary if type 1 algorithms were used. However, the technique of *decoupling* of locks by reader processes and the use of *asymmetric* locks, as introduced in this scheme, was later adopted by many authors [KwW82].

5.3.1.2.3. Lehman and Yao's Algorithm

This is one of the most efficient algorithms provided among all type 2 algorithms. Only one type of lock is used in the protocol, but it, only handles searches and insertions [LeY81]. The B-tree data structure was modified into the B_{link} -tree, providing two paths to nearly all nodes in the tree. The trick employed was to search along another path while the natural path was being modified as a consequence of an update common to the path. Only updater processes locked nodes while reader processes proceeded freely down the tree. This scheme provided the best results among all protocols as discussed in the next chapter. Moreover, at one time, at most three nodes are locked, independent of the order of the tree. The occasion when three nodes are locked is also relatively rare (see [LeY81]). Since Algorithm 2 of the author closely follows this, we discuss details of this algorithm later.

5.3.1.2.4. Kwong and Wood's Algorithm

Kwong and Wood provides an algorithm [KwW79], that was a modification of the algorithm by Miller and Snyder. Asymmetric locking was used with repeated reading of nodes to avoid exclusive locking. The lock compatibility matrix is given in the Figure 5.8. The protocol also employs lock conversions from write to r_w locks. The algorithm is deadlock free but has potential problems of livelock. Moreover, the protocol only handles insertions and searches.

	r_p	r_u	w	e
r_p	y	y	y	n
r_u	y	n	n	n
w	y	y	n	y
e	n	n	y	n

Figure 5.8: Compatibility Matrix for Kwong and Wood's Algorithm

5.3.1.2.5. Sagiv's Algorithm

Sagiv provides an algorithm that is an improvement over the algorithm by Lehman and Yao [Sag86]. He includes boundary conditions, which were non-trivial and ignored by Lehman and Yao in their original paper. An inherent drawback of the Lehman and Yao protocol was that deletion processes wasted space by deleting the key only from the leaf level and not propagating any restructuring bottom-up. This resulted not only in wasting space by the existence of half-filled nodes, but also in the existence of redundant keys in the tree. To get rid of this problem, Sagiv used the concept of *compression processes* that merged less-than-half-filled nodes of a particular level. Note that such merging is possible only because the data structure was a B_{link} -tree. If the number of nodes in a particular level is odd, then the compression process would leave behind one node uncompressed. Also, the compression process only merges children of the same parent. Thus, in order to compress a tree from which all the nodes have been deleted, it takes $\log_2(n)$ passes of the compression process over the entire tree. Multiple compression processes can run at the same time, although it reduces concurrency since the compression process locks more nodes (usually three at a time) than the insertion (mostly two, rarely three), reader (none), or deletion (one) processes.

5.3.2. Protocols for Top-Down Trees

Few protocols for top-down trees have been proposed in literature. Since the restructuring of the nodes take place in a top-down manner, and since each node along the path from root to the leaves is traveled exactly once, the cost of insertions and deletions, when main memory size is fixed, is equal to or lower than bottom-up trees. However, as proven in Chapter 3, the number of splits or mergings during an update is more in the case of top-down trees. Thus, we have conflicting choices. On one hand more splits occur but on the other hand fewer nodes are locked during an update. The reduced amount of locking coupled with the low cost of insertion have resulted in the growing popularity of top-down B-trees over bottom-up B-trees in recent database systems (e.g., [Gra89], [CDK85]).

We explore the possibilities of improving concurrency using top-down tree locking protocols in this section. We give two new protocols, the first one of which does not provide overtaking of updater processes, though the second one does.

5.3.2.1. Algorithm One

The first algorithm is analogous to the first solution of Bayer and Schkolnick. It uses two types of locks whose compatibility matrix is given in Figure 5.9.

	<i>r</i>	<i>e</i>
<i>r</i>	<i>y</i>	<i>n</i>
<i>e</i>	<i>n</i>	<i>n</i>

Figure 5.9: Compatibility Matrix for Algorithm 1

Pure readers are allowed to proceed down the tree, coupling with read-locks to the appropriate node (or leaf, in the case of B^+ -trees). Figure 5.10 shows exactly how the protocol works. The procedure `get_child(node, key)` gets the next appropriate child of node `node`, along the path from the root to the appropriate leaf where `key` is expected.

Updater processes proceed down the tree in a lock-step manner, exclusively locking the nodes. After locking each node (except the root), the updater process checks to see if the node is safe or not. If the node is safe, then it releases the exclusive lock on the parent of the node and proceeds one step down the tree. If the node is unsafe, it splits the node, modifies the parent node, and then releases the lock on the parent. Figure 5.11 illustrates

```

procedure Search (key)
begin
  r_lock(dummy);
  r_lock(root);
  r_unlock(dummy);
  A := root;
  repeat
    if (A is a LEAF) then
      begin
        if (key is in A) then
          begin
            r_unlock(A);
            return FOUND;
          end;
        else
          begin
            r_unlock(A);
            return NOTFOUND;
          end;
        end;
        C := get_child(A, key);
        r_lock(C);
        r_unlock(A);
        A := C;
      forever;
    end; {Search}

```

Figure 5.10: Search procedure for Algorithm One

the insertion procedure and Figure 5.12 illustrates the deletion procedure in details.

Procedure `distribute(node, sibling, middle_key)` distributes the current contents of node `node` between `node` and `sibling`, passing the middle key value out. The

```

procedure Insert (key)
begin
  e_lock(dummy);
  A := root;
  e_lock(A);
  if (A is full) then
    begin
      new(new_root);
      new(sibling);
      distribute(root,new_root,middle_key);
      new_root→ptr[0]:= root;
      new_root→ptr[1]:= sibling;
      new_root→key[0]:= middle_key;
      root := new_root;
    end
  end
  e_unlock(dummy);
  repeat
    if (A is a LEAF) then
      begin
        ptr := pointer to the data record for 'key';
        insert_into_leaf(A,key,ptr);
        e_unlock(A);
        return;
      end;
    C := get_child(A,key);
    e_lock(C);
    if (C is not full) then
      begin
        e_unlock(A);
        A := C;
      end
    else
      begin
        new(C');
        distribute(C,C',middle_key);
        insert_into_nonleaf(A,middle_key,C');
        e_unlock(A);
        A := C;
      end;
    forever;
  end; {Insert}

```

Figure 5.11: Insertion procedure for Algorithm One

`middle_key` is used as the key for pointer `sibling` in the next level up. Procedure `x_unlock(lock)`, unlocks the lock `lock` of type `x` from a node. Procedures `insert_into_leaf(parent, key, ptr)` and `insert_into_nonleaf(parent, key, ptr)` insert a `key, ptr` pair in the appropriate position in `parent` when the parent is a leaf or an internal node respectively.

The procedures `merge` and `redistribute` do exactly what their names signify. `merge(node1, node2, parent)` merges the contents of nodes `node1` and `node2` keeping `node1` as the final node and makes appropriate changes in the parent — `parent`. The routine takes care of the situation when `parent` is the root by manipulating the dummy node. Calling `redistribute(father, node1, node2)` performs local rotations by redistributing the keys. Procedure `get_sibling(node, father)` gets the nearest sibling with which redistribution or merging is possible. Procedure `delete_from_leaf(leaf, key)` deletes the key `key` from leaf `leaf`, taking care of the boundary condition when `leaf` is the root.

In our algorithm we have assumed the presence of a dummy node *dummy* such that for all x in the tree $dummy \sqsubseteq x$. To see why we need the dummy node, consider x as the current root with $2d$ keys and two insertion processes I_1 and I_2 . Also, for simplicity let x be a leaf node. Let, I_1 appear before I_2 to x . By the locking scheme which we presented, I_1 gets the e-lock while I_2 waits on an e-lock for x . When I_1 , after shifting keys and creating a new root, releases the lock on x , I_2 gets the lock on x and may be misdirected if exhaustive information about the locks and the nodes which they correspond to are not kept. Moreover, keeping such forward links (locks with nodes) and backward links (nodes with locks) may be expensive and is not done in most implementations. The presence of *dummy* can come to the rescue in the case of such a boundary condition (root is split). Locking the *dummy* node ensures that the split of the root is done in a critical region and therefore the

```

procedure Delete (key)
begin
  e_lock(dummy);
  e_lock(root);
  if (root contains one or fewer keys and root is not LEAF) then
    begin
      child1 := get_child(root,key);
      child2 := get_child(child1,key);
      e_lock(child1);
      e_lock(child2);
      if (merging is possible) then
        begin
          merge(child1,child2,root);
          root := child1;
          current := root;
          e_unlock(child2);
        end
      else
        begin
          redistribute(child1,child2,root);
          if child1 = get_child(root,key) then
            begin
              e_unlock(child2)
              current := child1;
            end
          else
            begin
              e_unlock(child1);
              current := child2;
            end
          e_unlock(root);
        end;
      end
    end
  e_unlock(dummy);
  while (current is not a LEAF) do
    begin
      child := get_child(current,key);
      e_lock(child);
      if (child contains less than or equal to  $d$  keys) then
        begin
          sibling := get_sibling(child,key);
          e_lock(sibling);
          if (merging is possible) then
            begin
              merge(child,sibling,current);
              e_unlock(sibling);
              e_unlock(current);
              current := child;
            end
          else
            begin
              redistribute(child,sibling,current);
              child1 := get_child(current,key);
              if child1 = child then
                begin
                  e_unlock(sibling);
                  current := child;
                end
              end
            end
          end
        end
      end
    end
  end

```

```

                else
                    begin
                        e_unlock(child);
                        current := sibling;
                    end
                e_unlock(current);
            end;
        else
            begin
                e_unlock(current);
                current := child;
            end;
        end;
    delete_from_leaf(current,key);
end; {Delete}

```

Figure 5.12: Deletion procedure for Algorithm One

split does not cause anomalies. The case of merging the root or redistribution of keys in the level just below the root is similarly done in a critical region guarded by *dummy*.

Next, we prove a few properties of the algorithm.

THEOREM 5.1: *Updater processes do not overtake each other.*

Proof: Since exclusive locks are incompatible and all processes access the root before accessing any other node, therefore $\forall x, y$ if $x \sqsubseteq y$, and if a process P access x before Q , then P also accesses y before Q .

□

THEOREM 5.2: *A reader process may overtake another reader process.*

Proof: Follows from the fact that read-locks are compatible.

□

THEOREM 5.3: *A reader process cannot overtake another updator process or vice versa.*

Proof: Follows from the fact that read-locks and exclusive locks are incompatible.

□

THEOREM 5.4: *Algorithm 1 is deadlock free.*

Proof: To prove that the algorithm is deadlock free it is sufficient to prove separately that there cannot be deadlocks due to interactions between updator processes and reader processes (see [Bil87]).

Case 1: Updator vs. Updator. At any one time an insertion process may hold at most two locks — the lock on the parent of an unsafe node and the lock on the unsafe node itself. Let U_1 and U_2 be two update processes. Let x and y be two successive nodes in the path common to both the processes such that $x \sqsubseteq y$. Also, let U_1 access x before U_2 . Then, by theorem 5.1 U_1 also accesses y before U_2 .

Since, by algorithm 1, U_1 unlocks x only after it locks (and modifies if necessary) y , there can be no dependency of U_1 on U_2 in the waits-for graph. Hence, the processes cannot be deadlocked. In fact, the theorem follows directly from the fact that insertion processes sharing a common path are strongly ordered.

Case 2: Updator vs. Reader. Since an updator process cannot overtake another reader process or vice versa, they are strongly ordered. Hence, there cannot be deadlocks.

Case 3: Reader vs. Reader. Since read-locks are compatible, a set of reader processes cannot be deadlocked among themselves.

□

THEOREM 5.5: *Processes executing algorithm one are correct.*

Proof: When there is no split or merge of the root, the theorem follows from the fact that there cannot be deadlocks and that there exists a strong ordering on updator processes and between updator and reader processes. When the root is split, the insertion process which splits the root holds an exclusive lock on the dummy node and the previous root. Consequently, no other process can exist at the previous root and therefore there is no possibility of a process getting misdirected at the previous root. When the root is merged, the deletion process which performs the merge, holds an exclusive lock on the dummy node and the previous root. Therefore no process can exist at the previous root and hence none can get misdirected.

□

5.3.2.2. Algorithm Two

In algorithm two, only one type of lock is used. The lock is irreflexive and hence exclusive. The B-tree data structure is replaced by the B_{link} -tree data structure [LeY81]. Other than the root and the leftmost node at every level, every other node of the tree can be reached by two paths — one from the parent of the node and the other from the node at the immediate left.

Reader processes travel down the tree without locking any node. Instead of moving to a child always, the processes may move to the right sibling at a particular level, if necessary, to take care of concurrent updates.

The cases of insertion and deletions, however, have to be dealt with separately. Insertion processes travel down the tree like normal reader processes. Moreover, it keeps track of

the node in the previous level along the path it traverses. We call this node in the previous level the *pseudo-father*. Whenever the insertion process finds a node x that is full (and a potential victim of future splits), it creates a new node x_n and copies half the contents of x (both keys and pointers) into x_n . After doing this, the insertion process locks the pseudo-father of x . Then it checks whether the pseudo-father of x is the actual father of x . In case it is, let p be this actual father. Otherwise it finds in the level above x , the appropriate father p of x by moving right from the pseudo-father. Next, it locks exclusively both x and its father p and appropriately includes x_n in p . The process continues until x is a leaf, after which it inserts the key into the leaf. Note that, in this case we have reduced the time which both p and x are locked to the execution of a few machine instructions. The locking time includes the changing of the *link pointer* of x and shifting of the keys and pointers in p . Using Lamport's technique of writing and reading keys in opposite directions, we avoid reader processes getting misdirected at p . Exclusive locking prevents concurrent insertion processes from interfering, i.e., trying to write in the same node.

Deletion processes also proceed top-down from the root to the appropriate leaf. The child is locked before the parent when merging is to be done, as it was done in the case of insertion. Also, writing and reading are done from opposite directions, to avoid reader processes from getting misdirected. A flaw with this protocol, as with [LeY81], is that merging or redistribution cannot be done with a left sibling if the left sibling is locked by a different process at the time of merging. Since deletions are usually rare and mergings during deletions are even rarer in a large tree, we believe that avoiding a few merges will not create appreciable decline in the average storage utilization. If such decline in the average storage utilization is unacceptable, Sagiv's restructuring process can be used to merge the left-out nodes. Moreover, since some merging is already done by the deletion process, the restructuring process must be activated less frequently. Since the algorithm closely follows

that of algorithm one we avoid including the code.

Theorems 5.6 through 5.9 prove certain properties of the algorithm, its freedom from deadlock and correctness.

THEOREM 5.6: *In Algorithm two, an updatator process holds at most three locks at one time.*

Proof: In algorithm two, a node x is locked if and only if it is unsafe. When x is locked all its ancestors are safe and by the algorithm, unlocked. According to the algorithm, the only situation when three nodes need to be locked is under the following circumstances: when the updatator process needs to (merge) x , it finds that the old father of the split (merged) node is no longer the correct place to perform the insertion (deletion) of the pointer to the newly created node. In that case, the updatator process scans the level above the split (merged) node to find the correct insertion (deletion) position for the pointer. Moreover, three nodes are locked only for the duration of one operation — get the new father from the disk.

□

THEOREM 5.7: *Algorithm two is deadlock-free.*

Proof: Define a partial ordering (\sqsubseteq_ℓ) of nodes in B_{link} -trees in the following way. Let $x \sqsubseteq_\ell y$ if the link pointer of x points to y or the link pointer of x points to some node z such that $z \sqsubseteq_\ell y$. Moreover the partial ordering \sqsubseteq_ℓ includes the partial ordering \sqsubseteq . Since reader processes do not lock nodes, therefore they cannot be involved in deadlocks. Updatator processes lock nodes strictly following the ordering \sqsubseteq_ℓ . Since they use incompatible locks, therefore they cannot be deadlocked.

□

THEOREM 5.8: *Algorithm two produces correct executions.*

Proof: Update anomalies between updatator and reader processes are avoided by copying keys and searching for keys in opposite direction. The proof is exactly the same as in [KwW82]. Since updatator processes exclusively lock nodes from other updatator processes, the updatator processes cannot interfere to destroy correctness.

□

THEOREM 5.9: *Algorithm two does not ensure log-sequence serializability.*

Proof: Assume that a reader process R_1 accesses the nodes n_1, n_2, n_3, n_4 while searching from the root to a leaf, and that these nodes are common to a concurrent insertion process I_1 . Also let $n_1 \sqsubseteq_\ell n_2 \sqsubseteq_\ell n_3 \sqsubseteq_\ell n_4$. Then it is possible that I_1 accesses and changes n_1 and n_2 before R_1 but the other nodes after R_1 .

□

5.3.3. Operation Specific Locking Protocols

In the protocols used previously, the locks that the processes use do not bear the semantics of the processes (e.g., delete, insert, search) that use those locks. That is, given an arbitrary set of nodes and a set of lock-types (e.g., read, write, etc.) on the nodes of that set held by an arbitrary set of process, there may not be a way to categorize what operation is performed by which process. One protocol has been proposed in literature that locks a node with a lock-type that depends upon the nature of process that is locking the node. For example, insertion processes hold *insert* and *exclusive* locks, deletion processes hold *delete* and *exclusive* locks and reader processes hold *read* locks only. Given a set of nodes n_1, n_2, \dots, n_k , and a set of lock-types *insert*, *exclusive*, ..., *read* on these nodes currently held by a set of processes P, Q , it can be inferred that one of the two processes is an insertion process while the other is a read process. In this sub-section we discuss such a locking scheme and propose a new locking scheme analogous to it.

5.3.3.1. Biliri's Algorithm

Biliris was the first to propose a locking scheme that includes the semantics of the process on the locks it acquires [Bil87]. He used a B_{link} -tree data structure with additional pointers pointing to the left sibling of a node. A *low-value* per node was kept, along with the high-value as in the definition of B_{link} -tree in Chapter 3. In his algorithm, he uses 4 types of locks namely, *r*-locks (read), *i*-locks (insertion), *d*-locks (deletion) and *e*-locks (exclusive). Instead of using binary locks, he used counting locks. That is, with certain types of locks, a count is maintained. For example, the *i*-lock of a node may have a count of 5 to signify that 5 processes have currently *i*-locked the node. Such locks are easily implemented in systems where counting semaphores are provided, or with the help of binary semaphores. The compatibility and convertibility matrices of the different types of locks

are given in Figure 5.13 and Figure 5.14 respectively. In the figures, $\#x$ stands for the count of lock-type x for the node, where x is either of r , i , d or e and s stands for the number of keys currently in the node.

Reader processes walk down the tree locking nodes by coupling read-locks in a lock-step manner. Insertion processes i-lock their scopes on their way down as in the case of other type one algorithms. On reaching the appropriate leaf, the process exclusively locks the leaf and performs the insertion. In the case of an overflow, a restructuring of the tree is propagated upwards. The restructuring process uses a side-branching technique essentially the same as the one introduced in [KwW82]. Deletion processes walk down the tree per-

	r	i	d	e
r	n	n	n	n
i	y	$(\#i < 2d - s) \text{ or } (\#i = 0)$	n	n
d	y	n	$(\#d < d - s) \text{ or } (\#d = 0)$	n
e	n	y	y	n

Figure 5.13: Compatibility Matrix for Biliris' Algorithm

	r	i	d	e
r	y	y	y	n
i	n	n	n	y
d	n	n	n	y
e	n	n	n	n

Figure 5.14: Convertibility Matrix for Biliris' Algorithm

forming aggressive mergings. On each level, the deletion process d-locks the node x and examines whether x needs to be merged or not. If it needs to be merged, then it immediately d-unlocks the node and e-locks the parent, passing control to a restructure routine. The restructuring routine then performs the merging or the redistribution of keys between the appropriate children of the e-locked parent node. If the original node does not need to be merged, then it d-unlocks n and proceeds another level down the tree. Further details of the exact protocol can be found in [Bil87].

5.3.3.2. Algorithm Three

We present another protocol for concurrency control based on operation-specific locking. In the algorithm proposed by Biliris, merging of nodes is done aggressively while splitting of nodes is deferred. In practical database applications, the number of insertions far exceeds the number of deletions. Also given a bounded amount of main memory, the cost of insertion by aggressively splitting the nodes is less than the cost for deferred splitting of nodes [Gra89]. Moreover, performing deferred operations (i.e., splitting), require the locking of larger scopes as in type one algorithms or tricky manipulations of the insertion algorithm steps (type 2 algorithms). As an engineering tradeoff, we were interested in the performance of algorithms where insertion is done aggressively while deletion is deferred. For trees with smaller order, our algorithm is definitely a better choice over the algorithm by Biliris as the cost difference between aggressive and deferred insertions increases with increase in the height of the tree.

In our protocol, four types of locks are used, namely, *r-locks*, *u-locks*, *p-locks* and *a-locks*[†]. Reader processes use r-locks only, insertion processes use u-locks and a-locks and

[†]As the locks bear the semantics of the processes that use them, the locks taken together bear the semantics of the author's life.

deletion processes use p-locks and a-locks. The compatibility matrix and the convertibility matrix of the four types of locks are given in Figures 5.15 and 5.16 respectively. The locking mechanism in this algorithm is different from that of algorithm one (where more than one type of lock is used). From the compatibility matrix, note that, if a process requests a r-lock on a node that is already r-locked, the request will not be granted. In our algorithm, u-locking or p-locking a node also implicitly r-locks the node. Consequently, when locks are released, care has to be taken to handle the implied locks also. Reader processes, on their way down to the appropriate leaf, lock nodes in a lock-step manner with r-locks. The reader processes follow the algorithm given in Figure 5.17. The procedure is essentially

	<i>r</i>	<i>u</i>	<i>p</i>	<i>a</i>
<i>r</i>	<i>n</i>	<i>n</i>	<i>n</i>	<i>n</i>
<i>u</i>	<i>y</i>	$(\#u < 2d - s) \text{ or } (\#u = 0)$	<i>n</i>	<i>n</i>
<i>p</i>	<i>y</i>	<i>n</i>	$(\#p < d - s) \text{ or } (\#p = 0)$	<i>n</i>
<i>a</i>	<i>n</i>	<i>y</i>	<i>y</i>	<i>n</i>

Figure 5.15: Compatibility Matrix for Algorithm Three

	<i>r</i>	<i>u</i>	<i>p</i>	<i>a</i>
<i>r</i>	<i>y</i>	<i>y</i>	<i>y</i>	<i>n</i>
<i>u</i>	<i>n</i>	<i>n</i>	<i>n</i>	<i>y</i>
<i>p</i>	<i>n</i>	<i>n</i>	<i>n</i>	<i>y</i>
<i>a</i>	<i>n</i>	<i>n</i>	<i>n</i>	<i>n</i>

Figure 5.16: Convertibility Matrix for Algorithm Three

identical to that in algorithm one. The procedure `get_child(node, key)` gets the next appropriate child of node `node`, along the path from the root to the appropriate leaf where `key` is expected.

Insertion processes go down the tree to the appropriate leaf, u-locking the nodes one at a time. After u-locking the node, it checks if the node already contains $2d$ keys. If it does, then the process immediately releases the u-lock on the current node and a-locks the parent. A new node is thereafter created and the keys and pointers are shifted from the full node to the newly formed node appropriately. The parent node's contents are then appropriately changed and the a-lock on the parent is released. The insertion process then proceeds one

```

procedure Search (key)
begin
  r_lock(dummy);
  r_lock(root);
  r_unlock(dummy);
  A := root;
  repeat
    if (A is a LEAF) then
      begin
        if (key is in A) then
          begin
            r_unlock(A);
            return FOUND
          end
        else
          begin
            r_unlock(A);
            return NOTFOUND
          end;
        end;
        C := get_child(A, key);
        r_lock(C);
        r_unlock(A);
        A := C;
      forever;
    end; {Search}

```

Figure 5.17: Search procedure for Algorithm Three

level down. If the original node was less than full then the insertion process would simply go down another level. Figure 5.18 presents the insertion algorithm in detail.

The procedure `convert(node, type1, type2)` converts the lock type of `type1` on node `node` to `type2`. Procedure `distribute(node, sibling, middle_key)` distributes the current contents of node `node` between `node` and `sibling`, passing the middle key value out. The `middle_key` is used as the key for pointer `sibling` in the next level up. Procedure `unlock(node)` unlocks all current locks on the node `node`. Procedures `insert_into_leaf(parent, key, ptr)` and `insert_into_nonleaf(parent, key, ptr)` insert a `key, ptr` pair in the appropriate position in `parent` when the parent is a leaf or an internal node respectively. A deletion process walks down the tree locking the nodes with p-locks and stacking them. Whenever the deletion process finds a node that is deletion safe, it unlocks all nodes that have been stacked till that point and empties the stack. Thus by the time the deletion process reaches the appropriate leaf where the deletion is done physically, it has the local scope p-locked. If the deletion induces a bottom-up propagation, then a side-branching technique (as introduced in [KwW82]) is used to propagate the change upwards. The major operation of the delete process thus, consists of the following:

- 1) p-lock the scope.
- 2) restructure by node merging or key redistribution using a side branch.
- 3) discard the redundant branch.

Figure 5.19 shows the deletion algorithm in greater detail.

The procedures `merge` and `redistribute` do exactly what their names signify. `merge(node1, node2, parent)` merges the contents of nodes `node1` and `node2`, making appropriate changes in their parent — `father`. Calling `redistribute(father, node1, node2)` performs local rotations by redistributing the keys. Procedure `remove_branch(node)` removes the side-branch from node `current`. Procedure

```

procedure Insert (key)
begin
  a_lock(dummy);
  A := root;
  u_lock(A);
  if (A is full) then
    begin
      r_unlock(A);
      convert(A,u,a);
      new(new_root);
      new(sibling);
      distribute_keys(root,new_root,middle_key);
      new_root→ptr[0]:= root;
      new_root→ptr[1]:= sibling;
      new_root→key[0]:= middle_key;
      root := new_root;
      excl := true;
    end;
  a_unlock(dummy);
  r_unlock(dummy);
  if (A is a LEAF) then
    if (not excl) then
      begin
        r_unlock(A);
        convert(A,u,a);
      end;
  repeat
    if (A is a LEAF) then
      begin
        ptr := pointer to the data record for 'key';
        insert_into_leaf(A,key,ptr);
        a_unlock(A);
        return;
      end;
    C := get_child(A,key);
    u_lock(C);
    if (child is not full) then
      begin
        if (C is a LEAF) then
          begin
            unlock(A);
            convert(C,u,a);
            A := C;
          end
        end
      end
    else
      begin
        r_unlock(C);
        convert(C,u,a);
        if (not excl) then
          begin
            r_unlock(A);
            convert(A,u,a);
          end
        end
      end;
    new(C');
    distribute(C,C',middle_key);

```

```
        insert_into_nonleaf(A,middle_key,C');
        e_unlock(A);
        excl := true;
    end;
    forever;
end; {Insert}
```

Figure 5.18: Insertion procedure for Algorithm Three

```

procedure Delete (key)
begin
  p_lock(dummy);
  p_lock(root);
  current := root;
  if (current is deletion safe) then
    begin
      p_unlock(dummy);
      r_unlock(dummy);
      deepest_safe := root;
    end
  else
    deepest_safe := dummy;
    son := get_child(current,key);
    while (son is not a LEAF) do
      begin
        p_lock(son);
        current := son;
        if (current contains more than  $d$  keys) then
          begin
            deepest_safe := current;
            p_unlock all ancestors of current;
          end;
          son := get_child(current,key);
        end;
      end;
    if (key is in current) then
      begin
        rotate := false;
        while (current does not contain more than  $d$  keys) do
          begin
            father := father of current;
            if (father is dummy) then
              begin
                handle_root();
                goto special;
              end;
            sibling := get_sibling(current,father);
            p_lock(sibling);
            if (sibling contains less than  $d$  keys) then
              begin
                convert(sibling,p,a);
                merge(sibling,current,father);
                a_unlock(sibling);
                current := father of current;
              end;
            else
              begin
                rotate := true;
                convert(father,p,a);
                convert(sibling,p,a);
                convert(current,p,a);
                redistribute(father,current,sibling);
                a_unlock(sibling);
                if (depth(current) > depth(deepest_safe)) then
                  begin
                    a_unlock(father);
                    a_unlock all ancestors of father;
                  end;
                end;
              end;
            end;
          end;
        end;
      end;
    end;
  end;

```

```

    special:
    end;
  if (not rotate) then
    convert(current,p,a);
  remove_branch(current);
  a_unlock(current);
  while (current is not a LEAF) do
  begin
    current := get_child(current,key);
    convert(current,p,a);
    a_unlock(current);
  end;
end
else
  p_unlock current and all ancestors;
end; {Delete}

```

Figure 5.19: Deletion procedure for Algorithm Three

`get_sibling(node, father)` gets the nearest sibling with which redistribution or merging is possible. Procedure `handle_root()` handles the situation when the root is merged. Note that, in this case also we have also used the *dummy* node to take care of the boundary conditions.

A problem of this protocol, along with all other top-down algorithms in which updater processes do not use exclusive locks on their way down the tree, is that these protocols do not preserve *log-sequence serializability*. As an example, consider 4 nodes related as $r \sqsubseteq u \sqsubseteq p \sqsubseteq a$ which appear along the path from the root to leaves in that order and are common to the path of two insertion processes I_1 and I_2 . It is possible that I_1 accesses r and u before I_2 but that I_1 accesses p and a after I_2 . However the algorithms satisfy the criterion of *operation serializability*.

Next, we prove that the algorithm does not create deadlocks.

THEOREM 5.10: *Algorithm 3 is deadlock free.*

Proof: We prove the algorithm to be deadlock free by examining the synchronization achieved between the three types of processes.

Case 1: Reader vs. Reader. Reader processes do not deadlock by themselves as they proceed in one direction always.

Case 2: Reader vs. Insert. Reader processes always exclusively lock the parents before the full child is unlocked and the new sibling of the full child is added to the

parent. Thus, the reader cannot be misdirected due to an insertion. Moreover, since both reader and insertion processes lock nodes in a top-down manner and never lock bottom-up, they cannot be deadlocked by themselves.

Case 3: Insert vs. Insert. We have to consider separately the cases when the root is split and when any other node is split. Consider the case when the root needs to be split. The root must contain $2d$ keys and only one process can get a u-lock over the root. Since locking is done top down only, a strong ordering is thus imposed on all insertion processes. Hence, there cannot be a deadlock. In the case when any other node is split, the both the parent of the full node and the full node itself are a-locked. Since a-locks are incompatible with each other and other u-locks, a strong ordering is imposed on the insertion processes at this node. Hence there cannot be a deadlock.

Case 4: Insert vs. Delete: Operations performed by insertion and deletion processes which modify the tree are done with their local scope locked by u-locks and p-locks. For insertion processes, the local scope refers to the two full nodes being split and their parent, and for a deletion process it refers to the subtree rooted at the deepest safe node. Since u-locks and p-locks are incompatible, therefore there can be no interference among the processes.

Case 5: Delete vs. Reader: follows from the proof of deadlock-freedom in [KwW82]. Deletion processes do not perform any modifications during the searching phase or when the restructuring is done from the appropriate leaf to the deepest safe node. Recall that during that time the side-branches are not yet attached to the tree and are invisible to the reader processes. Therefore the compatibility of p-locks and r-locks create no problem. When the side-branches are attached to the tree in a top-down manner, the p-locked are converted to a-locks which are incompatible with r-locks and hence the processes do not conflict. These e-locks provide a total ordering and hence reader and delete processes cannot be interleaved once the deepest safe node is a-locked. Since the reader processes and the delete processes place the locks in the nodes visible to *both* in a top-down manner, there cannot be dead-locks.

Case 6: Delete vs. Delete: See [KwW82].

□

5.3.4. Optimistic Concurrency Control

Optimistic concurrency control protocols have been used to maintain concurrent access to B-trees in commercial database systems [KuR81]. Processes are allowed to proceed freely and make changes in the tree irrespective of the nature of the process (updater or reader). Such changes are made in a local buffer specific to the particular process and the changes propagated atomically to the common shared tree structure if the transactions satisfy certain conditions (called validation criteria). These conditions were discussed in Chapter 2.

Processes make separate copies of the nodes along the path of the tree, which they modify. Moreover, the processes propagate the modifications atomically (or in a critical region) upon validation. Each process is assigned an unique *process number* upon origination and the process keeps track of the set of nodes that it reads and modifies. Upon termination of the process, the global counter that is used to assign process numbers is examined. The read sets of all processes that have their process numbers lying between the current content of the global counter and the process number of the process being validated are compared with the write set of the process being validated. If the intersection set is non-null, the validating process is aborted (or restarted) without propagating the changes made by it to the common shared tree structure.

An advantage of this scheme is that it can be made to support log-sequence serializability. However, for trees of low-order, where restructuring is frequent, optimistic concurrency control is not a good scheme due to large number of conflicts resulting in transactions getting aborted.

5.4. Comparison of Algorithms

In this sub-section we present an empirical comparison of the algorithms discussed in this chapter in a tabular format similar to that in [KwW82]. Three separate tables summarize the features of the algorithms discussed in the chapter.

<i>Property</i> ↓	<i>Protocol</i> →	<i>Samadi</i> - <i>Parr</i>	<i>Bayer</i> and <i>Schkolnick</i> 1,2,3	<i>Ellis</i>	<i>Kwong</i> and <i>Wood</i>
<i>Types of locks used</i>		<i>1</i>	<i>2,2,3</i>	<i>3</i>	<i>3</i>
<i>Updaters are strongly ordered</i>		<i>yes</i>	<i>yes, yes, yes</i>	<i>yes</i>	<i>yes</i>
<i>Overtaking among readers</i>		<i>no</i>	<i>yes, yes, yes</i>	<i>yes</i>	<i>yes</i>
<i>Readers in the scope of updaters during searching phase</i>		<i>no</i>	<i>no, no, yes</i>	<i>yes</i>	<i>yes</i>
<i>Readers in the scope of updaters during restructuring</i>		<i>no</i>	<i>no, no, no</i>	<i>yes</i>	<i>yes</i>

Figure 5.20: Comparison table for bottom-up type 1 algorithms

<i>Property</i> ↓	<i>Author</i> → <i>Ellis</i>	<i>Miller and Snyder</i>	<i>Lehman and Yao</i>	<i>Kwong and Wood</i>	<i>Sagiv</i>
<i>Types of locks used</i>	<i>3</i>	<i>3</i>	<i>1</i>	<i>3</i>	<i>3</i>
<i>Updaters are strongly ordered</i>	<i>no</i>	<i>no</i>	<i>no</i>	<i>no</i>	<i>no</i>
<i>Overtaking among pure readers</i>	<i>yes</i>	<i>yes</i>	<i>yes</i>	<i>yes</i>	<i>yes</i>
<i>Data-structure used</i>	<i>B-trees with leaves as chains</i>	<i>B-trees with queues per node</i>	<i>B_{link}-trees</i>	<i>B-trees with queues per node</i>	<i>B_{link}-trees</i>
<i>Concurrent deletions allowed</i>	<i>only one</i>	<i>many</i>	<i>many</i>	<i>many</i>	<i>many</i>
<i>Maximum number of locks held at any time</i>	<i>depends on tree size</i>	<i>varies with order of tree</i>	<i>3</i>	<i>depends on tree size</i>	<i>3</i>

Figure 5.21: Comparison table for bottom-up type 2 algorithms

<i>Property</i> ↓	<i>Protocol</i> →	<i>Algorithm 1</i>	<i>Algorithm 2</i>	<i>Biliris</i>	<i>Algorithm 3</i>
<i>Nature of Protocol</i>		<i>top-down</i>	<i>top-down</i>	<i>operation specific</i>	<i>operation specific</i>
<i>Types of locks used</i>		<i>2</i>	<i>1</i>	<i>4</i>	<i>4</i>
<i>Locks are counting</i>		<i>no</i>	<i>no</i>	<i>yes</i>	<i>yes</i>
<i>Updaters are strongly ordered</i>		<i>yes</i>	<i>yes</i>	<i>no</i>	<i>no</i>
<i>Overtaking among pure readers</i>		<i>yes</i>	<i>yes</i>	<i>yes</i>	<i>yes</i>
<i>Data-structure used</i>		<i>top-down B⁺-trees</i>	<i>top-down B⁺-trees</i>	<i>B_{link}-trees with aggressive deletions and deferred insertions</i>	<i>B_{link}-trees aggressive insertions and deferred deletions</i>
<i>Concurrent deletions allowed</i>		<i>many</i>	<i>many</i>	<i>many</i>	<i>many</i>
<i>Maximum number of locks held at any time</i>		<i>depends on tree size</i>	<i>3</i>	<i>depends on tree size</i>	<i>depends on tree size</i>

Figure 5.22: Comparison table for top-down and operation specific algorithms

CHAPTER 6

Concurrency Control Performance

6.1. Introduction

In this chapter we compare the performance of nine concurrency control algorithms for B-trees. As discussed in the previous chapter, many algorithms utilize some modifications to the basic B-tree data-structure to achieve extra concurrency. In order to avoid comparisons between apples and oranges, only those algorithms that employ only minor changes to the basic data structure are included in our study.

In addition to the changes in the data-structure, the disparity in the approaches of the algorithms is also a problem. The simulation parameters have to be appropriately chosen to care of this disparity. As an example, consider what should be an appropriate simulation parameter to compare locking based protocols with optimistic protocols. Protocols based on locking have the inherent cost of maintaining locks, which is not incurred by optimistic concurrency control protocols. Thus we have to select a simulation parameter that overlooks the locking overhead. However, when we compare locking-based protocols, the overhead due to locking must be compared, too. Thus, it is difficult or impossible to compare the protocols with respect to one particular metric.

In this chapter, we describe the performance metric, the factors that led to our choice, our workload model and a comparative study of the protocols with respect to this metric.

6.2. Workload Model

Our simulations manipulated a B-tree stored entirely in main memory. We assume a fully shared memory multiprocessing environment. Each process consists of several operations on B-trees and is analogous to a transaction of general database systems. The operations are analogous to sub-transactions in a nested transaction environment. Transactions are categorized into three classes, namely, *short*, *medium* and *large* [CaM86]. *Short* transactions contain 500 operations, *medium* transactions contain 1,000 operations and the *large* transactions contain 2,000 operations. The simulations were performed on a Sequent Symmetry machine with eight processors, supporting a shared-memory architecture. Of the operations in each process, the ratio of update to read operations was varied.

The Unix system call `fork()` was used to create the children of a master process. Each child of this master process is a transaction. We observed that since the `fork` system call takes a while to execute, when there are many short transactions, some would finish before the others would get started. To avoid this, we forced all transactions to start at the same time. We made all processes start at the same time with the use of the `s_wait_barrier()` and the `s_init_barrier()` library routines provided with the operating system. The `s_init_barrier()` system call initializes an internal data structure, which contains information about how many processes are to synchronize at the barrier point. The master process calls this routine before forking the children. The `s_wait_barrier()` routine is called by each of the child transactions. The execution of this routine, in effect, makes a process wait until the previously specified number of processes reach the barrier point.

Each of the transactions is provided with a *process-id* and two random number generators unique to the process. The first random number is used to determine the nature of

the operation (i.e., insert, delete or search) and the second random number is used to generate the key. The keys that are inserted are stored in a file and in case the operation is a delete, another random number (common to all the processes) is generated. If this random number is even, a previously-inserted key is deleted, otherwise a random key is selected and a deletion of that key was attempted. To avoid the random deletion from being mostly unsuccessful, the range of the random numbers was reduced. Moreover, the time to generate random numbers is also included in our simulations. We found the above assumption to be acceptable because the time to generate random numbers is small compared to the total time and same for all algorithms. Moreover, the time for other alternatives is of the same order.

6.3. Implementation of Locks

Locks are implemented with the help of *spin-locks*, which are provided by the DYNIX operating system [87]. In the Sequent Symmetry, locks are shared bytes of memory of type **unsigned char**. We used spin locks because the only other available choice is semaphores which are extremely slow and does not guarantee freedom from livelocks, which compelled us to reject that choice. Moreover, with the current configuration of the operating system, a bounded number of semaphores (21) can be active at one time. For the simulation results to be stable, the trees that are to be manipulated have to have hundreds of nodes. Hence, we could not associate semaphores with every node of the tree. There are ways this problem can be circumvented [SiP87], but they incur unacceptable overheads, which we wanted to avoid.

However, there are problems with using the locks provided by the system. Unlike those of transaction-oriented systems (e.g., the BiiN Operating System), the locks in Dynix are not typed. That is, they cannot be categorized as read or write locks. Moreover, the

locks are binary — that is, a separate counter has to be maintained for manipulation of counting locks as necessitated in some of the protocols. The problem can be avoided by maintaining a separate lock table. Updates to this lock table have to be done in a critical-region to avoid inconsistencies. Since the operations are data-intensive, the lock table often acts as an unacceptable hot spot. These problems motivated us to implement our own locks.

In our simulations, in the most general case, a lock is an array consisting of four bytes of type `unsigned char`. Thus, the type declaration for locks is:

```
typedef rlock_t    unsigned char[4];
```

We chose that type because it is the smallest size that would meet our requirements. The first element of the array (first byte) is used to store the type of the lock (e.g., read, write, etc.). The second and the third bytes contain auxiliary information wherever necessary (for example, to store lock types in Algorithm 3). The fourth byte is used to store the count whenever necessary. The use of an extra lock to ensure atomic update of the count along with the lock is avoided by atomically transferring the long word (the entire array) to memory. Optimizations are made wherever appropriate (e.g., using only one byte in the protocol by Samadi and Parr). Using this definition of locks, the *lock* and the *unlock* routines were implemented in assembly language.

The Sequent assembly language (which closely follows that of the 80386 microprocessor) provides us with the `xchg{b|w|l}` opcode, which was extensively used in the manipulation of the locks. The opcode atomically exchanges the contents of the memory operand with a memory operand. While waiting for a lock, spinning is done in the cache to reduce bus traffic. Since our locks were quite densely distributed in memory, we also had to take care of the position of the locks in the structure of a node. We noticed that if the memory

operands are not aligned in the boundaries of memory words, the `xchg{b|w|l}` opcode takes appreciably more time to execute. Moreover, due to the cache coherence policy of the Sequent, additional overhead may be incurred. If, for example, there are three locks in one cache line, and if three processes (scheduled in three processors) simultaneously access each of them, then one of the processes gets the lock first and the other caches must be invalidated. Such frequent invalidation of caches is called the *ping-pong* effect and has been discussed in detail in [HwB81]. However, we did not employ any technique to overcome it.

6.4. The Performance Metric

The measure of performance we measure is the *total elapsed time* of a transaction. The total elapsed time of a process is defined as the total time taken by the process to execute in user mode (see [Gra89]). Since the protocols we implemented are deadlock-free and correct, all transactions complete. The time for a transaction was obtained by starting a stopwatch before the transaction started and measuring the time after the execution of the transaction. The system call `getrusage()` was used for this purpose. The time elapsed in executing in user mode was taken, since we are interested in the total elapsed time. We resorted to the real time instead of simulating the time because we are simultaneously interested in certain aspects like locking overhead which are difficult to simulate. Moreover, it is not our prime objective to get the exact time of transactions since we are interested in relative performance evaluations and not analytical simulations.

For a set of transactions, the total elapsed time of each of the member transactions was noted and the difference between the minimum and the maximum total elapsed times computed. For a given set of transactions, with the same number of operations in each transaction, the mean of the total elapsed times, we believe, best captures the overheads (e.g., locking, lock manipulation etc.) of the protocol. The computed difference between the

fastest and the slowest transactions reflects the standard deviation of the total elapsed times, which in turn signifies the *degree of concurrency* of the protocol. A good protocol should have small values for both the mean and the standard deviation.

6.5. Results

We have implemented nine protocols among those discussed in the previous chapter and compared their performance. For future reference, the protocols are indexed according to the table in Figure 6.1.

6.5.1. Total Elapsed Time

Figure 6.3 through Figure 6.5 plots the variation of the total elapsed time versus the number of processes, for the nine different protocols. Each figure (consisting of 9 graphs) corresponds to one class of transaction. In each of the graphs, the vertical lines corresponding to a particular number of processors, correspond to the computed difference between the

<i>Index</i>	<i>Author</i>
<i>Algorithm 1</i>	<i>Samadi and Parr</i>
<i>Algorithm 2</i>	<i>Bayer and Schkolnick (1)</i>
<i>Algorithm 3</i>	<i>Bayer and Schkolnick (2)</i>
<i>Algorithm 4</i>	<i>Bayer and Schkolnick (3)</i>
<i>Algorithm 5</i>	<i>Lehman and Yao</i>
<i>Algorithm 6</i>	<i>Algorithm 1</i>
<i>Algorithm 7</i>	<i>Algorithm 2</i>
<i>Algorithm 8</i>	<i>Biliris</i>
<i>Algorithm 9</i>	<i>Algorithm 3</i>

Figure 6.1: Index of the protocols compared

minimum and the maximum execution times of the constituent processes. The minimum and the maximum times are joined for convenience.

To get the data, each of the simulations was run 2 times to minimize the effect of the operating system (i.e., scheduling of the processes) and the minimum and the maximum times of the processes are noted. However, in most of the graphs there is a sudden increase in the total elapsed time when the number of constituent transactions exceeds 8. The data corresponding to more than 8 transactions is probably unreliable because the Sequent Symmetry has 8 processors. If the number of transactions (i.e., processes) is less than 9 then each of the processes is scheduled in a separate processor. However, the exact behavior could not be determined when the number of processes exceeded 8.

In each of the figures, we are particularly interested in the relative performances of Algorithms 2 and 6, Algorithms 5 and 7 and Algorithms 8 and 9, since each pair of these algorithms operate on the similar structural variant. From Figure 6.3, we note that the algorithm by Samadi-Parr (Algorithm 1) is the most time consuming compared to the others because it puts an ordering on all processes sharing a common path. The first algorithm of Bayer (Algorithm 2) is better than protocol 1 but the response time rises with the increase in the number of processes acting concurrently. The second solution of Bayer, on the other hand, performs appreciably better. At a first glance, it might appear that the third algorithm of Bayer is not especially efficient. But, considering that only 30 percent of the operations are updates, the behavior is predictable. Obviously, Algorithm 5 gives the best performance, among other bottom-up algorithms, since it allows other updaters to exist in the scope of another. Also note that the variance of the time for a given number of processes is also less, which is expected. Algorithm 6 behaves, more or less like that of Algorithm 2, but the slope of the curve is higher. This effect stems from the fact that splitting and merging is done top-down instead of bottom up and that, in the former case, the number of splits or

mergings is greater. Algorithm 7 performs like Algorithm 5, however with a larger slope due to the fact that there are more merges and splits. Moreover, since Algorithm 7 takes care of some instances of deletion, the mean total execution time is slightly more. Algorithm 8 and Algorithm 9 have behavior comparable to each other. However, the total elapsed time rises faster in Algorithm 9 because Algorithm 9 performs insertions aggressively as compared to Algorithm 8. Since the number of insertions is greater than the number of deletions, the total elapsed time is also greater.

With medium and long transactions (Figures 6.4 and 6.5) the same pattern in the variation of the total elapsed time with the number of processes is observed.

6.5.2. Effect of Read and Write Ratio

We also studied the variation of the total elapsed time with variations in the ratio of read and write operations per transaction. Figure 6.6 illustrates this variation for the different protocols. Only the medium sized transactions are considered. In each of the graphs, the mean of the total elapsed times is plotted against the number of processes. Figure 6.2 shows the percentage of the different operations corresponding to the line types of the graphs.

Line type	% insert	% delete	% search
•	7%	3%	90%
□	20%	10%	70%
△	33%	17%	50%

Figure 6.2: Percentages of different operations.

As expected, algorithm 1 is most sensitive to the read/write ratio because, with more writes, more subtrees have to be locked and, since all processes are ordered, all the processes have to wait until the subtree is released for traversal. Algorithm 6 is also more sensitive to the read/write ratio than algorithm 7, because the latter does not block read processes from entering the scope of updaters. Algorithm 9 is more sensitive to the read/write ratio than algorithm 8 because it performs aggressive insertions and as insertions outnumber deletions.

In general, we observe that our algorithms have comparable performance to the ones reported in literature. Thus, not only the cost of an operation is less in top-down trees, concurrency control of top-down trees is simpler and the overheads of concurrency control are comparable to the concurrency control overheads of bottom-up trees.

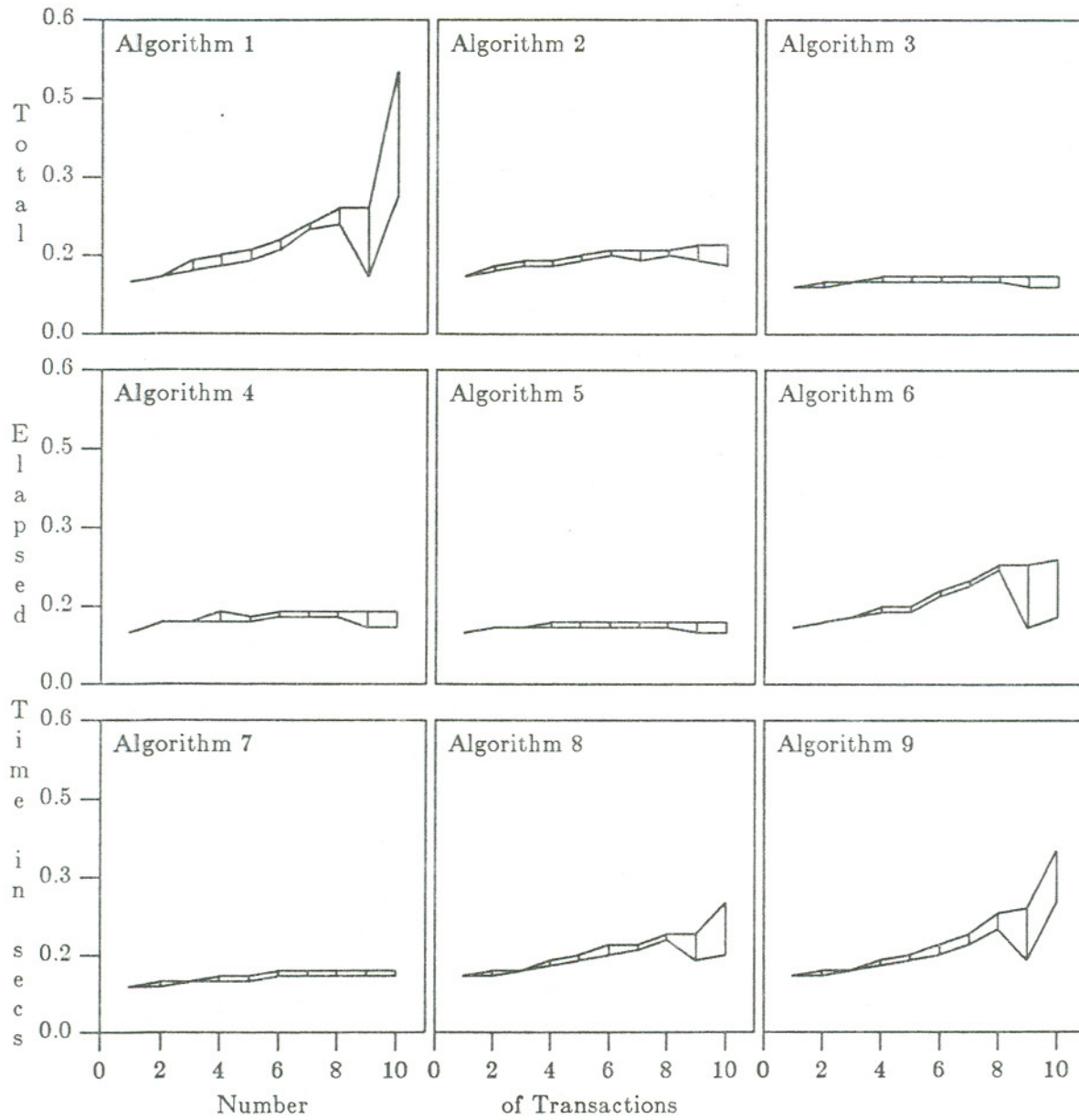


Figure 6.3: Total Elapsed Time versus number of *short* transactions

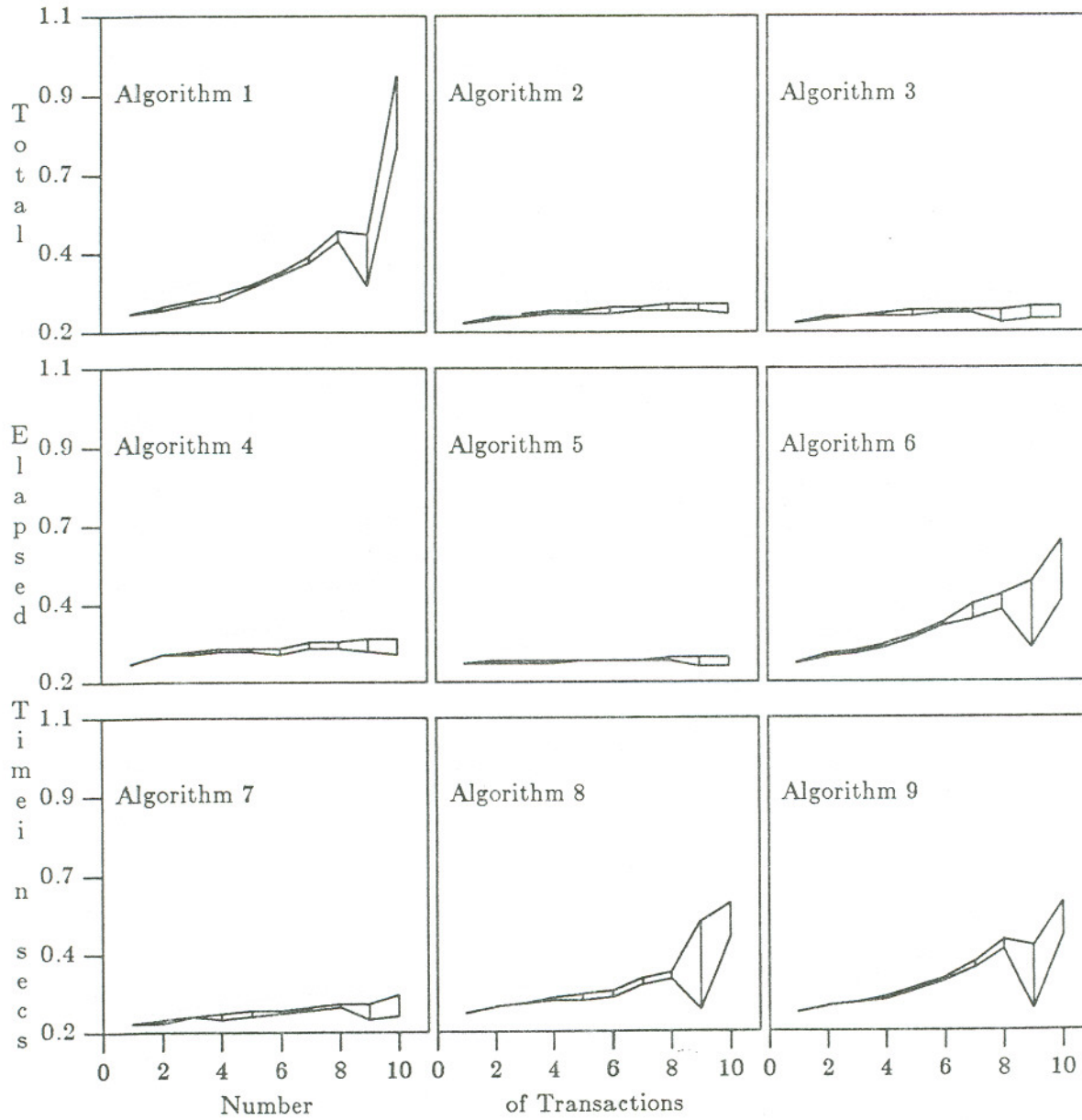


Figure 6.4: Total Elapsed Time versus number of *medium* transactions

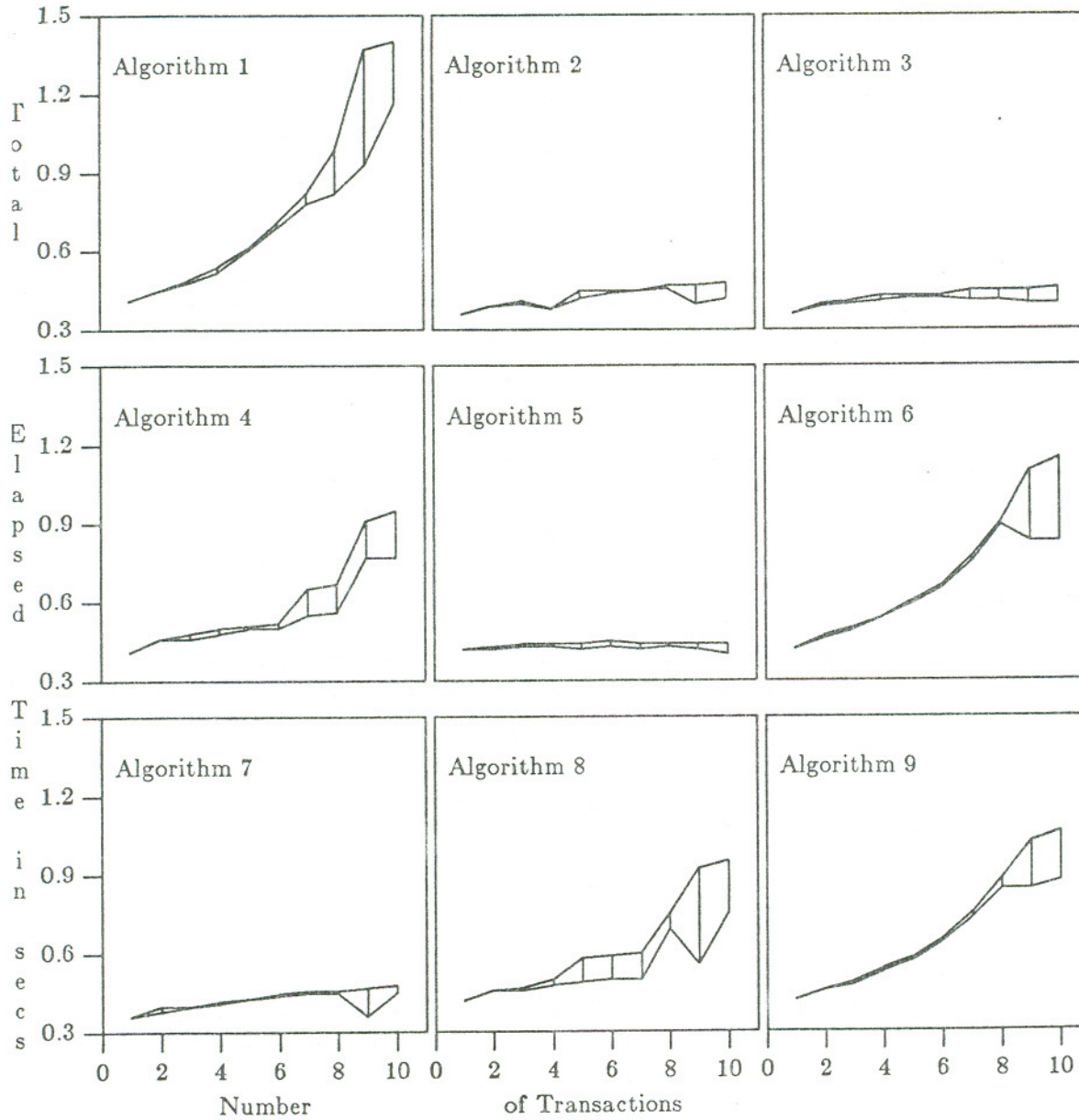


Figure 6.5: Total Elapsed Time versus number of *large* transactions

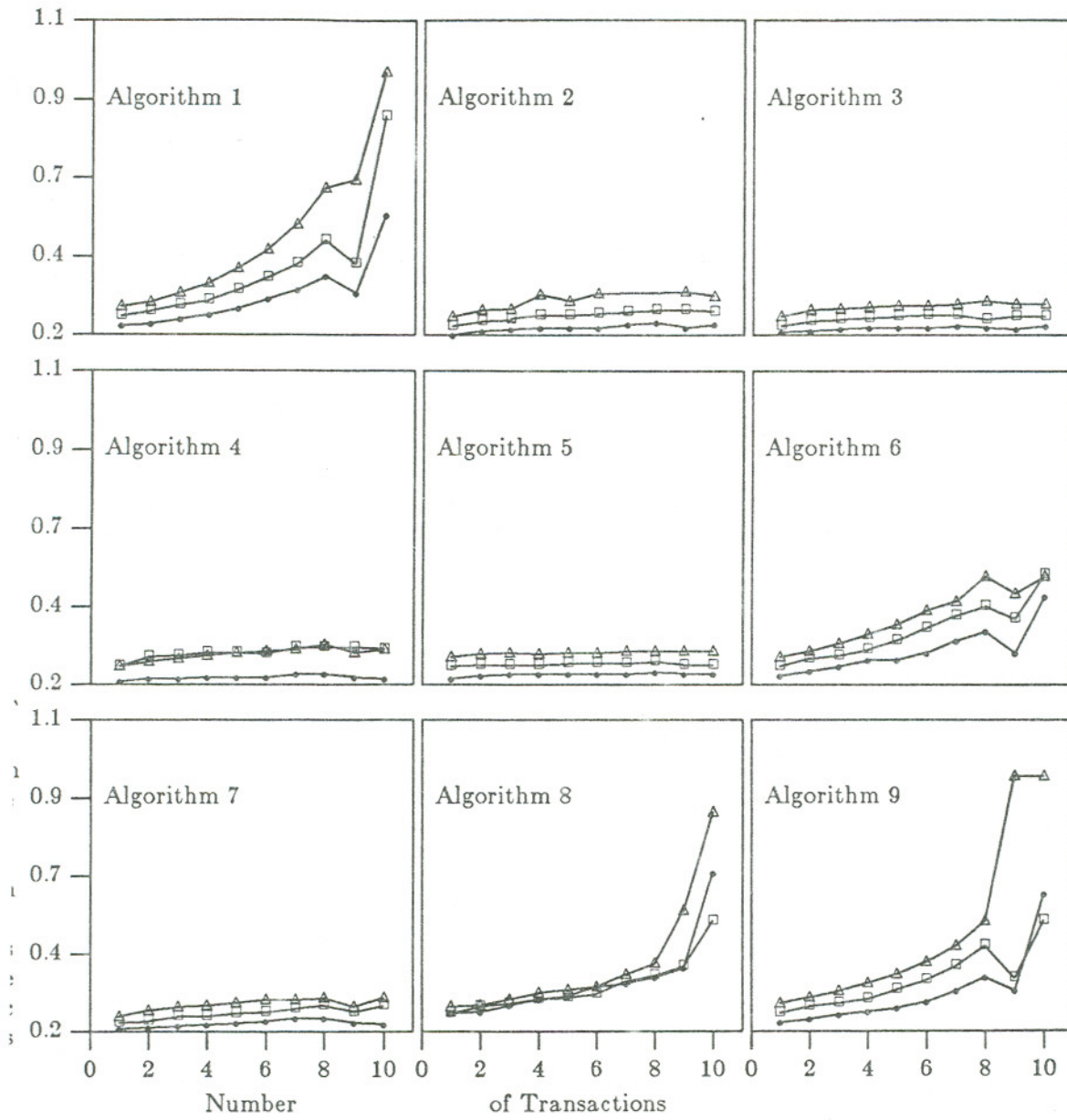


Figure 6.6: Effect of read-update ratio on Total Elapsed Time

Fuck you! you filthy troff

CHAPTER 7

Epilogue

7.1. Summary and Conclusions

The purpose of the thesis was to study and compare the performance and concurrency control protocols for top-down B-trees and bottom-up B-trees. Concurrency control of bottom-up B-trees is well studied and many efficient protocols have been provided for the same. However, concurrency control of top-down B-trees has not been studied in detail. It was conjectured in a paper ([LeY81]) that top-down B-trees are not efficient in a concurrent processing environment. In this thesis, we wished to verify the conjecture and explore the engineering trade-offs between the two data structures.

In Chapter 1 we discussed the problem of concurrency control from the perspective of general database systems. In Chapter 2 we surveyed the state of the art in concurrency control of general purpose centralized database systems. In Chapter 3 we discussed the several variants of B-trees reported in the literature and their advantages and disadvantages. We also presented a probabilistic analysis of the effect of slack factor on the average storage utilization of B-trees and compared the analyzed results with actual values obtained from simulations. Next, we discussed the different algorithms for operations on B-trees (i.e., top-down splitting and bottom-up splitting of nodes). Then we proved mathematically that the average number of node splits in the case of top-down splitting is strictly more than the number of node splits in the case of bottom-up trees. Finally, we presented some possible optimizations on the B-tree data structure to enhance some performance metrics and probabilistically analyzed one of them. In Chapter 4 we discussed the effect of different key

distributions on the performance of B-trees by probabilistic simulations of the different key distributions and supported probabilistic analysis. In Chapter 5 we discussed all protocols reported in the literature on the concurrency control of B-trees to date. Next, we presented two protocols for top-down B-trees and proved their correctness and freedom from deadlock. Then, we presented another protocol with operational semantics that employs deferred deletions and aggressive mergings. Finally in Chapter 6, we studied the performance of the various protocols as a function of the number of processes and of the read/write ratios.

We conclude from our study that top-down B-trees are a viable alternative to bottom-up B-trees when primary memory size is limited. Though extra overheads have to be taken care of in the case of top-down B-trees, the overheads are small enough to be ignored in most database applications. As in the case of bottom-up B-trees, there exist concurrency control protocols that have comparable performance with their bottom-up counterparts. However, in applications where there are more deletions, bottom-up B-trees are still more advantageous. We also verified experimentally that variations in key distributions affect top-down and bottom-up B-trees more or less similarly.

7.2. Future Study

We could not mathematically analyze the effect of different insertion and deletion probabilities on the performance of B-trees. The reason is, while taking into account the different probabilities of insertion and deletions, the solution of the transition probability matrix reduces to the solution of a tri-diagonal system of recurrence relations instead of a bi-diagonal system. It is well known that standard methods to solve tri-diagonal system of equations fail on certain values of the coefficient matrix. We would like to know what are the ranges of the insertion and the deletion probabilities for which parameters such as the average storage utilization and the probability of split stabilize in the asymptotic case.

Once that is done, it is simple to calculate the expected survival interval of a node. Although a quantitative approximation of the average survival interval of a node has been reported in the literature, the analysis is based on complex measure theory and the analysis is done only at the leaf level. Moreover, our analysis is exact at the leaf level and approximate at other levels. No solution for the exact analysis of B-trees has been reported in literature and is a topic of current research.

Another topic of interest is the performance of B-trees under arbitrary probability distribution of keys. When the keys are uniformly distributed, the distribution of the nodes (i.e., how many keys each node contains) is also uniform. The previous assumption considerably simplifies the calculations. When the keys are distributed following an arbitrary distribution, the distribution of the nodes are no longer uniform. Moreover, the distribution of the nodes may or may not follow the distribution of keys. Consequently, the problem becomes more complicated. No solution to this problem has been reported either.

A problem we encountered was caused by the process scheduling policy of the Sequent computer on which the simulations were done. The scheduling policy is not reported in full detail in the manuals and the total elapsed time becomes unpredictable when the number of processes exceed the number of processors. Analytical study of the processes-to-processor ratio on the time of operation is also a topic of future research.

Bibliography

- [BaR87] B. Badrinath and K. Ramamritham, "Semantics Based Concurrency Control: beyond commutativity", *Proceedings of the Third International Conference on Data Engineering*, Feb. 87.
- [Bae86] R. A. Baeza-Yates, "Expected Behaviour of B+ -Trees", *Technical Report, University Of Chile*, 1986.
- [BaM72] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indices", *Acta Informatica*, vol. 1 (1972), pp. 173-189.
- [BaS77] R. Bayer and M. Schkolnick, "Concurrency of Operations on B-Trees", *Acta Informatica*, vol. 9 (1977), pp. 1-21.
- [Bay77] R. Bayer, "Prefix B-Trees", *ACM Trans. Database Systems*, vol. 2 (1977).
- [BeG81] P. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *Computing Surveys*, vol. 13, 2 (Jun. 81).
- [BSR83] P. Bernstein, D. Shipman and J. Rothnie, "Concurrency Control in a System For Distributed Databases", *ACM Trans. Database Systems*, vol. 8, 1 (Jun. 83).
- [BGL83] P. A. Bernstein, N. Goodman and M. Y. Lai, "Analysing Concurrency Control Algorithms When User and System Operations Differ", *IEEE Transactions on Software Engineering*, vol. SE-9, 3 (May 83).
- [BHG87] P. Bernstein, V. Hadzalicos and N. Goodman, in *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

- [Bil87] A. Biliris, "Operation Specific Locking in B-trees", *Proceedings of the 6th ACM SIGMOD Symposium on Principles of Database Systems*, San Diego, CA, Mar. 87, pp. 159-169.
- [BDT83] D. Bitton, D. DeWitt and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach", *Proceedings of the Conference on VLDB*, Florence, Italy, Oct. 1983, pp. 8-19.
- [CaM86] M. J. Carey and W. A. Muhanna, "The Performance of Multiversion Concurrency Control Algorithms", *ACM Transactions on Computer Systems*, vol. 4, 4 (Nov. 86), pp. 338-390.
- [CBT74] D. Chamberlain, R. Boyce and I. Traiger, "A Deadlock Free Scheme for Resource Allocation in a Database Environment", *Information Proc*, vol. 74 (1974), North Holland.
- [CDK85] H. T. Chou, D. J. DeWitt, R. H. Katz and A. C. Klug, "Design and Implementation of the Wisconsin Storage System", *Software - Practice and Experience*, vol. 15, 10 (Oct. 85), pp. 943-962.
- [Com72] D. Comer, "The Ubiquitous B-Tree", *Computing Surveys*, vol. 11, 1 (1972), pp. 121-137.
- [EZG82] B. Eisenbarth, N. Ziviani, G. Gonnet, K. Melhorn and D. Wood, "The Theory of Fringe Analysis and Its Application to 2-3 Trees and B-Trees", *Information and Control*, vol. 55 (1982).
- [Ell80] C. S. Ellis, "Concurrent Search and Insertion in 2-3 trees", *Acta Informatica*, vol. 14, 1 (1980), pp. 63-86.
- [Fel68] W. Feller, in *An Introduction to Probability Theory and Applications (I & II)*, John Wiley, 1968.

- [Gra89] G. Graefe, "Volcano: An Extendible and Parallel Dataflow Query Processing System", *Technical Report, Oregon Graduate Center*, Jun. 1989.
- [GuS78] L. J. Guibas and R. Sedgewick, "A Dichromatic Framework For Balanced Trees", *Proceedings, 19th Annual Symposium on Foundations Of Computer Science*, 1978, pp. 8-21.
- [HaR84] T. Haerder and A. Reuter, "Transaction Oriented Database Recovery", *ACM Computing Surveys*, vol. 15, 4 (Dec. 84).
- [HwB81] K. Hwang and F. A. Briggs, in *Computer Architecture and Parallel Processing*, McGraw Hill, 1981.
- [Knu71a] D. E. Knuth, *The Art of Computer Programming Vol. 1, First Edition*, 1971.
- [Knu71b] D. E. Knuth, *The Art of Computer Programming Vol. 2, First Edition*, 1971.
- [Knu73] D. E. Knuth, *The Art of Computer Programming Vol. 3, First Edition*, 1973.
- [Koh81] W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", *Computing Surveys*, vol. 13, 2 (Jun. 81).
- [KuR81] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Trans. Database Systems*, vol. 6, 2 (1981).
- [Kus73] K. Kuspert, "Storage Utilization in B-Trees with a Generalized Overflow Technique", *Acta Informatica*, vol. 19 (1973).
- [KwW79] Y. S. Kwong and D. Wood, "Concurrency in B-trees, S-trees and T-trees", *Technical Report 79-CS-17, MacMaster University*, May 79.
- [KwW80a] Y. S. Kwong and D. Wood, "On B-trees: Routing Schemes and Concurrency", *Proceedings of the 1980 ACM/SIGMOD International Conference on Management of Data*, 1980.

- [KwW80b] Y. Kwong and D. Wood, "On B-Trees: Routing and Concurrency", *Proceedings 1980 SIGMOD Management of Data*, 1980.
- [KwW82] Y. S. Kwong and D. Wood, "A New Method For Concurrency in B-Trees", *IEEE Transactions in Software Engineering*, vol. SE8, 3 (May 1982).
- [Lam77] L. Lamport, "Concurrent Reading and Writing", *Comm. ACM*, vol. 22, 11 (1977), pp. 806-811.
- [Lam78] L. Lamport, "Time, Clocks and Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, 7 (Jul. 1978).
- [LeY81] P. L. Lehman and S. B. Yao, "Efficient Locking For Concurrent Operation on B-trees", *ACM Trans. Database Systems*, vol. 6, 4 (Dec. 81), pp. 650-670.
- [Leu84] C. H. C. Leung, "Approximate Storage Utilization of B-Trees: A Simple Derivation and Generalizations", *Information Processing Letters*, vol. 19, 4 (Nov. 84).
- [Lom77] D. B. Lomet, "Process Structuring, Synchronization and Recovery using Atomic actions", *Proc. Conference on Lang. Design for Reliable Software*, vol. 12, 3 (Mar. 1977).
- [MaS80] D. Maier and S. C. Salveter, "Hysterical B-Trees", *Information Processing Letters*, vol. 12, 4 (1980).
- [MaS87] D. Maier and J. Stein, "Development and Implementation of an Object Oriented Database System", in *Research Directions in Object Oriented Programming*, B. Shriver and P. Wegner (eds.), MIT Press, 1987.
- [Mol83] H. G. Molina, "Using Semantic Knowledge for Transaction Processing in Distr. Systems", *ACM Trans. Database Systems*, vol. 8, 2 (Jun. 83).

- [Pap79] C. Papadimitrou, "On Serializability Of Concurrent Database Updates", *J. ACM*, Jun. 1979.
- [Par77] J. R. Parr, "An Access Method For Concurrently Sharing a B-Tree Based Indexed Sequential File", *Technical Report 96, Dept. Of Computer Science*, Apr. 1977.
- [RaM89] M. V. Ramakrishna and P. Mukhopadhyay, "Analysis of Bounded Disorder File System", *Proceedings of the ACM Principles of Database Systems*, 1989.
- [RaL89] R. Ramakrishna and P. A. Larson, "Perfect Hashing with Composite Perfect Hashing", *ACM Trans. Database Systems*, vol. 10, 2 (May. 89).
- [Rao89] G. Rao, Personal Communication, 1989.
- [RiS77] D. Ries and M. Stonebraker, "Effect of Locking Granularity on Database Management Systems", *ACM Trans. Database Systems*, vol. 2, 3 (Sep. 77).
- [Rub81] R. Rubinstein, in *Simulation and the Monte Carlo Method*, John Wiley, 1981.
- [Sag86] Y. Sagiv, "Concurrent Operations in B*-Trees. With Overtaking", *Journal Of Computer and System Sciences*, vol. 33, 2 (1986), pp. 275-296.
- [Sam76] B. Samadi, "B-Trees in a System of Multiple Users", *Information Processing Letters*, vol. 5, 4 (Oct. 76).
- [SiP87] A. Siberschatz and P. Peterson, in *Operating System Principles*, Addison Wesley, 1987.
- [SiK82] A. Silbershatz and Z. Kedem, "A Family of Locking Protocols for Database Systems", *IEEE Transactions on Software Engineering*, vol. 8, 6 (Nov. 82).
- [SiK83] A. Silbershatz and Z. Kedem, "Locking Protocols: From Exclusive to Shared Locks", *J. ACM*, vol. 30, 4 (Oct. 83).

- [Wri85] W. E. Wright, "Some Average Performance Measures for the B-Trees", *Acta Informatica*, vol. 21, 6 (1985), pp. 541-557.
- [Yao79] A. Yao, "On Random 2-3 Trees", *Acta Informatica*, vol. 9, 4 (1979), pp. 171-181.
- [87] "Sequent Guide to Parallel Programming", *Users Manual*, 1987.

Biographical Note

The author born on the 6th of January, 1966. He attended the St. Xavier's Collegiate School from 1971 to 1984 and matriculated with a science major. He then joined the Indian Institute of Technology, Kharagpur in 1984 to pursue undergraduate studies in Computer Science and Engineering. After completion of undergraduate work, he came to the United States of America to pursue further research. During the first year of graduate study in the Oregon Graduate Center, he held a Research Assistantship. He graduated with an M.S. in less than one year, after which he leaves O.G.C for India.

Other than the topic of this dissertation, the author has broad interests on various topics, including the Object Oriented paradigm, language design, compilation techniques and centralized and distributed database systems.