

Concurrency in Interpolation Based Grid Files

by

Jalal Mohammad Abdul Ghaffar

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

INFORMATION AND COMPUTER SCIENCE

June, 1988

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 1355690

Concurrency in interpolation based grid files

Abdul Ghaffar, Jalal Mohammad, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1988

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

**CONCURRENCY IN INTERPOLATION
BASED GRID FILES**

BY

JALAL MOHAMMAD ABDUL GHAFAR

**A Thesis Presented to the
FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA**

**In Partial Fulfillment of the
Requirements for the Degree of**

**MASTER OF SCIENCE
In
COMPUTER SCIENCE**

JUNE 1988

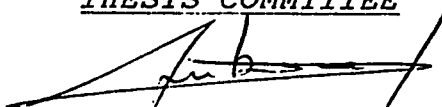
**LIBRARY
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
Dhahran - 31261. SAUDI ARABIA**


**KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA**

COLLEGE OF GRADUATE STUDIES

This thesis, written by **JALAL MOHAMMAD ABDUL GHAFAR** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE in COMPUTER SCIENCE**.

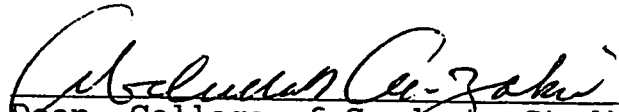
THESIS COMMITTEE


Thesis Advisor


Member


Member


Department Chairman


Dean, College of Graduate Studies

6/11/1985
Date



to my mother, father, brothers and sisters.

ACKNOWLEDGMENT

Acknowledgment is due to the King Fahd University of Petroleum and Minerals for providing the opportunity to carry out this research work.

I would like to express my appreciation to *Dr. Mohammad Oukse1* who served as my major advisor and whose guidance and support encouraged me to complete this research. I wish to thank the other members of my Thesis Committee, *Dr. Mohammad G. Khayat* and *Dr. Mohammad Al-Suwaiyel*.

THESIS ABSTRACT

NAME OF STUDENT: JALAL MOHAMMAD ABDUL GHAFAR

TITLE OF STUDY : CONCURRENCY IN INTERPOLATION BASED GRID FILES

MAJOR FIELD : COMPUTER SCIENCE AND ENGINEERING

DATE OF DEGREE : JUNE 1988

Concurrency control is the activity of preventing harmful interference among asynchronous parallel processes. In this research work, the problem of supporting concurrent operations in the interpolation based grid files is studied. A systematic approach that demonstrates an effective method for detecting conflict between processes is defined. A scheduling system that exploits the numbering property of the structure and employs the conflict detecting method is presented. The scheduler achieves a higher degree of concurrency than concurrency mechanisms developed for B-trees on the average case and the same degree of concurrency in the worst case. Efficient algorithms for concurrent operations in two structures; namely Multi-dimensional Linear Hashing and Interpolation-Based Index Maintenance are presented. Both of these structures are extensions of Linear hashing to the multi-dimensional case. The concurrent scheme presented is an adaptation of the one proposed for linear hashing. The algorithms include searching for, inserting, and deleting data elements. These algorithms support a high degree of concurrency and are shown to be correct based on the restrictions imposed by the compatibility scheme.

MASTER OF SCIENCE DEGREE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
Dhahran, Saudi Arabia

June 1988

خلاصة الرسالة

اسم الطالب : جلال محمد عبدالغفار .

عنوان الرسالة : التزامن في الملفات الشبكية المعتمدة على التوليد .

التخصص : علوم وهندسة الحاسب الآلي

تاريخ الشهادة : يوليو (حزيران) ١٩٨٨ م .

يلعب التحكم بالتزامن دوراً كبيراً في منع الإتصال غير المرغوب بين العمليات المتوازية غير المتزامنة . يقدم هذا البحث دراسة مشكلة تزامن العمليات في الملفات الشبكية المعتمدة على التوليد كما يقدم تعريفاً نظامياً يوضح طريقة اكتشاف التعارض بين العمليات المتزامنة ، ويقدم البحث نظاماً جدولياً يبين فيه خصائص التركيب الرقمية باستخدام طريقة اكتشاف التعارض . ويتضح أن درجة التزامن التي يستقبلها الجدول المقترح اكبر من تلك المطورة للتركيب المسمى (أشجار B) في الحالات العامة ، وتكون بنفس الدرجة في اسوأ الحالات .

ويقدم البحث خوارزميات فعالة لعمليات التزامن في نوعين من التراكيب هما تركيب البعثة الخطية متعدد الابعاد وتركيب التوليد باعتماد صيانة المعاملات . علماً ان هذين التركيبين هما امتداد من البعثة الخطية إلى حالة متعدد الابعاد . وتعتبر طريقة التزامن المقترحة تحويراً لتلك المستعملة في البعثة الخطية . وتتضمن الخوارزميات : خوارزميات البحث والإدخال والإلغاء . وقد تم توضيح صحة هذه الخوارزميات والتي تقدم درجة اكبر من التزامن بالإعتماد على القيود المعتبرة في طريقة التوافق .

درجة الماجستير في العلوم
جامعة الملك فهد للبترول والمعادن
الظهران - المملكة العربية السعودية
يوليو ١٩٨٨ م

TABLE OF CONTENTS

THESIS ABSTRACT	v
Chapter I: THE PROBLEM	1
Chapter II: CONCURRENCY	6
Introduction	6
Semaphores	7
Monitors	8
Locking	9
Lock-coupling	11
Two-phase locking	11
Compatibility and convertibility graph (CCG)	13
Locking Granularity	15
Time-stamping	16
Deadlock	17
Types of processes	19
Concurrency in B-trees	20
Introduction	20
Preliminaries	22
Basic solutions	26
Concurrent operations on B-trees with side branching	29
Concurrency in linear hashing	31
Introduction	31
Concurrent solution	34
Concurrency in Grid file	37
The grid file	37
Concurrency control	40
The concurrent search process	40
Insertion without overflow	41
Insertion with overflow	41
Deletion	43
Merging areas	43
Removing slices	44

Chapter III: INTERPOLATION BASED GRID FILE	45
Introduction	45
The data file	47
The Directory	53
Search Algorithm	56
Insertion and Deletion	57
 Chapter IV: A PRELIMINARY APPROACH TO CONCURRENCY IN IBGF	 65
Uniform Data Distribution	65
Preliminary solution	67
An improved solution	78
Discussion	100
 Chapter V: A CONCURRENCY SCHEME FOR THE IBGF	 102
Nonuniform data distribution	102
Preliminaries	104
A B-tree solution	104
Locking scheme	104
Concurrent behavior	107
An optimistic concurrency control scheme	115
Detecting conflict among processes	118
The model of computation	124
Scheduling of concurrent processes	130
Concurrent behavior of processes	135
Search algorithm sketch	135
The insert algorithm sketch	139
The delete algorithm sketch	140
Improved throughput	142
Further improvement	146
Freedom from deadlock	147
Correctness of file modification	148
 Chapter VI: CONCLUSION AND FUTURE WORK	 151
 References	 154

LIST OF FIGURES

1.	A's update is lost at time t_4	10
2.	B is forced to wait for A's update.	12
3.	Compatibility and convertibility graph (CCG)	14
4.	A deadlock involving two processes.	18
5.	An example of a B-tree structure.	21
6.	A nonleaf and a leave node.	24
7.	Scope of an updater U.	25
8.	Locks compatibility.	28
9.	Sequential linear hashing file.	32
10.	Locks compatibility graph.	35
11.	Grid file structure.	38
12.	Interpolation based grid file Insertion Process. Cyclic	49
13.	Partitioning the search space into regions.	51
14.	Partitioning the search space into regions. Dotted lines	55
15.	EXACT-MATCH algorithm.	59
16.	DECOMPOSE function.	60
17.	INSERT algorithm.	61
18.	Function MERGE algorithm.	62
19.	Function BUDDY algorithm.	63
20.	DELETE algorithm.	64
21.	An interpolation based grid file structure.	68
22.	The compatibility of lock types.	70
23.	Algorithm for the FIND operation.	73

24.	Algorithm for the INSERT operation.	74
25.	Algorithm for the DELETE operation.	75
26.	Algorithm for the SPLIT operation.	76
27.	Algorithm for the MERGE operation.	77
28.	An interpolation based grid file structure.	79
29.	A modified IBGF structure with LOCAL field.	83
30.	Algorithm for search phase.	86
31.	Algorithm for the FIND operation.	87
32.	Algorithm for the INSERT operation.	88
33.	Algorithm for the DELETE operation.	89
34.	Algorithm for the SPLIT operation.	90
35.	Algorithm for the MERGE operation.	91
36.	Example of parallel computation I.	93
37.	Progressive states of the IBGF : stage I.	94
38.	Progressive states of the IBGF : stage II.	95
39.	Example of parallel computation II.	97
40.	An initial state of an IBGF.	98
41.	Progressive states of the IBGF.	99
42.	An IBGF with two directory levels.	103
43.	Algorithm for Locate phase.	110
44.	SEARCH algorithm.	111
45.	Locate and lock an updater scope.	112
46.	INSERT algorithm.	113
47.	DELETE algorithm.	114
48.	A block diagram for an IBGF.	117

49.	Conflictingly overlapping paths.	121
50.	System model.	125
51.	An Ada-like definition of the scheduler data structure.	127
52.	A modified bucket structure.	129
53.	Scheduler algorithm.	133
54.	SEARCH algorithm.	138
55.	INSERT algorithm.	141
56.	DELETE algorithm.	143
57.	A modified version of the scheduler's table. . . .	145

Chapter I

THE PROBLEM

A database system is a collection of five components that interact to satisfy information needs of an enterprise. These five components are hardware, programs, data, people, and procedures. Database technology allows an organization's data to be processed as an integrated whole. It reduces artificiality imposed by separate files for separate applications and permits users to access data more naturally. Data integration offers several important advantages [5,14] :

1. Database processing enables more information to be produced from a given amount of data.
2. Elimination or reduction of data duplication.
3. Creation of program/data independence.
4. The data can be shared.

Restricting large databases to sequential operation is unnatural and inefficient. But if concurrent processes are allowed, then all the usual problems associated with parallel processing will arise. These include for example,

concurrency control, the danger of deadlock, and complexity of verification.

Process synchronization is one of the problems which are not very well understood in the field of database systems although parallel process systems have been investigated intensively in the operating system area [7,8]. When two or more processes execute concurrently, their database operations execute in an interleaved fashion. That is, operations from one process may execute in between two operations from another process. This interleaving may cause processes to behave incorrectly, or interfere, thereby leading to an inconsistent database. This interference is entirely due to the interleaving [6,29]. That is, it can occur even if each process is coded correctly (e.g. the lost update problem).

The term integrity is used in database contexts with the meaning of accuracy, correctness, or validity. The problem of integrity is the problem of ensuring that the data in the database is accurate, that is the problem of guarding the database against invalid updates. Another term that is sometimes used for integrity is consistency. Consistency preservation captures the concept of producing database states that are meaningful. If each process is correct, then an isolated execution of processes will preserve database

consistency. On the other hand, an interleaved execution of these correct processes may produce an inconsistent database state.

Concurrency control is the activity of coordinating the actions of processes that operate in parallel and access shared data in order to safeguard the consistency of the database [6]. The goal of concurrency control is to ensure that processes execute atomically, meaning that, each process accesses shared data without interfering with other processes. Processes will produce a correct view of the database if they are executed in isolation (i.e. processes are executed serially). It is the responsibility of the concurrency control scheme to ensure that the interleaved execution of such processes looks like some serial execution of the same processes. Many techniques have been developed for solving the problem, including locks, semaphores, monitors and time-stamping [7,8,6].

Concurrency control protocols may lead to problems of their own when they are not designed properly, in particular the problem of deadlock. Deadlock is a situation in which two or more processes are in a simultaneous and mutual wait state. In addition, the synchronization policies used should allow a high degree of concurrency. Hence, the problems of how best to introduce concurrency and to what extent and at

what cost should the degree of concurrency be maximized are challenging and interesting topics for investigation.

Concurrency control schemes have been developed for databases that are stored in terms of some popular data structures such as B-trees [1,17,18,28,26], cartesian product files [23], binary search trees [15,20], dynamic linear hashing files [10], and grid files [27]. In this research work, we investigate how appropriate are these existing concurrency control mechanisms when applied to a structure such as Interpolation Based Grid File (IBGF). Then the properties of IBGF are exploited to devise a specific concurrency control scheme which substantially improves the degree of parallelism.

The remainder of this thesis is organized as follows. Chapter II contains a literature survey on concurrency, particularly, on concurrency control algorithms developed for some popular data structures. In chapter III we present the structural properties of the interpolation based grid file. The discussion of designing a concurrency scheme for IBGF is divided into two main parts. The first part, in chapter IV, deals with the file structure when the data is uniformly distributed over the search space. In this part two schemes are presented. The first scheme is discussed here solely for the purpose of identifying the major

problems posed by multiple accesses and updates. We then address these shortcomings in the second scheme. This latter scheme is an adaptation of the concurrency schemes designed for linear and extendible hashing files. It eliminates some of the problems caused in the first one by the restructuring operations, which result in degradation in the degree of concurrency achievable. The second part, in chapter V, discusses the nonuniform data distribution case, in which it was found that the IBGF simulates some of the operational properties of B-trees. The IBGF structure has been shown to work smoothly when some of the concurrency protocols, which were developed for B-trees, are applied. The interpolation based grid file partitions the space embedding the data records until the number of records (or tuples in a partition) does not exceed the size of a page. Each subspace is assigned dynamically a unique number that stands as a surrogate for each subspace and its properties. Unfortunately, B-tree concurrency protocols do not possess such a numbering property. A new concurrency scheme is developed to exploit this dynamic mapping which results in a performance similar to the B-tree solution in the worst case and a better degree of concurrency in the average case. Finally, chapter VI summarizes the work.

Chapter II

CONCURRENCY

2.1 *Introduction*

Processes are concurrent if they exist at the same time [8]. Asynchronous concurrent processes require occasional synchronization and cooperation. Given a correct state of the database as input, an individually correct process will produce a correct state of the database as output if executed in isolation. Even if all processes are individually correct in this sense, however, it is still possible in a shared (multi-user) system for processes that execute concurrently to interfere with one another in such a way as to produce an overall result that is not correct. As a result, access to shared data should be controlled to get rid of the interference between processes. Processes should mutually exclude each other from accessing shared data simultaneously. Such interference can take many forms. One of which is the "lost update" problem [6]. Enforcing mutual exclusion is one of the key problems in concurrent environment. Many solutions have been devised [8,7,6], some software solutions and some hardware solutions; some rather

low-level and some high-level; some requiring voluntary cooperation among processes, and some demanding rigid adherence to strict protocols.

It is clear then that in a multi-user environment some sort of concurrency control mechanism is needed in order to avoid interference problems. Such mechanisms can be implemented by using semaphores, monitors, locking and time-stamping. These techniques will be discussed below. The usage of locking as a synchronization construct to control concurrent access to data records which are organized in terms of some data structures such as B-trees and Grid files ... etc, will be discussed also.

2.2 Semaphores

A semaphore is a primitive concurrency control construct [7,8]. It is a protected variable whose value can be accessed and altered only by the indivisible operations P and V and an initialization operation. Binary semaphores can assume only the value 0 or the value 1. Counting semaphores can assume nonnegative integer values. Semaphores may be used to implement a block/wakeup synchronization mechanism or allocation of resources from a pool of identical resources, as in the case of counting semaphores [7].

2.3 Monitors

A monitor (scheduler) is a concurrency construct that contains both the data and procedures needed to perform allocation of a particular shared resource or group of shared resources [7]. To accomplish a resource allocation function, a process must call a particular monitor entry. Many processes may want to enter the monitor at various times. But mutual exclusion is rigidly enforced at the monitor boundary. Only one process at a time is allowed to enter. Processes desiring to enter the monitor when it is already in use must wait. This waiting is automatically managed by the monitor. The data inside the monitor may be global to all procedures within the monitor or local to a specific procedure. If the process calling the monitor entry finds the resource already is allocated, the monitor procedure calls WAIT. The process calling wait is made to wait outside the monitor for the resource to be released. Eventually, the process that has the resource will call a monitor entry to return the resource to the system. There may be processes waiting for the resource, so the monitor entry calls SIGNAL to allow one of the waiting processes to acquire the resource. To ensure that a process waiting for a resource eventually does get it, the monitor gives priority to the waiting process over a new requesting process attempting to enter the monitor. Processes may need

to wait for many different reasons. So, a separate condition variable is associated with each distinct reason that a process might need to wait. When a condition variable is defined, a queue is established. A process calling wait is inserted into the queue; a process calling signal causes a waiting process to be removed from the queue. Different priority schemes can be used to regulate insertions into and removing processes from the queue.

2.4 Locking

Locking is a concurrency technique by which concurrent access to shared objects is regulated [6,29,5]. Locking can be used at the level of records, blocks(pages), files and the entire database. Let us consider the example shown in Figure 1, the essential problem is that process A and process B are both updating R on the basis of its initial value, that is, neither one is seeing the output of the other. To prevent this situation, a typical concurrency control mechanism might do the following :

1. It could prevent B from reading R on the ground that A already has seen R and may therefore be going to update it. Process B is forced to see the updated value.

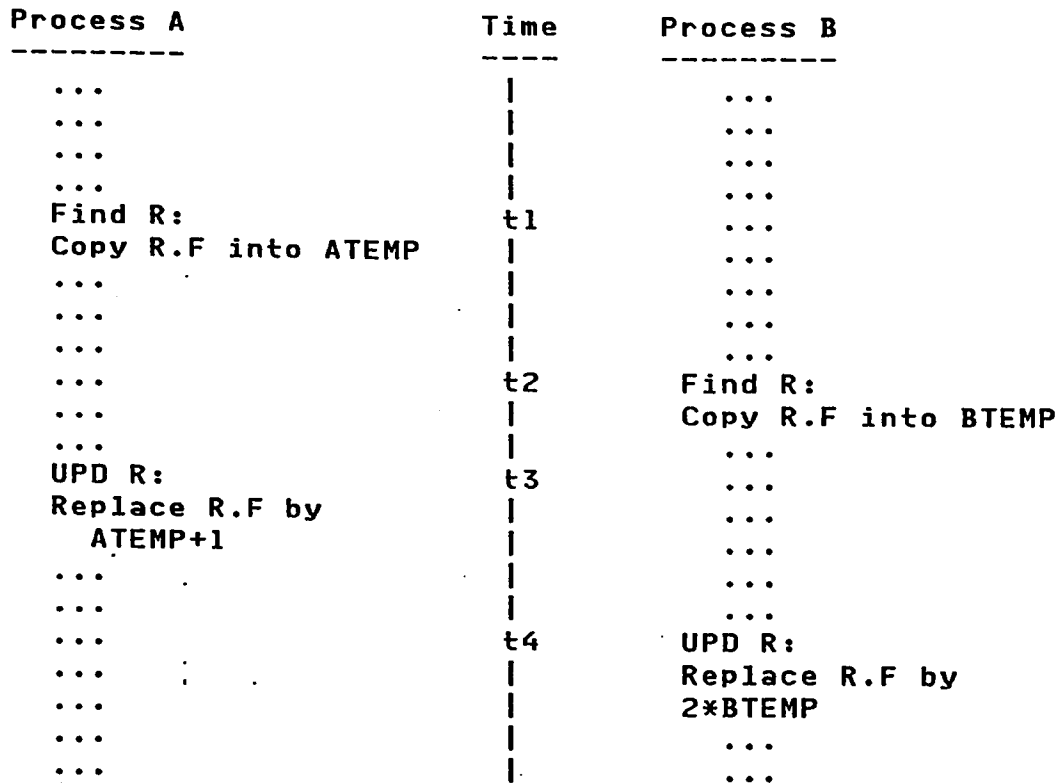


Figure 1: A's update is lost at time t4.

2. It could prevent the process A from updating R on the ground that process B already has seen R before the update.

The above two cases can be handled by locking as illustrated in Figure 2.

2.4.1 Lock-coupling

It is a locking scheme that is very useful in controlling concurrency in tree structures. In this scheme, a process which has locked a set of nodes π should not unlock all nodes in π until its request to lock a set of other nodes α is granted. Usually α consists of either the ancestors or the descendants of π depending on the direction in which the process is advancing.

2.4.2 Two-phase locking

Definition : A given interleaved execution of some set of processes is said to be **serializable** if and only if it produces the same result as some serial execution of these processes.

Definition : A process is said to obey **two-phase locking** if it can not acquire more locks if it has released a lock on some objects.

Processes which obey two-phase locking protocol proved to be serializable [6,12].

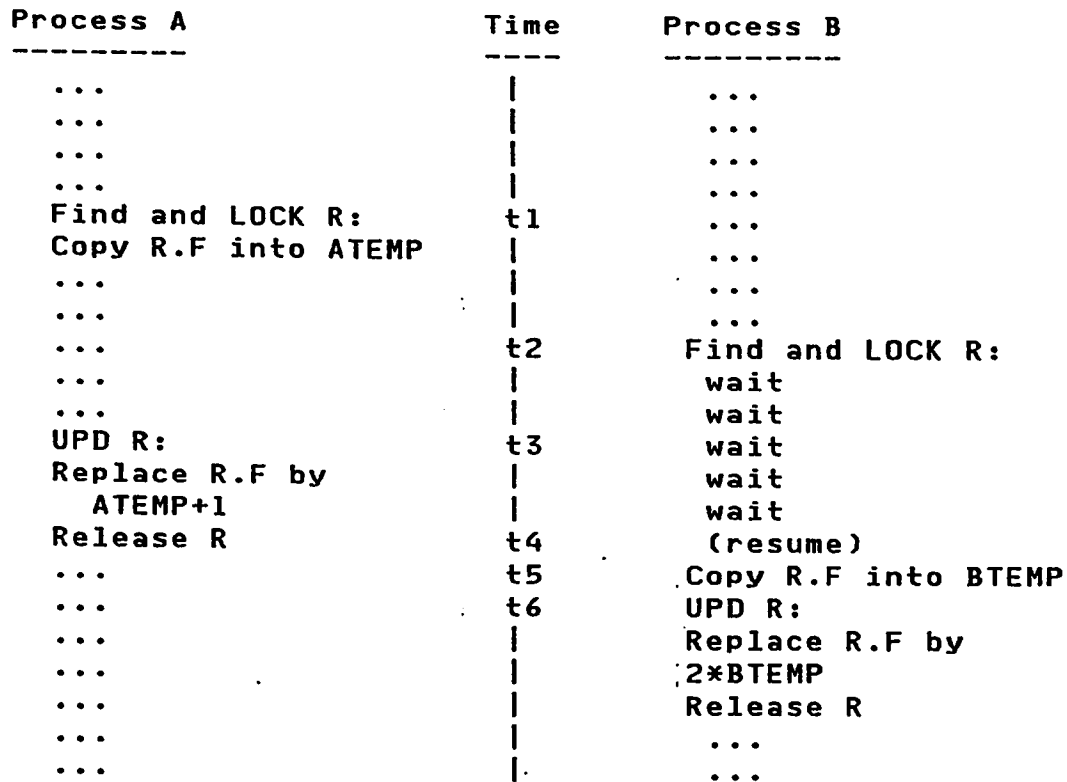


Figure 2: B is forced to wait for A's update.

2.4.3 Compatibility and convertibility graph (CCG)

In order to regulate concurrent access we need a concurrency control which specifies the type of locks to be used by processes operating on some objects. A **Compatibility and Convertibility Graph (CCG)** [1] specifies relations which must hold among the various types of locks on an object. The CCG, which is shown in Figure 3, is a directed graph whose vertices are labeled with lock types and edges are used to represent the compatibility and convertibility relations among the locks. For any two vertices α and β a solid edge directed from α to β means that a process with β -lock on an object would permit another process to put an α -lock on that object. A broken edge from α to β indicates that a process holding an α -lock on an object may convert it into β -lock. Two isolated vertices indicate that a process holding a lock of the first vertex type on an object would not permit another process to put a lock of the second vertex type on that object.

Processes can manipulate the locks via three types of indivisible operations, Lock, Unlock and Convert. When a process requests an α -lock on an object B it must execute a lock operation $Lock(\alpha, B)$. If the granting of such a request does not violate the compatibility relation defined by CCG then the process is allowed to continue, otherwise, it is

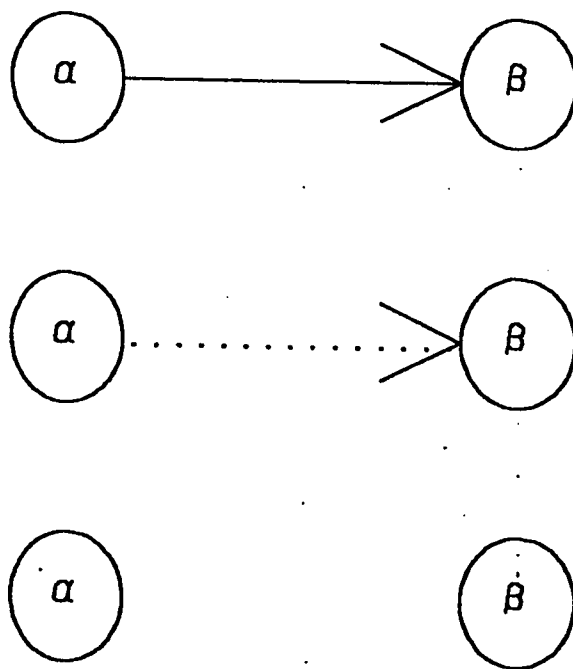


Figure 3: Compatibility and convertibility graph (CCG)

put to sleep in a queue associated with B. To release an α -lock on an object B a process must execute an Unlock operation, $Unlock(\alpha, B)$, which relinquishes the lock and wakes up some sleeping processes in the queue according to the CCG and some fair scheduling discipline. In addition, a process may convert a lock it is holding on an object B from one type into another. This action can be denoted by $Convert_lock(\alpha, \beta, B)$ which converts an α -lock on B into β -lock. If the conversion is granted then the process continues, otherwise the operation is undefined.

2.4.4 Locking Granularity

Two general options for locking appear feasible : physical locks on records, pages, segments, files, etc; and predicate locks (logical locks) can be set on the exact portion of the database required which is determined by a predicate or qualification [25]. As usual, there is a trade off : the finer the granularity, the greater is the concurrency; the coarser, the fewer are the locks to be set and tested and the lower the overhead [6]. It has been shown that a large number of granules, corresponding to locking a page or record is extremely costly [25]. While in the case of predicate locking, only a small number of locks must be maintained which is proportional to the number of active processes and not to the size of the database. As a result,

any advantages due to additional parallelism are outweighed by such cost introduced by large number of granules.

2.5 *Time-stamping*

The basic idea behind time-stamp technique is to assign a unique identifier to a process to distinguish it from other processes. A fundamental difference between time-stamping and locking techniques in general is that, locking synchronizes the interleaved execution of a set of processes in such a way that it is equivalent to some serial execution of those processes, where as, time-stamping synchronizes that interleaved execution in such a way that it is equivalent to a specific serial execution [6]. A process is restarted (rolled back) if it asks to see a record that has already been updated by a younger process, or it asks to update a record that has already been seen or updated by a younger process. The system should keep information on every record about the youngest process that has successfully set addressability on and the youngest process that has successfully updated the record.

2.6 Deadlock

A process is said to be in a state of deadlock, if it is waiting for a particular event that will not occur. Consider Figure 4, which shows a deadlock involving two processes. The problem of deadlock has been extensively studied, and various deadlock detection algorithms and deadlock avoidance protocols have been developed [7,8,6]. The four necessary conditions for a deadlock to exist are :

1. Mutual exclusion : processes claim exclusive control of their resources.
2. Wait for : processes may hold resources while waiting for additional requested resources to be allocated.
3. No-preemption : resources may not be removed from processes.
4. Circular wait : means that a chain of processes exists in which each process holds a resource being requested by another process that holds a resource being requested by another process, etc.

Deadlock detection is the process of actually determining that a deadlock exists, and of identifying the processes and resources involved in the deadlock. To facilitate the detection of deadlocks, a popular notation is used in which a directed graph indicates resource allocations and requests

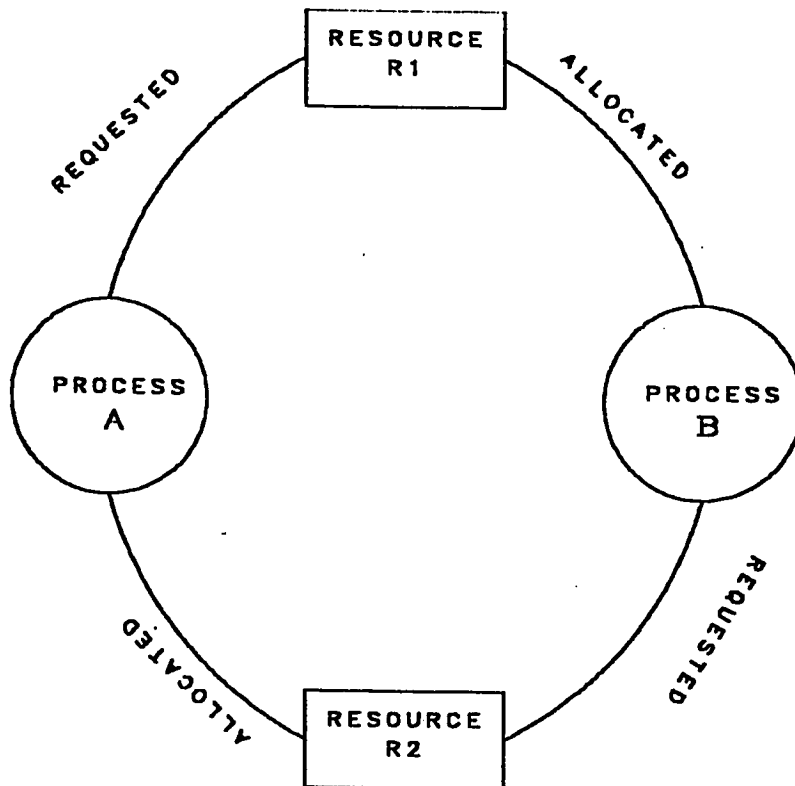


Figure 4: A deadlock involving two processes.

[7,8]. One technique useful for detecting deadlock involves graph reductions in which the process that may complete and the processes that will remain deadlocked are determined.

Deadlock prevention is the process of removing any possibility of deadlock occurring. It has been shown, that if any of the four necessary conditions is denied, it is impossible for a deadlock to occur. As a result, different strategies for denying various necessary conditions have been proposed [7].

2.7 *Types of processes*

A process P is an application that initiates some actions on the records in the file. An action can be a READ, INSERT, or DELETE operation. We shall use terms FIND, SEARCH, and READ interchangeably. Similarly, we shall use terms process and operation interchangeably. A READ action is as follows : determine if a record K is in the file; if it is then report success, otherwise report failure. An INSERT action is as follows : insert the record $K=(k_0, k_1, \dots, k_{d-1})$ into the file if it is not already there, which will cause an access to only one single data page since all values are specified. A DELETE action is as follows : retrieve the record K and then delete it if it exists. A process P is implemented as a sequence of actions. These actions map the structure from

one state to another. An INSERT or DELETE operation is called an update process. An updater usually goes through the following two phases :

1. Searching phase : locate the appropriate place for adding or removing a record.
2. Restructuring phase : add or remove the record and then rebalance the file structure.

Of course, the second phase may not be necessary in some cases, when an INSERT (a DELETE) process finds that the record is already present (absent).

In permitting an arbitrary number of processes to operate concurrently on a database, we assume that processes are asynchronous, and each of which is progressing at a finite, but undetermined rate.

2.8 Concurrency in B-trees

2.8.1 Introduction

The B-tree, shown in Figure 5, and its variants have been widely used in recent years as a data structure for storing large files of information. Methods for concurrent operations on B-trees have been discussed in [26,18,1,17]. Concurrent operations on B-trees pose the problem of insuring that each operation can be carried out without interfering with other operations .

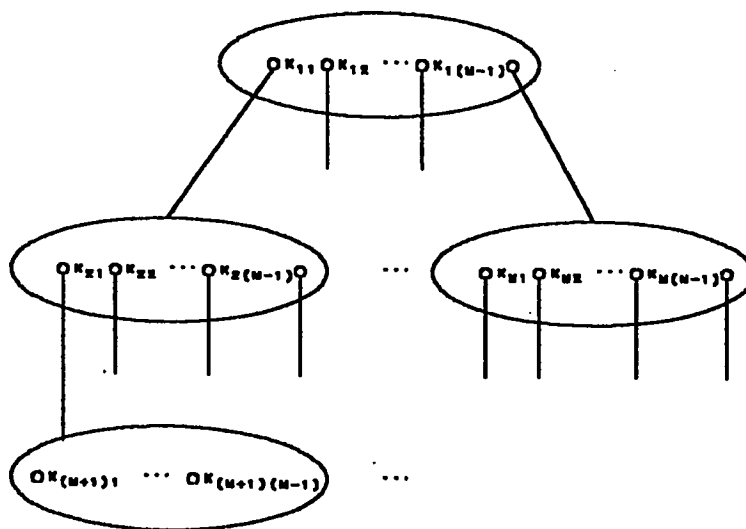


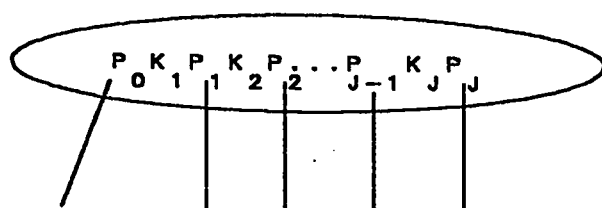
Figure 5: An example of a B-tree structure.

Accessing schemes which achieve a reasonably high degree of concurrency in using B-trees are presented. The schemes are deadlock free. This is achieved by providing a set of strict locking protocols which must be followed by each process accessing B-trees.

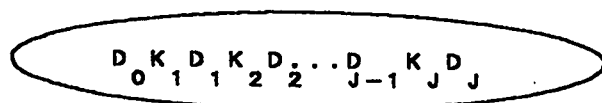
2.8.2 Preliminaries

Definition : A B^+ -tree of order m is a tree which has the following properties :

1. Every node has at most m sons.
2. Every node, except for the root and the leaves, has at least $(m/2)$ sons.
3. The root has at least two sons.
4. A nonleaf node with $j+1$ sons contains j separators and can be represented as shown in part(a) Figure 6, where the P_i 's are pointers to its sons and K_i 's are separators such that all keys in the subtree pointed by $P_{i-1}(P_i)$ are less than or equal to (greater than) K_i 's for $1 \leq i \leq j$.
5. All leaves appear at the same level. The keys appear in the leaves in ascending order when read from left to right. They are the same as nonleaf nodes except that instead of pointers, they contain the data D_{i-1} associated with each key K_i , as shown in part(b) Figure 6.



(A)



(B)

Figure 6: A nonleaf and a leaf node.

The path from the root to a leaf determined by a process on its passage to a leaf is called the **access path** of a process. In many of the tree structures, the following simple observation has been made. For an updater U operating in a sequential environment there exists a node which is the root of a subtree beyond which all changes in the data and structure due to U cannot propagate. It is called a **safe node** for U . To ensure that the sub-tree affected by U is as small as possible the safe node which is deepest in the access path of U is of particular importance. The portion of the access path from the deepest safe node for U to a leaf is called the **scope** of U , which is shown in Figure 7.

Definition : A node in a B-tree is **insertion-safe** if it is unsaturated, (i.e. it has less than $m-1$ keys), and it is **deletion-safe** if it is not minimal (i.e. it has more than $(m/2) - 1$ keys).

Concurrent control schemes proposed for B-trees can be classified into the following two categories :

1. **Type 1 solutions :** the scope for an updater remains invariant during restructuring, which implies that an updater must lock its scope so that no other updater can be in it.

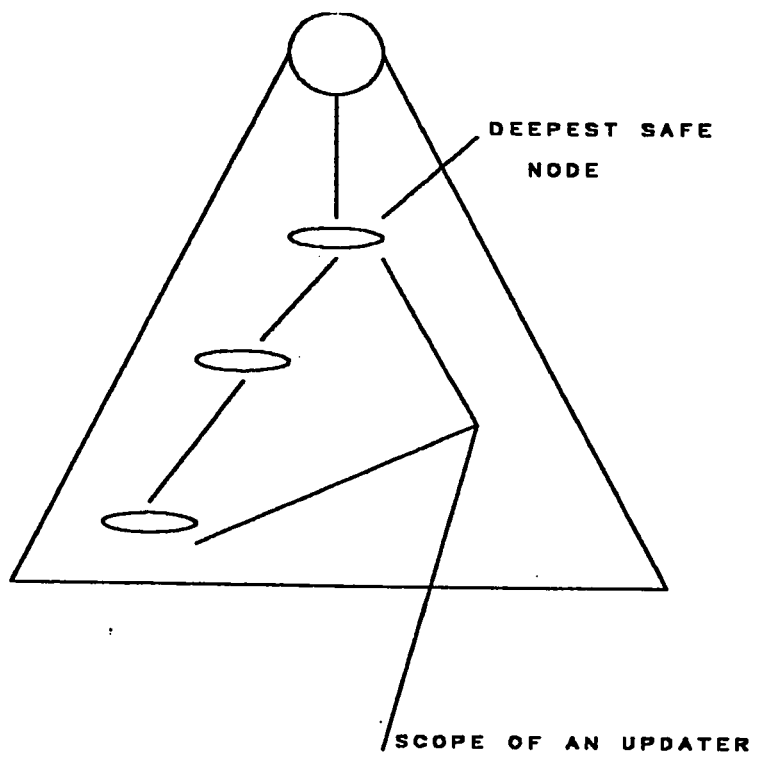


Figure 7: Scope of an updater U.

2. Type 2 solutions : the scope for an updater may change during restructuring. Only the nodes affected by an updater at each restructuring step are locked and made off-limits to other updaters.

2.8.3 Basic solutions

The scope of an updater must be locked before any necessary restructuring can begin. The locking is often done in such a way that no other updaters can be in the scope once it is locked, not even waiting in the queues associated with the nodes other than the deepest safe node. However, readers may or may not be allowed in the scope, depending on the particular proposed solution.

The earliest solution was proposed by Samadi [28] in 1976, based on node search B-trees. In this solution one type of lock is used by both readers and updaters and no two processes can lock a node simultaneously. Locking and unlocking are simply P and V operations on a binary semaphore. Two important policies for locking are stated :

1. Scope locking by updaters : an updater first locks the root and on its passage to the leaves level the appropriate nodes are locked and examined. When a safe node is found all its ancestors are unlocked.
2. Node locking by readers : A reader first locks the root and on its passage to the leaves level it unlocks a node only after it has locked its son.

It should be noted that when these locking protocols are employed, a total ordering is imposed on the processes sharing a common path on their passage to the leaves level.

Three solutions were proposed by Bayer and Schkolnick [1] for leaf search B-trees. Their first solution is very similar to that of Samadi except that there are two types of locks, read-locks and exclusive-locks whose compatibility is given by part (a) Figure 8. Clearly, more than one reader may be reading a node simultaneously and no ordering is imposed on readers. The second solution is based on the observation that restructuring by updaters is seldom necessary-about once every $m/2$ updaters. They propose that updaters should behave just like readers on their passage to the leaves level. On reaching a leaf the updater exclusively locks the leaf and if it is unsafe, the updater releases its lock and repeats the updating with the first solution. Their third solution requires the concurrency control to support three types of locks, read, write and exclusive locks whose compatibility is given by part (b) Figure 8. Readers and updaters in their searching phase use read and write locks respectively. Note that once an updater has write-locked its scope no other updaters can be in it, but readers may be present because read and write locks are fully compatible. However, before an updater begins its restructuring phase,

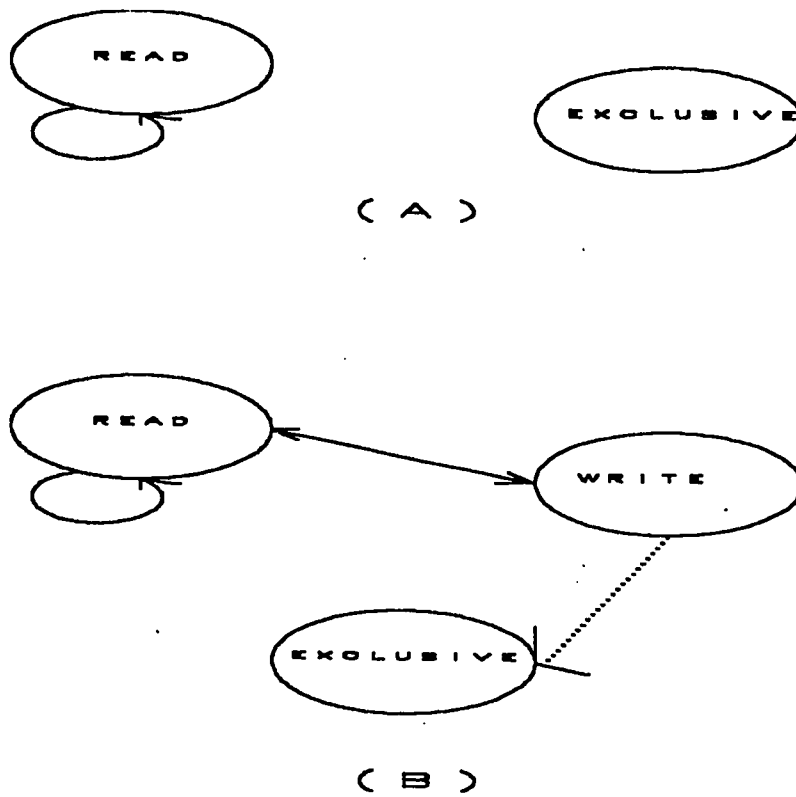


Figure 8: Locks compatibility.

it converts all write-locks into exclusive-locks starting at the deepest safe node. Since exclusive-locks are incompatible with read-locks this ensures that all readers in the scope are drained away.

Two solutions were proposed by Ellis [11]. The first solution is based on the third solution of Bayer and Schkolnick. In the second solution, readers are allowed to be in most of the scope during restructuring. At each restructuring step, the writer exclusively locks only its local scope which consists of two nodes in order to drive off readers. To continue with the next step of restructuring, the local scope is then shifted upward using a lock-coupling technique until the deepest safe node is reached.

2.8.4 Concurrent operations on B-trees with side branching

The new side branching that was proposed by Kwong and Wood [17] is based on the notion of making extra copies of keys and pointers while leaving nodes in the scope of a writer intact until the very last moment. In all proposed solutions in the literature, a key and a pointer are always added to a node at each restructuring step, starting at the leaf of the scope where the pointer is actually the data associated with the key. If a node is not insertion safe it becomes oversaturated and is split into two "halves" which consists

of $(m/2)-1$ and $m-(m/2)$ keys respectively. This leads to a key and a pointer being pushed upwards to be added to the father. After reaching its deepest safe node the writer will have completed its restructuring phase. It is noticed that this approach of actually adding a key and a pointer to every node in the scope requires two additional fields per node for the extra key and pointer since a node can become oversaturated. Moreover, in adding a key and a pointer to a saturated node, keys and pointers must be shifted to make space for the new entries. Such shifting is often redundant because "half" of the nodes has to be copied into a new node and deleted soon afterwards. Instead of actually adding a key and a pointer to a node at each restructuring step, a writer in this solution uses the key to determine whether they should be added to the left or the right "half". It then copies into the appropriate "half" into a newly created node where the key and pointer are added, and pushes a key and a pointer upwards while leaving the node in its scope intact. The writer continues restructuring upwards in this way until the leading node of its scope is reached. It then converts its write-lock on the deepest safe node into an exclusive-lock and adds a key and a pointer. In other words, the writer is simply reading keys and pointers from the nodes of its scope and building a side-branch which is

inaccessible to other processes. A DELETE process in this solution make use also of a side branch to avoid the actual removal of a key and a pointer from a node during node merge.

2.9 *Concurrency in linear hashing*

2.9.1 Introduction

Recently, a number of techniques for dynamic hashing have appeared. A solution that allows concurrency in linear hashfiles based on locking protocols and minor modification in the data structure was proposed by Carla Ellis [10]. The linear hashfile represents a different type of data structure from those of earlier concurrency studies. In particular, it is not a linked structure. It is assumed that the file occupies a contiguous logical address space of primary buckets each capable of holding some number b of records. Collisions (i.e. attempt to insert into a full primary bucket) are handled by creating a chain of overflow buckets associated with that particular bucket address, as illustrated in Figure 9.

The hash function to be applied changes as the file grows or shrinks. Each new hash function assigns new bucket addresses to some records previously placed using the old function. This new hash function is applied to one bucket chain at a time in linear ordering. The resulting

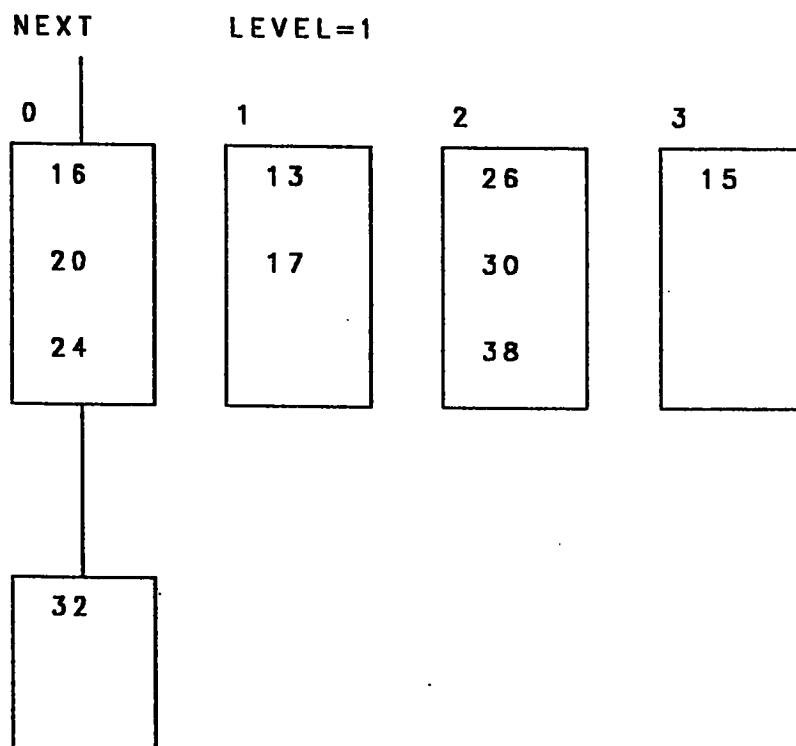


Figure 9: Sequential linear hashing file.

modification in the data structure is called a split and moves some records from the original bucket to a new primary bucket that is appended at the current end of the hashfile. The split operation is applied cyclically.

The function $h_0 : K \rightarrow \{0, 1, \dots, N-1\}$, initially used to load the file and a sequence of functions $h_1, h_2, \dots, h_i, \dots$ such that for any key value K , either $h_i(k) = h_{i-1}(k)$ or $h_i(k) = h_{i-1}(k) + 2^{i-1}N$. . A pointer NEXT indicates which bucket that is next in line to be split. For each pass with a new hash function h_j , the NEXT pointer travels from bucket 0 to $2^{j-1}N$. A variable LEVEL is used to determine the appropriate hash function for FIND, INSERT or DELETE operations using the following procedure :

bucket = $h_{level}(key)$

if bucket < next then bucket = $h_{level+1}(key)$

Splitting causes these variables to be updated as follows :

next = (next+1) mod $(N * 2^{level})$

if next = 0 then level = level + 1

The next bucket in the cycle is split whenever any bucket overflows. This policy is called uncontrolled split. Deletion of records may result in merging buckets, moving

next back, and readjusting level. Merging is performed when an individual primary bucket underflows.

2.9.2 Concurrent solution

Processes can execute FIND, INSERT and DELETE operations on a shared linear hashfile. Processes executing the INSERT and DELETE actions may operate in parallel if they are working on different bucket chains. Processes may not access the two buckets being merged and may not read the value of level and next while the merging process is using them.

Locks are used to control access to the shared variables level and next and to the bucket chains. The primary bucket and all its overflow buckets are locked as a unit. The compatibility of lock types is given by Figure 10. Processes apply lock-coupling technique in which the next component is locked before releasing the lock of the current component. The procedures Split and Merge are concerned with the restructuring of the hashfile. The procedure INSERT and DELETE read-lock next and level and selective-lock buckets with lock coupling. The split operation uses selective-locks. Exclusive-locks are used for merging chains and deallocating old overflow buckets.

Concurrency is enhanced by allowing a searching process to operate in parallel with a split operation, but there must be some means for it to reorient itself when the wrong

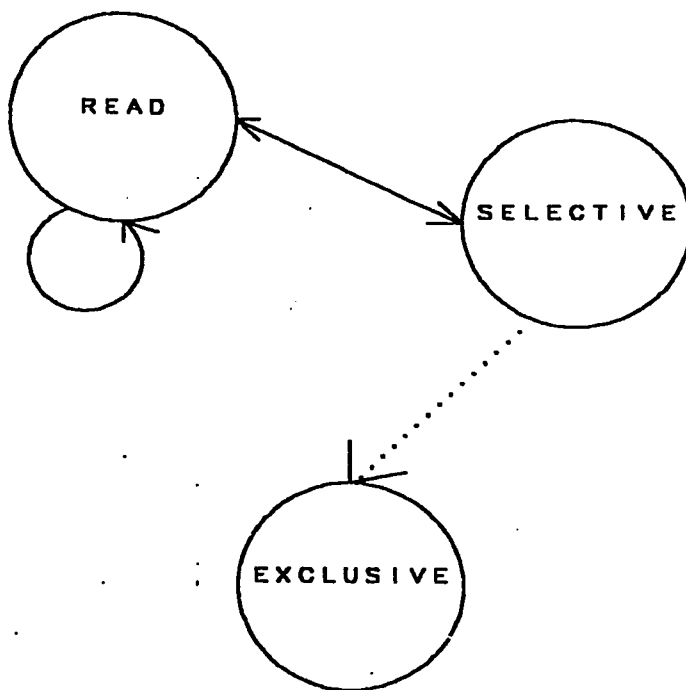


Figure 10: Locks compatibility graph.

chain is reached because of out-of-date root values. In this scheme, each chain includes an additional field `locallevel` that specifies the hash function appropriate to that bucket chain. This field captures the fact that in the most recent split affecting this bucket (not inverted by a subsequent merge), it was the hash function $h_{locallevel}$ that was used to divide up the keys. Basically, the `locallevel` value characterizes the set of key values that can belong in this chain.

A process executing in its searching phase behaves as follows : the root variables are read, and the values seen determine which hash function is to be used initially. Upon gaining access to a bucket, the process checks whether values read from root variables matches the bucket's `locallevel`, and if not, it increments the value of level that was read into a private buffer and then recalculates the address until a match is found. The calculated address at each iteration of the rehashing loop will always be less than or equal to the address of the eventual destination. Thus, the bucket chain in which the desired key belongs should be reachable using this rehashing strategy as long as each address calculated is within the valid address space at the time of access. A process responsible for merge, holds exclusive locks on root and both partners of the merge,

while it makes its changes. The read lock on root held by searching processes prevents a merge from decreasing the size of the address space during the initial bucket access.

During the searching phase of FIND, INSERT and DELETE, locks are placed according to a well-defined ordering. Merges and splits also respect the ordering in requesting their locks. Thus, deadlock cannot occur. Selective locks are placed on the chains during searching phase to serialize writers of the same individual buckets so that only up-to-date information is seen. So, there is no interference between concurrent execution of INSERT and DELETE. Merges and splits are completely serialized with respect to one another by incompatible locks on the root. All affected chains are also locked by a restructuring process for the duration of the step.

2.10 Concurrency in Grid file

2.10.1 The grid file

The grid file is a data structure for partial match or "multikey" retrieval on a file, recently proposed by Nievergelt et al [22]. The grid file, shown in Figure 11, concept consists of a grid directory, which is a multidimensional array kept on disk, linear scales, which are kept in memory, and the actual pages of data records kept on disk

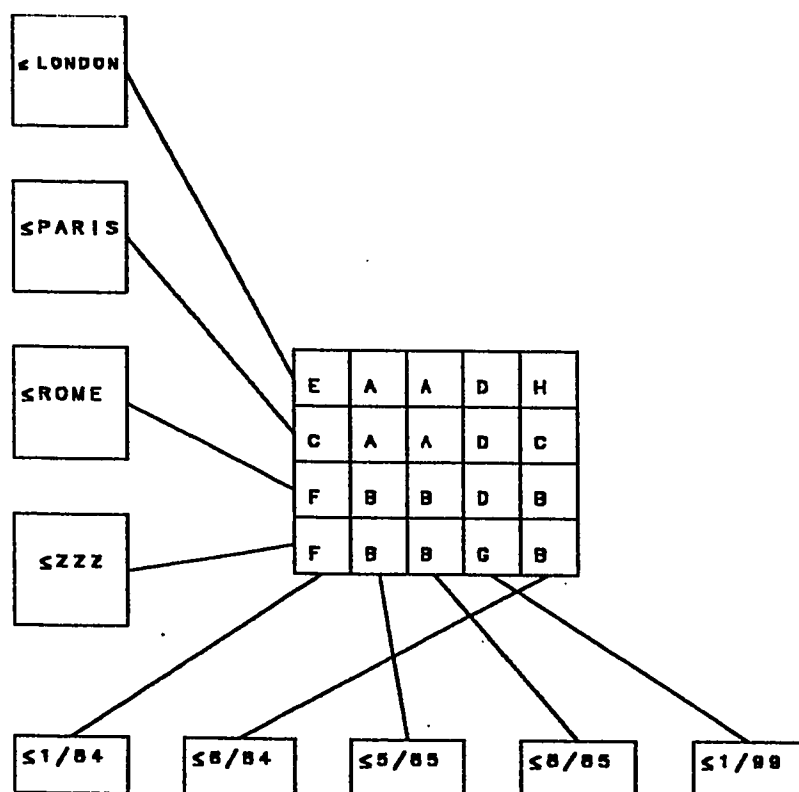


Figure 11: Grid file structure.

The grid directory is an array of K dimensions where K is the number of attributes in the key. Each block or cell in the grid directory contains one address of a page of data records. Several blocks may contain the same address. Each key attribute is assigned a linear scale. The linear scales determine an ordering of the grid directory subscripts. These lists point out which $K-1$ dimensional cross section of the grid directory contains the data pages with a given range of values for a given attribute. There is a constraint on what configurations of blocks in the grid directory may contain the same data page address. They must be CONVEX or RECTANGULAR. When a data page overflows, a new data page is allocated, and the blocks of the grid directory with that address, if there are more than one, are split according to values in one of the attributes, with some retaining the old address and some acquiring the new. The linear scales are not modified. The choice of which data records remain in the old location and which go to the new is determined by the direction in which the split was made. On the other hand, if there is only one block in the grid directory with the address of the overflowing data page, or if the possible split directions do not result in space for a new record, then a new $K-1$ dimensional slice must be added. In this case, one of the linear scales gets a new entry. When a

data page contains too few records, there is a possibility that these records can be merged with those in a data page whose address is adjacent in the grid directory. However, this is only allowed to occur if the union of all of the grid directory blocks with either one of the addresses or the other forms a convex area. The removal of a K-1 dimensional slice is occasionally possible as well. This is allowed whenever two such slices, adjacent in the ordering imposed by one of the linear scales, are identical.

2.10.2 Concurrency control

A solution that allows concurrent access to the grid file based on locking protocols was proposed by Betty Salzberg [27]. This solution uses some of the principles that were used to regulate concurrent accesses in other data structures. One of which is the principle called "draining" [27]. The proposed solution uses also the principle of delaying restructuring when it does not inhibit correctness to do so.

2.10.3 The concurrent search process

The searcher finds the correct positions in each linear scale. This will be a collection of subscripts for each dimension. All of the addresses of data pages in the grid directory whose subscripts are in the collection must be

investigated. No locks are placed by searchers. Certain changes are not made in the grid directory until processes which have old information are drained from the system. A searcher may be aborted if it does not leave the system within a given time, and restarted when changes in the directory are to be made. The locks placed by inserters and deleters do not affect the searchers. Inserters and deleters are considered searchers until a data page lock is requested.

2.10.4 Insertion without overflow

Insertion implies that all values of all the attributes in the key are known, so that exactly one grid block is accessed and exactly one data page is found. If there is enough space in the data page, and the record is not already in the data base, the data page is locked, the record inserted, and the data page unlocked. No locks are placed on the grid directory.

2.10.5 Insertion with overflow

If there is no room for the new record, an overflow procedure is called while the data page remains locked to other updaters. In order to change the grid directory by splitting a collection "A" of blocks with the same address into two such collections, all blocks of A are locked. It is

called a grid area lock. This is to prevent unsafe interactions with processes which are creating new slices or deleting slices. Processes which do not create or delete slices only lock grid areas whose corresponding data pages they have already locked. After a grid area is locked, if there is more than one block in it, it may be split into two areas. The split should be chosen so that the records in the data page are actually distributed into two distinct non-empty collections. If this is not possible, a new slice must be created. If the grid directory area consists of only one block, or all the values were those of one block, before a split can occur, the overflow process must call a process to insert a $K-1$ dimensional slice. This process copies a $K-1$ dimensional slice containing the data block in question and inserts it by adding one cell to one of the linear scales. The $K-1$ dimensional slice must be locked before being copied, so that no changes are made in the original slice while the new one is being created. The new cell in the linear scale must come after the cell representing the slice to be duplicated, so that other processes waiting to lock an area which intersects the old slice may pick up the blocks in the new slice after they are added. The grid area must be unlocked before the slice is locked in order to prevent deadlock. After the new slice is added, and the linear scale

changed, which does not affect searchers, the old slice is unlocked and the new grid area, including both the old and new blocks, and any blocks added by other processes can be locked, and the overflow procedure attempted once again.

2.10.6 Deletion

In order to delete a record, a process simply searches for the correct page, and locks that page, and deletes the record if it is there, and unlocks the page. If the page sparse, the deletion process puts the key of the record deleted on a queue for asynchronous restructuring.

2.10.7 Merging areas

An asynchronous restructuring process may look at the grid blocks areas whose key values have been put on the merge queue by a deleter process. The data page can be checked to see if it is still sparse, and the neighbor areas can be checked to see if they form a rectangular or convex area with a given area, and also corresponding data pages have few enough records to allow merging. The process then locks the two data pages in question, and the smallest grid area block. Then all the data is transferred to one of the data pages and the grid directory is changed.

2.10.8 Removing slices

When a long series of deletions and area merges has produced a great deal of redundancy in the grid file directory, slice deletion may be performed to increase efficiency of range queries, or partial match queries. In this process, a search is made in each linear scale comparing slices corresponding to neighboring pairs of entries. If any duplicate slices are found, they are locked lexicographically and one of them is removed. No data pages are locked. The locking is necessary to insure exact duplication at the time of removal.

Chapter III

INTERPOLATION BASED GRID FILE

3.1 Introduction

Files consist of large collections of individual records usually stored on external memory devices such as tapes, disks or drums. Files are often too large to be brought into the main memory of a computer all at once, and therefore operations on them must be performed through a succession of piece-wise accesses to groups of modest size (i.e. pages or data buckets).

A wide selection of file structures is available for managing a collection of records identified by a single key : sequentially allocated files, tree-structured files of many kinds, and hash files. They allow execution of common files operations such as FIND, INSERT and DELETE, with various degree of efficiency. Older file structures such as sequential files or conventional forms of hash files were optimized for handling static files, where insertions and deletions are considered to be less important than look-up or modification of existing records.

The advent of new file structures [4,21,19,22] which dynamically adapt to the varying collection of data they must store without any degradation of performance and allow efficient access to records, and which allow access based on the value of any one of several attributes or a combination thereof, was a major advance in the study of file structures.

All known file structure techniques appear to fall into one of two categories : those that organize the specific set of data to be stored and those that organize the embedding space from which the data is drawn. The Interpolation Based Grid File (IBGF) not only organizes the set of data records but also the embedding space [24]. The IBGF satisfies the following structural properties :

1. it is symmetric (i.e. multi-key access to records)
2. it is order-preserving. The set of records is partitioned into subsets such that these subsets are totally ordered.
3. it is dynamic (i.e. the structure adapts gracefully to addition and deletion operations without performance degradation).
4. it is clustering (i.e. each bucket contains all the records in a region of the search space that can be decomposed into a maximal set of uniquely identifiable adjacent convex partitions).

3.2 The data file

A data record is a d -dimensional tuple $K = (k_0, k_1, \dots, k_{d-1})$ of value which correspond to attributes A_0, A_1, \dots, A_{d-1} respectively. Each component of the record is scaled and mapped to a rational number in the half open interval $[0, 1)$, a record is thus viewed as a point in the d -dimensional space $U^d = [0, 1)^d$. Let $K = (k_0^0, k_1^0, \dots, k_{d-1}^0)$ be the result of the mapping. As an example consider the two-dimensional case where $D_0 = [0, 5000)$ and $D_1 = [0, 80)$, a record $K = (37500, 10)$ will be mapped to $K^0 = (37500/5000, 10/80)$ or $(0.110, 0.001)$ in binary of $U^2 = [0, 1) \times [0, 1)$.

The construction of an interpolation based grid file is best illustrated by an example. To simplify the discussion only the two-dimensional case is presented. Thus, the data search space is viewed as a rectangle delimited by the cartesian product $D_0 \times D_1$ of attributes A_0 and A_1 domains.

The vector $K = (k_0^0, k_1^0)$ which is the mapping of the original record refers to the coordinates of this search space. The IBGF initially consists of a single partition which embeds the whole search space. An insertion of a record to the database will be represented by exhibiting a dot in the

graph, see part(a) Figure 12. We shall assume that the data file bucket capacity $b_0=2$ and each partition may thus contain at most 2 dots. When two more records are inserted an overflow will occur. Splitting the search space is necessary to maintain the data file bucket limit. Different approaches can be used to split an overflowing partition. One of them is a cyclic splitting in half along the various axes, as shown in part(b) Figure 12, which we adopt here for simplicity.

Definition : Given a set of intervals along the i -th axis of U^d define d as follows :

$$I(l_i) = \{I_i(k) \mid I_i(k) = [k/2^{l_i}, (k+1)/2^{l_i}]\}$$

$$\text{where } 0 \leq k < 2^{l_i}$$

then l_i is called the interval partition level along axis i . Note $I_0(0) = [0,1)$ and the number of intervals in

$$I(l_i) \text{ is } 2^{l_i}.$$

Definition : The search space partition level l is defined as the summation of interval partition levels along the d axes forming the search space (i.e., $l = \sum_{i=0}^{d-1} l_i$). The

number of partitions in the search space is 2^l .

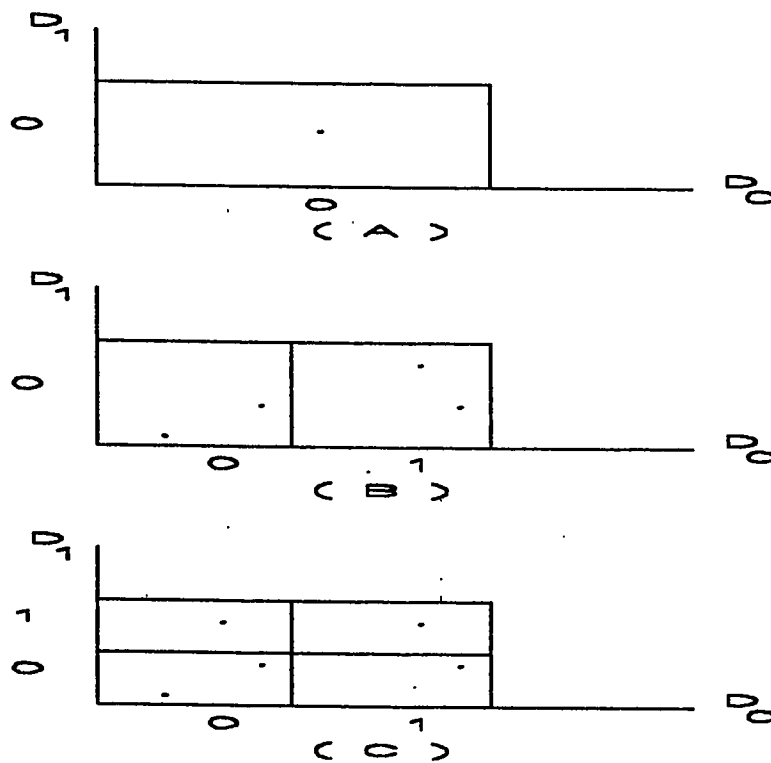


Figure 12: Interpolation based grid file Insertion Process. Cyclic splitting (2-dimensional case), bucket size $b = 2$.

An additional insertion to partition #1 will cause a split along axis #1. A partition can simply be represented by its coordinates (i,j) . So in the example, see part(c) Figure 12 on page 49, the possible pairs are $(0,0), (0,1), (1,0), (1,1)$. A pair of coordinates represents the leading binary digits of the fraction part of K_0^0 and K_1^0 respectively. In other words, coordinates are simply prefixes of elements located in the subspace they determine. The length of these prefixes is exactly the interval partition level along the corresponding axis. For example, if we assumed the partition level along axis zero and axis one are 2 and 1 respectively, then the search space will look like Figure 13.

It is possible to deduce a simple one-to-one storage mapping, if partitions are split in a linear order. Let $C = (c_0^0, c_1^0, \dots, c_{d-1}^0)$ denote the coordinates of the partition where K is contained. Each c_{ij}^0 indicates the j -th binary digit of c_i^0 starting from right, and assuming a cyclic splitting policy. Then the number of the partition in which a record $K = (k_0, k_1, \dots, k_{d-1})$ may be contained is given by :

$$M(k, l) = \sum_{i=0}^{d-1} 2^i \sum_{j=0}^{l_i-1} 2^{d(l_i-1-j)} c_{ij}^0$$

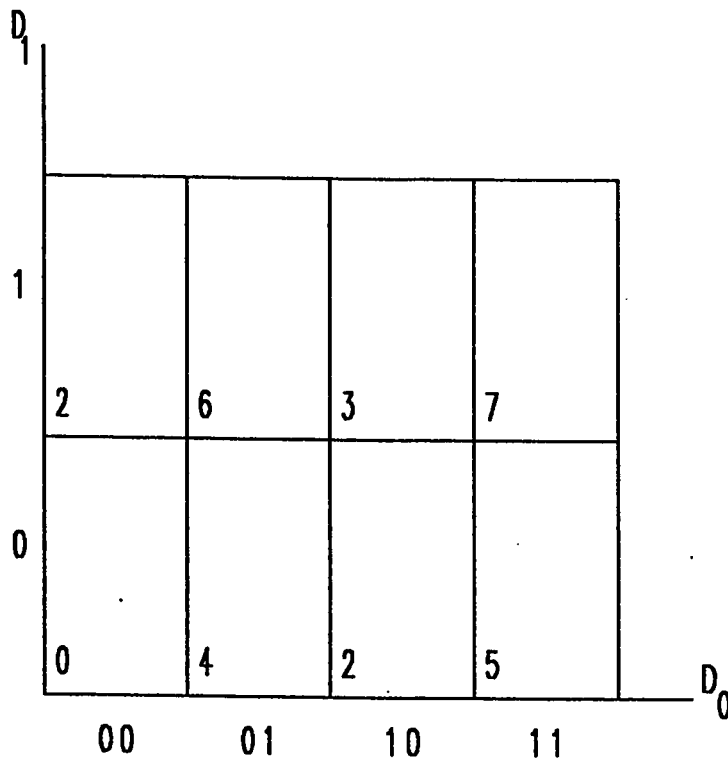


Figure 13: Partitioning the search space into regions.

While this function provides a simple search mechanism in a dynamic environment, it may, if applied as is, result in poor storage utilization. Indeed, this could happen if a split along a given axis caused both the overflowing partition and nonoverflowing partitions to split. As a result of such splitting, the number of empty buckets will increase exponentially with each round of splitting. Another approach is thus necessary while still taking advantage of the mapping function.

Clearly, a better approach, is to split only the overflowing partition and merge several of those emerging empty partitions into the largest possible region of the search space as long as the number of records does not exceed the bucket size and the search cost does not degrade. The rules governing the formation of regions can be stated as follows :

Rule 1 : Let π be the initial partition forming a region. Then the region can be expanded by adding any partition obtainable through a split of π , (i.e. partitions $(\pi + 2^j)$), where $\lfloor \log \pi \rfloor < j < l$ or $0 \leq j < l$ if $\pi = 0$ and as long as the number of records in the region does not exceed the bucket size.

Rule 2 : If a partition π' is merged into the same region as π , then all partitions π'' obtained from π' through splits must also merged into the same region.

Rule 3 : If a partition π' is removed from region π , then : all partitions obtainable from π' through splits must be removed from π .

Definition : Regions π and π' are **buddies** if the regions can be merged together without violating the region formation rules.

Rule 4 : Region π can be merged with its buddy (if it exists), say region π' , to form a single region identified by the smaller of π or π' , if the combined number of records does not exceed the bucket size.

3.3 The Directory

Splitting overflowing partitions only, as illustrated in part(a) Figure 14, is a more natural partitioning of the search space. Several partitions coalesce to form a single region. For instance, partitions 0,2, and 6 have coalesced to form region 0. The major problem now is the fact that region numbers are no longer consecutive. Therefore, a directory is necessary to hold the mapping of these regions into physical storage. Different data structures such as

B-trees can be utilized to hold the index. It is proposed instead to organize the directory in a manner identical to the data file because of the following two reasons : (i) Tries are inadequate structures for associative searching. (ii) Using B-trees to hold the index will violate the clustering property that is required from the structure.

Let $K = (k_0^1, k_1^1, \dots, k_{d-1}^1)$ be the vector obtained from K^0 by truncation or from C^0 as follows :

$$K_i^1 = c_i^0 2^{-1} i \quad \text{for } 0 \leq i < d$$

The directory may now be regarded as one storing records K^1 whose components are directly obtainable from the coordinated of the data file partitions, see part(b) Figure 14. As a result, the directory growth is only a linear function of the number of data buckets. On the other hand, in grid files, each time a bucket is split a $(d-1)$ -dimensional array is added to the directory thereby increasing its size exponentially and making its maintenance problematic.

The methodology used to build the directory suggests that it is possible to systematically build a hierarchy of directories. For example, if the partition numbers in the level #1 directory are not consecutive, a second level might be necessary. It would consist of storing records

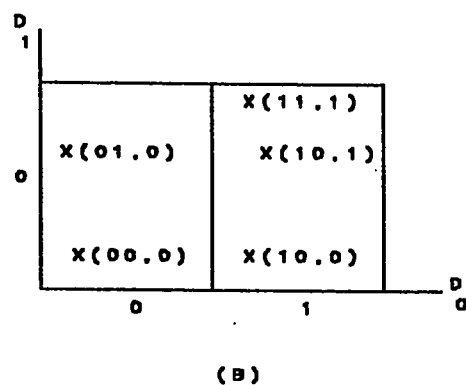
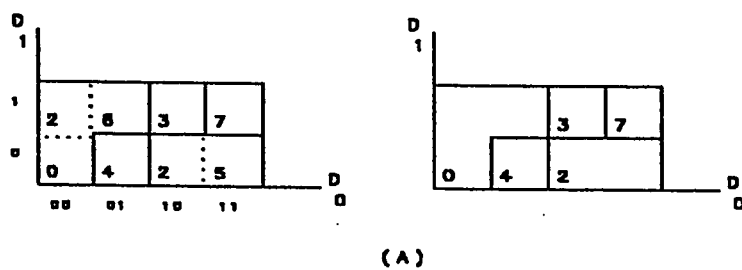


Figure 14: Partitioning the search space into regions. Dotted lines indicate nonexisting boundaries. The directory, indexes of x represents the bucket logical numbers.

$K = (k_0^2, k_1^2, \dots, k_{d-1}^2)$ where each k_i^2 is obtained from k_i^1 by truncation.

3.4 Search Algorithm

To search for a given record K , we have to obtain the partition number using the storage mapping M in which the record may be contained. If the partition does not exist, we are able to determine the embedding partition into which it is merged as follows : Given a partition number π , then all numbers of the possible embedding partitions are given by :

$$\pi, \pi - 2^\alpha, \dots, \pi - \sum_{i=0}^{\alpha} 2^i \pi_i \quad \text{where } \alpha = \lfloor \log \pi \rfloor$$

and $\pi_\alpha, \pi_{\alpha-1} \dots \pi_0$ is the binary representation of π .

Given the coordinates $C = (c_0^0, c_1^0, \dots, c_{d-1}^0)$ of the partition in which the record K may exist, we are able to compute $(c_0^h, c_1^h, \dots, c_{d-1}^h)$ where $0 \leq h < \max h$ which denote the coordinates of the h -level directory partition. Each c_j^h can be constructed from c_j^{h-1} where $0 \leq j < d$ by using the interval partition level along the i th axis of the h -th level directory l_i^h . An exact-match query

$Q=(A_0 = k_0, A_1 = k_1, \dots, A_{d-1} = k_{d-1})$ is a query where k_i is a key belonging to D_i . A partial-match query is an exact-match query except that a range of values may be specified. In the case where an exact-match query is used, searching for a given record located in a page is accomplished by using the algorithm shown in Figure 15.

3.5 Insertion and Deletion

The insertion and deletion procedures are straight-forward when the target data page exists and there is no overflow or underflow. Basically, they consist of searching for the relevant data page where the record must be inserted or deleted. The overflow problem may be solved by splitting the current set of partitions forming the overflowing region. However, this split must be performed in such a way as not to violate the region formation rules mentioned previously. Splitting is performed in such a way that one of the two regions obtained through a split contains some records from the initial set. The insertion algorithm is presented in Figure 16 and Figure 17.

The deletion procedure uses the EXACT-MATCH algorithm to determine the data page where the deletion must be performed. An attempt is then made to combine the region where deletion occurred with other regions. The merging must

be done with the region formation rules in mind. The deletion algorithm is presented in Figure 18, Figure 19, and Figure 20.

/* Given a record $K = (k_0, k_1, \dots, k_{d-1})$
 the algorithm returns α_0 the physical
 address of the page where record K
 may be located */

1. Compute $K_j^0 = [K - \min D_j / \Delta_j]$ for $0 \leq j < d$
 where : $\Delta_j = |\max D_j - \min D_j|$
 and D_j is the domain of attribute A_j
2. Compute $c_j^0 = [k_j^0 \cdot 2^{l_j}]$ for $0 \leq j < d$ where l_j is
 the interval partition level along the j-axis.
3. Compute $c_j^h = p(c_j^{h-1}, l_j^h)$ where p denotes the
 prefix of length l_j^h of binary string
 c_j^{h-1} and l_j^h is the interval partition level
 along the j-th axis of the h-th level directory
 for $0 \leq j < d$ and $0 \leq h < \max h$ where $\max h$ is the number
 of directory levels.
4. Search directories starting at top level

For $j := \max h - 1$ to 1 do

Search bucket α_j for the physical address
 associated with partition $M(K^{j-1}, l^{j-1})$ or the
 one embedding it. Let this address be α_{j-1} .
 If it does not exist, search is unsuccessful,
 exit. We assume that $\alpha_{\max h-1}$ is stored in
 main memory and corresponds to page
 $M(K^{\max h-1}, l^{\max h-1})$.

End;

End EXACT-MATCH.

Figure 15: EXACT-MATCH algorithm.

```
Function DECOMPOSE ( $M_j, M'_j$ ):boolean;
```

```
/* This procedure attempts to decompose region  $M_j$   
   into two regions  $M_j$  and  $M'_j$  */
```

```
If  $M_j$  can be decomposed into two  
(almost even) regions  $M_j$  and  $M'_j$   
using RULE 3 then DECOMPOSE := true;
```

```
End DECOMPOSE;
```

Figure 16: DECOMPOSE function.

```

/* Given a record  $K=(k_0, k_1, \dots, k_{d-1})$ 
the algorithm inserts record K into the file
if the record is not already there */

1. Use algorithm EXACT-MATCH to find the relevant
data page. Keep track of the access path leading
to this page. It consists of the following
triplets :  $(\alpha_h, M_h, S_h)$  for  $0 < h < \max h$  where  $\alpha_h$  is
a physical address of a page at the  $(h-1)$ -th
level directory,  $M_h$  its logical address, and
 $S_h$  the number of elements it contains.

2. Insert K into  $(\alpha_1, M_1, S_1)$ ;
 $S_1 := S_1 + 1$ ;
 $J := 1$ ;
While  $J \leq \max h$ 
begin
  If OVERFLOW in  $\alpha_j$  then
  begin
     $L^{j-1} = L^{j-1} - 1$ ;
    Repeat
       $L^{j-1} = L^{j-1} + 1$ ;
      Apply  $M(K^j, L^{j-1})$  to all records in  $M_j$ 
    Until DECOMPOSE( $M_j, M_j$ );
     $S_j := S_j - S_j$ 
    Modify  $S_j$  of triplet  $(\alpha_j, M_j, S_j)$ ;
    Insert triplet  $(\alpha_j, M_j, S_j)$ 
    into  $(\alpha_{j+1}, M_{j+1}, S_{j+1})$  using  $\alpha_{j+1}$ ;
     $S_{j+1} := S_{j+1} + 1$ ;
  End
   $J := J + 1$ ;
End

```

End INSERT

Figure 17: INSERT algorithm.

```

Function MERGE ( $\alpha_j, \alpha_{j+1}, M_j, S_j, S_{j+1}, Flag$ ) : boolean;
/* This procedure merges the partition  $M_j$ 
   with its buddy  $M'_j$  and then deletes from
    $\alpha_{j+1}$  either partition  $M_j$ 
   or  $M'_j$  depending on which is smaller */

1)  $M'_j := \text{Buddy}(\alpha_{j+1}, M_j)$ 
2) Case  $M_j < M'_j$  :
   Begin
     Merge  $\alpha_j$  and  $\alpha'_j$  into  $\alpha_j$ ;
      $S_j := S_j + S'_j$ ;
     Delete ( $\alpha'_j, M'_j, S'_j$ ) from  $\alpha_{j+1}$ ;
      $S_{j+1} := S_{j+1} - 1$ ;
      $Flag := \text{true}$ ;
     Merge ( $\alpha_j, \alpha_{j+1}, M_j, S_j, S_{j+1}, Flag$ );
   End;

Case  $0 \leq M'_j < M_j$  :
   Begin
     Merge  $\alpha_j$  and  $\alpha'_j$  into  $\alpha'_j$ ;
      $S'_j := S'_j + S_j$ ;
     Delete ( $\alpha_j, M_j, S_j$ ) from  $\alpha_{j+1}$ ;
      $S_{j+1} := S_{j+1} - 1$ ;
      $Flag := \text{true}$ ;
     Merge ( $\alpha'_j, \alpha_{j+1}, M'_j, S'_j, S_{j+1}, Flag$ );
   End;
3) Merge := flag;
4) Return;

End MERGE;

```

Figure 18: Function MERGE algorithm.

Function BUDDY (a_{j+1}, M_j) : integer;

/* This function returns the identifier of the buddy of region M_j . Note that this function is performed within page a_{j+1} .

The "if" checks whether region M_j in a_{j+1} is the buddy of M_j ? $f_j(\pi) = 2^i$ for

$(\lfloor \log \pi \rfloor < i < l^j \text{ or } 0 \leq i < L^j \text{ if } \pi = 0$ */

{ buddy := -1 }

if $[(S'_j + S_j \leq b_j \text{ and } M_j = M_j + f_j(M_j)]$ then

{buddy := M_j ; return;}

if $[(S'_j + S_j \leq b_j \text{ and } M_j = M_j + f_j(M_j)]$ then

{buddy := M_j ; return;}

End BUDDY:

Figure 19: Function BUDDY algorithm.

```

/* Given a record  $K=(k_0, k_1, \dots, k_{d-1})$ 
the algorithm deletes record K into the file
if it exists. The deletion may then cause the
merging of several partitions */

1. Use algorithm EXACT-MATCH to find the relevant
data page. Keep track of the access path leading
to this page. It consists of the following
triplets :  $(\alpha_h, M_h, S_h)$  for  $0 < h < \max h$  where  $\alpha_h$  is
a physical address of a page at the  $(h-1)$ -th
level directory,  $M_h$  its logical address, and
 $S_h$  the number of elements it contains.

2. If K is in  $\alpha$  then delete K else return;

3.  $S_1 := S_1 - 1$ ;

4. For  $j := 1$  to  $\max h - 1$  do
    Begin
    Flag := false;
    If Merge  $(\alpha_j, \alpha_{j+1}, M_j, S_j, S_{j+1}, \text{Flag}) = \text{false}$  then
        exit;
    end;

End DELETE

```

Figure 20: DELETE algorithm.

Chapter IV

A PRELIMINARY APPROACH TO CONCURRENCY IN IBGF

4.1 *Uniform Data Distribution*

In this part, we discuss the design of concurrency control schemes based on locking protocols and a minor modification in the data structure that will regulate the access of concurrent processes to shared data stored in the IBGF. It is assumed that, the data is uniformly distributed over the search space. When the data is uniformly distributed, partition numbers are consecutive in the range $(0, \dots, 2^l - 1)$ where l is the search partition level. Therefore, a directory is not necessary to hold the mapping of these partitions into physical storage, since physical addresses can be computed directly using the storage mapping (M) discussed previously.

Two types of solutions are presented. We will introduce these solutions in sequence to show improvement achieved in the degree of concurrency with no major modification of the data structure.

The file is assumed to reside on a direct access device such as magnetic disk. The storage space is divided into a

contiguous logical address space of fixed size blocks called primary buckets or pages. Each bucket is capable of holding some number b of records. A bucket is the unit of transfer between secondary and primary memory. Collisions (i.e. attempts to insert into a full primary bucket) are handled by creating a chain of overflow buckets associated with that particular bucket address.

There are a couple of rules that can be used to decide when and how to split in this modified interpolation based grid file. One possibility is to maintain an approximately constant storage utilization. In this approach, a split is performed only when the load factor exceeds some threshold. A split operation can be performed either by splitting every data page in the file in order to maintain a one to one mapping between search space partitions and physical data pages, or split one bucket at a time in a linear order. We assume that splitting the search space as a result of an insertion of a new record will not produce empty partition(s) (i.e. newly created partition(s) as a result of such splitting will contain at least one data record, which will preserve our initial assumption that the data is uniformly distributed). Unlike splitting, a merge is done only when the space utilization falls below some threshold. Similarly, a merge operation is performed either on the whole file, or only on two buckets.

In type I solution, splitting is done on every bucket in the file, whereas in type II solution, splitting is done on one bucket chain at a time in a linear order fashion. Similarly, if a merge is called for, in both solutions, it is accomplished by undoing the last split operation performed. That is, the address space is reduced by half in solution I, whereas in solution II it is reduced by one bucket only.

4.2 . Preliminary solution

A solution which allows concurrency among processes executing FIND, INSERT, and DELETE operations on a shared interpolation based grid file ,shown in Figure 21, is presented. The operations SPLIT and MERGE are maintenance processes concerned with restructuring of the interpolation based grid file. We will discuss the parallel behavior of this preliminary solution in terms of its five major operations.

The restructuring phase of a splitting process is triggered when the load factor which is defined as follows :

$$LOADFACTOR = RECCOT / (2^L \times Bucket_size),$$

where RECCOT is the number of records in the file, exceeds some threshold. On the other hand, the restructuring phase for a merging process is triggered when the load factor

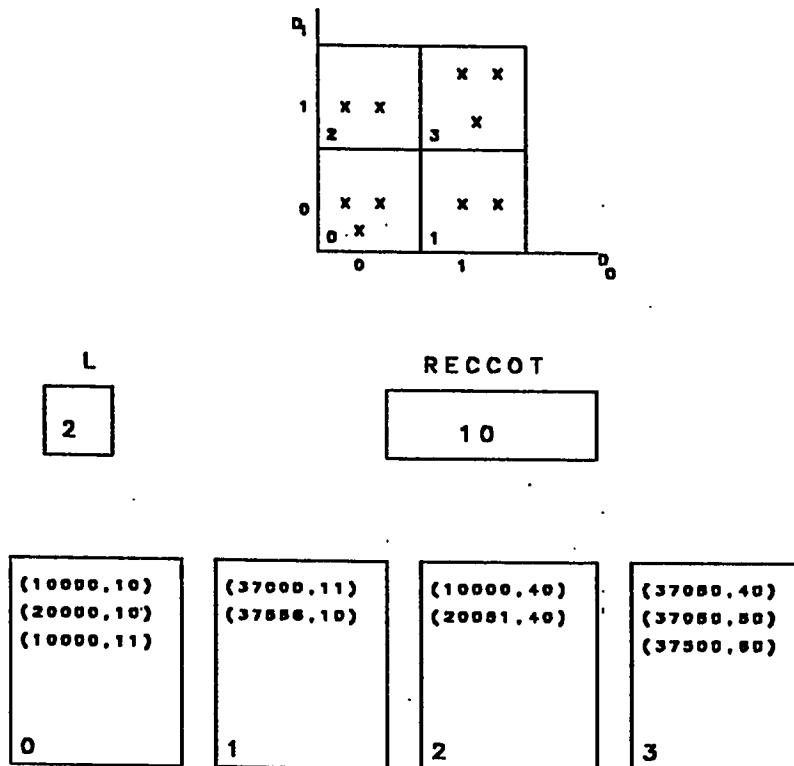


Figure 21: An interpolation based grid file structure.

falls below some threshold. These two restructuring operations can be viewed as separate background processes in spite of the fact that they are called from the procedures insert and delete. That is, when the need for a split or a merge is determined by an INSERT or DELETE process respectively, a separate asynchronous split or merge process is activated accordingly and associated with the calling process.

In this solution, the FIND operation can be performed concurrently with other processes executing FIND, INSERT, and DELETE. Processes executing the INSERT and DELETE operations may operate in parallel only if they are working on different bucket chains. At most, one restructuring operation (split or merge) can be executed at any time.

Locks are used to control access to the shared variables L and RECCOT, and to the bucket chains. The primary bucket and all its overflow buckets are locked as a unit. The compatibility of lock types is given in Figure 22.

The FIND operation places read-locks on L and bucket chains, whereas, the operations INSERT and DELETE place read-lock on L and selective-locks on buckets and RECCOT. The split and merge operations use exclusive-locks on L to prevent new coming processes from accessing the whole file until it is restructured, and on buckets chains to drain off

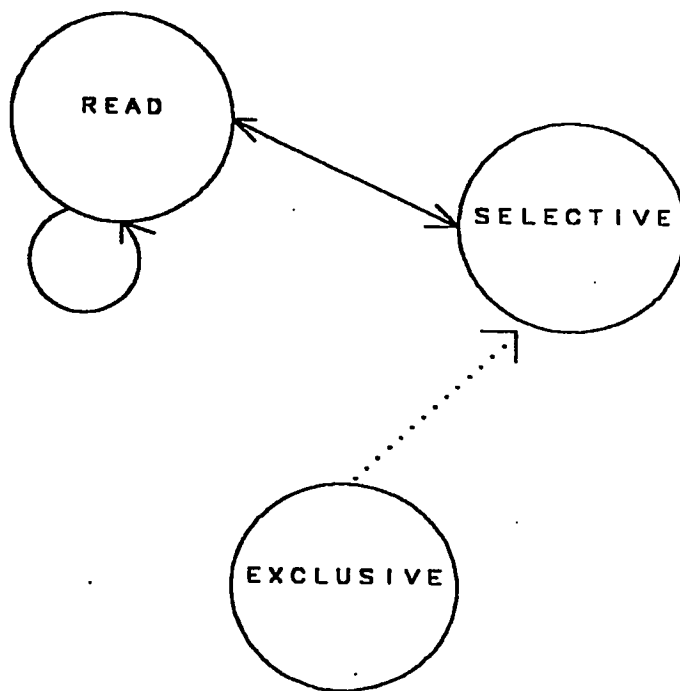


Figure 22: The compatibility of lock types.

already existing processes which are using the old file structure. This means that the whole data file is locked during the restructuring phase.

The manner in which the records are organized within a chain becomes important when the occurrences of insertion and deletion with search is considered. When multiple disk pages make up the chain, reading the chain is not an indivisible step. If records are kept ordered according to some keys, within a chain, one insertion can affect every page, and thus, care must be taken that intermediate states are not visible while the chain is being rewritten. This can be implemented by building a new chain of buckets whose content is copied from the original chain. The designated record is then inserted into or deleted from the appropriate place within the new chain. The process then replaces the primary bucket in the original chain with new contents including a pointer to the new chain. This last disk write makes the new chain available.

If a FIND process reads the chain before the new record is added, it is as if the FIND process occurred before the inserter process. Otherwise, it is as if the inserter occurred first. Similarly, if a FIND process reads the chain before the target record being deleted, it is as if the FIND operation occurred before the DELETE process. Otherwise it is as if the DELETE process occurred first.

During the restructuring phase, the load factor is initially computed to determine if a split or a merge is still necessary. Indeed it may happen that another restructuring operation has just taken place and thereby preempting the need for another such phase. Since restructuring phase is delayed and active processes are allowed to finish, the load factor is recomputed again to prevent restructuring of the file if those processes have caused the load factor to be changed due to successive insertions or deletions.

The pseudocode procedures for FIND, INSERT, DELETE, SPLIT, and MERGE are presented in Figure 23, Figure 24, Figure 25, Figure 26, and Figure 27.


```

Procedure FIND(K);
  Var
    LL      : Integer;
    Bucket_chain : Integer;

  Begin
    Lock(Read,L);
    Get_value(L,LL) /* LL <--- L */
    Bucket_chain := M(K,LL);
    Lock(Read,Bucket_chain);
    Unlock(Read,L);

    /* Read the primary bucket of the chain
       from disk into primary buffer.
       search for K in the primary bucket and
       in subsequent overflow buckets */

    If (found) then
      report success
    else
      report failure;
    Unlock(Read,Bucket_chain);
  end;

```

Figure 23: Algorithm for the FIND operation.

```

Procedure INSERT(K);
Var
  LL          : Integer;
  Lfactor     : Integer;
  Bucket_chain : Integer;
  Reccount    : Integer;

begin
  Lock(Read,L);
  Get_value(L,LL)
  Bucket_chain := M(K,LL);
  Lock(Selective,Bucket_chain);
  Unlock(Read,L);

  /* Build a new chain copied from the
     original chain and insert the new
     record if it does not exist.
     Replace the old chain by the new one */

  If (inserted) then
    begin
      Lock(Selective,RECCOT);
      Get_value(RECCOT,Reccount);
      Reccount := Reccount + 1;
      Lfactor  := Reccount / (2*LL * Bucket_size);
      Put_value(Reccount,RECCOT);
      Unlock(Selective,RECCOT);
    end;
  Unlock(Selective,Bucket_chain);
  If (overflow) then SPLIT;
end;

```

Figure 24: Algorithm for the INSERT operation.

```

Procedure DELETE(K);
Var
  LL          : Integer;
  Lfactor     : Integer;
  Bucket_chain : Integer;
  Reccount    : Integer;

begin
  Lock(Read,L);
  Get_value(L,LL)
  Bucket_chain := M(K,LL);
  Lock(Selective,Bucket_chain);
  Unlock(Read,L);

  /* Build a new chain copied from the
     original chain and delete the record
     if it is there.
     Replace the old chain by the new one */

  If (deleted) then
    begin
      Lock(Selective,RECCOT);
      Get_value(RECCOT,Reccount);
      Reccount := Reccount - 1;
      Lfactor := Reccount / (2**LL * Bucket_size);
      Put_value(Reccount,RECCOT);
      Unlock(Selective,RECCOT);
    end;
  Unlock(Selective,Bucket_chain);
  if (underflow) then MERGE;
end;

```

Figure 25: Algorithm for the DELETE operation.

```

Procedure SPLIT;
Var
  LL      : Integer;
  Lfactor : Integer;
  Reccount : Integer;

begin
  Lock(Exclusive,L);
  Get_value(L,LL)
  Lock(Selective,RECCOT);
  Get_value(RECCOT,Reccount);
  Lfactor := Reccount / (2*LL * Bucket_size);
  Unlock(Selective,RECCOT);

  /* if split is not necessary then Unlock L
    and terminate. otherwise, continue on
    splitting. Drive off active processes
    from the system */

  For I := 1 to 2*LL
    Lock(Exclusive,ith Bucket_chain);

    Lock(Read,RECCOT);
    Get_value(RECCOT,Reccount);
    Lfactor := Reccount / (2*LL * Bucket_size);
    Unlock(Read,RECCOT);

    /* if split is not necessary then Unlock L
      and terminate, otherwise, continue on
      splitting. Start the task by createing
      new bucket chains and appending them to
      the end of the file and move some of the
      records into new locations */

    For I := 1 to 2*LL
      Unlock(Exclusive,ith Bucket_chain);

      LL := LL + 1;
      Put_value(LL,L);
      Unlock(exclusive,L);
    end;
end;

```

Figure 26: Algorithm for the SPLIT operation.

```

Procedure MERGE;
Var
  LL      : Integer;
  Lfactor : Integer;
  Reccount : Integer;

begin
  Lock(Exclusive,L);
  Get_value(L,LL);
  Lock(Selective,RECCOT);
  Get_value(RECCOT,Reccount);
  Lfactor := Reccount / (2*LL * Bucket_size);
  Unlock(Selective,RECCOT);

  /* if merge is not necessary then Unlock L
    and terminate, otherwise, continue on merging.
    Drive off active processes from the system */

  for I := 1 to 2*LL
    Lock(Exclusive,ith Bucket_chain);

  Lock(Read,RECCOT);
  Get_value(RECCOT,Reccount);
  Lfactor := Reccount / (2*LL * Bucket_size);
  Unlock(Read,RECCOT);

  /* if merge is not necessary then Unlock L
    and terminate, otherwise, continue on merging.
    Start the task by undoing the last split and,
    then deallocate unused bucket chains */

  For I := 1 to 2*LL
    Unlock(Exclusive,ith Bucket_chain);

  LL = LL - 1;
  Put_value(LL,L);
  Unlock(exclusive,L);
end;

```

Figure 27: Algorithm for the MERGE operation.

4.3 *An improved solution*

The main drawback of the preliminary solution presented previously is that the whole data space is locked during restructuring. In addition, on the average 50 percent of the records are deleted from some of the buckets and inserted into new ones. As a result, the restructuring operation is not only very costly but imposes stringent limits on the degree of concurrency achievable.

The restructuring operation is unavoidable, but it is desirable to localize it to a minimum number of buckets in the file at a time. Obviously, such a strategy will yield a higher degree of concurrency among processes than the previous approach. The solution described below is an adaptation of the approaches presented in [9,10] for concurrency in extendible and linear hashing files respectively. It is based on locking protocols and a minor modification in the data structure. The modified data structure is shown in Figure 28.

Rather than locking the data space globally, the restructuring operation is now restricted to a single bucket chain at a time. The split operation is applied to each chain in linear order and cyclically. A split is performed on the bucket that is next in line to be split. A variable NEXT is used as a pointer to indicate the chain that should

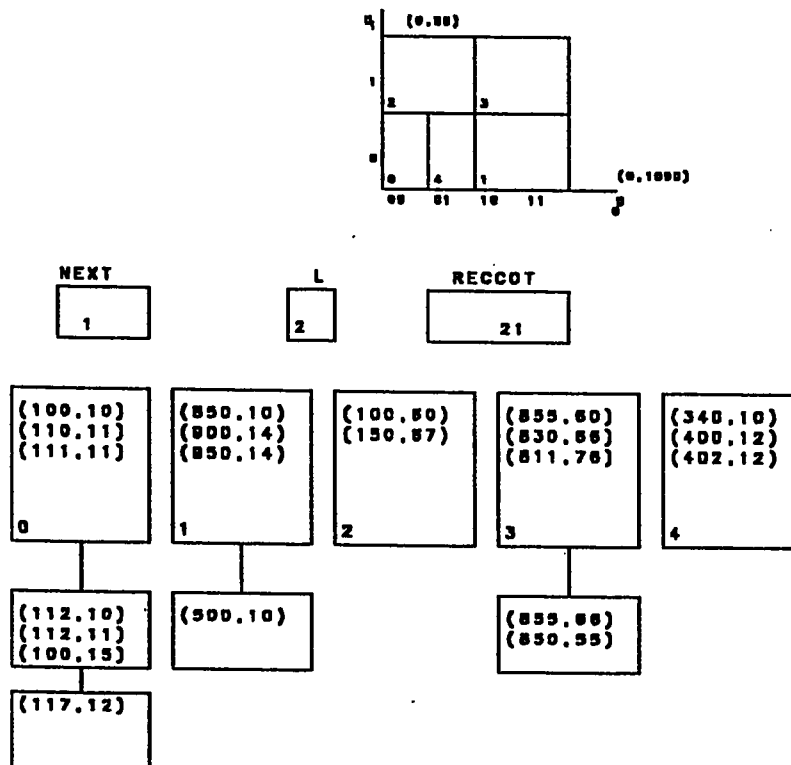


Figure 28: An interpolation based grid file structure.

be split. The resulting modification in the data structure is movement of some records from the original bucket chain being split to a new primary bucket that is appended at the current end of the file. The variables NEXT and L are then updated as follows :

```

If L = 0 then
  Begin
    L := L + 1;
    NEXT := 0;
  End
Else
  begin
    Next := (NEXT + 1) mod 2**L;
    If NEXT = 0 then L := L + 1;
  End;

```

Unlike a split operation, a merge operation is performed on the bucket chain which is at the current end of the file and the bucket chain from which that particular bucket chain was generated due to a split. That is, undo the last split operation performed on the file. The variables NEXT and L are then updated as follows :

```

If NEXT = 0 and L not = 0 then
  Begin
    NEXT := (2**L - 1) - (2**(L - 1));
    L := L - 1;
  End
Else
  NEXT := (NEXT - 1) mod 2**L;

```

In this solution, the FIND operation can be performed concurrently with other processes executing the procedures FIND, INSERT, DELETE, and SPLIT. Processes executing the

INSERT and DELETE procedures may operate in parallel if they are accessing different bucket chains. A split may be performed in parallel with INSERT and DELETE operations that are not accessing the particular chain being split. The interaction between a MERGE and FIND, INSERT, or DELETE processes is more complicated. Those processes may not access the two buckets being merged and may not read the value of NEXT and L while the merging process is using them. At most, one restructuring operation can be executing at any time.

The FIND algorithm calls for the use of lock-coupled read locks on L, NEXT, and the bucket chains. The procedures INSERT and DELETE read-lock L, NEXT and selective-lock bucket chains. The SPLIT operation uses selective locks. The MERGE operation places exclusive locks on L, NEXT and both of the partner bucket chains being merged. After the values of the variables L and NEXT have been changed to reflect the smaller data space that will result from the merge, the locks on these variables are converted to selective locks, and processes entering their searching phase may then concurrently access the variables L and NEXT. The compatibility graph of lock types is given in Figure 22 on page 70.

Clearly, concurrency is enhanced by allowing a searching process to operate in parallel with a split operation, but there must be some means for it to reorient itself when the wrong chain is reached because of an out-of-date L value. The current value of L always reflects a smaller search space for new coming processes. Assume that a FIND process is operating in parallel on the file structure shown in Figure 28 on page 79, with a SPLIT operation. If the FIND process decided, using the current value of L, that the record being searched for is located in bucket number 1 which is also subject to a split operation as indicated by the variable NEXT, then that particular process may not find the designated record if the SPLIT process was able to split the bucket into two buckets before the FIND process can gain access to it. In this scheme, each chain includes an additional field LOCAL that specifies the most recent split affecting this bucket. Storing LOCAL in the primary bucket ensures that the searching process can decide if it has the right chain without requiring the accuracy of the shared variable L. The modified data structure is shown in Figure 29.

A process executing in its searching phase behaves as follows : the value of L is read and the value seen determines which bucket should be accessed initially. Let

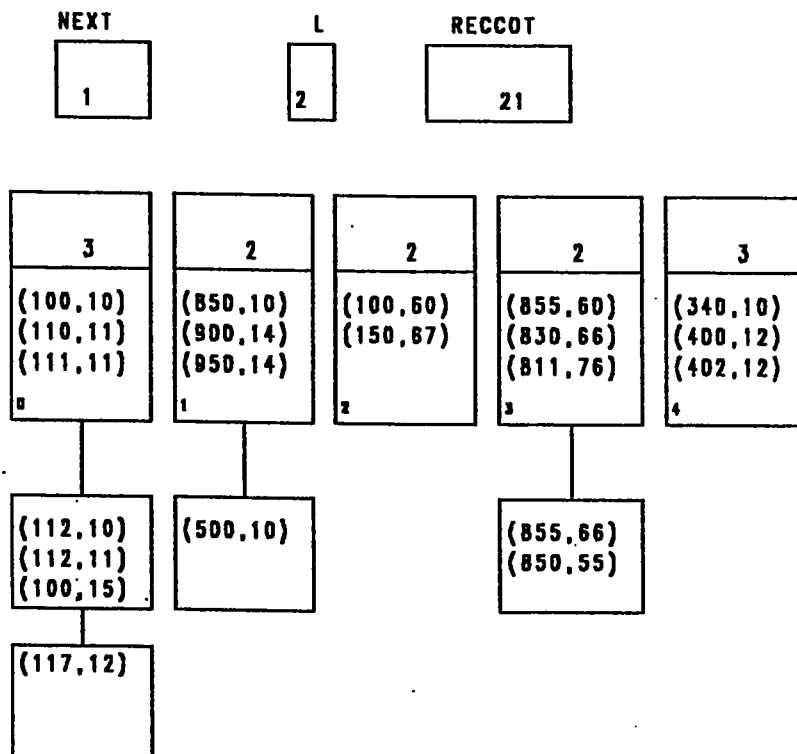


Figure 29: A modified IBGF structure with LOCAL field.

the private variable PRL record the value of L at the time it was read. Upon gaining access to a bucket, the process checks whether PRL matches that bucket's LOCAL, and if not, it increments its PRL value and recalculates the address $M(K, PRL)$ until a match is found. The calculated address at each iteration will always be less than or equal to the address of the eventual destination. This is a direct consequence of Rule 01 discussed previously (i.e. records moved from a bucket during a split go to the higher address chain $\pi + 2^l$). Thus the bucket chain in which the desired record belongs should be reachable using this strategy as long as each address calculated is within the valid address space at the time of access. The two new chains resulting from a split appear atomically to other processes because of the order in which they are written to disk. Specifically, the chain at the new bucket address is written before the new version replaces the chain at the target bucket address. At this point, no information contained in the IBGF points to the existence of this new bucket. Once the primary bucket at the head of the chain at the target address has been written, its LOCAL value indicates that the bucket has split and a new bucket has been incorporated into the file. After the reorganized chains are safely in place, the value of L and NEXT are changed to allow direct calculation of

the address of the new chain. A process responsible for merging two buckets, holds exclusive-locks on both partners of the merge while it makes its changes. The read-lock held by the searching process prevents a merge from decreasing the size of the address space during the initial bucket access. The pseudocode procedures for FIND, INSERT, DELETE, SPLIT, and MERGE are presented in Figure 30, Figure 31, Figure 32, Figure 33, Figure 34, and Figure 35.

```

Procedure LOCATE_BUCKET(K,Locktype,Bucket_chain);
  Var
    PRL      : Integer;
    Previous : Integer;

  Begin
    Lock(Read,L);
    Get_value(L,PRL); /* PRL <--- L */
    Bucket_chain := M(K,PRL);
    Lock(Locktype,Bucket_chain);
    Unlock(Read,L);

    /* read the primary bucket of the chain from
       disk into primary buffer */

    While LOCALL not = PRL do /* wrong bucket */
      begin
        PRL := PRL + 1;
        Previous := Bucket_chain;
        Bucket_chain := M(K,PRL);
        Lock(Locktype,Bucket_chain);
        Unlock(Locktype,Previous);

        /* read the primary bucket of the chain from
           disk into primary buffer */

      end;
    end;
  end;

```

Figure 30: Algorithm for search phase.

```
Procedure FIND(K);  
  var  
    Bucket_chain  : Integer;  
begin  
  LOCATE_BUCKET(K,Read,Bucket_chain);  
  
  /* read the primary bucket of the chain from  
    disk into primary buffer, search buffer  
    and subsequent buckets of chain for K */  
  
    if (found) then report success  
    else           report failure;  
  Unlock(Read,Bucket_chain);  
end;
```

Figure 31: Algorithm for the FIND operation.

```

Procedure INSERT(K);
  Var
    LL           : Integer;
    Pntr         : Integer;
    Bucket_chain : Integer;
    Lfactor      : Integer;
    Reccount     : Integer;

  Begin
    LOCATE_BUCKET(K, Selective, Bucket_chain);

    /* Build a new chain and insert the new record
       if it does not exist */

    If (inserted) then
      begin
        Lock(Read, L);
        Get_value(L, LL);
        Lock(Read, NEXT);
        Get_value(NEXT, Pntr);
        Lock(Selective, RECCOT);
        Get_value(RECCOT, Reccount);
        Reccount := Reccount + 1;
        Lfactor  := Reccount / ((2 * LL + Pntr) * Bucket_size);
        Put_value(Reccount, RECCOT);
        Unlock(Read, L);
        Unlock(Read, NEXT);
        Unlock(Selective, RECCOT);
      end;

    /* Replace the old chain by the new one */

    Unlock(Selective, Bucket_chain);
    if (Overflow) then SPLIT;
  end;

```

Figure 32: Algorithm for the INSERT operation.


```

Procedure DELETE(K);
  Var
    LL           : Integer;
    Pntr         : Integer;
    Bucket_chain : Integer;
    Lfactor      : Integer;
    Reccount     : Integer;

  Begin
    LOCATE_BUCKET(K, Selective, Bucket_chain);

    /* Build a new chain and delete the record
       if it is there */

    If (deleted) then
      begin
        Lock(Read, L);
        Get_value(L, LL);
        Lock(Read, NEXT);
        Get_value(NEXT, Pntr);
        Lock(Selective, RECCOT);
        Get_value(RECCOT, Reccount);
        Reccount := Reccount - 1;
        Lfactor  := Reccount / ((2 * LL + Pntr) * Bucket_size);
        Put_value(Reccount, RECCOT);
        Unlock(Read, L);
        Unlock(Read, NEXT);
        Unlock(Selective, RECCOT);
      end;

    /* Replace the old chain by the new one */

    Unlock(Selective, Bucket_chain);
    If (underflow) then MERGE;
  end;

```

Figure 33: Algorithm for the DELETE operation.

```

Procedure SPLIT;
Var
  Target_bucket_chain  : Integer;
  Lfactor, LL, Pntr    : Integer;
  Reccount             : Integer;
Begin
  Lock(Selective,L); Get_value(L,LL);
  Lock(Selective,NEXT); Get_value(NEXT,Pntr);
  Lock(Selective,RECCOT);
  Get_value(RECCOT,Reccount);
  Lfactor := Reccount/((2*L+Pntr)*bucket_size);
  If (split is not necessary) then
    begin
      Unlock(Selective,L); Unlock(Selective,NEXT);
      Unlock(Selective,RECCOT); Terminate;
    end
  else
    begin
      Unlock(Selective,RECCOT);
      Target_bucket_chain := Pntr;
      /* Allocate newchain and append it at the
         current end of the file. Move some of
         records in the target bucket chain
         pointed by NEXT to newchain. Write the
         primary bucket of the target chain with
         the new value of LOCALL */
      if LL = 0 then
        begin
          LL := LL + 1; Pntr := 0;
        end
      else
        begin
          Pntr := (Pntr+1) mod 2*LL;
          if Pntr = 0 then LL := LL + 1;
        end;
      Put_value(Pntr,NEXT); /* NEXT <--- Pntr */
      Put_value(LL,L); Unlock(Selective,L);
      Unlock(Selective,NEXT);
      Unlock(Selective,Target_bucket_chain);
    end;
  end;
end;

```

Figure 34: Algorithm for the SPLIT operation.

```

Procedure MERGE;
Var
  Fpartner, Spartner, Lfactor : Integer;
  LL, Reccount, Pntr          : Integer;
Begin
  Lock(Exclusive,L); Get_value(L,LL);
  Lock(Exclusive,NEXT); Get_value(NEXT,Pntr);
  Lock(Selective,RECCOT); Get_value(RECCOT,Reccount);
  Lfactor := Reccount/((2*LL+Pntr)*bucket_size);
  If (Merge is not necessary) then
    begin
      Unlock(Exclusive,L); Unlock(Exclusive,NEXT);
      Unlock(Selective,RECCOT); Terminate; end
  else
    begin
      Unlock(Selective,RECCOT);
      if Pntr <= 0 then
        begin
          Fpartner := Pntr - 1;
          Spartner := (2*(LL+1)-1) - Pntr; end
        else
          begin
            Fpartner:=(2*LL-1)-(2*(LL-1));
            Spartner:=2*LL; end;
      If Pntr = 0 and LL <= 0 then
        begin
          Pntr:=(2*LL-1)-(2*(LL-1)); LL:= LL-1;
        end
      else
        Pntr := (Pntr-1) mod 2*LL;
      Put_value(Pntr,NEXT); Put_value(LL,L);
      Convert_lock(Exclusive,Selective,L);
      Convert_lock(Exclusive,Selective,NEXT);
      Lock(Exclusive,Fpartner);
      Lock(Exclusive,Spartner);
      /* Merge partners and deallocate Spartner.
         Decrement LOCALL for Fpartner. */
      Unlock(Selective,L); Unlock(Selective,NEXT);
      Unlock(Exclusive,Fpartner);
      Unlock(Exclusive,Spartner);
    end; end;

```

Figure 35: Algorithm for the MERGE operation.

An example of parallel computation involving two requests is given in Figure 36. The vertical columns of the text give the steps executed by each of the two processes. The horizontal alignment of these steps indicates when concurrent execution is assumed. Gaps show when delays occur. The IBGF shown in Figure 29 on page 83, is assumed to be the initial state. Figure 37 and Figure 38, show the updated data structure at designated points in the computation. This example illustrates some of the important interactions between processes.

At the beginning of the computation, the process attempting to insert the record (850,11), succeeded to locate the bucket chain where that particular record will be inserted in. Process I then manages to acquire its selective lock on bucket chain 1, and to insert the record. Such an update, triggered a restructuring phase. Meanwhile, process F starts executing the FIND operation in parallel with the split and involving the same bucket. This demonstrates the recomputation strategy that is fundamental in the locate phase when the wrong chain is reached because of incorrect value of L. The first address calculated using the value of L (2) read by the process F is for bucket chain 1. Process F reads the primary bucket after the newly split version has been written by process I. Seeing a LOCALL value of 3,

Process I:INSERT(850,11)

Process F:FIND(850,10)

```

Locate and lock chain
Lock(read,L)
read L : PRL = 2
Bucket-chain = 1
Lock(Selective,1)
Unlock(read,L)
Read chain(1)
LOCALL = PRL, quit
Insert the record into chain
(Figure 37)
Lock(Read,L); Lock(Read,NEXT)
Lock(selective,RECCOT)
Compute load factor
Unlock(selective,RECCOT)
Unlock(Selective,1)
Overflow : SPLIT
Lock(Selective,L)
Lock(Selective,NEXT)
Lock(Selective,RECCOT)
Compute load factor
Unlock(Selective,RECCOT)
Target_bucket_chain = 1
Lock(Selective,1)
Construct new chain and split
records
Write new chain
Rewrite bucket 1
Increment NEXT
(Figure 38)
Unlock(selective,L)
Unlock(selective,NEXT)
Unlock(selective,1)

```

```

Locate and lock chain
Lock(Read,L)
Read L : PRL = 2
Bucket-chain = 1
Lock(read,1)
Unlock(Read,L)

Read chain(1)
LOCALL -= PRL
Recompute with PRL = 3
Bucket-chain = 5
Lock(Read,5)
Unlock(Read,1)
Read chain(5)
LOCALL = PRL, quit
Search for (850,10)
Found
Unlock(read,5)

```

Figure 36: Example of parallel computation I.

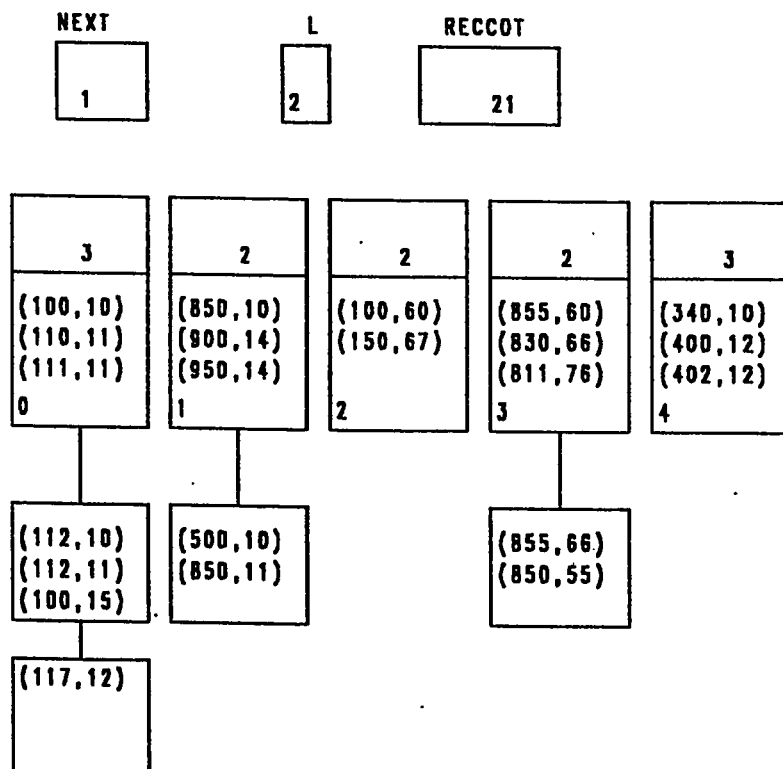


Figure 37: Progressive states of the IBCF : stage I.

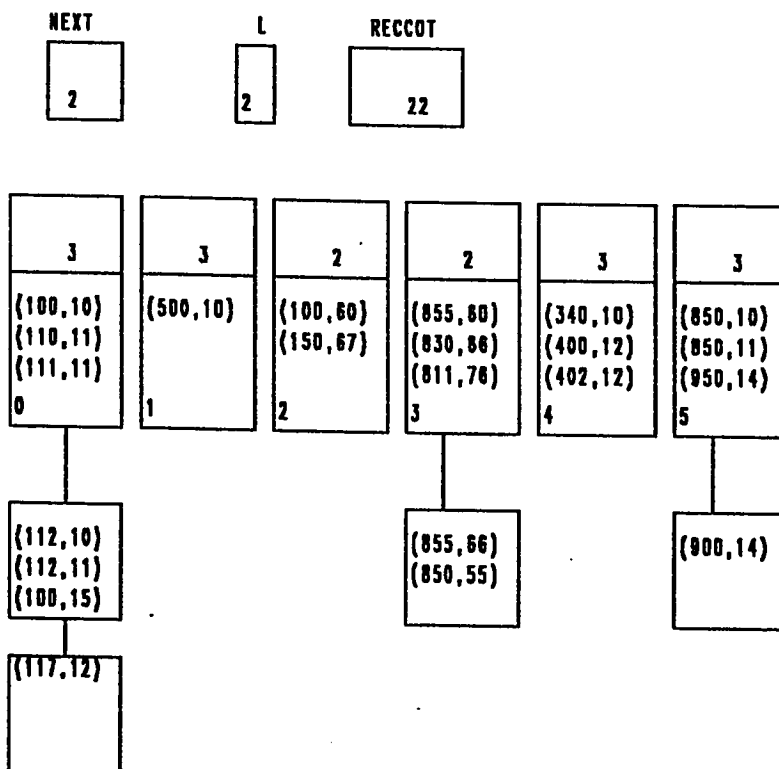


Figure 38: Progressive states of the IBGF : stage II.

process F recomputes the address 5. It places a read lock on bucket chain 5 and releases the lock on bucket chain 1 (lock-coupling). Eventually, the target record is found in bucket chain 5.

Another example of parallel computation involving two requests is given in Figure 39. The IBGF shown in Figure 40, is assumed to be the initial state. Figure 41, shows the updated data structure at designated points in the computation. This example illustrates the interaction between a DELETE and consequently a MERGE operation from one side and a FIND operation from the other. At the beginning of the computation, the process attempting to delete the record (20051,40), succeeded to locate and the bucket chain where that particular record will be deleted from. Process D then manages to acquire its selective lock on bucket chain, and to delete the record. Such an update, triggered a restructuring phase. Meanwhile, process F starts executing the FIND operation in parallel with the merge operation. Process D places an exclusive lock on L and NEXT first to decrement their values while process F is blocked. In the final part of the computation, process D has finished updating the values of L and NEXT, allowing process F to get its lock on those variables which will enable process F to operate in parallel with the actual merge operation since it is working on a different bucket chain.

Process D:DELETE(20051,40)	Process F:FIND(10000,10)

Locate an lock chain	
Lock(Read,L)	
Read L : PRL = 2	
Bucket chain = 2	
Lock(selective,2)	
Unlock(Read,L)	
Read chain(2)	
LOCALL = PRL, quit	
Delete the record	
(Part(a) Figure 41)	
Lock(selective,RECCOT)	
Compute load factor	
Unlock(Selective,RECCOT)	
Unlock(Selective,2)	
Underflow : MERGE	
Lock(Exclusive,L)	Locate and lock chain
Lock(Exclusive,NEXT)	Lock(Read,L) --- wait
Lock(Selective,RECCOT)	
Compute load factor	
Unlock(Selective,RECCOT)	
Decrement L and NEXT	
Downgrade lock on L and NEXT	Succeed :
(Part(b) Figure 41)	Read L : PRL = 2
Fpartner = 3	Bucket chain = 0
Spartner = 1	Lock(read,0)
Lock(Exclusive,Fpartner)	Unlock(read,L)
Lock(Exclusive,Spartner)	Read chain(0)
Merge partners	LOCALL = PRL , quit
Unlock(Exclusive,1)	Search for (10000,10)
Unlock(Exclusive,3)	Found
	Unlock(Read,0)

Figure 39: Example of parallel computation II.

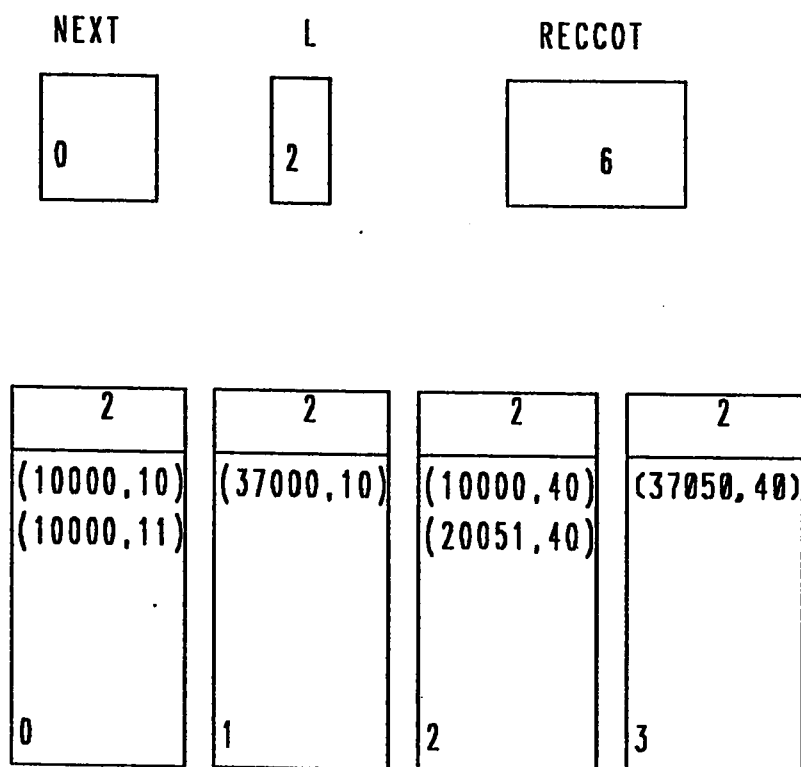


Figure 40: An initial state of an IBGF.

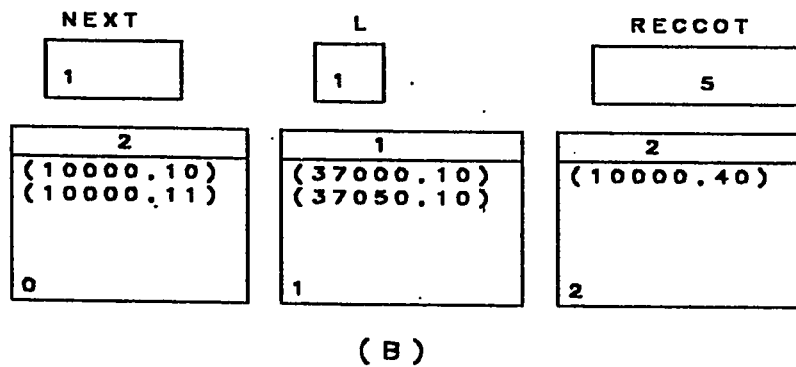
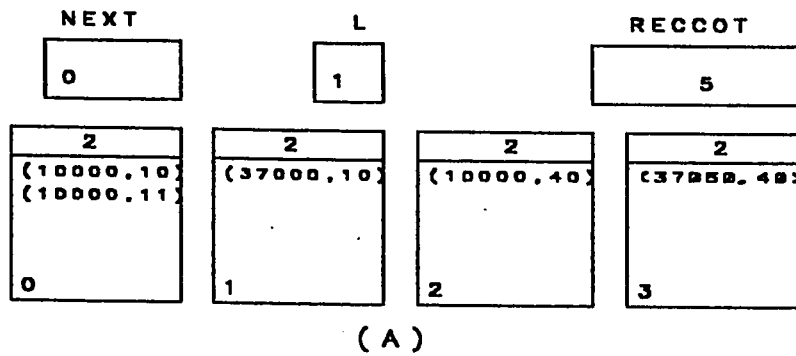


Figure 41: Progressive states of the IBGF.

4.4 Discussion

During the searching phase of FIND, INSERT, and DELETE, locks are placed according to a well-defined ordering. Merges and Splits also respect the ordering in requesting their locks. Thus, deadlock can not occur. Searching for the record as part of a deletion, or for the place to insert as part of insertion, requires that the effect of previous updates (even those still active) be seen. Selective-locks are placed on the chains during the searching phase to serialize writers of the same individual buckets so that only up-to-date information is seen. This guarantees that there is no interference between concurrent executions of INSERT and DELETE.

Merges and Splits are completely serialized with respect to one another by incompatible locks on L and NEXT. All affected bucket chains are also locked by a restructuring process for the duration of the step. The SPLIT procedure allows, because of its selective-locks, only processes executing the FIND routine to concurrently access the chain being split. The exclusive-locks held during a merging process do not permit any concurrent use of the partner buckets of the merge. Processes executing FIND, INSERT, and DELETE are allowed to operate in parallel with a MERGE if they are working on a different bucket chain than the ones

being merged. When the variables L and NEXT are updated to reflect smaller address space, the locks on them are converted to Selective-locks, and processes entering their searching phase may then access L and NEXT values.

Chapter V

A CONCURRENCY SCHEME FOR THE IBGF

5.1 *Nonuniform data distribution*

In this part, we will investigate the problem of supporting an arbitrary number of processes to operate concurrently on the IBGF with the directory option being used as shown in Figure 42. Two solutions are presented. The first solution is based on the approaches proposed to handle concurrent operations in B-trees [17,18,26,28,1]. It was suggested there that they may also be applicable in other tree structures. We will investigate this solution when it is applied on the IBGF. The second solution is a new one which exploits some of the structural properties of the IBGF and achieves the following results.:

1. A higher degree of concurrency than the first solution on the average case is achieved and the same degree of concurrency in the worst case.
2. No major modifications of the data structure are involved.

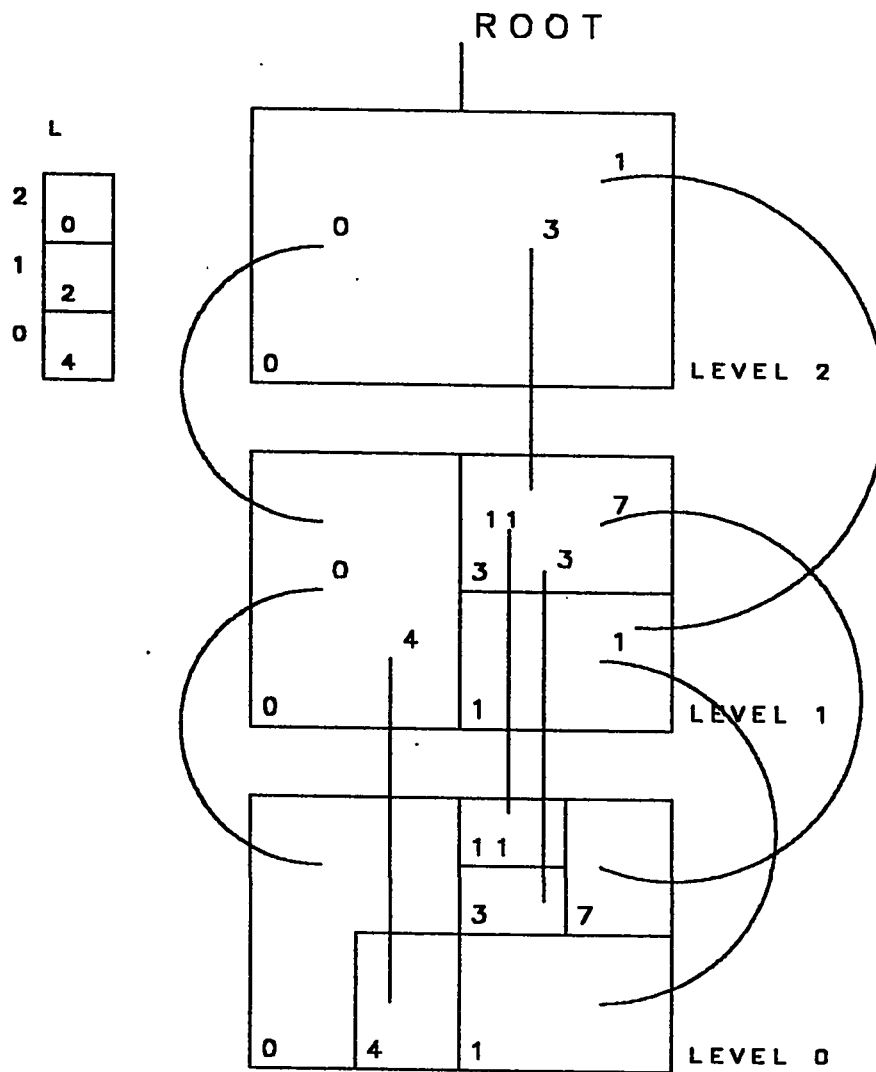


Figure 42: An IBGF with two directory levels.

5.2 Preliminaries

In the IBGF a path from the top level directory to a data bucket determined by a process on its passage is denoted by the triplets (α_h, M_h, S_h) for $0 < h < \max h$, where α_h is a physical address of a page at the $(h-1)$ -level directory, M_h its logical address, and S_h the number of elements it contains.

Definition : A bucket B at the h -th level directory of an IBGF is *insertion-safe*, if it is unsaturated, (it has records less than the bucket size), (i.e. $(S_h < b_h$, where b_h is the bucket capacity at the h -th level directory), and it is *deletion-safe*, if it is not minimal (it has more than $b_h/2$) (i.e. $S_h \geq b_h/2$).

5.3 A B-tree solution

5.3.1 Locking scheme

In this solution, the SEARCH operation can be performed concurrently with other processes executing the same operation. In addition, SEARCH processes may be working concurrently with other processes executing the INSERT and DELETE operations when they are at the phase of establishing their access paths. Processes executing the INSERT and

DELETE operations may operate in parallel if they are working on different access paths.

The scope for an updater must remain invariant during restructuring. This implies that the scope of an updater must be locked before any necessary restructuring can begin. The locking is often done in such a way that no other updater can be in the scope once it is locked, not even waiting in the queues associated with the buckets other than the deepest safe bucket. On the other hand, searchers may be present in the scope of an updater during its searching phase. However, before the restructuring phase can begin the updater should drive away all searchers from its scope.

A process waiting at the deepest safe bucket for an updater to terminate may reach an incorrect data bucket if an attempt was made prior to the actual access to compute its access path only once. A searcher or an updater in its searching phase must continuously examine the value of L in order to determine the next move from the h -th to $(h-1)$ level directory correctly. It turns out that such processes are required only to examine the value of L_0 (the interval partition level at the level 0 directory) to determine the next move in their access paths no matter what directory level they are at. The following Lemma provides the number of the region which should be accessed next in the access path.

Lemma : Given a partition number π at the 0-th level directory, then the region number containing π at the h-th level directory is identified by the greatest logical partition number inferior or equal to π without violating Rule 02.

During restructuring of the h-th level directory, an updater is allowed to hold an exclusive lock on the partition level at the h-th level directory only just for the duration needed to restructure that particular directory level. This means that an updater should lock L^h just before restructuring and unlock it immediately before the actual restructuring can begin at the h+1 directory level. Clearly, such a strategy will enhance the degree of concurrency achievable than if L was locked for the whole duration of the restructuring phase.

Two important schemes for locking the buckets are enforced :

1. Bucket locking by searchers : A searcher first locks the root and on its passage to the target bucket at the data level (i.e. level 0 directory) it unlocks a bucket at the h-th level directory only after it has locked the bucket at the (h-1) level directory.

2. Scope locking by updaters : An updater first locks the root and on its passage to the frontier the appropriate buckets are locked and examined. When a safe bucket is found all its ancestors are unlocked.

The compatibility of lock types to be used by different processes is given in part (b) Figure 8 on page 28. A search process adopts the usage of read-locks with lock coupling technique. An updater uses write-locks to lock its scope from other updaters during the searching phase, but searchers may be present because read-locks and write-locks are fully compatible. However, before restructuring can begin the updater converts all write-locks into exclusive-locks starting at the deepest safe bucket of the scope. Since exclusive-locks are incompatible with read-locks this ensures that all searchers in the scope are driven off.

5.3.2 Concurrent behavior

To search for a record K, the SEARCH process begins at the root and proceeds by computing the next move in the path down to the appropriate data bucket. When the target bucket is reached, the process simply scans the bucket and reports success if the record exists otherwise it reports failure.

To insert a record K , the INSERT process begins at the root and proceeds by locking its access path, from other updaters, down to the appropriate data bucket where K should be contained. When the deepest safe bucket in the path is determined, the process unlocks all ancestors of the safe bucket which will allow another updater if any to proceed if it is accessing a different path. The insertion of the record K into the target data bucket may necessitate splitting that particular bucket. The original region M is decomposed into two regions, M and M' , in such a way that some of the records from the initial set will map to the new one and the region formation rules are not violated. The new region identifier along with its corresponding physical address and capacity is then inserted into the region containing M in the first level directory. In turn, this may cause an overflow. Clearly, this will have a ripple effect on all levels of the directory, all the way to the deepest safe bucket.

To delete a record K , the DELETE process performs operations similar to that for INSERT during its searching phase. The deletion of the record K may require the process to combine the region where the deletion occurred with other regions. The merging must be done with the region formation rules in mind. The identifier of the region being merged is then deleted from the first level directory. In turn, this

may cause an underflow that will propagate all the way up to the deepest safe bucket. It should be noticed that merging is not performed on the whole search space; rather, it is constrained to the set of regions determined by the access path.

The pseudocode for the procedures SEARCH, INSERT, and DELETE are presented in Figure 43, Figure 44, Figure 45, Figure 46, and Figure 47, respectively. The reader can refer to chapter III for the definition of the functions MERGE and DECOMPOSE.

Procedure SEARCH(K);

- 1) Lock(Read, $\alpha(\text{Root})$);
- 2) Current := Root;
- 3) COMPUTE_NEXT_MOVE;
- 4) For j := maxh-1 to 1 do
begin
 Lock(Read, $\alpha(\text{NEXT})$);
 Unlock(Read, $\alpha(\text{Current})$);
 Current := NEXT;
 COMPUTE_NEXT_MOVE;
End;
- 5) Search for the record in the current bucket and
 report success if found, otherwise report failure.

End SEARCH;

Figure 44: SEARCH algorithm.

```

Procedure UPDATER_SEARCH_PHASE;

1) Lock(Write,  $\alpha(\text{root})$ );

2) Current := Root;

3) COMPUTE_NEXT_MOVE;

4) For J := maxh-1 to 1 do
    Begin

        Lock(Write,  $\alpha(\text{NEXT})$ );

        /* Locate the deepest safe bucket */
        If (NEXT is safe) then
            Unlock(Write, Current);
        Endif

        Current := NEXT;

        COMPUTE_NEXT_MOVE;

    End;

5) Convert write locks placed on the scope into
   exclusive locks.

End UPDATER_SEARCH_PHASE;

```

Figure 45: Locate and lock an updater scope.

```

Procedure INSERT(K);

1) UPDATER_SEARCH_PHASE;

2) Insert K into  $(a_1, M_1, S_1)$ ;
    $S_1 := S_1 + 1;$ 
    $J := 1;$ 
   While  $J \leq (\text{level deepest safe bucket})$  do
     begin
       Lock(Exclusive,  $L^{J-1}$ );
       If OVERFLOW in  $a_j$  then
         begin
            $L^{J-1} := L^{J-1} - 1;$ 
           Repeat
              $L^{J-1} := L^{J-1} + 1;$ 
             Apply  $M(K^j, L^{J-1})$  to all records in  $M_j$ ;
           Until DECOMPOSE( $M_j, M_j$ );
            $S_j := S_j - S_j$ 
           Modify  $S_j$  of triplet  $(a_j, M_j, S_j)$ ;
           Insert triplet  $(a_j, M_j, S_j)$ 
             into  $(a_{J+1}, M_{J+1}, S_{J+1})$  using  $a_{J+1}$ ;
            $S_{J+1} := S_{J+1} + 1;$ 
         End
       Unlock(Exclusive,  $L^{J-1}$ );
        $J := J + 1;$ 
     End

3) Unlock(Exclusive, (updater scope));

```

Figure 46: INSERT algorithm.

Procedure DELETE(K);

1) UPDATER_SEARCH_PHASE;

2) If K is in α_1 then delete K else return;

3) $S_1 := S_1 - 1$;

4) For J := 1 to (level of deepest safe bucket) do

 Begin

 Lock(Exclusive, L^{J-1});

 If Merge ($\alpha_j, \alpha_{j+1}, M_j, S_j, S_{j+1}, Flag$) = false then
 exit;

 Unlock(Exclusive, L^{J-1});

 End;

End DELETE;

Figure 47: DELETE algorithm.

5.3 *An optimistic concurrency control scheme*

A concurrency control mechanism performance is measured by the set of operations "fixpoint" [15,13] it can authorize for execution in parallel without any delay. In general, the more information available to the mechanism the higher is the degree of parallelism (or performance) achieved. The information used in this context is typically either syntactic information about the process (i.e. the names of data base entities accessed and updated at each step), or semantic information about the meaning of the data and the operations performed, or the integrity constraints that the database must satisfy. Our goal is to devise a concurrency control mechanism which enhances the degree of concurrency without introducing any additional overhead in terms of the amount of information needed.

In a tree structure such as B-trees processes are forced to follow very strict locking protocols in order to safeguard their access paths. This is basically due to insufficient a priori information about access paths since they are only constructed during the actual access of the structure. As a result, the root of the tree structure becomes a bottleneck. For instance, an updater with its deepest safe bucket being the root itself will block, for the whole duration of the searching and restructuring phase,

other updaters even if their safe buckets are located beyond the root in totally divergent paths as illustrated in the discussion below.

Let P_i and P_j be updaters and specifically inserters operating on the IBGF structure shown in Figure 48. Assume that 11 is the target bucket for P_i and 4 is the target bucket for P_j . Furthermore, assume the deepest safe buckets for P_i and P_j are the root and 4 respectively. Also assume, for a moment, that the bucket numbering is meaningless. If P_i occurred first then the root and all other elements in the access path of P_i will be locked. Process P_j will thus be blocked at the root for the whole duration of the searching and the restructuring phase of P_i . But as one can easily observe both processes can operate in parallel since the modifications to the structure resulting from these operations have no effect on each other. A scheduler can allow this to happen only if such independence can be detected prior to access. Clearly, this is not possible in the case of B-trees and other tree structures.

On the other hand, in the IBGF, given a record K, we can determine the access path leading to that particular record before accessing the structure. As a result, we shall show

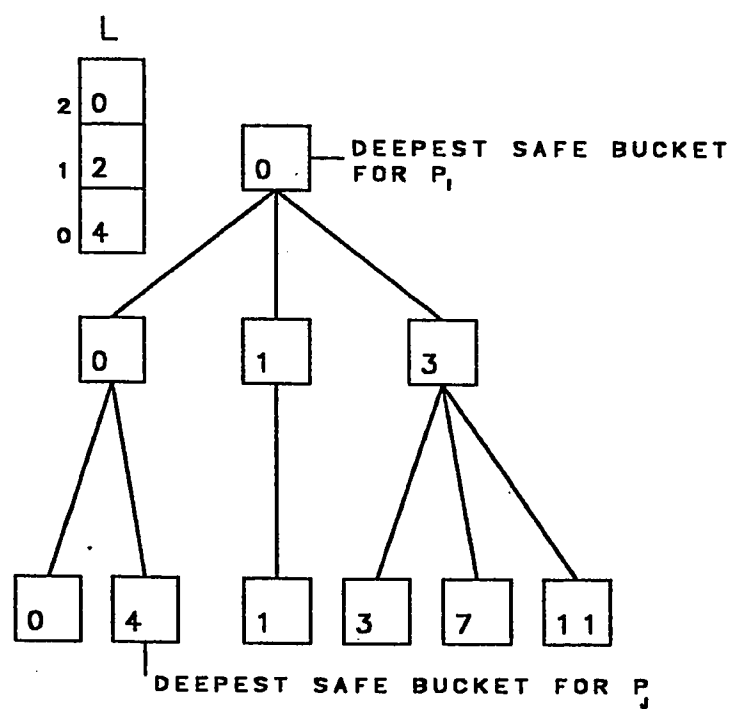


Figure 48: A block diagram for an IBGF.

that a concurrency control scheme using this additional information will not only be sufficient to handle successfully the case discussed above (and other cases) but also allow the design of a systematic method for detecting independence between processes and hence provide a basis for enhancing the degree of concurrency.

5.3.1 Detecting conflict among processes

Definition : The Access Path for process P_i , from the top level directory (root) to a data bucket, denoted by AP_i , is defined as the ordered set :

$$AP_i = \{M_{i(h-1)}, M_{i(h-2)}, \dots, M_{i0}\}$$

where $(h-1)$ is the level of the root directory and M_{ij} is the logical address of a partition at the j -th directory level where $0 \leq j < h$.

The data bucket for process P_i is referred to as the Target Bucket for process P_i and denoted TB_i .

Definition : The portion of the access path AP_i from the deepest safe bucket to TB_i at the data level is called the Scope Access Path for process P_i and denoted SAP_i . It is defined as the ordered set :

$$SAP_i = \{M_{i(k-1)}, M_{i(k-2)}, \dots, M_{i0}\} \text{ for } 0 \leq k < h$$

where $M_{i(k-1)}$ is the deepest safe bucket DS_i for P_i .

Note that it is sufficient to know only the last element of the access path AP_i , that is M_{i0} , to determine the other logical numbers at higher levels. Indeed, let the string $b_{l_0-1}b_{l_0-2}\dots b_0$ be the binary representation of the logical number M_{i0} and l_0 is the search partition level at directory level 0 (data level), then the logical number M_{ij} in the access path AP_i at the j -th directory level is identified by the suffix of M_{ij} of length l_j where l_j is the search partition level at the j -th level.

For example, in the IBGF structure shown in Figure 48 on page 117, the search partition levels for directory levels 2, 1, and 0 are 0, 2, and 4 respectively. Let $TB_i=11$ in decimal, its binary representation is (1011), then the access path leading to this particular bucket is (0 3 11) which corresponds to the binary suffixes (0 11 1011). Similarly, the access path leading to $TB_i=3$ (0011) is (0 3 3) which corresponds to the binary suffixes (0 11 0011).

Clearly, as long as we have only readers no scheduling is necessary. But when updaters are involved it is easy to show that not every schedule of concurrent processes is

correct. An updater may make changes in some subtree (access path) and if another process should access the same path, the two processes have to be scheduled somehow since they may interfere with each other. The discussion below presents a systematic approach for detecting conflict among processes and hence scheduling of concurrent processes.

Definition : Let the two processes P_i and P_j be updaters having the diverging paths SAP_i and SAP_j respectively, then SAP_i and SAP_j are said to be **conflictively overlapping** if they overlap from the root to at least one level beyond the lowest of their deepest safe buckets DS_i and DS_j , as illustrated in part(a) Figure 49.

Definition : Let P_i be a reader and P_j be an updater then the diverging paths AP_i and SAP_j are said to be **conflictively overlapping** if they overlap from the root to at least one level beyond the deepest safe bucket DS_j , as shown in part(b) Figure 49.

Two processes P_i and P_j are said to conflict on an access path if one of the following conditions occurred :

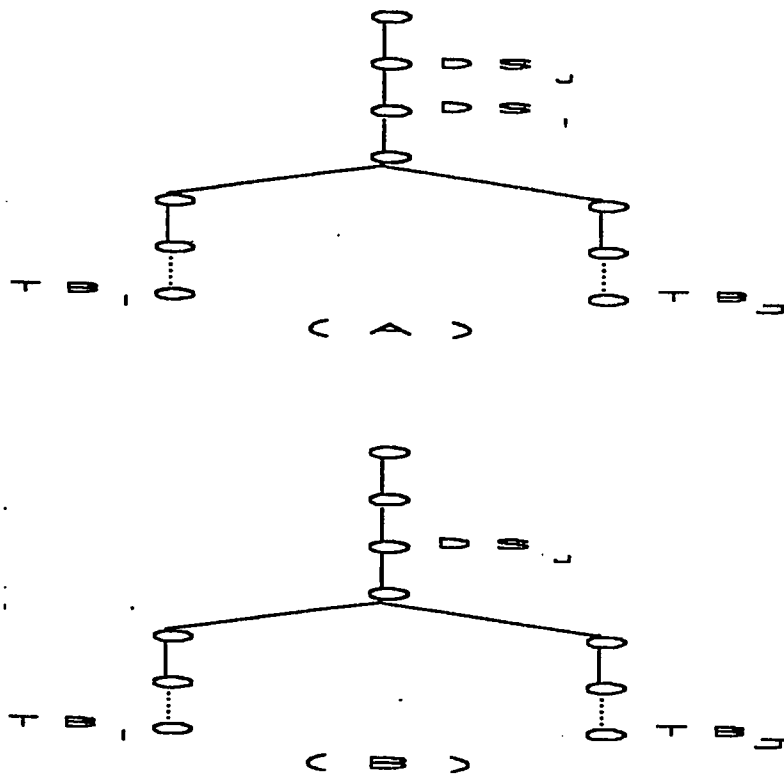


Figure 49: Conflictingly overlapping paths.

- c1. : AP_i and AP_j are identical and at least one of the processes is an updater.
- c2. : Both processes are updaters and SAP_i and SAP_j are conflictingly overlapping.
- c3. : P_i is a read process, P_j is an updater and AP_i and SAP_j are conflictingly overlapping.

The first condition considers the case in which both processes access the same bucket at the data level (i.e. $TB_i = TB_j$). In such a case, the interference between the two processes can take many forms. For instance, an inserter causing an overflow (or a deleter causing an underflow) may lead a searcher to go to a wrong path. If both processes are updaters then it is possible that one process may cancel the effect of the other if the two changes being performed in private buffers are based on the same version (content) of the bucket.

The second condition deals with the case in which the two processes access two different data buckets but they are conflictingly overlapping. The interference here can occur during the propagation of splits and merges being performed by the two processes.

Finally, the third condition addresses the problem where the paths of a reader and an updater are conflictingly overlapping. In this case, the reader may be led to a wrong path due to a split or a merge operation caused by the updater.

The above discussion has shown that there are two types of possible interaction that are of interest whenever two processes must execute concurrently. The first type is the interaction in which an UPDATE and a SEARCH processes are involved. The second type involves the interaction between two UPDATE processes. In the next sections we shall see that in a system of concurrent processes, the interaction between inserters and inserters or inserters and searchers is essentially resolved by using techniques devised for B-trees such as the "LINK" technique and local scope locking [10,18], whereas the interaction between a delete process and other types of processes is more complicated and would require additional treatment from the concurrency control component because of the possible occurrence of a merge operation.

5.3.2 The model of computation

The various modules forming the system controlling concurrent processing are shown in Figure 50. It is made up of a set of concurrent processes $P = (P_1, P_2, \dots, P_n)$, where n is a nonnegative integer. Each new process P_i comes into the system with the request Search(K), Insert(K), or Delete(K). The new process submits its initial access path to a local scheduler which uses it to detect conflict between the new process and the other processes. The scheduler then stores the new process access path in the associated data structure. As mentioned previously, it is sufficient to store only the last element of the access path (i.e. the logical address of the target bucket at the data level). Similarly, when a process completes execution, the scheduler deletes that process access path from the system.

The layout of the scheduler data structure is shown in Figure 51. An entry in the table corresponds to a process. The definition of most of the fields of an entry is self explanatory. The scheduler also uses part of its data structure to implement a block/wakeup system. For each process, the scheduler maintains a list of processes, if any, that are blocked by that process. It also keeps track of the count (Blocking_count) of other processes which are blocking that process. This count is used as a

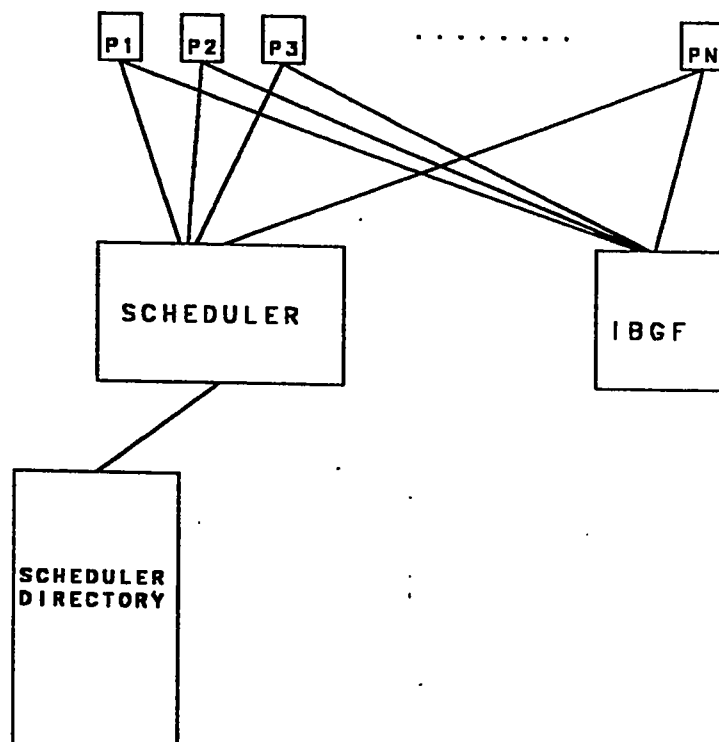


Figure 50: System model.

synchronization variable to signal a process at the time when no other process is blocking it. We shall see later, how the scheduler manipulates information in its data structure. Clearly the number of entries in the table at any given time is exactly the same as the number of the existing processes in the system at that particular time. Its worst-case space complexity is of $O(n^2)$, where n is the number of processes in the system.

The interaction described below involves the concurrency allowed between the split operation and that phase of the search, insert, and delete operations in which the target bucket is being located. Concurrency is enhanced by allowing a process in its searching phase to operate in parallel with a split operation, but there must be some means for it to reorient itself when the wrong bucket is reached. In this scheme the IBGF is modified by adding a single "LINK", pointer field, and a field (PARTVAL) which designates the logical number of the bucket being pointed by LINK, to each bucket as shown in part(a) Figure 52, in a similar fashion as used in B-trees [18]. The purpose of the link pointer is to provide an additional method for reaching a new bucket created as a result of a split operation. When a bucket is split because of data overflow, the set of records is divided into two subsets. The first subset is

```

Type Process_Identifier : Integer;

Type Buffer;
Type Waiting_List is access Buffer;
Type Buffer is
  Record
    Process_ident : Process_Identifier;
    Index          : Integer;
    Next           : Waiting_List;
  End record;

Type Process_Type is (Search, Insert, Delete);
Type Deepest_Safe_Bucket : Integer;
Type Target_Bucket      : Integer;
Type Directory_Level     : Integer;

Type Scheduler_Directory_Entry is
  Record
    Process_id      : Process_Identifier;
    Process_ty      : Process_Type;
    Process_AP      : Target_Bucket;
    Process_DSF     : Deepest_Safe_Bucket;
    DSF_DL          : Directory_Level;
    Blocking_Count  : Integer;
    Blocked_Processes : Waiting_List;
  End record;

Type SchDir is array(1..n) of Scheduler_Directory_Entry;

```

Figure 51: An Ada-like definition of the scheduler data structure.

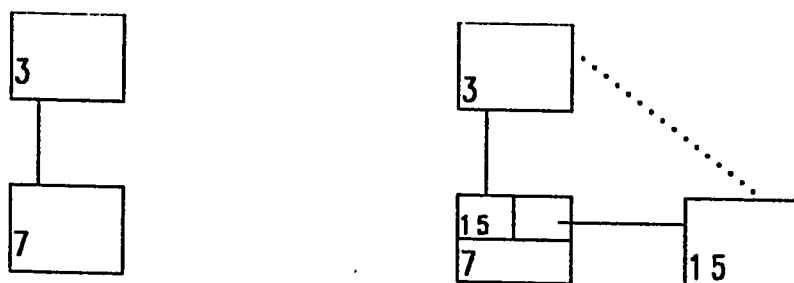
written back into the original bucket, whereas the second set is stored in a newly inserted bucket. The link pointer of the original bucket is then set to point to the new bucket, while the link pointer of the new bucket is assigned the link pointer in the original one, as illustrated in part(b) Figure 52.

The split procedure consists of the following steps :

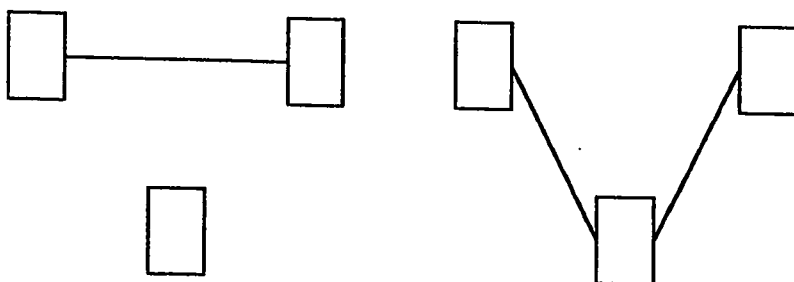
1. Request a new bucket on the disk.
2. Copy some of the records from the original bucket into the new bucket if the split dictates that these records are to reside in the new bucket. Note that at this point, the data is redundant since all the records are still in the original bucket.
3. Set the pointer of the new bucket to point to the bucket pointed by the original one.
4. Replace the original bucket with the new contents including a pointer to the new bucket.

Reading or writing a single bucket is assumed to be inherently atomic. As a result, only the last disk write in the split procedure will make the new bucket available to other processes.

The intent in the scheme is to consider the two buckets resulting from the split, since they are joined by a link, to be functionally the same as a single bucket until the



(A)



(B)

Figure 52: A modified bucket structure.

proper pointer from the father bucket, at the next higher directory level, can be added. We shall see in the next section that link pointers have the advantage that they enhance the degree of concurrency by allowing a searching process to operate in parallel with a split operation.

5.3.3 Scheduling of concurrent processes

The scheduler coordinates access to the file structure. Hence each new process must submit its initial access path to the scheduler, which first stores the access path in its associated directory then tries to detect conflict. A process must also signal the completion of its searching phase to the scheduler which then decides whether to grant it the permission to start its restructuring phase in the case of an updater.

The rules governing process scheduling taking into consideration the conflict detection conditions discussed previously can be stated as follows :

Rule 01 : Let P_i be a reader, then P_i will be blocked if P_j is a deleter in its restructuring phase and C1 or C3 occurred and $DS_j \neq M_{j0}$.

Rule 02 : Let P_i be an inserter intending to start its searching phase, then P_i will be blocked if P_j is a

deleter in its searching phase and C1 occurred or P_j is in its restructuring phase and C3 occurred and $DS_j \neq M_{j0}$.

Rule 03 : Let P_i be a deleter intending to start its searching phase, then P_i will be blocked if P_j is a deleter in its searching phase and C1 occurred or P_j is in its restructuring phase and C3 occurred and $M_{j0} \neq DS_j$.

Rule 04 : Let P_i be a deleter intending to start its restructuring phase, then P_i will be blocked if P_j is a reader and C3 occurred and $DS_i \neq M_{i0}$.

The pseudocode of the scheduler algorithm is shown in Figure 53. The algorithm is written in Ada like notation. The scheduler serves requests through three entry points. A request can be : either start searching phase, or start restructuring phase, or terminate execution. In order to see how this works, we consider a system in which $i > 0$ processes are already running and a new process P_j is requesting to access a path. The scheduler must perform the following steps :

1. If : P_j is in its searching phase then

```

Task body Scheduler is
  n      : Integer := 0; --number of processes.
  Directory : SchDir;
Begin
  Loop
    Select
      Accept Receive_Initial_Access_Path
        (Process_id : in Process_Identifier;
         Process_ty  : in Process_Type;
         Process_AP  : in Target_Bucket) do
        n := n + 1;
        Directory(n).Process_id := Process_id;
        Directory(n).Process_ty := Process_ty;
        Directory(n).Process_AP := Process_AP;
        noconf := 0;
        for I in 1..(n-1)
          Loop
            -- Apply conflict detecting rules on the new
            -- process and the i-th process.
            If conflict then
              noconf := noconf + 1;
              -- Allocate a new cell in the waiting list of
              -- the i-th process.
              -- Insert the new process identifier and index
              -- into the waiting list of the i-th process.
              -- Increment the count of processes blocking
              -- the new process.
            End if;
          End loop;
        If noconf > 0 then
          -- BLOCK the new process;
        Else
          -- SIGNAL the process to start executing;
        End if
      End Receive_Initial_Access_Path;

      Accept Receive_Deeppest_Safe_Bucket
        (Process_id : in Process_Identifier;
         Process_DSF : in Deepest_Safe_Bucket;
         DSF_DL      : in Directory_Level) do
        -- Search for the process_id in the
        -- directory and get its index.
        Directory(index).Process_DSF := Process_DSF;
        Directory(index).Process_DL  := DSF_DL;
        noconf := 0;
        for I in 1..n
          Loop
            If Directory(I).Process_id <> Process_id then

```

```

-- Apply conflict detecting rules on the current
-- process and the i-th process.
If conflict then
    noconf := noconf + 1;
    -- Allocate a new cell in the waiting list of
    -- the i-th process.
    -- Insert the new process identifier and index
    -- into the waiting list of the i-th process.
    -- Increment the count of processes blocking
    -- the new process.
End if;
End if;
End loop;
If noconf > 0 then
    -- BLOCK the new process;
Else
    -- SIGNAL the process to start executing;
End if
End Receive_Deeppest_Safe_Bucket;

Accept Terminate_Execution
    (Process_id : in Process_Identifier) do
    -- Search for the process_id in the
    -- directory and get its index.
    -- Search through the waiting list of the
    -- current process and determine the position
    -- of the processes that are blocked by this
    -- process.
    -- For each process decrement the Blocking_Count
    Directory(position).Blocking_Count :=
        Directory(position).Blocking_Count - 1;
    If Directory(position).Blocking_Count = 0 then
        -- SIGNAL the process to start execution.
    Endif
    n := n - 1;
End Terminate_Execution;
End Select;
End loop;

End Scheduler;

```

Figure 53: Scheduler algorithm.

-store the process id, process type and its access path.

Else : P_j is in its restructuring phase

-store the deepest safe bucket of P_j along with the directory level at which it was found.

2. Apply the conflict detecting rules mentioned previously for P_j against each process P_i in the system.
3. If P_j conflicts with process P_i then insert the process P_j into the list of processes that are blocked by P_i . Increment the counter of processes that are blocking P_j . This means that P_i must access the path before P_j .
4. If P_j does not conflict, the scheduler signals P_j to start with its current phase.

Once a process completes execution, it informs the scheduler. In turn the scheduler deletes the entry corresponding to this process from the table, and decrements the counter of each process that was blocked by this process. If decrementing the counter of a process makes it

zero then the scheduler must inform the process being blocked about the availability of the access path. Note that if a process, say P_j , conflicts with other processes on a particular path, the block/wakeup queue associated with P_j will only contain process identifiers of those conflicting processes that were already scheduled to access that path. This is because we have chosen, for simplicity, to have a FIFO discipline.

5.3.4 Concurrent behavior of processes

5.3.4.1 Search algorithm sketch

To search for a record K in the file, the search process begins at the root directory and proceeds by examining the content of the current bucket to determine the next move in the path down the structure. In each level a pointer will be followed from that level, either to the next level or to a bucket on the same level using the LINK pointer. At each directory level (not including the data level), the process should search the current bucket for the logical number of the target bucket or the logical number of the bucket embedding that one. If the target bucket was found then the process should follow its associated pointer down to the next lower directory level. On the other hand, if an embedding partition was found instead and the link field in

the current bucket is not nil then the process must examine the logical number (PARTVAL) of the bucket pointed by the current bucket. If that number is a possible embedding partition higher than the current one then the process should assume this number to be the current embedding partition. The search process then rectifies the error in its position by following the link pointer to the right instead of following a son pointer as it would ordinarily do. This would mean that the changes that have taken place in the current bucket had not been indicated in the father at the time the father was examined by the search. As a result, the pointer field can be used by a process as a detour to adjust its position.

The search process eventually reaches the data bucket in which K may reside if it exists. If the logical number of the data bucket reached is exactly the same as the logical number of the target bucket computed initially then the search process must start looking for the record inside this bucket. On the other hand, if the two logical numbers are not identical then searching now must begin with the next bucket down the list. In the case the record was not found the search process then moves to the next bucket down the linked list if that one has been created from the current bucket as a result of a split operation to check whether it is an possible embedding partition of the target bucket

higher than the previous one. This procedure is repeated until the record is found or no other embedding partition can be located.

Note that the search process behaves just as a nonconcurrent search and it also does no locking of any type. In the IBGF as mentioned previously, several partitions may be combined to form one region which in turn is mapped into one page in physical storage. A search process working in parallel with an inserter process on such a configuration can assume two possibilities. The first possibility deals with the case in which the actions of the two processes are logically performed in two different partitions. In this case, the reader is not supposed to see the record being inserted. The other possibility addresses the case in which both actions are performed on the same partition. In such a case, if the searcher reads the data page before the new record is added, it is as if the searcher process occurred before the inserter process. Otherwise it is as if the inserter occurred first. This is similar to the assumptions made in [27]. The same argument can be used to analyze the case in which a search process is working in parallel with a delete process that has its deepest safe bucket at the data level. The pseudocode of the search algorithm is shown in Figure 54.


```

Procedure SEARCH(K);
1) /* Compute the logical number of the target
   bucket (TB) */
2) /* start searching */
   Scheduler.Receive_Initial_Access_Path
       (process_id,process_ty,TB);
3) Current := root;
4) While (Current is not a data bucket) do
   begin
       /* search the current bucket for TB or
       the one embedding it. Let this bucket
       be CUREMB. */
       While (TB <> CUREMB and Current.Link <> nil) do
       begin
           If (Current.PARTVAL is a possible embedding
               partition higher than CUREMB) then
               CUREMB := Current.PARTVAL;
               Current := Current.Link; /* move right */
           endif;
       end;
       /* move to the next lower level */
       Current := (Physical address of CUREMB);
5) /* now we have reached to the data level */
   /* Adjust position if necessary */
5.1 If (TB <> CUREMB) then
5.2 If (Current.Link <> nil) then
   /* move to the right bucket if it has been
   created from the current bucket as a
   result of a split and it is a possible
   embedding partition of TB higher
   than CUREMB. */
   If (TB is embedded in Current.PARTVAL) then
       Oldbucket := Current; Current:=Current.Link;
       Lock(Current); Unlock(Oldbucket);
       Goto step #5.2;
   endif;
   /*Search for K in the embedding bucket*/
   endif
   else
       /*Start searching for K in the current bucket*/
       If (not found) Goto step #5.2; endif;
5.3 If (found) report success else report failure;
6) Scheduler.Terminate_Execution(process_id);
End SEARCH;

```

Figure 54: SEARCH algorithm.

5.3.4.2 The insert algorithm sketch

To insert a record K, the insert process, in its searching phase, performs operations similar to that for search process. Unlike the search process, the inserter must keep track of the access path during the descent through the structure.

The insertion of the record K into a data bucket may necessitate splitting the bucket (in the case where it was unsafe). In this case, the process splits the bucket as explained previously. The process then backs up the structure (using the "remembered" list of addresses) to insert the identifier of the new bucket into its parent. This parent bucket, too, may need a split. If so, the process backtracks up the structure, splitting buckets and inserting new pointers into their parents or adjacents, stopping when the process reaches a safe bucket. In all cases, the insert process must lock a node before modifying it.

Note the possibility that as the process backtracks up the structure, due to bucket splitting, the bucket into which the process must insert the new pointer may not be the same as that through which it passed on the way down to the data bucket. The old bucket may have been split, as a result, the correct position for insertion is now some

adjacent bucket down the list to the right of the one where it expected to insert the pointer. This new bucket is reachable using the link field. The pseudocode of the insert algorithm is shown in Figure 55.

5.3.4.3 The delete algorithm sketch

To delete a record K, the delete process, in its searching phase, performs operations similar to that for an insert process. The scheduler gives the delete process an exclusive access to its access path during the restructuring phase of the process. This means that no reader or another updater can exist within the scope once the decisive phase of the delete process has been triggered.

The deletion of the record K from a data bucket may necessitate merging the bucket (in the case where it was unsafe) with the bucket that satisfies the region formation rules. The process then backs up the structure (using the "remembered" list of addresses) to delete the identifier of the bucket being merged from its parent. This parent bucket, too, may need a merge. If so, the process backtracks up the structure, merging buckets and deleting pointers from their parents and adjusting pointers of adjacent buckets, stopping when the process reaches a safe bucket.

Note the possibility that as the process backtracks up the structure, due to bucket previous splitting, the bucket

```

Procedure INSERT(K);
1) /* Use the search algorithm to find the initial
   relevant data page. Keep track of the access
   path leading to this page in (STACK). */
2) Scheduler.Receive_Deeppest_Safe_Bucket
   (process_id, process_DSF, DSF_DL);
3) /* We have a candidate data bucket */
   j := 0;
   Lock(Current);
4) /* Adjust position by moving to the right if
   necessary Recompute TB using  $L_j$ ; */
5) If (Current.Link <> nil) then
   If (TB is embedded in Current.PARTVAL) then
     Oldbucket := Current;
     Current := Current.Link;
     Lock(Current);
     Unlock(Oldbucket);
     Goto step #3
   endif;
endif
6) /* Insert the record K */
   If (Current is safe) then
     Insert K;
     Unlock(Current);
   else
     /* Perform the split procedure */
     K := (coordinates of new created bucket);
     Oldbucket := Current;
     Current := Pop(STACK); /* Backtrack */
     Lock(Current);
     Unlock(Oldbucket);
     j := j + 1;
     Goto step #3;
   endif;
7) Scheduler.Terminate_Execution(process_id);
End INSERT;

```

Figure 55: INSERT algorithm.

from which the process must delete the pointer may not be the same as that through which it passed on the way down to the data bucket. The old bucket may have been split, and hence, the correct position for deletion is now some bucket down the list to the right of the one where it expected to delete the pointer. This new bucket is reachable also using the link field. The pseudocode of the delete algorithm is shown in Figure 56.

5.3.5 Improved throughput

A search process simply reads the buckets along a path from the root directory to some particular target bucket. It makes no changes whatsoever to the data or the structure of the file. Unlike the insert and delete processes, the search process does not need to remember the physical address of each element in its path. Consequently, if we can provide the physical address of the target bucket for a searcher without having to access the structure then it can directly access its target bucket. This would mean that the search process would skip the descent through the structure, and thus, searching processes would execute faster than would have been possible otherwise. As a result, the number of processes executing within one quantum of time will increase and thus throughput will greatly improve.

```

Procedure DELETE(K);
1) /* Use the search algorithm to find the initial
   relevant data page. Keep track of the access
   path leading to this page in (STACK). */
2) Scheduler.Receive_Deeppest_Safe_Bucket
   (process_id, process_DSF, DSF_DL);
3) /* We have a candidate data bucket */
   Lock(Current);
4) /* Adjust position if necessary */
   4.1 If (TB <> CUREMB) then
   4.2 If (Current.Link <> nil) then
       If (TB is embedded in Current.PARTVAL) then
           Oldbucket := Current;
           Current := Current.Link;
           Lock(Current);
           Unlock(Oldbucket);
           Goto step #4.2
       endif;
       /* Search for K in the embedding bucket */
   endif
   else
       /*Start searching for K in the current bucket*/
       If (not found) Goto step #4.2
   endif
5) If (found) then
   /* Delete the record K */
   If (Current is safe) then
       Delete K;
       Unlock(Current);
   else
       /* Perform the merge procedure */
       K := (Identifier of the deleted bucket);
       Oldbucket := Current;
       Current := Pop(STACK); /* Backtrack */
       Lock(Current);
       Unlock(Oldbucket);
       /* Adjust position by moving to the
          right if necessary */
       Goto step #5;
   endif;
endif;
6) Scheduler.Terminate_Execution(process_id);
End DELETE;

```

Figure 56: DELETE algorithm.

To achieve the above goal, a minor modification of the data structure used by the scheduler is needed. The modified version of the structure is shown in Figure 57. This modification includes the addition of two fields to each entry in the table. These two new fields are assigned values at the time when updaters complete their searching phase. The first field represents the logical number of the embedding partition of the target bucket, whereas the second one indicates its physical address. Clearly, the embedding partition and the target bucket may be identical.

As mentioned previously, the physical addresses of data buckets in the access path of updaters will be available to the scheduler only at the time when these processes complete their searching phase. This means that the only search processes which can benefit from this feature are those processes that come at the time updaters are about to start their decisive phase. Moreover, the only type of processes that a search process can operate in parallel with are inserters. Therefore, this feature is limited also to the information provided by inserters.

The intent of using an embedding partition field is to incorporate some of the region formation rules into the logic of the scheduler which in turn will be used to improve the decision making process. The two examples discussed

```

Type Process_Identifier : Integer;

Type Buffer;
Type Waiting_List is access Buffer;
Type Buffer is
  Record
    Process_ident : Process_Identifier;
    Index          : Integer;
    Next           : Waiting_List;
  End record;

Type Process_Type is (Search,Insert,Delete);
Type Deepest_Safe_Bucket : Integer;
Type Target_Bucket      : Integer;
Type Directory_Level    : Integer;
Type Physical_Address   : Integer;

Type Scheduler_Directory_Entry is
  Record
    Process_id      : Process_Identifier;
    Process_ty      : Process_Type;
    Process_AP      : Target_Bucket;
    Embed_Part      : Target_Bucket;
    Targ_Phys_Addrs : Physical_Address;
    Process_DSF     : Deepest_Safe_Bucket;
    DSF_DL          : Directory_Level;
    Blocking_Count  : Integer;
    Blocked_Processes : Waiting_List;
  End record;

Type SchDir is array(1..n) of Scheduler_Directory_Entry;

```

Figure 57: A modified version of the scheduler's table.

below show how the knowledge embodied by the region formation rules can be exploited to improve performance.

Example 1 : Let the target bucket for an update process P_i , already stored in the scheduler data structure, be $TB_i = 3$. Then a process P_j with $TB_j = 3$ should be given the physical address of partition #3 because the two target buckets are identical.

Example 2 : Let the target bucket for the update process P_i be $TB_i = 7$. If the target bucket of process P_i is embeded in partition #3 then a search process P_j with $TB_j = 15$ should be given the physical address of partition #3 because 15 must be embeded in 3 since 7 does not exist.

5.3.6 Further improvement

The previous solution can be generalized to handle also inserters and deleters. This would mean that the scheduler must provide the physical address of each element in the path of an updater since the updater will use these addresses to backtrack up the structure during its restructuring phase. The scheme can be used not only with identical paths but also with paths that have some elements in common. In this case the scheduler will submit only the physical address of each element in that common part. A

process can use this partial information to continue its descent through the structure.

It can be easily observed that the amount of information needed is dependent on the number of directory levels. As a result, an additional overhead will be incurred. It has been shown in [24] that the average number of levels in an IBGF is :

$$h_{avg} = \lceil \log(N/b_0 \cdot \ln 2) \rceil / \log(b_1 \cdot \ln 2)$$

thus the number of directory levels will be small even with large number of records (N). This result shows that the improvement discussed above is achievable with a small information overhead.

5.4 *Freedom from deadlock*

Deadlock freedom is guaranteed by the order that the locking schemes enforce on the processes. The updater process places locks on the buckets following one direction. This means that, once it places a lock on a bucket, it never places a lock on any bucket below it nor on any bucket to the left on the same level. The full proof of freedom from deadlock is given in [18,1].

5.5 Correctness of file modification

The notation of serializability is a great aid to measure correctness of an interleaved execution of some processes. A given interleaved execution of some set of processes is said to be serializable (key set computation [30]) if and only if it produces the same result as some serial execution of those processes. This means that, given an arbitrary initial database state as input, the interleaved execution produces the same output as some serial execution operating on the same initial database state.

It is in this context, we want to show that given a correct initial state of the IBGF then any interleaved execution of some processes adopting the solutions presented previously is serializable and therefore correct. Shasha [30] described an abstract search structure (dictionary) model and has characterized serializable computations on that model. The IBGF can be transformed to capture the characteristics of such a structure. Records are the elements that can be inserted and deleted from a structure.

A data record is a d -dimensional tuple $K = (k_0, k_1, \dots, k_{d-1})$ of value which correspond to attributes A_0, A_1, \dots, A_{d-1} respectively. We assume these records come from a possibly infinite set identified by the cartesian product $D_0 \times D_1 \times \dots \times D_{d-1}$ of attributes

A_0, A_1, \dots, A_{d-1} domains. This set is called Record-Space. A state of the structure consists of a five-tuple $(B, E, \text{root}, \text{contents}, \text{edgeset})$, where B is a set of buckets, E is a set of directed edges, and root is a distinguished member of B . Contents is a function from buckets to subsets of Record-Space. Intuitively, $\text{edgeset}(b, b')$ is the set of values x such that a process searching for x that arrives at bucket b would continue to b' . As a result, this section takes advantage of that model to show correctness of these concurrency algorithms.

A process is implemented as a sequence of operations. An operation O is guaranteed to be atomic in a process P if no operation outside P can modify any data accessed by O while it executes. Operations are classified into **decisive** and **nondecisive** operations. The decisive operations change the global contents of the structure. On the other hand, nondecisive operations do not change the global contents of the structure.

In all the algorithms presented previously, a process must hold a lock on all the buckets accesses before the process issues the operation and must hold the locks until the operation completes. If the operation modifies a bucket, the lock the process holds on that bucket must be an exclusive lock.

To support the model, the concurrency algorithms are based on **lock-coupling** and **link** techniques. Lock-coupling ensures that the searching phase of each process never deviates from the most direct path. Link technique establishes a reachability relationship from a wrong bucket to a right bucket. According to the key set computation theory [30], if the structure begins in a good state and the processes use the lock-coupling or the link techniques then they are serializable.

Chapter VI

CONCLUSION AND FUTURE WORK

The IBGF exhibits structural properties and performance characteristics which makes it highly suitable for organizing relations in relational database systems and for the efficient execution of relational operators.

In this research, we have presented some algorithms which performs correct concurrent processes on the IBGF. A survey of other solutions developed for some popular data structures has been presented. The discussion of concurrent operations in an IBGF has been divided into two main parts. The first part dealt with the case of uniform data distribution, whereas the second part investigated the nonuniform case.

The solution presented for uniform data distribution was an adaptation of the solution developed for linear hashing files. In the second part, one version of B-tree solutions was used to investigate the problem of supporting concurrent processes in the IBGF. The solution proved to work successfully because of the similar operational properties between the two structures.

The major concern of this research was to design a concurrency scheme which exploited and incorporated the properties of the IBGF to maximize the degree of concurrency. A new concurrency scheme was developed which exploited the numbering property of the structure. This solution resulted in a performance similar to the B-tree solution in the worst case and a better degree of concurrency in the average case.

In the new solution, we have defined a systematic approach which demonstrates an effective method for detecting conflict between processes. This approach was expressed in terms of conditions and rules. A scheduling system was designed to employ these rules in a concurrent environment. It has been shown that this system uses the minimum amount of information to achieve higher degree of concurrency. Finally, the numbering mechanism has been investigated also to insure more improvement in the throughput.

As in other cases of concurrency in data structures, making modifications to the data structure has proved to be a useful technique for achieving greater concurrency. In the solutions presented, the modifications were relatively minor (i.e. the addition of a locallevel field to each bucket chain as in the first part and the addition of the link

field to each bucket as in the second part), yet sufficient to detect the effects of concurrent updates and allow the search to resume from that point along an alternative path. These solutions also demonstrated that lock-coupling protocols, found to be useful in other structures, carry over to the IBGF structure.

Immediate future work is to implement these algorithms on a database stored in terms of an IBGF. The scheduler system can be generalized to include predicate locking. The methodology used to construct the scheduler system suggests that it is possible to realize such system in a distributed environment. The scheduler represents the allocator (server) while the buckets in the file represents the resources. The function of allocation can be centralized in one dedicated site or distributed on remote ones. Similarly, the IBGF structure can be centralized or duplicated on different sites. As distributed configurations become widespread, it is worth investigating how to manage concurrent processing in IBGF which in turn may be an important contribution to performance of database systems.

REFERENCES

1. Bayer, R., and Schkolnick, M. Concurrency of Operations on B-Trees. *Acta Informatica*. 9(1977), Pages 1-21.
2. Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.
3. Bernstein, Philip A., and Goodman, Nathan. Multiversion Concurrency Control Theory and Algorithms. *ACM Transactions on Database Systems* 8(4), December 1983, Pages 465-483.
4. Burkhard, Walter A. Interpolation-Based Index Maintenance. *Proc Second ACM-SIGACT-SIGMOD Symp on Principles of Database System*. 1983, Pages 76-85.
5. Date, C.J. *An Introduction to Database Systems volume I Fourth edition*. Addison Wesley, 1986.
6. Date, C.J. *An Introduction to Data Base Systems volume II*. Addison Wesley, 1986.
7. Deitel, Harvey M. *An Introduction to Operating Systems*. Addison Wesley, 1984.
8. Coffman Edward G., and Denning, Peter J. *Operating System Theory*. Prentice-Hall, N.J., 1973.

9. Ellis, Carla Schlatter. Extendible Hashing for Concurrent Operations and Distributed Data. *In Proceeding of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database systems (Atlanta, Ga., Mar. 21-23)*. ACM, New York, 1983, Pages 106-116.
10. Ellis, Carla Schlatter. Concurrency in Linear Hashing. *ACM Transactions on Databases Systems* 12(2), June 1987, Pages 195-217.
11. Ellis, Carla Schlatter. Concurrent Search and Insert in 2-3 Trees. *Acta Informatica*. 14(1), 1980, Pages 63-86.
12. Hsu, Meichun, and Chan, Arvala. Partitioned Two-Phase Locking. *ACM Transactions on Database Systems*. 11(4), December 1986, Pages 431-446.
13. Khayat, Mohammad G. A Concurrency Measure. *IEEE Transactions on Software Engineering*, SE-10(6), November 1984, Pages 804-810.
14. Kroenke, David. *Database Processing*. Science Research associate, Inc, 1983.
15. Kung, H.T., and Papadimitriou, C.H. An Optimality Theory of Concurrency Control for Databases. *Acta Informatica*. 19(1983), Pages 1-11.
16. Kung, H.T., and Lehman, Philip.L. Concurrent Manipulation of Binary Search Trees. *ACM Transactions on Database Systems*. 5(3), September 1980, Pages 354-382.

17. Kwong, Yat-Sang, and Wood, Derick. A new Method for Concurrency in B-Trees. *IEEE Transactions on Software Engineering*, SE-8(3), May 1982, pages 211-222.
18. Lehman, Philip L., and Yao, S. Bing. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*. 6(4), December 1981, Pages 650-670.
19. Litwin, Witold. Linear Hashing : A New Tool for File and Table Addressing. *In Proceedings, 6th Conf on very Large Databases*. 1980, Pages 212-223.
20. Manber, U., and Ladner, R. E. Concurrency Control in A Dynamic Search Structure. *ACM Transactions on Database Systems*. 9(3), September 1984, Pages 439-455.
21. Michael, Freeston. The BANG File : a New Kind of Grid File. *In Proc ACM-SIGMOD Conf., 1987*, Pages 260-269.
22. Nievergelt, J., Hinterberger, H., and Sevcik, K. C. The Grid File : An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*. 9(1), March 1984, Pages 38-71.
23. Onuegbu, E. O., and Du, H. C. A Locking Scheme for Associative Retrieval, Unpublished Paper.
24. Ouksel, M. The Interpolation-Based Grid File. *In Proc of Symposium on Principles of Database Systems*. 1985, Pages 20-27.

25. Ries, Daniel.R., and Stonebraker, Michael. Effects of Locking Granularity In Database Management System. *ACM Transactions on Database Systems*. 2(3), September 1977, Pages 233-246.
26. Sagiv, Yehashua. Concurrent operations on B-trees with overtaking. *In proc of the 4th ACM-SIGACT-SIGMOD Symposium on Principles of Database Systems*. 1985, Pages 28-37.
27. Salzberg, Betty. Grid File Concurrency. *Inform Systems*. 11(3), 1986, Pages 235-244.
28. Samadi, B.S. B-Trees in A System With Multiple Users. *Inform Process Lett.*, 5(4), 1976, Pages 107-112.
29. Schlageter, Gunter. Process Synchronization in Database Systems. *ACM Transactions on Database Systems*. 3(3) September 1978, Pages 248-271.
30. Shasha, Dennis and Goodman, Nathan. Concurrent Search Structure Algorithms. *ACM Transactions on Database Systems*. 13(1) March 1988, Pages 53-90.