A method for on-line reorganization of a database

by G. H. Sockut
T. A. Beavin
C.-C. Chang

Any database management system may need some type of reorganization. However, reorganization typically requires taking a database off line, which can be unacceptable for a very large or highly available (24-hour) database. A solution is to reorganize on line (concurrently with users' reading and writing of data in the database). This paper describes a method for performing one type of reorganization on line. The type of reorganization distributes free space evenly, removes overflow, and clusters data. The method for on-line reorganization copies data while arranging the data in the new copy in reorganized form. The method then applies the database log to bring the new copy up to date (to reflect users' writing of the old copy). The method maintains a table that maps between old and new record identifiers, to match log entries with data records in the new copy.

We define *reorganization* of a database as changing some aspect of the logical or physical arrangement of the database. In Reference 1, general issues in reorganization and types of reorganization are discussed, but in this paper we discuss one type of reorganization, along with the problem in reorganizing off line.

The type of reorganization that this paper describes involves restoration of clustering. *Clustering* is the practice of storing records near one another if they meet certain criteria. One popular criterion is consecutive values in a column of the records. Clustering should reduce disk input and output for records that users often access together. We use the word *user* to refer to a person who develops or executes application programs that use the database. When users write data into the database, this writing can decrease the amount of clustering and thus degrade performance. Reorganization can restore clustering and performance.

During most types of reorganization in a typical database, the area being reorganized is *off line* or only partially available; users cannot write (and perhaps cannot even read) data in that area. However, a highly available database (a database that is to be fully available 24-hours-per-day, 7-days-per-week) should not go off-line for significant periods, of course. Applications that require high availability include those for reservations, finance (especially global finance), process control, hospitals, police, and armed forces. Even for less essential applications, many *database administrators* (people who supervise the use of a database) prefer 24-hour availability. The maximum tolerable period of unavailability is specific to the application. We asked customers of database management systems (DBMSs), not all of whom have highly available databases, to state the maximum tolerable period, and their answers ranged from 0 to 5 hours.

Even without such a preference for 24-hour availability, reorganizing a very large database might require much longer than the maximum tolerable period of unavailability. Giving examples of very large databases, Reference 2 mentions a database with several terabytes of data and the desire for one with petabytes. The author of Reference 3 considers off-line reorganization such an important problem for very large databases that he defines a very large database as one "whose reorganization by reloading takes a longer time than the users can afford to have the database unavailable."

In this paper we describe a method for on-line reorganization of a database.

These considerations call for the ability to reorganize the database *on line* (concurrently with usage or incrementally within users' transactions), so that users can read and write the database during most or all phases of reorganization. In the context of papers that do not concentrate on on-line reorganization, many people have stated the need for the ability to reorganize on line.[2,4–12] As the amount of information and dependence on computers both grow, the number of very large or highly available databases will grow, and with them the importance of on-line reorganization.

This paper describes the design of a method for on-line reorganization (specifically, for restoration of clustering). We oriented the paper primarily toward DBMS researchers and designers, and to a lesser extent toward database administrators. We include some details of design and some decisions that an implementor must make, but we do not concentrate on any actual implementation. The REORG utility in Version 5 Release 1 of IBM's DATABASE 2* (DB2*) for OS/390* (see References 13, 14) includes on-line reorganization that is very similar to what this paper describes.

Our method for on-line reorganization involves copying data from the area that users access into a new copy of that area, in reorganized form. After the copying, the database log is applied to the new copy, to reflect users' writing of the old copy. The application uses a table that maps between old and new record identifiers. The interaction between maintenance of this mapping table and processing of the log is the novelty of our work. We give an overview of our method and the underlying database structures. We then cover more details of our method. Finally, an appendix describes some alternatives.

Reorganization methods

To motivate and explain the type of reorganization that we perform and our method for performing it, we begin by sketching relevant aspects of database storage structures. These structures can degrade and thus require reorganization. We then introduce our method for reorganizing on line. This method involves a problem in identification of records, and we discuss our solution to the problem. Finally, we compare our method with previous work.

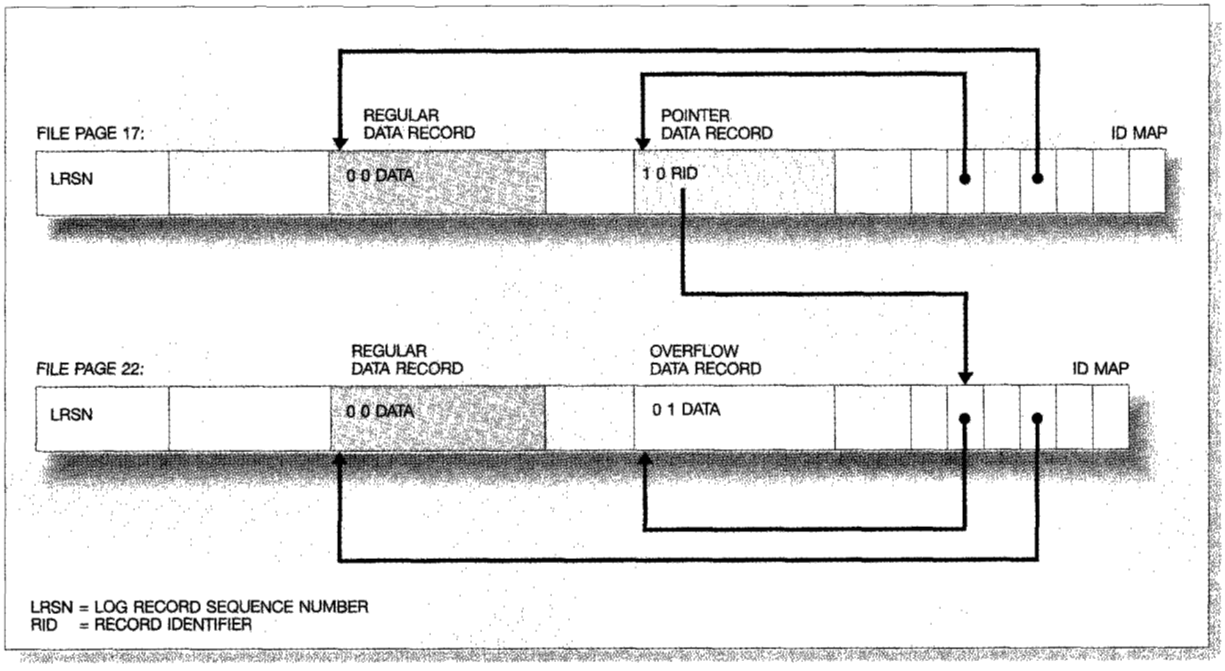
Storage structures, structural degradation, and reorganization. In this section we describe the storage structures for which we designed our method, the *degradation* (e.g., reduction in the amount of clustering) that can occur for these structures, the need for reorganization (to remove the degradation), and the control of off-line (or on-line) reorganization. We designed the method for a set of storage structures for relational databases, specifically the style of structures used in IBM's DB2 and System R[15] DBMSs. Several other relational DBMSs use comparable structures.

Storage structures for data. We begin our discussion of storage structures by describing the structures for data.

A *row* of a table in a database is a logical unit within the table. For example, if Jones is an employee, a table of employees includes a row for Jones. A row contains columns of data, e.g., for name, job title, and salary. A column can have a fixed length (which does not change) or a variable length. A variable length changes for each row according to the length of the data that users place in that column in that row.

Ordinarily, the DBMS implements a row by one *data record*, which is a lower-level (more physical) unit in

Figure 1 Example of file pages in a table space



storage. We explain shortly that sometimes the DBMS implements a row by two data records. Users see rows but do not directly see data records.

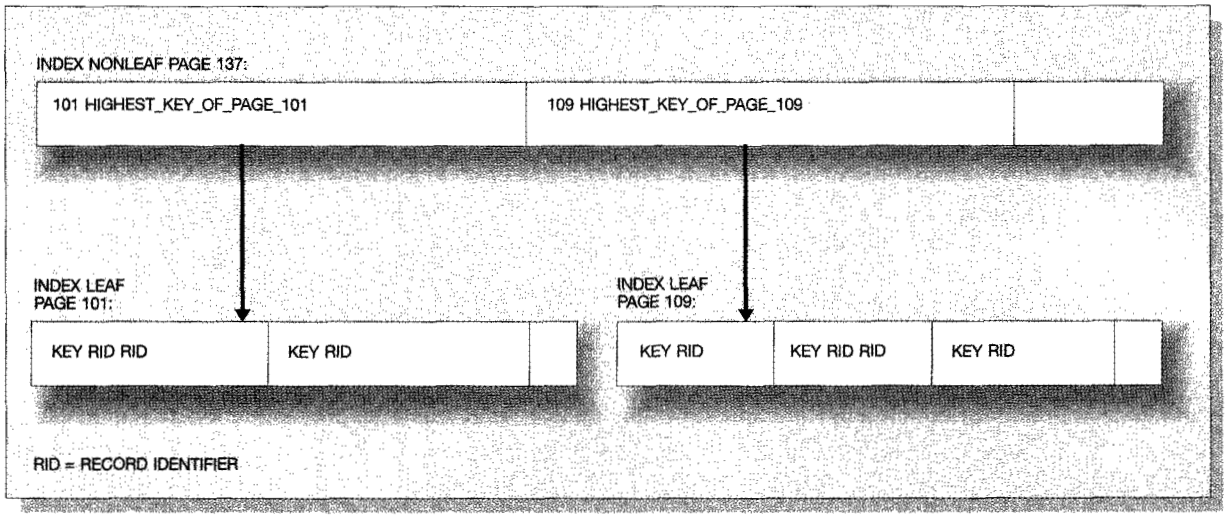
When users write rows (and thus the DBMS writes records to implement the writing of rows), the DBMS tracks the writing by appending corresponding entries to a collection of entries called the *log*. Later, it is possible to recover the data after an accidental loss by reloading from a backup copy of data and then *applying* (performing on the data) the log entries that the DBMS appended after creation of the backup copy. The *log record sequence number* (LRSN) of a log entry is a number that represents the position of that entry in the log.

A *table space* is a region of storage that stores the data records for one or more tables. For simplicity, we discuss only one table per table space. The DBMS divides a table space into units called *file pages*. Figure 1 shows the structure of file pages, which we describe gradually. The header of each file page includes the LRSN of the most recently written log entry that corresponds to writing of that page. A file page contains zero or more data records, which the DBMS allocates at the beginning of the page (after the header). Deletion of records can cause gaps between remaining records. The end of a file page contains an *ID map*, which is an array of pointers (offsets of data records within the page). We use *slot* to mean the place (if any) to which an ID map entry points.

An ID map helps to identify records. In DB2 and several other DBMSs that use the SQL[16] database language, not every table has a *unique key* (a set of columns that identifies rows). Therefore, file pages, entries in the log, and *indexes* (structures that speed access to individual records) cannot use a key for identification. Instead, they use a record's *record identifier* (RID), which consists of the page number for the record and the offset of the entry for the record within the ID map. The RID for a record can change only during reorganization.

Now we turn to the effects on storage structures during insertion of a row or during growth by update (modification) of a variable-length column of an existing row. During these operations, if the desired page lacks enough contiguous *free space* (the space available for insertions and growth), the DBMS compacts the page to make its free space contiguous. During compaction, when the DBMS moves a record,

Figure 2 Example of index pages



the DBMS updates the ID map pointer to the record; compaction does not change the record RID. If compaction produces enough free space, the data go into the desired page.

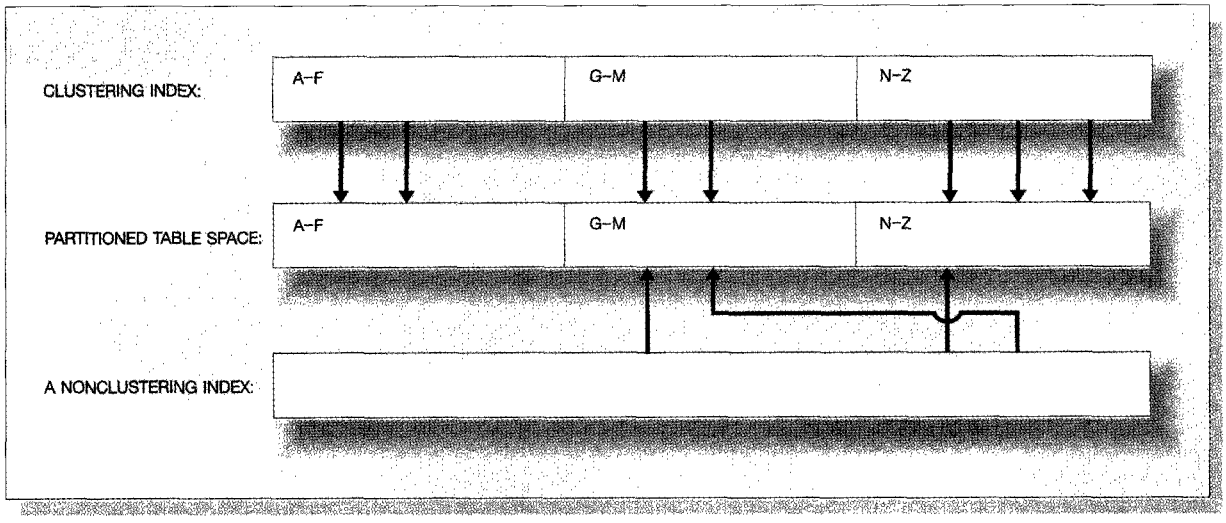
If such compaction does not produce enough space, the data go into another page. On an insertion, a new record goes into that other page. On growth of existing data, the data move into a new *overflow* data record in the other page, and the existing data record in the original page becomes a *pointer* data record, which contains the RID of the new overflow data record. Thus the DBMS sometimes implements a row by two records (a pointer record and an overflow record). Data records that do not involve overflow (hopefully, most data records) are *regular* data records. In Figure 1, the file page that we numbered 17 contains a regular data record and a pointer data record. File page 22 contains a regular data record and an overflow data record. The pointer record in page 17 contains the RID of the overflow record in page 22. The two bits in the header of each data record in Figure 1 indicate whether the record is a pointer and whether it is an overflow, respectively.

Storage structures for indexes. We continue our discussion of storage structures with the structures for indexes. A table has zero or more indexes, each of which uses an associated *key* (set of columns). For example, an employee table might have an index whose key is the department number, an index whose key is the social security number, and an index whose key is the combination of last name and first name. Within an index, the DBMS maintains the key values in sorted order. Defining a key to be *unique* means that no two rows can have the same values in the key columns.

The DBMS divides the storage of an index into units called *index pages*. Figure 2 shows the structure of index pages, which the DBMS arranges in a hierarchy. In this example, the index pages that we numbered 101 and 109 are leaves of the hierarchy, and index page 137 is a nonleaf. Each entry in a leaf page contains a key value and a list of RIDs whose records have that key value. The database designer optionally specifies that the DBMS will sort each list by RID.

Each entry in a nonleaf page points to another nonleaf page or a leaf page, although this simple figure shows no entries that point to other nonleaf pages. Each entry in a nonleaf page also contains the value of the highest key of the page to which the entry points. For example, the first entry in index page 137 contains 101 (the number of another index page) and the highest key value of page 101. The second entry contains corresponding information for page 109. A possible alternative implementation is for each entry in a nonleaf page to contain the value of the lowest key of the page after the page to which the entry points.

Figure 3 Example of the clustering index and a nonclustering index for a partitioned table space



Now we describe more about clustering, which our introduction defined. For each table, the database designer declares at most one index as a *clustering index*. In reorganization (and, whenever possible, in subsequent insertions), the assignment of data records to file pages reflects the order of the data records in the key of the clustering index. For example, the records having the first few values of the key might reside (be stored) in one file page, the records having the next few values might reside in a second file page, etc. This clustering speeds some queries.

The database designer optionally declares the clustering index to be a *partitioning* index. Here, the DBMS divides the table space (and the clustering index) into partitions according to values of the indexed key. We call the table space a *partitioned* table space. Partitions reside in separate files, whereas a nonpartitioned table space can reside in one file.

For a partitioned table space, Figure 3 shows an example of the clustering index, the table space, and a *nonclustering* index (i.e., an index that is not the clustering index). For example, if the key of the clustering index is an employee's name, partitions might represent names that begin with A through F, G through M, and N through Z, respectively, as in the figure. The arrows in the figure denote RIDs in indexes. Within the set of leaf pages of the clustering index for a partitioned table space, the RIDs for each partition are contiguous.

Within a nonclustering index, however, the RIDs for a partition need not be contiguous. For example, suppose that the key of a nonclustering index is an employee's job title. The order of RIDs in the index might be the RIDs for accountants (for all partitions), the RIDs for architects (for all partitions), the RIDs for artists (for all partitions), etc. Thus the RIDs for a partition are scattered throughout the index, since some of them might be accountants, some might be architects, etc. Figure 3 illustrates lack of contiguity via two RIDs for the G–M partition with an intervening RID for the N–Z partition. Within each value of the index key (e.g., "architect"), the RIDs might be sorted. Therefore, the RIDs for a partition might be contiguous within each key value, but they are not contiguous throughout the nonclustering index.

Structural degradation and reorganization. The storage structures that we have described can degrade. One type of degradation occurs when free space becomes unevenly distributed among the file pages of a table space. After subsequent insertions, the order of some records no longer reflects the clustering index. This type of degradation slows some queries.

A second type of degradation occurs when variable-length data grow too large to fit in their original file page. The DBMS then creates an overflow in another page and makes the original record a pointer. Indexes still contain the RID of the original record. This

causes an extra page reference and thus extra time on some queries.

Reorganization removes such structural degradation. Specifically, reorganization distributes free space evenly, removes overflows (so that each row uses just one record, not two), and clusters data. Reorganization can move records between file pages; therefore, the page that contains the record for a row after reorganization might differ from the page or pages that contained the record or records for that row before reorganization. Off-line reorganization operates by (1) unloading (copying out) the data, (2) sorting the unloaded data by clustering key, and (3) reloading the data in sorted order. Users have read-only access (i.e., they can read but cannot write the data) during unloading and sorting, but have no access during the reloading.

The method of fuzzy dumping was the inspiration for our method.

To start off-line reorganization, a database administrator issues a command where a parameter of the command specifies the name of the table space that the DBMS should reorganize. For a partitioned table space, another parameter, which is optional, specifies the partition to reorganize; absence of this parameter signifies reorganization of the entire table space. We use the term *area being reorganized* (often shortened to just *area*) to mean the table space or partition on which reorganization operates. A command for on-line reorganization will have additional parameters (discussed later).

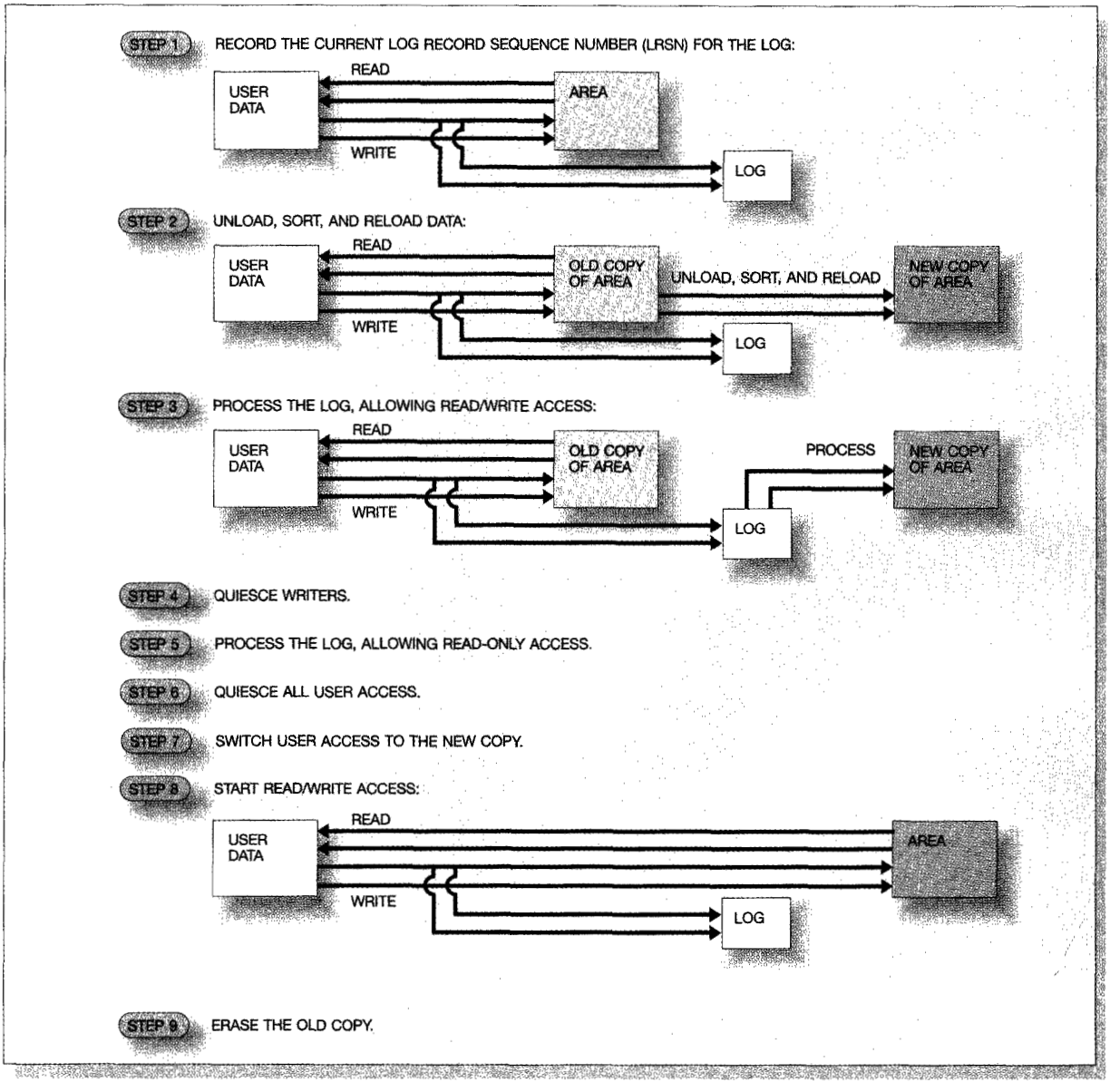
Steps of the method for on-line reorganization. Before describing our method for on-line reorganization, we describe a well-known method for unloading data during users' writing. This method for unloading inspired much of our method for reorganization. The method for unloading data is *fuzzy dumping* (also called *fuzzy image copying*).[17–23] In this method the current LRSN for the log is recorded and the desired part of the database is unloaded. If the unloading references a page that users have written in the DBMS main storage buffers, but the DBMS has not yet written the page to disk, the version that is in main storage is used. Later, if there is a need to perform recovery that uses the unloaded data, the unloaded data can be reloaded and then brought up to date (to reflect users' writing) by applying log entries (starting from the recorded LRSN). This application of the log ignores an entry whose LRSN is less than or equal to the LRSN of the page that the entry indicates, since the page already reflects that logged writing.[19]

Fuzzy dumping inspired much of our method for on-line reorganization, which we call *fuzzy reorganization*. In fuzzy reorganization, the current LRSN for the log is recorded, the area being reorganized is unloaded (while letting users read and write it), the unloaded data are sorted by clustering key, and the data are reloaded into a new copy of the area. The new copy is then brought up to date (to reflect users' writing of the old copy) by applying log entries (starting from the recorded LRSN). Future access by users is then switched to the new (reorganized) copy of the area. Figure 4 shows the main steps of fuzzy reorganization. Arrows represent the flow of information. We next briefly discuss each step of the reorganization, and we provide more details later in the paper.

In step 1, the *reorganizer* (the process that performs the reorganization) records the current LRSN for the log. During this step, users can use the normal facilities of the DBMS to read and write the area being reorganized. Users' reading and writing consist of copying data between the database (shown in the figure as "area") and variables in users' programs (shown as "user data"). In the log, the normal facilities of the DBMS also append log entries that correspond to users' writing.

In step 2, the reorganizer sequentially scans each file page in the old (original) copy of the area being reorganized, to unload the data, as in off-line reorganization. The data are then sorted by clustering key, and reloaded into a new copy of the area (unlike off-line reorganization, which involves only the original copy). We often use the terms *old copy* and *new copy* to refer to the old and new copies of the area. A backup copy of the new copy is also created as a basis for future recoverability; the figure omits this creation. We discuss this step in more detail later.

Figure 4 The main steps of fuzzy reorganization



This reorganization of the data in the area also requires changes to indexes. If the area is an entire table space, then during the reloading in step 2, all of the indexes for the table space are reconstructed (new copies of them are created). This reconstruction corrects degradation in the indexes and assures that the leaves of the indexes contain the correct new RIDs. If the area is just a partition, a partition of the clustering index is reconstructed. However, when just a partition is reorganized, the RIDs (for the partition being reorganized) must be corrected in any *nonclustering* indexes. The correction replaces the old RIDs with corresponding new RIDs. During the reloading in step 2, a copy of a subset of each nonclustering index is constructed; the subset corresponds to the records in the partition being

reorganized. Step 7 (discussed shortly) corrects the nonclustering indexes, using the copied subsets. Within a nonclustering index, the RIDs for the partition being reorganized need not be contiguous (see Figure 3).

Concurrently with the reorganizer activities in step 2, users continue to have read/write access to the old copy of the area, which the figure identified as simply "area" in step 1. At the end of step 2, the reorganizer records the current LRSN for the log. This value of the LRSN exceeds the previously recorded value if users have written into the database since the previous recording.

Step 3 (processing of the log) can execute iteratively. In each iteration, the reorganizer reads a subset of the log, namely the entries between the two most recently recorded LRSNs. The log entries are sorted by RID and applied to the new copy of the area, to bring the new copy up to date. The subset of the log reflects users' writing that occurred during the previous step or during the previous iteration of this step. Users have read/write access to the old copy during this step. At the end of an iteration of this step, criteria (which we discuss later) are used to choose between performing this step again or going to the next step. If this step is performed again, the current LRSN for the log is first recorded. If this step is not performed again, the backup copy is brought up to date by appending to it the changed pages of the new copy, before continuing to the next step. We discuss step 3 in much more detail later.

The sorting of log entries by RID in step 3 improves the *locality of reference* of log application, i.e., the extent to which successive log entries refer to data records that reside on the same file page or nearby file pages. Thus the sorting should speed the log application. It also eases the detection (and omission during application) of a sequence of logged operations that has no net effect (e.g., insert . . . update . . . delete).

Also in step 3, if the area being reorganized is an entire table space, when a log entry is applied to data, the corresponding changes to the new copies of the indexes are made. For example, if a log entry deletes a record, the RID for that record is deleted from any indexes that contained the RID. If the area is just a partition, corresponding changes are made to the new copy of the partition of the clustering index and to the new copies of the subsets of the nonclustering indexes.

In step 4, the reorganizer quiesces *writers* (user transactions that write into the area). This *quiescing* blocks new writers and waits for existing writers to finish. The current LRSN for the log is recorded, and users continue to be able to read.

In step 5 (processing of the log), the reorganizer processes the log entries between the two most recently recorded LRSNs for the log, as in an iteration of step 3. This last step of processing of the log is needed only to handle writing that was in progress when (or that began after) the previous step of processing finished reading its subset of the log. Users have read-only access to the old copy during this step. At the end of this step, as at the end of the last iteration of step 3, the recently changed pages are appended to the backup copy.

In step 6, the reorganizer quiesces all user access of the area.

In step 7, the reorganizer switches users' future access to the new (reorganized) copy of the area. This switch is performed by renaming (exchanging the names of) the files that underlie the old and new copies. This renaming effectively changes the mapping from logical to physical.

Also in step 7, users' future access to indexes is switched. If the area being reorganized is an entire table space, the names of the files that underlie the indexes are exchanged.

If, instead, the area is just a partition, the names of the files that underlie the partition of the clustering index are exchanged. The individual RIDs for this partition in any nonclustering indexes are also corrected (in place, not by copying). In each nonclustering index, for each key value, this correction consists of replacing the old RIDs for this partition by the new RIDs, which are found in the constructed subset of the index. Within each key value, the old RIDs for this partition are contiguous only if the index definition specified that the DBMS will sort the RIDs (and thus group them by partition). Therefore, the correction of RIDs is faster for a sorted index than for an unsorted index.

During step 7, users have no access to the area, with one exception. The exception applies during the correction of nonclustering indexes for reorganization of just a partition. Queries that read the clustering index and the data are allowed (after the files are renamed), and updates are allowed for columns that

do not appear in any nonclustering indexes. In each nonclustering index, an implementation might also allow queries that read just the index key values but not the RIDs.

In step 8, the reorganizer allows read/write access of the area to resume. Users can then use the normal facilities of the DBMS to read and write the new copy (identified in the figure as simply "area"), in the same way that they formerly read and wrote the old copy.

Step 8 also provides an opportunity for an alternative technique for creating a backup copy of the new copy. Our description so far includes creation of a backup copy during step 2 and bringing the backup copy up to date during steps 3 and 5. In the alternative technique, we start creating a backup copy at the beginning of step 8 via a facility that allows concurrent writing,[24,25] and we allow read/write access to resume as soon as the backup copying begins.

The novelty of our work is in the interaction between the table and processing of the log.

In step 9, the reorganizer erases the mapping table and the file for the old copy of the area. Similarly, if the area being reorganized is an entire table space, the old copies of the indexes are erased. If the area is just a partition, the old copy of the partition of the clustering index and the copies of the subsets of the nonclustering indexes are erased.

This method for reorganization allows reading and writing during almost all steps, including the first step of log processing. Subsequent steps, which occur after processing of most of the logged writing, involve a period of read-only access and a period of no access.

Identification of records. The fuzzy reorganization that we just described can have a problem in identification of records. We mentioned that an entry in the log identifies a record by the RID. In fuzzy dumping, RIDs do not change. In recovery that uses the result of fuzzy dumping, application of an entry in the log can use the entry RID to identify the record to which the entry should apply. As an inherent part of reorganization, however, RIDs do change. Log entries in our method for reorganization correspond to users' writing of the old copy and thus use the old RIDs. Application of a log entry to the new copy requires identification of the record in the new 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 RIDs. The method uses this table to translate log entries before sorting them by new RID and applying them to the new copy.

The interaction between maintenance of this mapping table and processing of the log (discussed later) constitutes the main novelty of our work. The main novel feature is the appropriate writing of the mapping table for each log entry. This writing reflects (1) the state of the data record before processing of the log entry and (2) the type of log entry. For a log entry that represents an insertion, this writing includes the use of an estimated new RID (as a basis for sorting) and eventual translation of the estimated RID to an actual new RID. A patent is pending on the interaction.

Comparison with previous work. In this subsection we compare our method for reorganization with previous work.

Our calculation of clustering is straightforward and not novel. We sort the data records by the clustering key, and we assign the appropriate amount of data to each page. Therefore, we do not compare our method with previous work in calculation of clustering. References 26 and 27 discuss issues and survey previous work in many aspects of on-line reorganization, including methods that restore clustering.

Instead, our novelty is in the interaction between maintenance of the mapping table and fuzzy reorganization processing of the log. Therefore, we compare our method only with previous work in fuzzy reorganization and in tables that map identifiers. We are unaware of any previous work that combines fuzzy reorganization log application and use of a mapping table.

Several authors mention (but do not describe in detail) what we call fuzzy reorganization.[28–30] Also, use of a specialized file (which has some similarities

to a log) can bring a newly created index up to date,[30–34] but in this case the RIDs do not change. Reference 35 mentions the use of such a specialized file for on-line reorganization, but it does not describe the changing of RIDs.

In systems where every record has a unique identifier that does not change during reorganization, fuzzy reorganization does not need a mapping table. One example of such a system is the National Crime Information Center of the United States government agency, the Federal Bureau of Investigation. The system reorganizes by copying data and then applying deletions from the log.[1] During reorganization, the system allows deletions but not insertions or updates. A second example is a facility for replication, e.g., DataPropagator* Relational[36] or Replidata/MVS.[37] Here reorganization can copy data and then use the replication facility; using such a facility resembles applying the log. A third example involves on-line splitting of a partition into two partitions; here reorganization applies the log after copying the second part (e.g., half) of the original partition into the new partition.[38,39] In our environment, however, many users dislike a requirement for every row to have a unique identifier.

Our technique of using a mapping table has several advantages.

In languages that support linked data structures, *garbage collection*[40] is reclamation of storage that is no longer reachable from variables and thus is no longer used. Reference 41 describes on-line garbage collection for persistent data by copying data and applying the log. This garbage collection does not cluster by values of a key. Each record in the old copy of the area has a field that stores the address of the corresponding record in the new copy. Processing of the log uses this field to translate addresses in log entries. In a database context (which was not the context for Reference 41), our technique of using a mapping table has several advantages over the technique of storing the address of the new record in the old record.

One advantage involves deletion (which does not apply to the environment of Reference 41, whose users do not explicitly delete records). Suppose that after the reorganizer has copied a record, a user deletes the record (in the old copy of the area), and the DBMS appends to the log an entry for the deletion. The reorganizer will eventually find the log entry, translate its address (RID) from old to new, and apply the deletion to the new copy of the area, to delete the record there. Between the user's deletion and the reorganizer's processing of the log entry, the DBMS might reuse the space that the deleted record occupied. Therefore, we could not safely store the new address in the old (deleted) record. A mapping table can safely store the mapping of addresses.

The second advantage involves input and output. A data record can be much larger than a mapping table entry, so the set of all data records can require many more pages than the set of all mapping table entries. Therefore, adding and reading mapping table entries can involve less page input and output activity than writing and reading the new addresses in the old data records.

The third advantage is that our technique requires less locking for each record in the old copy. Both techniques require a shared lock while unloading the old record. However, our technique requires no lock while reloading (and adding an entry to the mapping table) and no lock while processing the log (and translating the address). Storing the new record address in the old record can require an exclusive lock while reloading (to write a new address in the old record) and a shared lock while processing the log (to translate the address). Therefore, our technique is faster and allows more concurrency in the database.

The fourth advantage is avoidance of extra space (which is permanent) in each data record for the address of the new record. In the most general case, an implementation of storing the new address in the old record can require the space, although the implementation in Reference 41 uses space that already existed.

The technique of storing the address of the new record in the old record seems simpler and has another advantage in requiring only one field for storing the new address, instead of fields for both the old address and the new address (in a temporary mapping table).

Table 1 Structure of the mapping table

Type (TYPE)	Source Record Identifier (SOURCE_RID)	Target Record Identifier (TARGET_RID)	Log Record Sequence Number (LRSN)
CR	RID of old regular or overflow	Actual RID of new regular or pointer	LRSN
CE	RID of old regular or overflow	Estimated RID of new regular	LRSN
P	RID of old pointer	(Unused)	LRSN

First character represents the source
C = Record contains columns of data
P = Record is a pointer
Second character represents the target
E = Estimated RID of a new record
R = Actual RID of a new record

References 42 and 43 describe the use of a mapping table for loading data into an object database. One of the stated purposes of loading is restoration of clustering, although the authors do not mention on-line restoration. Objects can contain references to other objects. In the database, these references use object identifiers. Since the object identifiers are unknown before the loading, the file that is the source of loaded data uses surrogates (e.g., integers) for identifiers. During loading, the system constructs a mapping table and uses it to translate surrogates to identifiers in the database. Our method satisfies several requirements that do not arise in the environment of References 42 and 43. These requirements include the ability to handle (1) two sources of data (the log and the old copy of the area), not just one, (2) a variety of possible timing relationships between unloading of data and generation of log entries for those data, (3) updates and deletions (not just insertions) in the log, and (4) overflow and pointer records (not just regular records).

Reference 44 describes reorganization by copying. This method allows read-only access during reorganization, so it does not use the log.

Finally, a method for reorganization in place (i.e., not by copying) uses a table that maps RIDs to translate entries in the leaves of indexes.[45,46] This method does not use the log.

Structure of the mapping table

We begin our more detailed discussion of our method for reorganization by describing the mapping table. This table (shown in Table 1) is a database table or a special structure; the choice is an implementation decision. The columns are TYPE, SOURCE_RID, TARGET_RID, and LRSN. The TYPE column actually contains numbers (or single characters), but the figure uses symbols for them. In the first character of a symbol, C means that the record corresponding to the old RID contains columns of data, and P means that the record is a pointer. The second character (if any) of a symbol is R or E. R means that the target record identifier (TARGET_RID) is the *actual* RID of a new regular record or pointer record. E means that TARGET_RID is the *estimated* RID of a new record that we will insert later when we apply the log. We will explain this estimation mechanism later. For a TYPE of CR or P, the log record sequence number (LRSN) contains the LRSN of the old page that contained the old RID, as of the time when reorganization unloads the page. For a TYPE of CE, LRSN contains the LRSN of an insertion entry that we find in the log.

Whenever we read the mapping table, we access it by finding the entry whose source record identifier (SOURCE_RID) contains a specified value. Therefore, an efficient implementation of the mapping table might use only an index (containing all the columns, starting with SOURCE_RID), without actually storing separate data in a table. In our discussions, we will assume that the implementation uses only an index.

Behavior of the DBMS during writing

To explain more about the unloading in step 2 of our method for reorganization, we must first explain some behavior of the DBMS during users' writing. This behavior, which is not part of reorganization, influenced our design of on-line reorganization, particularly unloading of data and processing of the log.

Table 2 High-level operations on rows and corresponding low-level operations on records

High-Level Operation on a Row	Condition Before the Writing	Low-Level Operations on Records
(1) Insertion	(Any)	One insertion of a regular record
(2) Update	There is no overflow, and the new data fit in the page that contains the record.	One update (from regular to regular)
(3) Update	There is no overflow, but the new data are too large for the page that contains the record.	One update (from regular to pointer) and one insertion (of an overflow record); an overflow is created.
(4) Update	There is an overflow, and the new data fit in the page that contains the overflow.	One update (from overflow to overflow)
(5) Update	There is an overflow, the new data are too large for the page that contains the overflow, and there is now room in the page that contains the pointer.	One update (from pointer to regular) and one deletion (of an overflow record); the data return to the page that contains the pointer.
(6) Update	There is an overflow, the new data are too large for the page that contains the overflow, and there is still not room in the page that contains the pointer.	One update (from pointer to pointer), one deletion (of an overflow record), and one insertion (of an overflow record in a third page)
(7) Deletion	There is no overflow.	One deletion of a regular record
(8) Deletion	There is an overflow.	Two deletions (of a pointer record and an overflow record)

Table 2 contains a summary of the discussion that follows.

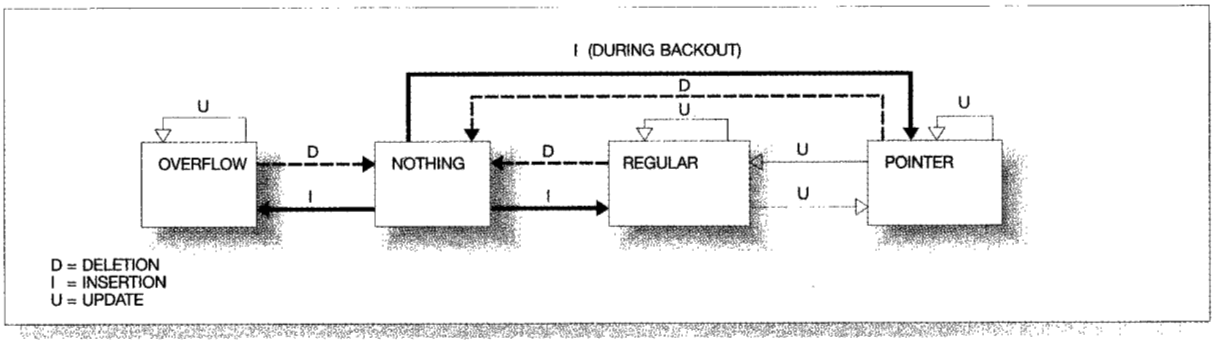
Users write rows of tables via high-level operations of insertion, update, and deletion. For each such high-level operation on a row, the DBMS performs one, two, or three low-level operations on records. The DBMS also appends log entries that correspond to the operations on records. A log entry contains an LRSN, an old RID, and a low-level operation (among other things).

For each high-level operation on a row in the left column of Table 2, the right column shows the corresponding low-level operations on records. The description of a low-level operation includes the type of record to which the operation applies. For example, an update (from regular to pointer) in row 3 of Table 2 is an update that operates on a record that is a regular record before the update but becomes a pointer record as a result of the update. For each set of low-level operations, the center column shows the condition (immediately *before* the writing) under which the DBMS chooses that set of low-level operations to implement the high-level operation. The condition of "any" in row 1 of Table 2 means that to implement the high-level operation of insertion, the DBMS *always* chooses one insertion of a regular record.

For example, suppose that a user requests an update of a row. Suppose that there is no overflow yet (i.e., the row resides in a regular record), but the update would cause the record to become too large to fit in its current page. Then the DBMS updates the regular record to become a pointer record, and the DBMS inserts an overflow record to contain the data. Row 3 in Table 2 describes this behavior.

During *backout* of a transaction (i.e., the undoing of writing if a failure occurs during the transaction), the DBMS performs operations and appends corresponding log entries to reverse the original operations. For example, row 8 in Table 2 shows two deletions of

Figure 5 State transition diagram for a RID's slot during users' writing



records. To reverse these operations, we use two insertions (of a pointer record and an overflow record).

Reflecting the operations in Table 2, Figure 5 shows a state transition diagram for a RID slot during users' writing. The states indicate what the slot for the RID contains (an overflow record, nothing, a regular record, or a pointer record). The transitions represent operations on records, i.e., deletion (D), insertion (I), and update (U).

For example, row 8 of Table 2 shows that if there is an overflow, the DBMS represents a user's deletion of a row as deletions of a pointer record and an overflow record. Figure 5 depicts the deletion of a pointer record by the D transition from POINTER to NOTHING. It depicts the deletion of an overflow record (which appears in rows 5, 6, and 8 of Table 2) by the D transition from OVERFLOW to NOTHING. Similarly, Figure 5 uses transitions to depict all the other low-level operations in Table 2. We noted that during backout of a transaction, the DBMS performs operations to reverse the original operations. The transition labeled "I (during backout)" occurs *only* during backout of a transaction that deleted a pointer.

Unloading, sorting, and reloading of data (step 2)

The behavior of the DBMS during users' writing, which we just described, influenced our design of step 2. This step unloads the data, sorts the data by clustering key, and reloads the data. It also builds the mapping table. Figure 6 shows these activities in more detail than the UNLOAD, SORT, AND RELOAD action in step 2 of Figure 4. In the sections that follow, we gradually explain the activities in Figure 6.

Unloading and sorting of data. First, we unload data by scanning the file pages and their ID maps (in the old copy of the area) and by unloading data into a file (the *unload file*). The UNLOAD activity in Figure 6 depicts this unloading.

In this scanning, when we find a regular or overflow record, we unload the data, the old RID, and the LRSN of the old page that contains the record.

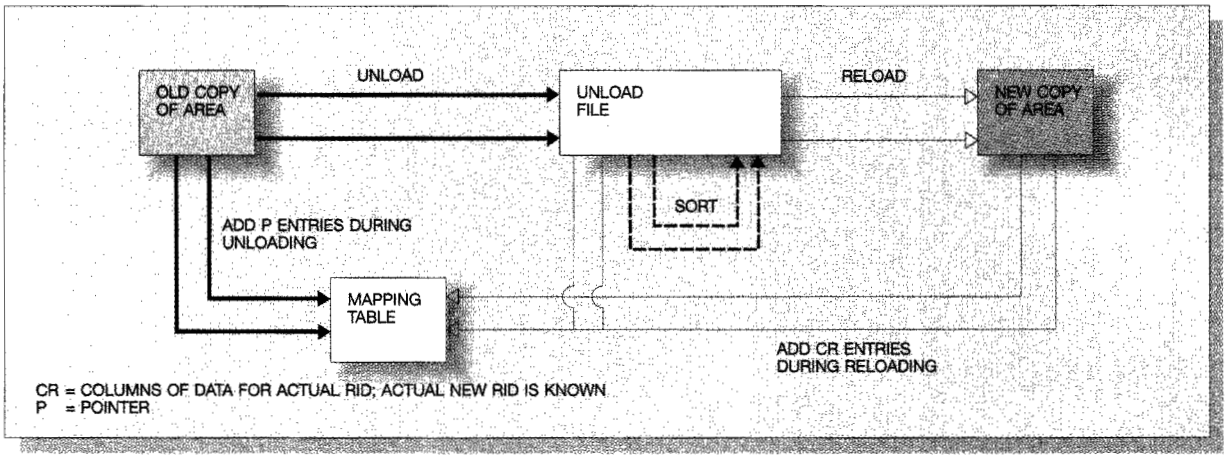
When a pointer record is found, a P entry (including values for the SOURCE_RID and LRSN columns) is added to the mapping table (Table 1), as the ADD P ENTRIES activity in Figure 6 shows. The pointer is not followed. Unloading an overflow when the page that contains the overflow is scanned has more locality of reference and thus is faster than following the pointer and unloading the overflow when the page that contains the pointer is scanned.

After unloading, the unload file is sorted by clustering key, as the SORT activity in Figure 6 shows.

Reloading of data. After the unloading and sorting, records are reloaded into the new copy of the area, as the RELOAD activity in Figure 6 shows. For each page in the new copy, the LRSN in the page header is set to 0; any future recovery of the new copy will use log entries starting after completion of reorganization.

This reloading does not produce any overflow in the new copy, even if an old record overflowed. Only the

Figure 6 Activities in unloading, sorting, and reloading of data



later application of an update found in the log can cause an overflow in the new copy.

When a record is reloaded, a CR entry is added to the mapping table, using the old RID and LRSN from the unload file and the new RID from the new copy. The ADD CR ENTRIES activity in Figure 6 depicts this addition.

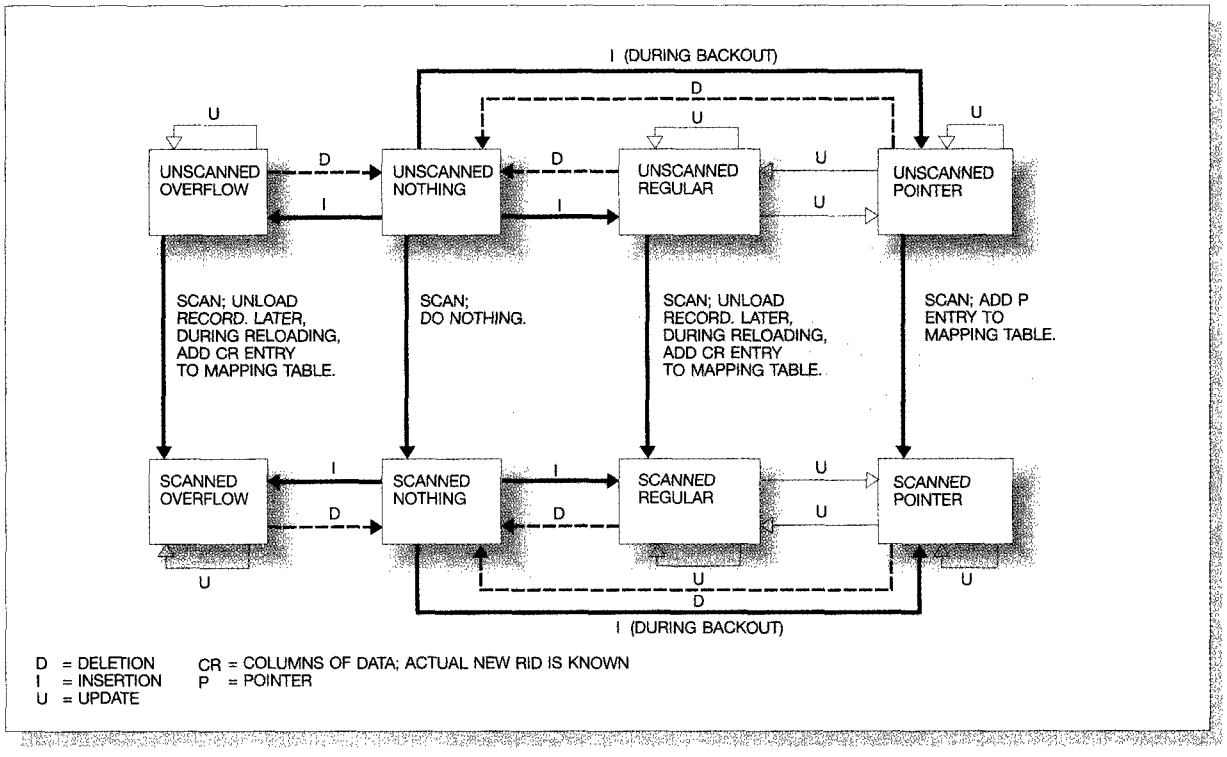
To summarize what we have described for unloading (and, to a lesser extent, what we have just described for reloading), Figure 7 shows a state transition diagram for a RID slot. It shows the combinations of events that can occur to a slot during users' writing and the reorganizer's unloading. In Figure 7, we have extended the diagram of Figure 5 by adding the scanning and unloading. The states indicate (1) whether the reorganizer has scanned that RID yet during the unloading in step 2 and (2) what the RID slot contains (an overflow record, nothing, a regular record, or a pointer record). The four states in the top row are the possible initial states (before scanning of the RID). The four states in the bottom row are the possible final states (after scanning of the RID). The transitions among the top four states and the transitions among the bottom four states represent the DBMS operations (resulting from users' writing) on unscanned and scanned records, respectively. These transitions use the labels of Figure 5 (D, I, and U).

Each vertical transition between top and bottom states represents reorganizer scanning of a RID during the unloading in step 2. For each such transition, the diagram shows the actions of the reorganizer during the scanning. For the two cases of scanning that unloads a record, the diagram also shows reorganizer actions during the later reloading in step 2. For example, we explained that when an overflow record is scanned, its data (among other things) are unloaded. We also explained that when a record is reloaded (in the new copy), a CR entry is added to the mapping table. The diagram depicts these actions by the transition from UNSCANNED OVERFLOW to SCANNED OVERFLOW. Scanning a regular record uses a similar transition. When an empty slot is scanned, nothing is done, as shown by the transition from UNSCANNED NOTHING to SCANNED NOTHING. When a pointer record is scanned, a P entry is added to the mapping table, as shown by the transition from UNSCANNED POINTER to SCANNED POINTER.

Processing of the log (steps 3 and 5)

Now we turn from unloading and reloading (which construct the mapping table) to processing of the log (which uses the mapping table). Steps 3 and 5 in the reorganization method process subsets of the log. This processing includes sorting and other manipulation of log entries. To speed the processing, and to avoid modifying the original log (which later recovery, if any, might need), the processing uses a buffer. The buffer contains copies of the log entries, and it contains pointers to the copies. When log entries are copied into the buffer, the pointers are also constructed. The old RID, new RID, and LRSN of the

Figure 7 State transition diagram for a RID's slot during users' writing and the reorganizer's unloading



log entry are stored in a *prefix* (extra space) for each copy. The sorting and most of the other manipulation during log processing operate on the pointers and the prefixes, and some of the manipulation operates on the copies. In the remainder of this paper, most references to the log actually refer to the copy of the log in the buffer.

If the number of log entries to process in an iteration of log processing (step 5 or an iteration of step 3) exceeds the capacity of the buffer, then several minor iterations (each operating on one buffer of log entries) are performed within that major iteration. For simplicity, this paper usually describes log processing as if each major iteration contained just one minor iteration.

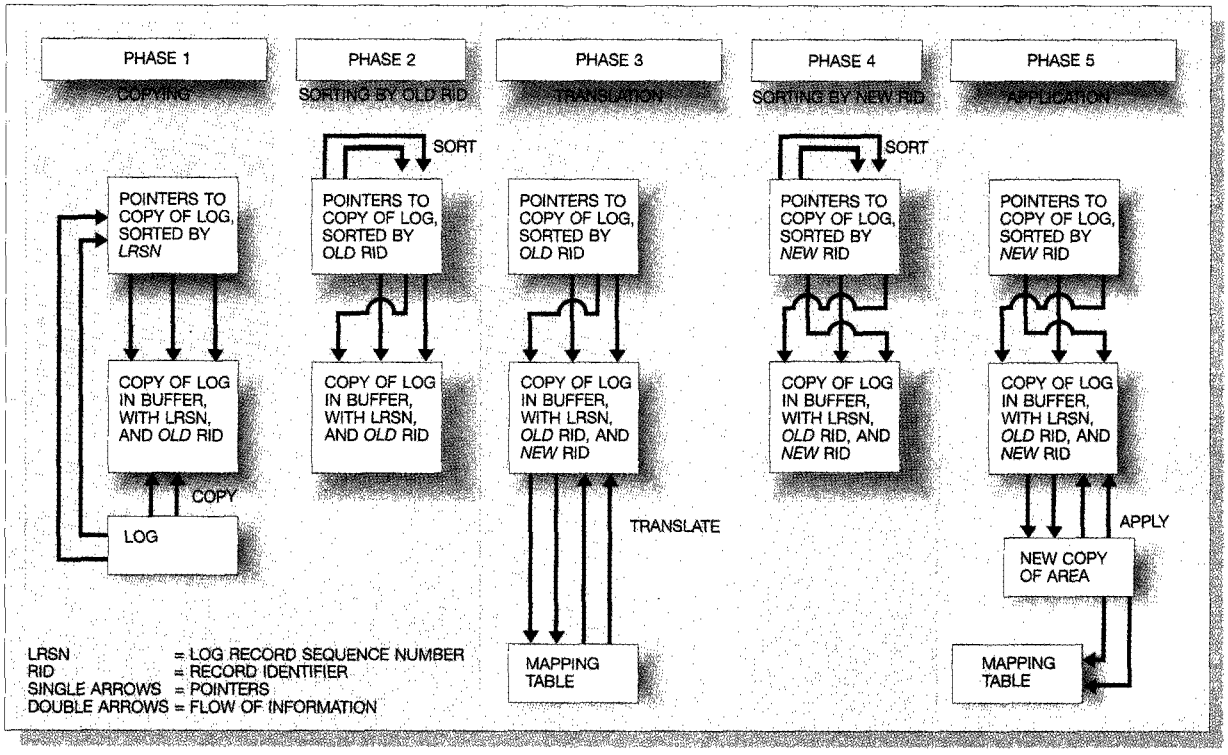
In the processing, sorting the pointers by old RID speeds the access to the mapping table for translation from old to new RIDs.

Sorting the pointers by new RID speeds application of the log by increasing the locality of reference in the access to the new copy of the area. This sorting also eases the detection (and omission during application) of a sequence of logged operations that has no net effect (e.g., insert, update, delete). This omission covers log entries for which the mapping table no longer contains the appropriate entries.

Therefore, each iteration of processing of the log includes two types of sorting, and it has the phases in Figure 8. In the figure, single arrows represent pointers, and double arrows represent flow of information. For each phase, the text that describes the copy of the log (including prefixes) and the pointers represents the state at the end of the phase.

In phase 1 of processing of the log, the log entries are copied and pointers to the log entries are constructed. In the prefixes, the old RIDs and LRSNs, but not the new RIDs, are filled in. In phase 2, the pointers are sorted by old RID. In phase 3, the translated (new) RID of each log entry is calculated, and the new RID is stored in the prefix. In phase 4, the pointers are sorted by new RID. In phase 5, the log entries

Figure 8 Phases of processing of the log



are applied to the new copy of the area. An implementation might merge phase 2 into phase 1 and merge phase 4 into phase 3. We will explain much more about the phases later.

In phase 3 (translation), for an insertion, the new RID is not yet known, since the application of the insertion will occur in phase 5. Therefore, phase 3 calculates an estimated new RID for the inserted record. Phase 5 replaces the estimated new RIDs with actual new RIDs.

To speed the processing of the log, an implementation might use two buffers and process them concurrently. It can perform phases 3, 4, and 5 using one buffer (for the log entries in one minor iteration) while performing phases 1 and 2 using the other buffer (for the log entries in the next minor iteration).

Next we discuss the phases of processing the log in more detail, along with a discussion of the control of iterations of log processing.

Phase 1: Copying. In the first phase, the log entries are copied and pointers to them are constructed. The DBMS appended entries to the log (and they are read) in LRSN order, so the pointers are initially sorted by LRSN; no explicit sort by LRSN is performed.

Phase 2: Sorting by old RID. After the copying, the pointers to the copies of log entries are sorted using a major sort by old RID and a minor sort by LRSN. The major sort speeds the later access to the mapping table. The minor sort preserves the order of operations on a RID slot.

Phase 3: Translation. After the set of pointers is sorted, the set is scanned and the RIDs are translated. As we describe in detail below, the translated (new) RID of each log entry is calculated and stored in the prefix.

One aspect of this translation imitates an aspect of a DBMS log application. We noted that log application (as part of recovery) ignores a log entry whose LRSN is less than or equal to the LRSN of the page

that the entry indicates. This behavior handles entries that are inapplicable to the unloaded data. Such a situation can arise if a user writes a record after unloading begins (e.g., in fuzzy dumping) but before the unloading reaches the record. Here is an example sequence of events: (1) A RID slot contains a record when unloading begins. (2) A user deletes the subject record, and the DBMS appends to the log a deletion entry for that RID. (3) The unloading reaches that slot and finds nothing. (4) During log application, the DBMS finds the entry. Applying the entry for deletion would not make sense, since the unloading found nothing. Therefore, the DBMS ignores the entry. Similarly, the translation (as part of on-line reorganization) deletes (effectively ignores) the copy of an inapplicable log entry and the pointer to the copy. The LRSN comparison (which the translation performs for updates and deletions) is necessary during the first iteration of log processing; it is optional during later iterations.

The next three sections describe how log entries for insertion, update, and deletion are translated. Then we summarize these translations in a figure.

Translation of a log entry for insertion. We begin with translation of an insertion. If the old RID has an entry in the mapping table, the copy of the log entry and the pointer to the copy are deleted. Otherwise, the log entry is processed, as described here.

If the insertion represents a regular or overflow record, a new RID is estimated, based on the page numbers for records that have similar values for the clustering key and that already exist in the new copy of the area being reorganized. It is efficient (for application after sorting) but not necessary for the estimated new RID to approximate the eventual actual new RID. For unique identification during the later phase for application, the estimated new RID must differ from all estimated or actual new RIDs now in the mapping table. Therefore, a counter of insertions during the current iteration of log processing is maintained, and the counter is concatenated to the estimate. A CE entry is then added to the mapping table, using the old RID of the log entry, the estimated new RID, and the LRSN of the log entry. In the prefix for the copy of the log entry, the estimated new RID is stored.

Suppose instead that the insertion represents a pointer record, which the DBMS inserts only in a backout. Then a P entry is added to the mapping table, using the old RID and LRSN of the log entry. The copy of the log entry and the pointer to the copy are deleted. This behavior resembles the scanning of a pointer record during unloading.

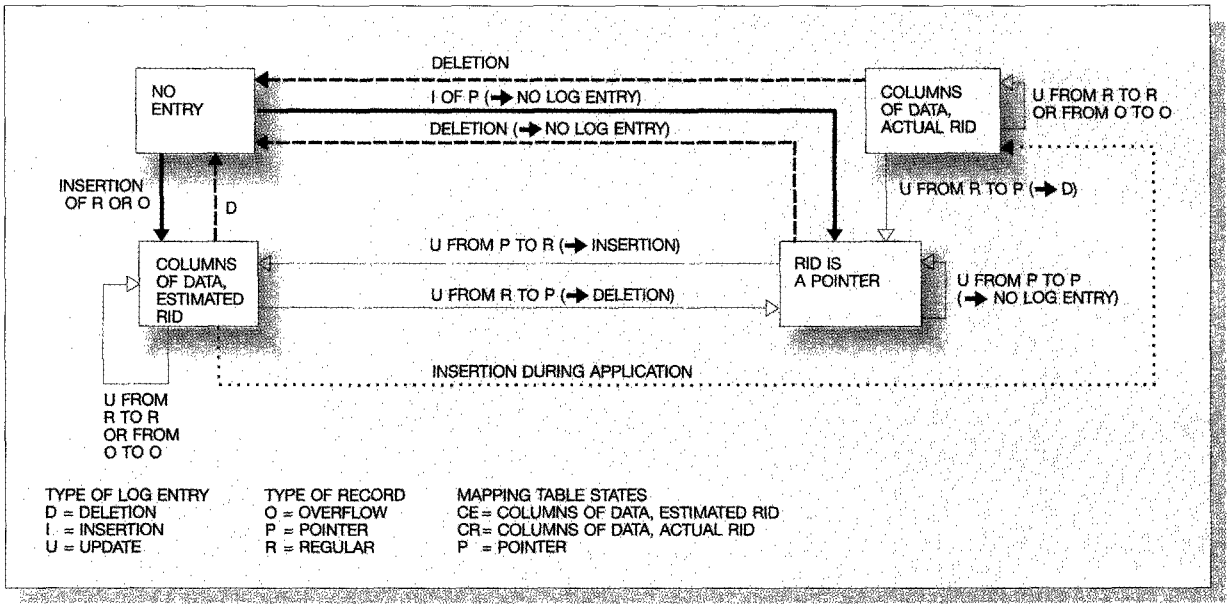
Translation of a log entry for update. Now we turn to translation of an update. Suppose that the old RID has no entry in the mapping table, or the LRSN in the log entry is less than or equal to the LRSN in the mapping table entry. Then the copy of the log entry and the pointer to the copy are deleted. Otherwise, the log entry is processed according to the type of update (e.g., from regular to regular) and the type of entry in the mapping table (CR, CE, or P), as described here. The right column of Table 2 includes all the possible types of updates.

First, consider a log entry that updates from regular to regular or from overflow to overflow. If the old RID has a P entry in the mapping table, an error is announced by terminating the reorganization abnormally; any log entry with an appropriate LRSN should not contain an inappropriate update. If the old RID has a CR or CE entry in the mapping table, then in the prefix for the copy of the log entry, the content of TARGET_RID is stored as the new RID.

Alternatively, consider a log entry that updates from pointer to pointer. If the old RID has a CR or CE entry in the mapping table, the reorganization is terminated abnormally. If the old RID has a P entry in the mapping table, the copy of the log entry and the pointer to the copy are deleted.

Consider instead a log entry that updates from regular to pointer. If the old RID has a P entry in the mapping table, the reorganization is terminated abnormally. If the old RID has a CR or CE entry in the mapping table, these actions are performed: In the prefix for the copy of the log entry, the content of TARGET_RID is stored as the new RID. In the log entry, the type of log entry is changed from update to deletion. In the mapping table, TYPE is changed to P. The following discussion contains the reason for this behavior. Data for this slot were already found in the old copy or in the log. However, this slot (in the old copy) then became a pointer. Therefore, since the data that we have already found are not useful, a deletion entry is created in the log. The desired data are found when the RID for the corresponding overflow is processed in the old copy or in the log. Since the new copy of the data corresponds to old regular records and old overflow records but not old pointer records, an update from regular to pointer is treated similarly to a deletion.

Figure 9 State transition diagram for an old RID's mapping table entry during processing of the log



Finally, consider a log entry that updates from pointer to regular. If the old RID has a CR or CE entry in the mapping table, the reorganization is terminated abnormally. If the old RID has a P entry in the mapping table, the following actions are performed. As in an insertion, a new RID is estimated; in the prefix for the copy of the log entry, the estimated new RID is stored; in the log entry, the type of log entry is changed from update to insertion; and in the mapping table, TYPE is changed to CE, and TARGET_RID is changed to the estimated new RID. Since the new copy of the data corresponds to old regular records and old overflow records but not old pointer records, an update from pointer to regular is treated similarly to an insertion.

Translation of a log entry for deletion. Next we discuss translation of a deletion. Suppose that the old RID has no entry in the mapping table, or the LRSN in the log entry is less than or equal to the LRSN in the mapping table entry. Then the copy of the log entry and the pointer to the copy are deleted. Otherwise, the log entry is processed as described next.

Suppose that the old RID appears in a CR or CE entry in the mapping table. Then the contents of TARGET_RID are stored as the new RID in the prefix for the copy of the log entry, and the entry is deleted from the mapping table.

Suppose instead that the old RID appears in a P entry in the mapping table. Then the copy of the log entry and the pointer to the copy are deleted along with the entry in the mapping table. The reason is that the log contains another entry to delete the corresponding old overflow, and that other entry, not this entry, is translated to delete the data.

States of a mapping table entry during translation. The phase for translation (which we have been discussing) and the phase for application (which we discuss shortly) can write (i.e., add, delete, or change) entries in the mapping table. For each old RID, at any time, the mapping table contains a CE entry, a CR entry, a P entry, or no entry; these are the four possible states of the RID in the mapping table. The writing of an entry can change the state of the RID for an entry.

To summarize the changing of states during translation (and, to a lesser extent, during application), Figure 9 shows a state transition diagram for an old RID's entry (if any) in the mapping table. The rectangles represent the four possible states. The tran-

sition from CE to CR (dotted line in Figure 9) represents the processing of an insertion log entry during the phase for application, which we will discuss shortly. The other transitions represent the processing of log entries during the phase for translation, which we discussed in the previous three sections. Each transition is labeled with the type of log entry (I, D, or U). For an insertion (I), we also show the type of old record (R for regular, P for pointer, or O for overflow). For an update (U), we also show the type of old record before and after the update. If the translation changes the type of log entry, we show an arrow and the changed type in parentheses. If the translation deletes the copy of the log entry and the pointer to the copy, we also show an arrow and "no log entry" in parentheses.

For example, consider the horizontal transition from state CE to state P. This transition represents a log entry that updates (U) the old record from a regular record (R) to a pointer record (P). Before the update, the old RID has a CE entry in the mapping table. In this transition, the log entry is changed from update to deletion (D), and the type of entry in the mapping table is changed from CE to P.

Next we list the translations of log entries, in the order in which we described them in the previous three sections. For each translation, we identify the corresponding transition in the diagram: (1) Translation of an insertion of a regular or overflow record includes adding a CE entry to the mapping table. The corresponding transition in the diagram (labeled "I of R or O") changes the state of a RID from "no entry" to "CE." (2) An insertion of a pointer corresponds to the transition labeled "I of P." (3) An update from regular to regular or from overflow to overflow corresponds to "U from R to R or from O to O," which the diagram shows twice (on the CR and CE states). (4) An update from pointer to pointer corresponds to "U from P to P." (5) An update from regular to pointer corresponds to "U from R to P," which the diagram shows twice (from CR and CE to P). (6) An update from pointer to regular corresponds to "U from P to R." (7) A deletion when the old RID appears in a CR or CE entry corresponds to "D," which the diagram shows from CR and CE to no entry. (8) A deletion when the old RID appears in a P entry corresponds to a third "D," which the diagram shows from P to no entry.

Phase 4: Sorting by new RID. After the phase for translation, we sort the pointers to the copies of log entries, using a major sort by new RID and a minor sort by LRSN. The major sort increases the locality of reference (and thus the speed) during later application. The minor sort preserves the order of operations on a slot of a RID.

Phase 5: Application. After the sorting by new RID, the set of pointers to log entries is scanned, and the log entries are applied to the new copy of the area being reorganized. For each new RID value in the set of prefixes, the log is applied by performing three activities:

In activity 1, all the pointers for the RID are found. They will be contiguous.

In activity 2, if at least one D log entry for the RID is found, certain log entries (and the pointers to them) are deleted. Specifically all the entries before the last D entry are deleted. Also the D entry itself is deleted if the first log entry was I; i.e., the slot is initially empty. The last D entry is kept if the first log entry was D or U; i.e., the slot is initially occupied. These deletions omit log entries for which the mapping table no longer contains the appropriate entries. These deletions also omit log entries whose effects would be nullified by a later D entry. For example, the sequence I U U D has no net effect, so the sequence should not be applied. If these deletions omit all the log entries for the RID, activity 3 is skipped.

In activity 3, if any log entries still exist, they are applied sequentially, as described next. For each affected page in the new copy, the LRSN in the page header is set to the current LRSN for the log.

The next three sections describe how to apply log entries for insertion, update, and deletion.

Application of a log entry for insertion. We begin with application of an insertion, for which the following procedure is performed: (1) The record is inserted in the new copy of the area, and its actual new RID is obtained. (2) In the mapping table, the CE entry whose SOURCE_RID matches the old RID of the log entry is found. (3) In that entry in the mapping table, the TYPE is changed from CE to CR, and the TARGET_RID is changed from its estimated value to the actual value. (4) In the prefixes for all the remaining log entries for the current RID, the RID is changed from the estimated value to the actual value.

For each new RID value, the log entries for that RID are applied before the log entries for any later new

RID value are applied. Therefore, if the actual new RID for an inserted record equals the estimated new RID for any inserted record, the equality is not a problem.

Application of a log entry for update. Now we turn to application of an update. The RID in the log entry identifies a regular record or a pointer record. The behavior of the log application (in the new copy) resembles the behavior of DBMS processing of a user's update of a row, as in the five Update rows of Table 2. For example, if there is no overflow, but the new data are too large for the page that contains the record, the regular record is updated to become a pointer, and an overflow record is inserted. In this case, the RID of the record that changed from regular to pointer continues to occupy the TARGET_RID column in the mapping table.

Application of a log entry for deletion. Finally, we discuss application of a deletion. The RID in the log entry identifies a regular record or a pointer record. The behavior of the log application (in the new copy) resembles the behavior of DBMS processing of a user's deletion of a row, as in the two Deletion rows of Table 2. If there is no overflow, the regular record is deleted. If there is overflow, the pointer record and the overflow record are deleted.

Control of iterations (step 3). The phases that we have just discussed take place in each iteration of log processing. We mentioned earlier that at the end of an iteration of step 3, we have criteria to choose between performing this step again and going to the next step. We next describe these criteria.

At the end of an iteration, the estimated time for the next iteration is compared to a parameter of the reorganization command. The parameter specifies the maximum desired time (for an iteration) during which users' access to the area can be read-only. If the estimated time is within the maximum, then step 4 is performed; step 5 is a final step of log processing, during which users have read-only access.

If, instead, the estimated time exceeds the maximum, the amount of the log that the next iteration would process is checked as to whether it is sufficiently lower than the amount that this iteration processed. Here, "sufficiently lower" means lower than a certain fraction of this amount of iteration. If the amount of the next iteration is sufficiently lower, then reorganizer reading of the log is "catching up" quickly enough to DBMS appending to the log. Therefore, the reorganizer iterates (perform step 3 again).

If, instead, the amount of the next iteration is not sufficiently lower, then the reading is not catching up quickly enough. An implementation (or another parameter of the reorganization command) can choose an action to solve this problem. Possible actions include aborting this invocation of reorganization, continuing with a higher maximum desired time for read-only access, continuing after quiescing writers, and continuing after increasing the priority of the reorganizer. A database administrator might increase the priority of the reorganizer via scheduling by the operating system or via allocation of buffers in main storage.

Additional information

Next we present additional information on a few aspects of our method. This information is not essential to the main flow of description of the method for reorganization.

Scheduling of on-line reorganization. The first topic of additional information involves scheduling. We have identified several criteria that a database administrator can use to decide when to schedule on-line reorganization.

Suppose that the rate of users' writing in the database, and thus the rate of the appending by the DBMS to the log, varies over time. Then scheduling on-line reorganization during a relatively slack period makes it easier for the reading of the log by the reorganizer to catch up to the appending to the log by the DBMS.

Suppose that the tolerance of delay varies from one application to another. Then scheduling on-line reorganization when no low-tolerance applications execute avoids irritation of users of those applications. For example, the tolerable delay for a batch program that produces a report at night can exceed the tolerable delay for interactive query processing during normal business hours.

The execution of a long-running user transaction can increase the time required to quiesce transactions. Therefore, scheduling on-line reorganization when no long-running transactions are expected to execute can speed on-line reorganization.

Omission of the mapping table if a key is unique. The second topic of additional information involves

an optimization. If the table that we reorganize has at least one unique key, an implementation might use one of the unique keys (instead of RIDs) to match log entries with records in the new copy of the area. This identification obviates the mapping table.

To permit this identification, the DBMS must include the value of that key in each log entry, even for operations that do not involve that key (e.g., update of a different column). Also, the DBMS must either (1) assure that the unique key is always unique in the old copy, even during the intermediate states of each user transaction, or (2) compensate for any temporary violation of uniqueness during the intermediate states.

During most of the reorganization, users can read and write the old copy.

Temporary violation of unique indexes. The third topic of additional information involves handling of unique indexes. During reloading and log processing, a temporary violation of unique indexes might occur in the new copy of the area, as in the following example: (1) An early page of the area contains a record whose unique key value is "Jones." (2) The reorganizer unloads the record for Jones. (3) A user deletes Jones, and a user then inserts Jones in a later page of the area. (4) The reorganizer unloads the second record for Jones. (5) The reorganizer reloads both records for Jones into the new copy, thus temporarily violating uniqueness. (6) The reorganizer eventually applies the log (using the RIDs) and deletes the first record for Jones, thus removing the violation of uniqueness.

In the old copy, the unique indexes enforce uniqueness. The set of values in the new copy will eventually equal the set of values in the old copy. Therefore, any such violation of uniqueness is temporary.

In IBM's DB2, a nonunique index but not a unique index contains a field (for each key value) for a count of RIDs that have the key value. Therefore, we could not solve this problem by temporarily marking the unique indexes in the new copy as nonunique.

Our solution involves these actions: (1) The data are always inserted into the new copy. (2) We try to insert entries into the unique indexes. If an insertion would violate uniqueness, the RID is instead saved in a special list. The list is probably short, since violation is probably rare. Application of the log can cause modification of this list. (3) At the end of the last step of processing of the log, index entries are inserted for the saved RIDs.

Reorganization of just an index. The last topic involves reorganization of just an index without reorganizing the corresponding data. Reorganizing an index on line closely resembles reorganizing data on line, and here we sketch on-line reorganization of an index: the old copy of the index is unloaded, a new copy is constructed, the log entries that deal with the index are processed, and access by users is switched to the new copy of the index. This reorganization changes no data and thus changes no data record RIDs, so it needs no mapping table.

Summary

We have described a method for performing a certain type of reorganization on line (concurrently with usage), so that a very large or highly available (24-hour) database need not go off line for reorganization. The reorganization includes restoration of clustering and removal of overflows. The method involves (1) copying data from the old copy of the area being reorganized to a new copy in reorganized form, (2) applying the log to the new copy, and (3) switching users' access to the new copy. During most of the reorganization, users can read and write the old copy.

This method for on-line reorganization uses the log, which identifies a record by its RID, which can change during reorganization. Therefore, the method includes maintenance of a table that maps between the old and new RIDs. The method uses this table to translate entries in the log before applying them to the new copy of the data. The novelty is in the interaction between processing of the log and maintenance of the mapping table.

In our discussion of this method and in our discussion of alternatives (in an appendix), we have identified design issues in several topics in on-line reorganization (in addition to the identification of

records). These topics include performance, creation of a backup copy, correction of nonclustering indexes during reorganization of a partition, and many other topics. These details are necessary for practical application of our method.

We believe that more research in on-line reorganization would be useful. One topic for such research is the control of iterations of log processing. Another topic is reduction (or even elimination) of the periods of read-only access and no access.

Finally, as the amount of information and dependence on computers both grow, the number of very large or highly available databases will grow, and with them the importance of on-line reorganization.

Acknowledgments

A suggestion from Robert Goldberg led to the first author's early work in on-line reorganization. We thank Chuck Bonner, Harry Campbell, Dick Crus, Greg Davoll, Dan Dionne, Craig Friske, John Garth, Rob Goldring, Kevin Gougherty, Thanh Ha, Don Haderle, Craig Heacock, Bala Iyer, Koshy John, Andy Johnson, Laura Kunioka-Weis, Angela Lee, Bob Lyle, Tom Majithia, Debbie Matamoros, Claus Mikkelsen, Roger Miller, Frank Mix, C. Mohan, Inderpal Narang, Mai Nguyen, Bill Niffenegger, Jim O'Toole, Paul Rao, Jim Ruddy, Dave Schwartz, Akira Shibamiya, Kalpana Shyam, Bryan Smith, Rob Sokohl, V. Srinivasan, Steve Strawbridge, Judy Tang, Jim Teng, Steve Turnbaugh, Michele Van Patten, Terry Walker, Julie Watts, Steve Watts, Roy Weise, Gordon Yamamoto, and Jay Yothers. These people discussed issues in reorganization or reviewed parts of an earlier draft of this paper. The editors and referees suggested several improvements.

Appendix A: Alternatives

In this section we discuss some alternatives to techniques that were discussed earlier. We explain the disadvantages of these alternatives.

Minor alternatives. We begin with minor alternatives. They apply to a few specific parts of our method for reorganization, and they still involve fuzzy reorganization.

Scanning for unloading (step 2). The first set of minor alternatives deals with the scanning for unloading data. We explained that to unload data, the table space is scanned, and the records are then sorted by clustering key. Here we discuss two possible alternative types of scanning.

Instead of scanning the table space, a possible alternative is to scan the most recent backup copy of the table space. An advantage of this alternative is elimination of reorganizer access and locking of the table space (which users access). Disadvantages include processing of additional log entries (all the entries since the backup) and access of tapes (if the backup copy resides on tapes).

Another alternative is to obviate the sorting activity by scanning the clustering index (which is already sorted by clustering key) and following the index RIDs to data in the table space. However, here we explain two reasons (involving correctness and performance) for our decision to scan the table space.

We explain the reason that involves correctness by an example. In scanning the clustering index, consider this sequence of events:

1. Initially, the slot for RID 3 contains a record whose clustering key value is "Jones."
2. The reorganizer reaches the "J" portion of the clustering index, and it copies the Jones record from RID 3 in the old copy to RID 17 in the new. Now in the new copy, RID 17 contains "Jones."
3. A user updates the record, changing "Jones" to "Smith"; thus a log entry indicates an update of RID 3 from "Jones" to "Smith." Alternatively, users might delete Jones and insert Smith, and the DBMS might use the former RID of Jones for Smith.
4. The reorganizer reaches the "S" portion of the clustering index, and it copies the Smith record from RID 3 in the old copy to RID 90 in the new. Now, in the new copy, RID 17 contains "Jones," and RID 90 contains "Smith." The mapping table might now have two entries for old RID 3.
5. At the end of loading, we apply the log to the new copy. Specifically, we find that RID 3 changed from "Jones" to "Smith," and we change RID 17 in the new copy from "Jones" to "Smith." Now, in the new copy, two RIDs (17 and 90) contain "Smith." This is incorrect; there should be only one record for Smith. We would need some special handling to prevent this error. We should delete (not update) the record for Jones in the new copy.

The second reason involves performance. We probably would reorganize an area only if the clustering in the area has degraded. Studies have found that

for off-line reorganization of such an area, scanning the table space and sorting are usually faster than scanning the clustering index. The difference in speed arises because scanning the index for such an area can result in many jumps between file pages.

Checking of log entries during translation (steps 3 and 5). Next we discuss translation of log entries. Within a step of log processing, the phase for translation checks for existence of an entry in the mapping table (for an update or deletion). For an insertion, the phase checks for nonexistence of an entry. This phase also compares the LRSN of the log entry to the LRSN of the record in the mapping table (for an update or deletion). The phase also checks the mapping table for appropriateness of logged updates; e.g., updating a record from overflow to overflow is inappropriate if the record is a pointer.

For an update, the LRSN comparison prevents unnecessary (but harmless) application of an update that occurred before the reorganizer scanned the record. The LRSN comparison also effectively checks for appropriateness, as we explained when we discussed log application as part of recovery. Thus the explicit checking for appropriateness is redundant with the LRSN comparison.

For a deletion, the LRSN comparison prevents unnecessary (but harmless) application of a pair of a deletion and an insertion that both occurred before the scanning.

This checking involves some redundancy and prevents some harmless applications. Several other methods for checking are possible. All the methods include checking for existence or nonexistence in the mapping table.

In the method that we have described, we compare the LRSNs, and we also check for appropriateness of updates. The LRSN comparison can result in deletion of the copy of the log entry and the pointer to the copy. The appropriateness checking can result in announcement of an error via abnormal termination of reorganization; any log entry with an appropriate LRSN should not contain an inappropriate update.

In a second method, we compare the LRSNs, and we also check for appropriateness of updates, as in the first method. Here, however, either type of checking can result in deletion of the copy of the log entry and the pointer to the copy. The disadvantage of this method is its failure to announce an error upon finding an inappropriate update.

In a third method, we compare the LRSNs, but we do not check for appropriateness of updates. This method, like the second, fails to announce an error.

Finally, in a fourth method, we omit the LRSNs from the mapping table, thus saving space and omitting the LRSN comparison. The appropriateness checking for updates can result in deletion of the copy of the log entry and the pointer to the copy. Here, however, log processing might unnecessarily but harmlessly apply an update or a pair of a deletion and an insertion. Again, this method fails to announce an error.

Correction of nonclustering indexes. Now we turn to correction of nonclustering indexes (for reorganization of a partition), for which we have described a method. In a possible alternative method for this correction, during the reloading in step 2, we construct a copy of all (not just a subset) of each nonclustering index. In steps 3 and 5, for each nonclustering index, when we apply a log entry to data for this partition, we make the corresponding change to the new copy of the index, using a translated log entry. When we find a log entry for data for another partition, we make the corresponding change to the new copy of the index, using an untranslated log entry. In step 7, we quiesce all access to the nonclustering indexes (even for the other partitions), and we rename the files that underlie the indexes. This renaming effectively replaces the old copies of the indexes by the new copies, thus correcting all the RIDs at once. In step 9, we erase the old copies of nonclustering indexes.

The advantage of this alternative is a great reduction in the time in step 7; we avoid an expensive operation of changing individual RIDs in nonclustering indexes. A disadvantage of the alternative is its added quiescing of all access to all the nonclustering indexes (even for the other partitions), although those indexes are off-line only briefly. Another disadvantage is prevention of concurrent reorganizations of two or more partitions.

Creation of a backup copy. The final set of minor alternatives deals with creation of a backup copy of the new copy of the area being reorganized, as a basis for future recoverability. We mentioned that we create the backup copy during step 2 (while users have full access to the old copy of the area). We bring

the backup copy up to date during the last iteration of step 3 (while users still have full access) and during step 5 (while users have read-only access). We also mentioned an alternative technique that takes place during step 8. Here we discuss more alternative techniques and their disadvantages.

One alternative, again during step 8, is to start read-only access, create a backup copy while allowing read-only access, and then start read/write access (after the backup copying completes). The entire period of creation restricts users to read-only access.

Another alternative is to create a backup copy during the reloading in step 2, append translated log entries to the original log in steps 3 and 5, and start read/write access immediately in step 8. Such a backup copy represents a less recent time and thus would require more log application in a recovery. The appending of translated log entries would also complicate the log.

Major alternatives. Now we turn from minor alternatives to major alternatives, which do not involve fuzzy reorganization.

On-line reorganization in place. The first major alternative is reorganization in place (instead of reorganization by copying). There are disadvantages of reorganization by copying: (1) It can require more disk space for the area being reorganized. (2) It involves a transition between directing users' accesses into the old copy and directing them into the new copy. (3) It benefits users (by giving them access to a reorganized copy) only after the transition. Reorganization in place might begin to benefit users immediately (as we begin to reorganize the area that users access).

However, if users release some locks before commitment of a transaction, one disadvantage of reorganization in place is complexity. Reorganization in place is especially complex if reorganization includes changing the assignment of records to pages, as in restoration of clustering. If, instead of releasing some locks early, we use page-level locks and hold them until commitment, we avoid the complexity,[45,46] but this latter style of locking can dramatically reduce concurrency and thus throughput.

Another disadvantage of reorganization in place is that it might cause more contention with users and thus degradation of users' performance, since it writes (instead of just reading) the copy that users access.

Off-line reorganization of fine-grained partitions. The second major alternative to our on-line reorganization is off-line reorganization of fine-grained partitions. A partition of a table space can be a unit of off-line reorganization or other utilities, during usage or off-line reorganization of other partitions, as in DATABASE 2.[2] Use of a fine granularity of partitioning can reduce the time to reorganize a partition off line, and use of a fine enough granularity might even reduce the time to approximate 24-hour availability. However, a limitation of making the granularity fine is an increase in the probability that areas of growth and areas of shrinkage will be in different partitions. This increase in the probability increases the likely variation (among the partitions) in growth rates, thus increasing the total recommended amount of free space to reserve in the database. Also, off-line reorganization (like some methods for on-line reorganization) has a prerequisite period of quiescing of users' activities, and it requires correction of nonclustering indexes. Finally, in some DBMSs, making the granularity fine can slow the routing of users' accesses into partitions and increase the total space required for partition storage descriptors.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Tandem Computers Incorporated.

Cited references

1. G. H. Sockut and R. P. Goldberg, "Database Reorganization—Principles and Practice," *Computing Surveys* **11**, No. 4, 371–395 (December 1979).
2. C. Mohan, "A Survey of DBMS Research Issues in Supporting Very Large Tables," D. Lomet, Editor, *Foundations of Data Organization and Algorithms (Proceedings of the 4th International Conference, Foundations of Data Organization and Algorithms)*, Springer-Verlag, New York (October 1993), pp. 279–300.
3. G. Wiederhold, *Database Design*, McGraw-Hill, Inc., New York (1983).
4. R. F. Schubert, "Directions in Data Base Management Technology," *Datamation* **20**, No. 9, 48–51 (September 1974).
5. H. S. Meltzer, "An Overview of the Administration of Data Bases," *Proceedings of the 2nd USA-Japan Computer Conference*, AFIPS Press, Reston, VA (August 1975), pp. 365–370.
6. H. S. Meltzer, "Structure and Redundancy in the Conceptual Schema in the Administration of Very Large Data Bases," P. C. Lockemann and E. J. Neuhold, Editors, *Systems for Large Data Bases (Proceedings of the 2nd International Conference on Very Large Data Bases)*, North-Holland Publishing Co., Amsterdam, Netherlands (September 1976), pp. 13–25.

7. A. Silberschatz, M. Stonebraker, and J. Ullman, "Database Systems: Achievements and Opportunities," *Communications of the ACM* **34**, No. 10, 110–120 (October 1991).
8. J. Keyes, "DBAs Face Challenge of 24 by 7 Availability," *Software Magazine* **12**, No. 11, 58–63 (August 1992).
9. D. J. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," *Communications of the ACM* **35**, No. 6, 85–98 (June 1992).
10. P. Valduriez, "Parallel Database Systems: Open Problems and New Issues," *Distributed and Parallel Databases* **1**, No. 2, 137–165 (April 1993), Kluwer Academic Publishers, Boston, MA.
11. G. Graefe, "Query Evaluation Techniques for Large Databases," *Computing Surveys* **25**, No. 2, 73–170 (June 1993).
12. P. G. Selinger, "Predictions and Challenges for Database Systems in the Year 2000," *Proceedings of the 19th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, Palo Alto, CA (August 1993), pp. 667–675.
13. D. J. Haderle and R. D. Jackson, "IBM Database 2 Overview," *IBM Systems Journal* **23**, No. 2, 112–125 (1984).
14. C. J. Date and C. J. White, *A Guide to DB2*, 4th edition, Addison-Wesley Publishing Co., Reading, MA (1993).
15. M. M. Astrahan et al., "System R: Relational Approach to Database Management," *ACM Transactions on Database Systems* **1**, No. 2, 97–137 (June 1976).
16. *Database Language SQL*, X3.135-1992, American National Standards Institute, New York, NY (1992).
17. J. Gray, "Notes on Data Base Operating Systems," R. Bayer et al., Editors, *Operating Systems—An Advanced Course*, Springer-Verlag, New York, NY (1978), pp. 393–481.
18. T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *Computing Surveys* **15**, No. 4, 287–317 (December 1983).
19. R. A. Crus, "Data Recovery in IBM Database 2," *IBM Systems Journal* **23**, No. 2, 178–188 (1984).
20. *IMS/VS—Version 1, Utilities Reference Manual*, SH20-9029-9, IBM Corporation (June 1986); available through IBM branch offices.
21. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions on Database Systems* **17**, No. 1, 94–162 (March 1992).
22. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Palo Alto, CA (1993).
23. C. Mohan and I. Narang, "An Efficient and Flexible Method for Archiving a Data Base," *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (June 1993), pp. 139–146.
24. D. J. Rosenkrantz, "Dynamic Database Dumping," *Proceedings of the ACM SIGMOD International Conference on Management of Data* (May 1978), pp. 3–8.
25. *Implementing Concurrent Copy*, GG24-3990-00, IBM Corporation (December 1993); available through IBM branch offices.
26. G. H. Sockut and B. R. Iyer, *Reorganizing Databases Concurrently with Usage: A Survey*, Technical Report 03.488, IBM Santa Teresa Laboratory, San Jose, CA (June 1993).
27. G. H. Sockut and B. R. Iyer, "A Survey of Online Reorganization in IBM Products and Research," *Data Engineering*, **19**, No. 2, 4–11 (June 1996).
28. T. B. Wilson, "Data Base Restructuring: Options and Obstacles," P. A. Samet, Editor, *EURO IFIP 79 (Proceedings of the European Conference on Applied Information Technology)*, North-Holland Publishing Co., Amsterdam, Netherlands (September 1979), pp. 567–573.
29. M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The Design of XPRS," *Proceedings of the 14th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, Palo Alto, CA (August 1988), pp. 318–330.
30. C. Mohan and I. Narang, "Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates," *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (June 1992), pp. 361–370.
31. V. Srinivasan and M. J. Carey, *On-Line Index Construction Algorithms*, Technical Report 1008, Computer Sciences Department, University of Wisconsin, Madison, WI (March 1991); presented at the 4th International Workshop on High Performance Transaction Systems (September 1991). For more details, see Reference 32.
32. V. Srinivasan, *On-Line Processing in Large-Scale Transaction Systems*, Ph.D. thesis, Computer Sciences Department, University of Wisconsin, Madison, WI (January 1992).
33. V. Srinivasan and M. J. Carey, "Performance of On-Line Index Construction Algorithms," A. Pirotte et al., Editors, *Advances in Database Technology—EDBT '92 (Proceedings of the 3rd International Conference on Extending Database Technology)*, Springer-Verlag, New York (March 1992), pp. 293–309; more details appear in Technical Report 1047, Computer Sciences Department, University of Wisconsin, Madison, WI (September 1991). For more details, see Reference 32.
34. V. Srinivasan and M. J. Carey, "Compensation-Based On-Line Query Processing," *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (June 1992), pp. 331–340. For more details, see Reference 32.
35. M. Stonebraker, "Hypothetical Data Bases as Views," *Proceedings of the ACM-SIGMOD 1981 International Conference on Management of Data* (April 1981), pp. 224–229.
36. *An Introduction to DataPropagator Relational Release 1*, GC26-3398-01, IBM Corporation (1993); available through IBM branch offices.
37. *Replidata/MVS User's Guide*, BLD-REP-UG-00, IBM Corporation and Integrated Systems Solutions Corp. (January 1994).
38. J. Troisi, "NonStop Availability and Database Configuration Operations," *Tandem Systems Review* **10**, No. 3, 18–23 (July 1994).
39. S. Englert, "NonStop SQL: Scalability and Availability for Decision Support," *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (May 1994), p. 491.
40. J. Cohen, "Garbage Collection of Linked Data Structures," *Computing Surveys* **13**, No. 3, 341–367 (September 1981).
41. J. O'Toole, S. Nettles, and D. Gifford, "Concurrent Compacting Garbage Collection of a Persistent Heap," *Proceedings of the 14th ACM Symposium, Operating Systems Principles* (December 1993), pp. 161–174.
42. J. L. Wiener and J. F. Naughton, "Bulk Loading into an OODB: A Performance Study," *Proceedings of the 20th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, Palo Alto, CA (September 1994), pp. 120–131.
43. J. L. Wiener and J. F. Naughton, "OODB Bulk Loading Revisited: The Partitioned-List Approach," *Proceedings of the 21st International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, Palo Alto, CA (September 1995), pp. 30–41.
44. F. Mix, "DB2 Reorg and Continuous Select," *Proceedings of*

the 6th Annual North American Conference, International DB2 Users Group, IDUG, Chicago, IL (May 10, 1994), pp. 545–563.
45. E. Omiecinski, L. Lee, and P. Scheuermann, "Concurrent File Reorganization for Record Clustering: A Performance Study," *Proceedings of the 8th International Conference on Data Engineering, IEEE-CS* (February 1992), pp. 265–272. For more details, see Reference 46.
46. E. Omiecinski, L. Lee, and P. Scheuermann, "Performance Analysis of a Concurrent File Reorganization Algorithm for Record Clustering," *IEEE Transactions on Knowledge and Data Engineering* **6**, No. 2, 248–257 (April 1994).

Accepted for publication March 26, 1997.

Gary H. Sockut *IBM Santa Teresa Laboratory, P.O. Box 49023, San Jose, California 95161-9023 (electronic mail: ghs@vnet. ibm. com).* Dr. Sockut is an advisory programmer at the IBM Santa Teresa Laboratory. He has been a member of the team that developed on-line reorganization for DB2 for OS/390. Previously he was with BGS Systems, the National Institute of Standards and Technology, and the IBM T. J. Watson Research Center. He received an Sc.B. (magna cum laude) in applied mathematics from Brown University, an S.M. in electrical engineering from the Massachusetts Institute of Technology, and a Ph.D. in applied mathematics from Harvard University. His main areas of interest are database management (especially reorganization and graphical languages), office systems, and operating systems. He has written 25 papers. He is a member of ACM and the IEEE Computer Society, and he has been a member of Sigma Xi.

Thomas A. Beavin *IBM Santa Teresa Laboratory, P.O. Box 49023, San Jose, California 95161-9023 (electronic mail: beavin@vnet.ibm.com).* Mr. Beavin is an advisory programmer at the IBM Santa Teresa Laboratory. He is currently a member of the DB2 development organization and has worked in that department and in the OS/390 development organization for the past 11 years. His focus is on utility development, including the design of on-line reorganization, and more recently on query optimization. Mr. Beavin received a B.S. in applied computer science from Illinois State University in 1985.

Chung-C. Chang *Oracle Corporation, Redwood Shores, California 94065 (electronic mail: ccchang@us.oracle.com).* Mr. Chang is now a principal consultant of Oracle Corp. He worked with the DB2 development team at the IBM Santa Teresa Laboratory for more than 12 years, and was a member of the team that developed on-line reorganization for DB2 for OS/390. Previously he was with BLI (Britton-Lee, Inc.) for the development of intelligent database machines. He received a B.S. in mathematics from National Taiwan University and an M.S. in computer science from Wayne State University. His main areas of interest are operating systems, database management, and network computing architectures.

Reprint Order No. G321-5651.