A STUDY ON CONCURRENT OPERATIONS

IN EXTENDIBLE HASHING

INVOLVING MERGING

By

PRASAD VENKAT SIVALENKA

Bachelor of Engineering

Osmania University

Hyderabad, India

1991

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 1993

OKLAHOMA STATE UNIVERSITY

A STUDY ON CONCURRENT OPERATIONS

IN EXTENDIBLE HASHING

INVOLVING MERGING

Thesis Draft Approved:

Huizhu Lu

Thesis Adviser

Mitchell L. Neilsen

Donald D. Fisher

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and gratitude to my major advisor Dr. Lu, for her guidance, motivation, encouragement and invaluable assistance. I am also thankful to Dr. Mitch Neilsen and Dr. Paul Benjamin for serving on my graduate committee.

I am grateful to my parents, Mr. S.A.V. Sastry and Mrs. Leela for their moral encouragement, direction and financial support. My family members deserve my deepest appreciations for their love, understanding and sacrifices through out my studies. I extend a special and sincere thanks to Miss Padma for her unfailing confidence, love and support.

Finally I would like to thank God, without His blessings this thesis would not have possibly moved an inch.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. LITERATURE REVIEW 7

Brief review of extendible hashing 7
Ellis's Algorithm 8
Find Operation 10
Insert Operation 11
Delete Operation 11
Hsu and Yang's Algorithm 12
Search Operation 13
Insert Operation 13
Delete Operation 15
Kumar's Algorithm 16
Search Operation 17
Insert Operation 18
Delete Operation 20
Summary 23

III. A NEW MERGING TECHNIQUE FOR
CONCURRENT OPERATIONS IN EXTENDIBLE HASHING 27

Modification to Directory 27
Search Algorithm 31
Insert Algorithm 32
Delete Algorithm 33

IV. PROOF OF CORRECTNESS 36

Proof of Termination 36
Correctness of Search Algorithm 37

Chapter Page

Correctness of Insert Algorithm 39
Correctness of Deletion Algorithm 42
Search Algorithm 45
Insertion Algorithm 46
Deletion Algorithm 47

V. PERFORMANCE ANALYSIS 48

VI. CONCLUSIONS 56

REFERENCES 58

LIST OF TABLES

Table — Page

I. Locking Protocol 9

II. Compatibility Matrix 16

III. Transaction Set 48

LIST OF FIGURES

Figure Page

1. Directory and Bucket 3

2. Bucket Splitting 3

3. Directory Splitting 3

4. Potential for Merge 4

5. Merging Bucket 4

6. Structure Before Transaction 15

7. Structure After Transaction 15

8. Structure Before and After Transaction 23

9. Modified Data Structure with Data Pages 30

10. Initial State of File 50

11. Transition State 50

12. State after Split - Set1 52

13. State After Split - Set2 53

14. State After Merge - Set2 55

CHAPTER I

INTRODUCTION

Since the inception of database systems, the processing and space demands have been increasing more rapidly than the power of transaction processing systems. The database sizes have grown to such a magnitude, they require a lot of memory, usually in the range of gigabytes. Because of this size, designing an efficient data structure for a fast access mechanism has become the order of the day.

The most often used data structures in databases are B-tree [4] or its variants as main data structures and a combination of dynamic and static hashing [5] as auxiliary data structures. But the B-trees cannot grow dynamically or shrink efficiently. A dynamic data structure called extendible hashing that can accommodate expansion and contraction of data concurrently with the other operations on it has been discussed [4]. Since extendible hashing does not require any major reorganization, they are a powerful data structure for database systems.

The power of a data structure can be utilized only with the help of efficient algorithms. There are a few algorithms for synchronizing concurrent operations on extendible hash files. The first one is proposed by Carla Ellis [1], the second one by Hsu and Yang [2] and the latest one by Kumar [3]. Hsu and Yang used a new concept, called *Verification bits*, which has greatly reduced the usage of locks, thereby reducing

locking overhead. Kumar further developed this concept, i.e., verification processes at the right moment during the execution of concurrent operations to guarantee data consistency, eliminating to some extent, the need for locking the directory. The verification process minimizes the number of atomic actions for serializing concurrent operations and reduces the locking overheads without effecting the performance.

Though the above said algorithms are good, they do not deal with either bucket collapsing or merging. Merging is very important because it saves a lot of memory and also saves processing time. Because merging may lead to directory merging, the number of entries will be more at any given time in the RAM, there by resulting in fast accesses. Thus merging saves both time and space which are very important. But there can be a trade off between overhead cost of transferring data from disk to RAM and fast access if a lot of merges and splits take place.

There has been a new technique for merging proposed by Shasha and Johnson[16] called the free at merge, in B-trees. But this has a disadvantage, a lot of memory is wasted, because merging does not take place until the whole leaf is empty. Reorganization in a B-tree takes a lot of time. While proposing this, they assumed that there would be more insertions than deletions, which is not possible in a real-time data. Another merging technique is also presented which is known as the merge at half. These techniques which were presented are for B-trees and not for Extendible Hashing. The merging concept in Extendible Hashing is explained in brief here.

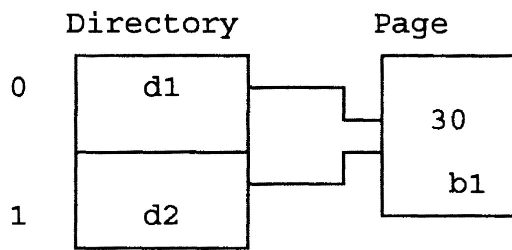

Figure 1. Directory and Bucket

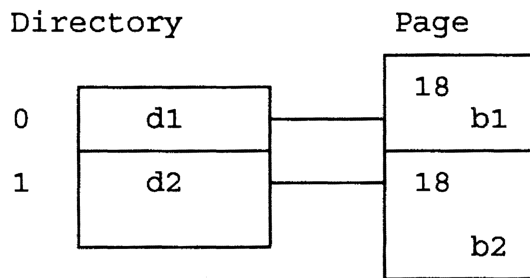

Figure 2. Bucket Splitting

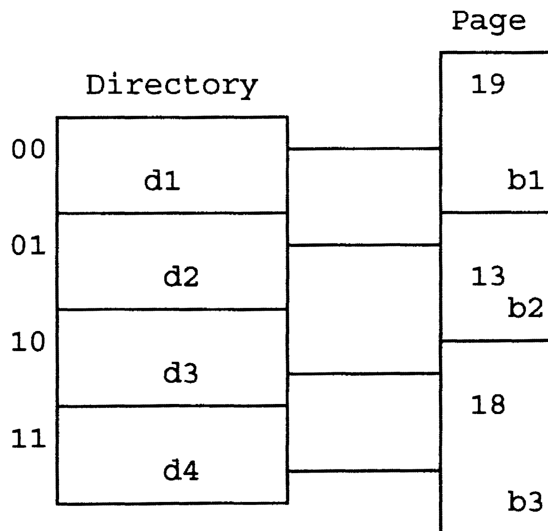

Figure 3. Directory Splitting

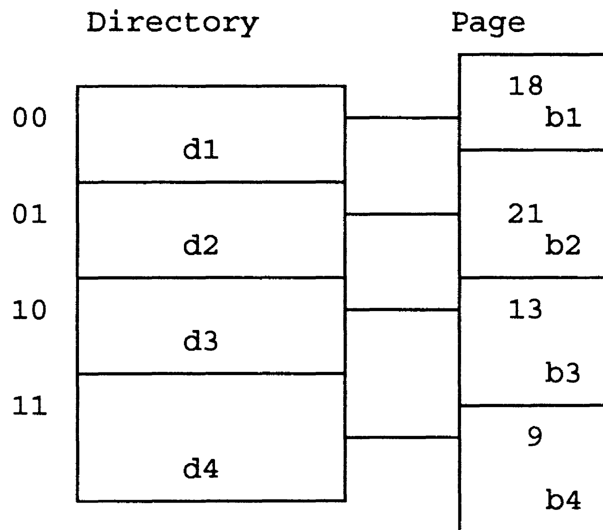

Figure 4. Potential for Merge

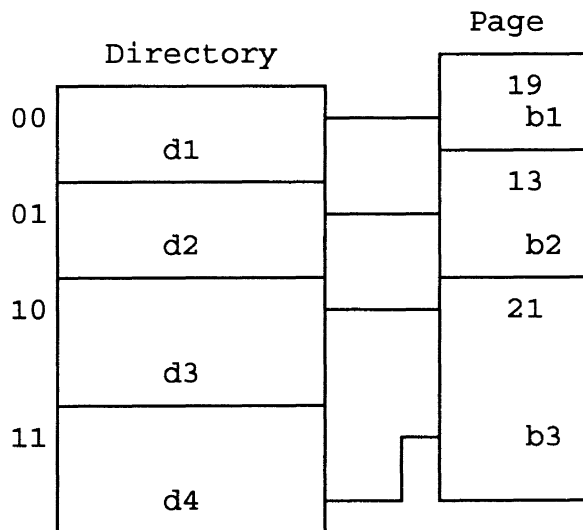

Figure 5. Merging Bucket to Save Space

Both local depth and global depth are not explained in this example, but they are taken care of. This is a case in a concurrent operation. Let us consider a bucket having two directory entries pointed to it, d1 and d2. Let us assume there can be 30 entries in the bucket. After having 30 entries filled up, the bucket splits, because it cannot accept

any more entries. Again if the bucket is full this time both directory and bucket have to be split. So, it splits (Figure 3). After a few more insertions the bucket 3 splits, after this we say there were few deletions. The resulting state of the file is shown in Figure 4. We see that number of entries in bucket 3 and 4 total together are only 21. But each bucket can hold up to 30 entries. Thus, there is a potential for merge. But in the present algorithms, merging can take place only if there is no lock on bucket at that moment. If it has a lock then merging does not take place, resulting in a loss of space. Because what could have been in one bucket now there are two buckets storing the same amount of keys. To avoid this, merging has to take place. Thus Figure 5 shows how it would be if merging was to take place.

In the above example, we have seen bucket 3 splitting into bucket 3 and bucket 4. But immediately there were few deletions taking place, resulting in a potential for merge. This is a drawback because immediately after a split we are trying for a merge, which results in wastage of our system resources. This type of splitting is known as blind splitting. One more major drawback which has not been taken care is blind merging. Blind merging leads to page split and may even lead to directory split. This would not be beneficial, because within few seconds after merging, if there were few insertions which may lead to a split, then it would result in heavy loss of system resources.

The research objective of the thesis is to propose a new merging technique for concurrent operations in extendible hashing. The new technique requires less storage space and may provide faster access speed, than other major techniques.

In this thesis, Chapter II tells us about the past work done on concurrent operations in extendible hashing. This also gives us an idea about how operations are performed on an extendible hash file. Chapter III, explains the new algorithm proposed. It also introduces *Forced insert, Check Lock* and *Wait Status*. Chapter IV gives the proof of correctness of the proposed solution. Chapter V gives an analysis of the proposed solution with appropriate examples and figures. We conclude the thesis with Chapter VI which includes conclusion and a discussion for future work.

CHAPTER II

LITERATURE REVIEW

The growing size of database systems has made an extendible hash file, which is a dynamic data structure, an alternative to B-trees. In this chapter some solutions proposed by certain researchers are discussed. The first algorithm allows concurrency by using locks. The other two algorithms allow concurrency without having to acquire locks on both the directory entries and on the data page. These algorithms use a process called verification.

Brief Review of Extendible Hashing

Consider a typical data structure which is generally used in extendible hashing. This data structure has two parts (1) a directory, which is an array of pointers and (2) a set of buckets which are on the secondary storage and contain keys and associated information. There is a hash function to generate a pseudo key which is used to index into the directory. This pseudo key directly gives the depth of the directory which changes as the file grows or shrinks [15]. The significant bits (left most) are used to give the depth of the directory. For example, if the directory depth is two, it means that there are four entries. If the directory depth is three then the number of entries is said to be eight. Each bucket has a local depth, which match with the common bits of the

keys present in that bucket [1, 14]. It is possible for multiple directory entries to point to the same bucket, provided the local depth is less than or equal to the directory depth or the global depth [9, 14]. Splitting of a bucket results in rehashing the keys and redistributing these keys, along with the key to be inserted in these two buckets. This may also increase the local depth which might double the directory. Merging may result in directory contraction.

Researchers like Carla Ellis, Hsu and Yang, and Vijay Kumar employ the above data structure in the solutions they propose however, they make some modifications to the basic structure. The following sections explain how these algorithms work. Appropriate examples are given at the end of each section.

Ellis's Algorithm

The structure proposed by Ellis is the modification of the data structure (explained at the beginning of the chapter) which includes three new features.

1. links
2. common bits
3. count

The first feature, links are used when there is a split or merge in a bucket. Links aid in easy recovery from concurrent restructuring operations. Each bucket points to the next bucket. For example, if there is a split or merge, the new bucket will have the next pointer of the old's next pointer, and the old bucket will point to the new bucket. Thus there is an added advantage of having an extra path for the desired data during a search

operation when the information is involved in a split or merge operation. With the help of this field i.e., links, we make sure that there is no loss of memory. Because, if there was a potential for merge then with the help of the links we can merge the buckets. With the help of the second feature, common bits, it can be determined whether a process has a wrong or right bucket by comparing the common bits in the data structure with that of the pseudo key. Common bits field has proved to be effective and indispensable and has been used widely in various other solutions with slight modifications.

The third feature, count (named depth count) allows us to record the number of buckets whose local depth equals depth [1]. This solution uses a locking protocol which is shown in Table I. This has mainly three locks, which are used in different combinations to avoid deadlocks. These locks are on the directory and on the individual buckets. The following section explains how Ellis's algorithm works.

TABLE I

LOCKING PROTOCOL

Lock	Request	Existing Lock		
		P	Q	E
P	Read lock	Yes	Yes	No
Q	Selective lock	Yes	No	No
E	Exclusive lock	No	No	No

Find Operation

This process executes a search or find operation, a P-lock locks the directory and then (1) reads the depth of the directory (2) gets the required bucket pointer. After identifying the bucket which is to be searched, the reader places a P-lock on the directory, and writes the contents into a private buffer. If a split occurs after selecting the bucket and before placing a P-lock on the bucket (data page), this type of locking may result in the reader getting a wrong bucket. Here, local depth, low order bits of the pseudo key do not match with the common bits of the bucket [l]. The correct bucket can be reached by using links. Moreover, a P-lock on a bucket is released only when the data received is correct. While using links the next bucket is always P-locked before releasing the P-lock on the current bucket. This helps processes from leapfrogging each other [l]. A find operation attempts to lock more than one bucket.

Insert Operation

In an insert operation a Q-lock is placed on the directory, and is removed only when there is no need for further directory manipulation. Moreover, any changes made to the structure during this operation appear as atomic operations. The splitting of a bucket also appears atomic. During an insert operation, if the bucket is not full, then the key can be inserted into the bucket (data page). But, if the inserter's bucket is full, the bucket is split into a pair of buckets, and the keys are distributed between the two buckets. The new key is placed according to the significant bits of the key into either

of the buckets. Since the keys are divided into two halves, the second half of the pair is written first into a newly allocated disk page and the first half of the pair replaces the old bucket. Since the first half is written last, the new bucket is not reached by the pointer in the hash file. This shows that the operation is equivalent to a single operation of writing the first partner.

If the data has moved to the second half this is detected and it is followed by the link. An inserter may have to double the directory, depending upon the depths. So, this shows that the whole operation is atomic. The directory space is extended and the old contents copied prior to incrementing depth make the new directory entries visible [l]. The reader may see the intermediate stages of an insertion operation [1, 13].

Delete Operation

A delete operation exclusively locks the directory, the target bucket and its partner (i.e., another bucket) while modifications are taking place. Moreover, in a delete operation, unlike the insert operation, the intermediate stages are not visible. A deleting process uses E-lock. An E-lock is used, because a reader might interfere during a delete operation by trying to access an invalid directory entry. Inorder to avoid the above inconvenience, an E-lock is used. If a bucket has only one entry, and that entry is to be deleted, then an attempt is made to merge the empty bucket with its partner.

Two buckets are defined as partners with respect to bit position d if their common bits match in bits d-l to 1, and differ at bit d. E-locking depends upon bucket order. For instance, if there are two buckets 0 and 1, a process trying to delete from the second

bucket must release its lock on that bucket inorder to get both partners locked according to ordering. In addition, it is impossible for a process to read a pointer for a bucket that will be deallocated before it can make its lock request. This is because a deleter excludes the other process from parts of the data structure that contain pointers to the buckets being removed. This shows that it is important to ensure that lock requests are eventually satisfied.

The above solution, however has some disadvantages. There may be locks on the directory for an insertion or deletion operation which may create a deadlock problem. This solution is a single step transaction i.e., no two insertions or deletions can be performed at the same time. The use of locks extensively increases the locking overhead considerably. This type of locking does not bring about any marked improvement in the performance. The above solution was proposed to overcome the shortcomings in the two phase locking systems [12]. But this solution is not satisfactory as splitting leads to the formation of new links, which increase complexities such as space and time. This has become more or less a linked list implementation. This type of locking when used may result in block transactions which may severely effect the concurrency. The above disadvantages are minimized in the algorithm proposed by Hsu and Yang [2].

Hsu and Yang's Algorithm

In the algorithm proposed by Hsu and Yang [2], the issues of underflow and compaction are ignored. This algorithm uses a technique called the verification process which is used in the processes such as Search, Insert and Delete.

The data structure (which was explained at the beginning of the chapter) is modified by including one more column called the *verification field*, in the directory field. Let us now examine how Hsu's algorithm works.

Search Operation

In this process the key to be searched is sent to the hash function to obtain the pseudo key. The directory entry is determined by indexing the significant bits of the pseudo key. The directory gives a pointer to the data page or the bucket. Then there is a sequential search in the bucket for the key. A wrong page search may result if a split occurs in the data page (bucket) during a search operation. Normally, to avoid a wrong data page search we have a lock on the directory until the operation ends. This type of locking is avoided in this algorithm by re-reading the directory entry when a search operation cannot find the key in the data page it has just read. This form of re-reading, or verification continues until the key is found, or the value of the directory does not change between two consecutive readings [2]. This algorithm verifies the directory entry it has read previously, thereby making sure that search operation is not a failure.

Insert Operation

The key to be inserted is sent to the hash function to obtain the pseudo key. By seeing the significant bits (that are decided depending upon the directory depth), it is determined which directory entry has to be read. By reading the directory entry, a

pointer to the data page is obtained. The data page is searched to determine whether the key is present in the data page, and if not the key is inserted into the page. If the data page is full then a split is performed, resulting in the formation of a new data page (bucket). Then the directory entry is updated. When two different insertions are performed at the same time interference occurs. To eliminate interference, in the previous algorithm locks were used on the directory, and on the data page until the operation was over. But in this algorithm locks are not used. Instead during this operation, verification is done on the content of the directory it has previously read after locking the data page and before performing updates on the data page [2]. If the *verification* fails, the operation will unlock the page and will lock a different one and perform another verification. So, by this it can be concluded that another page or directory is not locked, while one is locked. Thus, deadlock is eliminated.

During splitting, the newly allocated page is locked until the effected directory entry or entries are updated. The algorithm deals with the splitting of a data page and decides where to insert the key in a different fashion than the one described by Ellis [l]. In this algorithm, the splitting page and also the newly allocated pages are locked. Then these keys (contents) are kept in an order in a separate buffer and the key to be inserted is also inserted into this buffer. These are written back to the database. Then all the new pages and the splitting page are unlocked one after the other. In this way, the multiple directory entries are updated one by one but in one atomic action. Single entries are updated in one atomic action.

Delete Operation

The deletion algorithm is similar to the insertion operation, except that this algorithm does not deal with the issue of underflow and page merging. An example given below to show how deletion works. Example: Consider a situation having two adjacent directory entries d1 and d2, pointing to the same data page p, where p at present has a local depth which is less than the global depth. An insertion operation T1 is supposed to take place.

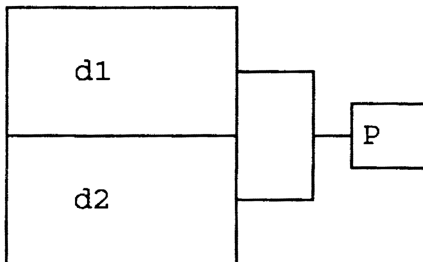

Figure 6. Structure Before Transaction

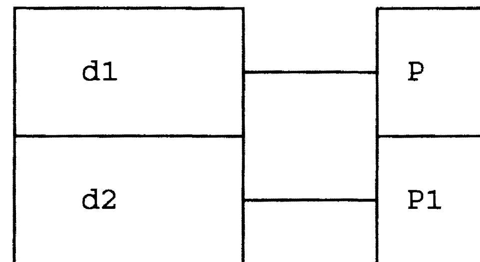

Figure 7. Structure After Transaction

Transaction T1 indexes the directory entry d1 to insert the key in the data page P. Transaction T1 after indexing the directory entry and before inserting the key is suspended. When T1 is rescheduled, it tries to insert the key in page P. By doing so, it may have inserted in a wrong page. Because there might have been modifications to the directory, before T1 was rescheduled. This is how the whole process works in the conventional system. But Hsu algorithm works in this way.

Transaction T1 knows in what page the key is to be inserted. T1 goes to that

page and sees whether the page is locked or unlocked. If it is unlocked then this locks the page, just before locking the page it verifies that it is locking the correct page by using directory verification. The bits are compared. This is done because just after reading and before putting a lock on the page many things might have happened. That is there, might have been a directory split. So, after verifying and when it is found correct, the page is locked. The page is split and the keys are rehashed inorder to give them the correct pages to reside in. After this, it tries to update the directory entries after directory verification and then unlocks the page. This is how we insert the key in the page. If the directory verification fails, it ends the transaction without completing. It is rescheduled to perform the same operation after some time.

Though this algorithm eliminates the locking problem considerably and insertion can be done while searching, it does not take into account the problems created because of compaction and overflow. This is a single step transaction and therefore two insertions are not possible concurrently into the same page. The third solution proposed by kumar [3] uses the two phase locking and has the advantages of the above solutions.

Kumar's Algorithm

This algorithm called the Extendible Hashing Cautious Waiting was proposed by Kumar [3]. It is based on a two-phase locking policy.

The directory entry in the data structure (explained at the beginning of the chapter) has two additional columns.

1. *Verification Bits* denoted by VB

2. *Page Status* denoted by PS.

The verification bits are the most significant bits of the pseudo key, which increase with the increase in the directory entries. The page status (PS) in the directory gives the number of pages in each data page. The algorithm uses a principle called rolling back or blocking requestor. This principle is used whenever there is a conflict between two transactions, i.e, which one should be allowed to complete its operation first. This algorithm also uses three locks and the locking protocol is given in Table II. The following section explains how Kumar's algorithm works.

TABLE II

COMPATIBILITY MATRIX

	read	write	ce*
read	YES	NO	YES
write	NO	NO	YES
ce	YES	YES	NO

* Contraction/Expansion mode

Search Operation

The key to be searched is taken and sent to the hash function to obtain the pseudo key. Then, with the help of this key, we go to a directory entry. The verification bits are compared with the index bits (obtained from pseudo key). This is a directory level

verification. If we find during comparison that both the bits are same, then the directory entry is correct. If the comparison gives that the bits are different then, the whole process of checking the directory is repeated by checking the most significant bits of the pseudo key. Indexing onto the directory is done again. The VB are compared with the index bits. This process is repeated until we find a match. The process ends when no match is found, because the upper limit of the directory is fixed.

When a match is found, then it is said that we have found a directory entry. By taking the pointer from the directory to the bucket or page we perform page level verification. In this verification the VB bits are compared with the local depth of the page. If the VB bits were equal to the local depth bits, then the key is supposed to be there in the page. By storing the address of the page and the local depth, the key is searched in the page [14]. If found, it returns a success message, else it may give another type of success message. But if the VB bits were less than the local depth, it means that a split has occurred and the whole process is repeated.

Insert Operation

For an insertion, we apply the search function. If the search did not find a page then it is said that there is no record with this key. So, insertion can be done. The page address and the local depth are passed to the calling routine. The page that is obtained is the one where the key can be inserted. This page is locked in an exclusive mode and its current local depth value is compared with that of the local depth value obtained from the search operation. This is done in order to verify that no split has occurred in

between. If they do not match, then we again restart the whole process right from the beginning i.e., from search operation.

If the page's local depth matches, then it is a correct page. The page status gives the number of records in the pages. If the page status number is equal to the number of keys that can be inserted in a page, then we split the page. If it is less, then we will directly insert the key. For a page split, local depth of the page and the directory depth (global depth) are compared. If it is equal, then there will be a directory split, resulting in doubling of the directory size [12]. The split is allowed to take place, and the new page which resulted due to split is locked in an exclusive mode. Redistributing of keys is done, and the insertion of the keys takes place in the right pages and pointers are updated.

During the process, if a directory is locked then the real problem comes into effect. But, Kumar in his algorithm has effectively tackled this problem by using his principle "Rolling Back or Blocking Requestor." If the directory is free, then it is locked in a ce *(contraction/expansion)* mode. If the directory expansion is required, the directory is doubled (expanded) and the links to pages are rearranged. The entries in the directory, namely VB and PS are also updated. If there is no requirement for a directory expansion, meaning that local depth is less than the directory depth or global depth, then the pointers are readjusted and the PS fields are updated. The locks on the directory are released, but, not on the page because verification has not been done on the page. After verification on the pages is done, the locks are removed. This is how insertion works.

Delete Operation

The delete operation also uses the application of search. The page address and the local depth are passed to the calling routine. A delete can proceed only if a search operation succeeds. The page is locked in an exclusive mode which we have obtained from the search operation. The local depth of this page and the local depth of the page to be searched is compared. If they match, then the page is correct, and the record is deleted. If the local depths of the two pages do not match, a split or merge might have occurred due to another process (such as insert or another delete operation). So the entire process right from the search operation is repeated. Since the upper limit of the directory is fixed there is no chance of it going into an infinite loop.

It is checked whether the page can be merged or not. PS value of the present page and its sibling page is added and checked (for a potential merge). If the value is equal or less than that of the number of pages in a bucket, then there can be a page merge. For example, we have the value PS of present page as 2 and the value of PS of the sibling as 1. The total value is three. As our page capacity is three (example) so, there can be a merge. Thus reducing the local depth, and sometimes even the directory depth. To achieve a page merge the sibling data page is locked in the write mode. If this page is locked by another transaction/process, it skips the merge operation and execution is continued. Merging is done only to save memory. Further details are given in the summary section. All the records are moved into one page, if the sibling page was not locked.

If the directory was locked, then a conflict would arise as to which one should be

allowed first. This is resolved by using the rolling back or blocking requestor principle. If the directory is free then it should be locked in ce mode. If the directory contraction has occurred, then the entries in the directory are updated like the VB and PS fields. The pointers are re-arranged. If only page merge is required then the pointers from the two directory entries point to the same page. Again, here also, similar to the insertion operation, the locks on the directory are removed first. After verifying the page then the pages are unlocked. This ends the deletion operation.

Let us consider an example, where there are two directory entries adjacent to each other pointing to the same bucket. The local depth of this bucket is considered to be one less than the global depth. Let there be two transactions T1 and T2 both wanting to insert a record. The page is already full (to show how split works). The transactions T1 and T2 are indexed onto the directory entries d1 and d2 respectively. The transaction T1 accesses this directory entry d1 and stores in its working area, the value of page status, VB bits and local depth. It locks P and then gets suspended. T2 begins execution, it sees that page P is locked. T2 then stores the number of bits, local depth and page status in its working area.

When T1 is rescheduled, it compares successfully VB bits and local depth of P. It finds that a split has to occur. The page P splits and a new page P1 is obtained. This page P1 is locked by T1. All the keys are rehashed. If it is a worst case, that is if all the keys go to the same bucket, then there is another split. But this case is not considered here. The records which were in P before are redistributed between pages after rehashing. The key is inserted during this process. The local depth is increased by

one. Now the directory is locked in the *ce* (contraction/expansion) mode and P1 is linked to correct directory entry. Then the lock is released on the directory. This completes the execution of transaction T1. T2 is rescheduled and it Locks P. Then it compares the VB bits in its working space with the new local depth. It is different. T2 unlocks page P and invokes search and locks P1, finds it and inserts key into it.

Let us consider deletion from the above structure (modified one). The transactions T3 and T4 are mapped onto two adjacent directories d1 and d2. T3 has to delete the two records that are in the Page P1 and T4 to delete one record from the page P. The transaction T3 access d2 and see that there is no lock on the page P1 and reads into its working space the VB bits, page status and local depth, and it gets suspended. Then transaction T4 comes and writes into its working space the VB bits, page status and local depth. Then the transaction is rescheduled and compares the VB bits and local depth of P1. It locks the page P1 and deletes the two keys in the page. Then there is a chance for a merge operation. It sees whether the parent page is locked or not. If it is not locked then it merges with P. Then it locks the directory in ce mode, Updates all the pointers and unlocks the pages. But in this case merge would not have been possible if page P was locked by the transaction T4. This would have resulted in an empty bucket. This case was not considered in Kumar's algorithm. But if there was no lock, the pages get merged. When the transaction T4 was rescheduled, it compares the VB bits with the local depth which changes due to a merge operation. Then T4 invokes search and locks P and deletes the required record (key).

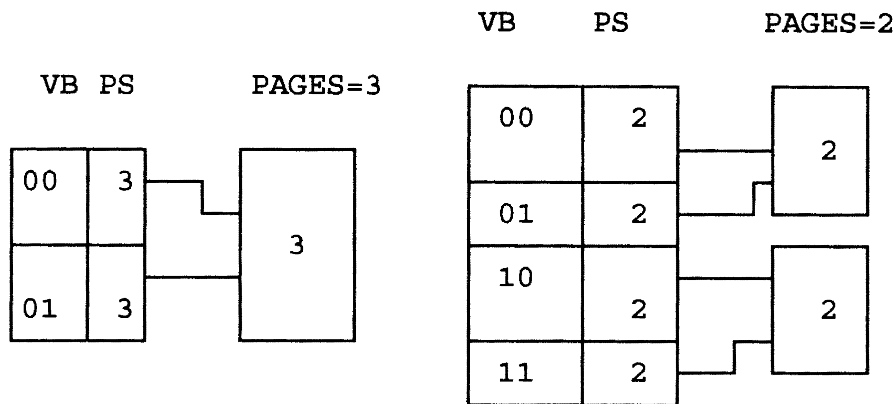

Figure 8. Structure Before and After Transaction

Summary

It was explained how the three algorithms on concurrent operations in extendible hashing work. The solution proposed by Ellis has one advantage that it has no upper bound on the directory. The main disadvantage is that when a split occurs then the buckets expand thereby increasing the links. Thus, this becomes a linked list approach. Ellis uses locks in insert, delete and search operations. By using locks extensively, the locking overhead is increased. The lockhead takes one atomic action. This increases the cost of serialization of locking processes. This is a single step transaction. This means that no two insertions or two deletions can take place simultaneously. The benefit which we seek to achieve from a dynamic data structure like extendible hashing is lost and this more or less becomes a two-phase locking system in which the directory and page are locked during any operation. Though her solution can be applied in distributed data bases the other solutions have greater merit as they give higher degree of concurrency [8].

The solution given by Hsu and Yang also has some advantages and disadvantages. They do not use the linked list approach. Their solution paves a way for the verification process. It includes a field for the verification. This generally helps in avoiding locks all the time. Moreover, this gives better concurrency than Ellis's algorithm. By using this verification process we can insert a key while search or delete operation is going on. This algorithm is deadlock free. This algorithm is also a single step transaction. This locks one data page at a time. No two simultaneous insertion or deletion operations are possible. It has some disadvantages. This algorithm has an upper limit on the directory size. They have made an assumption that underflow and compaction would not occur. They have not taken into account the problem of merging. This solution was the first of its kind which presented a higher degree of concurrency during operations. It is also the root of the next algorithm which Kumar developed.

Kumar's algorithm achieves optimal memory utilization by supporting directory expansion, contraction, page split and merge efficiently. The solution given by Kumar [3] uses verification at two stages both at the directory level and the page level to guarantee data consistency and to avoid locking of directory to a large extent. Thus at any time only the particular entry is locked. This reduces locking overheads and increases concurrency by allowing page modification and directory modification to happen concurrently. It was assumed in this algorithm that if the sibling page is blocked by another processes during a merge operation, merging is not taking place, but the execution is continuing. Kumar [3] states that usually in database operations there would be more page splits than page merging. Merging of buckets was not given much of

importance. Though this algorithm works in the present environment, it can be made more useful and helpful to even small data bases. This can be taken care by merging the buckets at the right time.

With the help of concurrent operations in extendible hashing insertion, deletion and search operations can be performed simultaneously on a particular entry or on different entries. There by creating a very useful multi user environment system. The algorithms which were explained are for synchronizing concurrent operations in extendible hash files. These are very useful for Main Memory Data Bases. It was seen how locking overhead could be minimized by using verification processes. It was also seen how operations such as search insert and delete are performed concurrently, without using locks. These algorithms are deadlock free. Kumar has developed a simulation model [3] which proved his algorithm to provide a higher level of concurrency. Though Kumar's algorithm is deadlock free and offers a higher level of concurrency, modifications are necessary.

In a real-time data application we can know the number of transactions wanting a particular key or entry. With the help of this information we can avoid unnecessary splits or merges. By saving a merge or a split we increase the concurrency. During a merge or a split we lock the directory. We can also decrease the overhead like lockhead and system resources. It will be interesting to see how space complexity versus time complexity would behave. This is where my thesis research would take place. There can be a trade off between space and time. In the next chapter we propose a new merging technique for concurrent operations in extendible hashing. We also give the

algorithms for search insert and delete operations.

CHAPTER III

A NEW MERGING TECHNIQUE FOR CONCURRENT OPERATIONS IN EXTENDIBLE HASHING

Modification to the Directory

Some modifications to the directory are necessary for implementation of our algorithm. The modified directory is shown in figure 9. Each Directory along with the original fields has two extra fields, *check lock* (CL) and *wait status* (WS). Therefore we have totally four fields together, *verification bits* (VB), *page status* (PS), *check lock* (CL) and *wait status* (WS). The *verification bits* (VB) of a directory entry are the most significant bits of the pseudo key. This is same as the directory entry prefix [3]. The *page status* (PS) field of directory tells us the number of pages each corresponding page has. The *check lock* tells us whether there is any lock on the corresponding page. The *wait status* tells us the number of transactions waiting for that particular entry. The number of bits change whenever there is a modification in the directory entry. The *page status* changes whenever there is a change in the number of records in the page. The *check lock* depends upon the lock changes on the corresponding data page. The *wait status* depends on the number of transactions waiting for that particular directory entry.

The *wait status* (WS) can help in avoiding page splitting which may be

unnecessary. If more deletions were to occur in the near future after an insertion, we use a new idea called *forced insertion*. This type of insertion would increase the page size, but it would not split the page. If there were deletions to occur shortly, then it would get the page to the normal size. By doing this we are able to save lock head, memory and other system resources which would have gone waste.

Let us consider a case which occurs in a concurrent operation extendible hash file. The case is shown in Figure 9. Let there be four transactions T1, T2, T3 and T4 which want to perform some operation such as insert, delete or search on two particular buckets (data page) pointed by directory entries d3 and d4 respectively. Let us assume that transactions T1, T3 and T4 want to perform insertions on the bucket pointed by directory entry d3, and T2 wants to perform deletion on the bucket pointed by directory entry d3, and read one record in the bucket pointed by directory entry d4. Before performing any transactions on the hash file, let us see the hash file closely. We see that the number of records in the buckets pointed by directory entries d3 and d4 when combined together is less than the total page (bucket) capacity (page capacity in this example has been fixed at four). There is a potential for merge. When algorithms proposed previously are applied, the buckets pointed by d3 and d4 will be merged. Then immediately there will be a page split, because we have insertions taking place into the bucket pointed by directory entry d3. We see that when the previous algorithms are applied we are unnecessarily merging and splitting. The amount of time we will be losing during a split or merge is more, because the keys have to be rehashed and then redistributed. But with the help of the *wait status* we can tell whether it will be good to merge or not.

In the same situation if the proposed algorithm is applied, We see that *wait status* plus the *page status* of the page and the sibling page cross the page capacity. We would not merge, even though there was a potential for merge. Instead of merging we allow insertions to take place, there by effectively saving a page merge and a page split. During a page merge or a page split we have to rehash all the keys in the effected page and redistribute within the pages. We save a lot of overhead like the lock head and system resources, by not doing the page merge and a split. We increase the concurrency by not locking the page. During a page merge or a split we lock the directory.

In the same example the buckets pointed by directory entries d1 and d2 can be merged but when the previous algorithms were applied merging is not done, because we have a lock on the sibling page. But when the proposed algorithm is applied, with the help of our wait status we see it could be merged, since there would be no split in the near future. So after lock removal we merge the page. We save a lot of space which would have gone waste. Thus we can say the proposed algorithm saves both time and space which are very valuable. This is an innovative and revolutionary concept in concurrency control in extendible hashing.

A conflict between two operations over the directory or a data page is resolved either by rolling back or blocking one of the conflicting transactions. Previously a transaction had to wait till it gets to the data page, to check whether a roll back or blocking requestor should be called but in the proposed algorithm the roll back or blocking requestor call is decided whenever it indexes the directory entry, there by saving time. A rollback or a blocking requestor is given top priority in implementation.

This is based on a two phase locking system [3]. In the next section, the algorithm on concurrent operations in extendible hashing is explained.

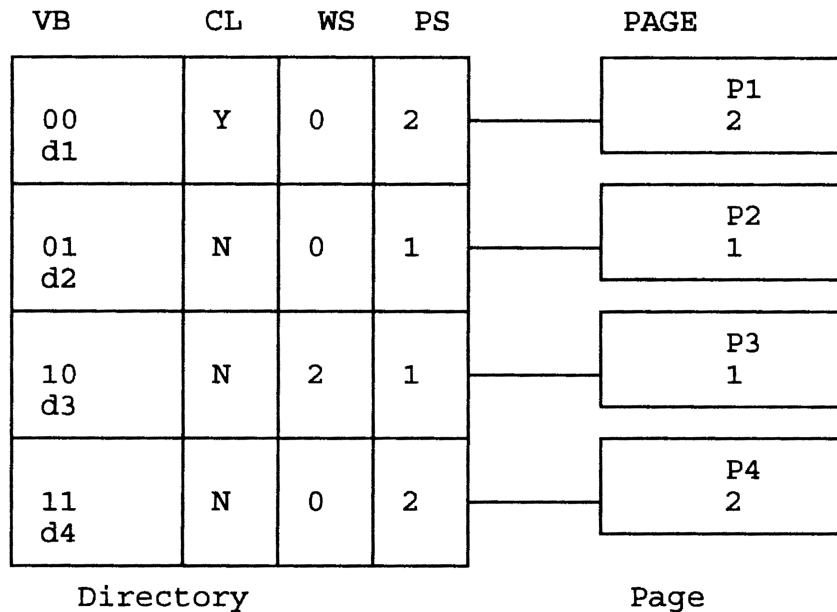

Figure 9. Modified data structure with data pages.

"A directory search is non-atomic, i.e., a search operation is not effected by any other transaction nor the directory is locked. This has been made possible by the use of verification process. The verification process is applied at the directory as well as the data page level. The directory level verification obtains the correct directory entry pointing to the desired page. The page level verification gets the correct page containing the desired record. A search may proceed concurrently with the directory expansion or contraction. Accessing and processing of the data items are done under write(exclusive) and read (shared) locks. Concurrent directory expansion or contraction is serialized with

the help of a verification scheme, a new lock mode called *contraction/expansion*(ce), and by rolling back or blocking conflicting transactions" [3].

The algorithm is composed of three phases: search, insert and delete. The search phase comes in both insert and delete. Search uses the verification technique.

Search Algorithm

The search operation on an extendible hash file consists of:

1. Send the key K to the hash function and obtain the pseudo key K'.
2. Extract the most significant bits of the key to determine the directory to be read.
3. Index the directory and read the values of VB, PS, CL and the page address stored in the directory entry.
4. Compare VB bits with the index bits (that of the key to be searched). This is a directory level verification. If the number of bits differ it means that there had been a directory modification and then the search operation should repeat from step 2. This is repeated until two consecutive readings remain same. A successful comparison means correct directory verification.
5. The Verification bits VB is compared with that of the local depth (page). If there is a difference in the number of bits then there is a change in page. So, we go to step 2. If there was no change in the bits, then it is the correct data page.
6. This procedure returns the address of the page, if there is no lock or a read lock on the page.
7. End of Search.

What the search operation is vulnerable to is the concurrent insert operation that may split a data page and reallocate the keys in a different data page. However, this type of interference is avoided by verification process.

Insertion Algorithm

The insertion operation in an extendible hash file consists of:

1. Apply the search algorithm, pass the page address and local depth to the calling routine.
2. Check whether the *check lock* is active.

   If check lock is active, then it may be one of the following cases:

   (a) There is a transaction going on which is updating one of the records in the data page.

   (b) There is a transaction which is inserting a new record into that page at that time.

The above two conditions can be known at the moment we see the check lock. Here we update the *wait status* and use the rollback or blocking policy.

3. If *check lock* is not active, lock the page in exclusive mode, update the check lock status. The current local depth value is compared with the local depth of that of the search. An unsuccessful comparison indicates interference from other transactions; hence release the page and go to step 1.

   A successful comparison indicates that the page is correct page.
4. Check the *page status*(PS) value. If the page is full, check the wait status if the wait status shows a negative number prepare for a forced insertion, if *wait status* (W) is

positive prepare for a page split, else update the page status.

5. *Forced Insert*: Check the wait status, if there are more number of deletions to occur, then insert the existing record into the page. Update the *page status*.
6. *Page split*: Compare the local depth of the page and the global depth. If they are equal then the page will initiate a directory expansion. Split the page. The new allocated page is also locked in the memory. The keys are rehashed into the two pages. The new record is inserted into the right page. The pointers are updated. The *page status* for both the pages are updated.
7. If the directory was locked, then a conflict would have arisen. Then the conflict is resolved by rolling back or blocking the requestor.
8. If the directory was not locked, then lock the directory in the *ce* mode. If an expansion is required then expand the directory. Update the directory entries, the *page status, check lock, wait status* and the *verification bits*. Release the lock on the directory but not from the pages. At the end of transaction i.e., at commit time we release the locks on the data page.
9. End of insert.

Deletion Algorithm

The Deletion operation in an extendible hash file consists of:

1. Apply the search algorithm, pass the page address and local depth to the calling routine.
2. Check whether the *check lock* is active.

If *check lock* is active, then there might be two cases

(a) There is a transaction going on which is updating one of the records in the data page.

(b) There is a transaction which is inserting/deleting a new record into that page at that time.

Here we update the *wait status* and call the roll back or blocking policy.

3. Lock the page in exclusive mode and compare its current local depth with the local depth value received from the search. An unsuccessful comparison indicates interference from other transactions hence release and go to step 1. A successful comparison indicates that page is the correct page so we delete.

4. Add the PS and WS values of this and its sibling (page that can be merged with this page). If this value is less than the page capacity then prepare for a page merge. A page can be merged only if the sibling page is not locked. This is checked by seeing the check lock on the sibling page.

5. If there was a potential for merge and it could not happen, because of the lock on the sibling page. Check the *wait status* on the sibling page. If the wait status plus the page status of the page and the sibling page put together is less than the total page capacity then we call the timer. (This function is called when the lock on the sibling page is off, it tries to merge with the parent immediately and then goes onto the next step)

6. If the directory is locked then a conflict occurs. Resolve the conflict by rolling back or blocking the requestor.

7. If the directory is free the directory is locked in ce mode. If a contraction is required then contract the directory and link the right page with the directory entry. Update the PS, WS, CL and VB fields. If no directory contraction occurs then link the two directory pointers to this page. Release lock from the directory, after updating all the fields but not from the data pages.
8. If there is an empty page return it to the memory, by releasing the lock on it.
9. At the end of transaction unlock the pages.
10. End of delete.

CHAPTER IV

PROOF OF CORRECTNESS

We follow an approach similar to that taken in [2] to establish the correctness of our algorithm.

This algorithm has used an innovative merging technique which is first of its kind. It uses two fields the *check lock* (CL) and the *wait status* (WS). We call our algorithm *Optimistic Extendible Hashing Algorithm.*

The following assumptions are made.

1. There exists an upper bound for global depth, and the page.
2. The search operation consists of a sequence of read steps. Each read step involves a directory entry or a data page.
3. The insert/delete operations consist of a set of read and write steps that are atomic.
4. An interleave definition is given [3].

Proof of termination

<u>Lemma 1. All operations terminate</u>

Proof: Since no operation would hold any lock while waiting for another, no circular wait is possible, therefore no deadlock is possible. Therefore the termination proof

amounts to proving that the potential loop in the termination will terminate. Since all operations use the verification technique the loop will terminate whenever the value it has read previously remains the same for the next time (Two consecutive times). Thereby we end in finding the correct page. We check this by page level verification. In case of a failure, it will terminate in finite time, since there is an upper limit to the global depth and the number of pages pointed to by the largest directory.

Correctness of Search Algorithm

Lemma 2. The Search Operation is correct

Proof: To prove that search operations are correct, we investigate what could possibly be the cause for it to be incorrect. Since all the search operations terminate, they either succeed or fail. We consider each of these two cases separately.

If a search operation S succeeds i.e., if it finds the key it is looking for, then it must be correct. This can be shown as follows. Let us suppose that we have three operations insert search and delete one particular record. A serial schedule would be: <I(r) S(r) D(r)>. For a successful search a correct serializable schedule is the one where S(r) must complete before D(r). This tells us there must exist an insertion operation that inserts r. The only way search can find r after it has been deleted by D(r) is:

Let us assume a transaction T wants to insert a record in page p. The insertion has resulted into splitting the page p into p and p'. The record has migrated into p'. If there was a search operation going on at the same time, it finds page p as the one which

has the record r since page p was in a transient state holding r. After completion of the insertion operation we have a directory entry corresponding to r is pointing to p'. The search operation accesses the page p and finds r, and returns to the corresponding operation. The following are the steps which represent the above case.

1. r is inserted into data page by p by I(pr).
2. Page p was split to p and p' and r was migrated to p'.
3. p is in a transient state and still holding r.
4. Directory entry corresponding to r is pointing to p'.
5. Search accesses p and finds r.

A serializable schedule is the one when all steps of an interleaved operation preserve their order as they appear in the serial schedule. In the case explained above the last step i.e 5 cannot possibly occur since I(pr) has already changed the directory pointer to p' that contains r. S will therefore, access p' and not p. Moreover, any operation would check the check lock status, and if there was any lock on the data page, then the search or insert or delete operation would roll back. The operation gets repeated from beginning. This implies that there is no such insert operation after which S will be looking into a wrong data page and fail to find r. S is therefore correct since it finds the record r which is present in the file.

Search fails: If search fails then r cannot be in the file, or when there is a concurrent reallocation. In other words, it will be not the case that search looks into a wrong page and terminates with failure when the record r is in another page. We assume that search

fails after the record r was inserted by I(r) in another page. We can see that there are few possible interleaved schedules which can lead to that type of situation, but we show that these do not appear in our algorithm.

1. Insertion of a record into a page, then searching the directory entry and the page. This can be represented in an interleave schedule as <I(pr) S(d) S(pr)>. If the insertion has split the page p into p and p1 and moved the record into p1. If S(pr) fails then S was looking for r in a wrong page. i.e., p. But this cannot happen since if the record was moved into p1, then the directory entry would be pointing to p1 and not to p. But if the insertion expands the directory then the directory verification will take S(d) to the correct directory entry and lead S(pr) to the correct page.
2. Searching the directory entry, then insertion and searching the page for the record. In this case the interleaved schedule can be represented as <S(d) I(r) S(pr)>. If the search fails as we have assumed, then S will be looking for r in page p. This cannot happen since the verification process will eventually find the correct directory pointer.

By combining both proof of success and failure of the search algorithm we conclude that search operation is correct.

Correctness of Insertion Algorithm

Since search operations do not update the database, they do not effect the correctness of an insertion operation. Therefore, to prove that insertion operations are correct we only need to take into account the interferences among insertion operations

themselves, and between insertion and deletion. So inorder to prove our above point, we follow a similar approach to that taken in [3]. Let us establish an insert operation that inserts a record in a correct page and also that two interleaved inserts produces a correct result. In a sense insertion is correct. There are three cases which occur in insertion.

Case I. When WS is non-negative integer, and page is full.

Case II. When WS is negative and PS less than page capacity.

Case III. When it is a simple insertion, PS is less than page capacity.

Case II and Case III are trivial cases, they involve direct insertions in to the page. These cases follow the steps of the insertion algorithm. The formal proof is included after lemma 3. Case I is proved by proof by contradiction in lemma 3.

Lemma 3. Any two concurrent insertion or insertion/update operations T1 and T2 always interleave correctly

Proof: Let there be two transactions T1 and T2 which want to insert I(r) and update U(r) the same record respectively. If they are to be inserted and updated correctly they would follow this interleaved schedule. First there would be a directory search and then there would be a page search to insert the record. This can be represented in an interleave schedule as <S(d) S(r) V(d) I(p1r) S1(d) S1(r1) V1(d) U(p1r)>. Let us say that the insertion of record r into page p has split into p and p1. And the record goes into p1 page. As soon as insert operation is finished, all the locks on the page are removed, and the second operation update is started. The second operation also behaves

the same way, as the first.

Suppose we say that the two interleaves behaved incorrectly and they have got wrong pages for insertion and update, this could have been possible for the following reason. If S and S1 have found the same directory entry pointing to the data page p. Both the verification process would succeed. The first insertion I would split the page p into p and p1, and redistribute the keys, and the record is inserted into p1. For we say that U(r) has got a record which is not the one I inserted into p or p1, which is in the wrong page. The interleaved schedule for this can be shown as follows, <S(d) S(r) S1(d) V(d) S1(r) V1(d) I(P1r) U(pr)>, note the difference. It should have worked as the original interleave schedule and must have updated the record in p or p1. But we show this interleave (Second one) is not possible since, I(r) will be executed under 2PL policy.

The page split will be performed under exclusive locks, so it is not possible to perform another operation on the same page. And after first operation, insertion is complete then the second operation update, which was blocked previously as there was a lock on the page p would start. The update operation begins with a directory verification and a page level verification. If the directory verification succeeds then there would be a page level verification. If this also succeeds then we update the record in the page. If the directory level verification has failed then it would restart the whole process i.e., from search. By this we can conclude that the interleaving between I and U is equivalent to serializing I and U therefore they are correct. But we say we have another transaction R which wants to read the record r. This has to wait since the page

is under lock and no other transaction can accesses it. Soon after releasing the lock the transaction R is rescheduled. Let us now examine case II where there is no split in the page, i.e., where it uses the forced insertion.

Let us say there are four transactions T1, T2, T3 and T4 which want to insert, delete, delete and update respectively, some records in the same page. The interleaving schedule for this will be <S1(d) S1(r) V1(d) I1(pr) S2(d) S2(r) V2(d) D2(pr) S3(d) S3(r) V3(d) D3(pr) S4(d) S4(r) V4(d) U4(pr)>. Let us say that insertion of a record would split the page, but before we split the page we see the wait status, which tells us that there are two deletion operations to be done. We force insert the record into the page. As soon as this insert is finished, the lock on the page is removed. We are ready for the next transaction. The next transaction which was blocked previously due to lock on the page, would resume with a directory and page level verification. Since there was no directory or page split then the verification remains same and the second operation locks the page. Since these transactions are executed under the two phase lock policy, it is not possible for other operations to be performed on the same page at the same time. An interleaving schedule is equivalent to serializing. The remaining transactions also behave the same way.

By this lemma we can conclude that insertion algorithm is correct.

Correctness of Deletion Algorithm

In deletion algorithm as in insertion algorithm we have three cases to be taken into account.

Case I. When WS + PS is less than the page capacity and when lock is present.

Case II. When WS + PS is less than the page capacity but when lock is present.

Case III. When WS + PS is greater than the page capacity.

Case I and Case II formal proof is included in lemma 4. The formal proof for Case III is simple, because it is just an ordinary deletion and it follows the step of the deletion algorithm.

Lemma 4. Any two concurrent deletions or delete/insert operations always interleave correctly

proof: Let I and D be two operations which want to insert and delete respectively. If they were to be inserted and then deleted they had to follow the following serial interleave schedule. <S(d) S(r) V(d) I(p1r) S1(r) V1(d) D(p1r)>. First there would be a directory search and S(r) finds the page where the record r should be. Directory verification V(d) is done before I(pr) splits the page into p and p1. The records are distributed into p and p1. When the insertion is complete then all locks are released, and the deletion from the second transaction begins.

Suppose these two transactions interleave incorrectly and insert and delete their records incorrectly. This is possible if one of their interleaved schedules could be <S1(d) S1(r) S2(d) V1(d) S1(r) V2(d) I(p1r) D(pr)> . Under this schedule S1 and S2 find the same directory entry pointing to the same data page p. Both the verifications V1(d) and V2(d) will succeed. If I(p1r) splits the page p into p and p1 then redistributes the keys, and inserts r into p1. By the same time the verification for the second operation is correct so it deletes a record which is from a wrong data page. This

interleave schedule is impossible since I(r) will be executed under 2PL policy. The page split is performed under exclusive locks. The second operation deletion would be blocked because there is a lock on the page. When the insertion operation is done and all the locks on the pages have been removed. The deletion transaction is started, then it does a directory verification if it succeeds then it does a page level verification, it finds the record and deletes it. Here we see that transaction deletion cannot delete the record from the page p1, because there was a lock being held by the insertion transaction, and therefore it had to wait until I released the lock on it. If there were to be a potential for merge, then it would check the wait status along with page status of the page and sibling page and a merge is performed. The consistency of the algorithm at any stage wouldn't change because of wait status. The wait status checks whether it would be advisable for a merge. If there was a lock on the sibling page and still there is a potential for merge, immediately after removing the lock, the merge is performed.

Thereby we can conclude from the above four lemmas that the proposed algorithm for concurrent search/insertion/deletion operations are correct.

In the next section we see the algorithms of our Search Insert and Delete operation algorithms.

Search Algorithm

Input : Given Key

Output : Page address

```
Search (given key k);
begin
  Initialization:
  Xold:=0, dnew:= 0;
 hash function(K);
  k' = p0 p1 p2 .....pn-1;
 getpointer;
  read d, base;
 check the global and local depth;
 t:= p0 p1 p2 .....pn-1  /* take the initial g bits */
 take the directory index
 new is compared with the xold; /* re verification so as to confirm it is correct */
   while dnew < >xold do
    begin
      comparison and mapping on to the directory entry
       If CL == 1;
        break;
       roll back or block;
     end
    return(directory entry contents);
end
```

Insertion Algorithm

Input : The given key

Output : Whether Successful or not

```
begin
  / * send the key to the search function */
  tk: = search( key k);
  if result then
  begin
     read( directory .vb, directory .cl, direntry.old_data_page);
     free:= checkstatus
    if free(1)
     A: = get(direntry->page) /* read the data page pointed by directory entry */
     if A: == k /* if the key is already present */
      return;
 else
     /* prepare for insertion */
   case.1 |A| < ps /* no need to split */
               A: = insert(A,k);
   case.2 |A| >= ps && ws < 0 /* no need to split forced insert */
               A: = forced_insert(A,k);
   case.3 |A| = ps && ws >= 0 /* split the page */
                 rp: = this is the number of required pages.
               p1,p2,p3,p4,p5......pn-1: = allocate p new pages
              lock(p1,p2,p3,p4,p5......pn-1) /* the new pages are locked*/
              A,A1,A2....An:= rearrange old A and new directory pointers
              writeback(A,A1...An->P or P1);
      directory.modify(D,p1,p2,p3,p4,p5......pn-1);
     begin
              update all the directory pointers.
              unlock(p1,p2,p3,p4,p5......pn-1);
              update check lock status;
     end

end
```

The function of the directory modify is

```
Procedure directory.modify(D,p1,p2,p3,p4,p5......pn-1);
begin
  for all directory entries j affected by split do;
    i: = subscript of newly allocated page 0
    put(pi,d[j]);
  end
end
```

Deletion Algorithm

Input : Send the Given Key

Output: Whether Deleted or not

```
begin
  / * send the key to the search function */
 tk: = search( key k);
    if result then
      begin
             read( directory .vb, directory .cl, direntry.olddatapage);
             free:= checkstatus
             if free(1)
             A: = get(direntry->page) /* read the data page pointed by directory
entry*/
             if A: != k /* if the key is already Deleted */
             return;
      else
             /* prepare for Deletion */
             case.1 |A| < (ps + ps of sibling page)+ ws /* no need to merge */
                    A: = Delete(A,k);
             case.2 |A| >= (ps + ws ) page and sibling page && no lock is present
                    A: = Delete(A,k);
                          Merge(A);
             case.3 |A| >= (ps + ws ) page and sibling page && lock is present  /*
                                      Merging can be done after lock removal */
                     A:= Delete(A,K);
                    Merge_lock();
    end

end
```

CHAPTER V

PERFORMANCE ANALYSIS

In this chapter we present few examples to show concurrent execution of transactions. We use two sets (set1 and set2) of transactions and run them concurrently using our algorithm. Figure 10 gives the initial state of the file. Here we assume that each data page can hold up to a maximum of three records (for simplicity).

TABLE III

TRANSACTION SET

SET	T1	T2	T3	T4	T5
1	1,4	1,4	17,2	4,1	7,10
2	12,10	17,10 18	9,2	11,4	19,7

In Table III, we have two sets each consisting of five transactions, and each transaction requires one or more records. We investigate the different possible cases

which would occur in a concurrent operation on extendible hash file. And throughout the explanation of cases we have made some assumptions. For example, if we come across numbers like 3, 6, 19 etc., then those are the record (key) numbers. If we come across 00 11 10 etc., it is the directory entry prefix.

Case 1: No Page Split, No Directory Modification, No Forced Insert.

Here we would be seeing an elementary case, where there is no page, directory split or merge. We have our initial state represented in Figure 10. We execute concurrently the transactions of set 1. T1 and T2 wish to modify records 1 and 4. T1 is scheduled and the first two bits of the pseudo key index the directory entry 00. It checks whether there is any lock on the page by seeing the status of the check lock. If it finds there is no lock on it. It accesses and stores in its working area the page status, VB bits, wait status, check lock and the local depth, locks p1, and then gets suspended. T2 begins execution, and the directory entry 01 is accessed. T2 is blocked over p1, because as soon as it accesses this directory entry, it sees a check lock active, if this transaction wanted to insert/delete a record it would update the wait status. It was assumed that T2 wants to update the record, so it would not update the wait status. When rescheduled T1 does not verify since the data page locked by it cannot be acted by any other transaction. T1 completes its work and commits. When T2 is rescheduled it compares the number of VB bits so as to confirm whether there was any change in the directory or page. If the comparison is successful (it is successful in this case) T2 locks p1, finds the record modifies and commits.

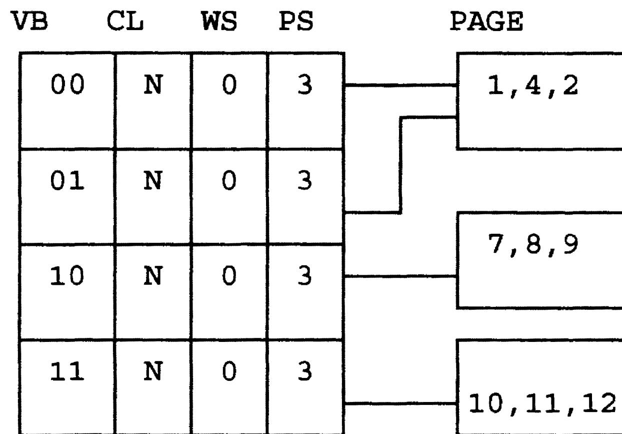

Figure 10. Initial state of the file

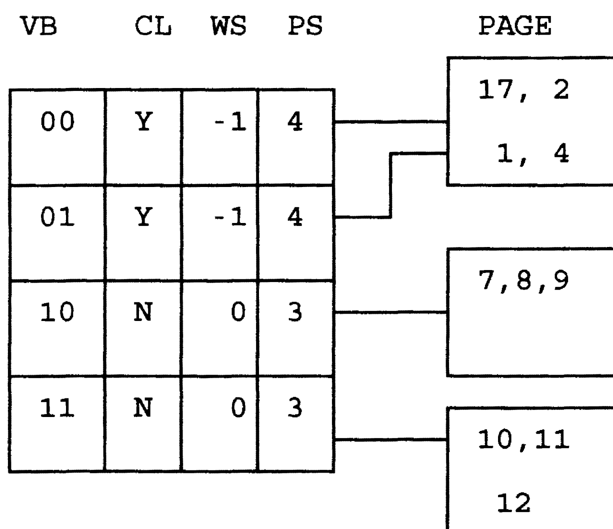

Figure 11. Transition state.

Case 2 : No Page Split, Forced Insertion, No Directory Split.

Consider Figure 10. T3 wants to insert a record 17 and modify 2. T4 wants to modify 4 and delete 1. T5 wants to modify 7 and 10 records which are indexed by (10) and (11)

respectively. T3 is scheduled and gets suspended after reading the VB bits and page status check lock by the entry prefixed by 00. Before getting suspended this locks the page and updates all other fields. When transaction T4 is scheduled it prefixes directory entry 01. T4 checks whether there is any lock on the page. T4 finds a lock on the page, so it updates the wait status telling there is a deletion going to be performed, and writes into its buffer space the directory entry contents, and then gets suspended. T3 is rescheduled, it updates its wait status in its working area, it modifies the record 2 and now tries to insert record 17. T3 finds the page status as four (full), it then checks wait status, which tells T3 that there is going to be a deletion on this page, this is a case for forced insertion. Here we do not split the page which is normally the case, and would have even lead to directory modification. Ordinarily for a page split, directory is locked and the page is linked to the correct directory entry; and ce lock is released from the directory, there by not allowing any other transaction to start. After forced insertion before committing we update all the fields in the directory entry 00 and commit. T4 is rescheduled, it compares the number of VB bits it stored in its working area before it was suspended, with the local depth of the page p1. In this case the comparison would be successful since there was no split in the page or directory. So, T4 locks p1, finds and modifies the record 4 and deletes the record 1. Before committing T4 updates all the entries on the directory and commits. Here we have saved not only a page split, but a re-verification process which T4 would have done if there was a split. Thus we see the advantage of forced insertion. T5 in the meantime would complete its transactions since they are not in conflict with any other transaction. Figure 11 gives the transition

state of forced insertion. Figure 12 gives the state if a split had occurred. In a real time data we would have information before, if a insert or a delete operation would be next.

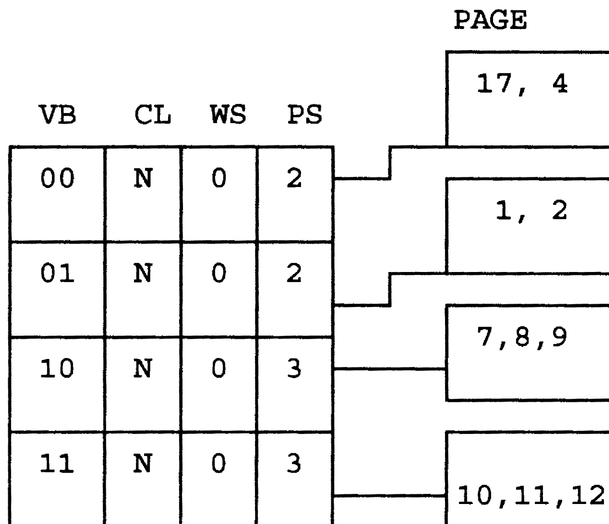

Figure 12. The State after a Split.

Case 3 : Page Split, Transaction Blocking or Roll back, and no Directory Modification. Consider Figure 11 and set 2. T2 wishes to modify records 17(00) and wants to insert record 18(01), and delete 10(11). T3 wishes to modify record 2, which is prefixed by 00, and wants to delete record 9 prefixed by 10. T4 wants to modify 11(11) and 4(00). T2 begins, reads VB bits, WS, PS, CL and locks p1. T3 begins and reads all the values of the data page pointed by directory entry 00 and gets suspended. T4 locks p3 and modifies record 11. It then tries to lock page p1. Page p1 is already locked by T2, and T2 is not blocked by any other processes we block T4. T2 starts and modifies record 17. It then tries to insert record 18 into this. Before inserting we check the wait status to see whether there are going to be any deletions, but in this case we do not have any.

Then we see the page status, it is full then we prepare for a page split. The page is split into pl and pl'. We see that records 2 and 17 go to page pl, and 18 and 4 go to page pl'. Then T2 tries to lock page p3, which was locked by T4 and is in a suspended state. Here in this case we roll back the requestor T2 since T4 is in a blocked state. We thereby make the algorithm deadlock free. T4 gets started and modifies record 4. During a roll back operation T2 removes the lock on the page pl. T4 then successfully completes its operation and releases the lock on page 3 after modifying record 11. Then T2 is scheduled and deletes 10. In the mean time T3 deletes 9, and gets ready to modify 2. In this case we show when we call a requestor and when we block a requestor. This case also gives how a page splits, and Figure 13 shows the final state.

In the next case we see how and when we merge.

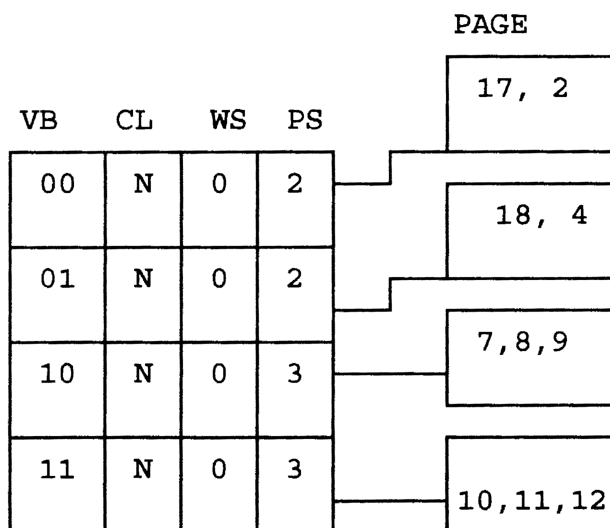

Figure 13. The State after a Split for Set 2.

Case 4 : Page Merge, No Page Split, No Directory Modification, Forced Insertion.

Consider Figure 13. Let us run set2. T1 wants to delete 12(11) and 10(11). T5 wants to insert 19(11) and delete 7(10). T4 wants to delete 11(11) and modify 4(01). T5 when scheduled reads the values of the directory indexed by 10 and 11 respectively. Before getting suspended it locks p3 (data page pointed by directory entry 10) and p4 (data page pointed by directory entry 11) respectively. T1 comes and sees that lock on the directory entry 11 and it updates the wait status and gets blocked. T4 comes to directory entry 11 and sees a lock on the page 4, it updates the wait status, and before getting suspended it puts a lock on the page 2, for modifying record 2. T5 is rescheduled. T5 compares it wait status in its working area and the new wait status. There is no verification of bits, since the page was locked no other transaction can act on the page. It sees there are going to be three deletions in near future. It checks the page status, but it is full. This is a case for forced insert. It force inserts the record 19(11) in to the page 4. It also completes deletion of record 7(10) before committing. T1 which was blocked previously is scheduled. It sees the directory entry 11 is lock free, before putting a lock on page 4, it writes all the contents into its buffer, and gets suspended. T1 is rescheduled and it performs its operation and releases the lock just before committing. T4 is scheduled and sees there is no lock. T4 deletes record 11 and sees whether there is any possibility of merging with its sibling page. We see it could be merged because p3 has two records and p4 one. This is a potential for merge, and we merge, because there is no lock on the sibling page. The resulting state is shown in Figure 14. We saved in this case one merge and one split. We have also not locked the directory, by saving merge and a split.

The proposed algorithm at the worst case will behave as the previous algorithms, but in an average case or a best case this algorithm saves time and memory. This algorithm avoids unnecessary thrashing, by splitting or merging whenever it is safe.

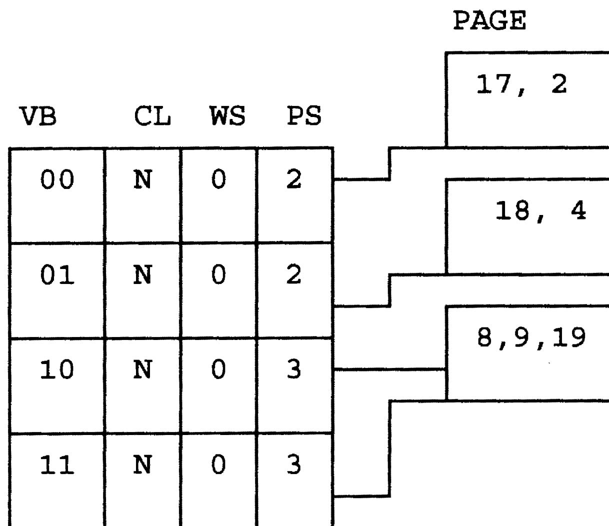

Figure 14. The State after a Merge for Set 2.

CHAPTER VI

CONCLUSIONS

In this thesis, we present an algorithm on concurrent operations in extendible hashing which shows a higher level of concurrency than the traditional one. Locking of directory is eliminated to a large extent in cases, which occur very frequently [1]. We have also seen a new idea called forced insertion at the right moment during the execution of concurrent operations to guarantee there is no unnecessary lock on the directory. This thesis also introduces two new fields Check Lock and Wait Status. The algorithm supports directory expansion and contraction, Forced insertion, optimistic merging and splitting. This algorithm does not split a page or merge whenever there is a potential, but does only when required. This revolutionary concept would make extendible hashing a more favorite one for databases.

Our algorithm manages to reduce locking overheads and increases concurrency by allowing page modification and directory contraction/expansion to proceed concurrently. Our algorithm has optimal memory utilization since it tries to split or merge, when there is guarantee that at least in the near future there would be deletions or insertions. The extra overheads generated by directory management is not significant and does not offset the advantages gained by improved concurrency. We believe that this algorithm would be an efficient one, for both small and large databases. We make

REFERENCES

[1] Ellis, C.S. "Extendible hashing for concurrent operations and distributed data". In Proceedings of the Second ACM SIGACT - SIGMOD Symposium on principles of Database Systems (Atlanta, Ga., March 21 -23). ACM, New York, 1983 , 106-119.

[2] Hsu, M., and Yang, W.P. "Concurrent Operations in extendible hashing". In Proceedings of the 12th International Conference on Very Large Databases. IEEE, New Jersey, 1986.

[3] Kumar, V. "Concurrent operations on extendible hashing and its performance". Communications ACM 33 (June 1990), 681-694.

[4] Fagin, R., Nievergent, J., and Strong, H.R., "Extendible hashing: A fast access for dynamic files". ACM Transactions on database systems, 4, 3(Sept. 1979), 315-344.

[5] Bayer R. and Schkolnick M. , "Concurrency of Operations on B-trees". Acta Informatica 9 (1977), 1-21.

[6] Vomer, D., "The Ubiquitous B-tree". Computing Surveys 11 (1979), 121-137.

[7] Sagiv, Y., "Concurrent Operations on B-trees with Overtaking". Fourth Annual ACM SIGACT/SIGMOD Symp. on Principles of Database Systems (Portland, Or.), ACM, New York (1985), 28-37.

[8] Shasha, D. and Goodman, N., "Concurrent Search Structure Algorithms". ACM Trans. on Database Systems 13 (1988), 53-90.

[9] Ellis, C. S., "Extendible Hashing for Concurrent Operations and Distributed Data". TR110, Computer Science Department, University of Rochester, October 1982.

[10] Kung, H. T. and Robinson J. T., "Optimistic Methods for Concurrency Controls". ACM Transactions on Database Systems, 6, 2, June 1991

[11] Kedein, Z. and Silberschatz, A., "A Locking Protocols: From Exclusive to

Shared Locks". Journal of ACM, 30, 4, October 1983.

[12] Papadimitriou, C. H., "The Serializability of Concurrent Database Updates". Journal of ACM 26, 4, October 1979.

[13] Kwong, Y. S. and Wood, D., "New Method for Concurrency in B-trees". IEEE Transactions on Software Engineering, Vol. SE-8, No.3, May 1982.

[14] Litwin, W. "Linear Hashing: A New Tool for File and Table Addressing". In Proceedings of the Sixth International Conference on Very Large Databases (Montreal, Canada, Oct. 1-3). IEEE, New Jersey, 1980, 212-223.

[15] Larson, P. A. "Dynamic Hashing". BIT, 18, 2(1978), 184-201.

[16] Shasha, D. and Johnson, T., "The performance of Current B-tree Algorithms". ACM Trans. on Database Systems 18, 1 (1993), 51-101.

[17] Kumar, A. and Segev, A. "Cost and Availability Trade offs in Replicated Data Concurrency Control". ACM Transactions on Database Systems 18, 1 (1993), 102-131.

[18] Agarwal, D and Sengupta, S. "Modular Synchronization in Distributed, MultiVersion Databases: Version Control and Concurrency Control". IEEE Transactions on Knowledge and Data Engineering, 5, 1 (1993), 126-137.

VITA

SIVALENKA VENKAT PRASAD

Candidate for the Degree of

Master of Science

Thesis: A STUDY ON CONCURRENT OPERATIONS IN EXTENDIBLE HASHING INVOLVING MERGING

Major Field: Computer Science

Biographical:

Personal Data: Born in Republic of India, December 11th 1969, son of Mr. S.A.V. Sastry and Mrs. Leela.

Education: Graduated From Osmania University, with a Bachelor's in Mechanical Engineering in August 1991. Completed Requirements for the Master of Science Degree at Oklahoma State University in November 1993.