Bulletin of the Technical Committee on

Data Engineering

June, 1996 Vol. 19 No. 2

IEEE COMPUTER SOCIETY 50 YEARS OF SERVICE • 1946-1996

Letters

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

Letter from the Editor-in-Chief

Bulletin Related Items

The last few years in the history of the Data Engineering Bulletin have seen great changes. The Bulletin was revived in December 1992 and began publication electronically in 1993. The initial associate editors were holdovers from the prior incarnation of the Bulletin. Their experience with the Bulletin proved very valuable in smoothing the transition to the new dual publishing mode.

In 1993, I appointed the current editors, and they continued the role of providing special issues on timely subjects from the leading researchers in the field. During this period, the Bulletin also published articles from people at commercial database companies. Now, the terms of this group of Associate Editors is coming to a close. I want to take this occasion to thank them for their very successful efforts in making the Bulletin an exciting and valuable publication. The retiring editors are Mei Hsu, Goetz Graefe, Jennifer Widom, Shahram Gandeharizadeh, and Eliot Moss. It is only by efforts such as theirs that this field can continue to prosper as an intellectual arena and with a lively exchange of ideas.

I am currently in the process of appointing new Associate Editors. The first appointee, and the editor of this issue is Betty Salzberg, from Northeastern University. As many of you know Betty is a long-time colleague of mine. Her work in access methods, dynamic reorganization, and workflow is of the highest quality. I am delighted that she has agreed to serve as an editor.

The second new editor I can announce with this issue is Mike Franklin, from the University of Maryland. Mike and I have a long association, dating back to the period in which he was a graduate student at the Wang Institute and I was on the faculty there. Mike graduted from Wisconsin a few years ago and is expert in caching behavior for object oriented systems as well as systems aspects for a variety of database related systems.

About this Issue

In this issue, Betty Salzberg has assembled articles from both the university and commercial world on the topic of on-line reorganization. This involves such physical database elements as indexing and data partitioning. Reorganization is essential for systems to realize ultra high availability. Downtime, whether it is scheduled or unexpected, is no longer permissible in many “bet your business” applications. What you will see in this issue is both the latest in commercial practice as well as the latest thinking from the research world. I want to thank Betty for doing a fine job on an issue that you, the reader, should find of both intellectual and practical interest.

David Lomet
Editor-in-Chief

Letter from the Special Issue Editor

In this issue, we look at some existing commercial online reorganization utilities and some proposals for online reorganization. Online reorganization will become more important as more demands are made on database management systems for both performance and availability. Todays systems are straining to meet these demands.

The first two papers, by Gary Sockut and Balakrishna Iyer (in the section describing IBM's DB2) and by Jim Troisi describing Tandem's Nonstop SQL/MP illustrate a similar commercial online reorganization utility in two very different settings.

IBM's DB2 has a "clustering index" for database tables, which is a B^+-tree whose leaves contain key values and RIDs. A RID (Record Identifier) consists of a record's page number (within a region of storage) and a record slot number within the page. Once loaded in the database, a record is pinned to this location unless a database reorganization takes place. Existing data is loaded into the database in the order of the clustering index. After a time, clustering degrades because new records cannot be inserted in clustering order. In addition, when records are updated, sometimes they grow in length. If there is no room for the additional bytes on the record's page, the new record is written to a different page, and the original record is replaced by the forwarding address of the new record.

One solution to these problems of degrading performance in RID databases is described in the enclosed paper by Sockut and Iyer. The Version 4 Release 2 of DB2 recently announced will have a reorganization utility which allows concurrent reading and writing by users while creating a new copy of the table and creating new secondary (non-clustering) indexes for the table.

Tandem, on the other hand, provides primary B^+-trees where the leaves of the tree contain the actual data records of the database table. Secondary B^+-trees have pairs of secondary keys and primary keys in their leaves. Thus, the data records can change their physical location (for example, when a primary B^+-tree leaf splits) without having to update entries in secondary indexes. Tandem also provides horizontal fragmentation of tables based on ranges of the primary B^+-tree. The paper by Troisi explains how to change the horizontal fragmentation boundaries, creating new fragments, by moving some records to a new location. This can be used to improve load balancing.

Both the Tandem and the DB2 reorganization utilities copy records to a new location while allowing users to read and write to the old location. They both use the database log to capture the changes made to the table after the reorganization has begun. After catch-up, they forbid updates for a short time and switch to the new organization.

The paper from Sybase by T. K. Rengarajan, Lucien Dimino, and Dwayne Chung concentrates on Sybase utilities which make some kinds of online reorganization unnecessary. Sybase has spent a lot of engineering effort on managing the physical resources like cpu, disks, and memory online. This is quite critical for performance tuning of databases, while they are being used. In addition, the Sybase Replication Server can maintain a copy of the database which can have a different design and a different set of indexes from the original. It can thus be considered as a copy that can be changed without stopping update activity. (The same can be said of IBM's DataPropagator Relational and Replidata/MVS as described in the article by Sockut and Iyer.)

Academic and industrial research papers such as those summarized here by Edward Omiecinski and by Chendong Zou and myself have tended to focus on the scenario where there is not enough room to have two full copies of the data being reorganized. They therefore concentrate on small units of reorganization. They may try to minimize the locks being held and the amount of I/O necessary for reorganization.

Omiecinski has written a number of papers summarized nicely in his article. He has looked at changing from one organization to another, such as from B^+-trees to linear hashing, repartitioning data in parallel database systems, compacting indexed sequential files and creating clustering organizations described by hypergraphs.

Zou and Salzberg describe a method for deferring secondary index updates which could be used for any record-moving reorganization. They have also investigated how to compact a primary B^+-tree which had become sparse. This is separated into small reorganization units when working with the leaves, but uses a copy and switch discipline when working with the upper levels of the tree. Zou and Salzberg as well as Mohan and Narang (whose work in online reorganization at IBM is described in the paper by Sockut and Iyer) have also considered making online reorganization restartable—if there is a system failure before reorganization is complete, not all work is lost.

As we can see, major DBMS providers regard online reorganization as a priority. It is demanded by more and more customers who want both performance and availability. We are fortunate to have a number of articles here which illustrate the state-of-the-art in online reorganization. I would like to thank all the authors for their efforts on our behalf.

Betty Salzberg
Associate Editor

A Survey of Online Reorganization in IBM Products and Research

Gary H. Sockut and Balakrishna R. Iyer [*]

IBM Santa Teresa Laboratory
P. O. Box 49023
San Jose, CA 95161-9023 USA

1 Introduction

Here we survey work that has taken place in online reorganization in IBM[1] products and IBM research. The products include Database 2 (DB2) for MVS/ESA (a relational DBMS), DataPropagator Relational (a replication facility), Replidata/MVS (another replication facility), the PARS/ACP airline reservation system, the IMS Fast Path DBMS, and the concurrent copy facility for creating a backup copy. The types of reorganization that we discuss are restoration of clustering, purging of old data, creation of a backup copy, compaction, and construction of indexes. A technical report [12] surveys IBM and non-IBM work in online reorganization.

2 Restoration of Clustering for DB2 for MVS/ESA

Database 2 (DB2) for MVS/ESA [2] is a relational DBMS. The sections below describe several methods of reorganization for DB2, roughly in increasing order of concurrency.

2.1 Clustering and Reorganization

In DB2, each table can have a *clustering index*. DB2 tries to assign data records to pages to reflect the data records' order in the key (set of columns) of the clustering index. Periodically, reorganization is necessary to restore clustering, distribute free space evenly, and remove overflows. DB2 has supported reorganization that operates by unloading from the table being reorganized (while users can read but cannot write the table), sorting the unloaded data, reloading into that table (while users have no access), and then allowing read/write access again. A backup copy of the table should also be created, as a basis for future recoverability.

DB2 supports division of a table into partitions, based on ranges of key values. Partitions reside in separate files. DB2 can reorganize one partition offline while the other partitions are online.

Next we will describe ways to increase the concurrency during reorganization.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*] Starting in September 1996, G. H. Sockut's address is School of Management, Boston University, Commonwealth Ave., Boston, MA 02215 USA. Correspondence about this article should go to B. R. Iyer.

[1] DATABASE 2, DataPropagator, DB2, IBM, IMS, and MVS/ESA are trademarks of the International Business Machines Corp.

2.2 Read-Only Access While Reorganizing into a Shadow Copy

Mix [6] describes a strategy that extends read-only access to cover the reloading step. This strategy, which is not part of the DB2 product, does *not* require any modification to DB2's normal facility for reorganization. Here are the main steps of the strategy:

1. Quiesce writing of the area to be reorganized, while continuing to allow users to read the area. This quiescing blocks new writing, and it waits for existing writing to finish.
2. Apply the unloading step of DB2's normal reorganization facility, while continuing to allow read-only access to the area.
3. Reload the data into a *shadow* copy of the area. Continue to allow read-only access to the original copy of the area.
4. Quiesce reading of the original copy.
5. By renaming the files that underlie the area being reorganized, change the logical-to-physical mapping. This change will cause DB2 to direct users' future accesses into the shadow copy. The area is offline during this brief step.
6. Create a backup copy of the shadow copy (as a basis for future recoverability), and allow users to read and write the shadow. Here are two possible sequences for this step:
   - Allow users to read the shadow, create a backup copy while allowing just reading, and then allow users to read and write the shadow (after the backup completes).
   - Use a strategy (e.g., concurrent copy [4], described in Section 3.3) that allows users to read and write the shadow as soon as the backup begins.
7. Delete the original copy.

Mix's description includes the tasks that a database administrator performs.

2.3 Read/Write Access While Reorganizing and Using Replication

DataPropagator Relational [3] and Replidata/MVS [5] are products that support replication of DB2 databases. They are *not* facilities for reorganization, but either of them can be used to achieve the effect of reorganization that allows read/write access. They require a table to have a unique key.

A database administrator can perform the following steps to reorganize using a replication facility:

1. Initially, allow users to read and write the primary copy of the area to be reorganized; use the other copy as a backup.
2. Reorganize the backup copy offline, using DB2's normal reorganization facility, while users can read and write the primary copy.
3. Use the replication facility to copy recent writing from the primary copy to the backup copy, while users can read and write the primary copy.
4. After the copying has caught up with the writing of the primary copy, quiesce writing of the primary copy, while continuing to allow reading of the primary copy.
5. Use the replication facility to copy any very recent writing from the primary copy to the backup copy, while users can read the primary copy.

6. Quiesce reading of the primary copy.

7. Exchange the roles of the primary copy and the backup copy.

8. Allow users to read and write the primary copy, which was originally the backup copy.

We can then apply the same strategy for reorganization of the backup copy, which was originally the primary copy.

2.4 Read/Write Access While Reorganizing into a Shadow Copy

IBM recently announced the future availability of Version 4, Release 2 of DB2. This release includes reorganization that allows concurrent reading and writing by users. This reorganization does *not* require a unique key. The reorganization operates by unloading data from the original copy of the area being reorganized, reloading the data (in reorganized form) into a shadow copy, applying the database log to bring the shadow copy up to date (to reflect writing that users have performed in the original copy), and exchanging the names of the underlying files to switch users' accessing from the original copy to the reorganized shadow copy.

Here are the steps in more detail:

1. Record the current position in the log. Users use DB2's normal facilities to read and write the area. In the log, DB2's normal facilities also append log entries that correspond to users' writing.

2. Unload the data from the original copy of the area, sort the data by the key used for clustering, and reload the data into the shadow copy, in sorted (reorganized) order. Also, create shadow copies of the indexes. Also, create a backup copy of the shadow copy of the area, as a basis for future recoverability. Continue to allow read/write access to the original copy of the area.

3. Process the log, starting from the recorded position. This processing consists of reading the log entries and applying them to the shadow copy of the area and the shadow indexes. These activities bring the shadow copy up to date, to reflect writing that users have performed in the original copy. Continue to allow read/write access to the original copy. This step can execute iteratively; each iteration after the first one processes the writing that users performed during the previous iteration. Stop the iterations when DB2 estimates that the next iteration's duration will be within a threshold. At the end of the last iteration of this step, bring the backup copy up to date by appending to it the shadow pages that have changed due to log processing.

4. Quiesce writing of the area. Continue to let users read.

5. Process the log one last time, allowing read-only access. Again, bring the backup copy up to date by appending to it the shadow pages that have changed recently due to log processing.

6. Quiesce all access of the area.

7. Switch users' future accessing from the original copy to the reorganized shadow copy, by exchanging the names of the files that underlie the original and shadow copies of the area and the indexes. This renaming effectively changes the mapping from logical to physical. During this step, users have no access to the area.

8. Allow read/write access of the area to resume. Users can then use DB2's normal facilities to read and write the shadow copy, in the same way that they formerly read and wrote the original copy.

9. Delete the original copy.

An entry in the log identifies a data record by the record's identifier. As an inherent part of reorganization, the identifiers change. Log entries in the method for reorganization correspond to users' writing of the original copy and thus use the old identifiers. To apply a log entry to the shadow copy, we must identify the record in the shadow copy to which the entry should apply. The method for reorganization solves the problem of identification by maintaining a temporary table that maps between the old and new identifiers. DB2 uses this table to translate the identifiers in log entries before applying the entries to the shadow copy.

3 Online Reorganization for Other Products

We now turn from DB2 to other products.

3.1 Purging of Old Data in PARS/ACP

In the PARS/ACP system for airline reservations [10, 11], flights for the next *n* days have a fine index (for quick access), and flights for later days have a coarse index (to save space). Every night, the system purges records for flights that have already flown, and it constructs fine indexes for the flights for the *n*th day. The granularity of locking is fine enough to limit blockage of users to just a few seconds. The system marks the space for the purged records as deleted but does not yet make it available for reallocation. Another online process scans the records and indicates in an allocation table that the space for the purged records is available for reallocation. This process, which typically requires 3 to 20 hours, executes at least once a week. In each area of the database, uniformity of the size of records obviates any compaction. This strategy for purging is still in use.

3.2 Offline Reorganization of a Partition in IMS Fast Path

In the IMS Fast Path DBMS [13], reorganization removes storage fragmentation and makes the physical sequence of data reflect the logical sequence. A file can be partitioned by key range, and a partition can be reorganized offline while the other partitions are online.

3.3 Online Creation of a Backup Copy of Data in Concurrent Copy

Concurrent copy [4], which uses both hardware (in a storage control unit) and software, produces a backup copy of data. The copy reflects the state of the database at the start of copying. Initialization of concurrent copy involves (1) quiescing writing while continuing to allow reading and then (2) initializing the copying and allowing writing again. During the copying, for the first writing (if any) of each not-yet-copied disk track, concurrent copy saves the old version of the track in a file and then performs the writing. When producing the copy, concurrent copy uses the old version of each written track. This product can be used in connection with DB2.

4 Research in Online Reorganization

Besides the work described above for products, work has also taken place in research.

4.1 Incremental Restoration of Clustering Within Users' Transactions

Bennett and Franaszek [1] describe a strategy for incremental restoration of clustering within users' transactions. The concepts in the strategy can apply to any storage hierarchy in which a unit of data transfer between two levels in the hierarchy contains smaller units. In the description of the strategy, a block (a unit of transfer from secondary storage into main storage) contains pages. After a block is brought into main storage, its pages are then managed individually; i.e., they can be paged out separately. The strategy for clustering involves only the blocks that are already in main storage. Each process has an associated clustering task, which performs a sequence of exchanges of the assignments of pages to blocks. Each exchange involves (1) a page that a process references and (2) a page in a cluster point, i.e., an unreferenced page. A *cluster point* is the set of unreferenced pages in a block that contains a referenced page. Each process can have several cluster points.

Here are the main characteristics of the strategy:

- For each page, we assign the page to a process (i.e., the clustering task for that process is responsible for improving the clustering of that page) if that process was the first process to reference the page during its current stay in main storage.
- For each process, the first cluster point is the set of unreferenced pages in the block that contains the page that causes the first page fault for the process.
- As we assign pages to a process, we append their identifiers to a list associated with the current cluster point.
- After the size of the list grows to the size of the cluster point, we select the next cluster point for the process as the set of unreferenced pages in the block that contains the page that causes the next page fault. This cluster point will have its own list.
- When a page from a cluster point is about to be paged out, we exchange its block assignment with a page in the associated list. Thus the block containing the cluster point accumulates (and clusters) referenced pages.
- When the last page from a cluster point is paged out, we delete the list for the cluster point.

Bennett and Franaszek report experiments that use a trace of 3 days of transactions on a database (6 million database references, with an average of 38 references per transaction). The physical record size (1693 bytes) constitutes 1 page, and a block contains 8 pages. The size of main storage varies from 1024 to 16384 pages. For low storage sizes, use of reorganization lowers the miss ratio (the fraction of references that cause page faults) by about 8%. For high sizes, reorganization has little effect. If we stop reorganization after a period, the miss ratio with reorganized data is still less than the miss ratio for unreorganized data for a while, but later it is not less. Therefore, the optimal organization is time-varying.

4.2 Online Compaction in a Log-Structured File System

In a *log-structured* file system, the mechanism for logically modifying a preexisting block of data is to append a new block of data, thus invalidating and deallocating the preexisting block. Over time, this appending can fragment the free space. Therefore, a *reorganizer* (process that performs reorganization) compacts the valid data, merging free areas into larger free areas. Of course, this reorganization consumes disk bandwidth and thus degrades system performance.

If a reorganization algorithm moves only a subset of the data instead of all the data, this reduction in movement might improve the performance. Robinson and Franaszek [9] model the performance of a log structured file system and the performance improvement that results from moving only a subset of the data. The analytic model, which can apply to any criterion for choosing the subset, predicts how online reorganization affects available disk bandwidth.

In the model, a read or the first write after a read is random access and thus includes a seek, latency, and data transfer. A write after a write omits the seek, and it also omits the latency if an earlier write is still in progress. Parameters to the model include the arrival rate, the probability that an access is a read, the disk storage utilization, and the times for a seek, latency, and data transfer.

Robinson and Franaszek develop formulas for the mean service time and the number of blocks that the reorganizer can move during any idle period. They also calculate the rate of reorganization that is required to keep up with a rate of writing. The need for reorganization limits the maximum steady-state arrival rate of users' disk requests.

The model predicts that the maximum permitted steady-state arrival rate decreases as disk storage utilization increases, since the increased utilization requires more reorganizations. Reduction in the amount of movement required by reorganization increases the maximum permitted steady-state arrival rate.

4.3 Online Construction of Indexes

Mohan and Narang [8] describe two algorithms for online construction of an index. The reorganizer scans the data for key values and record identifiers, sorts them, and builds the index while periodically checkpointing the highest key inserted. The main difference between the algorithms is in their handling of user transactions' insertions and deletions in the index during construction. The algorithms include logging and are restartable. They use a restartable sorting algorithm, which avoids the need to scan the data again if a failure occurs. The sorting includes checkpointing of the sorted streams.

The paper discusses two problems that each algorithm must solve. In the *duplicate-key-insert* problem, the reorganizer and a user transaction that inserts data might try to insert the same pair of key value and record identifier in the index. This can occur because the latch on a data page is not held during insertion in the index, to avoid deadlock. In the *delete-key* problem, a race condition can cause the reorganizer's activities to straddle a user's activities. Specifically, the reorganizer extracts a key from a data page, a user transaction deletes the key from the data page (and does not find the key in the index), and then the reorganizer erroneously inserts the key in the index.

In one algorithm (called *NSF* for "no side file"), users insert and delete directly in the index; there is no separate area to save the insertions and deletions. NSF tolerates interference by users. Both the reorganizer and users write log records for activities in the index. When the duplicate-key-insert problem arises, whichever process is second (the reorganizer or a user) refrains from inserting the duplicate. Even if a user transaction refrains from inserting a key, the transaction still writes a log record, to assure that any later rollback would delete the key. To solve the delete-key problem, a user transaction that tries unsuccessfully to find a key to delete in the index instead inserts a pseudo-deleted key and writes a log record. The reorganizer will not later insert a key there.

NSF begins by quiescing writing against the index, to assure the absence of uncommitted writing. NSF then creates a descriptor for the index, making it visible to users for insertion and deletion. It then releases the quiesce. Without the quiesce, a rollback of an uncommitted transaction that inserted a record would not delete the record's key from the index. The index is not yet visible to users for reading, although it is possible to optimize the later building phase by making the index incrementally available for reading (up through the most recently completed key value). NSF then extracts the keys and record identifiers from the data (possibly including uncommitted records), and it sorts the keys.

After the sorting, NSF inserts the keys in the index (while latching index pages) and periodically commits the insertions. It can periodically record the highest key inserted, to avoid restarting from the beginning if a failure occurs. User transactions log their insertions, even if they refrain from actually inserting due to the duplicate-key-insert problem. The reorganizer logs insertions only when the insertions succeed. The behavior of a user transaction that deletes is more complex. If the key already exists in the index, the user writes a log record and modifies the key to become pseudo-deleted. The user transaction does not know whether the reorganizer has already extracted the key from the data page, so we need a pseudo-deletion instead of a physical deletion. The presence of the pseudo-deleted key will prevent the reorganizer from inserting the key. If, on a user's deletion, the key does not yet exist in the index, the user writes a log record (for deletion) and inserts a pseudo-deleted key. The log record will assure reinsertion in the index if the transaction later rolls back. The initial quiesce assures that the deleting transaction will write the log record.

After inserting all the keys, NSF makes the index visible for reading. Optionally, a background process physically deletes any pseudo-deleted keys whose deletions are known to be committed.

In the other algorithm (called *SF* for "side file"), users append insertions and deletions to a side file if the index is visible, i.e., if the reorganizer has already scanned the relevant data page. Otherwise they ignore the index being constructed. User transactions do not interfere with index construction. The reorganizer and user transactions write log records for the side file, not for the index. The duplicate-key-insert problem does not arise, since a user transaction appends an insertion to the side file only if the reorganizer has already scanned the relevant data page, which did not yet contain the inserted key. The delete-key problem does not arise, since the reorganizer processes the side file (which includes the deletion) *after* the initial construction of the index (which includes insertion of the key). When a user transaction appends a deletion to the side file, the log record includes the number of visible indexes, to assure proper rollback if necessary later.

SF creates a descriptor for the index, scans data, sorts the data, builds the index from the data (while periodically checkpointing), and performs the side file's insertions and deletions in the index (while periodically checkpointing).

SF is more efficient than NSF, involves less logging, avoids pseudo-deletions, probably yields a more clustered index (since user transactions do not interfere), and need not initially quiesce writing. However, NSF needs no side file and has less sophisticated logging and undo requirements for user transactions.

The IBM Application System/400 allows writing during construction of indexes [7]. The unpublished algorithm is similar to SF and preceded the work of Mohan and Narang.

5 Summary

We have surveyed work in online reorganization in IBM products and IBM research. Much of the work deals with restoration of clustering. The techniques include both reorganizing in place and reorganizing a copy of the original data.

Acknowledgements

A suggestion from Robert Goldberg led to the first author's early work in online reorganization. We thank Peter Franaszek, Rob Goldring, Mike Halliday, Andy Johnson, Claus Mikkelsen, Frank Mix, C. Mohan, Shep Nachbar, Inderpal Narang, Gary Ricard, John Robinson, Betty Salzberg, Chris Saul, and Joe Siwiec. These people discussed issues in reorganization and/or reviewed parts of an earlier draft of this paper.

References

[1] Bennett, B. T., and Franaszek, P. A. Permutation Clustering: An Approach to On-Line Storage Reorganization, IBM J. Research and Development, Vol. 21, No. 6, Nov. 1977, pp. 528–533.

[2] Haderle, D. J., and Jackson, R. D. IBM Database 2 Overview, IBM Syst. J., Vol. 23, No. 2, 1984, pp. 112–125.

[3] IBM Corp. An Introduction to DataPropagator Relational Release 1, GC26-3398-01, 1993.

[4] IBM Corp. Implementing Concurrent Copy, GG24-3990-00, Dec. 1993.

[5] IBM Corp. and Integrated Systems Solutions Corp. Replidata/MVS User's Guide, BLD-REP-UG-00, Jan. 1994.

[6] Mix, F. DB2 Reorg and Continuous Select, Proc. 6th Ann. N. Amer. Conf. Intl. DB2 Users Group, IDUG, Chicago, May 10, 1994, pp. 545–563.

[7] Mohan, C. IBM's Relational DBMS Products: Features and Technologies, Proc. 1993 ACM SIGMOD Intl. Conf. Management of Data, May (SIGMOD Record, Vol. 22, No. 2, June), pp. 445–448.

[8] Mohan, C., and Narang, I. Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates, Proc. 1992 ACM SIGMOD Intl. Conf. Management of Data, June (SIGMOD Record, Vol. 21, No. 2), pp. 361–370.

[9] Robinson, J. T., and Franaszek, P. A. Analysis of Reorganization Overhead in Log-Structured File Systems, Proc. 10th Intl. Conf. Data Engin., IEEE-CS, Feb. 1994, pp. 102–110.

[10] Siwiec, J. E. A High-Performance DB/DC System, IBM Syst. J., Vol. 16, No. 2, 1977, pp. 169–195.

[11] Sockut, G. H., and Goldberg, R. P. Database Reorganization — Principles and Practice, Computing Surveys, ACM, Vol. 11, No. 4, Dec. 1979, pp. 371–395.

[12] Sockut, G. H., and Iyer, B. R. Reorganizing Databases Concurrently with Usage: A Survey, Tech. Report 03.488, IBM Santa Teresa Lab., San Jose, CA, June 1993.

[13] Strickland, J. P., Uhrowczik, P. P., and Watts, V. L. IMS/VS: An Evolving System, IBM Syst. J., Vol. 21, No. 4, 1982, pp. 490–510.

NonStop SQL/MP Availability and Database Configuration Operations

Jim Troisi [†]
Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599

1 Introduction

Application outages cost customers money. Big money. It has been estimated that the cost associated with application outages is over a thousand U.S. dollars per minute of downtime. To minimize the downtime associated with hardware and software failures, Tandem^{tm} Computers has been shipping fault tolerant systems for nearly 20 years. These systems have a shared nothing architecture that, in conjunction with the Tandem operating system, guarantee that Tandem systems continue running after any single hardware failure. Figure 1 shows a typical Tandem system block diagram. A 3 CPU system is detailed though systems can be as large as 4096 CPUs.

While the Tandem systems architecture minimizes planned and unplanned hardware outages, software configuration outages are handled at a higher level. Specifically, for database applications, Tandem systems provide a client/server based transaction monitor facility and transaction management system. If a CPU fails, for example, these systems abort all transaction being worked on by servers in that CPU and the client software restarts the transaction. In addition, online hardware configuration (addition, deletion, and reconfiguration of hardware components), online software installation, and online physical file reorganization are available to minimize the outages associated with database reorganization.

The remainder of this article talks about the online partition operations found in Tandem's relational database management system, NonStop SQL/MP. Database configuration and reconfiguration operations can have a significant effect on the availability of user applications. Although most users perform these operations infrequently, their duration can account for thousands of minutes of application outage per year. The enhanced data configuration features in the Tandem NonStop^{tm} SQL/Massively Parallel (SQL/MP) relational database management system eliminate most of this downtime. Thus, a user with an application requiring permanent, continuous availability could save 3.5 million dollars per year by using the new NonStop SQL/MP features [1].

This article focuses on one changed database configuration operation, Split Partition. The article explains each phase of the operation and shows how it maintains application availability. (The few required outages are noted, together with estimates of how long they last.) The other database configuration operations function in a similar way and have a similar effect on availability. Thus, the description of Split Partition also applies, for the most part, to the other operations.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[†] Current address: Informix Software,921 SW Washington Street Suite 670, Portland, Oregon 97205
[1] A discussion of the cost of application outages appears in [2].

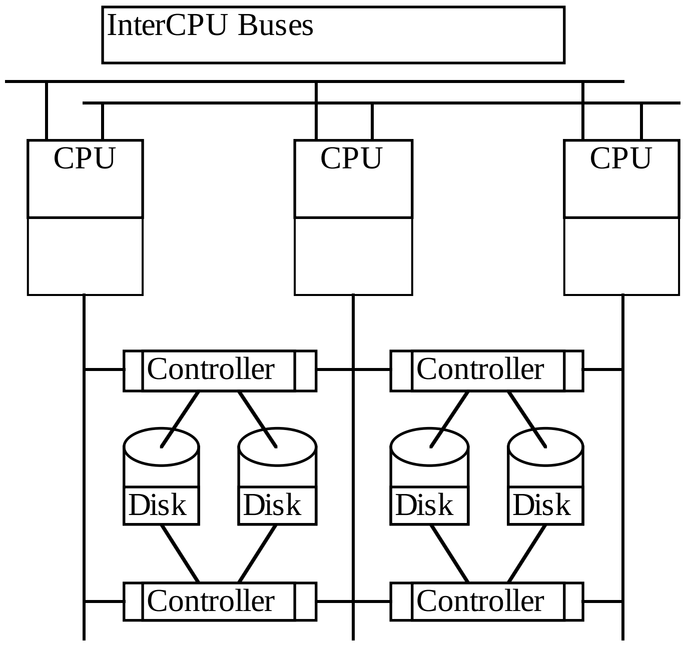


Figure 1: A Tandem System

2 Physical Database Configuration Features

In NonStop SQL/MP, partitions are used to create a large logical table from multiple smaller physical files and to enhance performance by balancing the I/O workload among multiple disk devices. Currently, a partition can grow to 2 gigabytes and a base table or index can consist of up to 286 [2] partitions, the exact number of partitions depending on key size. One can enlarge a table by splitting a partition and moving part of the data from the original partition into another newly create partition, or by adding a new partition where no data movement is required. The Add Partition command, available in previous releases of NonStop SQL, accomplishes the second of these operations; it requires minimal outage times. Move Partition moves a partition that resides on one disk to another disk. Create Index orders data according to the value specified in the key columns. As a result, queries that use these key columns run quickly.

A new command for NonStop SQL/MP, Move Partition Boundary, moves rows between adjacent partitions, typically shipping rows from nearly full partitions into less full partitions. One can use this command to delete a partition by moving all of its rows into an existing adjacent partition.

As part of Tandem's NonStop Availability (NSA) Initiative, the SQL/MP developers added new options to the Split Partition, Move Partition, Create Index commands and Move Partition Boundary. One of these new options, With Share Access, allows concurrent read and write transactions to execute during the reconfiguration operation.

The new implementations of these commands improve database availability. They minimize, but do not eliminate, associated outages. Even with the improvements in NonStop SQL/MP, user database activity is restricted for about one minute (or less) per operation. The outage time varies depending on the number of user transactions running against the table, the size of the transactions, and the number of partitions in the affected table. For example, with long-running user transactions, these commands need more downtime because they require locks that cannot be granted until all outstanding transactions against the source table have been completed. For this reason, one should run these commands at times when the user workload against the database is at a low point.

[2] The limit of 286 partitions will be raised in a future release.

(Consult the documentation on NonStop SQL/MP for additional considerations.)

These NonStop SQL/ MP operations use a technique that requires the use of the Tandem Transaction Manager/Massively Parallel (NonStop TM/MP) audit trail [3]. Tandem Systems support both audited and unaudited tables. When the With Share Access Option is specified, the operations work only on audited tables; they cannot work when the source table is unaudited. When the With Share Access options is not specified, the Split Partition, Move Partition, and Create Index operations work for both audited and unaudited tables.

These reconfiguration operations work equally well for managing indexes as for base tables. In NonStop SQL/MP, indexes are stored SQL tables. Records of an index consists of the alternate key value and the associated logical primary key value for each record in the base table. Since no physical link information is kept in the index, the reconfiguration operations can repartition a base table without making any changes to the associated indexes. Similarly, index partitioning can occur without any required changes to the associated base table.

3 The NSA Split Partition Operation

When splitting a partition, one decreases the size of the source partition; some percentage of the source partition's data goes to the new partition. Previous releases of NonStop SQL provided the Split Partition command, which allowed concurrent read access to the partition being split and read and write access to all other partitions. In contrast, the NSA Split Partition operation, available in NonStop SQL/MP, allows complete read and write access to all partitions during the split operation.

The NSA Split Partition operation works in four phases. The phases guarantee that modifications (inserts, updates, and deletes) made to the data in the original table by user applications are present in the table after the split finishes executing.

Figure 2 shows an example of a table partition being split to accommodate growth in user data. The table's primary key is the user's U.S. state postal code. The table has three partitions, one for states whose two-letter postal code precedes HA, one for states whose postal code is HA through NZ, and one for the states after that. Because sales in New Jersey (NJ) have far exceeded expectations, the second partition of the table is becoming full. Therefore, the user wants to split this partition, moving the entries from NJ through NZ into a new partition. Figure 2 shows the partitions in the original table and the partitions planned for the new table.

3.1 Phase 1: Sequential Read

In the first phase, Sequential Read, of an NSA Split Partition operation, the SQL catalog manager (SQLCAT) process creates the new target partition. Since a new partition is not yet logically a part of the table's definition, user applications cannot use it.

The SQLCAT process then sequentially reads the existing (original) partition and writes the relevant portions of its data to the new target partition. (The reads begin at the location at which the data is split.) The SQLCAT process performs the reads without holding database locks. This technique enables user applications to run concurrently without experiencing additional lock contention.

This technique, however, allows inconsistent data to be copied to the target partition. (Another name for this technique, "dirty reads," comes from this inconsistency.) Assume, for example, that a row is copied that is part of a user transaction and the transaction is later aborted. Moreover, since other applications can continue to insert, update, delete rows already read by the sequential-reading SQLCAT process, the data may be incomplete or even incorrect. In the second phase of this operation, the audit for the table is read to correct this situation. To prepare for the second phase, the Split Partition operation notes the current end-of-file (EOF) of the audit trail for the original partition before the sequential reads begin.

[3]The NonStop TM/MP product is the new version of TMF^{tm} (Transaction Monitoring Facility) software. See [1] for a description of the new features found in NonStop TM/MP.

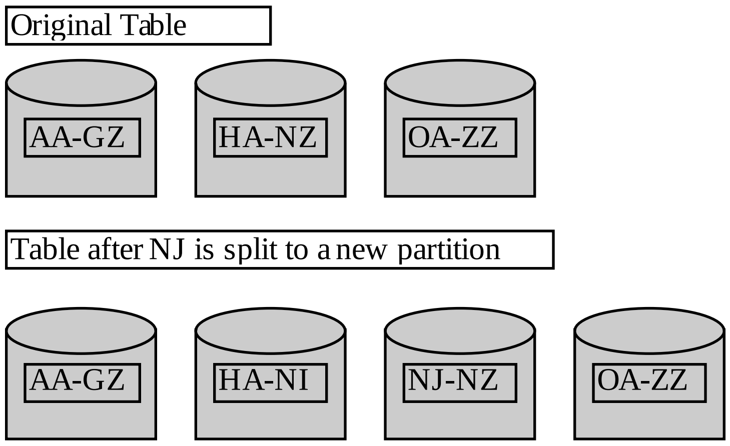


Figure 2: Table partitioning before and after the Split Partition operation.

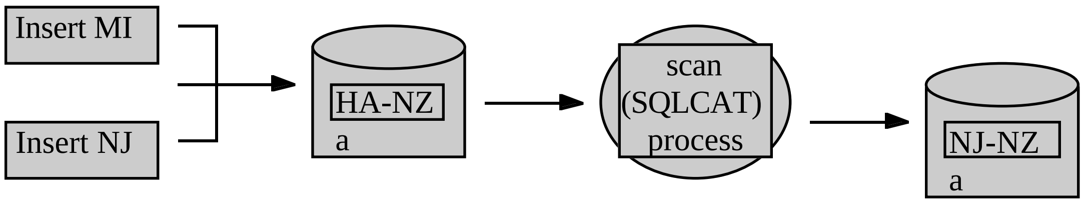


Figure 3: The user application inserts records while the table is sequentially read.

Figure 3 shows the sample table in phase 1, during which the NJ through NZ data is split off from the original partition to a new disk. To understand how availability is maintained during the operation, assume that while the sequential reader is running, two rows (containing the keys MI and NJ) are inserted into the original partition. Assume also that the new NJ row is added after the reader has already read all of the NJ rows and written them to the new partition. Under these circumstances, the new NJ row is not copied to the target partition during phase 1.

During phase 1, user applications have complete read and write access to the data in all partitions of the table. Selects, updates, inserts, and deletes are executed against the original table.

3.2 Phase 2: Audit Fixup 1

In the second phase, Audit Fixup 1, the dirty copy of the data is brought up-to-date with the original partition's data. This is accomplished by examining the audit trail for the original partition to find records that have changed since the dirty copy was made. In this phase, the audit-fixup process reads the audit trial, starting with the first record inserted into the audit trail after the dirty read began. From these audit records, the audit-fixup process identifies any changes (inserts, updates, or deletes) made to the data that also resides in the new partition. It then applies those changes to the data in the new partition.

This phase ends when the audit-fixup process reads the last record in the audit trail. At this point, the data in the target partition looks exactly like the data in the original partition except for uncommitted transactions. To find these last changes, the audit-fixup process continues to read and apply audit, as needed, to the new partition.

Figure 4 shows the sample table in phase 2. The audit-fixup process finds the new NJ and MI records in the audit trail. Since the MI record will not reside in the new partition, the process ignores it. The NJ record,

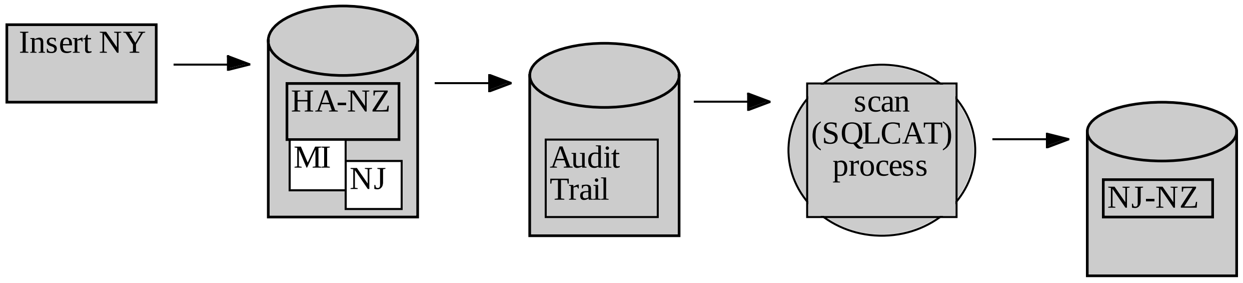


Figure 4: The user application adds records as the audit-fixup process reads audit.

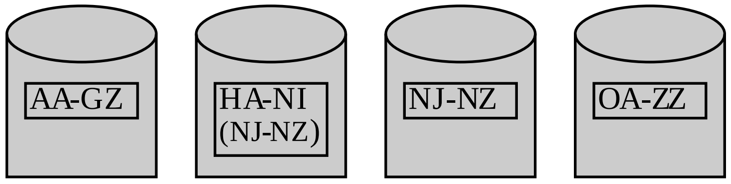


Figure 5: The table's partition after phase 3.

however, should reside in the new partition, so the audit-fixup process applies this change to the new partition.

However, the target partition still may not be an exact copy of the original one. Assume, for example, that as soon as the audit-fixup process reaches the end of the audit trail, a new record, NY, is added to the original partition. If this record is inserted after the audit-fixup process reaches the audit EOF, the record is not copied to the target partition during phase 2.

During phase 2, user applications have complete read and write access to the data in all partitions of the table. Selects, updates, inserts, and deletes continue to execute against the original table.

3.3 Phase 3: Audit Fixup 2

In the third phase, Audit Fixup 2, locks are requested to make sure all outstanding user transactions against the table are completed and applied to the new partition before it is made logically part of the table's definition. First, the Split Partition operation requests a file lock against the original partition being split. The lock is made as part of a transaction; it does not allow any other locks to exist concurrently against the specified partition. Thus, once the lock is in effect no changes outside the transaction can be made to the partition until the transaction is completed.

When the lock is granted, the audit-fixup process is called again to finish applying relevant audit. It stops examining audit when it reaches the point where the file lock was granted. In the example, the process reads the NY record at this time.

After this audit is applied, a filelock is requested for all other partitions of the table. When the file lock is granted, the labels of each partition are updated to include the necessary information about the new partition. (Partition information is distributed to all partitions to aid availability in certain circumstances.) The redefinition timestamp of the table is update to indicate that a change has been made. Figure 5 shows the sample table after phase 3 is completed.

When an executing user application makes its next request to the table, the NonStop SQL/MP file system and executor components notice that the table has undergone a change since their last request. NonStop SQL/MP compiles and reuses access plans but when changes to the table's DDL are made, recompilation occurs if the user has allowed runtime compilation for this application. Since the table now logically has four partitions, the NonStop SQL/MP compiler produces appropriate query execution plans so that Data Definition Language (DDL) and Data Manipulation Language (DML) operations occur against the proper partitions. Recompilation in itself can be a source of outage. A separate NonStop SQL/MP feature, late binding, is available to handle this situation.

At the beginning of phase 3, users have complete read and write access to all unaffected partitions, but by its end, the table is inaccessible to all applications. The time it takes to complete phase 3 depends on the number of partitions of the table and the transaction activity on the table. In practice, this phase should usually last less than a minute.

3.4 Phase 4: Cleanup

In phase 4, Cleanup, the Split Partition operation cleans up the table, removing the physically present, but logically absent, rows from the (original) split partition. In the example, the rows that have been moved from the second partition to the third partition need to be deleted from the second partition. The cleanup operation occurs in the background and does not affect user applications. Phase 4 ends when all the unnecessary records have been deleted. During this phase, the user has complete read and write access to the table.

3.5 Database Recoverability

A major design goal for the NSA database configuration operations required that users who had media protection (used NonStop TM/MP file-recovery protection) would be able to recover their database tables if a media failure occurred at any point in the operation. To this end, the NSA operations allow users to make online dumps of the target partition before the end of phase 3. Because of this capability, and because audit is generated for the target partition in phase 2 and 3, one can recover from a media failure at each point in the operation. If a failure occurs before phase 3 ends, the new partition is not visible to the user application, and one can use the original table's online dump to recover the file. If a failure occurs after phase 3, one can use the online dump made in phase 2 and phase 3 to recreate the new partition.

4 Conclusion

The NSA physical database configuration features introduced in NonStop SQL/MP significantly reduce database outages. By allowing full read and write access to all partitions of a table during the lengthy data-movement phase, these features limit the outages associated with the Split Partition, Move Partition, Move Partition Boundary, and Create Index operations to about one minute per operation.

References

[1] Chandra, M. and Eicher, D. 1994 Enhancing Availability, Manageability, and Performance with NonStop TM/MP, Tandem Systems Review, Tandem Computers Incorporated. Part Number 104400

[2] Ho, F., Jain, R., and Troisi, J. 1994, An Overview of NonStop SQL/MP, Tandem Systems Review, Tandem Computers Incorporated. Part Number 104400

Sybase System 11 Online Capabilities

T.K.Rengarajan and Lucien Dimino and Dwayne Chung
Sybase Inc.
1650 65th Street
Emeryville, CA 94608
{ranga, dimino, dwayne}@sybase.com

Abstract

Online capabilities are essential to database management systems to improve their availability. We present a summary of the online capabilities of Sybase System 11 relational database management system. The capabilities are summarized under the broad categories: server features that eliminate the need for online utilities, online backup/restore, online DBCC utilities, online management options and the warm stand-by server that forms the basis for further online operations.

1 Introduction

Database servers process transactions. In addition, database servers require additional maintenance operations. These maintenance operations are required for media recovery (eg., database backup/restore), data reorganization for performance (eg., adding table partitions), utilities for checking the consistency of the database (eg., DBCC) and changes in available resources (using one more processor). Examples of such operations include database backup/restore, adding table partitions, DBCC and bringing more processors online respectively.

These maintenance operations cannot always be postponed until a time the database server, or a database, can be shut down and transaction processing stopped. Even when the operations can be postponed, there is often insufficient down time available to perform them all. Hence, there is a need to perform these maintenance operations online to improve the availability of databases.

We present a summary of the online capabilities of Sybase System 11 relational database management system. First, we present some of the interesting aspects of System 11 that makes special online utilities unnecessary. Here we have picked those features that are not available in mainstream database management systems. Second, we present the online backup and restore of the database that helps save the database from media failures. Third, we discuss a number of management features in System 11 that can be performed online, as well as the database consistency checker (DBCC). Finally, we mention the role of the warm standby as the basis of major reorganization requirements.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Self-tuning Server Capabilities

System 11 includes a number of features that eliminate the need for a class of online utilities. These are the first step towards a self-tuning server. This set of features reduce the need for special management utilities let alone having to make them online.

2.1 Clustered Indexes

Clustered index support has been a feature of Sybase SQL server from the first version. A clustered index keeps reorganizing the rows of a table so that they are sequential on disk in the best case. When a table is loaded into the database, it is loaded in the sorted order of the primary key. A suitable “fill factor” is chosen to leave room on data pages for future row insertions. If the data distribution is uniform, this fill factor maintains the sequential nature of the table on disk. However, once the extra room on pages is filled up, the physical order on disk is changed to the extent necessary. When the extra pages are filled, the rows are still in order and clustered within the page. The data pages are still kept in a sorted order logically, but not physically. An attempt is made to allocate new pages physically close to the logical order.

Clustered indexes keep the table stored on disk in sequential order and reduce the need for table reorganization facilities to a large extent. They do not involve much extra overhead compared to a non-clustered index.

2.2 Automatic large IO

System 11 supports multiple sizes of in-memory buffers and size of IO to disk. However, the pages still remain at the same size, i.e., 2K bytes. This allows the system to perform different IO sizes to the same pages on disk at different times. A system administrator (SA) does not have to decide a-priori that a certain table requires large IO blocks. For sequential scans, the system attempts to perform large IOs. For random row accesses, the system picks small (2K bytes) IOs. In addition, the system picks a “fetch-and-discard” strategy for buffer management for sequential scans in some cases. The IO sizes are handled as a “hint” within the server. If the requested page is available in memory, the IO size is ignored. Thus, the System 11 buffer manager uses appropriate sizes of IOs for various applications based on the available buffer sizes and the query.

In systems that require the SA to decide on IO sizes, there is a need for the SA to change her mind and change IO sizes online. The automatic choice and handling of IO sizes eliminates this entire class of online reorg utilities.

3 Online backup

3.1 Database backup online

System 11 provides online database backup capability. The backup function for System 11 is performed by a separate server called the Backup Server. Its architecture utilizes concurrent parallel processing. The SA specifies the number of tape drives (or disk drives for that matter). The database is divided evenly between the threads. The SQL server provides the backup server with a list of database pages to be backed up. The backup server then backs up the pages directly from disk to tape, while transactions are being processed on the database by SQL server. Benchmarks have demonstrated backup rates of 46.5 GB/hour to 12 8mm tape drives in parallel. At this point, the backup server is limited only by the hardware in scaleup and speedup.

SQL server uses physical logging on database pages. It is possible to examine a database page and the log record to determine if the log record needs to be applied. This makes it possible to backup pages to tape without extensive coordination with SQL server. The synchronization between the backup server and SQL server is minimized. The online backup has only 2-4impact on the transaction rates of the server, while running the debit-credit benchmark.

3.2 Log backup online

The database log is considered to be another table in Sybase SQL server. The log records are formated the same way on database pages. This allows users and the system itself to execute queries on the log in a convenient way. Note, the processing of log records during runtime is handled in a special way, optimized for group commits. It is only the data format on pages that is different.

The backup server is able to backup the log pages online as well. This is done either in the context of a database backup or a separate log backup. The checkpoint record maintains the "active portion" of the log, i.e., the first log record of the oldest active transaction at the time of checkpoint. Only the non-active portion of the log is backed up.

System 11 allows the SA to specify free space thresholds. When free space reaches a user-defined level, a user-defined stored procedure is executed. This feature enables the automatic backup of the log when the log becomes full, leverages the ability to backup the log online and automates a critical SA function.

4 Online Resource Management

The job of a database management system can be simplistically summaried as follows. The server is given certain amounts of system resources, like processors, memory, disk space and system, IO and network bandwidths. The server is required to execute a workload of transactions using these resources and meeting response time and throughput requirements. This problem statement requires the system to make efficient use of the resources.

System 11 strives to provide the SA with the flexibility to precisely control the allocation of processor, memory and disk, while at the same time automatically optimize the use of these resources online. More processors can be provided to System 11 online. The memory use can be changed online. New disks can be added to the system online.

The next few sections describe some of the online capabilities for these resources.

4.1 Online processor options

One continued focus for SQL server development is to make most of the SA controlled knobs dynamically tunable. This property is a requirement and foundation for self-tuning database servers. Here are some interesting configuration options that can be tuned dynamically.

4.1.1 Online engines

System 11 partitions the use of processors in an SMP system via the concept of engines. A SQL server consists of a number of operating system processes. Each such OS process is hard-affinitied to a specific processor. The SA has precise control in an SMP system to "allocate" a certain number of processors to SQL server. The client user connections are handled via threads that are managed by SQL server. These threads are scheduled on specific processors via internal algorithms that balance the need for affinity and load balancing. The engines server as a proxy and container for processors and hardware caches. They also serve as the foundation for hardware cache sensitive algorithms for managing memory in SQL server. System 11 allows the SA to add new engines online during processing. The workload of the server is symmetric: the network IOs, the disk IOs, and computation are balanced evenly between engines. When a new engine comes online, threads are automatically redistributed between all the engines smoothly.

4.1.2 Housekeeper

System 11 adds a feature called Housekeeper (HK), that serves as the basis for lazy processing in SQL server. The HK is a special thread created by System 11 that uses idle cycles to perform useful work. The essential idea is to postpone as much work as possible to the future in all algorithms in SQL server. Once this is done, the HK can exploit idle cycles to perform this work. The benefit is load balancing across a time interval. Contrast this to load balancing at any time between processors discussed previously.

In System 11, the HK writes dirty buffers to disk in order to minimize the amount of work to be done during a checkpoint and to reduce recovery time. In ideal conditions, the HK also performs a free checkpoint automatically. All this is performed automatically by the system, depending on the system utilization. The HK is non-intrusive. The HK is not required for operation of the system. If a system does not have any idle time, the HK will never be required to execute. The postponed work will be handled by the usual threads that execute user queries.

The HK can be started and stopped online at any moment in SQL server. Its aggressiveness can be increased or decreased to any extent online.

4.2 Online Memory options

System 11 lays the foundation for partitioning in SQL server as the basis of parallel execution. System 11 offers the SAs the ability to partition tables, memory and cpu and exercise precise control over this partitioning.

The Logical Memory Manager is the component that enables memory partitioning. The SA is able to partition the memory available to the SQL server into many named caches. Databases or tables can be bound to any specific cache. These caches can further be split into buffer pools. Each buffer pool has a configurable IO size associated with it. The IO size is chosen by the system automatically based on the query, the cache binding for the tables and the availability of buffer pools. The binding of databases or tables to named caches can be done online at any time. The system moves all such buffers already in memory to the appropriate named cache at run time without having to stop transactions in flight. The partitioning of named caches into buffer pools can also be changed at run time.

The combination of the ability to partition memory, change it online and dynamically pick IO sizes and buffering algorithms provides great flexibility and a solid foundation for future work on self-tuning parallel system.

4.3 Online disk operations

4.3.1 Add new devices online

Disks can be added to computer systems online. SQL server abstracts the available disk space into Sybase devices. One Sybase device corresponds to a raw partition of an OS disk. One OS disk may consist of a stripe set, a RAID disk array. The striping and RAID mgmt can be done in software on the host or in hardware in the controller.

Sybase devices can be added to SQL server online, while transactions are being processed. They can then be provided to any database and to one or more tables in that database. Such disk space will be consumed for the transaction processing immediately after the addition. This capability avoids having to shut down the server for adding disk space.

One example of this ability to add more disk space online is for the log. It is not uncommon to run into situations when the configured database log becomes full. In these situations, transaction processing cannot continue. The log threshold manager in combination with the ability to backup the log online and the ability to add more disk space online addresses this situation.

4.3.2 Mirror/unmirror Sybase devices online

Sybase SQL server provides a facility for mirroring Sybase devices. It is possible to mirror any two Sybase devices, irrespective of the underlying disk subsystem, disk type, disk manufacturer. This capability allows customers a smooth upgrade path from non-mirrored databases to mirrored databases. They can simply buy a new disk (most likely of a different type) and mirror their existing disks.

The mirroring and unmirroring of Sybase devices can be performed online, while transactions are in flight. There is a price to be paid at run time for this ability. Every disk IO issued in SQL server checks if a disk is mirrored. If so, the writes are executed in parallel and reads are routed to one of the mirrors.

4.3.3 Add new table partitions online

System 11 enables the SA to partition one database table into a number of logical partitions. These logical partitions may also be physically separated. This partitioning is a foundation for extensive parallelism in the server. System 11 allows the creation of partitions of tables online. It also allows the merge of partitions online. These operations are fast, since the data pages that comprise the partitioning are not moved.

5 Database Consistency Checker (DBCC)

Providing data integrity is a prime function of an RDBMS. In order to provide integrity of data, it is necessary to ensure integrity of the on-disk data structures. Unfortunately, no hardware, OS or RDBMS is immune to faults. These faults should be discovered and isolated before they lead to catastrophic failure. This is the purpose of the DBCC utility.

DBCC locates database corruptions like corrupt database pages, inconsistent storage allocation, inconsistent database catalogs and failure to maintain sorted order in indexes.DBCC also has a secondary function of collecting and displaying information regarding the use of disk space and the distribution of data on disks. The DBCC utility can be executed online, while transactions are being processed actively by the SQL server.

6 Warm standby with replication server

There are a number of other facilities outside the System 11 SQL server that provide some capabilities for online operations. A number of tools vendors provide capabilities to perform "online reorganization" for Sybase databases.

The Sybase Replication Server enables an SA to create a separate database at a different site and propagate updates to this warm standby. The warm standby can have a different physical database design. The warm standby can have different set of indexes. It is also possible to buffer updates to the original database and apply such updates to the warm standby later on.

The warm standby presents a different approach to performing big database design changes online. The warm standby can be considered as a copy that can be reorganized online, while not stopping any update activity at the primary. The transactions that execute at the primary during the reorganization can be applied to the warm standby after the reorganization. If reorganization is desired at the primary, the roles of the two copies needs to be reversed.

7 Summary

Sybase System 11 SQL server provides several useful online capabilities. Clustered indexes and automatic selection of large IO prevent the need for online utilities to perform data reorganization. The database as well as the log can be backed up online to tape. The processor, memory and disk resources can be partitioned for scaling. These resources can also be added or redistributed online. The database consistency checker can be executed online while transactions are in progress to verify the integrity of the internal database structures. In addition there are other solutions provided by the Sybase Replication Server and other third party tools vendors to further enhance the ability to perform online operations.

Acknowledgements

Building a commercial production-quality database management system is a big team effort. The capabilities we have described here have been developed over the last decade, by numerous talented engineers. While we've had our part in building System 11, our primary contribution here is to present their work.

References

[1] Sybase System 11 SQL Server documentation set.

[2] "SYBASE Replication Server: A Practical Architecture for Distributing and Sharing Corporate Information", Alex Moissis, http://www.sybase.com/Offerings/Whitepapers/repserver_wpaper.html.

Concurrent File Reorganization: Clustering, Conversion and Maintenance

Edward Omiecinski
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

1 Introduction

In this paper, we highlight some of the work that we have done, during the past ten years, in the area of online reorganization. Our original motivation for pursuing online reorganization (or reorganization in general) was based on our work in the area of record clustering. We found that although several algorithms had been proposed on how to effectively cluster records of a database, algorithms were lacking on how to reorganize the database to produce the desired clusterings. Certainly, if the clustering was based on sorting, then an external sort algorithm could be used but we were considering more general criteria for clustering. In addition, we realized that reorganization that was normally done off-line could be done efficiently on-line and today where systems are available 7 days a week and 24 hours a day, on-line reorganization becomes a neccessity.

Besides developing on-line reorganization algorithms for the clustering problem, we started to examine other physical database reorganization problems, such as eliminating overflow records in indexed-sequential files (a predecessor of the B$^+$-tree file structure). Also in the mid-eighties, several dynamic hash-based file structures appeared such as Linear Hashing and Extendible Hashing. Since the appropriate file structure depends on the type of queries to be processed and the query types may evolve over time, both the B$^+$-tree and hash-based file structures may be appropriate at different times of a file's existence. Consequently, we moved into the area of on-line file conversion in the late eighties.

Also, in the late eighties and early nineties, research into parallel database systems became popular. To capitalize on the parallel I/O capability to improve database performace (e.g., with the shared-nothing parallel database system architecture) data and indexes were partitioned by some algorithm (e.g., using round-robin or range partitioning) across multiple disk drives (or nodes). Partitioning was motivated by the need to provide "good" performance but once again the performance may change depending on user access patterns. Trying to maintain that level of performance may entail a repartitioning of the data and/or indexes across the nodes. In addition, maintaining indexes across multiple nodes also carries extra overhead. For example, when the indexed attribute(s) of tuples are updated, those updated tuples may migrate to another node depending on the partitioning method. Besides moving the tuples, the associated indexes will also have to be modified (e.g., deleting entries in the index on the original or source node and inserting entries in the index on the destination node). These are representative of the problems we have pursued and which we will discuss in this paper.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

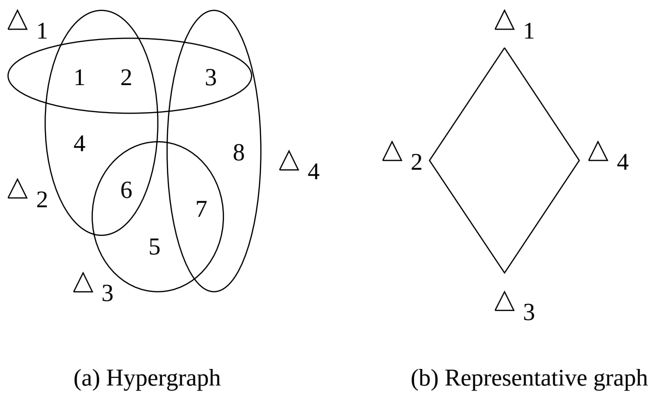


Figure 1: Hypergraph and its representative graph

2 Clustering

In the work presented in [3, 9] we consider the problem of changing the placement of records assigned to pages of a file on secondary storage. One example of this would be to place records which are frequently accessed together on the same page or pages in order to reduce retrieval time and another example would be to move records from overflow pages to primary pages of a file. The approach we present in [3] is incremental and on-line. The main idea is to lock a small part of the file at any one time while permitting access to the remainder of the file. Our goal was to design an algorithm that would minimize the number of page accesses made to/from secondary storage and in particular the number of pages swapped in and out of the buffer during the reorganization process.

The basic algorithm is to read in a set of currently existing pages (old pages) and to extract the records from those pages that are needed to construct a reorganized page (new page). We present a formal definition (or model) of the problem in terms of a hypergraph and its representative graph. The vertices in the hypergraph correspond to the current pages and the edges in the hypergraph correspond to the set of current pages needed to be accessed to make a new page. The representative graph took the edges of the hypergraph and modelled them as vertices where edges existed between those vertices if the original hypergraph edges had vertices in common. For example in Figure 1(a), we have a hypergraph with 4 edges, $\Delta_1, \ldots, \Delta_4$. In the representative graph, we see an edge between Δ_1 and Δ_2 since they have two vertices in common, i.e., *Page 1* and *Page 2*. In the representative graph, we also assign a cost associated with each edge. This reflects the number of page acceses needed in constructing the new page corresponding to one endpoint of the edge having already accessed the pages needed for the other enpoint of the edge. For example, if we started out by constructing new page Δ_1, we would have to access *Page 1*, *Page 2* and *Page 3*. So the cost associated with edge (Δ_1, Δ_2) would be 2 since only two more pages would be needed in order to construct new page Δ_2.

In [3], we show that the problem is equivalent to the traveling salesman problem which is NP-complete. In the paper, we also develop two classes of heuristics, *Static* and *Dynamic*, which try to minimize the cost of our reorganization. With the *Static* method. we find a complete tour of the representative graph before any reorganization is performed. With the *Dynamic* method, we alternate between finding the next edge in the tour and doing the reorganization. We also incorporate an intelligent buffer paging policy that chooses the page for replacement as the one that holds the fewest records needed by new pages. We compare the performance of six variations of our two classes of heuristics through a number of experiments.

In [10], we realized that there were some limitations to our reorganization approach in [3, 9]. In particular, a minimum buffer capacity was needed with a size (in number of pages) equal to the number of records that fit on a page and there was some inefficiency associated with when pages were written back to disk as well as some inefficiency associated with forming a cluster of records smaller than the page size. We corrected these inefficiencies in [10] with an improved and more general clustering reorganization algorithm. We also proved some important properties associated with our algorithm and did a performance study of our new algorithm with the original algorithm [3] showing that our new algorithm provides a substantial reduction in the number of page accesses.

In the work that we discussed so far [3, 9, 10], we only focused on minimizing the number of page accesses required to perform the reorganization. To actually show that our algorithms could be efficiently used in an on-line manner, we needed to perform additional experiments. These experiments were the motivation for our work in [7, 8].

In [7, 8], we present a simulation-based performance analysis of our concurrent reorganization algorithm from [10]. We examined the effect on throughput of

(a) buffer size,

(b) degree of reorganization,

(c) write probability of transactions,

(d) multiprogramming level and

(e) degree of clustered transactions.

In [8], we show how the performance of the cluster reorganization process can be further enhanced using a differential file approach for updating the index for the file. We also show that the reorganization process can be run concurrently with user transactions without violating serializability. We developed an analytical model to measure throughput in the steady state and validated the results with our simulation.

The simulation model is pictured in Figure 2. The simulation model uses a given multiprogramming level and a dynamic locking scheme where the granularity of locking is at the page level. The total amount of buffer space is fixed and is shared between user transactions and the reorganization process. The model simulates transactions made against the database which include *cluster* requests or *random* requests. Each *random* request accesses a variable number of pages. Some of those requests are read-only while others are read-write. A *cluster* query retrieves records which comprise one of the many clusters of records.

¿From our work in [8], we found that in general, increasing the buffer size for reorganization results in a decrease in system throughput during the reorganization process. This can be explained by the fact that a smaller reorganization buffer will cause less transaction blocking since fewer locks will be held by the reorganization process. However, by increasing the reorganization buffer size, the reorganization process was able to reorganize the database sooner, allowing transactions to make use of the clustering and for the system to realize the improved throughput. We also observed that increasing the multiprogramming level or increasing the percent of clustered transactions had an inverse effect compared to increasing the allocated buffer space for reorganization. By increasing the multiprogramming level the ratio of the number of locks held by transactions to the number of locks held by the reorganization process increases, thus creating a similar effect to reducing the reorganization buffer size. Increasing the percent of clustered transactions causes increased data contention. Understanding the tradeoffs involved with concurrent reorganization will aid the database administrator in tuning the performance of the system during the reorganization period.

3 Conversion

In our work presented in [5, 6], 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. In other words, we can call it *file conversion*. This conversion is motivated by a change in user access patterns. For example, an indexed file structure is chosen originally since it can efficiently handle range queries. If after

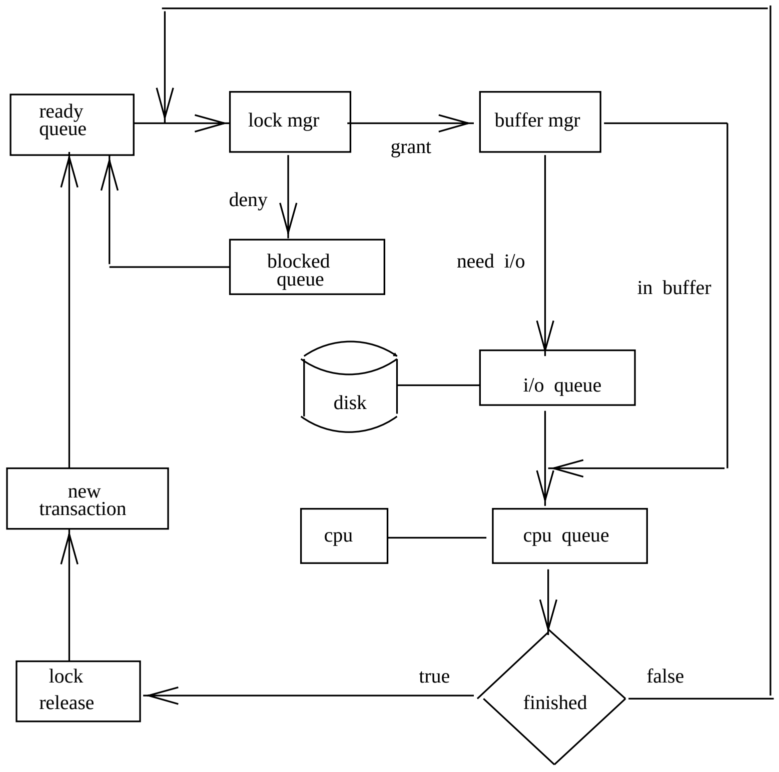
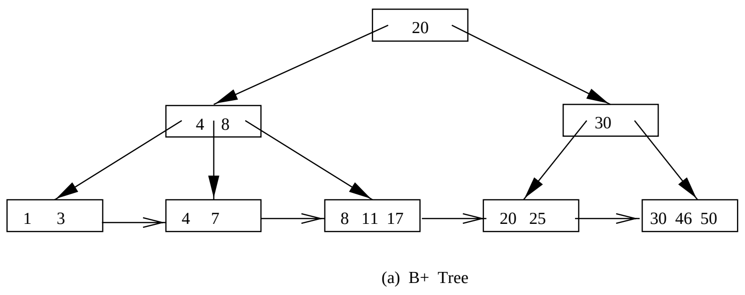


Figure 2: Database simulation model

time, the predominate type of query becomes exact match, then performance can be improved by having a file structure that more efficiently handles exact match queries (e.g., a hash-based file structure).

We propose concurrent reorganization schemes which allow an on-line conversion of a B^+-tree to a linear hash file and vice versa. The conversion in either direction works quite 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. Due to the reorganization procedure, which proceeds in key sequence, it is clear which file has to be accessed for a given query. Figure 3 shows a sample B^+-tree and the different steps associated with its conversion to a linear hash file. The basic approach is to proceed along the leaf level of the B^+-tree accessing the associated data pages. We extract the records from those data pages and insert them into the linear hash file. If the index is a *primary index* then the keys in the data pages appear in the same order as the keys in the leaf pages. When the file and index are created this ordering holds but (in some database systems) due to insertions and the allocation of free space within the file, this ordering may be perturbed somewhat. If the ordering of keys in the data pages is not the same as in the leaf pages then a number of data pages will have to be read in for each leaf page conversion. Each step in Figure 3(b) shows the linear hash file after each successive leaf page (i.e., the associated data page(s) have been converted).

In this work [6], we present a simple analytical model for each of the two conversion processes. In particular with our models, we wish to determine the *breakpoint* associated with the reorganization. The *breakpoint* is defined as the number of pages that have to be converted (or equally the time) before the throughput of the system with concurrent reorganization equals the throughput of the system with only transaction processing when the original file structure is used. We know that after the *breakpoint* the performance of the system with concurrent reorganization becomes better. Besides our analytical models, we once again include a simulation study. The simulation model is similar to that presented in Figure 2. In the simulation study, we examine the throughput of processing transactions using the original file structure (e.g., the B^+-tree) and examine the throughput of pro-



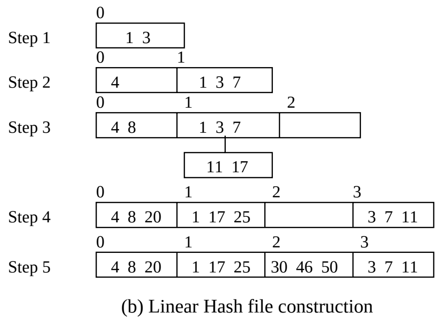


Figure 3: File conversion from B$^+$-tree to Linear Hash file

cessing the same set of transactions along with the reorganization process (e.g., converting the B$^+$-tree to a linear hash file). We determine when we have the largest decrease in throughput under the on-line reorganization scenario as compared with transaction processing using the original file. Likewise, we determine when we have the largest increase in throughput. We also look at the number of transactions that would not be processed during the reorganization time if reorganization was done off-line. The maximum decrease/increase depends on a number of factors, including the following:

(a) the unit of reorganization,

(b) the percentage of range queries and the lower and upper limit of the range.

The unit of reorganization represented the part of the original file that had to be reorganized before it could be accessed through the new file structure. A reorganization unit larger than 1 might be necessary when the records are not physically in key value order. For the B$^+$-tree to linear hash file conversion, we must proceed in key value order to be able to direct searches to the correct file structure. As one would expect, increasing the reorganization unit delayed the *breakpoint* and increased the maximum decrease in transaction throughput. However, even when the reorganization unit was relatively high, the on-line reorganization performance was still acceptable.

The percentage of range queries was varied as well as the range size. We should also mention that due to the modulo hash function used, there was some locality of keys within the linear hash file since consecutive key values were placed in adjacent buckets of the linear hash file. So for processing a range query, a number of adjacent buckets in the linear hash file had to be accessed and that number was equal to size of the range. If the range size exceeds the number of buckets in the linear hash file, then the entire file must be searched. In addition, there may be only a few occurrences of key values (in the file) that fall within a large range specified in a query. Certainly for that situation, the linear hash file will perform poorly whereas the B$^+$-tree will perform quite well. The main observation about the linear hash file to B$^+$-tree conversion was that by increasing the range query probability or by increasing the range size, the *breakpoint* occurred sooner.

4 Maintenance

Under the area of file maintenance, we have examined a number of problems ranging from on-line compaction of indexed sequential files [4] to the on-line (re)placement of declustered fragments across mutiple disks [1] to on-line index modification in shared-nothing parallel databases [2].

In [4], we present an approach for removing overflow pages and for compacting the nodes (pages) of an indexed sequential file. The overall order of reorganizing the nodes in the tree is equivalent to a postorder traversal of the tree. As a set of leaf nodes for a given index node are reorganized, the lowest key and highest key and the address of the leaf nodes are saved. This information is returned to the leaf node's parent and is used to reorganize that node. As index nodes are reorganized the same type of information is saved and is used to reorganize the parent of the parent of a leaf node. This process is repeated, reorganizing nodes in postorder until the root is finally modified. To improve throughput we try to minimize the number of locks held by the reorganization process. Although, no performance study was performed with regard to this algorithm.

So far, all of the reorganization work presented has been in the context of a single processor computing environment, with the file reorganization involving a single secondary storage device as well. In [1], we start to look at reorganization problems within a multiprocessor environment, to be more precise, a parallel database system. In this work [1], we consider the shared-nothing parallel database architecture where each node (processor) has its own local memory and its own disk. Communication between nodes is done by message passing.

With a shared-nothing architecture, there are several alternatives for the placement of data across the nodes, also known as *declustering*. They include *round-robin*, *hash* and *range* declustering strategies to name a few. In addition the number of disks which should store a given file (called the *degree of allocation*) may differ for each relation and can be as small as 1 disk or as great at the total number of disks. This number depends on the frequency of access as well as the size of the relation. Since the frequency of access can change as well as the size of the relation, the optimal degree of allocation may likewise change, thus prompting a reorganization. The main point is that an effective data placement is one that distributes the I/O load across all of the disks. In [1], we examine the influence that the initial declustering strategy has on the (eventual) reorganization. In particular how does the reorganization cost depend on the declustering strategy? In this work we focused on one particular type of reorganization. That is, the reorganization needed to change the degree of allocation of a file. In particular, we consider the amount of data that has to be reorganized (assuming different intitial declustering schemes) since that amount will impact the duration of the reorganization which also influences the system throughput.

We studied six data placement strategies and their associated reorganization costs under a transaction processing workload (TPC-C). As it turned out, all but one of the data placement strategies performed equally well in balancing the load across the disks. However, the reorganization costs associated with those strategies differed significantly. The point to be made is that if different data placement strategies are effective in balancing the I/O load then we need to consider their associated reorganization costs to decide on the most cost effective placement method.

The last and most recent work [2] also deals with shared-nothing parallel databases. However in [2], we examine the problem of on-line index maintenance. Whenever data is moved across nodes in a shared-nothing database system, the indexes must be modified too. We assume that local indexes exist for each fragment of a relation stored on the different nodes. The cost of index modification can be quite high, especially when several indexes exist for each relation, We study two alternatives for performing the necessary index modifications, called one-at-a-time *OAT* page movement and *BULK* page movement. These two strategies differ in two main ways:

1. the *OAT* method uses very little extra disk space whereas the *BULK* method can use a large amount,
2. the *BULK* method uses sequential prefetch I/O to optimize the number of I/Os performed during the index modification while the *OAT* method does not.

An experimental testbed was used to compare the performance of the two methods which showed that the *BULK* method was an order of magnitude faster than the *OAT* method. As far as how these reorganization methods affected transaction performance, we found that for tables with one or two indexes, that the *OAT* method degraded throughput less than the *BULK* method. However, when the number of indexes was greater than two, both techniques had a similar effect on transaction performance.

5 Conclusion

In this paper, we have tried to highlight some of the work that we have done, during the past ten years, in the area of online reorganization. The objective was to provide a broad overview of the problems we have pursued. In particular, we wanted to present some insight into how we approached those problems and some general information about our results. The interested reader is encouraged to examine some of the papers listed in the References section for more details.

References

[1] K. Achyutuni, E. Omiecinski and S. Navathe, Maintaining an Effective Data Placement in Parallel Databases, submitted for publication, August 1995.

[2] K. Achyutuni, E. Omiecinski and S. Navathe, Two Techniques for On-Line Index Modification in Shared Nothing Parallel Databases, 1996 ACM SIGMOD International Conference on Management of Data, June 1996.

[3] E. Omiecinski, Incremental File Reorganization Schemes, 11th Very Large Database Conference, August 1985, pp. 346-357.

[4] E. Omiecinski, Concurrency During the Reorganization of Indexed Files, 9th IEEE Computer Software & Applications Conference, October 1985, pp. 482-488.

[5] E. Omiecinski, Concurrent Storage Structure Conversion: from B+ Tree to Linear Hash File, 4th International Conference on Data Engineering, February 1988, pp. 589-596.

[6] E. Omiecinski, Concurrent File Conversion between B+ Tree & Linear Hash Files, Information Systems, 14, 5, 1989, pp. 371-383.

[7] E. Omiecinski, L. Lee and P. Scheuermann, Concurrent File Reorganization for Record Clustering: A Performance Study, 8th International IEEE Data Engineering Conference, February 1992, 265-272.

[8] E. Omiecinski, L. Lee and P. Scheuermann, Performance Analysis of a Concurrent File Reorganization Algorithm for Record Clustering, IEEE Transactions on Knowledge & Data Engineering, 6, 2, April 1994, 248-257.

[9] 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, Cambridge, England, July 1984, pp. 201-219.

[10] P. Scheuermann, Y. Park and E. Omiecinski, A Heuristic File Reorganization Algorithm Based on Record Clustering, BIT, 29, 1989, pp. 428-447.

Towards Efficient Online Database Reorganization

Chendong Zou and Betty Salzberg [*]
College of Computer Science
Northeastern University
Boston MA 02115

1 Introduction

In recent years, as more businesses start using massive databases as their main source of information, more emphasis is placed on the performance of the database system. These require not only that the database system have good performance in terms of time and space, but also that it be continually available. This means that the database can not be shut down at any time. All the operations should be done on-line, including database reorganization. In this paper, we will outline our research in developing more efficient on-line reorganization utilities. We will discuss two particular problems.

The first problem is the on-line reorganization of *Sparsely-Populated* primary B^+-trees. Primary B^+-trees have the data records in their leaves. They are used for clustering indexes in Tandem and Sybase systems, for example. After a B^+-tree has been used for a while, its performance will often degrade. The degradation could be caused by both insertions and deletions. Large numbers of insertions often cause page splits in the B^+-tree, with the result that the leaf pages within a key range in the B^+-tree are not in contiguous disk space. This will cause more disk read time to be required for a range query. Large numbers of deletions will cause the pages, mostly leaf pages, in the B^+-tree to be sparse. This also makes reads inefficient, because for the same amount of data, it will take more page reads for a sparsely populated B^+-tree than for an unsparse one.

The second problem is the change of references for the secondary or "non-clustering" indexes during on-line reorganization. For example, in an RID organization, (such as IBM's DB2), records are accessed by their Record ID, which is usually a page number and a slot number in the page. If the records are moved to another page for any reason (for example, to place the records in a primary B^+-tree or to place in one page records which had been fragmented over two or more pages), all the secondary indexes referring to the moved records must be updated. Our work has focused on doing the updates to the secondary indexes lazily, piggybacking them on user transactions which request the leaf pages of the indexes that need to be updated. Our method could be used with any record-moving reorganization.

In this paper, we will outline our solutions to these two problems and discuss their implications in developing efficient on-line reorganization algorithms. The solution to the B^+-tree reorganization problem is discussed in section 2. Section 3 addresses the problem of efficient reference updating during on-line reorganization. Section 4 summarizes the results and generalizes the common grounds between the algorithms presented in this paper.

[*]This work was partially supported by NSF grant IRI-91-02821.

2 On-line Reorganization of Sparsely-Populated B^+-trees

2.1 Assumptions

In this section, we discuss a proposed method for reorganizing a primary B^+-tree. (i.e., where leaf pages contain the data records). We assume that the B^+-tree is *sparsely populated*, that is, a large portion of many leaf pages is unused. There are also some free pages available on the disk, which are not connected to the B^+-tree.

We assume that there is no ongoing node consolidation in the B^+-tree and the free-at-empty [2] policy is adopted for leaf pages. That is, when a leaf becomes completely empty, its page is deallocated and its parent is updated to reflect this. Most commercial systems adopt this policy [2].

Since availability of the database system is essential, we lock as little of the database as possible. In addition, we log some information so that reorganization work already done is not lost in case of system failure. Since log size is a concern, we try to make the amount of information logged small.

2.2 The Proposed Solution

There are basically two ways to perform on-line reorganization. One is to concurrently create a new structure somewhere else in the disk, and switch to the new structure when it is built. We call this *new-space reorganization*. The other method is to do the reorganization in-place, that is, to try to move data around disk pages without using new disk space. We call this *in-place reorganization*.

Our approach is a combination of both methods. We reorganize the leaf pages of the B^+-tree using both the in-place method and the new-place method. Since the B^+-tree is used as the primary index of the database, and the size of the database could be very large (e.g., several terabytes), it is unlikely that there is enough space in the disk to create the new B^+-tree. So an in-place method has to be used in order to reorganize the leaf pages. However, as we expect the tree to be sparsely populated and as we expect to have some free space not used by the tree, we are able to mix the in-place operations with some new-space operations.

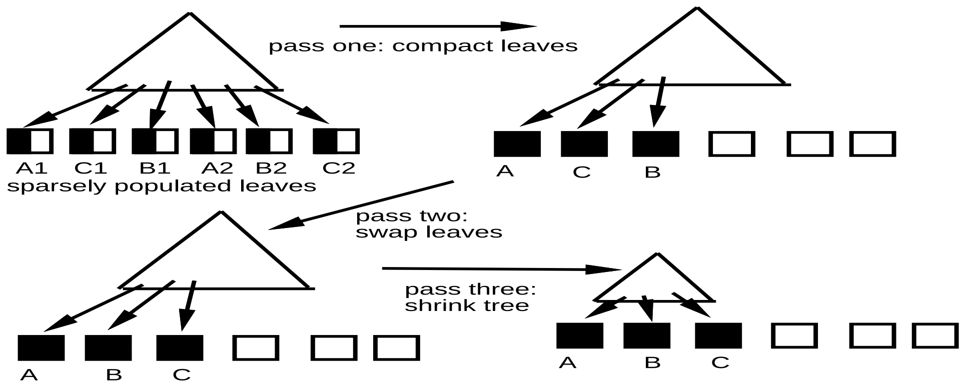
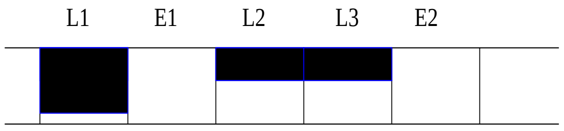
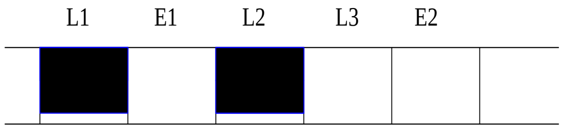


Figure 1: Three-pass Algorithm

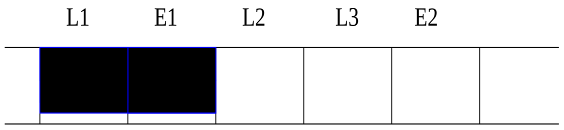
Our complete algorithm is a three-stage algorithm. We first compact the leaf pages using both in-place and new-place operations. In the second stage, we swap leaf pages and move to empty pages to make the disk space used by the leaves contiguous and in key order. During the first and second stages, ongoing database activity may split leaves in areas that have already been reorganized. We do not try to clean this up, so the result at the end of the first two stages is not necessarily a *perfectly* ordered compacted set of leaves. In the third stage, we reorganize the internal pages using a new-place algorithm to make the internal pages more compact so that it might reduce the height of the tree. Our three-stage algorithm is illustrated in Figure 1.



(a) The initial layout of the disk. L1, L2 and L3 are B+tree leaf pages, E1 and E2 are empty pages.



(b) The disk layout after compacting L2 and L3 to L2 using the in-place method.



(c) The disk layout after compacting L2 and L3 to E1 using the new-place method.

Figure 2: An example of disk page layout

2.3 Leaf Page Compacting

When there is empty space available and using the empty space is likely to improve system performance, we use this space to reorganize part of leaf pages with a new-place method. Figure 2 shows an example of disk page layout where using the new-place method will improve system performance better than the in-place method. In Figure 2, $L1, L2, L3$ are B$^+$-tree leaf pages, $E1, E2$ are empty pages on the disk. Suppose we compact leaf pages $L2$ and $L3$ to one page. We could either use in-place compacting or new-place compacting. Obviously, the result of using a new-place operation to compact $L2$, $L3$ to the empty page $E1$ is better than that of the in-place operation which will compact $L3$ to $L2$, because $E1$ is closer to page $L1$ then $L2$ is.

Choosing a "good" empty page can also means less swaps in the second pass. Moving to an empty page allows more concurrency than swapping since swapping usually involves two distinct base pages. In addition, swapping cannot take advantage of careful writing [3], since the entire contents of at least one of the pages being swapped must be written to the log. Since log size is a significant factor in reorganization methods, this is important.

In our algorithm, we choose the first empty page which is in front of the leaf page that is going to be reorganized, C, and after the largest finished leaf page ID, L. This implies that the new page constructed will be in the correct relative order with all the leaf pages that have already been compacted and that C will be moved closer to L. Experiments showed that our algorithm can greatly reduce the number of swaps needed at the second pass [7].

2.4 Leaf Swapping

After the leaves are compacted, they still may not be in key order on the disk, and there may be empty pages scattered among the new compacted leaf pages. At this stage, we may choose to do some swapping of leaf pages and moving of leaf pages to empty pages to put the pages in order. Swapping two leaf pages is an in-place operation and moving to an empty page is a new-place operation. We use a separate pass for swapping and moving after the compacting pass because (1) sometimes this pass could be omitted if the performance of the B$^+$-tree is not so bad; and (2) this two-pass method can increase concurrency and reduce the amount of information written to the log by the reorganization process [9] .

Granted Mode	Requested Mode							
	IS	IX	S	U	X	R	RX	RS
IS	Yes	Yes	Yes					Yes
IX	Yes	Yes						Yes
S	Yes		Yes			Yes		Yes
U			Yes					Yes
X								
R			Yes					
RX								

Table 1: The Lock Table

2.5 A New Concurrency Method for Compacting and Swapping the B$^+$-tree Leaf Pages

We propose a new locking protocol for compacting and swapping leaf pages. Let us call B$^+$-tree internal pages that are right above the leaf pages in the B$^+$-tree *base* pages. The base pages are at the parent-of-leaf-level in the B$^+$-tree. Our new protocol only holds an X lock on base pages for a short period of time, after the records in the leaf pages have been reorganized.

We introduce three new lock modes: R, RS and RX. Table 1 shows the compatibility of the locks. We assume readers and updaters may request or hold intention locks (IX or IS [1]) *on leaf pages only* if they are doing record-level locking. (RS is not listed in granted mode in Table 1 because as an "instant duration lock", it is never actually granted.)

The R mode is used by the reorganizer to lock the base pages which contain pointers to those leaf pages that are in one reorganization unit. The R lock is used before the reorganizer modifies the keys in the base page. The reorganizer will hold RX locks on the leaf pages in a reorganization unit, where the pointers associated with those keys point to the leaf pages that are being reorganized. The RX mode is not compatible with any lock mode. RX is not the same as X, because the action of the lock manager when a conflicting request arrives is different. With a X lock, a conflicting request causes the requester to wait. With an RX lock, a conflicting request causes the requester to forgo the conflicting request, release its lock on the base page and request an RS lock on the base page.

The solution we propose here is that the reader request an *unconditional instant duration* [4] RS lock instead of a normal lock request after it is blocked by the reorganizer. An *unconditional instant duration* lock means that the lock is not to be actually granted, but the lock manager has to delay returning the lock call with the success status until the lock becomes grantable. Since the RS lock is not compatible with the R lock, this will achieve the goal of blocking the reader from reading the reorganizing data. After the success status is returned for the RS lock request, (but the reader doesn't actually hold the lock), the reader will request a S lock on the base page and proceed. This is a good solution because the cost of an *instant duration* lock is relatively small, the lock is not actually granted, and the protocol can block the reader from reading any reorganizing data.

Deadlocks can happen with this protocol since the reorganizer will lock multiple nodes in the B$^+$-tree. When there is a deadlock, we always opt to abort the reorganizer. For detailed discussion on the deadlock issue using this protocol, please refer to [7]. The reorganizer's protocol, reader's protocol and updater's protocol can be found in [9].

2.6 Reorganizing the Internal Pages

Since the space needed for the upper levels of the tree is very small, we assume we can reorganize the internal pages using a new-place method. Since the keys in the *base* pages are already sorted, we start by reading the *base* pages from left to right, that is, we read the keys in ascending order. We start building a new B$^+$-tree in a bottom-up fashion as described in chapter 5 section 5 of [5].

The entries read in the base pages are copied to newly allocated empty pages. When a new page is added, no splitting is necessary. The first page is filled to a pre-assigned fill factor, and then the next records go in the next page. Each new page requires a new entry in the level above.

The reorganizer only holds an S lock on the *base* page that it is reading, so other readers could also access that page. In order to deal with possible updaters (this happens when there is a page split at leaf level or when deallocating an empty leaf page), we keep track of where we are in terms of the position in the *base* pages. If the update is on a *base* page that has already been read by the reorganizer, we make the update in the old tree, then we record that update in an append-only *side-file*.

If the update is on a *base* page that has not been read yet, then we do nothing, since that update will be read and we won't miss it. When the reorganizer finishes building the new B$^+$-tree, we will apply the updates in the *side-file* to the new B$^+$-tree to catch up. While the reorganizer is doing catch-up, some more updates may be appended to the *side-file*. Since leaf page splits don't happen very often, we will eventually catch up all the changes. After that, we will switch over from the old tree to the new tree. Details of concurrency control on the side file and during switching can be found in [9].

2.6.1 Forward Recovery

By carefully writing information in the log records for the reorganization process [9], the state of the reorganization unit in progress at the time of a system failure can be reconstructed from the log at restart. The reorganization unit will then go forward and finish the work instead of rolling back and wasting the work that has already been done. We call this *Forward Recovery*.

This usually isn't an option for normal transaction processing, because you don't know what the user transaction wants to do next. It is a special case here, since we know this is a reorganization processing unit, and we know its operations by looking at its log records.

3 Efficient Reference Updating During On-line Reorganization

This section describes a *general* technique for efficiently changing the references in the secondary indexes during on-line database reorganization. The algorithm uses a deferred and incremental approach to make the changes in the secondary indexes. When a database record is moved, changes to index pages already in memory are made immediately and the pending changes are stored in some in-memory tables called *look-up tables*. Then those tables are consulted by the buffer manager when a secondary index leaf page is brought into the buffer by a *user* request, and changes are made at that time. The reason for its efficiency is that the changes for the secondary index leaf pages are piggybacked with user transactions, thus less I/O is caused by the reorganization process.

3.1 Assumptions

For simplicity, we assume that there is one relation in the database whose records will be moved from one physical location to another during reorganization. We assume that there are N secondary indexes for that relation and that the secondary indexes are B$^+$-trees. We further assume that all the top levels of the secondary indexes and some of their leaf pages are in memory. User transactions will use hierarchical locking [1] and follow the two phase locking protocol when they access the database. The Write-Ahead-Logging (WAL) rule is used for logging. A *unit of reorganization* consists of the moving of one record and all associated work done with reference updates and look-up table entries.

1. Check to see if the look-up tables are full. If they are, do clean-up for those tables.
2. Obtain all the locks necessary for the move. These also include locks on the secondary index leaf pages that need to be updated.
3. Move the record.
4. For each secondary index leaf page that needs to be updated:
   (a) If the page is in the buffer, make the update and log the change.
   (b) Otherwise, put the change in the look-up table.
   (c) Release the lock on the page.

Figure 3: Algorithm for Deferred Reference Updating during Reorganization

3.2 The algorithm

Figure 3 illustrates the main idea of the algorithm for one unit of reorganization (one record-move). The first step of a unit of reorganization is to see if there is any room left on the book-keeping tables. Because those tables are fixed size and they sit in memory collecting information about postponed updates, they may fill to their capacity. In this case, the algorithm would clean up those tables by bringing the corresponding secondary index leaf pages into memory and making the updates.

If there is space left on the look-up tables, we proceed to move one record. For each secondary index leaf page whose reference should be updated, if the page is already in the buffer, we make the update and log it. If the page is not in the buffer, we record the update in the look-up tables.

An interesting thing to note is that the algorithm releases the lock on the secondary index leaf page after the page is processed. This is possible, as explained in detail in [8], because we also use *forward recovery* as the recovery technique. Thus no work by the reorganization process will be lost.

3.3 Page-Oriented Piggybacking Changes of References

The reason our algorithm has an advantage over [6] is that it delays changing the references in the secondary indexes. We piggyback the reference changes with user transactions. Whenever an index leaf page is brought into the buffer, we check if there are some changes pending in the look-up tables that can be applied. If there are changes pending, then *all* those changes are made. Because all the possible changes in one page are made at the time the page is brought into buffer, we call this the *page-oriented* piggybacking. This will save some disk I/Os for the reorganization process.

Piggybacking the changes should be done by the *bufferfix* routine [1] of the database buffer manager (DBM). When a secondary index leaf page is requested, the buffer manager will return the page immediately if the page is already in the buffer. Otherwise, a disk read is done to bring it into the buffer. Before the buffer manager returns the page to the requester, it will do the piggybacking (if any) and then returns the page to the requester.

Figure 4 shows the interactions between the requester and the buffer manager. The dotted line represents the buffer manager's action when the requested page is already in the buffer, the bold line represents the buffer manager's actions when the requested page is not in the buffer.

Piggybacking of changes can be tricky if there isn't enough room left on the page for those changes. This would happen if the new address for the moved record is larger than the old address. We call this the *overflow*

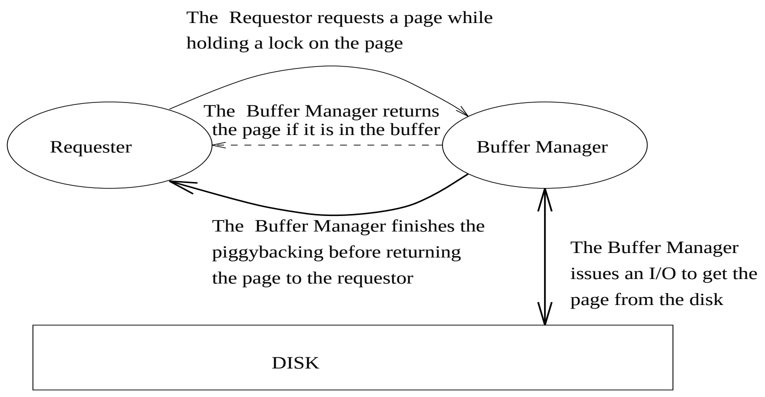

The Requestor requests a page while holding a lock on the page
The Buffer Manager returns the page if it is in the buffer
Requester
Buffer Manager
The Buffer Manager finishes the piggybacking before returning the page to the requestor
The Buffer Manager issues an I/O to get the page from the disk
DISK

Figure 4: Interaction between the requester and the buffer manager

problem. Detailed discussion of the overflow problem and our solution can be found in [8].

4 Conclusions

In this paper, we described two efficient on-line reorganization algorithms. The algorithm for on-line B^+-tree reorganization uses a three-pass approach. It first reorganizes the leaf pages of the B^+-tree using an in-place method. It selects empty space, which leads to good space utilization and reduces the log space needed. Then optionally, it resequences the leaf pages of the B^+-tree. The third pass, which reorganizes the internal pages of the B^+-tree, uses the new-place method. At any time during the reorganization of the internal pages, the algorithm at most locks one internal page with an S lock. The algorithm is efficient, recoverable and interferes minimally with user transactions.

The algorithm for reference updating during on-line reorganization will make the update if the secondary index leaf page containing the reference is already in the buffer. Otherwise it will defer the updates whose corresponding pages are not in the buffer at the time. It piggybacks those deferred updates with other user transactions' disk I/Os. Because it saves a significant number of disk I/Os, the algorithm is very efficient [8]. Comparisons of our work with other related literature can be found in [8].

The common characteristics of both methods suggest that good reorganization utilities probably should: 1) reorganize in small units and let the reorganization unit have lower priority than user transactions, 2) explore the semantics of the reorganization process in order to do less locking and logging, 3) and use forward recovery to avoid losing work.

References

[1] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc, 1993.

[2] Theodore Johnson and Dennis Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *Journal of Computer and System Sciences*, 47(1):45–76, 1993.

[3] David Lomet and Mark Tuttle. Redo recovery after system crashes. In *International Conference on Very Large Data Bases*, pages 457–468, 1995.

[4] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *International Conference on Very Large Data Bases*, pages 392–405, Brisbane, Australia, August 1990.

[5] Betty Salzberg. *File Structures: An Analytic Approach*. Prentice Hall, 1988.

[6] B. Salzberg and A. Dimock. Principles of transaction-based on-line reorganization. In *International Conference on Very Large Data Bases*, pages 511–520, 1992.

[7] Chendong Zou. *Dynamic Hierarchical Data Clustering and Efficient On-line Database Reorganization*. PhD thesis, Northeastern University, College of Computer Science, Boston, MA02115, 1996. In preparation.

[8] Chendong Zou and Betty Salzberg. Efficiently Updating Reference During On-Line Reorganization. Technical Report NU-CCS-96-08, Northeastern University, College of Computer Science, Boston, MA (USA), 1996.

[9] Chendong Zou and Betty Salzberg. On-Line Reorganization of Sparsely-Populated B+trees. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 116–125, Montreal, Canada, 1996.

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398