GEORGIA INSTITUTE OF TECHNOLOGY — OFFICE OF CONTRACT ADMINISTRATION

PROJECT ADMINISTRATION DATA SHEET

[X] ORIGINAL [ ] REVISION NO. ______

Project No./(Center No.) G-36-668 (R6238-0A0) GTRC/GIT DATE 11 / 18 / 86

Project Director: E. Omiecinski School/Lab ICS

Sponsor: NSF

Agreement No.: IST-8696157

Award Period: From 9/1/86 To ~~1/31~~ 2/28/88 (Performance) 4/31/88 Reports

Sponsor Amount:

	New With This Change	Total to Date
Contract Value:	$	$ 27,600
Funded:	$	$ 27,600

Cost Sharing No./(Center No.) G-36-364 (F6238-0A0) Cost Sharing: $ 1,449

Title: Incremental File Reorganization Schemes

ADMINISTRATIVE DATA OCA Contact John Schonk X-4820

1) Sponsor Technical Contact:

Joseph Deken
National Science Foundation
BBS/IST
Washington, D.C.
202/357-9569

2) Sponsor Issuing Office:

Joanna Rom
National Science Foundation
DGC/BBS
Washington, D.C.
202/357-9653

Military Security Classification: ______ ONR Resident Rep. is ACO: ______ Yes ______ No

(or) Company/Industrial Proprietary: ______ Defense Priority Rating: ______

RESTRICTIONS

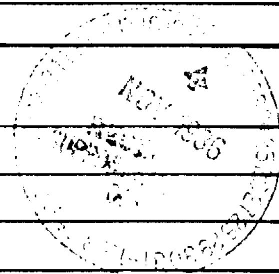
See Attached NSF Supplemental Information Sheet for Additional Requirements.

Travel: Foreign travel must have prior approval — Contact OCA in each case. Domestic travel requires sponsor approval where total will exceed greater of $500 or 125% of approved proposal budget category.

Equipment: Title vests with GIT

COMMENTS:

This grant is transferred from North Dakota State University.



COPIES TO: SPONSOR'S I.D. NO. ______

Project Director
Research Administrative Network
Research Property Management
Accounting
Procurement/GTRI Supply Services
Research Security Services
Contract Support Div.(OCA)(2)
Research Communications
GTRC
Library
Project File
Other

GEORGIA INSTITUTE OF TECHNOLOGY OFFICE OF CONTRACT ADMINISTRATION

SPONSORED PROJECT TERMINATION/CLOSEOUT SHEET

Date 6/10/88

Project No. G-36- 668 School/~~Lab~~ ICS

Includes Subproject No.(s) N/A

Project Director(s) E. Omiecinski GTRC/~~GIT~~

Sponsor National Science Foundation

Title Incremental File Reorganization Schemes

Effective Completion Date: 2/28/88 (Performance) 5/31/88 (Reports)

Grant/Contract Closeout Actions Remaining:

- [ ] None
- [ ] Final Invoice or Copy of Last Invoice Serving as Final
- [ ] Release and Assignment
- [X] Final Report of Inventions and/or Subcontract: Patent and Subcontract Questionnaire sent to Project Director [X]
- [ ] Govt. Property Inventory & Related Certificate
- [ ] Classified Material Certificate
- [ ] Other ____________

Continues Project No. ____________ Continued by Project No. ____________

COPIES TO:

Project Director
Research Administrative Network
Research Property Management
Accounting
Procurement/GTRI Supply Services
Research Security Services
Reports Coordinator (OCA)
Program Administration Division
Contract Support Division

Facilities Management - ERB
Library
GTRC
Project File
Other ____________

NATIONAL SCIENCE FOUNDATION
Washington, D.C. 20550

FINAL PROJECT REPORT
NSF FORM 98A

PLEASE READ INSTRUCTIONS ON REVERSE BEFORE COMPLETING

PART I—PROJECT IDENTIFICATION INFORMATION

1. Institution and Address	2. NSF Program	3. NSF Award Number
Georgia Tech Research Corp Georgia Institute of Technology Atlanta, GA 30332-0420	Information Science & Technology	IST-8696157
	4. Award Period From 9-1-86 To 2-28-88	5. Cumulative Award Amount $43,560

6. Project Title

INCREMENTAL FILE REORGANIZATION SCHEMES

PART II—SUMMARY OF COMPLETED PROJECT *(FOR PUBLIC USE)*

The purpose of this project is to examine the feasibility of performing incremental /concurrent file reorganization. The Typical motivation for any reorganization is to improve the performance of the database system. We examined two classes of concurrent file reorganization: intra-structural and inter-structural. For the first class, we developed an efficient reorganization algorithm based on record clustering. We also adapted our method to the problem of efficiently performing a relational join operation. For the second class of reorganization, we developed simulation models and analytic models for the conversion of B+ tree and linear hash files. The results show that concurrent conversion is a viable approach.

PART III—TECHNICAL INFORMATION *(FOR PROGRAM MANAGEMENT USES)*

1. ITEM *(Check appropriate blocks)*	NONE	ATTACHED	PREVIOUSLY FURNISHED	TO BE FURNISHED SEPARATELY TO PROGRAM: Check (✓)	TO BE FURNISHED SEPARATELY TO PROGRAM: Approx. Date
a. Abstracts of Theses	X				
b. Publication Citations		X			
c. Data on Scientific Collaborators		X			
d. Information on Inventions	X				
e. Technical Description of Project and Results		X			
f. Other *(specify)*					

2. Principal Investigator/Project Director Name *(Typed)*	3. Principal Investigator/Project Director Signature	4. Date
Edward Omiecinski	Edward Omiecinski	5-12-88

NSF Form 98A (1-87) Supersedes All Previous Editions

Form Approved OMB No. 3145-0058

PART IV - SUMMARY DATA ON PROJECT PERSONNEL

NSF Division Information Science & Technology

The data requested below will be used to develop a statistical profile on the personnel supported through NSF grants. The information on this part is solicited under the authority of the National Science Foundation Act of 1950, as amended. All information provided will be treated as confidential and will be safeguarded in accordance with the provisions of the Privacy Act of 1974. NSF requires that a single copy of this part be submitted with each Final Project Report (NSF Form 98A); however, submission of the requested information is not mandatory and is not a precondition of future awards. If you do not wish to submit this information, please check this box ☐

Please enter the numbers of individuals supported under this NSF grant.
Do not enter information for individuals working less than 40 hours in any calendar year.

*U.S. Citizens/ Permanent Visa	PI's/PD's		Post-doctorals		Graduate Students		Under-graduates		Precollege Teachers		Others	
	Male	Fem.	Male	Fem.	Male	Fem.	Male	Fem.	Male	Fem.	Male	Fem.
American Indian or Alaskan Native ...												
Asian or Pacific Islander												
Black, Not of Hispanic Origin												
Hispanic												
White, Not of Hispanic Origin	1					1						
Total U.S. Citizens	1					1						
Non U.S. Citizens					1	1						
Total U.S. & Non-U.S. ..	1				1	2						
Number of individuals who have a handicap that limits a major life activity.												

*Use the category that best describes person's ethnic/racial status. (If more than one category applies, use the one category that most closely reflects the person's recognition in the community.)

AMERICAN INDIAN OR ALASKAN NATIVE: A person having origins in any of the original peoples of North America, and who maintains cultural identification through tribal affiliation or community recognition.

ASIAN OR PACIFIC ISLANDER: A person having origins in any of the original peoples of the Far East, Southeast Asia, the Indian subcontinent, or the Pacific Islands. This area includes, for example, China, India, Japan, Korea, the Philippine Islands and Samoa.

BLACK, NOT OF HISPANIC ORIGIN: A person having origins in any of the black racial groups of Africa.

HISPANIC: A person of Mexican, Puerto Rican, Cuban, Central or South American or other Spanish culture or origin, regardless of race.

WHITE, NOT OF HISPANIC ORIGIN: A person having origins in any of the original peoples of Europe, North Africa or the Middle East.

THIS PART WILL BE PHYSICALLY SEPARATED FROM THE FINAL PROJECT REPORT AND USED AS A COMPUTER SOURCE DOCUMENT. DO NOT DUPLICATE IT ON THE REVERSE OF ANY OTHER PART OF THE FINAL REPORT.

Incremental File Reorganization Schemes
(NSF IST-8696157 Final Project Report)

Edward Omiecinski
School of Information and
Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

1. INTRODUCTION

The purpose of this project is to examine the feasibility of performing incremental/concurrent file reorganization. The typical motivation for any reorganization is to improve the performance of the database system. Incremental reorganization is an on-line strategy where the file is reorganized concurrently with usage at specified times. With this approach, the part of the file which is being reorganized is locked while user access is permitted to the remainder of the file. In any environment where the database system must be available 24 hours per day, typical off-line reorganization would be intolerable. In this work, we develop algorithms for several incremental/concurrent file reorganization problems.

Our research project deals with two categories of physical structure change: intra-structural and inter-structural change. By intra-structural change, we mean that the file structure created by the reorganization process is of the same type as the structure that existed prior to the reorganization, e.g., rebuilding a B+ tree to decrease its height. Inter-structural change refers to the fact that the reorganization process changes the initial file structure to one of a different type, e.g., converting from an indexed file to a hash based file. The next two sections summarize the work that has been accomplished in each of these categories.

Before proceeding, we want to first provide some measures for judging the efficiency of our proposed algorithms. Accessing a file which is stored on a secondary storage device is accomplished by using a single main memory buffer area of a specified size. Our objective as stated earlier is to develop efficient reorganization algorithms. To measure efficiency, we can consider the minimization of the number of pages swapped in and out of the fixed size buffer during the reorganization process. As an alternative, if we can request any size buffer area, we can consider the minimization of the buffer size so that each page involved in the reorganization will be fetched from secondary storage only once. Hence, we achieve the minimum number

of page accesses at the expense of using more main memory. However, having a larger buffer area for some types of reorganization may mean that a larger portion of the file will be locked and unaccessible to users. This gives rise to a third measure for efficiency, namely how the reorganization process interferes with user access, e.g., are fewer transactions processed when reorganizing.

2. INTRA-STRUCTURAL REORGANIZATION

Work in this area, done before the inception of the grant, involved the following:

1) clustering records [4]

2) removal of overflow records [5].

The first problem involves the rearrangement of records, from a file, on pages of secondary storage. The reorganization process uses the output of some record clustering algorithm. The record clustering algorithm determines a near optimal placement of records on pages so as to minimize the total number of page accesses for some set of queries. Knowing the new placement of records to pages, our reorganization algorithm would then rearrange the records to reflect the new clustering. The original approach, presented in [4] was limited since it required a minimum buffer size to work. There were also some additional improvements that could be made so as to further reduce the number of page accesses made during the reorganization. This later work was done as part of the grant and is described in [6]. Overall, our improvements lead to an additional 25% saving in page accesses as compared with the original work in [4].

While working on the clustering reorganization, we realized that it was similar to the problem of scheduling page fetches for a relational join operation when indexes are used. For both problems, a sequence of page fetches must be determined so as to minimize the total number of page accesses, either in computing the join or in per-

forming the reorganization. From the join problem perspective, we can think of a cluster as consisting of a record from one of the joining relations and several records from the other joining relation, where the records that join together (and their addresses) can be easily found by searching the two indexes. Adapting our reorganization approach, we developed an efficient join processing algorithm which uses non-clustered indexes [2]. We developed two variations: one which tried to find the minimal buffer size that would still guarantee a single access per page and the other which tried to minimize the number of page accesses for a fixed size buffer. For the first variation, we compared our method with one that recently appeared in the literature, and our method used about 30% less buffer space. For the second variation, we compared our method with that of the nested-loops and sort-merge join methods. Under our particular assumptions, our method performed the join with about 40% fewer page accesses. The results of this work is presented in [2].

3. INTER-STRUCTURAL REORGANIZATION

This part of the project involves the conversion from one file structure to another. Until the reorganization is complete, part of the file would reside in the original file structure (unreorganized) and part in the new file structure (reorganized). The motivation for doing the conversion is that the old file structure is no longer optimal for answering the current set of queries. Hence, a more approriate file structure is needed.

We decided to start with the conversion of a B+ tree file to that of a linear hash file. The B+ tree is used widely in database systems as the primary index structure. It allows for direct access (retrieval of a record by its key value) with a cost of 3 or 4 page accesses, depending on the height of the tree. It also allows for efficient sequential processing (accessing some or all of the records in key sequence). However, if only direct access is needed, then a hash based file is more efficient with a cost of about 1 page access, on the average. Thus, the conversion process, which we first

examine, is motivated by a change in database processing, where efficient sequential and direct access is originally needed but now, only efficinet direct access is needed.

The type of hash file which we use is a linear hash file which has appeared often in the database literature since its inception in the late seventies. This type of hash file is dynamic (like the B+ tree) in that it grows and shrinks gracefully, one page at a time. One requirement is that we want to reorganize in-place, i.e., using the storage of the original file and perhaps a small additional amount. The reorganization process takes one page at a time from the B+ tree and inserts the records from that page in the linear hash file. As we reorganize a page from the B+ tree file, that page can be added to the linear hash file. During the reorganization, it is also clear which file would have to be accessed to find a particular key. This allows the benefits of the partial linear hash file to be gained immediately.

We have developed a database simulation program which incorporates the reorganization of a B+ tree file to that of a linear hash file [1]. In addition, we have devised an analytic model of the conversion process [1]. The results from the analytic model are within 3% (on the average) of those observed from the simulation. The results of the simulation support the idea of doing file conversion concurrently with database usage. Here, our measure of efficiency is the system throughput. In other words, how does the reorganization affect other concurrently executing transactions. As we show in [1], the effect is not very determental.

The second problem, under this category of reorganization, is the companion problem of doing the file conversion in the reverse direction. That is, converting a linear hash file to a B+ tree file. For this problem, we also develop an efficient algorithm. We utilize our database simulation model in evaluating our method. In addition, we develop an analytic model for this conversion problem [6]. The results show that our model is representative of the simulation process. The difference between the two is only 4%, on the average.

4. CONCLUSION

In conclusion, we would like to summarize our accomplishments, as they relate to the two components of this research project.

Category I (intra-structural reorganization)

a) improved our original clustering algorithm

b) adapted our clustering algorithm for join processing

Category II (inter-structural reorganization)

a) developed a simulation and analytic model for the B+ tree to linear hash file conversion

b) developed a simulation and analytic model for the linear hash file to B+ tree conversion

Overall, I feel that the accomplishments/results of this project were quite good. We looked at a problem area that few researchers have examined and have reasonably shown the feasibility of doing incremental/concurrent reorganization. We presented one paper at the 1988 IEEE Data Engineering Conference. Another paper has been accepted for publication in IEEE Transactions on Software Engineering and two more papers have recently been submitted to journals.

As a last remark, we have assumed (in our work) that the database administrator has decided that reorganization is necessary. However, it would be nice to devise an adaptive database system which would recognize the need for reorganization as well as the specific reorganization function needed. This could then be coupled with incremental/concurrent reorganization to have a more fully automated database administer.

REFERENCES

1. Omiecinski, E., "Concurrent Storage Structure Conversion: from B+ Tree to Linear Hash File," 4th International Conference on Data Engineering, Los Angeles, 1988, 589-596.

2. Omiecinski, E., "Heuristics for Join Processing using Nonclustered Indexes," accepted for publication in IEEE Transactions on Software Engineering.

3. Omiecinski, E., "Concurrent File Conversion between B+ Tree and Linear Hash Files," submitted for publication in IEEE Transactions on Software Engineering.

4. Omiecinski, E., "Imcremental File Reorganization Schemes," 1985 VLDB Conferenece Proceedings, 1985, 346-357.

5. Omiecinski, E., "Concurrency During the Reorganization of Indexed Files," 1985 COMPSAC Proceedings, 1985, 482-488.

6. Scheuermann, P., Park, Y. and Omiecinski, E., "A Heuristic File Reorganization Algorithm based on Record Clustering," submitted for publication in BIT.

TECHNICAL REPORTS

1. Omiecinski, E., "Concurrent Storage Structure Conversion: from B+ Tree to Linear Hash File," 4th International Conference on Data Engineering, Los Angeles, 1988, 589-596.

2. Omiecinski, E., "Heuristics for Join Processing using Nonclustered Indexes," accepted for publication in IEEE Transactions on Software Engineering.

3. Omiecinski, E., "Concurrent File Conversion between B+ Tree and Linear Hash Files," submitted for publication in IEEE Transactions on Software Engineering.

4. Scheuermann, P., Park, Y. and Omiecinski, E., "A Heuristic File Reorganization Algorithm based on Record Clustering," submitted for publication in BIT.

RESEARCH ASSISTANTS (summer 87)

Eileen Tien, Ph.D. student

Mary Jane Causey, Ph.D. student

Cheong Hyeon Choi, M.S. student

Concurrent File Conversion between B+ Tree and Linear Hash Files*

Edward Omiecinski
School of Information and
Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

*This research was partially supported by the National Science Foundation under grant IST-8696157.

ABSTRACT

The motivation for this paper is to show that the efficient reorganization of (1) a B+ tree file into a linear hash file and (2) a linear hash file into a B+ tree file, can be done concurrently with user transaction processing. The conversion process, in general, is motivated by a change in database processing requirements. For case (1), efficient sequential and direct access are originally needed but now only efficient direct access is needed. For case (2), the opposite is true. This is quite reasonable for a database system which accomodates new and changing applications. Several existing database systems, e.g. INGRES [24], IMS [22] and IDMS [22], allow this type of reorganization but the reorganization is performed off-line. We devise an algorithm which performs the conversion for case (1) and case (2), and present an analytic model of the conversion process for each. We also employ a typical database simulation model to evaluate the reorganization scheme. The results from the analytic model for case (1) are within 3% (on the average) of the observed simulation results and for case (2) are within 4% (on the average) of the observed simulation results. The results of the simulations support the idea of doing file conversion concurrently with database usage, especially when compared to an off-line reorganization approach.

1. INTRODUCTION

We define file reorganization as the process of changing the physical structure of the file [22]. Reorganization may be performed for a variety of reasons such as to reduce retrieval time or compact space. Concurrent reorganization is an on-line strategy where the file is reorganized concurrently with usage [22]. With this approach, the part of the file which is being reorganized is locked while user access is permitted to the remainder of the file. The relational database system, System R [2], supports concurrent reorganization to some extent in allowing new attributes to be added to existing relations as well as allowing the creation of new indexes or the deletion of old ones without dumping and reloading the data, i.e. without performing off-line reorganization. In any environment where the database system must be available 24 hours per day, i.e. highly available systems [5], typical off-line reorganization cannot be tolerated. Additional work in concurrent reorganization can be found in [15,16,23].

In this paper we are concerned with a category of file reorganization where the file structure

created by the reorganization process is of a different type than that which existed prior to the reorganization. This might also be called file conversion. An example of this would be to convert an indexed file to a hash based file (or vice versa) as can be done in INGRES with the modify operation [24]. However, in INGRES this is done off-line, i.e. prohibiting user access during the process. This can also be done in IDMS [22] with an unload/reload utility and in IMS [22], changing between HIDAM and HDAM structures. The conversion is motivated by a change in user access patterns of the database. For example, an indexed file structure is chosen originally, since it can efficiently handle range queries. Now, however, the predominate type of query is exact-match. Performance can be improved by having a file structure that can more efficiently handle exact-match queries, e.g. a hash based file structure. We assume, the current access patterns will hold for some time in the future, so that the conversion is beneficial.

In this paper, we propose concurrent reorganization schemes which allow (1) an on-line conversion of a B+ tree to a linear hash file and (2) an on-line conversion of a linear hash file to a B+ tree. The conversion, in either case, works quiet nicely since both file structures are dynamic, i.e. they can grow and shrink one page at a time. Until the reorganization is complete, part of the file would exist as a B+ tree and part as a linear hash file. It will also be quiet clear which file has to be accessed for a given query. In section 2, we review some of the relevant work which exists. In section 3, we examine the B+ tree to linear hash file conversion process, as well as the linear hash file to B+ tree conversion process. In sections 4 and 5, we introduce an analytic model for the B+ tree to linear hash file conversion and an analytic model for the linear hash file to B+ tree conversion, respectively. The simulation model with results are included in section 6.

2. BACKGROUND

First of all we will briefly review linear hashing which was originally proposed in [13] and extended by various researchers [10,11,17,19]. Linear hashing is intended for files that expand and contract dynamically. For the expansion process buckets (i.e. pages) are split in a cyclic manner. One rule that can be used to decide when to expand is to split the next bucket in the cycle whenever any bucket overflows. This is referred to as uncontrolled splitting [13]. In addition, random access of a given

record, on the average, requires approximately 1 disk access [13].

In linear hashing, the hash function to be applied changes as the file grows or shrinks. The function, $h_0 : k \rightarrow \{0,1,...,N-1\}$ is used to initially load the file where k is a key. The hash function is dynamically modified creating a sequence of hash functions $h_1, h_2, ..., h_i$ such that for any k either $h_i(k) = h_{i-1}(k)$ or $h_i(k) = h_{i-1}(k) + 2^{i-1} \times N$.

When a key is to be inserted, the appropriate function is used to find the correct bucket. Collisions are handled by creating a chain of overflow buckets and in addition a split is performed. The splits are performed in linear order, starting from bucket 0. When all N buckets are split, the address space doubles in size and the splitting process starts again from bucket 0. Two variables are maintained for this process: NEXT, which denotes the next chain to be split and LEVEL, which represents the number of times the address space has doubled in size. These variables are updated as follows:

$$NEXT \leftarrow (NEXT + 1) \bmod N \times 2^{LEVEL}$$

$$\text{if } NEXT = 0 \text{ then } LEVEL \leftarrow LEVEL + 1.$$

Using these two variables, the bucket where a record is to be stored is determined as follows:

$$BUCKET \leftarrow h_{LEVEL}(key)$$

$$\text{if } BUCKET < NEXT \text{ then } BUCKET \leftarrow h_{LEVEL+1}(key)$$

where $h_{LEVEL}(key)$ is defined as key mod $2^{LEVEL} \times N$.

Hashing is the appropriate file organization when random access is needed but when both random and sequential access is needed a more appropriate structure to use is the B+ tree [3]. The B+ tree or other variants of the B-tree [3] have been widely used in recent years for storing large files of information on secondary storage, e.g. System R [2]. The average random access search time is typically 3 or 4 disk accesses depending on the height of the tree. Efficient sequential processing is provided by linking the leaf nodes of the B+ tree together in key sequence order. A sample B+ tree is shown in figure 1.

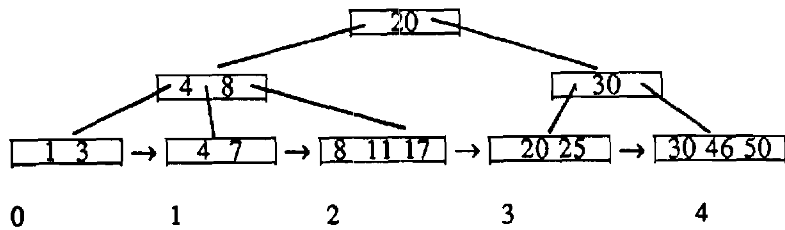


Fig. 1. B+ tree example

To allow for concurrent operations on these file structures, both have undergone modification. The method for achieving greater concurrency is to provide a way to detect and recover from the effect of concurrent updates. In [8,12], schemes were presented to allow for concurrent manipulation of B+ trees. In [12], a single additional link in each node allows a process to easily recover from tree modifications performed by other concurrent processes. Their solution uses a simple locking scheme and requires only a small and constant number of nodes to be locked by an update process at any given time. In [4], a solution for concurrent access to a linear hash file is presented. Part of the solution involves a minor modification to the hash file, i.e. keeping a LOCAL-LEVEL variable with each bucket. The other part carries over the idea of additional links from B+ trees in the form of recalculation [4]. The notion of recalculation is as follows: upon gaining access to a bucket, a process checks whether the LEVEL value used to calculate the address matches LOCAL-LEVEL, and if not, it increments its value and recalculates the address until a match is found. The buckets reached in this manner are those that were created by splitting buckets at addresses already accessed during this search. A simple locking scheme is also employed to control access to the shared variables, LEVEL and NEXT, as well as to the buckets. We should note that a primary bucket and all of its overflow buckets (i.e. a chain) are locked as a unit.

3. CONVERSION BETWEEN B+ TREE AND LINEAR HASH FILES

Besides performing the reorganization on-line, we also want to use the storage space of the original file with perhaps a small additional amount. The amount of additional storage will depend on the specific conversion process. We will first examine the B+ tree to linear hash file conversion and then the reverse conversion process. Initial work on the B+ tree to linear hash file conversion problem has

appeared in [14].

The reorganization process takes one page at a time from the B+ tree file and inserts the records from that page in the linear hash file. As we reorganize a page from the B+ tree file, that page, i.e. the storage unit, can be added to the linear hash file when a subsequent split operation is performed on the hash file. However, the first page, i.e. page 0, is added when the first B+ tree page is reorganized.

The reorganization process proceeds in physical address sequence of pages, e.g. 0,1,2,... . If the key sequence of the records corresponds to the physical address sequence then during reorganization the database system need only keep track of the highest key (record) moved from the original B+ tree file to the hash file. Knowing the highest key will allow the database system to direct searches, updates and deletes for a key of smaller or equal value to the linear hash file and for a larger key to the remaining B+ tree file. This allows the benefits of the partial linear hash file to be gained immediately, i.e. as each page is converted.

However, due to splitting pages in the B+ tree file, the physical sequence of data pages may not correspond exactly to the key sequence. Some B+ tree file systems like IBM's VSAM [3] group consecutive pages (control intervals, a la IBM) into larger physical areas (control areas). Therfore when a page is split, an empty page within the same area is used. If the empty page is not physically adjacent to the old one then the key sequence - physical sequence pairing is lost. However, the two pages are still within the same area. Although, in the worst case the area may be full, thus causing an additional area to be allocated. If there still is a somewhat limited form of clustering, in that key sequence is maintained between areas, then once an entire area has been converted to the linear hash file structure, the pages within that area will be accessible through the hash file. Once again, the decision to use the B+ tree or linear hash file will be based on the high key value previously mentioned.

As an alternative, we could proceed in key sequence, regardless of the key-page sequence correspondance, but we would have to employ indirect addressing. The hash function would produce an entry in a page table which would contain the relative page (bucket) number. Using a page table would increase i/o time if the table could not be maintained in memory. This tradeoff would have to be ex-

plored before committing to this approach.

A very high level (abstract) view of the B+ tree to linear hash file conversion algorithm is presented in figure 2. The Insert_separately procedure is basically the insert procedure of [4], where each key is inserted separately.

```
Procedure BtreeHash;
Begin
    For each leaf page, lpage, in sequence
     of the B+ tree do
        Begin
            Get exclusive lock on lpage;
            Get lpage;
            Get shared lock on state variables;
            Calculate bucket address in Linear
              Hash file for all keys in lpage;
            If the number of distinct bucket
              addresses + # of keys > threshold
            then
              Insert_Separately
              {each key is inserted as a separate
              operation, requires exclusive locks}
            else
              Insert_Group
              {keys belonging to the same bucket
              are inserted in one operation, requires
              exclusive locks};
            If an overflow has occurred
            then
              Split;
            Release locks;
        End;
End.
```

Fig. 2. B+ tree to linear hash file conversion algorithm

The Insert_group procedure in figure 2 generates a reorganization transaction which inserts multiple records on the same page. This saves i/o and cpu time since the same page need not be locked and written multiple times. However, during this process the state variables have a shared lock on them which means that concurrent updates on the hash file cannot take place. This procedure would be used initially during the conversion since the hash file is small and there is a greater likelyhood that keys will hash to the same bucket. On the other hand, the Insert_separately procedure generates a reorganization transaction which inserts a single record. This is useful when the linear hash file becomes

large enough such that keys from the B+ tree leaf pages hash to different buckets. The advantage here is that the lock on the state variables can be released once the linear hash file page has been locked. Which procedure to call is controlled by a threshold value. For example, if more than 50% of the keys from the B+ tree leaf page hash to different pages then call Insert_separately otherwise call Insert_group.

An example of the conversion process using the B+ tree of figure 1 follows. The algorithm and the example illustrate the simple case where the leaf pages appear in physical sequential order by key value. The algorithm is easily adapted to handle the more general case by keeping track of the pages that have been reorganized and by delaying the updating of the global variable high_key until the pages which represent a consecutive range of key values have been converted. The analytic and simulation models of sections 4 and 6 are based on this more general and more realistic situation. However, at this point the simple case will suffice to illustrate the conversion concepts.

Example 1 (refer to figure 3)

Step 1. get an exclusive lock on page 0, bring leaf page 0 into the buffer, make into hash page structure (i.e. containing records and local_level) using key mod $2^0 \times 1$ as the current hash function and update high_key to 3. This requires a shared lock on the state variables (next and level) and an exclusive lock on the high_key variable. Afterwards, all locks are released.

Step 2. get an exclusive lock on page 1, bring leaf page 1 into the buffer, using our previous hash function, both keys 4 and 7 would hash to page 0 so get an exclusive lock on page 0, get page 0 and add records to overflow chain for page 0. The overflow generates a split which is done after the records on page 1 have been inserted in the hash file. Performing the split at this time allows page 1 to be added to the hash file storage space. We should note that splitting is done if overflow occurs but after all the records on the current page, which is to be reorganized, have been inserted. This is necessary, as in this case, so that there will be a new page which can be used for the split. If the page needed for the split, i.e. at location NEXT + $2^{LEVEL} \times N$, has not been converted then the split is deferred. This requires exclusive locks on the state (since LEVEL is increased to 1) and

high_key (which is set to 7) variables. Afterwards, all locks are released.

Step 3. get an exclusive lock on page 2, bring leaf page 2 into the buffer, the keys 8, 11 and 17 hash to pages 0, 1 and 1 respectively, obtain exclusive lock on page 0 and insert key 8, obtain exclusive lock on page 1 and insert keys 11 and 17 on an overflow page (we should note that having an exclusive lock on a primary page precludes access by other transactions on the overflow chain as well as on that primary page), locks would be released. a split process is generated next which requires exclusive locks on pages 0 and 2, afterwards the locks are released.

Step 4. similar to previous steps except that page 3 is being converted.

Step 5. similar to previous steps but with page 4 being reorganized.

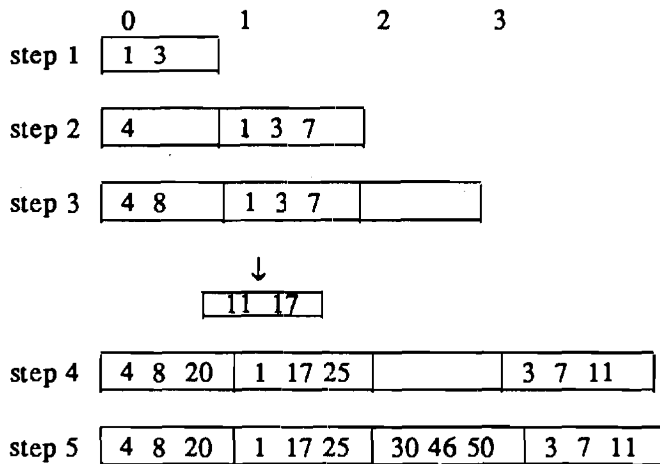


Fig. 3. File structure changes during conversion example 1

For this particular example, the linear hash file used one less page of storage as did the leaf pages of the B+ tree file. In general, this is not the case but as we will see in section 6, the additional space used by the linear hash file will be insignificant. If we consider the space used by the index, the linear hash file will probably require less.

At this time, we can examine the conversion of a linear hash file to a B+ tree file. The reorganization process takes one page (i.e. a bucket) and its associated overflow pages from the linear hash file and inserts the records contained on those pages in the B+ tree. As we reorganize a page from the linear hash file, that page, i.e. the storage unit, can be added to the B+ tree file when a subsequent split operation on a B+ tree node is performed. However, the first page, i.e. page 0, is added when the

first linear hash file page is reorganized.

The reorganization process proceeds in physical address sequence of pages, e.g. 0,1,2,...n, in the linear hash file. Knowing the highest page that has been converted will allow the database system to direct searches, updates and deletes to the appropriate file. For example, if a search request is made for a key, k, where $h_i(k)$ is greater than the address of the last converted page, then the linear hash file will contain the desired key, if it is present. Otherwise, the key will be in the B+ tree file, if present.

A very high level version of the linear hash file to B+ tree conversion algorithm is presented in figure 4.

```
Procedure HashBtree;
Begin
    For each page, hpage, of the linear
      hash file, in ascending order, do
        Begin
            Get exclusive lock on hpage;
            Get hpage and any associated overflow pages;
            Sort keys contained on those pages;
            Btree_Insertion;
            {insert sorted keys in order,
            requires locking}
            Release locks;
        End;
End;
```

Fig. 4. Linear hash file to B+ tree conversion algorithm

The Btree_Insertion procedure, in figure 4, inserts keys, in ascending order, in the B+ tree. Hence, if several keys belong on the same page of the B+ tree, multiple page accesses of the same pages (i.e. index and data pages) can be eliminated. A different approach which can be used is the batch modification algorithm in [9]. In [9], an efficient algorithm is presented which takes a differential file of sorted records which are tagged as inserts, deletes or updates; and modifies the B+ tree file accordingly. The keys are not only inserted in sorted order but as a group. Using this approach, it might be possible to reduce the number of page acceses even further.

An example of the linear hash file to B+ tree conversion, using the linear hash file in step 5 of figure 3, is shown below. Since we use the B^{link}-tree structure [12], only one type of lock is used which

prevents multiple updates but never prevents a read. In addition, with this scheme, at most 3 nodes will be locked at any one time, thus providing a high level of concurrency. To simplify matters, we do not mention the horizontal pointers of the B^{link}-tree in the example.

Example 2 (refer to figure 5)

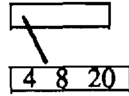
Step 1. Get a lock on page 0 of the linear hash file, bring page 0 into the buffer, convert it to a B+ tree leaf page and make a dummy root (i.e. root has pointer to leaf page but no key), release lock.

Step 2. Get lock on page 1, bring page 1 into the buffer, sort the keys, access the root of the B+ tree, follow pointer to leaf page, get lock on leaf page. Inserting key 1 causes a split. Likewise, inserting key 25 causes a split and a key insertion in the root node. So a new page is added and the keys are distributed. This also causes the insertion of key 8 into the root. To do this, a lock is required on the root and we must backtrack to the root. Release locks.

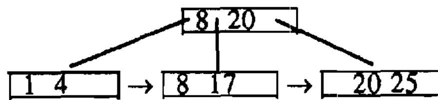
Step 3. Get lock on page 2, bring page 2 into the buffer, sort the keys, access root of B+ tree, follow pointer to leaf page, get lock on leaf page. Key 30 is inserted in the page with keys 20 and 25. Inserting key 46 causes a split, so a new page is added and the keys are distributed. We backtrack to the root, obtain a lock on it and insert key 30. In a similar manner, key 50 is inserted except that no splitting is required. Release locks.

Step 4. Get lock on page 3 of the linear hash file, bring page 3 into the buffer, access the root of the B+ tree, follow pointer to leaf page 0, get lock on leaf page, insert key 3 and key 7. Inserting key 7 causes a split process. The root is locked next and key 4 is inserted. This causes the root to split. The keys are distributed and a new root node is produced containing key 20. Locks are released.

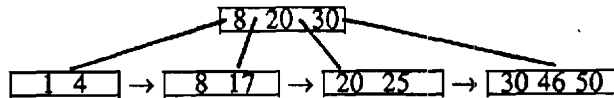
Step 1)



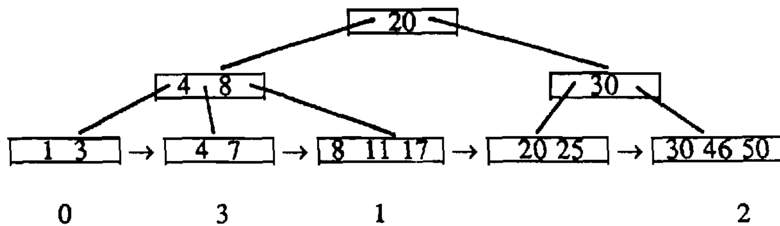
Step 2)



Step 3)



Step 4)



0 3 1 2

Fig. 5. File structure changes during conversion example 2

4. ANALYTIC MODEL OF B+ TREE FILE TO LINEAR HASH FILE CONVERSION

In this section we present a simple analytic model of the conversion process. In particular, we want to determine the breakpoint, i.e. how many pages need to be converted before the throughput of the system with concurrent reorganization will equal the throughput of a system with only transaction

processing using the B+ tree file. After this point, the performance of the system with concurrent reorganization becomes better. The various properties that we are interested in are as follows.

convert(i) : cost of converting ith page from B+ tree to linear hash file (in pages)

split(j) : cost of performing jth split in linear hash file (in page accesses)

btree : expected cost of a transaction for the B+ tree file (in page accesses)

bhash(i) : expected cost of a transaction using the B+ tree / linear hash file
after the ith page has been converted (in page accesses)

height : height of B+ tree

mp : multiprogramming level

n : number of leaf pages in B+ tree file

nrec : number of records in B+ tree leaf page

r : reorganization unit

rpr : probability that a transaction only reads

wpr : probability that a transaction does an update, i.e. 1-rpr

lfpr : probability that an insertion in the linear hash file is to
the left of the split pointer

rtpr : probability that an insertion is to the right of the split pointer

ovfl(m) : overflow chain length for home page left of the split pointer
for a linear hash file with m records

ovfr(m) : overflow chain length for home page right of the split pointer
for a linear hash file with m records

To simplify our analysis, we will not take into account the effect of locking. Access to our file is based on a single key, so the probability that two transactions will block each other will be small. (In the simulation of section 6, the queries were generated randomly with a uniform distribution so our assumption should not be too severe.) Even though the reorganization process locks at most nrec home pages in the linear hash file, the conflict will still be fairly small. We should also note that when a page is converted (excluding the first page), a split operation is performed to add that physical page to the linear hash file. This varies from the algorithm but it is a close approximation which will simplify our model.

The major cost of query processing is accesses to secondary storage. As such, we will use the number of i/o page accesses which a given conversion process makes, multiplied by the multiprogramming level (which is fixed) as the time to complete a given reorganization process. The number of transactions that can be processed using the B+ tree file during the time it would take to convert L pages for the B+ tree / linear hash file is defined below.

$$\sum_{i=1}^{L} \frac{mp(convert(i)+split(i))}{btree} \qquad (1)$$

For the file system where the conversion is taking place, we have the following number of transactions that can be processed for the same time frame.

$$\sum_{i=1}^{L} \frac{(mp-1)(convert(i)+split(i))}{bhash(i-1)} \qquad (2)$$

We have a factor of mp-1 in the numerator of (2) since the conversion process is being performed concurrently with transaction processing. Hence, the time available for user transactions is reduced.

The derivations of the following formulas are straightforward and are omitted. Similar formulas for the linear hash file operations are presented in [18]. The individual terms of (1) and (2) are derived as follows.

$$split(i)=2(1+ovfl(i*nrec))+(1+ovfr(i*nrec))$$

$$convert(i)=1+nrec[(2+ovfl(i*nrec))*lfpr+(2+ovfr(i*nrec))*(1-lfpr)]$$

$$btree=rpr*height+(1-rpr)(height+1)$$

$$bhash(i)=(rpr)[height\frac{n-\lfloor i/r \rfloor r}{n}+((1+ovfl(i*nrec))*lfpr+(1+ovfr(i*nrec))(1-lfpr)\frac{\lfloor i/r \rfloor r}{n}]+$$
$$(1-rpr)[(height+1)\frac{n-\lfloor i/r \rfloor r}{n}+((2+ovfl(i*nrec))*lfpr+(2+ovfr(i*nrec))(1-lfpr)\frac{\lfloor i/r \rfloor r}{n}]$$

The bhash formula has a component for searching/updating the part of the file that resides in the B+ tree and linear hash structure. For the reorganization unit of 1 page, there is a probability of i/n that the query accesses the linear hash structure and (n-i)/n that it accesses the B+ tree structure. If the reorganization unit is 5 for example, then groups of 5 pages would have to be converted before any of

the records from those 5 pages could be accessible through the linear hash structure. Hence, in general, for a reorganization unit of r, the probability that a B+ tree leaf page is accessed is $(n-\lfloor i/r \rfloor r)/n$ and the probability that a linear hash file page is accessed is $(\lfloor i/r \rfloor r)/n$.

To simplify our problem, we will make one last assumption. We will assume that overflow is negligible. This should not limit our model too much since a successful search in a linear hash file requires on the average a single access [13]. This assumption has a direct effect on convert(i) and split(i). It means that the cost of converting and splitting a page is constant. Hence, when we want to compute the breakpoint, i.e. the value such that

$$(mp-1)\sum_{i=1}^{L} \frac{convert(i)+split(i)}{bhash(i-1)} - mp\sum_{i=1}^{L} \frac{convert(i)+split(i)}{btree} = 0, \quad (3)$$

we can cancel out the convert(i) + split(i) term. Once again, this simplification is not entirely correct but it should yield a reasonable approximation.

In addition bhash(i) becomes the following:

$$rpr\,(height\frac{n-\lfloor i/r \rfloor r}{n} + \frac{\lfloor i/r \rfloor r}{n}) + (1-rpr)[(height+1)\frac{n-\lfloor i/r \rfloor r}{n} + 2\frac{\lfloor i/r \rfloor r}{n}].$$

Thus, through simplification, the breakpoint equation of (3) becomes

$$(mp-1)\sum_{i=0}^{L-1} \frac{1}{(height+1-rpr)\frac{n-\lfloor i/r \rfloor r}{n} + (2-rpr)\frac{\lfloor i/r \rfloor r}{n}} - mp\sum_{i=1}^{L} \frac{1}{rpr*height+(1-rpr)(height+1)} = 0 \quad (4)$$

Since the denominator of the term in the second summation is a constant we can further simplify (4) and group terms to get (5).

$$\sum_{i=0}^{L-1} \frac{1}{(height+1-rpr)n-(height-1)\lfloor i/r \rfloor r} - \frac{mp*L}{(mp-1)(height+1-rpr)n} = 0 \quad (5)$$

At this point, we would like a closed form expression for the summation in (5). We can replace the summation with

$$\frac{1}{height-1}\sum_{i=0}^{L-1}\frac{1}{\frac{(height+1-rpr)n}{height-1}-\lfloor i/r\rfloor r}.$$

With a little effort, we can derive an equivalent summation in a more suitable form, as shown below.

$$\frac{1}{height-1}\left(r\sum_{j=0}^{\lfloor L/r\rfloor-1}\frac{1}{\frac{(height+1-rpr)n}{height-1}-j}+\frac{L \bmod r}{\frac{(height+1-rpr)n}{height-1}-\lfloor L/r\rfloor}\right.$$

$$\left.-\frac{1}{\frac{(height+1-rpr)n}{height-1}}\right)+\frac{height-1}{(height+1-rpr)n}.$$

Using the Harmonic numbers [20], $H_n=\sum_{i=1}^{n}\frac{1}{i}$, we can do a further reduction to yield

$$\frac{1}{height-1}\left(r^*H_{\frac{(Height+1-rpr)n}{height-1}-1}-r^*H_{\frac{(height+1-rpr)n}{height-1}-\lfloor L/r\rfloor}+\frac{L \bmod r}{\frac{(height+1-rpr)n}{height-1}-\lfloor L/r\rfloor}\right.$$

$$\left.-\frac{1}{\frac{(height+1-rpr)n}{height-1}}\right)+\frac{height-1}{(height+1-rpr)n}.$$

Using the approximation [6], $H_n \approx ln(n)+\gamma+1/2n$, and substituting back into (5) yields the following :

$$\frac{1}{L}\left[ln\left(\frac{(height+1-rpr)n}{height-1}\right)+\frac{1}{\frac{2(height+1-rpr)}{(height-1)}}-ln\left(\frac{(height+1-rpr)n}{height-1}-\lfloor L/r\rfloor\right)\right.$$

$$-\frac{1}{2\left(\frac{(height+1-rpr)n}{height-1}-\lfloor L/r\rfloor\right)}+\frac{L \bmod r}{r\left(\frac{(height+1-rpr)n}{height-1}-\lfloor L/r\rfloor\right)}$$

$$\left.-\frac{1}{\frac{r(height+1-rpr)n}{(height-1)}}\right)+\frac{height-1}{(height+1-rpr)n}\Bigg]-\frac{mp(height-1)}{(mp-1)(height+1-rpr)n}=0 \qquad (6).$$

Since all the but one variable in (6) are known, we simply need to find the root of equation (6) which is not a difficult task. We can use an iterative method for approximating a real root of an equation [7]. In a subsequent section, we will compare the results of our analytic model with those of the simulation model.

5. ANALYTIC MODEL OF LINEAR HASH FILE TO B+ TREE FILE CONVERSION

For the linear hash file to B+ tree conversion, we are interested in the following properties.

convert(i) : cost of converting ith page from linear hash file to B+ tree (in pages)

bhash : expected cost of a transaction for the linear hash file (in page accesses)

btree(i) : expected cost of a transaction using the linear hash / B+ tree file
after the ith page has been converted (in page accesses)

m : branching factor of B+ tree

n : number of primary pages in the linear hash file

lrec : average number of records in linear hash file

erpr : probability that a transaction reads a single record (exact match read)

ewpr : probability that a transaction updates a single record (exact match write)

rrpr : probability that a transaction reads records within a specified range (range query)

min : minimum number of records in range query

max : maximum number of records in range query

brec : average number of records in a B+ tree data page

bprob(j,i,rs) : probability that j consecutive keys from a range of rs keys are
in the B+ tree, after i pages of the linear hash file have been converted

hprob(j,i,rs) : probability that j consecutive keys from a range of rs keys are in
the linear hash file, after i pages of the linear hash file have been converted

To simplify our analysis, as in the previous analytic model, we will not take into account the effect of locking. This is reasonable for an environment where the majority of the transactions are read-only. In addition, we assume, as we did previously, that overflow in the linear hash file is negligible. Lastly, we make the assumption that the cost to convert any page in the linear hash file is constant. This is obviously not true since the insertion of a certain key may just involve reading index pages and writing a data page whereas the insertion of some other key may involve splitting a data page as well as index pages. To incorporate the probability that a node either at the data level or at any index level will be split would overly complicate our model as well as being difficult. Since one goal of our model is simplicity, although not at the sacrifice of accuracy, we make the constant page conversion

assumption. In addition, for the particular environment which we study, this assumption should not be too bad since we will always maintain the root of the B+ tree in memory, the branching factor of the B+ tree will be 30 and the maximum height of the B+ tree will be only 3.

The breakpoint which we want to compute is the value of L such that

$$(mp-1)\sum_{i=1}^{L}\frac{convert(i)}{btree(i-1)} - mp\sum_{i=1}^{L}\frac{convert(i)}{bhash} = 0, \qquad (7)$$

The terms btree(i) and bhash are defined below.

$$btree(i) = erpr(\frac{n-i}{n} + \frac{i}{n}\left\lceil \log_m lrec*i \right\rceil) + ewpr(\frac{n-i}{n} * 2 + \frac{i}{n}(\left\lceil \log_m lrec*i \right\rceil) + 1) +$$

$$\frac{rrpr}{max-min+1}\sum_{k=min}^{max}[\sum_{j=0}^{k}(bprob(j,i,k)*\left\lceil\frac{j}{brec}\right\rceil + hprob(k-j,i,k)*(k-j)) +$$

$$\left\lceil \log_m lrec*i \right\rceil * \sum_{j=1}^{k} bprob(j,i,k)] \qquad (8)$$

$$bhash = erpr + ewpr*2 + rrpr*\left\lceil\frac{\max+\min}{2}\right\rceil \qquad (9)$$

In cost equations (8) and (9), we assume that the range queries are uniformly distributed between min and max keys per query. Hence, the summation from min to max is divided by 1/(max-min+ 1). We also assume that all keys within the range are present in the database. In equation (8), if j keys from the range are in the B+ tree, we assume that they are contained on the minimal number of pages, where each page contains brec records. As a reminder, brec really is the average number of records per data page. Again, this is a simplification, due in part to the fact that the keys are maintained in sorted order. If rs-j keys are in the linear hash file, then those keys are contained on rs-j distinct pages as we will see later. This is due to the modulo hash function and a file size greater than rs-j.

The interesting part of equation (8) is determining the probabilities: bprob(j,i,k) and hprob(k-j,i,k). One point, which we can make, about these probabilities is that bprob(j,i,k) = hprob(k-j,i,k). That is, if j keys out of k keys are in the B+ tree file then the remaining k-j keys must be in the linear hash file. We need to consider four cases to determine bprob(j,i,k). The individual cases are dependent upon the number of pages already converted, i; the number of pages remaining to be converted,

n-i; and the range size, k. The four cases are shown below and we will later show that these cases yield the correct probabilities.

Case I: if $i < k$ and $n-i \geq k$ then bprob(j,i,k) =

(a) $2/n$, if $j < i$

(b) $(k-i+1)/n$, if $j = i$

(c) 0, if $j > i$

Case II: if $i < k$ and $n-i < k$ then bprob(j,i,k) =

(a) 0, if $n-i < k-j$ or $j > i$

(b) $(k-(n-i)+1)/n$, if $n-i > k-j$

(c) $2/n$, if $n-i > k-j$ and $j \leq i$

Case III: if $i \geq k$ and $n-i \geq k$ then bprob(j,i,k) =

(a) $2/n$, if $j < k$

(b) $(i-k+1)/n$, if $j = k$

Case IV: if $i \geq k$ and $n-i < k$ then bprob(k-j,i,k) =

(a) $2/n$, if $j < n-i$

(b) $(k-(n-i)+1)/n$, if $j = n-i$

(c) 0, if $j > n-i$

For all four cases, we have the following:

$$bprob(0,i,k) = 1 - \sum_{j=1}^{k} bprob(j,i,k)$$

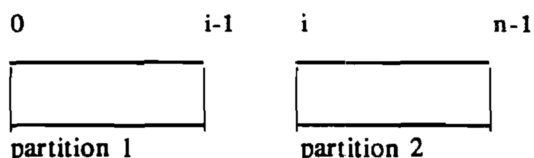
Hence, the probability that a specific file, either the B+ tree or linear hash file, has 0,1,...,rangesize number of records of the range query in it, is 1. We should also point out, that case II and case III cannot both occur during the conversion process for a fixed value of k. This can be seen by simply looking at the inequality conditions for these cases.

We will now show that the four cases reflect the actual probabilities. We should again mention

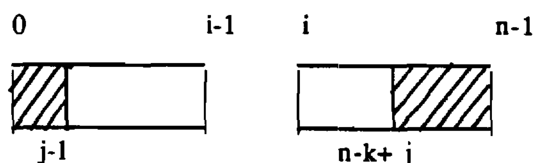
that the modulo hash function, i.e. $h_{level}(key)$ = key mod 2^{level}, forces consecutive keys into adjacent buckets. Note that we consider the last bucket to be adjacent to the first. For example, 14 mod 8 is 6, 15 mod 8 is 7 and 16 mod 8 is 0. We also assume that n≥ k, so each key in the range of size k is contained in a different bucket. The original file can be thought of as being divided into two partitions: the first, representing buckets already converted to the B+ tree and the second, representing the remaining buckets of the linear hash file.

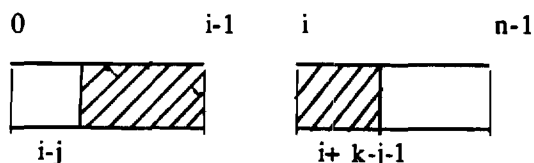
Case I: i< k and n-i≥ k

The following diagram illustrates the state of the converted file, where the left side represents the i converted pages and the right side represents the n-i pages to be converted.



(a) If j< i, we know that the j consecutive blocks have to be the first j blocks (i.e. 0 through j-1) or the last j blocks (i.e. i-j through i-1) of partition 1. This is necessary since all the blocks in the range query have to be consecutive and since partition 1 is larger than the number of consecutive blocks, of the range which it contains. That is, we cannot choose some other j blocks because this would result in a nonconsecutive sequence of blocks. The following two diagrams illustrate the two possible situations.

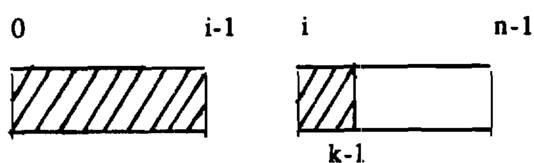




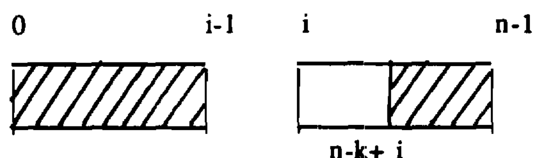
Thus, the first key of the range either appears in bucket n-k+ j or bucket i-j. Since it is equally likely, i.e., 1/n, for the first key of a range query to appear in any bucket, the probability for this part of case I is 2/n.

(b) For j = i, we have the following three possibilities.

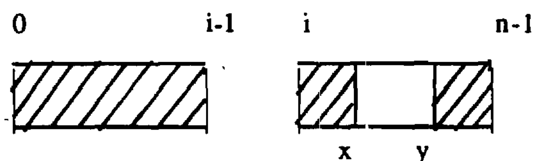
(1)



(2)



(3)



The situations depicted in (1) and (2) are analagous to those in part (a) of case I. Situation (3) has two variable bucket addresses, x and y. This situation represents the transition from (1) to (2) or vice versa. Hence the value of x ranges from k-2 to i. So for situation (3) we have a probability of (k-2-i+ 1)/n of this happening. With situation (1) and (2) included, we have a probability of (k-i+ 1)/n for part (b).

(c) If j> i, then it is obvious that partition 1 has a 0 probability of containing j keys.

Case II: $i < k$ and $n-i < k$

In this case, the number of keys in the range query is too large to be contained entirely in partition 1 or partition 2.

(a) If the size of partition 2, i.e., n-i, is smaller than k-j, then partition 1 cannot possibly contain j keys. Partition 1 would have to contain more than j keys. Hence, the probability that partition 1 contains j keys would be 0. Likewise, if $j > i$ then we would also have a 0 probability, since the number of buckets in partition 1 is less than the number of keys it is suppose to contain.

(b) If the size of partition 2 is equal to k-j, i.e. $n-i= k-j$, then we have a similar situation as we had in case I (b) (3), except that the roles of partitions 1 and 2 are switched. Hence, we have a probability of $(k-(n-i)+ 1)/n$. This holds for the same reason as does case I (b) (3).

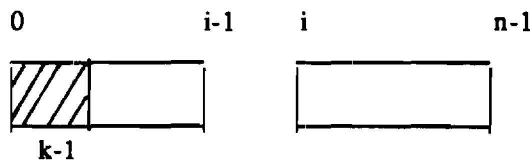
(c) If the size of partition 2 exceeds k-j, i.e. $n-i> k-j$, then we have a similar situation as in case I (b) (1) and case I (b) (2). Hence, we have a probability of $2/n$. This holds true as long as $j\leq i$.

Case III: $i\geq k$ and $n-i\geq k$

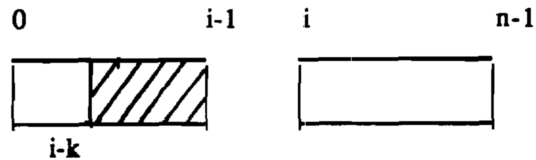
(a) Here we have the same situation as in case I (a) (1) and case I (a) (2) when $j< k$. Thus, we have a probability of $2/n$.

(b) Since the size of each partition is greater than the range size, k, all the k keys may be contained in a single partition. Two examples of this are shown below.

(1)



(2)



Both (1) and (2) illustrate a situation of k consecutive buckets. Hence if the starting key is in bucket 0,1,...,i-k; partition 1 will contain all k keys. So, we have a probability of (i-k+ 1)/n that partition 1 contains all k keys of the range.

Case IV: $i \geq k$ and $n-i < k$

This case is the mirror image of case I, i.e., we simply exchange the roles of partitions 1 and 2.

6. DATABASE SIMULATION MODEL AND RESULTS

The simulation model is an adaptation of the models presented in [1,21] and is shown in figure 6. The simulation model uses a fixed multiprogramming level and a dynamic locking scheme where the lockable units are pages. The simulation parameter values used for the experiments, which pertain to both conversion problems, are given in table I and are typical of those in [1,21]. Tables II and III contain the parameter values specific to the simulation of the given conversion problem.

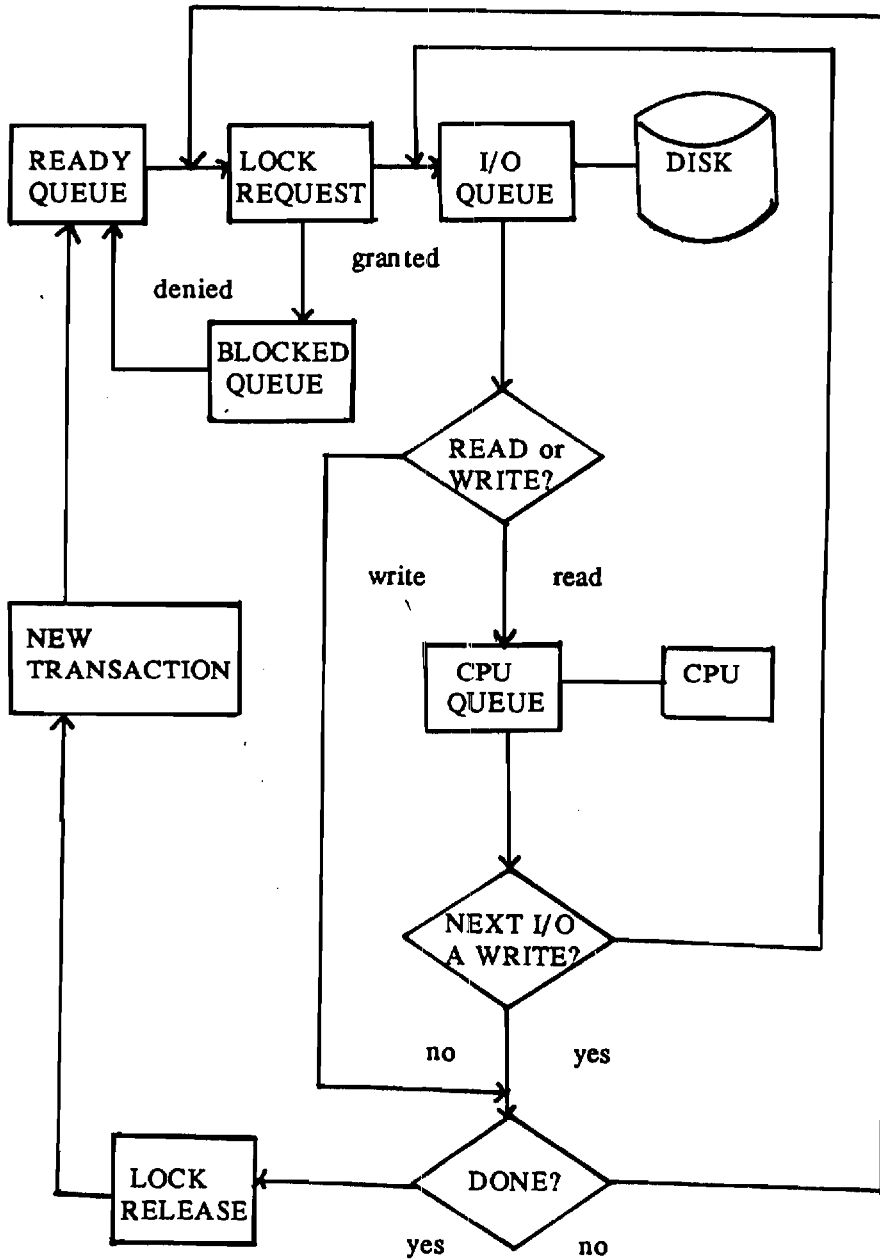


Fig. 6. Database simulation model

Table I. General simulation parameter settings

Parameter	Value
multiprogramming level	10
write probability	0.25
page i/o time	35 ms
page cpu time	15 ms
lock i/o time	0
lock request cpu time	2.5 ms
lock release cpu time	2.5 ms

Table II. B+ tree to linear hash file simulation parameter values

Parameter	Value
database size	143 pages
data page size	10 records
data page load factor	0.70
B+ tree height	3
reorganization unit	1,5,10,15, 20,25,...

Table III. Linear hash file to B+ tree simulation parameter values

Parameter	Value
database size	128 pages
hash page size	11 records
tree branching factor	30
range size	5-25,5-50,5-100
write probability	20%
read probability	50% to 20%
range probability	30% to 60%

The model simulates transactions made against the database. The transactions are of two different types: file processing and reorganization. File processing transactions are submitted by users for retrieval or update of the file. In the simulation, a fixed number of transactions are active at any one time. When the first user transaction terminates, a reorganization process is entered into the system. When each reorganization process terminates, a new one is generated. Therefore, only one reorganization process is active at a time. All reorganization processes require locks which are requested one at a time and which prevent update transactions. To prevent deadlock in our database model, locks are requested in a fixed linear order. We also assume that to write an existing page, a transaction must first read it.

Initially, user transactions arrive one time unit apart and are placed on the READY queue. A transaction then goes through the following stages.

(a) The transaction is removed from the READY queue and one lock is requested. If the lock is granted, the transaction is placed on the bottom of the I/O queue. If the lock is denied, the transaction is placed at the bottom of the BLOCKED queue. The blocking transaction is recorded.

(b) After completing the required I/O, the transaction does one of two things. If the I/O is a read then the transaction is placed at the bottom of the CPU queue. If the I/O is a write then the transaction, if it were not finished, would request another lock and cycle around again. If the transaction is finished after the write then all locks are released. All transactions blocked by the completed transaction are placed on the front of the READY queue.

(c) After completing the required cpu for the page accessed, the transaction, if it were not finished, would request a lock on another page and repeat the cycle or if the current page is to be written after being read, the transaction would be placed at the bottom of the I/O queue. If the transaction is finished after the cpu processing then all locks are released. All transactions blocked by the completed transaction are placed on the front of the READY queue.

The motivation for the B+ tree to linear hash file conversion is the need for efficient direct access only. As such, we will restrict the file processing transactions to just those. These transactions involve accessing two index pages (i.e. nonleaf nodes) and a data page (i.e. leaf node). To simplify the simulation we assume that the pages in the B+ tree do not split or merge. The reorganization transaction type can actually be divided into two classes. One class accesses a data page from the B+ tree file and places the records from that page into the linear hash file. This may involve accessing as many as ten (i.e. page size in records) primary pages in the linear hash file as well as additional overflow pages. The other class, due to a split, accesses two pages from the linear hash file and possibly some overflow pages.

For the linear hash file to B+ tree conversion, there are three types of file processing transactions: exact match read, exact match write (update) and range read (range query). For the exact match

transactions, a random key is chosen. For the range query, a random number, R, is chosen for the range size (between some minimum and some maximum value). Next, a random key, K, is chosen, such that the key whose value is K+ R-1 is also contained in the file. We should make note that the keys in the linear hash file are consecutive. Hence, when we have a range of size R, we know that R records will be retrieved. This is necessary so that we can determine how many pages will be accessed for a range query in the analytic model.

The results of the various simulation runs for the B+ tree to linear hash file coversion are summarized in table IV. The unit parameter indicates the increment (in pages) for which converted pages are made accessible through the linear hash file. For example, the value 5 indicates that after every group of 5 pages from the B+ tree file have been converted, the records from those pages can be accessed through the linear hash file. This means that the keys (records) that have been processed so far, i.e. in physical page sequence, represent a consecutive range of key values. Since the reorganization is done in page sequence, the higher values for unit represent a greater disparity between key and physical page sequences. The value of 1 indicates that page and key sequences are the same. Unit may be thought of as the size of a control area. The breakpoint gives the time in seconds in which the concurrent reorganization method starts to produce a higher throughput of transactions as compared with only transaction processing of the B+ tree. For the case where the unit is 15 (≅ 11 percent of the database size) we see that the breakpoint occurs around 375 seconds (≅ 2/5 of total reorganization time). The columns under max decrease indicate the maximum number of transactions delayed and the time at which this maximum delay occurred. The columns under max increase show the maximum improvement of throughput and the time at which this happens, i.e. when the reorganization is completed. The first line of the table shows the result of doing off-line reorganization which requires 125 seconds. During that time period the B+ tree file could have serviced 1092 transactions. We should also point out that for a much larger database, the off-line reorganization time would be much larger and more intolerable than for our case.

The average time, for the various simulations, to complete the conversion of the B+ tree to linear hash file was approximately 961 seconds. The average load factor for the linear hash file was 66

percent with 1.03 accesses per successful search. In addition, on the average only 5.6 percent more pages were used by the linear hash file as compared with (only) the leaf pages of the B+ tree file. We see from table IV that as the unit increases, so does the breakpoint and the delay in throughput while the improvement decreases. Of course, when the unit is 1, we have our best result. Although it probably is not a very practical situation. However, other situations where the unit is between 15 (≅ 11% of database) and 35 (≅ 24% of database) appear to be more practical and still yield good results, in terms of breakpoint and delay. The results are especially good when compared with the delay of off-line reorganization.

At this time, we would like to present a comparison of our analytic model with the simulation model. Table V shows the breakpoint, in number of pages, as computed using our analytic model and as observed from the simulation runs. The maximum difference between the two is only 7% and on the average it is about 2%. So, even though we made several simplifying assumptions for our analytic model, the value of the breakpoint predicted was fairly close to the observed value. The difference is probably due to the fact that there is a small amount of overflow with the linear hash file and that the time to convert and split a page is not constant.

Table IV. Summary of b+ tree to linear hash file simulation results

unit	break point (sec)	max decrease		max increase	
		trans	time (sec)	trans	time (sec)
offline		1092	125		
1	284	63	132	3134	958
5	314	75	168	2980	958
10	336	92	173	2816	962
15	375	111	175	2686	967
20	413	126	232	2448	961
25	434	130	228	2285	954
30	453	164	180	2124	960
35	485	193	208	1949	962
40	538	223	240	1959	969
50	575	294	308	1462	951
71	625	438	452	832	964
100	800	642	665	774	963

Table V. Comparison of analytic model & simulation results
for b+ tree to linear hash file conversion

unit	breakpoint (pages)		% diff.
	simulation	analytic model	
1	46	46	0.00
5	49	50	2.00
10	53	54	1.85
15	58	59	1.69
20	62	64	3.13
25	67	67	0.00
30	70	71	1.41
35	75	77	2.60
40	82	84	2.38
50	89	88	-1.14
71	97	101	3.96
100	120	129	6.98

In the linear hash file to B+ tree conversion simulation, we varied the range query size as well as the various transaction type probabilities so that we could see the effect that they have on the breakpoint. The results of the various simulation runs are shown in tables VI, VII and VIII. For this conversion problem, we were primarily interested in the comparison of the simulation results with the

analytic model results.

The first observation which we can make is that by increasing the probability of a range query for a given range size, the breakpoint occurs sooner. However, this appears to be true up to some point. For example, in tables VII and VIII, we see that, for the simulation, there is no difference when the breakpoint occurs when we have 50% or 60% range queries. This is probably due to a majority of the queries having to access records from both the old linear hash file as well as the new B+ tree file, as opposed to just accessing records in the B+ tree file which would be more efficient. Another point which we observe is that by increasing the range size, for a fixed transaction probability, the breakpoint also occurs sooner. For example, compare each row of table VI with the corresponding row of table VII. One last observation is that there is not much difference when the range size is between 5 and 50 as compared with a range size between 5 and 100. Again, an explanation for this is that range queries are accessing records from both file structures during the conversion process.

Looking at the simulation results in table VI, we see that in the first row, the breakpoint occurs when 42% of the linear hash file has been converted. If we examine the last row, we see that the breakpoint occurs when only 30% of the file has been converted. For table VII, the breakpoint varies from having 29% of the file converted to having 27% of the file converted. Likewise, for table VIII, the breakpoint ranges from 30% to 26%, as the percentage of linear hash file converted.

Comparing our analytic model with the simulation, we see that the maximum difference is approximately 11% (table VI), and the average difference is about 5% (table VI), 3% (table VII) and 4% (table VIII). So, we see that the analytic model is a fairly good reflection of the simulation.

Table VI. Comparison of analytic model & simulation results for linear hash file to B+ tree conversion ($5 \leq$ range size ≤ 25)

prob	breakpoint (pages)		% diff.
	simulation	analytic model	
rrpr= .3 erpr= .5	54	56	3.57
rrpr= .4 erpr= .4	49	49	0.00
rrpr= .5 erpr= .3	42	45	6.67
rrpr= .6 erpr= .3	38	43	11.63

Table VII. Comparison of analytic model & simulation results for linear hash file to B+ tree conversion ($5 \leq$ range size ≤ 50)

prob	breakpoint (pages)		% diff.
	simulation	analytic model	
rrpr= .3 erpr= .5	37	39	5.13
rrpr= .4 erpr= .4	34	36	5.56
rrpr= .5 erpr= .3	35	35	0.00
rrpr= .6 erpr= .3	35	34	-2.94

Table VIII. Comparison of analytic model & simulation results for linear hash file to B+ tree conversion ($5 \leq$ range size ≤ 100)

prob	breakpoint (pages)		% diff.
	simulation	analytic model	
rrpr= .3 erpr= .5	38	37	2.70
rrpr= .4 erpr= .4	34	36	5.56
rrpr= .5 erpr= .3	33	35	5.71
rrpr= .6 erpr= .3	33	34	2.94

7. CONCLUSION

The motivation for this work has been to show that the B+ tree to linear hash file conversion and the linear hash file to B+ tree conversion can be done concurrently with user transaction processing. The conversion is necessary for improving the performance of the database system and doing the conversion concurrently with database usage is necessary for any system which must be available 24 hours a day. We devised two algorithms which perform the two conversions and introduced an analytic model for each conversion. We also used a typical database simulation model and ran various experiments. The results of the experiments support the idea of doing file conversion concurrently with database usage, especially when compared to an off-line reorganization approach.

REFERENCES

1. Agrawal, R., Carey, M. and Livny, M., "Models for Studying Concurrency Control Performance: Alternatives and Implications," 1985 SIGMOD Conference Proceedings, ACM, May 1985, 108-121.

2. Astrahan, M., et al., "System R: Relational Approach to Database Management," ACM TODS, 1, 2, June 1976, 97-137.

3. Comer, D., "The Ubiquitous B-Tree," Computing Surveys, ACM, 11, 2, June 1979, 121-137.

4. Ellis, C., "Concurrency and Linear Hashing," Technical Report, Computer Science Dept., University of Rochester, NY, March 1985.

5. Kim, W., "Highly Available Systems for Database Applications," Computing Surveys, ACM, 16, 1, March 1984, 71-98.

6. Knuth, D., The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, Reading Mass., 1969.

7. Kronsjo, L., Algorithms: Their Complexity and Efficiency, John Wiley and Sons, New York, 1979.

8. Kwong, Y. and Wood, D., "A New Method for Concurrency in B-trees," IEEE TSE, 8, 3, May 1982, 211-222.

9. Lang, S., Driscoll, J. and Jou, J., "Improving the Differential File Technique via Batch Operations for Tree Structured File Organizations," 2nd International Conference on Data Engineering, Los Angeles, 1986, 524-532.

10. Larson, P., "Linear Hashing with Partial Expansions," 1980 VLDB Conference Proceedings, 1980, 224-232.

11. Larson, P., "Linear Hashing with Overflow Handling by Linear Probing," ACM TODS, 10, 1, March 1985, 75-89.

12. Lehman, P. and Yao, S., "Efficient Locking for Concurrent Operations on B-trees," ACM TODS, 6, 4, Dec. 1981, 650-670.

13. Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," 1980 VLDB Conference Proceedings, 1980, 212-223.

14. Omiecinski, E., "Concurrent Storage Structure Conversion: from B+ Tree to Linear Hash File," 4th International Conference on Data Engineering, Los Angeles, 1988, 589-596.

15. Omiecinski, E., "Incremental File Reorganization Schemes," 1985 VLDB Conference Proceedings, 1985, 346-357.

16. Omiecinski, E., "Concurrency During the Reorganization of Indexed Files," 1985 COMPSAC Proceedings, 1985, 482-488.

17. Ouksel, M. and Scheuermann, P., "Storage Mappings for Multidimensional Linear Dynamic Hashing," 1983 PODS Conference Proceedings, ACM, 1983, 90-105.

18. Ramamohanarao, K. and Lloyd, J., "Dynamic Hashing Schemes," The Computer Journal, 25, 4, 1982, 478-485.

19. Ramamohanarao, K. and Sacks-Davis, R., "Recursive Linear Hashing," ACM TODS, 9, 3, Sept. 1984, 369-391.

20. Reingold, E., Nievergelt, J. and Deo, N., Combinatorial Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1977.

21. Ries, D. and Stonebraker, M., "Locking Granularity Revisited," ACM TODS, 4, 2, June 1979, 210-227.

22. Sockut, G. and Goldberg, R., "Database Reorganization - Principles and Practice," ACM Computing Surveys, 11, 4, Dec. 1979, 371-395.

23. Soderlund, L., "Concurrent Database Reorganization - Assessment of a Powerful Technique through Modeling," 1981 VLDB Conference Proceedings, 1981, 499-509.

24. Stonebraker, M., Wong, E., Kreps, P. and Held, G., "The Design and Implementation of INGRES," ACM TODS, 1, 3, Sept. 1976, 189-222.

A Heuristic File Reorganization Algorithm based on Record Clustering

Peter Scheuermann and Young Chul Park
Department of Electrical Engineering
and Computer Science
Northwestern University
Evanston, Illinois 60201

Edward Omiecinski*
School of Information & Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

*The research of this author was partially supported by the National Science Foundation under grant IST-8696157.

ABSTRACT

The problem of file reorganization which we consider involves altering the placement of records on pages of a secondary storage device. In addition, we want this reorganization to be done in-place, i.e., using the file's original storage space for the newly reorganized file. The motivation for such a physical change is to improve the database system's performance. For example, by placing frequently and jointly accessed records on the same page or pages, we can try to minimize the number of page accesses made in answering a set of queries. The optimal assignment (or reassignment) of records to clusters is exactly what record clustering algorithms [1,4,9] attempt to do. However, record clustering algorithms usually do not solve the entire problem, i.e., they do not specify how to efficiently reorganize the file to reflect the clustering assignment which they determine. Our algorithm is a companion to general record clustering algorithms since it actually transforms the file. The problem of optimal file reorganization is NP-hard [3]. Consequently, our reorganization algorithm is based on heuristics. The algorithm's time and space requirements are reasonable and its solution is near optimal. In addition, the reorganization problem which we consider in this paper is similar to the problem of join processing when indexes are used [2,5].

1. INTRODUCTION

One of the ways in which the performance of a database system can be improved is through reorganization based on record clustering. In this paper, we deal with a single file, containing multi-attribute records which are grouped into pages (i.e, blocks) in secondary storage. The target records of a multi-attribute query will usually be distributed over the entire file space and the number of pages accessed in the file can be as many as the number of target records for the query [7]. Thus, the objective of record clustering and reorganization is to reduce the expected number of page accesses. The result of a query Q_i is the set of records, in the file, which satisfy the conditions specified in the query. By using some directory (or directories), it is possible to access only those pages containing one or more records in the answer of the query. A common measure of the efficiency of answering a query is the number of page accesses required to retrieve the desired records [7]. If each page retrieved, in

response to a query, contains only one record, then many pages will be accessed. On the other hand, if each retrieved page contains many records, then few pages will be accessed.

Record clustering algorithms [1,4,10] assign frequently and jointly accessed records to the same page(s) in such a way that the total number of pages accessed in response to a set of queries, with some probability of occurrence, can be minimized. Record clustering algorithms usually focus on finding an optimal clustering but stop short of specifying an optimal procedure for rearranging the records, in the file, to match the clustering [9,10]. We consider this rearrangement of records to be our file reorganization problem. The reorganization process involves bringing pages from the file into a main memory buffer area, constructing new pages by rearranging records on the pages in the buffer and writing new pages back to the file. The cost of reorganization equals the number of pages that must be transferred from (to) secondary storage to (from) main memory. The problem of optimal file reorganization has been shown to be NP-hard [3]. Therefore, our approach will be to develop a heuristic algorithm whose time and space requirements are reasonable and whose solution is near optimal.

We should mention that our reorganization problem is similar to the problem of scheduling page fetches for a relational join operation when indexes are used in processing the join [2,5]. For both problems, a sequence of page fetches must be determined so as to minimize the total number of page accesses, either in computing the join or in performing the reorganization. The main difference is that with file reorganization, we have to rearrange the records. Thus, when we have the necessary pages in the buffer, we must construct the new page and write it to secondary storage. This means that other records will also be moved. Some of those records may not be associated with any cluster, i.e. those records may reside on any page in the file. However, every page in the buffer must eventually be written back to secondary storage. This would cause additional page accesses for reorganization as opposed to join processing, since for join processing, nothing has to be done with records in the buffer that do not join with any other records. From the join problem perspective, we can think of a cluster as consisting of a record from one of the joining relations and several records from the other joining relation, where the records in the cluster agree on the joining attribute. The records that join together (and their addresses) can be easily found by searching the two indexes.

2. PREVIOUS RESEARCH ON FILE REORGANIZATION

In [1], a clustering algorithm, which is tightly coupled to the reorganization, is presented. Their clustering approach is to sort the records of the file in ascending order of the concatenated key. The concatenated key $A_1, A_2, \ldots, A_k$ is selected such that the attributes appear in nonincreasing order of their probability of appearence in a query. Since sorting is expensive, i.e., producing a total ordering of all records, they propose a partial-sort method that restricts the sorting to those records that can fit into the buffer. So the reorganization is efficient but is limited to their clustering method. In addition, experimental evidence [4,10] has shown that the partial-sort clustering method performs very poorly.

In [10], an adaptive record clustering scheme is introduced. They present an elegant as well as conceptually simple clustering algorithm. Their algorithm does not classify queries into types nor does it collect individual query statistics. Preliminary experiments has shown very good results [10]. Once the clusters have been determined, they assign reocrds in each cluster, i.e., from the first cluster to the last cluster, to pages. They have a few cases which they consider when deciding whether a new page or an existing page can be used. We consider this a logical assignment of records, i.e., some logical page should contain a certain set of records. (This is what our algorithm will expect as input.) We do not consider this an effective approach for reorganization for the following reasons.

1) Reorganization is constrained by the size of the main memory buffer area.
2) The actual numbering of the clusters is somewhat arbitrary and as such, the set of pages which currently store records from $cluster_{i-1}$ and $cluster_i$ may be disjoint. Thus, the pages which are brought into the buffer when constructing $cluster_{i-1}$ will be of no use when constructing $cluster_i$. This could cause an excessive number of page faults.

Another approach for file reorganization based on record clustering is the dynamic_cost_reorganization algorithm [3]. However, as we will show, there exist some problems and limitations with this method. Since we want to later compare our algorithm with the dynamic_cost_reorganization algorithm, we will briefly review it here. The algorithm assumes that input and output to the secondary storage device is accomplished by using a main memory buffer area of fixed size. Two mappings are required as input, one is PG which corresponds to the old (file) state and

the other is NPG which corresponds to the new (file) state. These mappings satisfy the following:

$$PG : R \rightarrow P$$

$$NPG : P' \rightarrow 2^{R'}$$

where R = set of record identifiers,

P = set of physical page numbers,

P' = set of logical page numbers and

$2^{R'}$ = set of subsets of R of size $\leq$ pagesize.

To implement the mapping PG, there is a Page_table which associates with every record identifier, the physical page number on which it resides. This introduces another level of indirection between any directory (index) and the data file, but has the advantage that moving records within the data file does not affect the directory. Normally, in a tree structured directory [6,7], the leaves contain pointers to records. In this case, the leaves contain record identifiers. As previously mentioned, the output of a record clustering algorithm is assumed to be a set of logical pages: 1,2,...,M where each logical page contains at least one record and at most pagesize records. The records for each logical page are mutually exclusive such that $NPG(i) \cap NPG(j) = \emptyset$ for $1 \leq i,j \leq M$ and $i \neq j$. Logical page 0 is a special case and is the set of records in R which are not related to any logical page i, where $1 \leq i \leq M$. Given the above two mappings, PG and NPG, the mapping $D : P' \rightarrow P$ is defined as follows: $D_k = \{PG(r) \mid r \in NPG(k)\}$ where $k \in P'$. Thus D_k gives the set of physical pages which contain record(s) of logical page k. Buffer is the set of physical pages currently residing in the buffer. D'_k is the set of pages containing record(s) of logical page k and currently residing in secondary storage, i.e., $D'_k = D_k$ - Buffer. The dynamic_cost_reorganization algorithm of [3] is shown below.

```
Algorithm: dynamic_cost_reorganization
    Begin
        While all the logical pages are not constructed do
        Begin
            1: Determine the current logical page, clp, based on the cost function
               function cost(k) = |D'k |/ |Dk |;
            2: Determine physical pages to be swapped from the buffer using the
               fewest_records buffer paging policy;
```

3: Bring in D'_{clp} physical pages;
4: Rearrange the records in the buffer such that all the records which make up clp are contained on a single physical page;
5: Write the physical page which contains clp to secondary storage;
6: Rearrange records in the buffer such that records which belong to the same logical page are grouped together as follows:
   a) for each logical page, k, which has records in the buffer, set $S_k = \{r \mid r \in NPG(k) \text{ and } PG(r) \in Buffer\}$;
   b) Order the above sets, excluding S_0, by nonincreasing size to obtain $S_1', S_2', ..., S_n'$;
   c) Allocate the sets in order: $S_1', ..., S_n'$, and s_0 to buffer pages in order $1,2,...,|Buffer|$;

End;

End;

The fewest_records buffer replacement policy (step 1.2) is to select the pages in reverse order of consolidation, i.e., page[|Buffer|],...,page[1], where page[x] represents the page in position x in the buffer. However, the dynamic_cost_reorganization algorithm suffers from the following problems and/or limitations.

1. The buffer capacity in pages must not be less than the page size in records, otherwise, the algorithm does not work in some cases.
2. If there is not enough buffer space to bring in D'_{clp} physical pages, several physical pages in the buffer are written to disk to make enough free buffer space. It may be inefficient to write out those physical pages at once before trying to construct clp, the current logical page, with the given buffer status.
3. If the number of records of clp is less than the pagesize, then other records are put on the page, e.g. page p, which is written to disk. If some of the records of another logical page k are contained on physical page p, then for the construction of logical page k, page p has to be brought into the buffer again. The construction of logical page clp may be destroyed.
4. Some logical pages do not need to be processed. For example, if there is only one physical page which contains records for logical page k, then logical page k does not need to be processed.

In the next section, we present our cluster_reorganization algorithm which works for any size of main memory buffer (i.e., ≥ 2 pages) and solves the previously mentioned problems. We assume a

paged buffer system as well as the existance of a Page_table. The same mapping definitions and same notations will be used to enhance the ease of cross-referencing with the dynamic_cost_reorganization algorithm [3].

3. EFFICIENT FILE REORGANIZATION

This section deals with the problem of efficiently reorganizing the file, for a given record clustering. We assume that the dominating cost for reorganization is that incurred by page accesses to/from secondary storage [7]. Input and output to the secondary storage device is accomplished by using a main memory buffer area of fixed size. Thus, our goal is not only to reorganize the file but to minimize the number of pages swapped in and out of the buffer during the reorganization process. As mentioned, the mappings PG and NPG are used. Buffer, D_k, D'_k and clp are defined exactly as in the previous section. We need to define one more mapping, $LPN : R \rightarrow P'$ which is defined as $LPN(r) = k$, where $r \in NPG(k)$ such that LPN gives the logical page number for the corresponding record. The implementation of the mapping LPN can be accomplished by using a hash table. Before we explain our cluster_reorganization algorithm, we need to present the following definitions.

1) A **void_record** r is a record which is not related to any logical page, i.e., $LPN(r) = 0$.

2) A **nonvoid_record** r is a record which is related to any one of the logical pages, i.e., $LPN(r) = k$ where $1 \leq k \leq M$.

3) A **void_physical** page p is a physical page that consists only of void_records, i.e., for each record $r \in p$, $LPN(r) = 0$.

4) A **perfect_physical** page p is a physical page where for each record $r \in p$, $LPN(r) = k$ and $k \neq 0$. Hence $d_k = \{p\}$.

5) A **composite_physical** page p is a physical page which contains one or more logical pages and there exists a record $r \in p$ such that $LPN(r) = k$, $k \neq 0$ and $D_k - \{p\} \neq \emptyset$.

6) A logical page is in the **ready_state** if all the records related to that logical page reside in the buffer.

7) A logical page is in the **perfect_state** if all the records related to that logical page reside

in a perfect physical page.

8) A logical page k is in the **post_ready_state** if $D'_k = D'_{clp}$ and $k \neq clp$.

The following notations are also used:

X = current available buffer space, i.e., Buffer_capacity - |Buffer|,

N_p = number of nonvoid_records in physical page p,

TVR = total number of void_records in the buffer,

RS = set of logical pages which are in the ready_state,

PS = set of logical pages which are in the post_ready_state,

MAX = subset of RS whose combined number of records for each constituent logical page $\leq$ pagesize and maximum among other subsets,

|MAX| = the number of constituent logical pages for set MAX and

SIZE = the combined number of records for each constituent logical page for set MAX.

To bring clp into the ready_state, it is necessary to bring in D'_{clp} physical pages from secondary storage into the buffer. If there is space in the buffer (i.e., $X \neq 0$), there is no problem in bringing in a physical page from secondary storage. However, if $X = 0$ and $D'_{clp} \neq \emptyset$, i.e., $X = 0$ while there are some physical pages which have to be brought into the buffer, then to bring in the next physical page in D'_k, a page frame needs to be made available in the buffer. In this situation, to free a page frame, the following steps are used:

step 1: Try to construct a perfect_physical page

step 2: If step 1 is not possible then try to construct a void_physical page

step 3: If steps 1 & 2 are not possible then build a composite_physical page

step 4: Write the above constructed physical page to secondary storage

When step 3 is satisfied, we call it overflow. However, if $X = 0$ and $D'_k = \emptyset$, i.e., if $X = 0$ after bringing in all the physical pages in D'_k into the buffer, then to continue the reorganization process, we also need to follow the above steps. In this case, if step 3 is satisfied, it is called underflow. We should note that a composite_physical page which is written to disk will eventually be read back into

the buffer.

Once clp is brought into the ready_state and SIZE plus TVR is greater than or equal to the pagesize, then it is possible to make a perfect_physical page. However, to reduce the possibility of overflow or underflow when the next logical page is constructed, the following two steps are used.

step A: While |MAX| = 1 & SIZE = pagesize do
construct a perfect physical page using records of logical page in MAX

step B: If X = 0 then follow the above 4 steps for freeing a page frame.

Overflow and underflow are not promising situations since they propagate additional page accesses. So the reorganization algorithm must try to minimize the possibility of those situations. The possibility of overflow/underflow is directly related to the order of logical page construction and by the access sequence of those physical pages related to each logical page. The sequence of bringing in those physical pages which are related to each logical page will be discussed first. Afterwards, the dynamic order of logical page construction, as directed by our cost function, will be shown. To bring logical page k into the ready_state, we must bring in D'_k physical pages from secondary storage into the buffer. Let the result of sorting those physical pages in D'_k in nondecreasing order of nonvoid_records in each physical page be denoted as the bring_in_sequence, BIS. There are three observations concerned with overflow and underflow conditions which are related to BIS. These observations are proved in the appendix and provide some measure of the "goodness" of our algorithm. For these observations and for the cost function which will be explained next, the following notations are used.

$$Y_k = \max(N_1, N_2, \ldots, N_m) \text{ where } \{1,2,\ldots,m\} = D'_k$$

$$Z_k = \sum_{i=1}^{m} N_i - Y_k$$

Observation 1: If BIS of D'_k causes overflow when bringing logical page k into the ready_state, then any other sequence also causes overflow.

Observation 2: If (X - 1) * pagesize + TVR $\geq Z_k$ then logical page k does not cause overflow by using BIS.

Observation 3: If X $\geq$ 2 and (X - 2) * pagesize + TVR $\geq Z_k$ then logical page k does not cause

overflow or underflow by using BIS.

Because of overflow and underflow, in some cases it is possible to be faced with the situation where some physical pages are brought into the buffer and written out repeatedly without making any progress. This condition, which causes an infinite looping, is referred to as thrashing. Bringing clp into the ready_state might cause thrashing. Checking whether this will happen is very difficult. So, instead of checking thrashing, it is much easier to check whether clp and the current buffer contents contain the possibility of causing thrashing. Thus, the thrashing_possibility (TP) is defined as true if clp causes overflow or underflow in the presence of at least one ready_state logical page. Therefore, once clp is decided according to the dynamic cost function which will be explained later, TP of clp is always checked before trying to bring clp into the ready_state. The algorithm for TP is given below.

```
Algorithm thrashing_possibility;
/* Let D'clp = {1,2,...,m} and the bring_in_sequence be 1,2,...,m */
Begin
    TP := false;
    If RS ≠ ∅ and X ≤ m
    Then
        Begin
            E := TVR + (pagesize - N1) + ... + (pagesize - NX-1);
            CRS := RS;
            i := X;
            While ¬ TP and i ≤ m and CRS ≠ ∅ Do
            Begin
                E := E + pagesize - Ni;
                If i = m Then CRS := CRS ∪ PS ∪ clp;
                find MAX and SIZE for set CRS;
                CRS := CRS - MAX;
                E := E - (pagesize - SIZE);
                If E ≥ 0 Then i := i + 1
                Else
                    Begin
                        TP := true;
                        PS := ∅ ;
                    End;
            End;
    End;
End;
```

When TP of clp is true, instead of bringing clp into the ready_state, we try to construct a perfect_physical page with logical pages in the ready_state and void_records in the buffer. If there is not a sufficient number of void_records in the buffer, then one physical page which has the maximum

number of void_records in secondary storage, needs to be brought into the buffer.

All the composite_pages on disk should be brought into the buffer at least once to finish the reorganization process. The number of pages needed to construct any logical page is not our concern. The ultimate objective is to minimize the total number of pages swapped in and out of the buffer. Based on the above three observations for the given buffer contents, the logical page which has the minimum possibility of causing overflow or underflow is the one which has minimum Z_k. Moreover, after bringing a logical page into the ready_state or perfect_state, we want to increase the probability of having sufficient buffer space. To accomplish this, we select a logical page which has minimum $Z_k + Y_k$ among logical pages which are not in either the perfect_state nor ready_state. Our cost function is defined as follows:

$$\text{COST}(k) = \text{cost to bring logical page k into the ready_state} = (Z_k, Y_k),$$

$$\text{COST}(i) < \text{COST}(j) \text{ if } Z_i < Z_j \text{ or } Z_i = Z_j \text{ and } Y_i < Y_j.$$

The above cost function takes into account the entire contents of the buffer. According to this cost function, our heuristic rule for the selection of clp is the following: find a logical page which has minimum Z_k value and if there is more than one logical page which has the same Z_k value, then select the one which has the minimum Y_k value. The purpose of our cost function, COST(k), is to provide a means for ranking those logical pages which are not in the perfect_state. This allows us to determine a sequence of constructing logical pages which will hopefully minimize the total number of disk accesses for reorganization. Based on the ideas discussed so far, our new heuristic rorganization algorithm is presented.

```
Algorithm cluster_reorganization;
Input: data file, NPG, PG, pagesize, buffer_capacity
Output: reorganized data file
Begin
    step 0: /* Initialization */
        X := Buffer_capacity;  Buffer := ∅ ;  RS := ∅ ;  TVR := 0;
        For each nonvoid_record r, determine LPN(r);
        For each physical page p, determine N_p;
        Find COST for each logical page & sort COST table in nonincreasing order;
        Find logical pages which are already in the perfect_state & delete them
          from the sorted COST table;
    step 1: /* Reorganization */
```

```
While the COST table is not empty do
Begin
    1.1: Find clp which has minimum cost among logical pages in the COST table
            & not in RS;
         If all the logical pages in the COST table are in RS then
            set clp to 0;
    1.2: If clp ≠ 0 and clp contains thrashing possibility (TS)
            then set clp to 0;
    1.3: if clp = 0 then
         Begin
            D'clp := ∅ ;
            Find MAX & SIZE for logical pages in RS;
            If SIZE + TVR < pagesize then
                Begin
                    Find a physical page p on disk which has a minimum number
                       of nonvoid_records;
                    If p is not a perfect_physical page and Np < pagesize or p is a
                       perfect_physical page and Np < SIZE
                    Then D'clp := {p}
                    Else
                        Begin
                            Select a logical page k whose number of records is maximum
                               among logical pages in RS;
                            For each record r of logical page k do
                            Begin
                                LPN(r) := 0;
                                TVR := TVR + 1;
                                NPG(r) := NPG(r) - 1;
                            End;
                            Delete k from the COST table;
                        End;
                End;
         End;
    1.4: While D'clp ≠ ∅ do
         Begin
            While X ≠ 0 and Dclp ≠ ∅ do
            Begin
                Bring in a physical page which has a minimum number of nonvoid_records
                   in D'clp;
                X := X - 1;
                Buffer := Buffer ∪ {p};
                D'clp := D'clp - {p};
                Examine each record r in physical page p, if necessary adjust TVR and
                   if LPN(r) enters the ready_state then RS := RS ∪ LPN(r);
            End;
            If clp ≠ 0 and D'clp = ∅
            Then
            Begin
                Find MAX and SIZE for logical pages in RS;
                While |MAX| = 1 and SIZE = pagesize do
                Begin
                    Construct & write out perfect_physical page p;
                    Delete logical page k in MAX from the COST table;
```

```
                    X := X + 1;
                    RS := RS - MAX;
                    Buffer := Buffer - {p};
                    Find MAX and SIZE for logical pages in RS;
                End;
            End;
            If clp = 0 or (clp ≠ 0 and X = 0 and RS ≠ ∅ )
            Then
            Begin
                Find MAX & SIZE for logical pages in RS;
                If SIZE + TVR ≥ pagesize
                Then
                Begin
                    Construct and write out perfect_physical page p
                      using records of logical pages in MAX and
                    (pagesize - SIZE) void_records;
                    Delete logical pages in MAX from the COST table;
                    X := X + 1;
                    RS := RS - MAX;
                    Buffer := Buffer - {p};
                    TVR := TVR - (pagesize - SIZE);
                End;
            End;
            If X = 0 and TVR ≥ pagesize
            Then
            Begin
                construct and write out void_physical page p;
                X := X + 1;
                Buffer := Buffer - {p};
                TVR := TVR - pagesize;
            End;
            If X = 0
            Then
            Begin
                Construct and write out composite_physical page p;
                X := X + 1;
                Buffer := Buffer - {p};
                Adjust TVR;
            End;
        End;
    1.5: For each logical page k whose cost has been changed & whose
         status is not the perfect_state do
         Find COST(k) and insert k into the proper position in the COST table;
    End;
step 2: /* Buffer clearance */
    For each physical page p in the Buffer do
    Begin
        write out physical page p to secondary storage;
        Buffer := Buffer - {p};
    End;
End;
```

To conclude this section, we want to make some remarks about the time and space complexity

of our algorithm. Let M be the total number of logical pages and N be the total number of records in the file. Since pagesize and buffer capacity are constants, it is easy to show that the worst case time complexity is O(MN) and moreover, if M = N / pagesize, then it is $O(N^2)$. As far as the worst case space complexity is concerned, the storage requirement is O(N).

4. A FILE REORGANIZATION EXAMPLE

To explain our cluster_reorganization algorithm more clearly, we provide the following example.

Example 1: Assume that the pagesize is 5 records, the buffer_capacity is 3 pages, and NPG and PG are given as follows:

NPG	PG
A: A_1 A_2 A_3 A_4 A_5	A_1 1 A_2 3 A_3 5 A_4 7
B: B_1 B_2 B_3	C_1 1 C_2 3 C_3 5 C_4 7
C: C_1 C_2 C_3 C_4	D_1 1 D_2 3 E_3 5 E_5 7
D: D_1 D_2	F_1 1 G_2 3 H_3 5 H_4 7
E: E_1 E_2 E_3 E_4 E_5	G_1 1 I_1 3 b_2 5 I_4 7
F: F_1 F_2 F_3	B_1 2 B_2 4 B_3 6 A_5 8
G: G_1 G_2 G_3 G_4	E_1 2 E_2 4 E_4 6 H_5 8
H: H_1 H_2 H_3 H_4 H_5	F_2 2 G_3 4 F_3 6 b_3 8
I: I_1 I_2 I_3 I_4	H_1 2 H_2 4 G_4 6 b_4 8
	b_1 2 I_2 4 I_3 6 b_5 8

For the above two mappings, A through I represent logical pages and A_1 through I_4 represent their corresponding record identifiers. The integer which follows each record identifier, in PG, indicates the physical page where the corresponding record resides. Each b_i represents a void_record. The cost for each logical page is calculated using the cost function, $COST(k) = (Z_k, Y_k)$. Those costs are sorted and are stored in the COST table as follows.

COST(k) : (5,5) (9,5) (9,5) (14,5) (15,5) (15,5) (15,5) (16,5) (18,5)

k : D B F C G H I A E

Logical page D becomes clp and to bring logical page D into the ready_state, physical pages 7 and 8 are brought into the buffer (Fig. 1a). Since MAX = {D} and SIZE + TVR < pagesize, we can not bring logical page D into the perfect_state.

BUFFER	BUFFER	BUFFER
1 A_1 C_1 D_1 F_1 G_1	1 A_1 C_1 D_1 F_1 G_1	1 A_1 C_1 A_5 F_1 G_1
3 A_2 C_2 D_2 G_2 I_1	3 A_2 C_2 D_2 G_2 I_1	3 A_2 C_2 H_5 G_2 I_1
	8 A_5 H_5 b_3 b_4 b_5	8 D_1 D_2 b_3 b_4 b_5
TVR=0 RS={D}	TVR=3 RS={D}	MAX={D} SIZE=2
(a)	(b)	(c)

Figure 1: Buffer Contents

The updated costs for logical pages are as follows:

COST(k) : (0,0) (4,5) (5,4) (5,5) (6,5) (9,5) (10,5) (15,5) (18,5)

k : D C F G A B I H E

According to the COST table, logical page C becomes clp. $D'_c = \{5,7\}$ and the bring_in_sequence is 5 and 7. However, clp causes overflow since logical page D is in the ready_state. Therefore, TP (thrashing possibility) becomes true and clp is set to 0. Since SIZE + TVR < pagesize, we find a physical page which has the maximum number of void_records, i.e., physical page 8. Now, physical page 8 is brought into the buffer (Fig. 1b), records in the buffer are rearranged (Fig. 1c) and the perfect_physical page 8 is written out to secondary storage. Now, the updated COST table is as follows.

COST(k) : (4,5) (4,5) (5,4) (5,5) (9,5) (10,5) (13,5) (18,5)

k : A C F G B I H E

At this point, logical page A becomes clp. To bring logical page A into the ready state, physical pages in D'_A = {5,7} have to be brought into the buffer. Physical page 5 is brought into the buffer first because N_5=4 and N_7=5 (Fig. 2a). Since X=0 and it is not possible to make a perfect physical_page or a void_physical page, we must construct a composite_physical page. The specified records are placed on physical page 1 (Fig. 2b) which is then written out to disk. Physical page 7 is then brought into the buffer (Fig. 2c).

BUFFER	BUFFER	BUFFER
1 A_1 C_1 A_5 F_1 G_1	1 E_3 H_3 H_5 I_1 G_1	7 A_4 C_4 E_5 H_4 I_4
3 A_2 C_2 H_5 G_2 I_1	3 A_2 C_2 A_5 G_2 F_1	3 A_2 C_2 A_5 G_2 F_1
5 A_3 C_3 E_3 H_3 b_2	5 A_3 C_3 A_1 C_1 b_2	5 A_3 C_3 A_1 C_1 b_2
TVR=1	TVR=1	TVR=1
RS=∅	RS=∅	RS={A,C}
(a)	(b)	(c)

Figure 2: Buffer Contents

Now, logical pages A and C are in the ready_state. Logical page A and C contain 5 and 4 records respectively. The records of logical page A are collected on physical page 7 (Fig 3a) and written out to disk.

BUFFER	BUFFER	BUFFER
7 A_4 A_1 A_2 A_3 A_5		2 B_1 E_1 F_2 H_1 b_1
3 E_5 C_2 I_4 G_2 F_1	3 E_5 H_4 I_4 G_2 F_1	3 E_5 H_4 I_4 G_2 F_1
5 H_4 C_3 C_4 C_1 b_2	5 C_2 C_3 C_4 C_1 b_2	6 B_3 E_4 F_3 G_4 I_3
TVR=1	TVR=1	TVR=1
MAX={A}	MAX={C}	RS={F}

(a) (b) (c)

Figure 3: Buffer Contents

The updated COST table is as follows.

COST(k) :	(0,0)	(5,4)	(9,5)	(9,5)	(10,5)	(10,5)	(14,5)
k :	C	F	H	B	G	I	E

At this point, logical page F becomes clp where D'_F={2,6}. However, clp causes underflow in the presence of logical pages in the ready_state. Thus, clp is set to 0. Due to the fact that SIZE + TVR $\geq$ pagesize, a perfect_physical page must be constructed from records of logical page C and one void_record b_2 (Fig. 3b). Physical page 5 is written to disk. Now, the only necessary change to the COST table is to delete the cost for logical page C. Therefore, clp={F} and physical pages in D'_F, i.e., {2,6}, are brought into the buffer (Fig. 3c). Now, because X=0, TVR=1, and SIZE=3; the reorganization process continues by making a composite_physical page. Page 2 is constructed (Fig. 4a) and is written to disk.

BUFFER	BUFFER	BUFFER
2 E_4 E_1 E_5 G_2 I_4	8 D_1 D_2 b_3 b_4 b_5	8 D_1 D_2 F_1 F_2 F_3
3 F_2 H_4 b_1 H_1 F_1	3 F_2 H_4 b_1 H_1 F_1	3 b_4 H_4 b_1 H_1 b_3
6 B_3 B_1 F_3 G_4 I_3	6 B_3 B_1 F_3 G_4 I_3	6 B_3 B_1 b_5 G_4 I_3
TVR=1 RS={F}	TVR=4 RS={D,F}	TVR=4 SIZE=5
(a)	(b)	(c)

Figure 4: Buffer Contents

The updated COST table is as follows.

COST(k) : (0,0) (0,5) (5,5) (10,5) (10,5) (10,5)

k : F B H G I E

The current logical page, clp, is set to logical page B where $D'_B=\{4\}$. However, clp causes overflow in the presence of the ready_state logical page so clp is set to 0 and because SIZE + TVR < pagesize, physical page 8 which has the maximum number of void_records is brought into the buffer (Fig 4.b). Since MAX={D,F} and SIZE=5, records from logical page D and F are collected on physical page 8 (Fig. 4c) which is written to disk. Now, the updated COST table is the same as the previous COST table except that the cost for logical page F has been deleted. The current logical page is set to B and physical page 4 is brought into the buffer (Fig. 5a). Records of logical page B and two void_records are collected on physical page 6 (Fig. 5b) which is written to disk. This is done since X=0 and SIZE + TVR $\geq$ pagesize.

(a)	(b)	(c)
BUFFER	BUFFER	BUFFER
4 B_2 E_2 G_3 H_2 I_2	4 b_5 E_2 G_3 H_2 I_2	4 b_5 E_2 G_3 H_2 I_2
3 b_4 H_4 b_1 H_1 b_3	3 b_4 H_4 G_4 H_1 I_3	3 b_4 H_4 G_4 H_1 I_3
6 B_3 B_1 b_5 G_4 I_3	6 B_3 B_1 B_2 b_1 b_3	1 E_3 H_3 H_5 I_1 G_1
TVR=4	TVR=4	TVR=2
RS={B}	SIZE=3	RS={H}

Figure 5: Buffer Contents

The updated COST table is as follows.

COST(k) : (0,5) (5,5) (5,5) (5,5)

k : H G I E

Logical page H becomes clp and $D'_H=\{1\}$. To bring logical page H into the ready_state, physical page 1 is brought into the buffer (Fig. 5c) and records from logical page H are collected on physical page 3 (Fig. 6a) which is written to disk.

BUFFER	BUFFER	BUFFER
4 b_5 E_2 G_3 b_4 I_2	4 b_5 E_2 G_3 b_4 I_2	4 b_5 G_2 G_3 b_4 I_2
3 H_2 H_4 H_3 H_1 H_5	2 E_4 E_1 E_5 G_2 I_4	3 E_4 E_1 E_5 E_2 E_3
1 E_3 G_4 I_3 I_1 G_1	1 E_3 G_4 I_3 I_1 G_1	1 I_4 G_4 I_3 I_1 G_1
TVR=2 SIZE=5	TVR=2 RS={E,G,I}	
(a)	(b)	(c)

Figure 6: Buffer Contents

The updated COST table is shown below and logical page G becomes clp with $D'_G=\{2\}$.

COST(k) : (0,5) (0,5) (0,5)

k : G I E

Now physical page 2 is brought into the buffer (Fig. 6b) and because MAX={E} and SIZE=5, records of logical page E are collected on physical page 2 (Fig. 6c). Physical page 2 is written out to disk and the COST table becomes the following.

COST(k) : (0,0) (0,0)

k : G I

Now, all the logical pages which are not in the perfect_state are in the ready_state so clp becomes 0. Since MAX={G}, SIZE=4 and TVR=2; we use the 4 records of logical page G and 1 void record to construct perfect_physical page 1 (Fig. 7a) which is written to disk. The updated COST table is the same as the above COST table except that the cost for logical page G has been deleted. For the same reason as stated above, we can use the records of logical page I and 1 void_record to construct physical_page 4 (Fig. 7b) which gets written to secondary storage.

BUFFER	BUFFER	BUFFER
4 b_5 G_2 G_3 G_1 G_4		
1 I_4 b_4 I_3 I_1 I_2	1 I_4 b_4 I_3 I_1 I_2	
TVR=2 SIZE=4	TVR=1 SIZE=4	
(a)	(b)	(c)

Figure 7: Buffer Contents

Now, the buffer is empty so the reorganization process is finished. The total number of disk accesses is 22 and all logical pages are in the perfect_state.

In closing this section, we want to mention that we have found an example where transforming each logical page into a perfect_state is not possible. This problem is due to a lack of void records in the file in conjunction with our requirement of doing an in-place reorganization.

5. EXPERIMENTAL RESULTS FOR FILE REORGANIZATION

In this section, we present the results of a number of file reorganization experiments. For each experiment, we compare our cluster_reorganization algorithm with the dynamic_cost_reorganization algorithm [3]. In [3], it was shown, for certain assumptions (which we are not restricted to), that the dynamic_cost_reorganization algorithm makes approximately 40% fewer page accesses than a reorganization strategy that uses a linear ordering. Linear ordering means that logical pages are converted into perfect_physical pages in order of their logical page names (or numbers), e.g., A,B,C,... (or 1,2,3,...). In our experiments, we randomly generate records for 25 logical pages where the pagesize is 10 records. The record identifiers for the file are in the range from 1 to 1000. Tables 1, 2 and 3 represent the experimental results for a buffer capacity of 7, 10 and 20 pages respectively. For each fixed size buffer capacity, the size of a logical page is determined as a percentage of the size of the physical page. It varies from 1% to 100% in some cases, 50% to 100% in others and is set to 100% in the remaining

cases. Another parameter in our experiments is the hit ratio, i.e., the percentage of physical pages containing relevant logical pages. In our experiments, the hit ratio varies from 30% to 100%. In the third and fourth columns in each table, the number of disk accesses necessary for the reorganization is shown for cluster_reorganization and dynamic_cost_reorganization algorithms, respectively. In the last column, we show the difference of disk accesses by (DYN - CLU) / DYN, where CLU and DYN represent the number of disk accesses made by cluster_reorganization and dynamic_cost_reorganization algorithms, respectively. We also compared the correctness of each reorganization algorithm by checking the number of logical pages which were not constructed correctly. Those numbers are represented in the fifth and sixth columns in each table for cluster_reorganization and dynamic_cost_reorganization, respectively. In each table, *** means, for the specific input and parameter values, that the corresponding algorithm does not work.

From our experiments, we find that cluster_reorganization generates a correct result for any value of buffer capacity, hit ratio and logical page size. Moreover, the number of disk accesses required for cluster_reorganization is much less than that of dynamic_cost_reorganization and when the buffer capacity is greater than or equal to the pagesize, the number of disk accesses is mostly optimal. Due to the assumptions made by the dynamic_cost_reorganization algorithm, in general it will not work correctly (or possibly not terminate) when the buffer capacity is less than the pagesize or when the logical page size does not equal the physical page size. If it does terminate in either of those cases, as evidenced in our experiments, around 30% of the logical pages are not correctly constructed. Considering those results, we find that our cluster_reorganization algorithm is superior to the dynamic_cost_reorganization algorithm since our algorithm is more general and makes fewer page accesses.

Table I. Buffer capacity = 7 pages

logical pagesize in %	hit ratio	page accesses		logical pages not made by		difference in %
		CLU	DYN	CLU	DYN	
1_100	30	60	***	0	***	***
1_100	40	80	114	0	5	29.8
1_100	47	94	***	0	***	***
1_100	56	112	144	0	6	22.2
1_100	66	132	162	0	8	18.5
1_100	74	146	***	0	***	***
1_100	83	166	***	0	***	***
50_100	30	66	104	0	9	36.5
50_100	40	84	122	0	9	31.1
50_100	50	102	144	0	12	29.2
50_100	58	116	152	0	8	23.7
50_100	65	130	168	0	6	22.6
50_100	69	138	170	0	6	18.8
50_100	84	168	200	0	7	16.0
100	30	80	88	0	0	9.1
100	40	94	114	0	0	17.5
100	50	110	122	0	0	9.8
100	60	130	***	0	***	***
100	79	164	***	0	***	***
100	88	180	***	0	***	***
100	96	194	***	0	***	***

Table II. Buffer capacity = 10 pages

logical pagesize in %	hit ratio	page accesses		logical pages not made by		difference in %
		CLU	DYN	CLU	DYN	
1_100	30	60	94	0	8	36.2
1_100	40	80	118	0	6	32.2
1_100	50	100	132	0	7	24.2
1_100	66	132	164	0	6	19.5
1_100	73	142	170	0	4	16.5
1_100	84	166	190	0	6	12.6
50_100	30	60	92	0	10	34.8
50_100	40	80	108	0	8	25.9
50_100	47	94	118	0	8	20.3
50_100	64	128	152	0	6	15.5
50_100	84	168	186	0	6	9.7
50_100	90	180	204	0	7	11.8
100	30	68	74	0	0	8.1
100	40	86	96	0	0	10.4
100	50	104	110	0	0	5.5
100	60	122	130	0	0	6.2
100	78	156	162	0	0	3.7
100	85	170	178	0	0	4.5
100	96	192	198	0	0	3.0

Table III. Buffer capacity = 20 pages

logical pagesize in %	hit ratio	page accesses		logical pages not made by		difference in %
		CLU	DYN	CLU	DYN	
1_100	30	60	92	0	7	34.8
1_100	38	76	116	0	6	34.5
1_100	48	96	124	0	9	22.6
1_100	52	102	136	0	8	25.0
1_100	62	124	156	0	8	20.5
1_100	78	156	180	0	8	13.3
50_100	30	60	88	0	8	31.8
50_100	40	80	108	0	8	25.9
50_100	50	100	116	0	5	13.8
50_100	60	120	152	0	9	21.1
50_100	72	144	168	0	8	14.3
50_100	86	172	198	0	10	13.1
100	30	60	60	0	0	0.0
100	40	80	80	0	0	0.0
100	50	100	100	0	0	0.0
100	60	120	120	0	0	0.0
100	69	138	138	0	0	0.0
100	80	170	170	0	0	0.0
100	96	192	192	0	0	0.0

6. CONCLUSION

In this work, we have developed an efficient heuristic algorithm for the problem of file reorganization which involves changing the placement of records on pages of secondary storage. The cluster_reorganization algorithm can be used for any size of main memory buffer area. Our algorithm utilizes heuristic functions to decide the reorganization sequence of logical pages and to decide the bring-in sequence of physical pages.

We have also considered the problem of overflow, underflow and thrashing which could occur during the reorganization period. We did a comparison with another approach [3] which showed that our algorithm is more general and caused fewer page faults during the reorganization. In addition, it generated the correct result for the given record clustering input with the same time and space complexity as the algorithm in [3].

REFERENCES

[1] M. Jakobsson, "Reducing Block Accesses in Inverted Files by Partial Clustering," Information Systems, Vol. 5, 1980, pp. 1-5.

[2] T. Merrett, Y. Kambayashi and H. Yasuura, "Scheduling of Page-Fetches in Join Operations," VLDB Conference Proceedings, Cannes, France, 1981, pp. 488-498.

[3] E. Omiecinski, "Incremental File Reorganization Schemes," VLDB Conference Proceedings, Stockholm, Sweden, 1985, pp. 346-357.

[4] E. Omiecinski and P. Scheuermann, "A Global Approach to Record Clustering and File Reorganization," in Research and Development in Information Retrieval, ed. C. J. van Rijsbergen, Cambridge Press, 1984, pp. 201-219.

[5] S. Pramanik and D. Ittner, "Use of Graph-Theoretic Models for Optimal Relational Database Accesses to Perform Join," ACM TODS, Vol. 10, No. 1, 1985, pp. 57-74.

[6] P. Scheuermann and M. Ouksel, "Multidimensional B-trees for Associative searching in Database Systems," Information Systems, Vol. 7, No. 2, 1982, pp. 123-137.

[7] T. J. Teory and J. P. Fry, Design of Database Structures, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[8] C. Yu and C. Chen, "Information System Design: One Query at a Time," ACM SIGMOD Conference Proceedings, Austin, Texas, 1985, pp. 280-290.

[9] C. Yu, K. Lam, M. Siu and C. Suen, "Adaptive Record Clustering," ACM TODS, Vol. 10, No. 2, 1985, pp. 180-204.

APPENDIX

Observation 1: If BIS of D'_k causes overflow when bringing logical page k into the ready_state, then any other sequence also causes overflow.

Proof: Let the result of sorting $D'_k=\{1,2,...,m\}$ in nondecreasing order of nonvoid_reocrds in each physical page in D'_k be 1,2,...,m. Let the number of nonvoid_records be $e_1,e_2,...,e_m$ in each physical page 1,2,...,m; respectively. If $m \leq X$, then no sequence causes overflow, so in the following two cases, only consider the situation where $M > X$.

Case I: $RS = \emptyset$

Because there is no logical page, in the ready_state, which is contained in the buffer, a perfect_physical page cannot be constructed until all the physical pages in D'_k are brought into the buffer. So, an overflow could occur only when X=0, the total number of void_records in the buffer is less than the pagesize, and one or more physical pages in D'_k are not in the buffer. Assume that n, $(n \leq X$ and $n < m)$, physical pages are brought into the buffer. Because $(e_i + e_{i+1} + ... + e_{i+n-1})$ is greater than or equal to $(e_j + e_{j+1} + ... + e_{j+n-1})$, where $i<j$, we know that if $(TVR + e_i + ... + e_{i+n-1})$ is less than the pagesize then $(TVR + e_j + ... + e_{j+n-1})$ is also less than the pagesize. This means that if BIS causes overflow then any other sequence also does.

Case II: $RS \neq \emptyset$

To make room in the buffer, if some number of logical pages are in the ready_state, there are enough void_records to construct a perfect_physical page and X=0; then this perfect physical_page will be written to disk. If $(TVR + SIZE + e_i + ... + e_{i+n-1})$ is less than the pagesize then $(TVR + SIZE + e_j + ... + e_{j+n-1})$ is also less than the pagesize. Therefore, if BIS causes overflow then any other sequence also does. □

Observation 2: If $(X-1) * \text{pagesize} + TVR \geq Z_k$, then logical page k does not cause overflow by using

BIS.

Proof: Let the result of sorting $D'_k=\{1,2,...,m\}$ in nondecreasing order of nonvoid_records in each physical page be 1,2,...,m. If $X \geq m$ then it does not cause overflow. Therefore, assume that $X < m$ and that the first X-1 physical pages were brought into the buffer according to the bring_in_sequence. After bringing in X-1 physical pages, the number of void_records in the buffer will be $TVR + e_1 + ... + e_{X-1}$. Because $Z_k = N_1 + ... + N_{X-1} + N_X + ... + N_m$, it is obvious from the given condition that $TVR + e_1 + ... + e_{X-1} \geq N_X + N_{X+1} + ... + N_{m-1}$. Because only X-1 physical pages were brought into the buffer, there is room for one more physical page. Now, physical page X can be brought into the buffer and because $TVR + e_1 + ... + e_{X-1} \geq N_X + N_{X+1} + ... + N_{m-1}$, N_X nonvoid_records from physical page X can be replaced by N_X void_records. Physical page X becomes a void_physical page and can be written to disk. This leaves $TVR + e_1 + ... + e_{X-1} - N_X$ void_records and room for one page in the buffer. Now physical page X+1 can be brought into the buffer and because $TVR + e_1 + ... + e_{X-1} - N_X \geq N_{X+1} + ... + N_{m-1}$, N_{X+1} nonvoid_records from physical page X+1 can be replaced by N_{X+1} void_records and physical page X+1 becomes a void_Physical page and can be written to disk. This leaves $TVR + e_1 + ... + e_{X-1} - N_X - N_{X+1}$ void_records and room for one page in the buffer. With the same approach for X+2,...,m-1 physical pages; m-X void void_physical pages can be made and room for one page can be made in the buffer at each step. Now, the last physical page can be brought into the buffer without causing any overflow. Therefore, if BIS is used and $(X-1) * pagesize + TVR \geq Z'_k$, then there will be no overflow when bringing logical page k into the ready_state. □

Observation 3: If $X \geq 2$ and $(X-2) * pagesize + TVR \geq Z_k$, then logical page k does not cause overflow or underflow by using BIS.

Proof: Because $(X-2) * pagesize$ is less than $(X-1) * pagesize$, it is trivial to prove that if $(X-2) * pagesize + TVR \geq Z_k$, then by using the bring_in_sequence, logical page k does not cause overflow. Now, we want to prove the second part of this observation. From the first part of this obser-

vation, after bringing in m-1 physical pages from disk, at least two buffer frames will be free. Hence, the last physical page can be placed in either of the two buffer frames and overflow will not occur. □

Heuristics for Join Processing using Nonclustered Indexes

Edward Omiecinski*

Technical Report: GIT-ICS-87/25

June 1987

School of Information & Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

*The research of this author was partially supported by the National Science Foundation under grant IST-8696157.

ABSTRACT

Finding efficient procedures for implementing relational database operations, such as the join, is an important database problem. In this paper, we examine join processing when the access paths available are nonclustered indexes on the joining attribute(s) for both relations involved in the join. We use a bipartite graph model to represent the pages from the two relations which contain tuples that are to be joined. We are interested in minimizing the number of page accesses needed to compute a join in our database environment. We explore this problem from two perspectives. The first is to reduce the maximum buffer size so that no page is accessed more than once and the second is to reduce the number of page accesses for a fixed buffer size. We have developed heuristics for these problems and include performance comparisons of these heuristics and another method which recently appeared in the literature. The results show that one particular heuristic performs very well for addressing the problem from either perspective.

1. INTRODUCTION

Relational query optimization has been the focus of much research in the past several years [4,5,6,7,8]. Optimization strategies try to minimize a particular cost function which might include one or more of the following: secondary storage access cost, main storage cost and computation cost. Typically, the dominating cost is that of accessing a secondary storage device, i.e., page accesses from a secondary storage device to a main memory buffer [4,11]. For the optimization of a relational query, we would like to find efficient procedures for implementing relational database operations. In this paper, we are concerned with one specific relational operator, namely the join, since it is one of the most time consuming operations [11].

An efficient implementation of the join with respect to specific database implementations, i.e., available access paths, has been widely studied [7,8,10,12]. The sort-merge method [4] is usually the best when no indexes on the joining attribute(s) are available. However, if one of the two relations is small enough to fit in the available buffer frames (pages), then the nested block method [4] is preferable; or if one relation is much larger than the other and indexes are available, then the nested block

method using indexes is better. We should note that the type of index, e.g., clustered, where the tuples of a relation are ordered based on the indexing attribute(s); or nonclustered, where the preceeding property does not hold, will affect the performance of a method for join processing which uses indexes.

In this paper, we examine join processing when the access paths available are nonclustered indexes on the joining attribute(s) for both relations participating in the join. We assume as in [10] that indexes are stored on pages separated from those containing the relation tuples. Typically, the indexes are implemented as B-trees. The leaves of a B-tree contain entries which consist of a key value and the addresses of all tuples in a relation which contain that key value. By using the joining indexes of the two relations we can determine which pages contain tuples to be joined. As in [7,8], we assume that not all combinations of pages will need to be examined. This is reasonable for either of the following two cases [7]: one is that a restriction operator is combined with a join and the other is that the tuples on a page(s) for one relation contain no common values of the joining attributes with tuples from a page(s) for the other relation.

As previously stated, we are interested in trying to minimize the number of page accesses needed to compute a join. We will examine this problem from two different perspectives. The first is to reduce the maximum buffer size so that no page involved in the join is fetched more than once. This implies that we need to determine a sequence of pages that are brought into the buffer. One approach to this problem is presented in [8]. The second problem is to try to reduce the number of page accesses for a fixed buffer size. The case for a two page buffer was considered in [7,8] and shown to be NP-hard, and consequently, it is unlikely that a polynomial time solution exists for this problem. We suspect that the first problem is also NP-hard. It is a variation of the minimum cut linear arrangement problem [3] which is NP-hard.

We use a bipartite graph as a model for our problems as done in [7,8]. In Section 2, we prove that the maximum buffer size needed to join relations R and S is bounded by $|R| + |S| - 1$, where $|R|$ is the number of pages in relation R, and so forth. We should note that this bound applies only when $|R|$ and $|S|$ are each greater than one. This assumes that any sequence of page accesses is possible. A similar statement, albeit for general graphs, is made in [8] and an algorithm is presented for finding a page

access sequence, which requires much less than this upper bound. However, no experimental or analytical evidence is presented in regard to how much less this would be.

In Section 2, we present our two heuristics which try to minimize or at least reduce the amount of buffer space needed when no page is reaccessed from a secondary storage device. For the purpose of comparison, we review the algorithm from [8] in Section 3. In Section 4, we present the results of experiments comparing our two heuristics with the method reviewed in Section 3. In Section 5, we show that our heuristics for the first problem are also applicable for the second problem and display additional experimental results in Section 6.

2. GRAPH MODEL AND HEURISTICS FOR REDUCING BUFFER SIZE

The particular graph model which we employ has been referred to as a page-pair bigraph [7] and a page connectivity graph [8]. In this paper, we shall adopt the terminology of [7], i.e., a page-pair bigraph. The page-pair bigraph, representing the join of relations R and S, consists of two sets of vertices: $\{r_1,r_2,...,r_n\}$ and $\{s_1,s_2,...,s_m\}$, where r_i represents a data page from relation R and so forth. An edge between r_i and s_j exists if r_i and s_j have tuples to be joined. This information can be obtained directly from the indexes for relations R and S. The page-pair bigraph may consist of several connected components depending on the tuples from R and S to be joined. An example of a page-pair bigraph is shown in Figure 1.

The first problem which we study involves determining a sequence of pages involved in the join such that each page is accessed at most once and the main memory buffer space needed is minimized. When any page is accessed (i.e., brought into the buffer) it must reside there until all pages that are to be joined with it have been accessed. This is necessary to guarantee that a page is only accessed once. This problem is closely related to the query locality set model [1], the hot set model [9] and the working set model [2]. In [1,9], the idea is to determine the pages and the number of buffer frames needed to provide efficient processing, i.e., a large number of page faults will not occur.

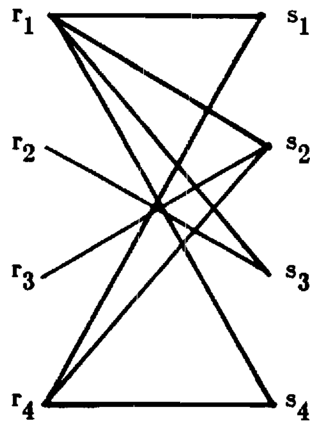


Figure 1. A page-pair bigraph representing the join of R and S

The following two theorems give a bound on the maximum buffer size needed to join two relations under different constraints.

THEOREM 1: The maximum buffer size needed to join relations R and S, when $|R| \geq 1$ and $|S| = 1$, is $|R| + |S|$.

PROOF: Clearly, the maximum buffer size cannot be greater than the total number of pages of both relations. So, our only concern is to show that this maximum can be achieved. Consider a page-pair bigraph where a single S node is connected to all of the R nodes. If all of the R pages are brought into the buffer, then the buffer size becomes $|R|$. Before any of the R pages can be released, the single S page must be brought into the buffer. This yields a maximum buffer size of $|R| + |S|$. □

THEOREM 2: The maximum buffer size needed to join relations R and S, when $|R| > 1$ and $|S| > 1$, is $|R| + |S| - 1$.

PROOF: (By contradiction) Let the maximum buffer size be $|R| + |S|$. Without loss of generality, let s_j be the last page to be accessed. Consider the contents of the buffer directly before s_j is accessed. The buffer size is $|R| + |S| - 1$. All of the R pages may remain in the buffer if they are connected to s_j. However, for the S pages to remain in the buffer, they would also have to be connected to s_j. This would contradict the fact that we started with a bipartite graph, where S nodes can only be connected to R nodes and vice versa. Hence, the maximum buffer size can not exceed $|R| + |S| - 1$.

This maximum can be achieved for a fully connected bipartite graph as follows. First, bring $|R| - 1$ pages of relation R into the buffer. Next, bring in $|S| - 1$ pages of relation S. Finally, the remaining R page is brought into the buffer. At this point, the buffer size is $|R| + |S| - 1$. In addition, all of the S pages in the buffer may be released. □

We define the following function for the buffer size:

$$B_1 = 1,$$
$$B_i = B_{i-1} - D_{i-1} + 1, \text{ for } i=2,3,\ldots, |R| + |S|, \qquad (1)$$

where B_i = buffer size after the ith page is brought into the buffer;

$D_1 = 0,$
D_j = number of pages (frames) in the buffer that are made available after the jth page has been accessed, for $j = 1,2,\ldots, |R| + |S|$.

The following inequalities hold for (1):

$$D_i \le B_i, \text{ for } i = 1,2,\ldots, |R| + |S|,$$

$$\sum_{k=1}^{j} D_k \le j, \text{ for } j = 1,2,\ldots, |R| + |S|,$$

$$B_j \le j, \text{ for } j = 1,2,\ldots, |R| + |S|.$$

The above inequalities characterize the state of the buffer. The first states that the number of pages (frames) in the buffer that can become available ranges from 0 to the current buffer size. The second inequality specifies that the total number of pages released, after the jth page is brought into the buffer, cannot exceed j. We should note, if we are only considering relations which satisfy Theorem 2, then the maximum value for j in the last inequality would be $|R| + |S| - 1$.

From equation (1), the maximum buffer size needed is defined as $MAX\{B_1, B_2, \ldots, B_N\}$ where $N = |R| + |S|$. Thus, our objective is to develop a process of determining a page access sequence which minimizes $MAX\{B_1, B_2, \ldots, B_N\}$ over possible page access sequences. Since the problem appears to be NP-hard, we consider a heuristic approach. We can develop heuristics based on the properties of optimal page access sequences.

PROPERTY 1: To minimize $MAX\{B_1, B_2, \ldots, B_N\}$, maximize D_i for all i.

PROOF: Consider the following two page access sequences $S_1 = (P_{i_1}, P_{i_2}, \ldots, P_{i_N})$ and

$S_2 = (P_{j_1}, P_{j_2}, ..., P_{j_N})$. The buffer size sequences for S_1 and S_2 are $(B_1, B_2, ..., B_N)$ and $(B'_1, B'_2, ..., B'_N)$, respectively.

Let us assume that

$$MAX\{B_1, B_2, ..., B_N\} < MAX\{B'_1, B'_2, ..., B'_N\}. \qquad (2)$$

Thus, at some point we have

$$B_i = MAX\{B_1, ..., B_N\} \text{ and } B'_j = MAX\{B'_1, ..., B'_N\}.$$

Hence, from (2) we know that $B_i < B'_j$.

From equation (1) we can derive the following:

$$B_i = i - \sum_{m=1}^{i-1} D_m.$$

B'_j can be derived in a similar manner as $B'_j = j - \sum_{k=1}^{j-1} D'_k$.

Therefore,

$$i - \sum_{m=1}^{i-1} D_m < j - \sum_{k=1}^{j-1} D'_k.$$

The second inequality, which holds for equation (1), implies

$$i \geq \sum_{m=1}^{i-1} D_m \text{ and } j \geq \sum_{k=1}^{j-1} D'_k.$$

We can consider the following two cases:

Case I: If $i \geq j$ for $B_i < B'_j$,

$$\text{then} \quad \sum_{k=1}^{j-1} D_k > \sum_{k=1}^{j-1} D'_k$$

since $B_j < B'_j$, which is yielded by $B_j \leq B_i$ and $B_i < B'_j$, where B_i is the maximum value for sequence S_1.

Case II: If $i < j$ for $B_i < B'_j$,

$$\text{then} \quad \sum_{k=1}^{j-1} D_k > \sum_{k=1}^{j-1} D'_k,$$

since $B_j \le B_i$ and $B_i < B'_j$ which in turn imply that $B_j < B'_j$.

Thus, if we maximize D_i, for each i, we can minimize $MAX\{B_1,B_2,...,B_N\}$, since $\sum_{k=1}^{j-1} D_k > \sum_{k=1}^{j-1} D'_k$. □

Thus, at any time i, if the page is chosen which makes available the most buffer frames, then this will lead to an optimal page access sequence. Unfortunately, Property 1 is of little help in determining the next page in the sequence, when none of the pages that can be accessed next will free a buffer frame.

From Property 1, we see that we can minimize $MAX\{B_1,B_2,...,B_N\}$ by maximizing D_i for each i. However, by Property 2, as we will show, it is sufficient to choose the next page to access if it frees at least one buffer frame.

PROPERTY 2: If there exist several pages $\{P_{j_1},P_{j_2},...,P_{j_K}\}$ which are candidates to be accessed next, i.e., $D_i > 0$ for ith page ϵ $\{P_{j_1},...,P_{j_K}\}$, then the order of accessing these K pages is immaterial in terms of affecting the maximum buffer size.

PROOF: Let the current buffer size be $B_{i-1} = n$. Without loss of generality, we can let $D_{i-1} = 0$. Whichever page is brought in first will increase the buffer size by 1. Using equation (1), we have $B_i = n+1$. The buffer size needed for accessing the remaining K-1 pages will not exceed n+1. This can be seen from equation (1) and the fact that $D_i > 0$ for the ith page $\in \{P_{j_1},...,P_{j_K}\}$. Hence, the order of accessing only these K pages is immaterial in terms of affecting the maximum buffer size. That is, the buffer size will not exceed n+1 when accesing any of the K pages. □

From Properties 1 and 2, we see that for an optimal page access sequence, it is sufficient to choose the ith page, the one that makes at least one buffer frame available. These properties form the basis for our primary heuristic, i.e., heuristic 1, which follows.

Heuristic 1.

Step 1. Choose an R and an S node from the page-pair bigraph G, e.g. r_i and s_j, to load in the buffer such that

(a) (r_i,s_j) is an edge in G, and

(b) the sum of the degree of r_i and the degree of s_j is minimal over all remaining r's and s's in G.

Then delete (r_i,s_j) from G.

Step 2. Choose the next R or S page, call it p, to bring in the buffer using the following strategy:

(a) find a node q in the buffer such that it is connected to the fewest number of nodes outside the buffer, and

(b) find a node p, such that (q,p) is an edge in G and that the number of edges connecting p to a node not in the buffer is minimal.

If there is more than one node that satisfies (a), then (b) determines which of the nodes is chosen.

Step 3. Delete all edges (r_i,s_j) from G where r_i and s_j are contained in the buffer (either r_i or s_j was the node selected in step 2).

If the degree of a node, r_i, in G becomes zero, then the page (frame) storing r_i in the buffer becomes available and node r_i is deleted.

If G contains no more edges, then quit;
else if no pages of the buffer are being used, then goto step 1;
else goto step 2.

The goal of heuristic 1 is to eliminate at least one page from the buffer after a new page is fetched. For example, if node q, chosen in step 2(a), is only connected to a single node p, chosen in step 2(b), then after p is fetched, the page holding q will be released. We can define the degree for each page (node) in the buffer as the number of pages (nodes) outside the buffer that are to be joined (adjacent) to it. If there does not exist a page in the buffer that has a degree of one, heuristic 1 will at least choose a page which has the smallest degree. Eventually, this will reduce the degree of some page to one.

If we apply heuristic 1 to the page-pair bigraph of Figure 1, the following page access sequence is produced: $r_2,s_3,r_1,s_1,r_4,s_4,s_2,r_3$. The buffer size after each page is accessed, i.e., B_1 through B_8, is

1,2,2,2,3,3,3,2. That is, the buffer contents are r_2; r_2 and s_3; s_3 and r_1; r_1 and s_1; r_1, s_1 and r_4; r_1, r_4 and s_4; r_1, r_4 and s_2; s_2 and r_3 where the page preceeding each semicolon is the page brought into the buffer. Hence, a buffer size of 3 pages is needed for accessing the pages in the generated access sequence. Theorem 3 gives an upper bound on the maximum buffer size produced by heuristic 1.

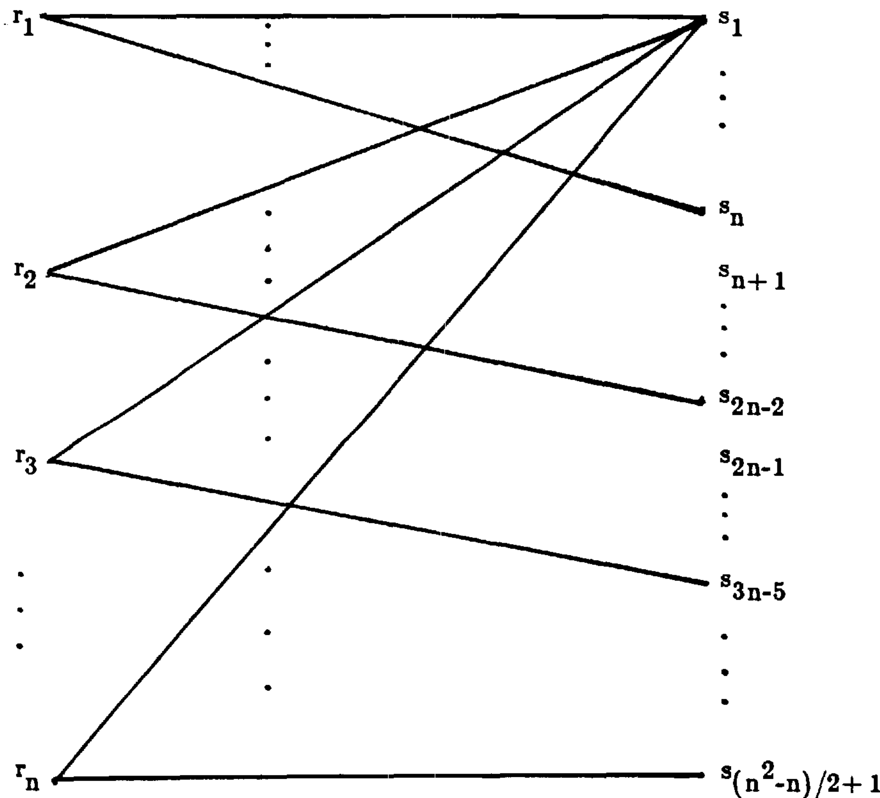


Figure 2. A page-pair bigraph for heuristic 1 yielding maximum buffer size

THEOREM 3: The maximum buffer size produced by heuristic 1 is no greater than MAX($|R|$, $|S|$) + 1.

PROOF: Consider the page-pair bigraph in Figure 2 where

$$|R| = n \text{ and } |S| = (n^2-n)/2+1$$

and for $i \geq 1$, r_i is connected to $s_1, s_2, ..., s_m$ where

$$m = 1 + \sum_{j=1}^{i}(n - j).$$

The graph in Figure 2 is constructed as such so that the maximum number of pages needed at any time (according to heuristic 1) to make a frame available will be accessed. The approach to create the maximum buffer size is to alternate between bringing in a single R page and a set of S pages which will free that R page. Since $|R| < |S|$, we want to delay as long as possible bringing the R pages into the buffer. The reason for this is that once all the R pages are in the buffer, every access to an S page will result in freeing that S page from the buffer. Thus, our buffer size will not exceed its current size.

To start with, heuristic 1 chooses r_1 and s_1 in step 1 to bring into the buffer. The maximum number of page accesses needed at this time to make a frame available is bounded by MIN($|R|$, $|S|$)-1 or n-1 to be specific. This can easily be seen because r_1 is connected to MIN($|R|$, $|S|$)-1 pages and s_1 is connected to MIN($|R|$, $|S|$)-1 R nodes and heuristic 1 looks for a page in the buffer which is connected to the fewest number of pages. Since the degree of the two pages are the same, heuristic 1 might bring in S or R pages. If it were to bring in all the R pages connected to s_1, then our entire buffer size would be bounded by MIN($|R|$, $|S|$)+1. We will show this later with an example. For our purposes let heuristic 1 bring in the MIN($|R|$, $|S|$)-1 S pages, i.e., the following n-1 pages: $s_2, s_3, \ldots, s_n$. At this point the buffer size is n+1 and when the frame holding r_1 becomes available the buffer size is reduced to n, i.e., the buffer contains $s_1, \ldots, s_n$. Now the maximum number of page accesses needed to make a frame available cannot be greater than MIN($|R|$, $|S|$)-1 or n-1 since there are only n-1 remaining R pages. The next page chosen is r_2 since it is connected to the fewest number of pages outside the buffer. We see that r_2 has a degree of n-2 and that all the S nodes in the buffer also have a degree of n-2. Let heuristic 1 bring in $s_{n+1}, \ldots, s_{2n-2}$ which will free r_2. Again, this requires the maximum number of page accesses possible to free a frame in the buffer. Thus, the buffer size is now n+(n-2)+1 and when the frame holding r_2 is freed the buffer size becomes n+(n-2). Once again, if the n-2 remaining R pages were brought into the buffer instead of the S pages, the maximum buffer size would be only n+n-1. At this time, the maximum number of page accesses needed to make a frame available is n-3. Page r_3 would be chosen and n-3 S pages would be accessed, i.e., s_{2n-}

$_1, \ldots, s_{3n-5}$. This yields a buffer size of n+(n-2)+(n-3)+1. By induction, it is easy to see that the buffer size produced when the nth (i.e., MIN(|R |, |S |)th) R page is accessed is

$$1 + \left\{ 1 + \sum_{j=1}^{n-1} (n - j) \right\}$$

giving the maximum buffer size of MAX(|R |, |S |)+1. □

The result of Theorem 3 may seem discouraging since the nested block method [4,11] only requires a maximum buffer size of MIN(|R |, |S |)+1. However, in the worst case example, i.e., a high degree for R and S nodes, we would not propose using our heuristic. As stated earlier, the premise for using heuristic 1 is that all (or a large percentage of) combinations of pages will not have to be examined. As will be shown in Section 4, experimental results support the usefulness of heuristic 1 for our environment.

Here, we show an example using a fully connected bipartite graph, i.e., the number of S vertices adjacent to each R vertex is |S |, giving a total of |R |* |S |edges in the graph. We can assume that |R | ≤ |S | since we can simply rename the relations so that this will hold. Heuristic 1 could choose any edge to start with, call the vertices of the chosen edge r_i and s_j. Now, the degree of r_i and s_j is decreased by one. The next page to be fetched will be adjacent to s_j since the degree of s_j will be less than or equal to that of r_i since |R | ≤ |S |. So, another R page will be accessed, call it r_k. Again, the degree of s_j will be decreased and will be less than that of either r_i or r_k in the buffer. Following this argument, the remaining R pages will be accessed and will remain in the buffer. This yields a buffer size of |R |+ 1, i.e., MIN(|R |, |S |) + 1. At this time, the page (buffer frame) for s_j will become available. The remaining S pages will be fetched one at a time and after each one is fetched the buffer frame which it occupies will become available for the next S page. Hence, the maximum buffer size will be MIN(|R |, |S |) + 1.

Before we leave heuristic 1, we would like to comment on its worst case running time by way of theorem 4.

THEOREM 4: The worst case time complexity of heuristic 1 is $O((|R| * |S|)^2)$.

PROOF: We consider a fully connected bipartite graph and let |R | ≤ |S |. As will be seen in step

2, the worst case time will be dependent on the product of the current buffer size and the degree of the pages (nodes) in the buffer. Although the buffer size will not reach the maximum possible, the degree of the pages in the buffer will attain MAX($|R|$, $|S|$).

Step 1 is done once and requires examinimg each edge. In the case where the bipartite graph is fully connected, there exist $|R|*|S|$ edges. Hence, step 1 requires O($|R|*|S|$) time.

In step 2, the pages in the smaller of the two sets, i.e., R, are fetched first. The buffer grows from 2 to $|R|$ frames and the degree of each of the $|R|$ pages is $|S|$. Thus, the time taken here is

$$\sum_{i=2}^{|R|} \sum_{j=1}^{i} j * |S|$$

which is O($|S| * |R|^3$). Next the remaining pages in the larger set, i.e., S, are fetched. The buffer size remains the same until all the pages have been accessed. As each S page is fetched the degree of each R page contained in the buffer is decreased by 1. Hence, the time taken is

$$\sum_{i=2}^{|S|} \sum_{j=1}^{|R|} j * (|S| - i)$$

which is O($|S|^2 * |R|^2$).

Step 3 is done $|R| + |S| - 2$ times but the total amount of work done is proportional to $|R| * |S| - 1$, i.e., all edges except for the one edge deleted in step 1 must eventually be deleted. This yields O($|R| * |S|$) time.

So, we have O($|R| * |S|$) time for steps 1 and 3 and O(($|R| * |S|)^2$) time for step 2. This produces a worst case time complexity for heuristic 1 of O(($|R| * |S|)^2$). □

Heuristic 2.

Step 1. Same as heuristic 1.

Step 2. Choose the next R or S node, call it p, to bring in the buffer such that

(a) an edge exists in G between p and a node in the buffer, and

(b) the number of edges connecting p to a node outside the buffer is minimal.

Step 3. Same as heuristic 1.

In step 1 of heuristic 2, we choose the first two pages (i.e., an edge in the page-pair bigraph) to be brought in the buffer such that they are connected to the fewest number of other pages. In step 2, we pick a page to be accessed such that it is connected to the fewest number of pages not in the buffer. The motivation behind this approach is to bring a page into the buffer that will hopefully remain there for a shorter period of time (i.e., for a fewer number of additional page accesses) than some other page. Thus, the buffer size will decrease sooner. This approach makes the assumption that the length of stay of a page in the buffer is dependent on the number of pages it is to be joined with, independent of the other pages in the buffer. This heuristic was developed for comparison purposes as discussed in Section 4. It is not expected to have a near optimal performance. If we apply this heuristic to the page-pair bigraph of figure 1, the following page access sequence is produced: $r_2,s_3,r_1,s_1,s_4,r_4,s_2,r_3$. The buffer size after each page is accessed, i.e., $B_1,B_2,...,B_8$, is 1,2,2,2,3,4,3,2. Thus, it requires a buffer size of 4 pages.

3. PREVIOUS APPROACH FOR DETERMINING THE BUFFER SIZE

In [8], an alternative approach for determining the buffer size is presented. Their goal is to find an access sequence requiring much less than the upper bound of $|R|+|S|-1$. We should note that in [8], they discuss the following algorithm in the context of a general graph, i.e., where tuples from more than one relation can be stored on a single data page.

Algorithm 1.

Step 1. Create an acyclic graph, G', from the original graph.

Step 2. Select any node, say n_r, of the acyclic graph and consider the tree with n_r as the root.

Step 3. Find Bound(r), where r is the root of the tree under consideration and Bound(r) is an expression for the upper bound on the buffer size for the given tree.

Step 4. Let node n_j be the root of the subtree that requires

the largest buffer among the subtrees of n_r; i.e., Bound(j)=MAX{Bound(son_r)}.

If the buffer size for the tree under consideration is larger than that for the previous tree or when the tree with root n_j has already been considered, then quit;
else consider the new tree with n_j as the root (i.e., r becomes j) and goto step 3.

With the preceeding algorithm, an upper bound on the buffer size for a given tree is computed. This bound quarantees enough buffer space for any access sequence derived from a tree traversal in which a page is accessed when it is first visited.

In [8], they prove that their algorithm selects the tree that requires the smallest buffer among all possible trees of the acyclic graph. However, the bound for this tree is independent of the order of traversal, except for accessing a node when it is first visited. Thus, the resulting tree gives the minimum upper bound but the question is "how good is this bound." An additional question concerning this approach is "how can we traverse the tree to further minimize the buffer size." One traversal of the tree might result in a smaller buffer size than those of others, but how do we find the corresponding tree traversal.

4. EXPERIMENTAL RESULTS OF METHODS FOR REDUCING BUFFER SIZE

In this Section, we present the results of numerous experiments which compare algorithm 1 and heuristics 1 and 2. For completeness, we include a third heuristic, random, which consists of three steps: the first step randomly selects a node to be fetched, the second step randomly chooses the next node to be fetched from the set of nodes that are connected to nodes in the buffer, and the third step is the same as that in heuristic 1.

For each of the entries in Table I, 40 random bipartite graphs were generated with the specified maximum degree. The maximum degree is defined as a fraction, α, times the number of S pages. We should note that the degree of an S node could be greater than the maximum degree, $\alpha |S|$, of an R node. Each bipartite graph had $|R| = |S| = 50$ giving a total of 100 nodes in the graph. For each of

the 50 R nodes, we generated a random number, M, between 1 and the maximum degree for an R node. We then chose M random S nodes to be connected to the R node. The results are shown in Table I, giving the mean and the standard deviation for the buffer size in pages.

Table I. Mean and standard deviation for buffer size in pages

	α					
	0.05		0.10		0.20	
METHOD	MEAN	STD DEV	MEAN	STD DEV	MEAN	STD DEV
RANDOM		5.175	1.358	23.950	3.707	51.625
ALGORITHM 1	5.325	1.058	19.375	3.071	45.100	4.576
HEURISTIC 2	3.750	0.829	15.275	2.408	44.675	5.110
HEURISTIC 1	3.375	0.696	11.300	1.568	27.350	2.505

In Table II, we show the percent of fewer pages needed for the buffer using heuristics 1 and 2, and algorithm 1, as compared to the random method. For all three sets of experiments where $\alpha = 0.05$, 0.1, and 0.2, corresponding respectively to a maximum degree for an R node of 2, 5, and 10 nodes, we see that heuristic 1 performs better than heuristic 2 which in turn performs better than algorithm 1. Not only is heuristic 1 better on the average but it is better in each individual experiment performed. For the case where $\alpha = 0.10$, we see that heuristic 1 uses approximately 53% fewer buffer frames on the average than the random solution while algorithm 1 uses only 19% fewer buffer frames on the average.

Table II. Per cent fewer pages for buffer as compared to random

	α		
METHOD	0.05	0.10	0.20
ALGORITHM 1	-2.899	19.102	12.639
HEURISTIC 2	27.536	36.221	13.462
HEURISTIC 1	34.783	52.818	47.022

Instead of only making a comparison between different methods which utilize nonclustered indexes, we can compare our experimental results to an alternate join method. For doing this, we choose the nested block method [4,11]. Using this approach, the required buffer size to guarantee a single access per page is MIN($|R|$, $|S|$) + 1. That is, we need enough pages (frames) to hold the

smaller of the two relations plus one page for the other relation. Relative to our experiments, this value would be 51. Hence, for $\alpha = 0.05$ and $\alpha = 0.10$, all of the methods in Table I seem good when their results are compared with a buffer size of 51. For the situation where $\alpha = 0.10$, algorithm 1 uses 62% fewer pages and heuristic 2 uses 78% fewer on the average. However, when we consider $\alpha = 0.20$, algorithm 1 uses only 12% fewer pages while heuristic 1 uses 46% fewer pages and the random solution uses approximately 1% more than that of the nested block method. We should mention that the sort-merge method [4,11] would not require fewer pages since for sorting a relation, each page of that relation has to be accessed, and has to remain in the buffer to guarentee a single access per page.

5. REDUCING THE NUMBER OF PAGE ACCESSES FOR A FIXED BUFFER SIZE

In this Section, we examine our second problem, i.e., reducing the number of page accesses for a fixed buffer size. Since the buffer size is fixed, we may need to access a page more than once to join the appropriate tuples. The minimum buffer size needed to compute the join of two relations is two pages. For the two page buffer case, the minimum number of page accesses is greater than or equal to E+ 1 where E is the number of edges in the page-pair bigraph which is connected [7]. The maximum is less than or equal to 2E [7]. Also in [7], they proved that checking the existence of a minimum solution (i.e., E+ 1) is NP-hard unless the graph is Eulerian. Hence, if the page-pair bigraph contains an Eulerian path, then a minimum solution exists. However, this is only a sufficient condition but not a necessary one [7]. In both [7,8], heuristics are suggested. However, we present heuristics for the more general case where the buffer size can be greater than or equal to two. The two heuristics which we propose are just slight modifications of heuristics 1 and 2, as presented earlier. The difference is the addition of the page replacement scheme (step 2.5) which is shown below. The page replacement scheme assumes a pessimistic view of the buffer situation in trying to reduce page accesses. That is, if a node (page) in the buffer is going to be swapped out, then in the worst case it could be brought in one additional time for each page it is connected to outside the buffer, assuming it were to be swapped out each time.

Step 2.5. If a node (i.e., a page) in the buffer has to be replaced to make room for the new node, then choose the node that

(a) is connected to the fewest number of nodes outside the buffer, and

(b) is not connected to the new node.

If all nodes in the buffer are connected to the new node then only (a) determines the node to be replaced.

6. EXPERIMENTAL RESULTS OF METHODS FOR REDUCING PAGE ACCESSES FOR A FIXED BUFFER SIZE

The same experiments generated in Section 4 were performed to compare the modified heuristics, called heuristics 1' and heuristic 2', with the random approach which uses the same page replacement policy. The results are shown in Tables III, IV, and V. For these experiments, we vary the buffer size from initially 2, by a multiple of 2, to B. The value of B is the greatest multiple of 2 which is less than the buffer size needed to ensure a single access per page (as found in the Section 4 experiments). In Table VI, we show the difference, in per cent of page accesses, between our heuristics and the random method.

Table III. Number of page accesses for $\alpha = 0.05$

	BUFFER SIZE	
	2	
METHOD	MEAN	STD DEV
RANDOM 1'	100.525	4.117
HEURISTIC 2'	92.775	3.965
HEURISTIC 1'	92.300	3.709

Table IV. Number of page accesses for $\alpha = 0.10$

	BUFFER SIZE					
	2		4		8	
METHOD	MEAN	STD DEV	MEAN	STD DEV	MEAN	STD DEV
RANDOM 1'	143.750	6.220	134.400	7.742	126.925	7.780
HEURISTIC 2'	130.000	6.116	120.600	5.847	109.050	5.244
HEURISTIC 1'	128.975	6.122	115.775	5.322	102.450	3.653

Table V. Number of page accesses for $\alpha = 0.20$

	BUFFER SIZE							
	2		4		8		16	
METHOD	MEAN	STD DEV	MEAN	STD DEV	MEAN	STD DEV	MEAN	STD DEV
RANDOM 1'	261.250	16.486	244.200	15.410	225.075	13.576	185.925	11.626
HEURISTIC 2'	242.975	16.658	225.175	15.305	198.750	12.868	163.000	10.445
HEURISTIC 1'	242.150	16.834	200.925	13.608	161.725	10.037	124.950	6.946

Table VI. Per cent fewer page access as compared to random'

		BUFFER SIZE			
α	METHOD	2	4	8	16
0.05	HEURISTIC 2'	7.710			
	HEURISTIC 1'	8.182			
0.10	HEURISTIC 2'	9.565	10.268	14.083	
	HEURISTIC 1'	10.278	13.858	19.283	
0.20	HEURISTIC 2'	6.995	7.790	11.696	12.330
	HEURISTIC 1'	7.311	17.721	28.146	32.795

From Tables III through VI, we see that in all experiments, heuristic 1' performs better than heuristic 2' which in turn performs better than random' for all buffer sizes considered.

During the experiments, information about the page-pair bigraphs was collected. The mean and standard deviation for the number of nodes and the number of edges are shown in Table VII.

Table VII. The mean and standard deviation for the number of nodes and the number of edges in the page-pair bigraphs

	α					
	0.05		0.10		0.20	
	MEAN	STD DEV	MEAN	STD DEV	MEAN	STD DEV
NODES	88.775	2.650	96.050	1.378	99.700	0.557
EDGES	74.650	4.120	122.150	7.020	240.725	16.852

In Table VII, for the case where $\alpha = 0.05$, we see that all the pages for the S relation are not used. Of the 88.775 mean number of nodes in the graph, 50 represents the pages in the R relation. So, on the average only 38.775 out of the 50 nodes of the S relation are used. In addition, the minimum number of edges required if the graph is connected is one less than the number of nodes. So we see that the

average graph consists of several connected components and hence cannot achieve the minimum of E+ 1 page accesses, i.e., 75.650. If we examine Table III, we see that heuristic 1' required 92.3 page accesses on the average. If we compare the mean page accesses (for a buffer size of 2) from Tables IV and V with the number of edges in Table VII, we find that for $\alpha = 0.1$, heuristic 1' requires only about 5% more than the minimum; and for $\alpha = 0.2$, heuristic 1' requires approximately 0.2% more.

As a further point of comparison, for experiments where the buffer size is greater than two, we can compare heuristic 1' with the nested block and the sort-merge methods. The number of page accesses for the nested block method as defined in [11] is $|R| * (1 + \lceil |S| / (B-1) \rceil)$ where B is the buffer size. The sort-merge method requires the following number of page accesses [6]: $2\,|R| \log_{B-1} |R| + 2\,|S| \log_{B-1} |S| + |R| + |S|$. Table VIII shows a comparison of these methods.

Table VIII. Comparison of heuristic 1' results with calculated number of page accesses for nested block and sort-merge, (α=0.2)

	BUFFER SIZE		
METHOD	4	8	16
HEURISTIC 1'	200.925	161.725	124.950
SORT-MERGE	812	510	388
NESTED BLOCK	900	450	250

The comparison in Table VIII fits our expectation, i.e., heuristic 1' performs the best. This is due to the environment we have assumed, i.e., a page from the R relation is to be joined with a small number (20% maximum) of pages from the S relation.

The values for heuristic 1' exclude the cost of searching the indexes for relations R and S. To be fair, we should include this cost. An approximation for the number of leaf pages in the index for relation R can be defined as $\sigma\,|R|$ where σ is the size of one index entry divided by the size of one tuple. This would represent the maximum number of leaf pages to be accessed. If the data is uniformly distributed, then the minimum number of leaf pages to be read would be $\sigma\,|R| / |\{\text{key values}\}|$. A reasonable estimate for σ might be 0.25 for both relations. So, in the worst case, an additional 26 page accesses would be needed to read the leaf pages for the indexes of R and S. The height of the index

(B-tree) is usually between 2 and 4. So, an additional 1 to 3 pages would be needed to reach a leaf page for each index. At that point the leaf pages can be traversed without accessing any other nonleaf nodes. Thus, the total page accesses for both indexes would be approximately 32. Even so, heuristic 1' would still require the fewest number of page accesses, 157 compared to 250 (for nested block) for a buffer size of 16.

To complete this Section, we include one further table which shows the relationship between the buffer size and the per cent of page accesses above the minimum, i.e., the number of nodes in the graph (refer back to Table VII). If we examine the entry for heuristic 1', $\alpha = 0.20$, in Table IX; we find that when the buffer size is 2 pages, we make about 143% more page accesses than the minimum. Notice, that the buffer size of 2 pages is only 7% of the buffer space required to achieve the minimum number of page accesses. However, as we increase the buffer size, the number of page accesses drops substantially. For example, a buffer size of 16 pages is only 48% of the buffer size required to achieve the minimum. In this case, we make only 25% more page accesses. From Table IX, we can see the tradeoff between the buffer size and the number of page accesses using a particular method. In a database system, where there is buffer pooling, the number of available pages (frames) at a given time may be smaller than that necessary to achieve the minimum number of page accesses. Using heuristic 1' we could determine the number of additional page accesses that would be needed and decide whether we should perform the join with a smaller buffer size or to wait until we have the needed number of buffer pages to ensure a single access per page. As an alternative (although not typical of current database systems), it might be viable to employ a dynamic buffer allocation scheme where we ask for additional buffer frames when needed during the join instead of having them allocated at the start of the join. In addition, since we know the page access sequence, ala heuristic 1', it might be necessary to only hold a specified (maybe large) number of frames over a short period of time.

Table IX. Per cent more page accesses as compared to the minimum of a single access per page

		BUFFER SIZE			
α	METHOD	2	4	8	16
0.05	RANDOM'	13.236			
	HEURISTIC 2'	4.501			
	HEURISTIC 1'	3.971			
0.10	RANDOM'	49.662	39.927	32.145	
	HEURISTIC 2'	35.346	25.560	13.535	
	HEURISTIC 1'	34.279	20.536	6.663	
0.20	RANDOM'	162.036	144.935	125.752	86.484
	HEURISTIC 2'	143.706	125.853	99.348	63.490
	HEURISTIC 1'	142.879	101.530	62.212	25.326

7. CONCLUSION

In this paper we have presented strategies for reducing the number of page accesses to perform a join when nonclustered indexes are available. We also assumed that under certain conditions [7], only a small number of the total possible page combinations will need to be examined. One direction is to find the minimum buffer size which would guarantee that no page would be reaccessed. Two heuristic approaches were developed and performance comparisons of these heuristics and the method in [8] were given. The second direction involves modifying our heuristics so that they would apply when there is a fixed buffer size. The results of these experiments showed that one particular heuristic worked well for addressing the problem from either perspective.

REFERENCES

[1] H. Chou and D. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," 1985 VLDB Conference Proceedings, eds. A. Pirotte and Y. Vassiliou, Morgan Kaufmann Publishers Inc., Los Altos, CA, August 1985, pp. 127-141.

[2] P. Denning, "The Working Set Model for Program Behavior," CACM, ACM, New York, Vol. 11, No. 5, May 1968, pp. 323-333.

[3] M. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Co., San Francisco, 1979.

[4] M. Jarke and J. Koch, "Query Optimization in Database Systems," Computing Surveys, ACM, New York, Vol. 6, No. 2, June 1984, pp. 111-152.

[5] W. Kim, "A New Way to Compute the Product and Join of Relations," 1980 SIGMOD Conference Proceedings, ACM, New York, 1980, pp. 179-187.

[6] W. Kim, "On Optimizing an SQL-like Nested Query," TODS, ACM, New York, Vol. 7, No. 3, September 1982, pp. 443-469.

[7] T. Merrett, Y. Kambayashi and H. Yasuura, "Scheduling of Page-Fetches in Join Operations," 1981 VLDB Conference Proceedings, ACM, New York, 1981, pp. 488-498.

[8] S. Pramanik and D. Ittner, "Use of Graph-Theoretic Models for Optimal Relational Database Accesses to Perform Join," TODS, ACM, New York, Vol. 10, No. 1, March 1985, pp. 57-74.

[9] G. Sacco and M. Schkolnick, "A Mechanism for Managing the Buffer Pool in a Relational Database System using the Hot Set Model," 1982 VLDB Conference Proceedings, Morgan Kaufmann Publishers Inc., Los Altos, CA, 1982, pp. 257-262.

[10] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price, "Access Path Selection in a Relational Database Management System," 1979 SIGMOD Conference Proceedings, ACM, New York, 1979, pp. 23-34.

[11] J. Ullman, Principles of Database Systems, Computer Science Press, Rockville, MD, 1982.

[12] S. B. Yao and D. DeJong, "Evaluation of Database Access Paths," 1978 SIGMOD Conference Proceedings, ACM, New York, 1978, pp. 66-77.

**Concurrent Storage Structure Conversion:
from B+ Tree to Linear Hash File**

Edward Omiecinski*

Technical Report: GIT-ICS-87/24

June 1987

School of Information & Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

*The research of this author was partially supported by the
National Science Foundation under grant IST-8696157.

ABSTRACT

The motivation for this paper is to show that the efficient reorganization of a B+ tree file into a linear hash file can be done concurrently with user transaction processing. This conversion is motivated by a change in database processing where efficient sequential and direct access are originally needed but now only efficient direct access is needed. This is quite reasonable for a database system which accomodates new and changing applications. Several existing database systems, e.g. INGRES [22], IMS [20] and IDMS [20], allow this type of reorganization but the reorganization is performed off-line. We devise an algorithm which performs the conversion and present an analytic model of the conversion process. We also employ a typical database simulation model to evaluate the reorganization scheme. The results from the analytic model are within 3% (on the average) of the observed simulation results. The results of the simulation support the idea of doing file conversion concurrently with database usage especially when compared to an off-line reorganization approach.

1. INTRODUCTION

We define file reorganization as the process of changing the physical structure of the file [20]. Reorganization may be performed for a variety of reasons such as to reduce retrieval time or compact space. Concurrent reorganization is an on-line strategy where the file is reorganized concurrently with usage [20]. With this approach, the part of the file which is being reorganized is locked while user access is permitted to the remainder of the file. The relational database system, System R [2], supports concurrent reorganization to some extent in allowing new attributes to be added to existing relations as well as allowing the creation of new indexes or the deletion of old ones without dumping and reloading the data, i.e. without performing off-line reorganization. In any environment where the database system must be available 24 hours per day, i.e. highly available systems [5], typical off-line reorganization cannot be tolerated. Additional work in concurrent reorganization can be found in [13,14,21].

In this paper we are concerned with a category of file reorganization which we call inter-structural change. By inter-structural change, we mean that the file structure created by the reorganization process is of a different type than that which existed prior to the reorganization. This might also be called file conversion. An example of this would be to convert an indexed file to a hash based file as

can be done in INGRES with the modify operation [22]. However, in INGRES this is done off-line, i.e. prohibiting user access during the process. This can also be done in IDMS [20] with an unload/reload utility and in IMS [20], changing a HIDAM structure to HDAM. This conversion is motivated by a change in database processing where efficient sequential and direct access are originally needed but now only efficient direct access is needed.

In this paper, we propose a concurrent reorganization scheme which allows an on-line and in place conversion of a B+ tree to a linear hash file. The conversion works quiet nicely since both file structures are dynamic, i.e. they can grow and shrink one page at a time. Until the reorganization is complete, part of the file would exist as a B+ tree and part as a linear hash file. It will also be quiet clear which file has to be accessed for a given search key request. In section 2, we review some of the relevant work which exists. In section 3, we examine the B+ tree to linear hash file conversion process. In section 4, we introduce an anlytic model of the conversion process and present the simulation model with results in section 5.

2. BACKGROUND

First of all we will briefly review linear hashing which was originally proposed in [12] and extended by various researchers [9,10,15,17]. Linear hashing is intended for files that expand and contract dynamically. For the expansion process buckets (i.e. pages) are split in a cyclic manner. One rule that can be used to decide when to expand is to split the next bucket in the cycle whenever any bucket overflows. This is referred to as uncontrolled splitting [12]. In addition, random access of a given record, on the average, requires approximately 1 disk access [12].

In linear hashing, the hash function to be applied changes as the file grows or shrinks. The function, $h_0 : k \rightarrow \{0,1,...,N\text{-}1\}$ is used to initially load the file where k is a key. The hash function is dynamically modified creating a sequence of hash functions $h_1,h_2,...,h_i$ such that for any k either $h_i(k) = h_{i-1}(k)$ or $h_i(k) = h_{i-1}(k) + 2^{i-1} \times N$.

When a key is to be inserted, the appropriate function is used to find the correct bucket. Collisions are handled by creating a chain of overflow buckets and in addition a split is performed. The splits are performed in linear order, starting from bucket 0. When all N buckets are split, the address space

doubles in size and the splitting process starts again from bucket 0. Two variables are maintained for this process: NEXT, which denotes the next chain to be split and LEVEL, which represents the number of times the address space has doubled in size. These variables are updated as follows:

$$\text{NEXT} \leftarrow (\text{NEXT} + 1) \bmod \text{N} \times 2^{\text{LEVEL}}$$

$$\text{if NEXT} = 0 \text{ then LEVEL} \leftarrow \text{LEVEL} + 1.$$

Using these two variables, the bucket where a record is to be stored is determined as follows:

$$\text{BUCKET} \leftarrow h_{\text{LEVEL}}(\text{key})$$

$$\text{if BUCKET} < \text{NEXT then BUCKET} \leftarrow h_{\text{LEVEL}+1}(\text{key})$$

where $h_{\text{LEVEL}}(\text{key})$ is defined as key mod $2^{\text{LEVEL}} \times \text{N}$.

Hashing is the appropriate file organization when random access is needed but when both random and sequential access is needed a more appropriate structure to use is the B+ tree [3]. The B+ tree or other variants of the B-tree [3] have been widely used in recent years for storing large files of information on secondary storage, e.g. System R [2]. The average random access search time is typically 3 or 4 disk accesses depending on the height of the tree. Efficient sequential processing is provided by linking the leaf nodes of the B+ tree together in key sequence order. A sample B+ tree is shown in figure 1.

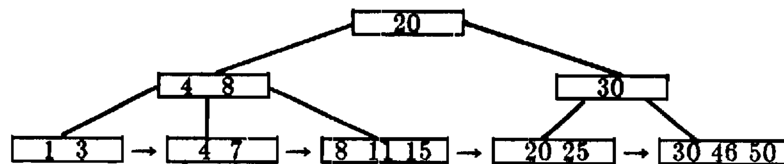


Fig. 1. B+ tree example

To allow for concurrent operations on these file structures, both have undergone modification. The method for achieving greater concurrency is to provide a way to detect and recover from the effect of concurrent updates. In [8,11], schemes were presented to allow for concurrent manipulation of B+

trees. In [11], a single additional link in each node allows a process to easily recover from tree modifications performed by other concurrent processes. Their solution uses a simple locking scheme and requires only a small and constant number of nodes to be locked by an update process at any given time. In [4], a solution for concurrent access to a linear hash file is presented. Part of the solution involves a minor modification to the hash file, i.e. keeping a LOCAL-LEVEL variable with each bucket. The other part carries over the idea of additional links from B+ trees in the form of recalculation [4]. The notion of recalculation is as follows: upon gaining access to a bucket, a process checks whether the LEVEL value used to calculate the address matches LOCAL-LEVEL, and if not, it increments its value and recalculates the address until a match is found. The buckets reached in this manner are those that were created by splitting buckets at addresses already accessed during this search. A simple locking scheme is also employed to control access to the shared variables, LEVEL and NEXT, as well as to the buckets. We should note that a primary bucket and all of its overflow buckets (i.e. a chain) are locked as a unit.

3. B+ TREE TO LINEAR HASH FILE CONVERSION

Besides reorganizing on-line, we also want to satisfy the requirement of reorgainizing in place, i.e. using the storage of the original file with perhaps a small additional amount. The reorganization process takes one page at a time from the B+ tree file and inserts the records from that page in the linear hash file. As we reorganize a page from the B+ tree file, that page, i.e. the storage unit, can be added to the linear hash file when a subsequent split operation is performed on the hash file. However, the first page, i.e. page 0, is added when the first B+ tree page is reorganized.

The reorganization process proceeds in physical address sequence of pages, e.g. 0,1,2,... . If the key sequence of the records corresponds to the physical address sequence then during reorganization the database system need only keep track of the highest key (record) moved from the original B+ tree file to the hash file. Knowing the highest key will allow the database system to direct searches, updates and deletes for a key of smaller or equal value to the linear hash file and for a larger key to the remaining B+ tree file. This allows the benefits of the partial linear hash file to be gained immediately, i.e. as each page is converted.

However, due to splitting pages in the B+ tree file, the physical sequence of data pages may not correspond exactly to the key sequence. Some B+ tree file systems like IBM's VSAM [3] group consecutive pages (control intervals, a la IBM) into larger physical areas (control areas). Therfore when a page is split, an empty page within the same area is used. If the empty page is not physically adjacent to the old one then the key sequence - physical sequence pairing is lost. However, the two pages are still within the same area. Although, in the worst case the area may be full, thus causing an additional area to be allocated. If there still is a somewhat limited form of clustering, in that key sequence is maintained between areas, then once an entire area has been converted to the linear hash file structure, the pages within that area will be accessable through the hash file. Once again, the decision to use the B+ tree or linear hash file will be based on the high key value previously mentioned.

As an alternative, we could proceed in key sequence, regardless of the key-page sequence correspondance, but we would have to employ indirect addressing. The hash function would produce an entry in a page table which would contain the relative page (bucket) number. Using a page table would increase i/o time if the table could not be maintained in memory. This tradeoff would have to be explored before committing to this approach.

The B+ tree to linear hash file conversion algorithm is presented in figure 2. The Insert_separately procedure is basically the insert procedure of [4], where each key is inserted separately.

```
Variables
    State consists of global variables NEXT and LEVEL,
    Leaf page consists of a set of keys and a link to the next page
Procedures
    X_lock and S_lock acquire exclusive and shared locks respectively,
    Get retrieves a page from secondary storage,
    X_unlock and S_unlock release exclusive and shared locks respectively,
    Addrecord places one record on a page (either primary or overflow),
    Add_multi_records places a group of records on a page

Procedure BtreeHash;
Begin
    lpage := address of first leaf page;
    lpage_set := ∅ ;
    While (lpage ≠ Null) Do
    Begin
        X_lock(lpage);
        Get(lpage);
```

```
        lpage_set := lpage_set ∪ {lpage};
        new_high_key := highest key ∈ lpage↑.keys;
        next_lpage := lpage↑.link;
        S_lock(state);
        bucket_set := ∅ ;
        bucket_key_set := ∅ ;
        overflow := false;
        split_is_needed := false;
        For each key, k ∈ lpage↑.keys Do
        Begin
            l := level;
            bucket := Hash(l,k);
            If bucket < next
            Then
                Begin
                    l := l + 1;
                    bucket := Hash(l,k);
                End;
            bucket_set := bucket_set ∪ {bucket};
            bucket_key_set := bucket_key_set ∪ {(bucket,k)};
        End;
        If |bucket_set| × (record_size ÷ page_size) > threshold
        Then
            Insert_separately(lpage)
        Else
            Insert_Group(lpage,bucket_set,bucket_key_set);
        if split_is_needed and (next + 2^level × n ∈ lpage_set)
        then
            Begin
                Split;
                split_is_needed := false;
            End;
        X_unlock(lpage);
        lpage := next_lpage;
    End;
End;


Procedure Insert_group(lpage,bucket_set,bucket_key_set);
Begin
    For each b ∈ bucket_set do
    Begin
        If lpage ≠ b
        Then
            Begin
                X_lock(b);
                Get(b);
                keys := {k_i | (b_i,k_i) ∈ bucket_key_set and b_i = b};
                Add_multi_records(b,keys,overflow);
                If overflow Then split_is_needed := true;
                X_unlock(b);
            End
        Else
            Begin
```

```
                keys := {k_i |(b_i,k_i) ∈ bucket_key_set and b_i = b};
                Add_multi_records(b,keys,overflow);
            End;
    End;
    S_unlock(state);
    X_lock(high_key);
    high_key := new_high_key;
    X_unlock(highkey);
End;


Procedure Insert_separately(lpage);
Begin
    For each key, k ∈ lpage ↑.keys Do
    Begin
        l := level;
        bucket := Hash(l,k);
        If bucket < next
        Then
            Begin
                l := l + 1;
                bucket := Hash(l,k);
            End;
        X_lock(bucket);
        S_unlock(state);
        Get(bucket);
        While (local_level ≠ l) Do
            Begin
                l := l + 1;
                previous := bucket;
                bucket := Hash(l,k);
                If bucket ≠ previous
                Then
                    Begin
                        X_lock(bucket);
                        X_unlock(previous);
                        Get(bucket)
                    End;
                Addrecord(bucket,k,overflow);
                X_unlock(bucket);
                If overflow Then split_is_needed := true;
            End;
        If all keys ∈ lpage↑.keys have not been inserted
        Then S_lock(state);
    End;
End;
```

Fig. 2. B+ tree to linear hash file conversion algorithm

The Insert_group procedure in figure 2 generates a reorganization transaction which inserts multiple records on the same page. This saves i/o and cpu time since the same page need not be

locked and written multiple times. However, during this process the state variables have a shared lock on them which means that concurrent updates on the hash file cannot take place. This procedure would be used initially during the conversion since the hash file is small and there is a greater likelyhood that keys will hash to the same bucket. On the other hand, the Insert_separately procedure generates a reorganization transaction which inserts a single record. This is useful when the linear hash file becomes large enough such that keys from the B+ tree leaf pages hash to different buckets. The advantage here is that the lock on the state variables can be released once the linear hash file page has been locked. Which procedure to call is controlled by a threshold value. For example, if more than 50% of the keys from the B+ tree leaf page hash to different pages then call Insert_separately otherwise call Insert_group.

An example of the conversion process using the B+ tree of figure 1 follows. The algorithm and the example illustrate the simple case where the leaf pages appear in physical sequential order by key value. The algorithm is easily adapted to handle the more general case by keeping track of the pages that have been reorganized and by delaying the updating of the global variable high_key until the pages which represent a consecutive range of key values have been converted. The analytic and simulation models of sections 5 and 6 are based on this more general and more realistic situation. However, at this point the simple case will suffice to illustrate the conversion concepts.

Example

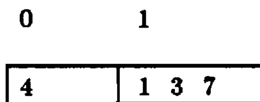
Step 1. get an exclusive lock on page 0, bring leaf page 0 into the buffer, make into hash page structure (i.e. containing records and local_level) using key mod $2^0 \times 1$ as the current hash function and update high_key to 3. This requires a shared lock on the state variables (next and level) and an exclusive lock on the high_key variable. Afterwards, all locks are released.

0

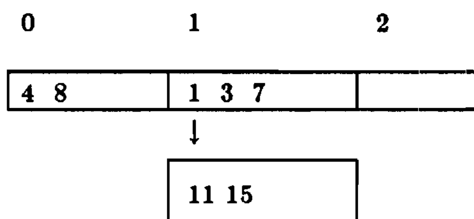
1 3

Step 2. get an exclusive lock on page 1, bring leaf page 1 into the buffer, using our previous hash function, both keys 4 and 7 would hash to page 0 so get an exclusive lock on page 0, get page 0

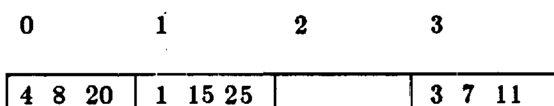
and add records to overflow chain for page 0. The overflow generates a split which is done after the records on page 1 have been inserted in the hash file. Performing the split at this time allows page 1 to be added to the hash file storage space. We should note that splitting is done if overflow occurs but after all the records on the current page, which is to be reorganized, have been inserted. This is necessary, as in this case, so that there will be a new page which can be used for the split. If the page needed for the split, i.e. at location NEXT + $2^{\text{LEVEL}} \times N$, has not been converted then the split is deferred. This requires exclusive locks on the state (since LEVEL is increased to 1) and high_key (which is set to 7) variables. Afterwards, all locks are released.



Step 3. get an exclusive lock on page 2, bring leaf page 2 into the buffer, the keys 8, 11 and 15 hash to pages 0, 1 and 1 respectively, obtain exclusive lock on page 0 and insert key 8, obtain exclusive lock on page 1 and insert keys 11 and 15 on an overflow page (we should note that having an exclusive lock on a primary page precludes access by other transactions on the overflow chain as well as on that primary page), locks would be released. a split process is generated next which requires exclusive locks on pages 0 and 2, afterwards the locks are released.



Step 4. similar to previous steps except that page 3 is being converted,



Step 5. similar to previous steps but with page 4 being reorganized,

0	1	2	3
4 8 20	1 15 25	30 46 50	3 7 11

For this particular example, the linear hash file used one less page of storage as did the leaf pages of the B+ tree file. In general, this is not the case but as we will see in section 5, the additional space used by the linear hash file will be insignificant. If we consider the space used by the index, the linear hash file will probably require less.

4. ANALYTIC MODEL OF CONVERSION PROCESS

In this section we present a simple analytic model of the conversion process. In particular, we want to determine the breakpoint, i.e. how many pages need to be converted before the throughput of the system with concurrent reoeganization will equal the throughput of a system with only transaction processing using the B+ tree file. After this point, the performance of the system with concurrent reorganization becomes better. The various properties that we are interested in are as follows.

convert(i) : cost of converting ith page from B+ tree to linear hash file (in pages)

split(j) : cost of performing jth split in linear hash file (in page accesses)

btree : expected cost of a transaction for the B+ tree file (in page accesses)

bhash(i) : expected cost of a transaction using the B+ tree / linear hash file after the ith page has been converted (in page accesses)

height : height of B+ tree

mp : multiprogramming level

n : number of leaf pages in B+ tree file

nrec : number of records in B+ tree leaf page

r : reorganization unit

rpr : probability that a transaction only reads

wpr : probability that a transaction does an update, i.e. 1-rpr

lfpr : probability that an insertion in the linear hash file is to the left of the split pointer

rtpr : probability that an insertion is to the right of the split pointer

ovfl(m) : overflow chain length for home page left of the split pointer for a linear hash file with m records

ovfr(m) : overflow chain length for home page right of the split pointer for a linear hash file with m records

To simplify our analysis, we will not take into account the effect of locking. Access to our file is based on a single key, so the probability that two transactions will block each other will be small. (In the simulation of section 5, the queries were generated randomly with a uniform distribution so our assumption should not be too severe.) Even though the reorganization process locks at most nrec home pages in the linear hash file, the conflict will still be fairly small. We should also note that when a page is converted (excluding the first page), a split operation is performed to add that physical page to the linear hash file. This varies from the algorithm but it is a close approximation which will simplify our model.

The major cost of query processing is accesses to secondary storage. As such, we will use the number of i/o page accesses which a given conversion process makes, multiplied by the multiprogramming level (which is fixed) as the time to complete a given reorganization process. The number of transactions that can be processed using the B+ tree file during the time it would take to convert L pages for the B+ tree / linear hash file is defined below.

$$\sum_{i=1}^{L} \frac{mp(\,convert(i)+split(i)\,)}{btree} \qquad (1)$$

For the file system where the conversion is taking place, we have the following number of transactions that can be processed for the same time frame.

$$\sum_{i=1}^{L} \frac{(mp-1)(\,convert(i)+split(i)\,)}{bhash(i)} \qquad (2)$$

We have a factor of mp-1 in the numerator of (2) since the conversion process is being performed concurrently with transaction processing. Hence, the time available for user transactions is reduced.

The derivations of the following formulas are straightforward and are omitted. Similar formulas

for the linear hash file operations are presented in [16]. The individual terms of (1) and (2) are derived as follows.

$$split(i)=2(1+ovfl(i*nrec))+(1+ovfr(i*nrec))$$

$$convert(i)=1+nrec[(2+ovfl(i*nrec))*lfpr+(2+ovfr(i*nrec))*(1-lfpr)]$$

$$btree=rpr*height+(1-rpr)(height+1)$$

$$bhash(i)=(rpr)[height\frac{n-\lfloor i/r\rfloor r}{n}+((1+ovfl(i*nrec))*lfpr+(1+ovfr(i*nrec))(1-lfpr)\frac{\lfloor i/r\rfloor r}{n}]+$$

$$(1-rpr)[(height+1)\frac{n-\lfloor i/r\rfloor r}{n}+((2+ovfl(i*nrec))*lfpr+(2+ovfr(i*nrec))(1-lfpr)\frac{\lfloor i/r\rfloor r}{n}]$$

The bhash formula has a component for searching/updating the part of the file that resides in the B+ tree and linear hash structure. For the reorganization unit of 1 page, there is a probability of i/n that the query accesses the linear hash structure and (n-i)/n that it accesses the B+ tree structure. If the reorganization unit is 5 for example, then groups of 5 pages would have to be converted before any of the records from those 5 pages could be accessible through the linear hash structure. Hence, in general, for a reorganization unit of r, the probability that a B+ tree leaf page is accessed is $(n-\lfloor i/r \rfloor r)/n$ and the probability that a linear hash file page is accessed is $(\lfloor i/r \rfloor r)/n$.

To simplify our problem, we will make one last assumption. We will assume that overflow is negligible. This should not limit our model too much since a successful search in a linear hash file requires on the average a single access [12]. This assumption has a direct effect on convert(i) and split(i). It means that the cost of converting and splitting a page is constant. Hence, when we want to compute the breakpoint, i.e. the value such that

$$(mp-1)\sum_{i=1}^{L}\frac{convert(i)+split(i)}{bhash(i)} - mp\sum_{i=1}^{L}\frac{convert(i)+split(i)}{btree}=0, \quad (3)$$

we can cancel out the convert(i) + split(i) term. Once again, this simplification is not entirely correct but it should yield a reasonable approximation.

In addition bhash(i) becomes the following:

$$rpr(height\frac{n-\lfloor i/r\rfloor r}{n}+\frac{\lfloor i/r\rfloor r}{n}) + (1-rpr)[(height+1)\frac{n-\lfloor i/r\rfloor r}{n}+2\frac{\lfloor i/r\rfloor r}{n}].$$

Thus, through simplification, the breakpoint equation of (3) becomes

$$(mp-1)\sum_{i=1}^{L}\frac{1}{(height+1-rpr)\frac{n-\lfloor i/r\rfloor r}{n}+(2-rpr)\frac{\lfloor i/r\rfloor r}{n}} - mp\sum_{i=1}^{L}\frac{1}{rpr*height+(1-rpr)(height+1)} = 0 \qquad (4)$$

Since the denominator of the term in the second summation is a constant we can further simplify (4) and group terms to get (5).

$$\sum_{i=1}^{L}\frac{1}{(height+1-rpr)n-(height-1)\lfloor i/r\rfloor r} - \frac{mp*L}{(mp-1)(height+1-rpr)n} = 0 \qquad (5)$$

At this point, we would like a closed form expression for the summation in (5). We can replace the summation with

$$\frac{1}{height-1}\sum_{i=1}^{L}\frac{1}{\frac{(height+1-rpr)n}{height-1} - \lfloor i/r\rfloor r}.$$

With a little effort, we can derive an equivalent summation in a more suitable form, as shown below.

$$\frac{1}{height-1}(r\sum_{j=0}^{\lfloor (L+1)/r\rfloor-1}\frac{1}{\frac{(height+1-rpr)n}{height-1}-j} + \frac{(L+1)\ mod\ r}{\frac{(height+1-rpr)n}{height-1} - \lfloor (L+1)/r\rfloor}$$

$$- \frac{1}{\frac{(height+1-rpr)n}{height-1}}).$$

Using the Harmonic numbers [18], $H_n=\sum_{i=1}^{n}\frac{1}{i}$, we can do a further reduction to yield

$$\frac{1}{height-1}(r*H_{\frac{(Height+1-rpr)n}{height-1}-1} - r*H_{\frac{(height+1-rpr)n}{height-1} - \lfloor (L+1)/r\rfloor} + \frac{(L+1)\ mod\ r}{\frac{(height+1-rpr)n}{height-1} - \lfloor (L+1)/r\rfloor}$$

$$- \frac{1}{\frac{(height+1-rpr)n}{height-1}}).$$

Using the approximation [6], $H_n \approx ln(n)+\gamma+1/2n$, and substituting back into (5) yields the following :

$$\frac{1}{L}[ln(\frac{(height+1-rpr)n}{height-1}) + \frac{1}{\frac{2(height+1-rpr)}{(height-1)}} - ln(\frac{(height+1-rpr)n}{height-1} - \lfloor (L+1)/r \rfloor)$$

$$- \frac{1}{2(\frac{(height+1-rpr)n}{height-1} - \lfloor (L+1)/r \rfloor)} + \frac{(L+1) \ mod \ r}{r(\frac{(height+1-rpr)n}{height-1} - \lfloor (L+1)/r \rfloor)}$$

$$- \frac{1}{\frac{r(height+1-rpr)n}{(height-1)}})] - \frac{mp(height-1)}{(mp-1)(height+1-rpr)n} = 0 \qquad (6).$$

Since all the but one variable in (6) are known, we simply need to find the root of equation (6) which is not a difficult task. We can use an iterative method for approximating a real root of an equation [7]. In the next section we will compare the results of our analytic model with those of the simulation model.

5. DATABASE SIMULATION MODEL AND RESULTS

The simulation model is an adaptation of the models presented in [1,19] and is shown in figure 3. The simulation model uses a fixed multiprogramming level and a dynamic locking scheme (two-phase) where the lockable units are pages. The simulation parameter values used for the experiments are given in table I and are typical of those in [1,19].

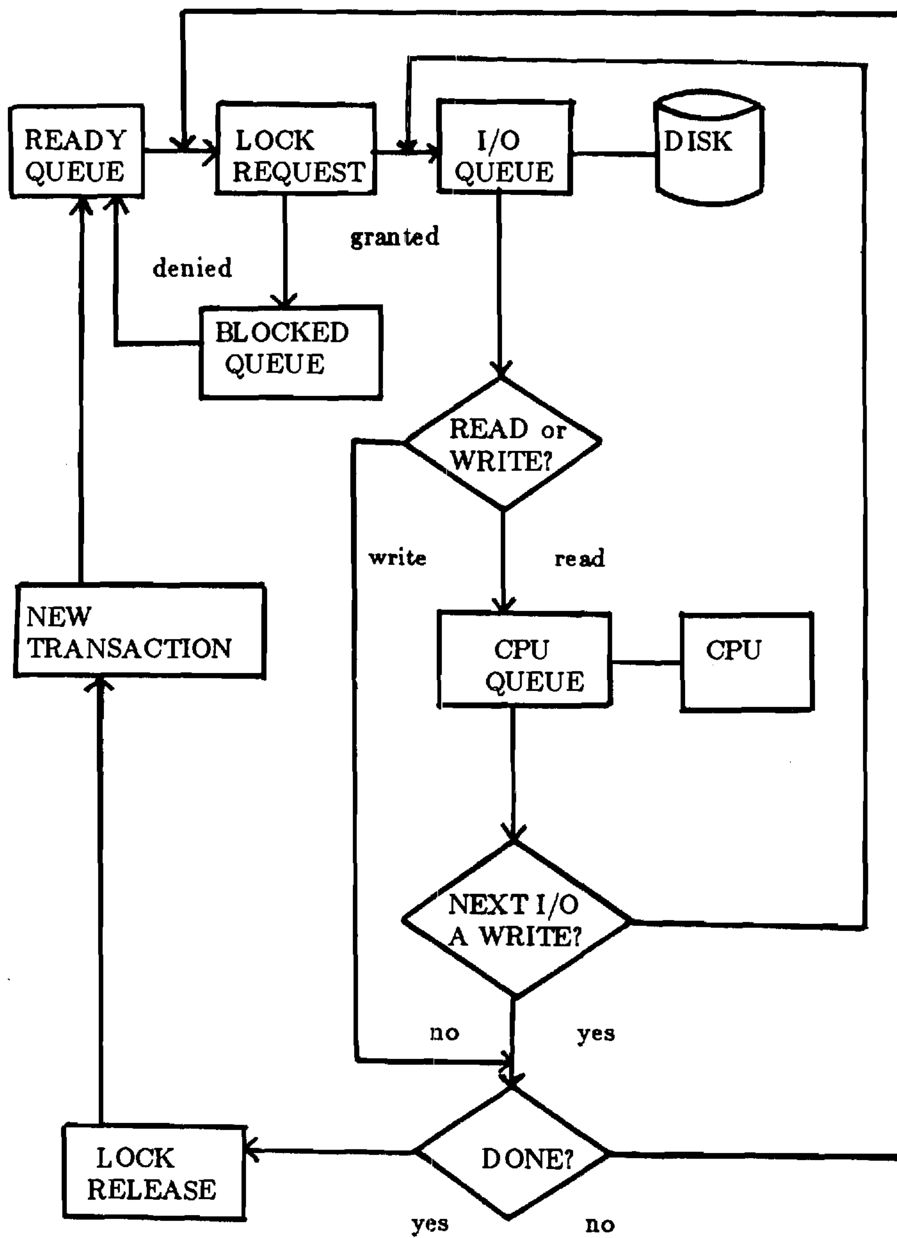


Fig. 3. Database simulation model

Table I. Simulation parameter settings

Parameter	Value
multiprogramming level	10
write probability	0.25
page i/o time	35 ms
page cpu time	15 ms
lock i/o time	0
lock request cpu time	2.5 ms
lock release cpu time	2.5 ms
database size	143 pages
data page size	10 records
data page load factor	0.70
B+ tree height	3
reorganization unit	1,5,10,15, 20,25,...

The model simulates transactions made against the database. The transactions are of two different types: file processing and reorganization. File processing transactions are submitted by users for retrieval or update of the file. Since the motivation for the conversion is the need for efficient direct access only, we will restrict the file processing transactions to just those. These transactions involve accessing two index pages (i.e. nonleaf nodes) and a data page (i.e. leaf node). To simplify the simulation we assume that the pages in the B+ tree do not split or merge. Since there will be only one reorganization process running at a time, the splitting and merging of B+ tree pages would have more of an effect on user transaction time than they would on reorganization time.

The reorganization transaction type can actually be further subdivided. One class accesses a data page from the B+ tree file and places the records from that page into the linear hash file. This may involve accessing as many as ten (i.e. page size in records) primary pages in the linear hash file as well as additional overflow pages. The other class, due to a split, accesses two pages from the linear hash file and possibly some overflow pages.

In the simulation, a fixed number of transactions are active at any one time. The file processing transactions are randomly generated where 25 per cent of them are writes. When the first user transaction terminates, a reorganization process is entered into the system. When each reorganization process terminates, a new one is generated. Therefore, only one reorganization process is active at a time.

All reorganization processes require exclusive locks which are requested one at a time. To prevent deadlock in our database model, locks are requested in a fixed linear order. We also assume that to write an existing page, a transaction must first read it.

Initially, user transactions arrive one time unit apart and are placed on the READY queue. A transaction then goes through the following stages.

(a) The transaction is removed from the READY queue and one lock is requested. If the lock is granted, the transaction is placed on the bottom of the I/O queue. If the lock is denied, the transaction is placed at the bottom of the BLOCKED queue. The blocking transaction is recorded.

(b) After completing the required I/O, the transaction does one of two things. If the I/O is a read then the transaction is placed at the bottom of the CPU queue. If the I/O is a write then the transaction, if it were not finished, would request another lock and cycle around again. If the transaction is finished after the write then all locks are released. All transactions blocked by the completed transaction are placed on the front of the READY queue.

(c) After completing the required cpu for the page accessed, the transaction, if it were not finished, would request a lock on another page and repeat the cycle or if the current page is to be written after being read, the transaction would be placed at the bottom of the I/O queue. If the transaction is finished after the cpu processing then all locks are released. All transactions blocked by the completed transaction are placed on the front of the READY queue.

The results of the various simulation runs are summarized in table II. The unit parameter indicates the increment (in pages) for which converted pages are made accessible through the linear hash file. For example, the value 5 indicates that after every group of 5 pages from the B+ tree file have been converted, the records from those pages can be accessed through the linear hash file. This means that the keys (records) that have been processed so far, i.e. in physical page sequence, represent a consecutive range of key values. Since the reorganization is done in page sequence, the higher values for unit represent a greater disparity between key and physical page sequences. The value of 1 indicates that page and key sequences are the same. Unit may be thought of as the size of a control area. The

breakpoint gives the time in seconds in which the concurrent reorganization method starts to produce a higher throughput of transactions as compared with only transaction processing of the B+ tree. For the case where the unit is 15 ($\simeq$ 11 percent of the database size) we see that the breakpoint occurs around 375 seconds ($\simeq$ 2/5 of total reorganization time). The columns under max decrease indicate the maximum number of transactions delayed and the time at which this maximum delay occurred. The columns under max increase show the maximum improvement of throughput and the time at which this happens, i.e. when the reorganization is completed. The first line of the table shows the result of doing off-line reorganization which requires 125 seconds. During that time period the B+ tree file could have serviced 1092 transactions. We should also point out that for a much larger database, the off-line reorganization time would be much larger and more intolerable than for our case.

The average time, for the various simulations, to complete the conversion of the B+ tree to linear hash file was approximately 961 seconds. The average load factor for the linear hash file was 66 percent with 1.03 accesses per successful search. In addition, on the average only 5.6 percent more pages were used by the linear hash file as compared with (only) the leaf pages of the B+ tree file. We see from table II that as the unit increases, so does the breakpoint and the delay in throughput while the improvement decreases. Of course, when the unit is 1, we have our best result. Although it probably is not a very practical situation. However, other situations where the unit is between 15 ($\simeq$ 11% of database) and 35 ($\simeq$ 24% of database) appear to be more practical and still yield good results, in terms of breakpoint and delay. The results are especially good when compared with the delay of off-line reorganization.

At this time, we would like to present a comparison of our analytic model with the simulation model. Table III shows the breakpoint, in number of pages, as computed using our analytic model and as observed from the simulation runs. The maximum difference between the two is only 5.5% and on the average it is less than 3%. So, even though we made several simplifying assumptions for our analytic model, the value of the breakpoint predicted was fairly close to the observed value. The difference is probably due to the fact that there is a small amount of overflow with the linear hash file and that the time to convert and split a page is not constant.

Table II. Summary of simulation results

unit	break point (sec)	max decrease		max increase	
		trans	time (sec)	trans	time (sec)
offline		1092	125		
1	284	63	132	3134	958
5	314	75	168	2980	958
10	336	92	173	2816	962
15	375	111	175	2686	967
20	413	126	232	2448	961
25	434	130	228	2285	954
30	453	164	180	2124	960
35	485	193	208	1949	962
40	538	223	240	1959	969
50	575	294	308	1462	951
71	625	438	452	832	964
100	800	642	665	774	963
121	925	805	817	271	968

Table III. Comparison of analytic model & simulation results

unit	breakpoint (pages)		% diff.
	simulation	analytic model	
1	46	44	4.55
5	49	48	2.08
10	53	53	0.00
15	58	57	1.75
20	62	62	0.00
25	67	65	3.08
30	70	70	0.00
35	75	75	0.00
40	82	82	0.00
50	89	86	3.49
71	97	100	-3.00
100	120	127	-5.51
121	138	143	-3.50

In addition, graphs for a few of the simulation runs appear in figures 4, 5 and 6.

6. CONCLUSION

The motivation for this work has been to show that the conversion of the B+ tree file to a linear hash file is reasonably done concurrently with user transaction processing. The conversion is necessary for improving the performance of the database system and doing the conversion concurrently with database usage is necessary for any system which must be available 24 hours a day. We devised an algorithm which performs the conversion and introduced an analytic model. We also used a typical database simulation model and ran various experiments. The results of the experiments support the idea of doing file conversion concurrently with database usage especially when compared to an off-line reorganization approach. In future work, we will examine the process of converting a linear hash file to a B+ tree file concurrently with usage.

□ = concurrent reorganization & database processing
Δ = database processing only

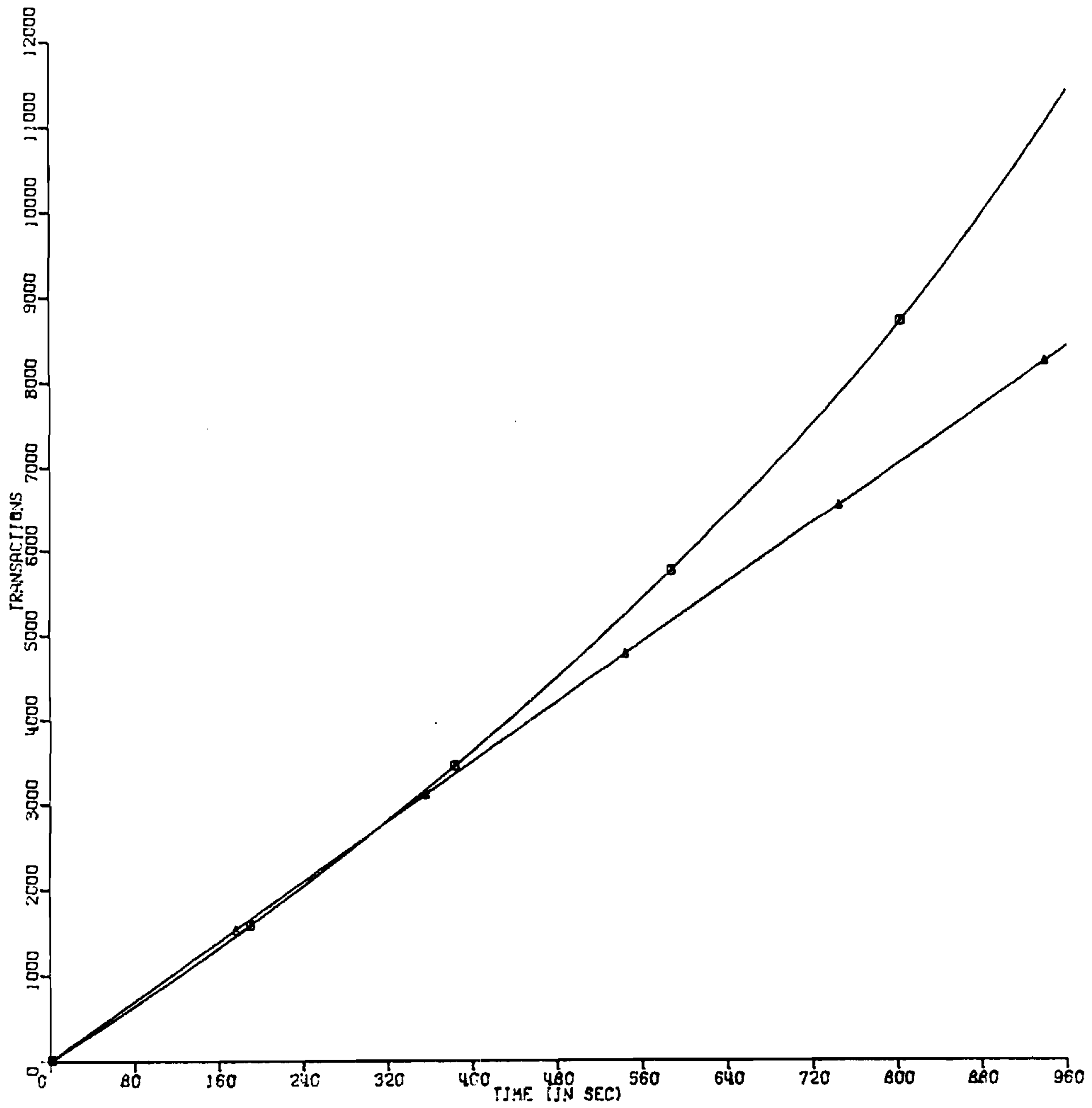


Fig. 4. Throughput graph for unit = 5

□ = concurrent reorganization & database processing
Δ = database processing only

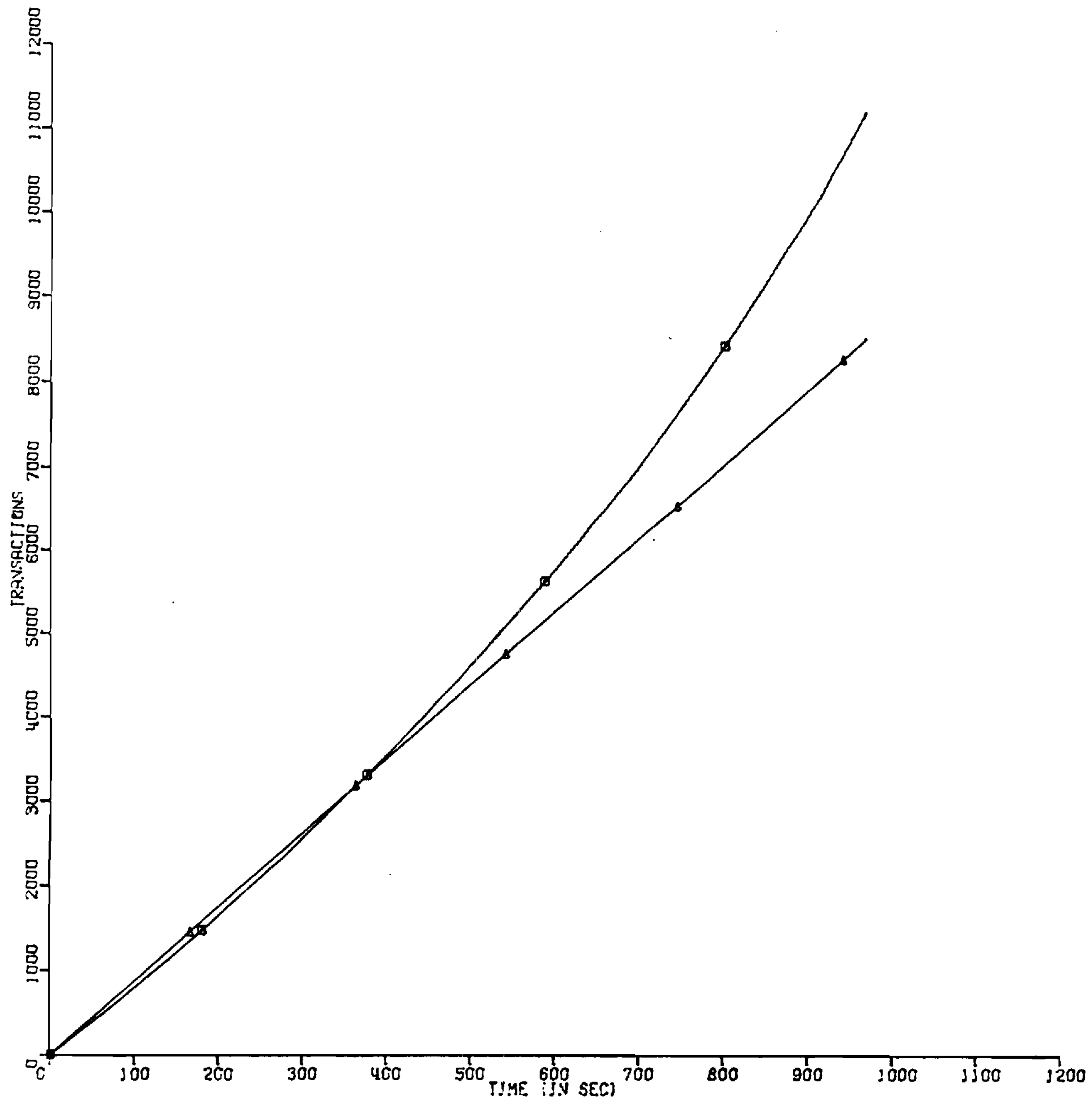


Fig. 5. Throughput graph for unit = 15

□ = concurrent reorganization & database processing
Δ = database processing only

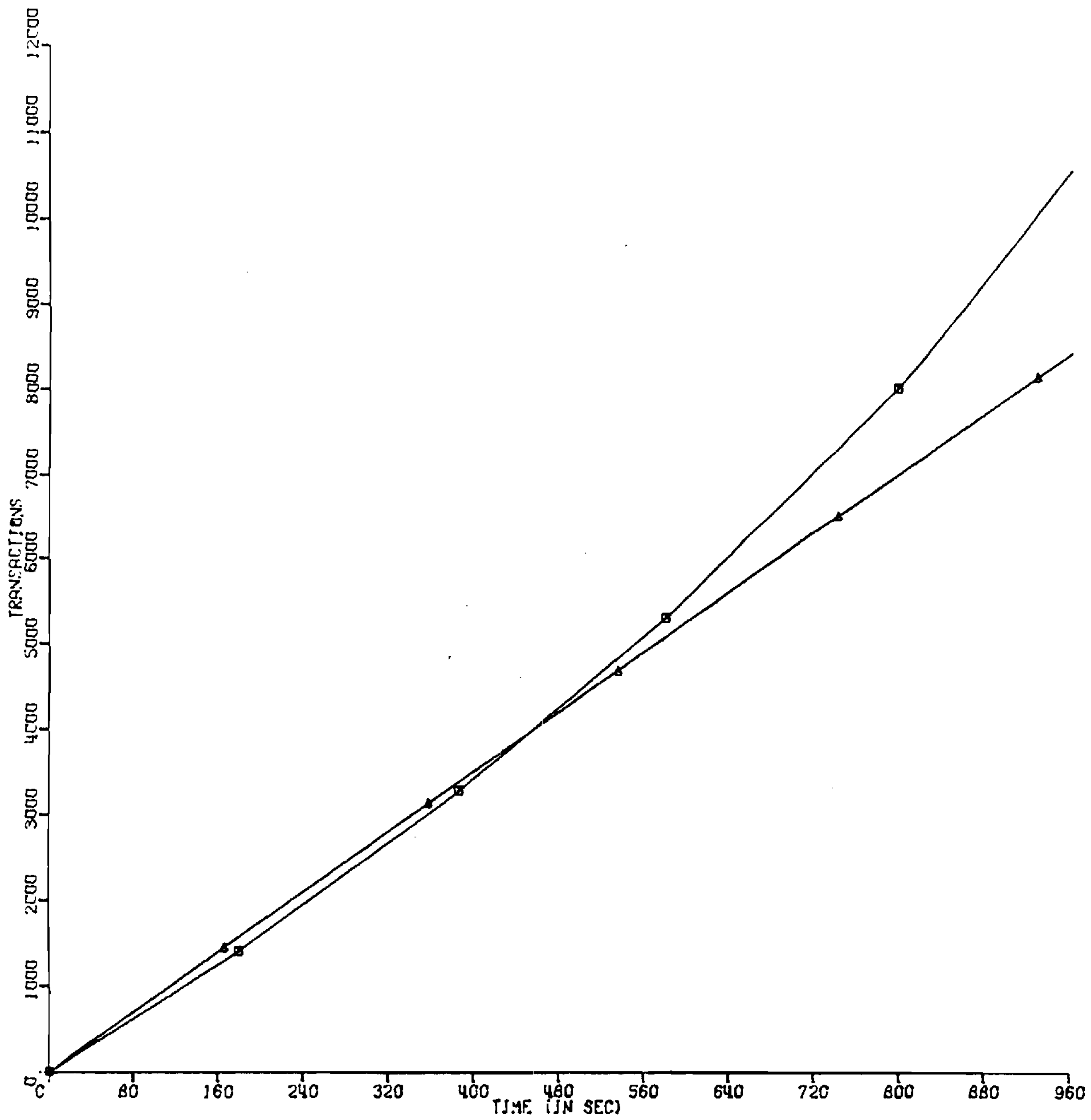


Fig. 6. Throughput graph for unit = 30

REFERENCES

1. Agrawal, R., Carey, M. and Livny, M., "Models for Studying Concurrency Control Performance: Alternatives and Implications," 1985 SIGMOD Conference Proceedings, ACM, May 1985, 108-121.

2. Astrahan, M., et al., "System R: Relational Approach to Database Management," ACM TODS, 1, 2, June 1976, 97-137.

3. Comer, D., "The Ubiquitous B-Tree," Computing Surveys, ACM, 11, 2, June 1979, 121-137.

4. Ellis, C., "Concurrency and Linear Hashing," Technical Report, Computer Science Dept., University of Rochester, NY, March 1985.

5. Kim, W., "Highly Available Systems for Database Applications," Computing Surveys, ACM, 16, 1, March 1984, 71-98.

6. Knuth, D., The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, Reading Mass., 1969.

7. Kronsjo, L., Algorithms: Their Complexity and Efficiency, John Wiley and Sons, New York, 1979.

8. Kwong, Y. and Wood, D., "A New Method for Concurrency in B-trees," IEEE TSE, 8, 3, May 1982, 211-222.

9. Larson, P., "Linear Hashing with Partial Expansions," 1980 VLDB Conference Proceedings, 1980, 224-232.

10. Larson, P., "Linear Hashing with Overflow Handling by Linear Probing," ACM TODS, 10, 1, March 1985, 75-89.

11. Lehman, P. and Yao, S., "Efficient Locking for Concurrent Operations on B-trees," ACM TODS, 6, 4, Dec. 1981, 650-670.

12. Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," 1980 VLDB Conference Proceedings, 1980, 212-223.

13. Omiecinski, E., "Incremental File Reorganization Schemes," 1985 VLDB Conference Proceedings, 1985, 346-357.

14. Omiecinski, E., "Concurrency During the Reorganization of Indexed Files," 1985 COMPSAC Proceedings, 1985, 482-488.

15. Ouksel, M. and Scheuermann, P., "Storage Mappings for Multidimensional Linear Dynamic Hashing," 1983 PODS Conference Proceedings, ACM, 1983, 90-105.

16. Ramamohanarao, K. and Lloyd, J., "Dynamic Hashing Schemes," The Computer Journal, 25, 4, 1982, 478-485.

17. Ramamohanarao, K. and Sacks-Davis, R., "Recursive Linear Hashing," ACM TODS, 9, 3, Sept. 1984, 369-391.

18. Reingold, E., Nievergelt, J. and Deo, N., Combinatorial Algorithms, Prentice-Hall, Englewood

Cliffs, NJ, 1977.

19. Ries, D. and Stonebraker, M., "Locking Granularity Revisited," ACM TODS, 4, 2, June 1979, 210-227.

20. Sockut, G. and Goldberg, R., "Database Reorganization - Principles and Practice," ACM Computing Surveys, 11, 4, Dec. 1979, 371-395.

21. Soderlund, L., "Concurrent Database Reorganization - Assessment of a Powerful Technique through Modeling," 1981 VLDB Conference Proceedings, 1981, 499-509.

22. Stonebraker, M., Wong, E., Kreps, P. and Held, G., "The Design and Implementation of INGRES," ACM TODS, 1, 3, Sept. 1976, 189-222.