Detection and Recovery Techniques for Database Corruption

Philip Bohannon, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, *Fellow*, *IEEE*, and S. Sudarshan

Abstract—Increasingly, for extensibility and performance, special purpose application code is being integrated with database system code. Such application code has direct access to database system buffers, and as a result, the danger of data being corrupted due to inadvertent application writes is increased. Previously proposed hardware techniques to protect from corruption require system calls, and their performance depends on details of the hardware architecture. We investigate an alternative approach which uses codewords associated with regions of data to detect corruption and to prevent corrupted data from being used by subsequent transactions. We develop several such techniques which vary in the level of protection, space overhead, performance, and impact on concurrency. These techniques are implemented in the Dalí main-memory storage manager, and the performance impact of each on normal processing is evaluated. Novel techniques are developed to recover when a transaction has read corrupted data caused by a bad write and gone on to write other data in the database. These techniques use limited and relatively low-cost logging of transaction reads to trace the corruption and may also prove useful when resolving problems caused by incorrect data entry and other logical errors.

Index Terms—Database corruption, database recovery, fault tolerance, data integrity, main-memory database.

1 Introduction

As hardware gets more reliable, software errors[1] are often the greatest threat to database system availability [10], [24], even in standard database systems where database data is protected by process boundaries from errors in application programs. Increasingly, however, for extensibility and performance, special purpose application code is being integrated with database system code. Extensible databases allow third party vendors and users to add new data types and storage methods to the database engine [22]. Performance-critical applications may require the performance which can be achieved by directly accessing the data stored in a main-memory database [3], [11]. In either case, due to the high cost of interprocess communication, direct access to database internal structures such as the buffer cache is critical to meeting the performance needs of these applications. Thus, database availability can be affected not only by software errors in the DBMS, but also by errors in application programs. One class of software error which has been shown to have a significant impact on DBMS availability is the "addressing" error [24]. This class of error includes copy overruns and "wild writes" through uninitialized pointers.

Software fault tolerance techniques (see, for example, [18], [21]) attempt to mitigate the damage done when software errors occur in a production environment. One approach to avoiding addressing errors is the use of type-safe languages for user applications. Similar results can be achieved with the technique of *sandboxing* [28]. However, type safe languages have yet to be proven in high-performance situations, and sandboxing may perform poorly on certain architectures (see Section 7, Related Work, for more details). Finally, communication across process domain boundaries to a database server process provides protection, but such communication is orders of magnitude slower than access in the same process space, even with highly tuned implementations [1]. With multi-gigabyte main memories now easily affordable, one can expect many OLTP databases to be fully cached, decreasing the impact of disk latency on performance and, consequently, increasing the relative impact of interprocess communication.

In [23], Sullivan and Stonebraker investigate the use of hardware memory protection to improve software fault tolerance in a DBMS environment by guarding data in the buffer cache. For example, calls were added to POSTGRES [23] to unprotect the page containing a tuple before it is updated and to reprotect it afterwards. In performance experiments, they found that this protection was relatively inexpensive. The overhead amounted to 7 to 11 percent of the processing time using a CPU bound workload which ignored disk latency, and 2 to 3 percent of processing time when disk latency was included. However, a number of factors have motivated us to consider other possible techniques for protecting data in the DBMS. First, memory protection primitives must be accessed through system calls which may be slow [21]. An informal test of several systems available to us confirmed that the performance of mprotect can vary widely among comparable workstations (see

1. An *error* occurs in a system when its behavior deviates from the behaviors allowed by its specification.

- *P. Bohannon and R. Rastogi are with Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ, 07974. E-Mail: {bohannon, rastogi}@research.bell-labs.com.*
- *S. Seshadri is with Strand Genomics, 237, Sir C.V. Raman Avenue, Rajmahal Vilas, Bangalore 560 080, Karnataka, India. E-mail: seshadri@strandgenomics.com.*
- *A. Silberschatz is with the Department of Computer Science, Yale University, PO Box 208285, New Haven, Connecticut 06520-8285. E-mail: avi@cs.yale.edu.*
- *S. Sudarshan is with the Indian Institute of Technology, Bombay, India. E-mail: sudarsha@cse.iitb.ernet.in.*

Manuscript received 14 Sept. 1999; revised 25 Jan. 2001; accepted 14 Feb. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 114115.

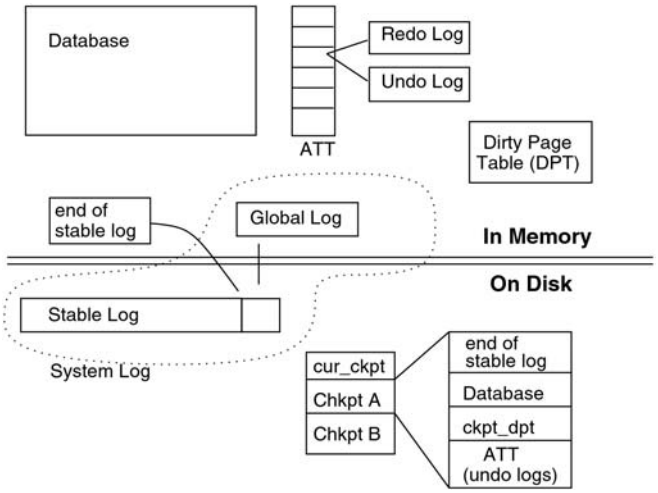

Fig. 1. Recovery Data Structures.

Fig. 1). Second, in a threaded application, threads can access pages unprotected by other threads, decreasing the effectiveness of the scheme. Finally, in a main-memory DBMS, information such as allocation information or other control information need not be stored on the same page as user data. This is the case in the Dalí system [4], [12] in which our experiments are implemented, and it can significantly increase the number of pages which must be protected and unprotected during a single operation.

In this paper, we address certain software errors which affect the database state. In particular, we refer to bytes modified in the course of an addressing error as *direct physical corruption*. Once data is directly corrupted, it may be read by a process, which then issues writes based on the value read. Data written in this manner is *indirectly corrupted*, and the process involved is said to have *carried* the corruption. While this process could be a database maintenance process, we focus on *transaction-carried corruption*, in which the carrying process is an executing transaction. As noted in [23], for a DBMS to effectively guard data from direct physical corruption, it must expose an *update model* by which correct updates can be distinguished from unintended or erroneous updates. In our update model, all updates are in-place, and correct updates are ones which use a prescribed interface.

We investigate techniques which protect data by dividing the database into *protection regions* and associating a *codeword* with each such region. Thus, using the prescribed interface ensures that when data in a region is updated, the codeword associated with the region is also updated. (This activity is referred to as "codeword maintenance.") When a wild write or other addressing error updates data with high probability, the codeword value computed from the region will no longer match the codeword maintained for that region.

We present several codeword-based techniques for the prevention or detection of corruption. The first scheme we describe, Read Prechecking, prevents transaction-carried corruption by verifying that the codeword matches the data each time it is read. The Data Codeword scheme, a less expensive variant of Read Prechecking, allows detection of direct physical corruption by asynchronously auditing the codewords. A variant of this scheme, Data Codewords with Deferred Maintenance, makes use of the database log to perform the updates to the codewords, reducing contention which may be caused by updating or auditing codewords.

For detecting indirect corruption, we introduce the Read Logging scheme in which a limited amount of information about each data item read by a transaction is added to the log. Since the data logged consists of the identity of the item and an optional checksum of the value, *but not the value itself*, the performance impact of this logging is limited. The addition of information about reads allows the database log to function as a limited form of audit trail [2] for the DBMS.

Since none of these techniques prevents direct physical corruption, techniques for *corruption recovery* must be employed to restore the database to an uncorrupted state. We introduce the *delete-transaction model*, a model of recovery which focuses on removing the effects of corruption from the database image, and present an algorithm which implements this model as a modification of the Dalí recovery algorithm. Recovery from errors when the error is not immediately detected (for example, not detected until after the transaction commits) is discussed in [7], but the delete-transaction model algorithm presented in this paper is the first concrete proposal we are aware of for defining and implementing corruption recovery in a transaction processing system.

We describe how techniques based on Read Logging can be extended in certain cases to *logical corruption*, which we define as any error introduced through data entry or incorrectly-coded transaction programs. We extend recovery in the delete transaction model to logical corruption, introduce the *redo-transaction* model of recovery, and present an algorithm for efficient recovery in this model when the underlying database uses logical redo logging. The redo-transaction recovery model includes the assumption that the submission of transactions does not depend on a current or past database state. Such transactions are referred to as *independent*.

To ascertain the performance of our algorithms for detecting and recovering from physical corruption, we studied the impact of these schemes on a TPC-B style workload implemented in the Dalí main-memory storage manager. Our goal was to evaluate the relative impact of the schemes described above on normal transaction processing. In addition to our schemes, we include a hardware-based protection technique similar to that of [23]. For detection of direct corruption, the overheads imposed cause throughput of update transactions to be decreased by 6 to 8 percent. Prevention of transaction-carried corruption with Read Prechecking costs between 12 and 72 percent, with the space overheads increasing as performance improves. Detection of transaction-carried corruption with Read Logging costs between 17 and 22 percent. Using hardware protection decreases throughput in Dalí by about 38 percent, even on a platform with relatively fast protection primitives.

The remainder of the paper is organized as follows: Our system model is discussed in Section 2. Section 3 describes schemes which prevent or detect corruption. Section 4 presents techniques for recovery when corruption is

detected after the fact. Extensions to deal with logical corruption are presented in Section 5. Section 6 describes a performance study of several of the algorithms presented in the paper, and Section 7 describes related work. Section 8 concludes the paper and discusses future work.

2 System Model

This paper assumes a database model in which database data is directly mapped into the address space of applications. In addition to user data, control information such as lock tables and log buffers may also be mapped into the address space of the application. Note that, even if user applications are not allowed direct access to database data, any multithreaded database engine will follow a similar model in that database server processes will access the buffer cache in shared memory [9].

The reliability of the algorithms we develop depend on details of the logging and recovery model of the DBMS and, in these cases, the logging and recovery model of the Dalí main-memory storage manager is used. The logging and checkpointing code in Dalí is modified to implement the protection schemes for the performance study in Section 6. Updates in Dalí are done in-place, and updates by a transaction must be bracketed by calls to the functions beginUpdate and endUpdate. Each physical update to a database region generates an undo image and a redo image for use in transaction rollback and crash recovery. Undo and redo logs in Dalí are stored on a per-transaction basis (*local logging*). When a lower-level operation is committed, the redo log records are moved from the local redo log to the system log tail in memory, and the undo information for that operation is replaced with a logical undo record. Both steps take place prior to the release of lower-level locks. A copy of the logical undo description is included in the operation commit log record for use in restart recovery.

As a main-memory system, Dalí is only page-based to the extent that it is convenient for tracking storage use and for the efficient layout of fixed-size records. For example, allocation information is not stored on the same page as tuple data, and extra free-space for expansion need not be reserved on each page. Benefits of this approach include efficient use of space and the ability to store objects larger than a page contiguously and, thus, access them directly without reassembly and copying.

Our corruption detection and recovery algorithms are expressed in terms of the Dalí recovery algorithm. The Dalí recovery algorithm provides very general support for high concurrency, multilevel operations, and minimal interference with transaction processing in a main-memory database system. We now outline those features of the Dalí recovery algorithm which are important for the algorithms in this paper. More details can be found in [3], [4].

Multilevel Recovery. Dalí implements a main-memory version of *multilevel recovery* [16]. A multilevel transaction processing system consists of n logical *levels of abstraction*, with operations at each level invoking operations at lower levels. Transactions themselves are modeled as operations at level n, with level 0 consisting of physical updates. Multilevel recovery permits some locks to be released early to increase concurrency. For example, many systems have two levels defined below the transaction level: a tuple level and a page level. Locks on pages are released early, but locks on tuples are held for transaction duration.

Data Structures. Fig. 1 illustrates some of the data structures central to transaction management in Dalí. The three main structures are the active transaction table, or ATT, the system log, and checkpoint images. Undo and redo logs in Dalí are stored on a per-transaction basis in the ATT (*local logging*), as shown in the figure. The portion of the log known to be stable on disk is indicated by the end_of_stable_log variable. A bitmap of dirty pages is maintained, the dpt, and used in checkpointing. Additional details are given below.

Undo and Redo Logging. Updates in Dalí are done in-place, and updates by a transaction must be bracketed by calls to the functions beginUpdate and endUpdate. Each physical update to a database region generates an undo image and a redo image for use in transaction abort and crash recovery. These images are referred to as *physical undo* and *physical redo* information, respectively. When a lower-level operation is committed, the redo log records are moved from the local redo log of that transaction to the system log tail in memory, and the physical undo information for the committing operation is replaced with a *logical undo* record. Both steps take place prior to the release of lower-level locks. A copy of the logical undo description is included in the operation commit log record for use in restart recovery.

Log Flush. The contents of the system log tail are *flushed* to the stable system log on disk when a transaction commits, or during a checkpoint. The *system log latch* must be obtained before performing a flush, to prevent concurrent access to the flush buffers. The stable system log and the tail are together called the *system log*. As already mentioned, the variable end_of_stable_log stores a pointer into the system log such that all records prior to the pointer are known to have been flushed to the stable system log. While flushing physical log records, we also note which pages were written ("dirtied") by the log record. This information about dirty pages is noted in the *dirty page table* (dpt).

Logical Undo. All redo actions are physical, but when an operation commits, an operation commit log record is added to the redo log, containing the logical undo description for that operation. These records are used so that, at system recovery, logical undo information is available for all committed operations whose enclosing operation (which may be the transaction itself) has not committed. For transaction rollback during normal execution, the corresponding undo records in the transaction's local undo log are used instead. Note that a transaction's local undo log always consists of some number of logical undo actions followed by some number of physical undo actions since physical operations not part of the current operation would have already been replaced by a logical undo record.

Transaction Rollback. When a transaction is rolled back during normal operation, its local undo log is traversed in reverse order and the undo descriptions are used to undo operations of the transaction. In the case of physical update operations, the undo is done physically, while in the case of

logical operations, the undo is done by executing the undo operation. Following the philosophy of *repeating history* [17], both these actions generate redo logs representing the physical updates taken on their behalf. Additionally, each undo operation also generates redo log records to note the begin and commit of the undo operation, just like a regular operation. At the end of rollback, a transaction abort record is written out.

Checkpoints. Since the database is assumed to fit in main-memory, Dalí does not have a buffer manager and does not write pages back to disk except during a *checkpoint* operation. During a checkpoint, dirty pages from the in-memory database image are written to disk. In fact, two checkpoint images Ckpt_A and Ckpt_B are stored on disk, as is the checkpoint anchor, cur_ckpt, which points to the most recent valid checkpoint image for the database. During subsequent checkpoints, the newly dirty portions of the database are written alternately to the two checkpoint images (this is called ping-pong checkpointing [7]). The anchor is switched to point to the new checkpoint only after all actions associated with the checkpoint have successfully completed. In addition to the database image, a copy of the ATT with the local undo logs (but not the local redo logs), a value for the end of the stable log prior to checkpoint start (ckpt_eol), and a copy of the dirty page table (dpt) are stored with each checkpoint.

Note that physical undo information is moved to disk only during a checkpoint. The undo information is taken by the checkpointer directly from the local undo logs of each transaction. (Thus, physical undo log records are never written to disk for transactions which take place between checkpoints.)

Restart Recovery. Restart recovery starts from the last completed checkpoint image and replays all redo logs (repeating history), starting with the end_of_stable_log noted with the checkpoint. When the end of log is reached, incomplete transactions (those without transaction-level commit or abort records) are rolled back, using the logical undo information stored in either the ATT or operation commit log records. Undo information is available in the redo log for all operations at level 1 and higher. For level 0 (physical updates), the undo information is available in the checkpointed undo log. Due to multilevel recovery, the rollback is done level by level, with all incomplete operations at level i being rolled back before any incomplete actions at level $i+1$ are rolled back. Finally, the Dalí recovery algorithm *repeats history* [17] on a physical level, so, during undo processing, redo records are generated as for forward processing and these redo records are used to ensure that logical undo actions are only taken once in the face of multiple failures. For more details, please see [3], [4].

3 Codeword Protection

In this section, we introduce three codeword-based schemes for detection and prevention of corruption: Read Prechecking, Data Codeword, and Data Codeword with Deferred Maintenance. We also discuss the Hardware Protection scheme which is included in the performance study in Section 6. When Hardware Protection prevents an addressing error, a trap is issued to the process and the offending write is not completed. Thus, this scheme prevents direct physical corruption. By contrast, the codeword schemes can only detect direct physical corruption during a subsequent audit, and this is the strategy taken with the Data Codeword and Data Codeword with Deferred Maintenance schemes. However, direct corruption only does damage when the corrupted data is read by a subsequent process, and this indirect corruption can be prevented using the Read Prechecking scheme.

3.1 Codewords and Codeword Maintenance

Schemes for computing codewords for data are well known, and selection of a good scheme is not a topic addressed in this paper. However, we require certain properties to hold for the actual scheme chosen:

1. We require that the effect of an update M on the codeword can be summarized as $\Delta(M)$. The update M consists of the old and new values of the updated part of the protection region and, hence, $\Delta(M)$ must be a function of only the location and the old and new values of the updated area. In particular, $\Delta(M)$ should not depend on the old codeword. Since in one of our schemes $\Delta(M)$ must be stored in the redo log record, we require that the information in $\Delta(M)$ be small, only about as big as the codeword itself. We also assume there is a codeword update operation $\oplus$, such that, if the codeword for a region prior to an update M is C_{old}, the codeword after the update is C_{new}, where
$$C_{new} = C_{old} \oplus \Delta(M).$$

2. There may be a sequence of updates to a region whose Δs are applied out-of-order to the codeword. We assume that, while an update is in progress, it has exclusive access to the updated data; such exclusive access can be ensured by means of region locks. Further, the codeword change $\Delta(M)$ is computed, while the update M has exclusive access to the updated data. Given the above, we require that for any pair of updates M_1 and M_2
$$(C \oplus \Delta(M_1)) \oplus \Delta(M_2) = (C \oplus \Delta(M_2)) \oplus \Delta(M_1).$$

3. It must be possible to treat M as if it is composed of two updates, one from the initial value of the updated area U to a value of all zeros, and a second from all zeros to the value R it holds after M. We use $\Delta^-(U)$, which we call the *undo value*, to denote $\Delta(M')$, where M' is the update from U to all zeros, while $\Delta^+(R)$, which we call the *redo value* denotes $\Delta(M'')$, denotes where M'' is the update from all zeros to R.

   Thus, we have
$$C \oplus \Delta(M) = (C \oplus \Delta^-(U)) \oplus \Delta^+(R).$$
   We shall assume a function $\otimes$ that can be used to combine $\Delta^-(U)$ and $\Delta^+(R)$ such that

$$\Delta(M) = \Delta^{-}(U) \otimes \Delta^{+}(R).$$

There are several codeword schemes that satisfy our requirements. In our implementation of the schemes in this paper, the codeword is the bitwise exclusive-or of the words in the region. Thus, the ith bit of the codeword represents the parity of the ith bit of each word on the region. For the case of parity, Δ, $\oplus$, and $\otimes$ are all the same—they compute word-wise exclusive-or. It is easy to check that the parity code has the properties that we require. In particular, $\Delta(M)$ for an update M simply consists of the exclusive-or of the before-update and after-update values of words involved in the update. Also, $\Delta(M)$ requires only one word of storage.

3.2 Control Structures

Corruption can occur not only on the database image, but also on transient database control structures such as lock information, etc. Since the interfaces to such data are typically less uniform than the interface to user data, using the techniques described below to protect control structures would entail a significant individual implementation effort for each structure covered. For this reason, we do not include protection of these control structures in this study.

Protecting and unprotecting an entire segment was used to provide protection for control structures in [23]. While we are aware of no segment-level protection mechanisms in the UNIX platforms available to us, were such a mechanism available, it could easily be combined with the schemes for protection of persistent data studied in this thesis. However, even with such a mechanism, protection of control structures in [23] was found to cost approximately 10 times as much as protection of data buffers in CPU-bound tests.

3.3 Hardware Protection

While not a codeword scheme, the hardware protection scheme is included in our performance study as a point of comparison. Since all updates in Dalí are in-place, the hardware protection scheme we implemented most closely resembles the Expose Page Update Model of [23]. On a call to beginUpdate, the page (or possibly pages) being updated are unprotected, and are reprotected at the call to endUpdate.

3.4 Read Prechecking

An alternative to preventing direct corruption of data is preventing the use of that corrupted data by a transaction. To accomplish this, the consistency between the data in a protection region and its codeword is checked during each read of persistent data. We now present the details of this scheme. Some notes on adapting this to a disk-based recovery scheme follow in Section 3.4.1.

A protection latch is associated with each protection region and acquired exclusively when data is being updated, or when a reader needs to check the region against the codeword.

A shared protection latch is associated with each protection region so that, by holding the latch in exclusive mode, the reader can obtain an update consistent image of the region and the codeword. Updates to data in the region hold this latch in shared mode, so that updates to different portions of the region may go on concurrently. Each codeword has an associated codeword latch, which is used to serialize updates to the codeword. More than one codeword can share a single latch; the loss in concurrency is not much since codeword updates take very little time and, thus, the latch is acquired only for very short durations.

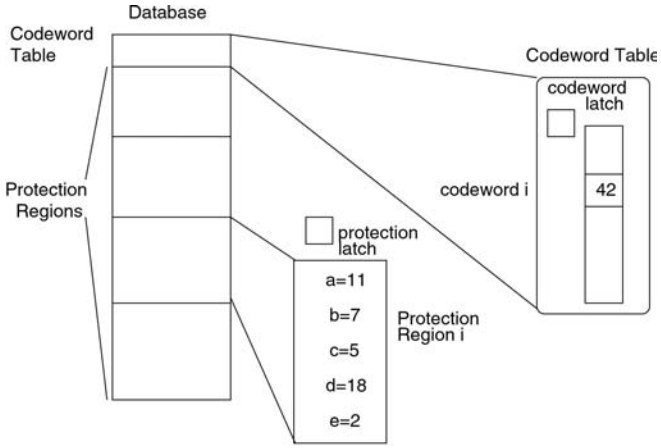

Fig. 2. Protection region and codeword table.

At endUpdate time, the undo image stored in the log and the current value of the updated region are used to update the codeword before the protection latch is released. A flag, codeword-applied, is stored in the undo log record for a physical update to indicate whether the associaed change to the codeword has been applied. If a rollback is necessary when the flag is set, the undo image for this update should be applied without updating the codeword. When reading data, the protection_latch is taken in exclusive mode and the codeword for the contents of the region which contains the data to be read is computed and compared with the stored codeword.[2]

The data structures associated with read prechecking are illustrated in Fig. 2. Database data items is indicated by $a, b, \ldots$, and this example shows a protection region with five data items, $a \ldots e$. For ease of exposition, in this and subsequent examples, the codeword operators $\oplus$ and $\otimes$ are both addition modulo 2^{32}. Thus, in Fig. 2, the codeword for the protection region is just the word-wise sum of a through e, the values in the region. Although not shown in the examples, data items may span multiple words, and need not begin or end on a word boundary.

We now present the actions taken during normal transaction processing.

- **Begin Update**. At the beginning of an update, the protection latch for the region is acquired, the undo image of the update is noted in the undo log, and the flag codeword-applied in the undo log record is set to false.
- **End Update**. When an update is completed, the redo image is recorded from the updated region and the undo image in the undo log record is used with this image to determine the change in the codeword. The

2. Note that, instead of obtaining the protection_latch in exclusive mode, which could be quite restrictive, readers can also obtain the protection_latch in a shared mode that is different from the shared mode used by updaters. That is, the shared mode used by readers is compatible with other readers, but conflicts with the shared mode for updaters.

codeword is updated with this change, and the flag codeword-applied in the undo log record for the update is set to true. Finally, the protection latch is released.

- **Undo Update**. When undo processing executes logical actions, the Dalí recovery algorithm generates redo log records for updates, just as during normal processing. Correspondingly, the codeword is changed to reflect each update, just as during forward processing. When undo processing encounters a physical undo log record, then it must be handled differently based on whether redo processing had taken place, as represented by the the flag codeword-applied in the undo log record. If this flag is set to true, the protection latch is acquired and the codeword for the protection region is modified to back out the update. If the flag is false, no change is made to the codeword, and the protection latch should not be acquired, as it is already held from forward processing. Regardless of the value of the flag, the other undo actions, such as applying the update and generating a log record for the undo, are executed. Finally, the protection latch is released.
- **Read**. To verify the integrity of the data in the protection region, the reader acquires the protection latch for the region, computes the codeword value of the region, and compares this with the stored codeword for that region.

In order to prevent corruption from reaching the disk image, the checkpointer performs the same integrity check as the reader. If either a reader or the checkpointer finds that a region does not match the codeword for that region, steps are taken as described in Section 4.3 to recover the data from this corruption.

3.4.1 Read Prechecking in ARIES

In this section, we describe how the Read Precheck scheme would need to be modified to work on a page-based architecture such as ARIES [18]. In ARIES or other page-based systems, the protection region may be chosen to be the page, in which case the page latch can be used as the protection latch. However, our performance study indicates that this may not lead to acceptable performance, thus it may be necessary to associate multiple codewords with a given page. The codewords and codeword-applied flag may be stored with the page itself (and excluded from codeword computations). When a page is propagated to disk, all regions on the page must be checked, as described above.

3.5 Data Codeword

Detecting (but not preventing) direct physical corruption can be accomplished with a variant of the Read Prechecking scheme described in Section 3.4. The maintenance of the codewords is accomplished in the same manner; however, the check of the codeword on each read is dropped in favor of periodic audits. The process of auditing is nothing more than an asynchronous check of consistency between the contents of a protection region and the codeword for that region. This can be carried out just as if a read of the region were taking place in the Read Prechecking scheme.

Since prechecks are not being performed, and audits are asynchronous, it makes sense to use significantly larger protection regions. In this case, the protection latch may become a concurrency bottleneck. If so, a new latch, the codeword latch, may be introduced to guard the update to the actual codewords, and the protection latch for a region need only be held in shared mode by updaters. During audit, the protection latch must be taken in exclusive mode to obtain a consistent image of the protection region and associated codeword. In particular, data is audited during the propagation to disk by the checkpointer (or at page-steal time in a page-based system).

3.6 Corruption Detection with Deferred Maintenance

An alternate approach for protecting data with codewords, the Data Codeword with Deferred Maintenance scheme is designed to reduce contention on the codeword table. This scheme stores codeword updates in log records and updates the codewords themselves during log flush rather than during the update of the data item. Updaters need not obtain any page latches, and checkpointing, codeword-auditing, and page-flushing do not interfere with updates. Since codeword updates are done during log flush, the system-log latch, which prevents concurrent flushes, serves to serialize access to the codewords. Thus, the Deferred Maintenance scheme can result in increased throughput for update transactions, particularly in main-memory databases, where the relative cost of latching can be substantial [9]. However, due to the greater cost of auditing a protection region, this scheme is not compatible with Read Prechecking.

For this scheme, each redo log record has an additional field, the redo delta, which stores $\Delta(M)$, the change in the codeword which would be caused by the update M, which generated the log record. Also, each undo record has a codeword-applied flag. We assume, for simplicity, that the area covered by a redo log record does not span two or more protection regions; the assumption can be easily ensured by splitting log records that span protection region boundaries.

3.6.1 Actions During Normal Processing

The actions taken at various steps are described below. To provide context, regular actions taken during these steps are included. Note that information is added to existing log records due to the corruption detection scheme, but all log records mentioned below are part of the standard Dalí recovery algorithm (see Section 2).

- **Begin Update**. Add an undo log record for the update to the local undo log of the transaction that performed the update and set codeword-applied in the undo log record to false.
- **End Update**. Create a redo log record for the update. Find the undo log record for the update and find the updated area from the undo log. Compute $\Delta(M)$ from the undo image in the undo log, and the new value of the updated area, and store it in the redo delta field in the redo log record. Add the redo log

record for the update to the redo log and set codeword-applied in the undo log record to true.

- **Undo Update**. For each physical update in the undo log, a proxy redo log record is generated whose redo image is the old value from the undo record. If the codeword-applied flag for the undo record is true, then a redo record has already been generated with $\Delta(M)$, so $\Delta(M^{-1})$ must be included in the proxy record to reverse the effect. $\Delta(M^{-1})$ is computed from the current contents of the region and the undo log record, and stored in the redo delta of the proxy log record.

  If the codeword-applied flag is false, then no redo log record has been generated and, thus, no codeword delta will be applied by the flusher. Since the codeword has not been changed and should not be changed, we just create the proxy record as usual, and store 0 (a special value that results in no change to the codeword) in its redo delta.

  In either case, we finish processing the physical undo log record by replacing the image in the database with the undo image in the log. Note that due to repeating history physically, executing a logical undo action during rollback causes physical updates; codeword processing for these updates is done as if for normal updates.
- **Flush**. When flushing a physical redo log record, the flusher applies redo delta from the log record to the codeword for that page. Note that the system log latch is held for the duration of the flush and that the variable end_of_stable_log is updated to reflect the amount of log which is known to have made it to disk.

3.6.2 Auditing

In the logging-based scheme, while the updater's task is simpler, the job of the auditor has become significantly more difficult. When it reads the page, it does so fuzzily, and partial updates may be captured. Log information must then be used to bring the page to an update-consistent state, so that it can be compared with a codeword value. This requires performing a small-scale version of the Dalí recovery algorithm at checkpoint time. To avoid the expense of executing this algorithm for the entire database, we introduce a *fuzzy precheck*. The idea of the precheck is simply to compute the codeword value of the page, ignoring ongoing updates and compare that to the codeword value in the codeword table. If the two match, we assume that the page is correct. Note that this introduces a slightly increased risk of missing an error because the effect on the codeword of a set of valid, in-progress updates might *exactly complement* the effect on the codeword of some corrupt data in the region. However, we consider the additional risk to be extremely small, approximately the same as the probability of an error going undetected by the codeword in the first place.

We now present the steps taken to audit the database:

1. For each page
   a. Note the value in the codeword table for the page.
   b. Compute its codeword (without latches or locks).
   c. Note the value in the codeword table for the page a second time.
   d. If the computed value does not match either noted value, add the page to AU_needed.
2. Note end_of_stable_log into AU_begin.
3. Copy pages in AU_needed to the side.
4. Extract the trailing physical-undo log records affecting pages in AU_needed for in-process transactions from the active transaction table ATT. Call this collection of physical records the AU_att. Records are gathered from different transactions independently, using a latch on the entry to ensure that operations are not committed by a transaction while we are gathering records from its log.
5. Get the flush latch and execute a flush to cause codewords from outstanding log records to be applied to the codeword table. Note the new end_of_stable_log in AU_end. Note the codeword values for all pages in AU_needed into a copy of the codeword table called AU_codewords. Finally, release the flush latch.
6. Scan the system log from AU_begin to AU_end. Physical redo records which apply to pages in AU_needed are applied to the side copy of those pages. Also, if the undo corresponding to this physical redo is in the AU_att, it is removed.
7. All remaining physical undo records from AU_att are applied to the checkpoint image.
8. Compute codewords of each page in AU_needed and compare them to the value in AU_codewords. If any differ, report that direct physical corruption has occurred on this page.

Discussion. We now give some intuition about why the above procedure is correct, though not a formal proof. The center of the audit procedure is the copy of the codeword table, AU_codewords, made during Step 5. The idea is that this codeword table reflects a certain set of updates—exactly those that appear on the system log prior to the point AU_end. The purpose of the rest of the algorithm through Step 7 is to ensure that the side copy of each page being checked contains (modulo any corruption) exactly those updates which preceed AU_end. All such updates which may have been partially or completely missing from the checkpoint image (due to it being acquired in a fuzzy manner) are reapplied from the system log in Step 6. Since in Dalí redo log records are not immediately placed in the system log, some updates may have taken place which are only recorded in various transactions' local redo and undo logs, thus the contents of the pages being audited may contain all or part of these additional updates. Further, the codeword deltas for these updates are *not* reflected in the codeword table since the codeword table is only updated during system log flush. Therefore, the effect of these updates must be undone in the side copy of pages in AU_needed. Once this is accomplished in Step 7, the side image of a page in AU_needed reflects exactly those updates which preceed AU_end and, thus, is consistent with the codeword for the page which was saved in

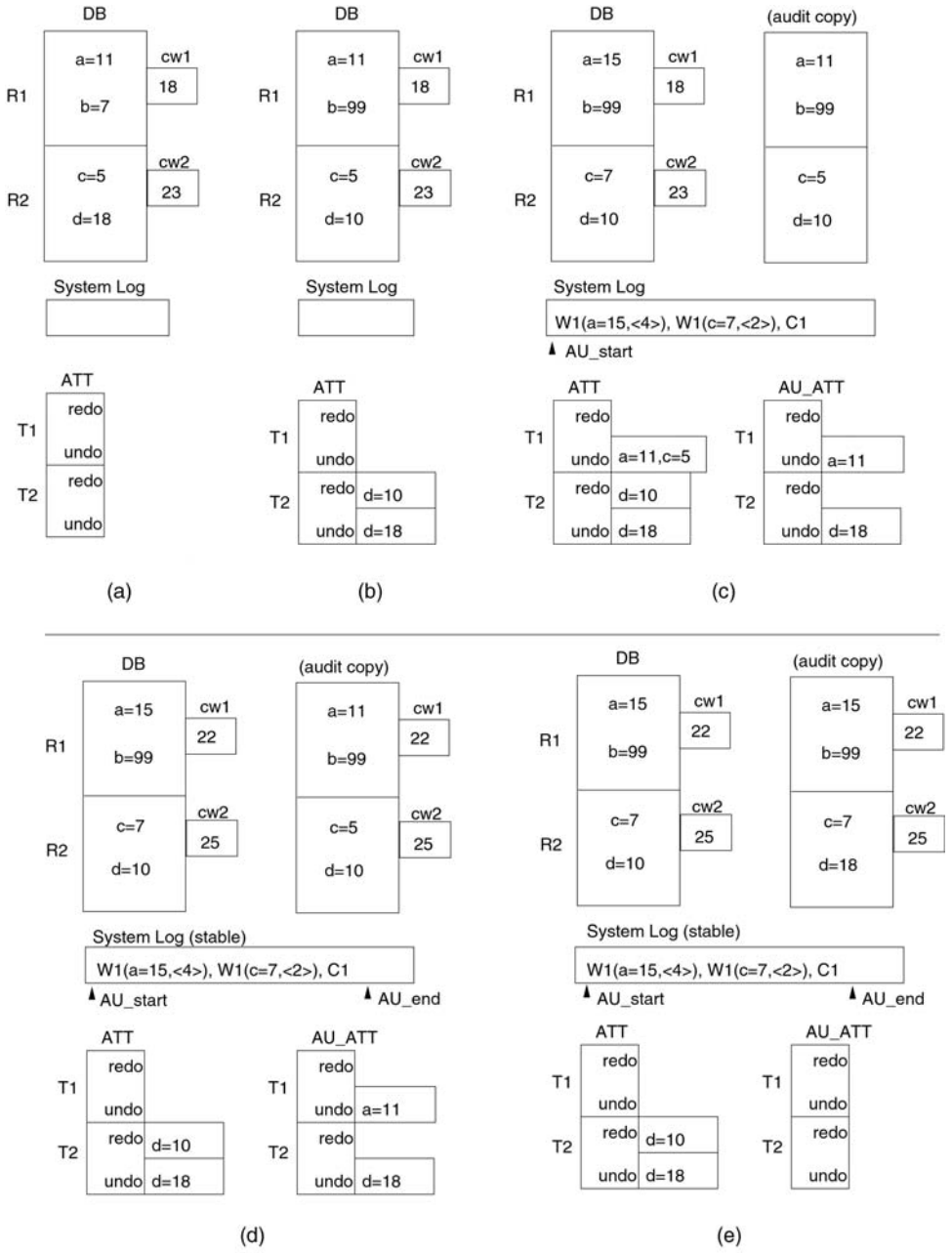

Fig. 3. Deferred maintenance audit example.

AU_codewords (as long as no corruption has occurred). From this consistency follows the basic correctness of the algorithm: A discrepancy detected at Step 8 reflects corruption of data, as opposed to a partially obtained checkpoint image or update for which the codeword has not been applied. Note, however, that the converse is not true—not all corruption is detected. This may be the case, for example, if corruption occurs on a data item for which an update appears in the physical tail of a transaction's undo log. In Step 7, such corruption would be overwritten in the side copy by the undo value and would remain undetected until the next audit.

In this algorithm, individual transactions are blocked long enough to gather the outstanding physical undo records for the current operation, otherwise, ongoing transaction processing continues without interference (other than holding the flush latch, which any committing transaction must also do).

Example 3.1. Consider a database consisting of two protection regions, $R_1 = \{a, b\}$ and $R_2 = \{c, d\}$, and two transactions, T_1 and T_2. Fig. 3 shows snapshots of the database, control structures, and audit data structures at five points in the audit procedure and illustrates some of the issues which arise due to concurrency between the audit process and ongoing transaction processing. The initial state of the database is shown in Fig. 3a, with the initial values of data items $a = 11, b = 7, c = 5, d = 18$. As mentioned earlier, we use unsigned addition for the codeword operator in examples for ease of presentation. Consequently, the codeword for region 1, cw_1, is 18, and

$cw_2 = 23$. In Fig. 3b, T_2 has updated d from 18 to 10, and an error in the program has corrupted data item b to have the value 99. In Fig. 3c, transaction T_1 has become active, but its actions are interleaved with the actions of the auditor. First, the auditor computes AU_needed. Since neither R_1 or R_2 has a codeword matching the corresponding data, both regions are added to AU_needed. Next, at Step 3, the pages in AU_needed are copied to the side; these pages are labeled "*audit copy*" in the figure. Next, T_1 performs two writes, setting $a = 15$ and $c = 7$. The audit procedure copies physical undo log records out of T_1's transaction-table entry in between the writes, capturing undo information only for a, at Step 4 of the audit algorithm. Then, T_1 begins a commit and it's redo log is moved to the system log, however, T_1's process is swapped out before it flushes the redo log (as would be typical for operation commits, not illustrated in this example, but which may also happen for transaction commits). Fig. 3c reflects the state of the data structures in the system when these steps are complete. Note that the codeword table, the audit copy, the ATT, and the AU_ATT are not mutually consistent.

The next step for the audit procedure (Step 5) is to flush the redo log, effectively committing T_1. As always during a flush with deferred codeword maintenance, the codeword is updated based on delta values stored in the log records being flushed, using the "codeword delta" values stored in the log record (shown in <> in the figure). Once the codeword-table updates have taken place, the codeword-table values for the pages in AU_needed are noted in the AU_codewords table and the current point in the stable log is noted as AU_end. The noted codeword values are shown in Fig. 3d as being attached to the audit copy, while AU_end appears as a pointer into the log. Clearly, the AU_codewords values are consistent with this point in the log and the goal of the remainder of the algorithm is to get the audit copy pages to also be *physically* consistent with this point in the log and, thus, with the associated codewords (modulo corruption). Fig. 3e shows the result of executing Steps 6 and 7. T_1's writes of a and c are applied to the audit copy and the value of d is returned to 18 using the physical undo log record in the AU_ATT. At this point, the data in the audit copy is checked against the codeword values noted in Step 5. R_2 passes ($7 + 18 = 25$), but R_1 fails ($15 + 99 \neq 22$). Thus, the corruption of data item c is detected, even while allowing concurrent updates to the database pages during the audit procedure.

4 Corruption Recovery

In this section, we consider how to recover from physical corruption which is detected rather than prevented. Since recovery is necessarily more complicated than prevention, it may seem an unnecessary effort to design recovery algorithms for this case, even if the corruption detection mechanism is significantly more efficient than the comparable corruption prevention mechanism. However, in the fault injection study performed by Ng and Chen, to study the reliability of a nonvolatile buffer cache in a DBMS, the addition of hardware protection only reduced the incidence of data corruption from 2.7 to 2.3 percent, indicating that a significant risk exists that physical corruption will go undetected [19]. Since they used a hardware protection scheme following [23], some of this corruption may have occurred during the time the page was unprotected. Such errors may be detected by the codeword schemes in this paper if the area being corrupted, while on the same page, is not specified as part of the update when the prescribed interface (beginUpdate/endUpdate) is called.

In other cases, however, the corruption may have been introduced through the prescribed routines, in which case, it does not meet the definition of physical corruption used in this paper. For these errors, codeword audit procedures will not be able to automatically detect the corruption. However, if other audit mechanisms such as those described in [13] or other asserts within the DBMS are available to determine the location and a lower bound on the time of the error, the recovery mechanisms described in this section can aid in the subsequent recovery.

4.1 Models of Corruption Recovery

We define three models of corruption recovery: the *cache-recovery* model, the *prior-state* model, and the *delete-transaction* model. Using the cache-recovery model with the Read Prechecking scheme and the two Data Codeword schemes is discussed in Section 4.3, and an algorithm which implements the delete-transaction model is given in Section 4.4.

In the cache-recovery model, direct physical corruption is removed from cache pages, assuming that indirect corruption has not occurred, and, because of that, corrupt data values are not reflected in any log records. In the prior-state model, the goal is to return the database to a transaction consistent state prior to the first possible occurrence of corruption. Most commercial systems support this model (see, for example, [16]).

In our final model, the delete-transaction model, we assume that corruption is dealt with by deleting the effects of certain transactions from the database image. Any transaction that read corrupted data must be deleted from history, and any data that such a transaction wrote after reading corrupt data is treated as being corrupted by the transaction. The identity of deleted transactions is then returned to the user to allow manual compensation for the effects of these transactions. Thus, we assume that corruption is detected relatively quickly and that the amount of corrupted data is limited. We do not attempt to analyze the speed at which corruption may spread since it is dependent on the details of the application, the DBMS implementation, and the initially corrupted data. Furthermore, it is up to the user to deal with any real-world actions or communication with external systems which has occurred during the execution of the deleted transactions. (Note that, in the prior-state model, it is up to the user to deal with compensating for *all* transactions which have occurred after the corruption, rather than just the ones determined to be possibly affected.) One important effect of recovery in the delete-transaction model is that consistency is restored to lower-level structures such as allocation tables and indexes.

To implement a recovery algorithm for this model, it must be clearly understood what it means to "delete a transaction from history." One possible interpretation would be to allow any serializable execution of the remaining, undeleted, transactions. However, this definition is not acceptable since the values read by other transactions and, thus, the values exposed to the outside world might change in the modified history.

To define correctness in this model, we consider two transaction execution histories, the original history, H_o, and the *delete history*, H_d. Each history is specified by the reads and writes issued by each transaction. In H_d, all reads and writes of certain transactions no longer appear. These histories include the values read or written by each operation, and, for a given operation, the value read or written in H_d is the same as for the corresponding operation in H_o. A delete history is *conflict-consistent*[3] with the original history if any read in H_d is preceded by the same write which preceded it in H_o. Similarly, H_d is *view-consistent* with H_o if each read in H_d returns the value returned to it in H_o. A correct recovery algorithm in the delete-transaction model recovers the database according to a delete history which is conflict or view-consistent with the original history. Note that it follows from this definition that, in a *conflict*-consistent delete history, the final state of any data item written by a transaction in the delete set will have the value it had before being written by the first deleted transaction.

4.2 Read Logging

In order to trace the indirect physical corruption as required by the delete-transaction model, we introduce the idea of *limited read logging*. When a data item is read, the identity of that item is added to the transaction log. Should it be determined through an audit or other means that certain data is corrupt, the read log records can help determine if any subsequent transactions have, in fact, read the corrupt data by serving as a form of audit trail [2]. The read log records combined with the rest of the log allow the transaction log to be used as a mechanism for tracing the flow of indirect corruption in the database. Since log records in Dalí are physical, we log reads at the physical level also. Thus, when data is read, the identity of that data is logged as a start point and a number of bytes. For efficiency, the data logged as read may overestimate the amount actually read.

4.2.1 Generating Checkpoints Free of Corruption

Since Read Logging supports recovery from indirect corruption, it becomes crucial that the disk image be free not only of direct corruption, but indirect corruption as well, so that a correct recovery does not require loading an archive image of the database. Thus, when propagating dirty pages from memory to disk, it is not sufficient to audit the pages being written. Even if none of the dirty pages has direct physical corruption, it is possible that a "clean" page has direct corruption and a transaction has carried this corruption over to a page that was written out. Thus, the checkpoint would have data that is indirectly corrupted.

The correct way of ensuring that the checkpoint is free of corruption is to create the checkpoint, and, after the checkpoint has been written out, audit every page in the database. If no page in the database has direct corruption, no indirect corruption could have occurred either. We can then certify the checkpoint free of corruption.

This technique cannot be directly applied to page flushes in a disk-based system since it amounts to auditing all pages in the buffer cache before any write of a dirty page to disk (at page steal time). However, a similar strategy can be followed if a set of pages are copied to the side and then an audit of all pages is performed before writing them to the database image on disk. To ensure that direct physical corruption does not escape undetected, a clean page which is being discarded to make room for a new page must also be audited.

4.3 Cache Recovery

This algorithm is useful to recover from direct corruption when recovery from indirect corruption is not required. It is invoked when a precheck fails in the Read Prechecking scheme or when an audit detects a codeword error in one of the Data Codeword schemes. By auditing pages before they are propagated to disk, we ensure that the disk image is free of corruption and, thus, repairing the corrupted cache image can be accomplished by applying standard recovery techniques to the region of data corrupted. For example, such a technique is described for ARIES in [17] for use when a page is latched by a process which was unexpectedly terminated. For Dalí, a similar technique can be used to reload some or all of the latest checkpoint and replay physical log records forward. We omit the details.

4.4 Delete-Transaction Model

The delete-transaction model of corruption recovery is tightly integrated with restart recovery. On detecting an error, we simply note the region(s) failing the audit and cause the database to crash, allowing corruption recovery to be handled as part of the subsequent restart recovery. For our delete-transaction model recovery algorithm, we need a checkpoint which is update-consistent in addition to being free from corruption. However, in Dalí, a checkpoint being used for recovery is not necessarily update-consistent until recovery has completed (that is, physical changes may only be partially reflected in the checkpoint image and certain updates may be present when earlier updates are not).

The algorithm to obtain an update-consistent checkpoint in Dalí is similar to the audit procedure for the Deferred Maintenance codeword scheme and uses a portion of the redo log and ATT to bring the checkpoint to a consistent state before the anchor is toggled and the checkpoint made active. Once performed, the checkpoint is update-consistent with a point in the log, CK_end. We omit the details.

4.4.1 Recovery Algorithm

The main idea of the following scheme is that corruption is removed from the database by refusing during recovery to perform writes which could have been influenced by corrupt data. In order to do this, the transactions which

3. Note that the notions of *conflict* or *view-consistency* are distinct from the standard notions of *conflict* or *view-equivalence*.

performed those writes must, at the end of the recovery, appear to have aborted instead of committed. Certain other transactions may also be removed from history (by refusing to perform their writes) in order to ensure that these "corrupt" transactions can be effectively removed and, thus, ensuring the final history as executed by the recovery algorithm is consistent with a delete history obtained from removing the "corrupt" transactions from the original execution (see Section 4.1).

Recovery must start from a database image that is known to be noncorrupt. Note that since errors are only detected during checkpointing or auditing, we may not know exactly *when* the error occurred; the error may have been propagated through several transactions before being detected. The algorithm below conservatively assumes that the error occurred immediately after Audit_LSN, the point in the log at which the last clean audit began.

Two tables, a CorruptTransTable and a CorruptDataTable are maintained. A transaction is said to have *read corrupt data* if the data noted in a read or write log record of that transaction is in the CorruptDataTable.

Redo Phase. The checkpointed database is loaded into memory and the redo phase of the Dalí recovery algorithm is initiated, starting the forward log scan from CK_end (which precedes Audit_LSN in the redo log).

During the forward scan, the following steps are taken (any log record types not mentioned below are handled as during normal recovery):

- If a read or write log record is found, then if this record indicates that the transaction has *read corrupted data*, then the transaction is added to CorruptTransTable (where it may already appear). Further, the data contained in physical log records in the transaction's undo log is added to CorruptDataTable.
- If a log record for a physical write is found, then there are two cases to consider:
  1. The transaction that generated the log record *is not* in CorruptTransTable: In this case, the redo is applied to the database image as in the Dalí recovery algorithm.
  2. The transaction that generated the log record *is* in the CorruptTransTable: In this case,, the data it would have written is inserted into CorruptDataTable. However, the data is *not updated*.
- If a begin operation log record is found for a transaction that is not in CorruptTransTable, then it is checked against the operations in the undo logs of all transactions currently in CorruptTransTable. If it conflicts with one of these operations, then the transaction is added to CorruptTransTable. This ensures that the earlier corrupt transaction can be rolled back. If it does not conflict, then it is handled as in the normal restart recovery algorithm.
- If a logical record such as commit operation, commit transaction, or abort transaction is found, the record is ignored if the transaction that generated the log record is in CorruptTransTable. Otherwise, the record is handled as in normal restart recovery.
- When Audit_LSN is passed, all data noted to be corrupt by the last audit is added to CorruptDataTable.

Undo Phase. At the end of the forward scan, incomplete transactions are rolled back. As in the normal Dalí algorithm, undo of all incomplete transactions is performed logically level by level. Note that, at the end of the redo phase, each transaction in CorruptTransTable has a (possibly empty) undo log, containing actions taken by the transaction before it first read corrupted data. During the undo phase, these portions of the corrupt transactions are undone as if the corrupt transactions were among those in progress at the time of the crash.

Checkpoint. The recovery algorithm is completed by performing a checkpoint to ensure that recovery following any further crashes will find a clean database free of corruption. If the checkpoint were not performed, a future recovery may rediscover the same corruption and, in fact, additionally declare transactions that started after this recovery phase to also be corrupted. Note that this checkpoint invalidates all archives. The log may be amended during recovery to avoid this problem, but this scheme is omitted due to space considerations.

4.4.2 Discussion

Following Section 4.1, the database image at the end of the above algorithm should reflect a delete history that is consistent (in this case, conflict-consistent) with the original transaction history. To see (informally) that this is the case, first observe that all top-level reads of nondeleted transactions read the same value in the history played during recovery as in the original history. This is because any data that could possibly have been read with different values was previously placed in CorruptDataTable, and top-level reads must be implemented in terms of reads at the physical-level where corruption is tracked. The second observation is that the database image is consistent and contains the original contents plus the writes of those transactions which do not appear in the delete set. This follows from the correctness of the original recovery algorithm, and the fact that the initial portion of corrupted transactions can be rolled back during the undo phase along with normal incomplete transactions to produce a consistent image. This is ensured since we do not allow any subsequent operations which conflict with these operations to begin.

Example 4.1. Consider a database containing five data items: a, b, c, d, and e. Furthermore, consider the redo log depicted in Fig. 4a containing read(R) and write(W) operations belonging to five transactions $T_1, \ldots, T_5$. (Operations belonging to transaction T_i have a subscript i). In the following, we trace the execution of the recovery algorithm in the delete-transaction model for the case when corruption of data item a, detected by an audit, causes the system to simulate a crash and recovery. Values of data items which appeared in the checkpoint used for recovery are shown by the name of the data item with a "0" subscript (see Fig. 4b). Further, we use subscript i to denote the value written by transaction T_i for an item in the database. (Thus, b_1 is the

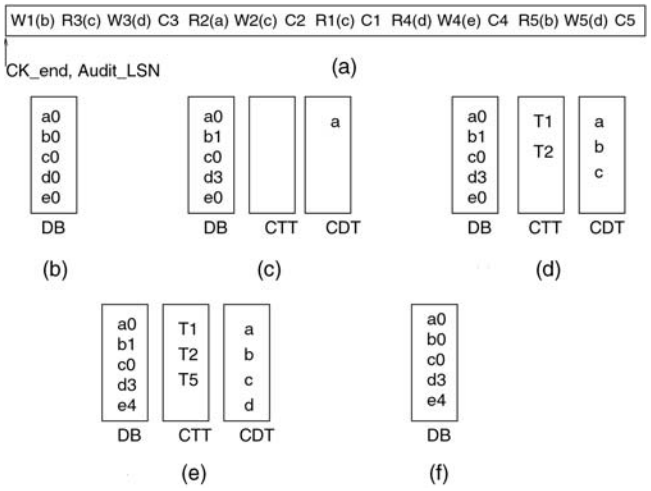

Fig. 4. Example of Recovery in the Delete-Transaction Model.

value written on data item b by transaction T_1). Finally, C_i represents the commit record for T_i.

At the start of the redo phase, the CorruptDataTable (CDT) and CorruptTransTable (CTT) are both empty. However, when the first operation $W_1(b)$ in the redo log is processed, since Audit_LSN is passed, data item a which was flagged as corrupt by the most recent audit is inserted into the CDT. Since neither transaction T_1 nor T_3 access a, writes performed by both of them are installed into the database and after T_3's commit, the database state is as shown in Fig. 4c. However, further along in the redo log, transaction T_2 reads a and is thus added to the CTT. Also, data item c written by it is subsequently inserted into the CDT (without updating the database itself), and this causes transaction T_1 also to be added to the CTT (since it reads c) and item b (written by it earlier) to be inserted into the CDT. Thus, after T_1's commit is processed, the database, CTT and CDT are as depicted in Fig. 4d. Processing the remainder of the redo log results in the database, CTT and CDT in Fig. 4e—here, T_4's updates are applied to the database since it does not access items in CDT, while T_5s are not since it reads item b in the CDT. In the final undo phase, transaction T_1's update of item b is undone (since T_1 belongs to the CTT). As a result, the final database state is as shown in Fig. 4f.

Thus, in the delete-transaction model, corruption of data item a causes writes of transactions T_1, T_2, and T_5 to be removed from history.

4.4.3 Extension: Codewords in Read Log Records

If codewords are stored in read log records, then detection of indirect corruption becomes more precise. In particular, the CorruptDataTable can be dispensed with, and instead, the definition of *reading corrupt data* given above is replaced by a definition in which a transaction read corrupt data if either of the following cases hold:

1. A codeword is stored in a read log record and it does not match the computed codeword for the corresponding region in the database being recovered.
2. A codeword is stored in a write log record (indicating that it should be treated as a read followed by a write) and the codeword does not match the computed codeword for the corresponding region in the database.

A second benefit of storing codewords in read log records is that, in the case of a true system failure (as opposed to one caused by a failed audit), it is possible to detect physical corruption which occurred after the last audit but before the crash. More precisely, physically corrupt data will be detected if any transaction read it since during recovery the codeword for these transactions will not match the database image being recovered. Thus, if codewords are present, the corruption recovery algorithm should be executed not only when an error is detected, but also on every restart.

Note that the modified algorithm produces a recovery schedule which is *view-consistent* with the original history, thus not propagating corruption when the corrupt transaction wrote the same data to a data item as it would have had in the delete-history.

5 Logical Corruption

In this section, we describe how Read Logging and the corruption recovery approach can be extended to handle certain cases of logical corruption. *Direct logical corruption* is defined as a point in the serialization order of a database execution where the *actual* and *intended* updates to the database differed. This difference may include updates which didn't happen but should have, as well as updates which happened but which should not have. All updates involved in an instance of direct logical corruption are defined to be made by one transaction; however, that transaction may not appear in the original transaction history—a missing transaction is one form of direct corruption. Examples of direct logical corruption include deletion of the incorrect customer, updates to account balances made by a program which computed interest incorrectly, failure to enter an item into the inventory database, and entry of an incorrect amount for sales commission. In the last example, if the sales commission is then used in payroll computations by another transaction, then that use of the incorrect data is an example of *indirect logical corruption*, which is analogous to the indirect physical corruption already discussed.

Unlike physical corruption, we assume that direct logical corruption does not include errors which are automatically detectable, for example, with the use of integrity constraints. Note that errors which do *not* cause internal inconsistencies generally are not detectable by integrity constraints—for example, incorrect entry of the amount of warehouse space occupied by a new inventory item. If the incorrect value is plausible, the error is difficult to detect. As another example, new inventory might be physically entered in a warehouse, but operators may fail to enter it in the database [9]. In this section, we describe recovering from logical corruption under two models. In Section 5.1, we show how the delete-transaction recovery algorithm of the previous section can be extended to handle logical corruption. Then, in Section 5.2, we introduce the redo-transaction model and show how recovery from logical

corruption can be implemented efficiently in a system with logical redo logging. We assume for each of these algorithms that the existence of the corruption has been detected by users of the system and traced to a single faulty transaction.

5.1 Recovery in the Delete-Transaction Model

The algorithm given in Section 4 for recovery in the delete-transaction model can be adapted in a straightforward manner to recover from logical corruption. While we assume the original direct corruption has been identified, the corruption may have occurred before the last checkpoint, thus an archive image that was taken prior to the introduction of the error is restored, and CK_end (the point at which recovery starts) is taken from that archive. Audit_LSN is defined to be the latest point in the log such that the corruption is known to have occurred after that point. Finally, when the point Audit_LSN is passed during recovery, any data which was directly affected by the corrupting error must be manually corrected, and these data items are added to the CorruptDataTable.

5.2 Recovery in the Redo-Transaction Model

The redo-transaction model is an extension of the prior-state model. The idea is that once a direct error has been identified and corrected, transactions affected by the error are *logically rerun in history*; that is, the transactions must be rerun in the same serialization order as the original set of transactions. We call this the *redo-transaction model of corruption recovery*. In order for this reexecution to be meaningful, the transactions must be *independent*; that is, the transactions which are submitted to the system by the users must not depend on the contents of the database. This would be the case, for example, if customers place orders without knowing whether an item is in inventory. Unfortunately, in a naive implementation of this model of corruption recovery, the logical reexecution of transactions may take time similar to their original execution. Thus, if an error is discovered after several days, it can reasonably be expected to take days to recover the database, during which time the database would be almost certainly be unavailable.

We now outline an alternative approach to implementing recovery for the redo-transaction model. This approach assumes that logical or physiological redo logging is performed by the underlying DBMS.[4] For simplicity of presentation, we assume further a traditional relational DBMS system in which the recovery model has two levels of abstraction: Record-level operations are logged, record-level locks are held for the duration of the transaction, and latches are held on pages for the duration of an operation. The goal of the algorithm is to use primarily the log for recovery, only reexecuting a minimal number of transactions.

In this algorithm, corruption is tracked logically in the data model of the DBMS, which has implications for how read logging is performed and how the CorruptDataTable data structure is maintained. In fact, we could continue to log reads physically as described in Section 4.2, depending on the property that a lack of a conflict at a lower level of abstraction implies the absence of corresponding conflicts in parent operations at higher levels of abstraction [27]. However, this will overestimate the flow of corruption in a manner analogous to the way that physical locking overestimates the conflict between operations. In fact, the above adaptation of the delete-transaction model recovery algorithm to logical corruption overestimates corruption flow in this way. In order to trace corruption more conservatively, we can log reads logically. For correctness, a simple property has to be maintained: If the system would return a different result to a transaction when running on the modified database, then it must be that the recovery algorithm will detect a conflict between one of the read log records for that transaction and the contents of the CorruptDataTable.[5]

In the algorithm given below, the CorruptDataTable contains tuples from the database and read log records are implemented as predicates over the relations in the database. If any tuple in CorruptDataTable table is matched by a predicate in a transaction's read log, the transaction is said to have *read corrupted data*.

Recovery. In addition to the actions normally taken during recovery, the corruption recovery algorithm proceeds by processing log records as follows:

- Log records for a transaction are saved until the commit for the transaction is seen.
- If a commit log record for a transaction is encountered, then
  - The read log records for the transaction are scanned and the predicates they contain are evaluated against the CorruptDataTable to determine if the transaction has *read corrupted data*. If so, the following steps are taken:
    1. The transaction is marked as corrupt.
    2. The transaction is reexecuted logically and the new logical redo records are used to replace its redo records in the log.
    3. Consider that the set of tuples updated can either be the original transaction (taken from the log) or the reexecuted transaction (noted by the reexecution). A tuple, τ, from this set is added to the CorruptDataTable if one of the following conditions holds:
       a. τ is present in the database following the original execution but not following the reexecution.
       b. τ is not present in the database following the original execution but is present following the reexecution.
  - If the transaction is not marked as corrupt, its log records are executed as normal.
- If an abort record for a transaction is found, the log records for that transaction are discarded.

4. This assumption does not hold for the Dalí system as described in Section 2, which does physical redo logging. However, many commercial systems have logical redo logs [17] and such logs are currently being introduced in Dalí for use in replication.

5. Note that we assume that the order of nondeterministic events, such as the order in which tuples are returned when a query result order is not specified, is the same in the original execution and the reexecution.

TABLE 1
Performance of Protect/Unprotect

Platform	pairs/second
SPARCstation 20	15,600
UltraSPARC 2	43,000
HP 9000 C110	3,300
SGI Challenge DM	8,200

Discussion. Since the redo records are logical and record-level locks are held to the end of transaction, executing the log records at the point where the commit record appears is correct. Note that when a transaction is reexecuted, it could generate different log records. In fact, the operations it performs may be completely different from the ones it originally performed. Since transactions are executed at their commit point, this new transaction will serialize with respect to other transactions which might have originally executed concurrently with it. It will read the data as written by any transaction which serialized before it, and if its actions cause any new conflicts with transactions that serialize after it, these transactions will read corrupt data and be reexecuted themselves.

6 Performance

The goal of our performance study was to compare the relative cost of different levels of protection from physical corruption, for example, detection versus prevention, as well as comparing different techniques for obtaining the same level of protection. In each case, we are interested in the impact of the scheme on normal processing as opposed to the time taken for recovery. Corruption recovery is expected to be relatively rare and the time required is highly dependent on the application and workload. The algorithms studied were implemented in the DataBlitz Storage Manager, a storage manager being developed at Bell Labs based on the Dalí main memory storage manager.

6.1 Performance of mprotect

Before describing the results of our study of protection schemes, we begin by looking at the relative performance of memory protection primitives on commonly available UNIX platforms. In Table 1, we evaluate the basic performance of the memory protection feature on a number of hardware platforms locally available to us. In each case, 2,000 pages were protected and then unprotected, and this was repeated 50 times. The number reported is the average number of these *pairs* of operations which were accomplished per second. Even this limited sample indicates the variability in the performance of mprotect: the HP 9000 C110, which has about twice the integer performance of the SPARCstation 20 (170.2 SPECint92 for the HP as opposed to 88.9 for the Sun[6]), gives less than one-fourth the performance of the SPARCstation when performing mprotect/unprotect operations.

6. From the database at http://performance.netlib.org. It was not possible to compare these numbers for all machines, as only SPECint95 numbers are available for the newer systems.

TABLE 2
Precheck Domain Sizes

Size	Ops/Sec	% Slower	Space
None	417	0%	–
64	366	12.2%	6.25%
128	348	16.5%	3.13%
256	329	21.1%	1.56%
512	311	25.4%	.78%
1024	277	33.5%	.39%
8192	115	72.4%	.05%

6.2 Workload

The workload examined is a single process executing TPC-B style transactions. The database consists of four tables, Branch, Teller, Account, and History, each with 100 bytes per record. Our database contained 100,000 accounts, with 10,000 tellers and 1,000 branches. The ratios between record types are changed from those specified in TPC-B, in order to increase the size of the smaller tables and, thus, limit the effects of CPU caching on these tables. The benchmarks were run on an UltraSPARC with two 200Mhz processors, and 1 gigabyte of memory. All tables are in memory during each run, with logging and checkpointing ensuring recoverability. In each run, 50,000 operations were done, where an operation consists of updating the (nonkey) balance fields of one account, teller and branch, and adding a record to the history table. Transactions were committed after 500 operations so that commit times do not dominate.[7] Each test was run six times and the results averaged. The results are reported in terms of number of operations completed per second.

6.3 Prechecking and Protection Domain Size

Before presenting general results, we discuss a trade off in the implementation of the Read Prechecking algorithm. The Read Prechecking algorithm verifies each read by computing the codeword of the regions which the read intersects. Since one codeword is stored for each protection domain, the size of the region leads to a time-space trade off for this scheme. We present the performance of Read Prechecking with Data Codeword maintenance for a variety of sizes of protection domains from 64 bytes to the 8K page size of our machine. With small size protection domains, this scheme performs well, but may add 3 to 6 percent to the space usage of the database. The scheme breaks even with hardware protection at about 1K protection domains. These results are shown in Table 2.

6.4 Results

In Table 3, a representative selection of the algorithms discussed in this paper are shown, along with the average number of operations per second the algorithm achieved in our tests, the space overhead, and the relative slowdown of the algorithm compared to the baseline algorithm, which is

7. The alternative was to design a highly concurrent test with group commits, introducing a great deal of complexity and variability into the test.

TABLE 3
Cost of Corruption Protection

Algorithm	Protect. Reg.Size	Corruption			Ops/ Sec	Pct. Slower	Space O'head
		Phys Dir	Phys Ind	Log Ind			
Baseline	–	–	–	–	417	0%	–
Data Codeword	8K	Cor	–	–	380	8.5%	.05%
Read Precheck	64B	Cor	Pvnt	–	366	12.2%	6.25%
ReadLog (+Data CW)	8K	Cor	Cor	Cor	345	17.1%	.05%
CW in ReadLog (+Data CW)	8K	Cor	Cor	Cor	323	22.4%	.05%
Read Precheck	512B	Cor	Pvnt	–	311	25.4%	.78%
Memory Protection	8K	Pvnt	Pvnt	–	257	38.2%	–
Read Precheck	8K	Cor	Pvnt	–	115	72.4%	.05%

just the system running with no corruption protection. To allow a qualitative comparison, the table also lists what forms of errors are either prevented ("Pvnt") or possible to correct ("Cor") with a given technique. In the case of correcting indirect corruption, it is assumed that a technique like recovery in the delete-transaction model is used; however, our experiments focus on performance during regular transaction processing so the particular recovery technique is not important.

Our experiments show that detection of direct corruption can be achieved very cheaply, with a 6 to 8 percent overhead, using simple or deferred maintenance data codeword protection. Interestingly, the deferred maintenance scheme had a slightly lower overhead, although the concurrency aspects of the two schemes was not tested in this study. Prechecking with a small domain size is economical at a 12 percent cost, depending on the acceptability of a 6 percent space overhead. Read logging lowers the space overhead, but raises the cost to 17 percent, which is significant, but may be worthwhile since automatic support for repairing the database can then be employed and logical corruption can be addressed. Logging the checksum of the data read, which increases the accuracy of the corruption recovery algorithms, adds 5 percent to the cost, bringing it to 22 percent. Memory protection using the standard *mprotect* call on an UltraSPARC costs 38 percent, more than double the performance hit of codeword protection with read logging. Finally, prechecking with large domain sizes fares very poorly.

By monitoring the number of mprotect calls for the hardware-based scheme, we determined that, on average, operations updated about 11 pages. Only four tuples are touched by an operation and the extra page updates arise from updates to allocation information and control information not residing on the same page as the tuple. Thus, this number may be significantly smaller for a page-based system, which would improve the performance of Hardware Protection and Read Prechecking relative to the detection schemes. However, even if a factor of three improvement is realized, the variability in performance of mprotect described in Section 6.1 will more than erase this gain on some systems.

Our conclusion from these results is that some form of codeword protection should be implemented in any DBMS in which application code has direct access to database data. Detection of direct corruption is quite cheap, and while limited, is still far better than allowing corruption to remain undetected in the database. Other levels of protection may be implemented or offered to users so that they may make their own safety/performance trade off.

7 Related Work

Sullivan and Stonebreaker [23] use the hardware support for memory protection to unprotect and reprotect the page accessed by an update. By writing special system calls into the Sprite operating system and making use of the Translation Lookaside Buffers on their hardware platform, they found that page protection could be turned on and off relatively cheaply. In contrast, the new techniques introduced in this paper do not require a special operating system or hardware support, easing portability of the DBMS. More importantly, the performance of our schemes is not limited by the operating system's implementation of the mprotect system call (see Section 6.1).

Presumably, type-safe languages could be used to provide protection from direct physical corruption, and such techniques were used, for example, in the SPIN operating system [5]. However, C and C++ are still dominant for CAD and other high-performance uses of memory-mapped database systems. Sandboxing (see, for example, [28]) provides an alternate technique for protecting data by rewriting object modules, and with only a minor performance impact. However, the object module rewriting must be redesigned for each target architecture, which may be a significant limit on portability and since the technique requires a number of free registers to perform well, it may not be applicable on architectures without a large register set, such as the Intel x86 architecture. By contrast, our techniques are language and instruction-set independent.

Ng and Chen [19] study the reliability of three different interfaces to a persistent cache—1) based on the I/O model, 2) based on direct read/write (without memory protection), and 3) based on direct read/write with memory protection, using the POSTGRES database system. They inject a variety

of faults (hardware and software) and then check if persistent data has been corrupted. They conclude that the three interfaces provide a similar degree of protection. What is more important to note is that about 2.5 percent of the crashes due to the injected faults resulted in persistent data being corrupted, a rather high number for a database system. Thus, techniques to detect and recover from physical corruption are important, even in the absence of application code with direct access to the database buffers.

Küspert [13] presents a number of specific techniques for detecting corruption of DBMS data structures. These techniques are ad hoc in the sense that they are designed for specific DBMS storage structures. For example, it is recommended that a few words be reserved between each page's image in the buffer manager as a "defense wall" to detect writes that erroneously extend beyond a page's boundary. These techniques serve to detect certain bad writes as well as imposing certain integrity constraints to catch DBMS programming errors. Our techniques address the first area (protection against bad writes), but are far more general. Taylor et al. [25], [26] provide some theoretical structure to the design of data structures that can recover from certain failures. However, this work is not in the context of a DBMS and does not apply to corruption of general application data since it handles only components such as pointers and counts.

Finally, we note that commercial databases may well have implemented techniques for detecting corruption; however, such information is not publicly available. Our page-based direct corruption detection scheme is inspired by a codeword scheme for protecting telephone switch data [14], but as far as we are aware, neither codeword-based detection and recovery techniques nor read logging techniques are present in any published description of either a research or commercial database system.

8 Conclusions and Future Work

We have described a variety of schemes for preventing or detecting physical corruption using codewords and an algorithm for tracing and recovering from physical corruption under the delete-transaction model. We have presented a performance study comparing alternative techniques of corruption detection and recovery. Our study demonstrates that detection of direct physical corruption is economical to implement, that transaction-carried corruption can be prevented cheaply if enough space is available for small protection domains, and that detection of transaction-carried corruption for later correction through read logging imposes about a 17 percent overhead on update transaction performance. However, this technique opens up interesting possibilities in tracing errors through the database system and aiding in their correction. For a non-page-based system such as Dalí, the new schemes are much cheaper than using the UNIX memory protection primitives for every update. Furthermore, since the codeword schemes depend on simple integer operations, we expect them to be easily portable, to perform consistently over all platforms, and to scale in speed with the integer performance of new machines, while hardware-based protection may be much slower on certain systems with expensive system calls.

We believe our techniques for physical corruption will be of increasing importance since applications are increasingly being provided direct access to persistent data. Since it is relatively cheap, we believe implementors of database systems in which application code has direct access to database buffers should always provide detection of direct physical corruption as a minimum.

We believe the delete-model recovery algorithm is the first concrete proposal for integrating recovery from cross-transaction errors into the crash recovery subsystem of a DBMS. The model of tracing the indirect effects of errors is applicable for other forms of corruption. When an error in the database results from incorrectly coded application programs or due to incorrect user input, we define this to be *logical* corruption, based on the intuition that the error is introduced at a higher level of abstraction than addressing errors. Unlike physical corruption, direct logical corruption cannot be efficiently detected (unless the violation of an integrity constraint causes the transaction which would enter the corruption to roll back). Thus, logical corruption may remain in the database until a user or auditor notices the error. Recovery from such corruption can be very difficult with problems ranging from accurate tracing of corruption over much longer times to dealing with external actions associated with transactions. We have presented an initial algorithm for efficient recovery of a database from logical corruption. We are convinced that 1) automatic tools like these can be a significant aid in logical corruption recovery, following the outline of [6], and 2) as with the implementation of read logging for the delete-transaction model, such tools can benefit significantly from support within the DBMS.

Acknowledgments

The authors would like to thank Dennis Leinbaugh for insightful discussions on codeword maintenance. The work of A. Silberschatz and S. Seshadri was performed, and the work of S. Sudarshan was performed in part, when the respective authors were at Bell Laboratories.

References

[1] B. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy, "Lightweight Remote Procedure Call," *ACM Trans. Computer Systems,* vol. 8, no. 1, pp. 37-55, Feb. 1990.

[2] L.A. Bjork Jr., "Generalized Audit Trail Requirements and Concepts for Data Base Applications," *IBM Systems J.,* vol. 14, no. 3, pp. 229-245, 1975.

[3] P. Bohannon, D. Lieuwen, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan, "The Architecture of the Dalí Main-Memory Storage Manager," *J. Multimedia Tools and Applications,* vol. 4, no. 2, pp. 115-151, Mar. 1997.

[4] P. Bohannon, J. Parker, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan, "Distributed Multi-Level Recovery in a Main-Memory Database," *Proc. Fourth Int'l Conf. Parallel and Distibuted Information Systems,* 1996.

[5] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S.J. Eggers, "Extensibility, Safety and Performance in the Spin Operating System," *Proc. 15th ACM Symp. Operating System Principles,* pp. 267-284, Dec. 1995.

[6] C.T. Davies Jr., "Data Processing Spheres of Control," *IBM Systems J.,* vol. 17, no. 2, pp. 179-198, 1978.

[7] D.J. DeWitt, R. Katz, F. Olken, D. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* pp. 1-8, June 1984.

[8] V. Gottemukkala and T. Lehman, "Locking and Latching in a Memory-Resident Database System," *Proc. Int'l Conf. Very Large Databases,* pp. 533-544, Aug. 1992.

[9] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* San Mateo, Calif.: Morgan Kaufmann, 1993.
[10] J. Gray, "A Census of Tandem System Availability between 1985 and 1990," *IEEE Trans. System Reliability,* vol. 39, no. 4, pp. 409-418, Oct. 1990.
[11] H.V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan, "Dalí: A High Performance Main-Memory Storage Manager," *Proc. Int'l Conf. Very Large Databases,* 1994.
[12] H.V. Jagadish, A. Silberschatz, and S. Sudarshan, "Recovering from Main-Memory Lapses," *Proc. Int'l Conf. Very Large Databases,* 1993.
[13] K. Kuspert, "Principles of Error Detection in Storage Structures of Database Systems," *Reliability Eng.: An Int'l J.,* vol. 14, 1986.
[14] D. Leinbaugh, Personal Communication, Nov. 1994.
[15] D. Lomet, " MLR: A Recovery Method for Multi-Level Systems," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* pp. 185-194, 1992.
[16] K. Loney, *ORACLE8 DBA Handbook.* Osborne McGraw-Hill, 1998.
[17] 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 Trans. Database Systems,* vol. 17, no. 1, pp. 94-162, Mar. 1992.
[18] D. Morgan and D. Taylor, "A Survey of Methods for Achieving Reliable Software," *Computer,* vol. 10, no. 2, Feb. 1977.
[19] W.T. Ng and P.M. Chen, "Integrating Reliable Memory in Databases," *Proc. Int'l Conf. Very Large Databases,* pp. 76-85, Aug. 1997.
[20] J. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware," *Proc. USENIX Summer 1990 Conf.,* pp. 247-256, 1990.
[21] B. Randell, "System Structure for Software Fault Tolerance," *Computer,* vol. 10, no. 2, Feb. 1977.
[22] "The Postgres Papers," technical report, UCB, Elec.Res.Lab, Memo No. M86-85, M. Stonebraker and L.A. Rowe, eds., rev. Jun. 1987, Nov. 1986.
[23] M. Sullivan and M. Stonebreaker, "Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems," *Proc. Int'l Conf. Very Large Databases,* pp. 171-179, 1991.
[24] M. Sullivan, "System Support for Software Fault Tolerance in Highly Available Database Management Systems," Technical Report ERL-93-5, Univ. California, Berkeley, Jan. 1993.
[25] D. Taylor, D. Morgan, and J. Black, "Redundancy in Data Structures: Improving Software Fault Tolerance," *IEEE Trans. Software Eng.,* vol. 6, no. 6, pp. 585-594, Nov. 1980.
[26] D. Taylor, D. Morgan, and J. Black, "Redundancy in Data Structures: Some Theoretical Results," *IEEE Trans. Software Eng.,* vol. 6, no. 6, pp. 595-602, Nov. 1980.
[27] G. Weikum, C. Hasse, P. Broessler, and P. Muth, "Multi-Level Recovery," *Proc. ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems,* pp. 109-123, June 1990.
[28] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-Based Fault Isolation," *Proc. 14th ACM Symp. Operating System Principles,* pp. 203-216, Dec. 1993.

Philip Bohannon received the BS degree in computer science from Birmingham-Southern College in 1986, and MS and PhD degrees in computer science from Rutgers University in 1998 and 1999, respectively. He is a member of the technical staff in the Computing Sciences Research Center at Bell Laboratories, Lucent Technologies. Active in database research, Dr. Bohannon's current research activities center around XML publishing, XML indexing, and cache conscious database processing. Other interests include database fault management, main-memory databases, transaction processing, index management, and database security.

Rajeev Rastogi received the BTech degree in computer science from the Indian Institute of Technology, Bombay, in 1988, and the master's and PhD degrees in computer science from the University of Texas, Austin, in 1990 and 1993, respectively. He is the director of the Internet Management Research Department at Bell Laboratories, Lucent Technologies. He joined Bell Laboratories in Murray Hill, New Jersey, in 1993, and became a Bell Labs Fellow in 2003. Dr. Rastogi is active in the field of databases and has served as a program committee member for several conferences in the area. He currently serves on the editorial board of *IEEE Transactions on Knowledge and Data Engineering*. His writings have appeared in a number of ACM and IEEE publications and other professional conferences and journals. His research interests include database systems, network management, and knowledge discovery. His most recent research has focused on the areas of network topology discovery, monitoring, configuration, provisioning, XML publishing, approximate query answering, and data stream analysis.

S. Seshadri received the BTech degree in computer science from the Indian Institute of Technlogy, Madras, and then the MS and PhD degrees from the University of Wisconsin, Madison. He was on the faculty of Indian Institute of Technology, Bombay, before he joined Bell Laboratories. Currently, he is the CEO of Strand Genomics in Bangalore, India. Prior to joining Strand, he was the director of the Distributed Computing Research Department at Bells Laboratories, Lucent technologies. His current research interests include data management for life sciences, database systems, and communication networks.

Avi Silberschatz is a professor of computer science at Yale University. Prior to joining Yale, he was the vice president of the Information Sciences Research Center at Bell Laboratories, Murray Hill, New Jersey. Prior to that, he held a chaired professorship in the Department of Computer Sciences at the University of Texas at Austin. His research interests include operating systems, database systems, real-time systems, storage systems, and distributed systems. In addition to his academic and industrial positions, Dr. Silberschatz served as a member of the Biodiversity and Ecosystems Panel on President Clinton's Committee of Advisors on Science and Technology, as an advisor for the US National Science Foundation, and as a consultant for several private industry companies. Professor Silberschatz is an ACM fellow and an IEEE fellow. He received the 2002 IEEE Taylor L. Booth Education Awars, the 1998 ACM Karl V. Karlstrom Outstanding Educator Award, the 1997 ACM SIGMOD Contribution Award, and the IEEE Computer Society Outstanding Paper award for the article "Capability Manager," which appeared in the *IEEE Transactions on Software Engineering*. His writings have appeared in numerous ACM and IEEE publications and other professional conferences and journals. He is a coauthor of two well-known textbooks—*Operating System Concepts* (John Wiley & Sons, Inc., 2001) and *Database System Concepts* (McGraw-Hill, 2001).

S. Sudarshan received the BTech degree from the Indian Institute of Technology, Madras, in 1987, and the PhD degree from the University of Wisconsin, Madison, in 1992. Currently, he is a professor in the Computer Science and Engineering Department at the Indian Institute of Technology, Bombay, India. He was a member of the technical staff in the database research group at AT&T Bell Laboratories, from 1992 to 1995, prior to joining ITT Bombay. His writings have appeared in numerous ACM and IEEE publications and other professional conferences and journals. He is coauthor of the well-known textbook *Database System Concepts* (McGraw-Hill, 2001). Dr. Sudarshan's research interests include processing and optimization of complex queries, keywrod search in databases, failure recovery, and main-memory databases.