A Survey of B-Tree Logging and Recovery Techniques

GOETZ GRAEFE, Hewlett-Packard Laboratories

B-trees have been ubiquitous in database management systems for several decades, and they serve in many other storage systems as well. Their basic structure and their basic operations are well understood including search, insertion, and deletion. However, implementation of transactional guarantees such as all-or-nothing failure atomicity and durability in spite of media and system failures seems to be difficult. High-performance techniques such as pseudo-deleted records, allocation-only logging, and transaction processing during crash recovery are widely used in commercial B-tree implementations but not widely understood. This survey collects many of these techniques as a reference for students, researchers, system architects, and software developers. Central in this discussion are physical data independence, separation of logical database contents and physical representation, and the concepts of user transactions and system transactions. Many of the techniques discussed are applicable beyond B-trees.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design—*Recovery and restart*; H.2.4 [**Database Management**]: Systems—*Transaction Processing*; H.3.0 [**Information Storage and Retrieval**]: General

General Terms: Algorithms

ACM Reference Format:
Graefe, G. 2012. A survey of B-tree logging and recovery techniques. ACM Trans. Datab. Syst. 37, 1, Article 1 (February 2012), 35 pages.
DOI = 10.1145/2109196.2109197 http://doi.acm.org/10.1145/2109196.2109197

1. INTRODUCTION

B-trees [Bayer and McCreight 1972] are primary data structures not only in databases but also in information retrieval, file systems, key-value stores, and many application-specific indexes. Compared to the simpler binary trees, B-trees are more suitable to paged environments such as virtual memory, traditional disk storage, flash devices, and perhaps even CPU caches. In addition to equality predicates ("=" and "in" predicates) supported by all indexes, the order implicit in B-trees enables efficient range queries ("≤" and "between" predicates) as well as efficient index creation by sorting all future index entries. B-trees permit high-performance retrievals and updates, in particular if the implementation includes techniques such as normalized keys (binary comparisons only), prefix and suffix compression (truncation) [Bayer and Unterauer 1977], pinning root node and perhaps additional B-tree levels in the buffer pool, and even pointer swizzling [Moss 1992], that is, direct memory-to-memory pointers rather than repetitious look-ups in the buffer pool index.

While basic operations and algorithms for B-tree data structures are well known, often taught and even prototyped during undergraduate education, strengthening their implementation with transaction semantics seems rather difficult. The present survey covers recovery techniques for B-tree indexes, with a focus on techniques based on

Author's address: G. Graefe, Hewlett Packard Laboratories; email: goetz.graefe@hp.com.

DOI 10.1145/2109196.2109197 http://doi.acm.org/10.1145/2109196.2109197

write-ahead logging. In addition to the basic techniques for write-ahead logging and log-based recovery, many techniques are specific to B-tree indexes, their semantics, their data structures, and their maintenance algorithms. Moreover, many techniques and many optimizations not widely known are required in order to achieve competitive performance and resource-efficient operation of relational databases, key-value stores, "No SQL" and "New SQL" databases, indexes for Web search and information retrieval, file systems, and many other forms of storage with support for ordered search and incremental updates.

A recent survey focuses on separation of logical index contents and their physical representation in concurrency control [Graefe 2010]. For example, two B-tree indexes might be logically equivalent even if they differ in the number of nodes, in the free space within nodes, etc. Therefore, splitting a B-tree node modifies the index representation but not its contents, and posting a separator key in a parent node is quite different from insertion of a new entry into an index leaf. This separation of logical contents and physical representation is similarly important for logging and recovery. It defines the distinctions between user transactions and system transactions (Section 3) and between logical, physical, and physiological logging (Section 4). Many optimizations in logging and recovery can be explained, understood, designed, implemented, and tested based on these concepts.

1.1. A Very Brief Primer

Central in any discussion of concurrency control, logging, and recovery is the transaction concept and the *ACID* properties that define transactions: "all or nothing" failure atomicity, database consistency before and after each transaction, mutual isolation of concurrent operations or synchronization atomicity, and durability even in cases of any failures including "fire, flood, or insurrection."

Write-ahead logging is the primary means for providing failure atomicity and durability. Its essence is that the recovery log describes changes before any in-place updates of the database. Each type of update requires a "do" method invoked during initial processing, a "redo" method to ensure the database reflects an update even after a failure or crash, and an "undo" method to bring the database back to its prior state. The "do" method creates log records with sufficient information for "redo" and "undo" invocations and instructs the buffer pool to retain dirty data pages until those log records have arrived safely on "stable storage." Recovery is reliable inasmuch as the stable storage with the recovery log is. Mirroring the log device is a common technique. Log pages, once written, must never be modified or overwritten.

Database logging and recovery ensure correct operation in three classes of failures, with the intention to cover all specific instances of failures as special cases of the following classes.

—A *transaction failure* is the database-internal reaction to a user or an application explicitly requesting to abort a transaction. Other possible causes include deadlock resolution or a space allocation failure. Only a single transaction must roll back while other transactions and the system as a whole remain operational.
—A *media failure* may be due to mechanical hard disk failure, removal from the system, or network partitioning in case of remote storage. The recovery process must restore the disk contents by restoring a backup copy and then scanning and applying the transaction log to repeat the recent history of updates.
—A *system failure* may result from a sudden power or hardware failure, a programming error, a crash of the operating system, or some other reason. Based on an analysis of the recovery log and of recently updated database pages, the recovery process must

ensure durability of all transactions committed before the failure and roll back all other transactions.

Some modern hardware, in particular semiconductor storage, may suffer another class of failure, namely a hardware failure limited to individual data pages. This case can be treated as a very small media failure or it can have its own specialized recovery methods. For example, if all log records pertaining to the same data page form a linked list, recovery can apply them quite efficiently to a backup copy of the page. This approach requires appropriate backup (or snapshot) copies, a search or an index to find the newest log record pertaining to a failed data page, and reversal of the linked list such that log records are applied in the correct order.

Finally, checkpoints are a crucial technique for efficient recovery from system failures. A checkpoint forces pending log records to stable storage and dirty data pages lingering in the buffer pool to the database. The log record describing the checkpoint includes pointers to the most recent log record for each active transaction. When the checkpoint is complete, a pointer to the most recent checkpoint is installed in a location outside the log (e.g., in the file system) where crash recovery will find it if necessary. Taking a checkpoint every few minutes ensures that recovery from a system failure also takes only a few minutes. The specific relationship depends on the level of update activity prior to the crash, the length of transactions active during the last checkpoint, the time required to load required data pages into the buffer pool, and more.

1.2. Historical Background

When commercial data processing consisted primarily of printing bills and merging master tapes with tapes with records of recent business events, there was no need for explicit logging and recovery. The tape holding recent events was the log, the old master tape the backup, and the recovery process was the same as the original process. Direct access storage (disks) permitted updates in place, including immediate updates when business events happened. This created the need for logging and recovery in cases of failures.

Speaking broadly, there are five approaches to providing failure atomicity and durability. Their recovery techniques for system failures differentiate them quite clearly. They may be associated vaguely with successive decades starting with the 1970s.

(1) Gray [1978] describes how system recovery applies "redo" actions to pages updated by committed transactions and "undo" actions otherwise. "Redo" and "undo" actions must be *idempotent*, that is, repeated application of the same action results in the same state as a single application. An underlying assumption is that the recovery process keeps the recovery log in read-only mode, that is, no logging during recovery from a failure. Another underlying assumption is that updated pages overwrite the prior page contents.
(2) Gray et al. [1981] describe shadow pages in System R, that is, all updated pages require new disk locations. Rollback simply goes back to the prior pages. A granularity of locking finer than pages is not possible because shadow pages cannot reflect the state of one transaction committed and another aborted. Härder and Reuter [1983] formalize the role of the buffer pool in correct transaction implementations. They define "force" and "no force" policies that do or do not write dirty data pages back to the database prior to transaction commit as well as "steal" and "no steal" policies that do or do not permit multiple transactions to share dirty pages in the buffer pool.
(3) ARIES [Mohan et al. 1992] logs recovery actions, that is, the recovery log is in read-write mode during recovery. Using the log, ARIES reliably applies "redo" and "undo" actions exactly once by recording in each data page the most recent log

record already applied. Moreover, it "compensates" updates logically rather than physically. For example, a deletion compensates an insertion, yet after a leaf split in a B-tree index, the deletion might occur in a different leaf page than the insertion. Aborting a transaction applies compensating updates and then commits normally, except that there is no need to immediately force the commit record to stable storage. A crucial advantage of the ARIES design is its support for record-level locking and, in B-tree indexes, of key range locking [Mohan 1990]. Weikum [1992] provides a thorough and more general treatment of the topic based on multilevel transactions. Similarly, Moss' open nested transactions [2006] seem more principled than ARIES top-level actions and closely related to system transactions, discussed shortly (Section 3.2).

(4) A recovery log on stable storage is not required if a sufficient number of copies of all data pages can provide durability. An "undo"-only recovery log is still required for transaction rollback. It can be private to the transaction, it can shrink back while the transaction rolls back, and it can be deleted when the transaction commits. "Log shipping" forwards a "redo"-only log describing updates at the primary site to one or more "hot stand-by" secondary sites, where it drives continuous "redo" recovery to keep all database copies precisely synchronized. When the primary site fails, the secondary sites can provide data for its recovery. Better yet, the role of the primary site can instantly "fail over" to one of the secondary sites. Concurrent updates at multiple sites are complex and even read-only query processing at all sites requires some advanced techniques as discussed later (Section 5.7).

(5) In-memory databases in volatile memory and soon in nonvolatile memory dispense with the buffer pool, even if it has been a foundation of write-ahead logging. In environments without a buffer pool to hold dirty pages, the "do" action must not modify the persistent data. Instead, it must merely create a log record, preferably in byte-addressable nonvolatile memory. A "redo" action eventually applies the log record. With the recovery log also in nonvolatile memory, appending a log record imposes hardly any delay. Since each transaction needs to "see" its own updates, the thread that generates a log record in a "do" action should apply it to the data pages, for example, using the "redo" recovery method. Thus, a possible descriptive name for this design of write-ahead logging is "log shipping to self." A side benefit of this design is that "redo" code runs frequently, which improves the probability for good maintenance, testing, and tuning.

Multiple books cover concurrency control and recovery in database systems including B-tree indexes. Bernstein et al. [1987], Gray and Reuter [1992], Kumar and Hsu [1998], and Weikum and Vossen [2002] reflect the state-of-the-art at their time. Lomet [1998] and Mohan [1999] describe the industrial state-of-the-art of the late 1990s. They both reflect the transition from strictly physical, idempotent, non-logged "redo" and "undo" recovery actions to exactly-once application of recovery actions based on physical "redo" and logged, logical compensating updates rather than physical "undo" actions.

1.3. Overview

The following section reviews preliminaries and our assumptions about databases, B-tree indexes, transactions, and transaction implementation techniques. Section 3 distinguishes between traditional "user transactions" and implementation-specific "system transactions." Sections 4 and 5 describe techniques and optimizations for logging and for recovery, respectively, in the context of B-tree indexes. Finally, Section 6 outlines some possible future directions, whereupon Section 7 summarizes and concludes the survey.

2. PRELIMINARIES AND ASSUMPTIONS

Our assumptions here about databases, B-trees, transactions, failures, and recovery are meant to be simple and general. The discussion here often uses terminology of databases, but the techniques also apply to other contexts using transactional B-tree indexes. Some familiarity with B-trees is assumed and basic B-tree concepts are not explicitly repeated.

None of the following material should be surprising for readers familiar with database transactions and database indexes. Such readers may elect merely to skim or even to skip this section.

2.1. Databases

Databases contain tables, each of which is indexed by one or more B-tree indexes. The set of indexes for a table might change over time. This separation of logical database design (tables, rows, columns, primary keys, foreign keys, etc.) and physical database design (indexes, records, fields, sort order, partitions, etc.) is called physical data independence. In addition, data structures such as B-trees permit some freedom in the representation of their contents, that is, some independence between contents and structure. This additional freedom or separation is crucial for efficient locking, logging, and recovery of transactional B-trees.

Users and applications access and update databases through server software, the database management system. It is immaterial here whether this software runs within the application process or in a separate process or even in a separate computer. The server has its own state, for example, the current contents of the buffer pool or the set of locks held by active transactions. This state is not part of the database and vanishes when the server shuts down or fails (crashes). Recovery activities after a server failure include building new server state, for example, loading pages into the buffer pool. In today's software architectures, this server state is built completely from scratch during database recovery, that is, as required by the recovery process guided by the log describing changes in the database. Recovery of server state as it existed prior to a crash is incidental to recovery of the database and of the transactions active at the time of the crash.

Distributed systems usually manage concurrency control, logging, and recovery locally in each participating node. The exception is transaction completion using a distributed protocol such as two-phase commit [Gray 1978], which makes "precommitted" or "undecided" local components of distributed transactions subject to the decision by a central transaction coordinator. The discussion that follows focuses on a single node but the techniques obviously apply to distributed systems as well. In fact, some of the techniques discussed shortly might be particularly valuable in distributed systems, for example, early transaction processing (Section 5.5) and forward recovery (Section 5.6).

2.2. B-Tree Indexes

While there are many forms of indexes, the focus here is entirely on B-trees. The term "B-tree" more specifically means B^+-trees, that is, all data items are in the leaf nodes and key values in parent nodes and other ancestor nodes serve only as separator keys guiding searches, without carrying additional information about database contents.

Figure 1 illustrates a very simple B-tree with only 2 levels and only 3 leaf nodes. Each node is typically a page, for example, a disk page, such that the terms are often used synonymously. The leaf nodes in Figure 1 contain 2–3 leaf entries, that is, key values plus associated user-defined function information. B-trees enable search within large collections by multiple levels of branch nodes, not just the root node. Efficient search for the information associated with a key value exploits binary search

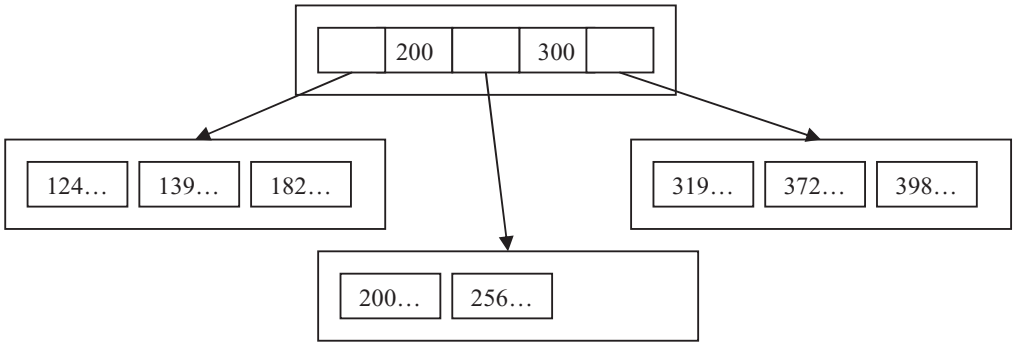

Fig. 1. A 2-level B-tree with root and 3 leaf nodes.

among separator keys and direct page access in the next B-tree level using the child pointers.

B-tree leaves contain 10s, 100s, or 1000s of data records and non-leaf B-tree nodes contain 100s or 1000s of child pointers and appropriate separator keys. An indirection vector near each page header variable-size records and separator keys; each slot in the indirection vector contains a byte offset within the page. Slot array and data space grow towards each other. Records are assigned to slots based on the sort order of the B-tree. While the location of record contents within the page may have no correlation to the sort order of the B-tree, the slots in the indirection vector are sorted by key values and thus enable efficient binary search within each B-tree node. Slots commonly contain the record length or, for efficient search, may contain some leading bytes of the key value (also known as poor man's normalized key) [Graefe and Larson 2001; Lomet 2001].

Leaf nodes and non-leaf nodes are stored as pages on some persistent block-access device such as a traditional disk or flash storage. This storage is fairly reliable but may fail ("media failure"). In addition, there is some device (or combination of devices) that is considered reliable. This "stable storage" is required for the recovery log. A correct implementation of logging and recovery can cope with transaction failures, media failures, and system failures inasmuch as it can rely on the stable storage.

An in-memory buffer pool enables immediate access to pages. Page replacement (victim selection) in the buffer pool might consider transaction boundaries but it typically does not. Thus, when a transaction commits, its modified ("dirty") pages may linger in the buffer pool and are not "forced" to storage when a transaction commits. Moreover, one transaction might "steal" a frame in the buffer pool from another transaction, even if the current contents are dirty and must be written back to disk. Härder and Reuter [1983] call this buffer pool policy "steal—no force."

Modifications of B-trees fall into two disjoint sets: contents changes insert and delete data entries in leaf nodes, whereas structural changes modify the set of nodes in a B-tree, for example, by splitting or merging nodes. Contents changes may introduce the need for structural changes; nonetheless, the structural change can happen as an independent preparatory action for the contents change. In fact, each structural change may itself proceed in multiple steps. For example, B^{link}-trees separate splitting a node into creation of a neighbor and posting of a separator key in the parent node [Lehman and Yao 1981], and Lomet and Salzberg [1992, 1997] generalize B^{link}-trees to permit "decomposing structure changes in an index tree into a sequence of atomic actions, each one leaving the tree well formed." This distinction between contents changes and structural changes also determines the types of transactions that implement them.

2.3. Database Users

In the present context, users may be humans engaged in an interaction with the data store or applications that invoke database functionality. Sometimes the word "user" may also refer to the software developer of such applications. Users initiate transactions, request transaction commit or abort, and run queries and updates within those transactions. All user requests execute in the context of transactions; if no transaction is explicitly initiated, one is created automatically for each statement or request. If it is possible that a single user runs multiple transactions concurrently, they do not share data any more than any other pair of transactions. For example, such transactions compete for locks like other transactions, with no privileged data sharing because the transactions pertain to the same user.

2.4. Transactions

Transactions are units of work that are atomic ("all or nothing"), preserve consistency (of database states), ensure isolation (from concurrent transactions), and guarantee durability even in cases of system failures immediately following transaction commit. Multiple transaction isolation levels have been defined; in the present context, serializability is assumed without the aid of versions or snapshots. Thus, locks are retained until the relevant transaction commits. Moreover, write-ahead logging is assumed as the chosen technique for failure atomicity and durability. Update-in-place is assumed unless otherwise noted.

The mapping between transactions and software threads can be complex. In parallel query processing and in parallel utilities, for example, many threads may serve a single transaction. On the other hand, a single thread may serve multiple transactions, for example, switch to another transaction when one transaction stalls due to an I/O operation or even an interaction with a user or application. Finally, multiple software threads may take turns serving a single-threaded transaction during its lifetime.

Traditional thinking applies transaction semantics only to database state, that is, the logical contents of the database and their physical representation on persistent storage, for example, disks. Transaction semantics could also be used to implement and recover server state, that is, the status of processes, threads, user transactions, connections, buffer pool, lock manager, etc. Durability in this case is limited to the duration of the server process. In other words, when the server process shuts down or restarts, the database state is retained across shut-down and restart, whereas all server state is reinitialized from scratch. This is not entirely true, however, in particular when one considers that the purpose of recovery using a transaction log is precisely to reestablish the state of active transactions such that they can be cleanly rolled back. A theory for applying transaction semantics to server state is an open research opportunity, in particular with respect to recovery after system failures.

2.5. Failure Classes

The traditional theory of failures and recovery considers three classes of failures that require log-based recovery. These are transaction failure, media failure, and system failure.

A transaction failure pertains to a single transaction only within a working system. A typical example is a deadlock or other failure during space or lock acquisition. Recovery requires undoing or compensating all changes in logical database contents and then committing the transaction. For this case, it is common that the log records of each transaction form a linked list. In this list, each log record points to the preceding one, because a list in this direction is easy to create and it is the direction required in transaction rollback.

A media failure pertains to storage such that the system (or server process) remains active and viable. Several transactions might need to roll back, but ignore all log records pertaining to the failed storage media. Recovery requires a backup plus reapplying relevant recovery logs. Transactions without effects, that is, aborted transactions, can be skipped in this process. The usual assumption is that such a failure is large, for example, an entire disk drive. Efficient recovery from small media failures, for example, individual blocks on a flash device, remains an opportunity for research and design.

A system failure is an abrupt and unplanned termination of the server process. Recovery from a system failure reinitializes all server state and then interprets as much of the recovery log as is available on stable storage. Transactions without a commit record on stable storage are aborted, that is, their effects undone or compensated. Compensations actions require "redo" logging of the compensation action. These log records prevent that the same compensation action is applied twice, for example, in case of a second system failure during recovery from an earlier system failure. Compensation actions never require "undo" logging. Once transaction rollback starts, the transaction is doomed and compensation will run to completion, if necessary during system recovery. Even partial rollback to a transaction savepoint never reverses a compensation action.

Any database access should begin with recovery. After an orderly shutdown, the last entry in the recovery log is a checkpoint record indicating no active transactions. In that case, recovery is instantaneous. Otherwise, the most recent checkpoint record guides and speeds up recovery after a system failure.

2.6. Write-Ahead Logging

A ship's logbook records all significant movements of the ship and events on board. The recovery log of a database or a file system records all significant updates [Gray 1978]. In the case of a file system, the log might cover only the metadata, for example, allocation and directory information, but usually not the contents of individual documents or data files. In the case of a database, the recovery log describes all changes such that the entire contents of the database can be recovered at any time and after any failure.

A recovery log contains at least three kinds of information or types of log records: "redo" information needed to provide durability, "undo" information in case a transaction cannot complete, and commit records to indicate that a transaction indeed completed. There might be additional types of log records, for example, information logged when a new transaction starts or when a transaction switches from forward processing to roll back and compensation of its earlier actions. Some additional types of log records are discussed shortly in the context of specific techniques and optimizations.

Usually "redo" and "undo" information is logged together and prior to modifying a database page in the buffer pool. Nonetheless, the "redo" part could be delayed as long as it is logged prior to transaction commit. The "undo" part could be also delayed. It must be written to stable storage before the dirty page in the buffer pool overwrites the prior page contents in the database. This requirement—logging before updating in place—has given rise to the name "Write-Ahead Logging" (WAL). With the large memory found in modern systems, the "undo" log record may be kept separate from "redo" log records and never be written to stable storage. This design requires retaining all dirty pages in memory until all appropriate transactions commit. Note that multiple transactions may modify the same page. Note also that large operations such as creation of a B-tree index might need special-case solutions.

Log records describing database changes are simply added to the current output buffer of the recovery log, to be written to stable storage when the buffer is full. Commit records must be written to stable storage before durability of the transaction can be guaranteed and the user's commit request can be acknowledged. In other words, the

output buffer must be flushed to the log before a transaction can be reported committed to the user or application.

If bandwidth of the log device is precious, writing half-full log pages is wasteful. Thus, many systems employ a technique commonly called group commit. It was pioneered in IMS/FastPath [Obermarck 1998]. Rather than forcing a log page immediately after appending a commit record, a small delay in the committing transaction might allow other transactions to fill the log page.

In order to enable fast recovery after a failure, another important type of log record is used. A checkpoint record saves the current state of transaction management, including undecided transactions, dirty pages in the buffer pool, etc., in order to provide a recent starting point for future crash recovery. For each undecided transaction, the checkpoint record points to the most recent log record. This information is commonly called a Log Sequence Number (LSN) although it often is a record identifier combining a page identifier and a slot number within the page. Early techniques for checkpoints required quiescence of transaction processing while all dirty pages in the buffer pool were written to disk, forcing a trade-off decision between fast recovery in case of a failure versus overhead during normal transaction processing. Modern techniques permit active transaction processing even while the checkpoint logic executes.

Since every change in a database is logged, log files can be quite large. Correspondingly, the required bandwidth for writing the recovery log to stable storage must be quite large. The problem is exacerbated by the size of log records and in particular their headers, which include transaction identifier, a reference to the previous log record by the same transaction (prior LSN by transaction), and perhaps a reference to the next log record to process during "undo" or transaction compensation (next undo LSN). Log records describing a change of a data page often include a reference to the previous log record pertaining to the same data page (prior LSN by page). Thus, in addition to reducing the amount of detail that must be logged, it can also be beneficial to reduce the number of log records. Unfortunately, there are various limits to such optimizations. For example, some optimizations for recovery performance require that a single log record must pertain to a single data page only.

2.7. Log-Based Recovery

Database recovery for all traditional failure classes requires a log of transactional updates. Modern techniques permit some carefully designed imprecision not permissible in earlier, strictly physical recovery. For example, log records may leave the placement of records within pages unspecified. Recovery from transaction failure nowadays eschews idempotent recovery actions and instead employs compensation [Mohan et al. 1992; Weikum and Schek 1992]. Rather than reversing a transaction's effects bit by bit, the database can be "updated back" to the original logical contents. Compensation actions are logged similar to the original update actions but there is no need to log "undo" information. The final "abort" record can be seen as committing a transaction with no logical change in the database.

Logging compensation actions permits guaranteed "exactly once" application of recovery actions, in contrast to the traditional "idempotent" recovery actions that would be applied during recovery whether or not a prior recovery attempt had already completed them. Crucial for the implementation of "exactly once" application of recovery actions is the LSN (Log Sequence Number) of the original log record and the page LSN. The page LSN is recorded in each page header and indicates the most recent log record applied to the page, whether the page is currently on disk or in the buffer pool in memory.

For recovery from media or system failure, a forward "redo" pass over the log starts from a recent checkpoint and ensures that all updates have been applied to

the database. It also collects information about incomplete transactions. Only those that require "undo" or compensation work are then reversed by following the backward chain of log records per transaction. The process may process one transaction at a time or all transactions and their log records may be merged using a simple priority queue.

These two main phases may be bracketed by an initial log analysis and a checkpoint after recovery is complete. Checkpoints immediately after and perhaps even during recovery are pragmatic in case of an additional failure; the cause for the initial failure may reoccur. Benefits of an initial log analysis without any "redo" or "undo" (compensation) actions may include simplified development or testing of all required software, checkpointing the results of the analysis in order to speed recovery from a system failure during recovery, and an estimate of recovery effort and time (enabling an accurate progress indicator if desired).

3. TYPES OF TRANSACTIONS

The following discussion relies on a distinction between logical database contents and physical representation. For example, a specific key range can be represented in a B-tree index by a single full node or by two nodes, each about half full. Transactions that modify only the representation have no effect on query results; their effects are internal to the database system and they may be called "system transactions." B-tree indexes and their implementations are prototypical for the use and value of system transactions, because B-trees can capture the same logical information with multiple physical representations.

This separation of logical index contents and physical data structure is exploited for concurrency control and for logging and recovery. In concurrency control, locks protect logical index contents and are held until transaction completion (in serializable transactions), whereas latches are lightweight semaphores protecting in-memory data structures and are held only briefly during critical sections [Graefe 2010]. For efficient logging, system transactions changing the representation of a B-tree permit several optimizations in log volume and commit overhead. During recovery, system transactions changing the representation of a B-tree may be treated differently than user transactions changing B-tree contents.

The present section explores the differences between user transactions and system transactions in more depth. The subsequent two sections cover logging and recovery in detail.

3.1. User Transactions

Within this survey, traditional transactions are called user transactions. Users request them directly or through applications, expect correct results, and care about transactional properties such as isolation from other transactions and durability in case of a subsequent failure. With respect to logging and recovery, transactional reversal ("undo") must be possible until the user or application request that the transaction be committed; after transaction commit, all effects of the transaction must be ensured even if primary storage is lost due to a failure ("redo").

The details of transaction "undo" are immaterial to a user. For example, a transactional insertion may be reversed by erasing the newly inserted record or by leaving it in place but marking it as logically invalid (also known as "pseudo-deleted"). Similarly, if an insertion forced splitting a B-tree node, transaction "undo" may or may not reverse the split operation. For simplicity and performance, it may be advantageous to omit some operations during transaction "undo."

The ARIES design [Mohan et al. 1992] for logging and recovery enables transaction reversal on a more logical level than prior schemes. Instead of reversing each physical page to its state prior, the "compensation" actions central to ARIES recovery can be

described as "updating back" and "committing nothing." For example, a deletion reverses an insertion, and a deletion might erase a record or leave it in place marked logically invalid. This sequence of insertion and deletion can be committed as it leaves the database logically unchanged, even if it has been changed in its physical representation.

Multilevel transactions [Weikum 1991; Weikum and Schek 1992] generalize these ideas further. Each transaction executes at one of several levels of abstraction. Each transaction is complete with all-or-nothing semantics, commit, etc., including both concurrency control and recovery. Each higher-level transaction may invoke as many lower-level transactions as required. When a higher-level transaction needs to roll back, it invokes new lower-level transactions to revert its effects. While general and applicable to multiple levels, two levels suffice in most implementations; the lower level is discussed as system transactions here following some earlier work [Graefe 2004; Sun et al. 2005].

3.2. System Transactions

System transactions modify the physical representation of a database but they must not affect the logical database contents. In other words, users and applications do not care about system transactions, whether they are committed or aborted, when and how they are invoked, etc. Users care only inasmuch as system transactions enable efficient and successful completion of user transactions. For example, if a user transaction requests an insertion but the appropriate physical location such as a B-tree leaf lacks sufficient free space, then the user transaction may invoke a system transaction to create free space by splitting the leaf. After the system transaction logs its changes and commits, the user transaction can resume and succeed.

When a user transaction invokes a system transaction, it waits for the system transaction to complete. Thus, the system transaction can run within the same software thread. The invocation overhead is therefore negligible. Moreover, if concurrency control for data structures (also known as latching) depends on threads, a system transaction can immediately use latched data structures just like the user transaction. For example, if a user transaction has latched a B-tree leaf page with insufficient free space and invoked a system transaction to remedy the situation, there is no need to acquire the same latch again.

With respect to locking, system transactions do not modify the logical database contents and therefore have no need to protect them by means of locks [Graefe 2010]. Thus, a system transaction can modify a B-tree leaf even while the invoking user transaction or even another user transaction hold locks on some of the keys within the leaf page. A system transaction may need to inspect the lock manager's hash table for conflicting locks, however. For example, if a system transaction is invoked to reclaim the space occupied by a record left logically invalid by a prior user transaction, the system transaction must ensure that no other user transaction holds a lock on that record or its key value (more details in Section 4.2).

A system transaction must log its database changes, reverse them upon failure, and insert a commit record into the recovery log upon completion. There is no need, however, to force the commit record to stable storage upon commit. This is because system transactions are limited to representation changes, with no change in logical database contents. Thus, users and applications never rely on completion and durability of system transactions for correct query results or logical database contents. Forcing their commit records to stable storage would render small system transactions unacceptably expensive and practically useless; avoiding this expense permits efficient database implementations with liberal use of small system transactions. In fact, there are cases (some discussed shortly) where a system transaction may invoke another system transaction, that is, system transactions may be nested.

	User transactions	System transactions
Invocation source	User requests	System-internal logic
Database effects	Logical database contents	Physical data structure
Data location	Database or buffer pool	In-memory page images
Parallelism	Multiple threads possible	Single thread
Invocation overhead	New thread	Same thread
Locks	Acquire and retain	Test for conflicting locks
Commit overhead	Force log to stable storage	No forcing
Logging	Full "redo" and "undo"	Omit "undo" in many cases
Recovery	Backward	Forward or backward

Fig. 2. User transactions and system transactions.

Nonetheless, a subsequent user transaction may rely on a change affected by a system transaction. For example, after a system transaction has split a B-tree leaf and thus has affected page identifiers and slot numbers, a user transaction's log record may refer to page identifiers and slots numbers within a page. In this case, the commit record of the system transaction will be written to stable storage prior to or together with the commit record of the user transaction. Should a failure occur such that the user transaction requires "redo" based on the recovery log, then both transactions are fully logged and committed.

The commit record of a system transaction might seem like overhead compared to other designs that maintain the database representation as part of user transactions. If, however, those other designs permit subsequent user transactions to rely on those representation changes and perhaps even change them further, the first representation change requires protection from "undo" if the original user transaction fails. ARIES employs nested top-level actions, dummy log records, and special backward pointers in the recovery log ("next undo LSN"). The dummy log record at the end of a nested top-level action is quite similar in function and overhead to a commit record of a system transaction. Arguably, the concept of a separate transaction seems simpler and a more principled design than a special backward pointer within the recovery log.

When a user transaction aborts, it applies "undo" (compensation) actions for all its logical database changes and then commits. With respect to logical database changes, an aborted user transaction "commits nothing." It can do so without forcing its commit record to stable storage. The transaction has not modified the logical database contents although it may have changed the physical representation of the database. For example, it may have inserted a record and then merely marked it logically invalid. An aborted user transaction is akin to a system transaction that also has no need to force its commit record to stable storage.

In some cases, a system transaction may invoke another system transaction. For example, a system transaction may split a leaf node in a B-tree, which requires posting a separator key in the parent node. If the parent node needs splitting, too, this may be executed as another, separate system transaction. It can be argued that a single system transaction would suffice. This is true, but some optimizations for system transactions would be harder to realize, for example, fusing log records (Section 4.3) or forward recovery (Section 5.6).

Figure 2 summarizes the differences between user transactions and system transactions. Most of the differences have been discussed earlier. The difference in the acquisition and retention of locks is covered elsewhere [Graefe 2010]. Section 4.3 covers omission of "undo" information and Section 5.6 introduces forward recovery of system transactions.

Many descriptions of techniques for concurrency control and recovery focus on updates and omit issues of creation and removal of logical or physical data items. For example, the traditional lock matrix comprises read and write locks (or shared and exclusive locks), but it usually is not clear or obvious which object to lock when a new object is created. Similarly, serializability theory usually explains reads and writes, omitting object creation and removal. This omission seems to be the reason for difficulties in explaining and understanding key range locking in B-trees, because the crucial aspect is locking gaps between existing key values in order to ensure serializability when key values are inserted or deleted. The distinction between user transactions and system transactions enables simple and concise explanations. User transactions update objects, which includes marking a record as logically deleted (Section 4.2), whereas system transactions create and remove objects, for example, a B-tree node or a logically deleted record.

Contrasting system transactions as described before from "top-level actions" in ARIES yields the following similarities and differences. Both focus on database representation rather than user-visible contents changes. System transactions require their data in memory, including in-memory images of disk pages, whereas "top-level actions" have no such requirement, perhaps because it would not have been acceptable 20 years ago when ARIES was designed. ARIES introduces "instant locks" whereas system transactions verify an appropriate server state, which may include absence of conflicting locks. In other words, system transactions retain latches to the end of a critical section and permit retaining all locks to the end of a transaction. Finally, ARIES top-level actions require dummy log records and "next undo LSN" backward pointers to skip log records correctly when "undo"ing (compensating) a user transaction whereas system transactions, by virtue of being separate transactions, require neither dummy log records nor backward pointers. In summary, system transactions and ARIES top-level actions are similar in intent and scope but quite different in their details, with system transactions deriving most of their properties and implementation requirements from the fact that they are independent transactions.

3.3. Utility Transactions

Most database systems include a number of utility operations that may be invoked explicitly by user request, implicitly as side-effect of other utilities or of query processing, and automatically by external tools such as an index tuning application. These utilities may implement index creation and removal, backup and recovery operations, compression and defragmentation, incremental maintenance of materialized views, etc. Some of these utilities are complex and some can run for long times. For example, creation of a new clustered index for a table might require replacing existing nonclustered indexes with new ones that refer to new record locations in the new clustered index, followed by removal of the old clustered and nonclustered indexes. Online index operations permit concurrent query and updates; online creation of a new clustered index requires various techniques for concurrency control and recovery in B-tree indexes.

Transactions used in utilities are not really a new or additional kind of transactions. Instead, utilities usually run as a user transaction invoking system transactions that themselves may invoke further system transactions. Transactions in utility operations can nicely illustrate how transactions interact. For example, a user transaction might request removal of a nonclustered index. The traditional execution of this request may complete the request in a single user transaction. Freeing millions of pages implies millions of updates in the allocation information as well as writing appropriate log records. In the best case, this merely takes a fairly long time; in the worst case, index removal may fail due to lack of log space.

An alternative technique for index removal lets the user transaction merely add an appropriate tag to the catalog entry for the index. Subsequent system transactions free the storage space and eventually erase the catalog entry. In other words, index removal is split into a semantic change implemented by the user transaction and the appropriate representation changes implemented by system transactions. These system transactions may run synchronously and asynchronously to the user transaction. In other words, control might return to the user instantly. If the index is large, incremental space deallocation might employ many small system transactions. In the case of a B-tree index, these small operations might be organized using the tree structure of the index.

Since many utilities move a large amount of database data, several important optimizations of logging and recovery are specific to utilities such as index creation. For example, creation of a new nonclustered index may log every record in the index. In order to save on the number of log record headers, the operation may log entire new pages. The most optimized technique, however, does not log the new index entries at all, logs basically only space allocation for the new index, and relies on other techniques for recovery in case of a media or system failure. These techniques are discussed in Section 4.5.

Some utility operations, however, require user transactions rather than system transactions. This is the case if multiple recovery logs are involved, for example, when moving records between multiple nodes in a shared-nothing database management system. Logic very similar to two-phase commit is required, including forcing precommit log records and the final log record to stable storage. With respect to concurrency control (locking like user transactions versus latching like system transactions), the deciding question is whether multiple threads participate in executing an operation; with respect to logging (forced log write upon commit like user transactions versus commit without forcing the log like system transactions), the deciding question is whether multiple recovery logs participate in the commit protocol.

3.4. Parallelism

Multithreaded execution of queries, updates, and utilities also illustrates some useful concepts in concurrency control and recovery for B-trees. With respect to concurrency control, data structures require concurrency control among concurrent threads of the same user transaction (latching), but database contents requires concurrency control only among distinct user transaction (locking). In fact, multiple threads of the same parallel query execution plan must mutually "see" their updates. With respect to logging and recovery, multiple threads do not require separate transactions or separate streams of log records. They do, however, require concurrency control (latching) for the log manager just as for lock manager, buffer pool, and all other shared in-memory data structures.

Each execution thread might invoke its own system transactions, for example, in order to reclaim fragmented space within pages with variable-size records or to split B-tree pages. Each of these system transactions may commit on its own, whether or not the various threads access the same or different B-trees. Thus, a single user transaction might perform multiple system transactions concurrently. These system transactions may rely on each other's effects, without danger of cascading aborts. For example, one thread might split a B-tree node and another thread of the same transaction might split the same node again.

Parallelism within a system transaction is conceivable but rarely employed. System transactions usually modify in-memory copies of database contents, for example, B-tree nodes, and they are more comparable to short critical sections than to long-running database transactions. It usually is possible to break changes of data structures into

multiple steps such that each step transforms one consistent and accurate representation of database contents into another. Thus, the tendency is to employ many small system transactions rather than large system transactions. The low overhead of system transactions (execution in the same thread as the user thread and no need to force the commit record to stable storage) also encourages use of many small rather than fewer large system transactions.

3.5. Checkpoints

Checkpoints are a long-known technique that shortens recovery time after a system failure. The distinction between user transactions and system transactions does not affect checkpoints. Both types of transactions may be active during a checkpoint, both types of transactions may start or finish during a checkpoint interval, etc. The only difference between the transaction types is that user transactions force their commit records to stable storage whereas system transactions have their commit record forced to stable storage by a checkpoint or a committing user transaction, whichever comes first.

Since system transactions are usually very short, they will often commit rather than remain active during a checkpoint interval. Moreover, checkpoints must acquire latches on the appropriate in-memory data structures in order to write consistent data to the recovery log as part of the checkpoint information. Since system transactions acquire latches on the data structures they modify, checkpoints usually include very few uncommitted system transactions.

3.6. Recovery

Recovery of system transactions is similar to recovery of user transactions. The basic rules remain that if a transaction is committed, its effects are redone; and if it is not committed, its effects are undone or compensated. With system transactions, however, there is additional flexibility that justifies why there is no need to force commit records of system transactions to stable storage.

Redoing a committed system transaction is required only if subsequent transactions depend on it. For example, if a system transaction has split a B-tree node but no subsequent transaction has performed further updates to the two resulting nodes or their parent node, then it is not required to redo the node split. Instead, the transaction could also be undone or compensated. Of course, all-or-nothing failure atomicity is required, so each system transaction must be recovered one way or the other.

3.7. Summary

In summary, B-trees enable a low-level form of physical data independence, that is, separation between the logical contents of a B-tree and the precise physical representation of database contents. Two logically equivalent B-tree indexes might differ in the number of nodes, in the free space within nodes, etc. This separation enables two types of transactions: those that modify index contents and those that merely modify the representation of contents. The latter kind is called system transactions here. They may run within the same thread as the invoking user transaction, they may access the same in-memory data structures as the invoking transaction (with respect to latching), they record their database changes in the recovery log just like user transactions, yet they can commit without forcing the commit record to stable storage. Invocation within the same thread and committing without forcing a write operation to stable storage ensure that system transactions are inexpensive and thus may be used liberally in the implementation of database management systems in general and of B-tree indexes specifically.

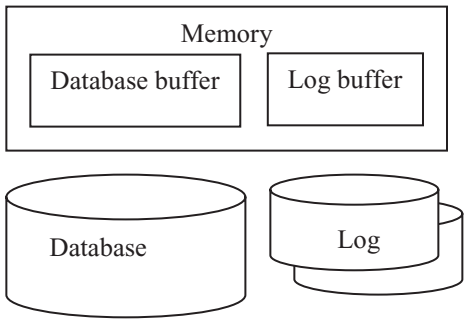

Fig. 3. System elements.

4. LOGGING TECHNIQUES

System transactions enable changes in database representation, in particular B-tree structures, without the overheads of starting a new thread or forcing a log buffer to stable storage. The remaining optimization opportunities for logging B-tree operations focus on reducing the number of log records and avoiding unnecessary detail in the log records. The present section surveys a number of techniques that are particularly well suited for B-trees, most of them already proven in multiple implementations. At the end of the present section, an example application (logging during creation and optimization of partitioned B-trees) illustrates how these logging techniques and optimizations complement each other. General optimizations for logging, for example, group commit [Helland et al. 1987; Obermarck 1998], are omitted due to the focus here on techniques particularly useful for B-tree indexes, although most of the techniques described next are applicable or can be adapted to storage structures other than B-trees.

Figure 3 illustrates the basic elements of a prototypical database system. The database is stored on direct-access storage such as a traditional disk drive. A buffer pool in memory holds images of database pages while they are in active use plus a limited time thereafter. The recovery log is saved on stable storage, shown as a mirrored disk drive. Pages of the recovery log are filled in the log buffer and may be retained there for a limited time even after they have been saved.

4.1. Physiological Logging

Logging and recovery can focus on two levels of abstraction, the logical database contents or its physical representation. Logical logging and recovery focuses on logical database contents and operations that modify it. Physical logging and recovery strictly focuses on the storage representation. As both approaches promise advantages over the other, a compromise approach attempts to realize the advantages of both.

In physical logging, changes to data pages and their data record are logged by their byte offsets and byte-for-byte copies of the data. A log record for a data change might even contain two complete page images, that is, copies of the data page before and after the modifications. An obvious optimization is to include copies of only the affected byte range. Compression using differences or run-length encoding is also possible. For example, insertion or deletion of a single record might log the record contents plus changes in the page header and the slot array. For free space reclamation within a data page (also known as page compaction), the complete page must be logged, with the free space possibly zeroed and then compressed by run-length encoding. Note that a record insertion or update to a variable-size field might require a page compaction. Early transactional databases relied on physical logging.

In logical logging, page and byte locations do not matter; only the logical operations are recorded in the recovery log. For example, insertion and deletion are logged but not their precise locations. In B-tree indexes, merely the unique key value and the record contents may be logged. The unique key value implies the B-tree leaf and the position within the page. Neither the page identifier of the affected B-tree leaf nor the byte offset within that page are recorded. Page compaction has no logical meaning and thus no log record is required. Thus, log volume may be reduced substantially compared to physical logging. Recovery from a logical log, however, is rather complex and possibly inefficient. Commercial production database systems have not used logical logging.

Physiological logging strikes a balance between logical and physical logging. It is the method of choice in modern database systems. Gray and Reuter [1992] describe physiological logging as "physical to a page, logical within a page." Log records refer to a data record by its page identifier and its slot number. Thus, data references in log records are physical with respect to pages but logical with respect to records. Byte offsets are not included in the recovery log. Thus, when a data record moves within its data page, for example, during page compaction, there is no effect on log records. Even if page compaction is not logged at all, efficient and correct recovery is possible if the recovery code invokes compaction when required. Thus, the log record for an insertion contains only the new record, even if the insertion requires a page compaction.

In B-tree pages, the unique key value substitutes for the slot number, because it implies the slot number in a sorted B-tree page. Thus, insertion and deletion need to log only the page identifier plus the B-tree record including its unique key value. There is no need to log the slot number, the shifting of slot numbers when inserting or deleting within a sorted slot array, or page compaction. Thus, physiological logging for B-tree indexes produces a log volume very similar to logging for unsorted heap files.

As physiological logging refers to pages directly by the page identifier but to records only indirectly by their slot number or only by their unique key value, and recovery from a system or media failure reapplies updates to the same pages as during original transaction processing. The logical contents of each page after "redo" recovery are the same as at the point of failure; the physical organization of that contents might be different, however. Due to the specific page references, "redo" and "undo" (compensation) actions can be very efficient with direct I/O operations, yet the recovery log need not record details of data movement within a page, for example, page compaction in a B-tree node.

4.2. Ghost Records

For more reliable and efficient user transactions, a commonly used technique employs ghost records, also known as pseudo-deleted records or simply invalid records. Each record includes a bit for this purpose, often called the "valid bit" or the "ghost bit." Query processing ignores records marked invalid. User transactions delete a record logically by marking it invalid, leaving it to asynchronous system transactions to reclaim the space.

There are three principal advantages of ghost records in B-tree indexes. First, they simplify concurrency control. For example, in B-trees, ghost records participate in key range locking just like valid records [Graefe 2010]. A user transaction can turn a valid record into a ghost record by merely locking the key value, leaving the intervals between neighboring key values entirely unlocked. Second, ghost records ensure reliable rollback of user transactions. Specifically, a record's data space is retained and locked until the user transaction commits. Transaction rollback can be fast and simple, with no need for space allocation, possible node splits, etc. Third, ghost records reduce the log volume in user transactions. A user transaction merely logs toggling a bit rather than the "before" image of an entire data record. A small disadvantage is that another

transaction and its log records are required for space reclamation, unless a subsequent insertion reuses the record and its key value.

Once a ghost record is committed and unlocked, only its key value remains meaningful, whereas all other fields are logically NULL. Thus, space reclamation by removal of a ghost record from a B-tree page needs to log at most its key value, or typically only the slot number in the page. In a clustered index, the size of the key value is often a small fraction of the record size. Thus, even if an additional transaction is needed, overall log volume might be reduced due to ghost records and system transactions.

In order to prevent duplicate key values in a B-tree leaf, a logical insertion must reuse a ghost record with the same key if such a ghost record exists. In fact, a user transaction can simplify its locking requirements by invoking a system transaction to create an appropriate ghost record if none exists [Graefe 2010]. Insertion by marking a ghost record valid does not need to log the old record contents, only the new values. The new key value is equal to the old key value and does not need to be logged if records are identified by their slot numbers in the recovery log. Thus, the log volume for insertion by turning a ghost record valid is no larger than for insertion by creation of a new record.

Page compaction, by default, removes all ghost records, except those currently locked by active user transactions. A single log record may capture a list or range of removed ghost records. Ghost records currently locked by active transactions are protected from removal because doing so modifies or even annuls the semantics of those locks. Therefore, it is not sufficient to log merely the fact that a page compaction has modified the page. An explicit list of removed ghost records is also needed unless, indeed, all ghost records could be removed. Otherwise, if the set of ghost records removed is in doubt during recovery from a media or system failure, a page compaction during recovery might erase a different set of ghost records from the original page compaction, invalidating subsequent log records and their references to data records by slot number.

In addition to the benefits with respect to concurrency control, logging, and recovery, ghost records are also useful in nontransactional B-trees. Since the indirection vector is sorted on key values, insertion of a new record requires that, on average, half the indirection vector must be shifted by one position. Removal of a record carries similar overhead. Logical deletion by turning a valid record into a ghost avoids this overhead. If a page compaction removes multiple ghost records in a single step, the overall overhead is lower. Just as importantly, however, an insertion might reduce the required amount of shifting by erasing a ghost record. In fact, deliberate insertion of ghost records can reduce the overhead of record insertions, very similar to the effect of gaps in library sort [Bender et al. 2006; Bender and Hu 2007]. Finally, careful insertion of ghost records can enable efficient interpolation search even in B-tree pages in which interpolation search would be inefficient without them [Graefe 2006].

A single log record should suffice to describe which ghost records have been inserted into a B-tree page. Thus, the log volume due to insertion of ghost records should be relatively small. The same is true when multiple ghost records are appended to a B-tree page in anticipation of future insertions. For example, consider a B-tree index on invoice numbers with very predictable future values. Rather than creating one record at a time, prepopulating a page with an appropriate ghost record and key values is beneficial for both concurrency control (eliminating a hot spot) and logging (a single log record describing many insertions).

4.3. Fusing Log Records

Ghost records and the system transactions to remove them permit a further optimization. In the worst case, such a system transaction requires three log records: transaction start, removal of the ghost record, and transaction commit. Since this pattern of log

records is frequent in database systems with ghost records in B-trees, these log records can be fused into a single one. In the least, this reduces the number of log records and thus saves on log record headers.

In addition, because the log records for the database modification and transaction commit are actually the same, there is no possibility for a failure between database modification and transaction commit. Thus, there is no possibility that the database modification might require "undo" or compensation actions, and thus no need to log "undo" information in the recovery log. While traditional record deletion requires logging the "before" image of the old record, neither turning a valid record into a ghost record nor removing the ghost record requires the "before" image. For large B-tree records, for example, in a clustered index of a database table, this cuts the log volume for record deletion to a fraction compared to the traditional technique.

The basic idea (fusing log records for database modifications with the corresponding commit record) may also apply to operations other than ghost removal. Since user transactions commit only upon explicit request by the user or application, the technique seems limited to system transactions. Since the savings pertain to "undo" information, the technique seems more valuable for deletions than for insertions. Thus, ghost removal in B-trees seems to be the primary application. If, however, a log record may refer to multiple data pages (in spite of the possible complexity during recovery), fusing multiple log records with the corresponding commit record is useful more broadly. For example, load balancing between two B-tree nodes affects three pages, because the separator key in the parent node must be updated. Nonetheless, a single log record can describe the operation very compactly. In the ideal case, this log record does not include any record contents. In order to prevent any danger of data loss, however, the following additional technique is required.

4.4. Careful Write Ordering

While fusing log records reduces the need for detailed "undo" logging, other techniques reduce the need for detailed "redo" logging. One such technique is careful write ordering Lomet and Tuttle [1995, 2003]. It is particularly useful while restructuring B-tree nodes and their contents. Its goal is to avoid logging individual records and their contents, instead only logging the operation and its parameters. Such log records contain only a few bytes (in addition to the log record header) rather than half or full pages.

When data records move from one page to another, for example, when a B-tree node splits, the source page (specifically, its old image still on disk) can serve as backup for the destination page until the source page is overwritten. This backup must be preserved until no longer required. Thus, the destination page must be written successfully to the database before the source page may be written. In addition to node splits, other B-tree operations that move records from one page to another and thus may benefit from careful ordering of write actions include load balancing between nodes, node migration, and defragmentation of indexes.

This ordering dependency between dirty data pages in the buffer pool is different from the ordering dependency at the foundation of write-ahead logging. In the case of logging, the write order of log pages is strictly sequential. Thus, progress can be expressed by a single page identifier (within the recovery log), also known as the "high water mark" in write-ahead logging [Gray 1978]. Dirty data pages with an LSN (page LSN) higher than the high water mark must overwrite pages in the database.

A dependency between a pair of dirty data pages in the buffer pool requires a different implementation method, as a single high water mark will not suffice. A useful restriction lets each page depend on only one other page. For example, load balancing between two sibling nodes in a B-tree moves some entries from a source (the old, overflowing page) to a destination (the newly allocated page) and thus renders both

pages dirty in the buffer pool. Unless these moves are logged in detail, the destination page now depends on the source page and the source page (with fewer entries) must not overwrite its prior version in the database (with all entries). In order to capture this restriction, the buffer pool descriptor of the destination page may refer the source page and the source page may employ reference counting. After the destination page has been written successfully, its dependency on the source page ends and the counter for the source page is decremented. Page replacement in the buffer pool must skip over pages with nonzero counters.

Alternatively, the buffer pool descriptor of the source page may reference the destination page. When the source page is chosen as replacement victim in the buffer pool, the destination page is forced to the database first.

If the "right edge" (highest key values) of a B-tree is a hot spot for insertions, that is, if rapid insertions append ever-increasing key values, the current right-most leaf is split repeatedly. This situation permits reducing the log volume with careful write ordering most of the time but probably not all the time. Otherwise, the recently added leaf pages form a chain of dependencies; the last leaf node is not yet due for replacement, the prior leaf node must not be written until the last leaf node has, etc. In the extreme case, such a chain can create performance problems in checkpoints. Load operations can avoid those problems by fully logging some of the leaf splits or by avoiding leaf splits altogether. Instead of filling the right-most leaf node entirely and then splitting it, load operations fill the right-most leaf in a B-tree only to the desired degree (e.g., 90%) and then start a new leaf without moving any records from the prior to the new right-most leaf.

As an aside, IMS Fast Path includes the message buffer in this scheme. It ensures that the message acknowledging a transaction commit is retained in the communication buffer and cannot be sent from the server to the user until the commit record has been written to the recovery log on stable storage. Thus, the user or application cannot receive a message acknowledging successful transaction commit until the transaction is truly durable [Obermarck 1998]. A timer ensures that no transaction suffers an undue delay; this seems to be the earliest form of group commit.

4.5. Minimal Logging

Without special treatment, each creation and removal of an index results in a substantial growth of the transaction log. Thus, many vendors have implemented non-logged index operations. The term "non-logged" is not really accurate since both catalog updates and space allocation are logged. The index contents, however, are not logged. Alternative names are "minimal logging" and "allocation-only logging." In "bulk logging," record contents is logged but in entire pages, which saves log volume by reducing the number of log record headers.

When dropping an index, minimal logging requires that the index remain unchanged until the index removal is committed. The index is merely marked as obsolete in the catalogs, the requirement for an appropriate task is logged, and the actual task of merging the allocation information of the index into the global free space map is executed asynchronously. This task may run immediately after the index removal is committed, during times of low system load, when free space is running low in the global free space map, or some other heuristic.

For index creation, minimally logged operations rely on the "force" buffer pool management [Härder and Reuter 1983] during commit processing, that is, on forcing a transaction's dirty data pages from the buffer pool to storage before committing the transaction. If a system failure occurs before index creation is completed and committed, recovery can be accomplished by reversing all space allocation and removing the catalog entry. If index creation completes prior to a system failure, all index pages have been forced to storage. Thus, redo after a system failure is not required.

After a media failure, redo during media recovery must employ the sequence of page allocation actions described in the recovery log to recreate precisely the same index, with the same separator keys in the same parent pages, etc. This precision is required if subsequent user transactions modify the index and their log records reference index entries by page identifier and slot number.

Creation of a B-tree index usually requires a sort operation. Large indexes require an external merge sort with run files on disk. Database indexes can be larger than the disk space for reserved temporary data; in that case, the sort can use the "target space" of the future index. Minimal logging, sorting in the target space, and media recovery interact. In particular, the sort operation during recovery of the index must run with the same parameters and ensure the same sequence of allocation and deallocation actions as the sort operation during the original index creation. If the sort algorithm supports growing and shrinking its memory allocation dynamically based on memory contention, this functionality might be disabled during minimally logged index creation.

4.6. Pulling the Concepts Together: Logging in Partitioned B-Trees

An example might illustrate how these logging techniques and optimizations combine to permit efficient B-tree operations. This example focuses on partitioned B-trees including their creation and their incremental optimization. A partitioned B-tree is a standard B-tree structure with lightweight partitioning within the B-tree. Partitioned B-trees can serve as a good example for logging techniques and logging optimizations because at first sight the cost of logging creation and optimization of partitioned B-trees might seem prohibitive.

The essence of partitioned B-trees [Graefe 2003][1] is an artificial leading key field to a B-tree index. If all records contain the same single value in this leading B-tree field, which is the most common and most desirable state, the B-tree index is rather like a traditional B-tree index. If multiple values are present at any point in time, which is only a transient state in all but a few techniques, they effectively partition the set of index entries. Creation and removal of partitions requires merely insertion and removal of B-tree entries, with no changes in the database metadata (catalogs). This form of partitioning is orthogonal to partitioning across multiple nodes in a computer network or across multiple storage devices within such a node.

It is rather surprising how many problems this simple technique can help address in a database management product and its real-world usage, including the following. First, partitioned B-trees permit putting all runs in an external merge sort into a single B-tree (with the run number as artificial leading key field), which in turn permits improvements to asynchronous read-ahead and to adaptive memory usage. Second, partitioned B-trees substantially reduce, by at least a factor of two, the wait time until a newly created index is available for query answering. Until the B-tree is fully optimized by merging the initial partitions, queries must search all partitions, which is slower than query processing with a fully optimized B-tree but usually faster than scanning all data without any indexing. Third, when adding a large amount of data to a large, fully indexed table, partitioned B-trees resolve the dilemma between dropping all indexes and updating all indexes one record at a time. The former requires rebuilding all indexes prior to subsequent query processing; the latter implies many random insertions, poor performance, a large log volume, and a large incremental backup. In this usage, a partitioned B-tree is reminiscent of a log-structured merge-tree [O'Neil et al. 1996] but uses only standard implementation techniques, that is, a B-tree and a buffer pool rather than multiple storage structures and special recovery logic.

[1]This description of partitioned B-trees and their benefits is from Graefe [2003].

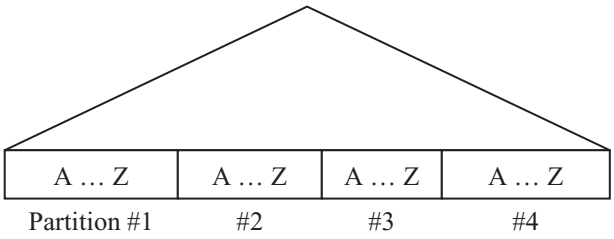

Fig. 4. A partitioned B-tree.

Figure 4 illustrates a partitioned B-tree (the root node at the top and leaves along the bottom) with partitions (each a box indicating key range A. . .Z) identified by an artificial leading key field in each record (shown as Partition #). A query (e.g.,. "between 'R' and 'T"") enumerates the partitions (here, 1. . .4) and searches each one. In the most naïve implementation, a root-to-leaf probe is required to find the next actual value in the artificial leading key field and another probe is required to apply the query predicate within the partition [Leslie et al. 1995]. The number of probes can be reduced to one per partition, and the cost per probe can be optimized to less than a full root-to-leaf search.

Initial index creation requires only run generation, whereas traditional index creation requires run generation followed by merging. Each run is a partition within the partitioned B-tree, with the run identifier as artificial leading key field. Optimization of a B-tree with many partitions requires reorganization by merging multiple partitions into one. Merging can be eager or lazy. Eager or immediate merging attempts to complete index optimization as quickly as possible, similar to traditional index creation. Lazy or on-demand merging focuses on only those key ranges required for an actual query.

With respect to logging, run generation can exploit minimal logging just like traditional index creation. Thus, initial creation of a partitioned B-tree index can be as fast as extracting future index entries, sorting them in memory into runs, and saving them in permanent storage just once. Saving the new index entries a second time, namely in the recovery log, is not required.

After run generation, merging combines all index entries into a single partition, if necessary in multiple merge steps. Moving records from one partition to another does not modify the logical contents of the B-tree. Thus, user transactions are not required. Instead, a large merge step may invoke many small system transactions, including all their optimizations for threading, concurrency control, and commit processing.

Minimizing the log volume during merging is an important optimization and requirement for the practical use of partitioned B-trees. As the merge logic moves records from the merge input runs to the merge output run, both deletion and insertion require logging optimizations.

Insertions into the merge destination is like run generation in many respects. Thus, the logging requirements are similar: page allocations must be logged but the index contents need not be logged if the pages are written to permanent storage prior to transaction commit. Careful write ordering permits saving the merge output without logging record contents.

Deletions may apply to entire leaf pages and to key ranges at the ends of a merge range. Removal of an entire leaf page can fuse all its actions into a single log record, omitting all "undo" information. In a special case of careful write ordering, commitment of the page removal must wait until the appropriate pages in the merge output are on permanent storage.

Deleting all records in a key range within a leaf page may turn these records into ghost records. In physiological logging, a single log record suffices for each affected leaf page. This log record must indicate the key range or the range of slot numbers within the page. When space reclamation eventually removes these ghost records, again a single log record suffices for each B-tree leaf, again omitting all record contents. Thus, allocation-only logging applies to both run generation and merging, within merge steps to both insertions and deletions, and for deletions to both removal of entire pages and deletion of key ranges.

4.7. Summary

In summary, numerous techniques can reduce space and bandwidth needs for the log device while logging transactional changes in B-tree indexes. Many of them depend on the separation of the logical database contents and their physical representation, which enables the distinction of user transactions and system transactions. It is not sufficient to optimize logging in only one type of operation. Consequently, the techniques described earlier pertain to individual B-tree entries (e.g., ghost records), B-tree structure (e.g., careful write ordering), and B-tree utilities (e.g., minimal logging during index creation). Combining most of the techniques discussed before, merging partitions in a partitioned B-tree can be transactional without ever logging the contents of the merged records.

5. RECOVERY TECHNIQUES

While the preceding section focuses on writing the recovery log in preparation of a possible need for recovery, the present section focuses on reading the recovery log in order to recover after a failure. These recovery techniques cover all three failure classes commonly considered in database recovery.

Recovery from a media failure starts with a backup copy and then applies the most recent database changes while scanning the transaction log in forward direction, that is, from oldest to newest entry. Recovery from a transaction failure reads the pertinent log records backward (from newest to oldest entry; recall that each log record contains the prior LSN of the same transaction) and all changes of the transaction are undone or compensated. Recovery from a system failure scans the transaction log first in the forward direction in order to reapply logged updates and then reads in the backward direction in order to roll back those transactions that remained undecided at the time of the system failure.

Most of the following recovery techniques apply to data structures other than B-trees but also aspects and optimizations particularly suited to B-tree indexes and to system transactions. At the end of the present section, an example application (log shipping in replicated key-value stores) illustrates how these recovery techniques and optimizations enable efficiency in new usage scenarios for B-tree indexes.

Figure 5, adapted from Gray [1978], shows the duration of five transactions ($T_1 - T_5$), two checkpoints (S_0, S_1), a system failure (S_2), and a checkpoint after recovery (S_n). The crash at time S_2 does not affect T_1 because checkpoint S_1 has saved all dirty data pages of that transaction. Transaction T_2 has committed prior to the crash and therefore its effects must be guaranteed by appropriate "redo" recovery for all updates after checkpoint S_1. Transaction T_3 never reached its commit point. Therefore, "undo" recovery (compensation) must erase all traces of the transaction, including those saved in the database by the checkpoint S_1. Transaction T_4 requires "redo" recovery for all its updates, whereas transaction T_5 requires "undo" recovery (compensation).

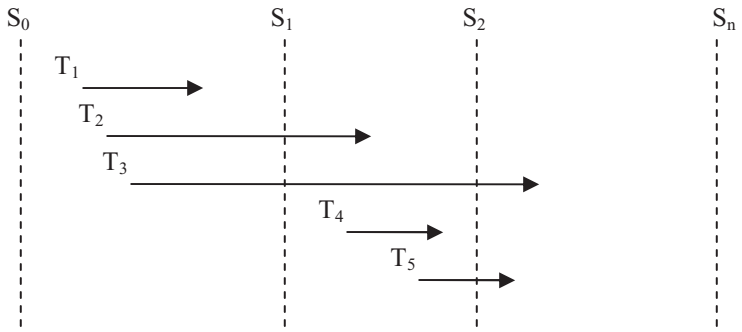

Fig. 5. Transactions, checkpoints, and recovery.

5.1. Asynchronous Prefetch

Database recovery exploits all available hardware only if there is an active I/O operation for each device at most times. If a single recovery log describes changes to data on multiple storage devices, it is clear that concurrent and asynchronous I/O is required in order to make the best use of all available hardware. Asynchronous write-behind merely requires that the recovery logic employs the buffer pool, marks pages dirty after reapplying updates, and leaves it to the normal buffer pool logic to write pages back to the database. In addition, asynchronous read-ahead or prefetch may also be required for better recovery performance.

During the "redo" pass over the recovery log, simply reading ahead in the log can drive asynchronous page prefetch. In most cases, it is not required to read log pages asynchronously, since each page in the recovery log usually contains references to many data pages. Asynchronous prefetch of the data pages referenced in a single log page is often sufficient to achieve most of the possible speed-up for the "redo" pass over the recovery log.

During the "undo" (compensation) pass over the recovery log, individual transactions and their chains of log records must be traced back to transaction starts. Many of the required log pages might remain in the buffer pool from the "redo" pass. Some transactions, however, might have started prior to the most recent checkpoint, the starting point of the "redo" pass. Thus, the "undo" pass usually requires reading additional, older log pages. Nonetheless, most I/O pertains to data pages, where asynchronous prefetch is more effective than for log pages.

Data pages required for "undo" operations may be fetched into the buffer pool in any order. Transaction compensation may proceed one transaction at-a-time or strictly in reverse chronological order (LSN order). Correctness requires merely that compensation actions follow the chain of log records within each individual transaction. Locks acquired during transaction processing before the system failure prevent ill effects between transactions. Lock acquisition during recovery is not required to ensure correct coordination of recovering transactions. (Note that multiple threads participating in the recovery logic still require latches to coordinate access to in-memory data structures).

Recovery of B-tree operations and B-tree updates can exploit asynchronous prefetch in some special ways. For example, system transactions require that all affected data pages be resident in memory. Careful orchestration of prefetch can ensure that system transaction can be redone as quickly as possible or that new system transactions required for logical transaction compensation can proceed as soon as possible.

Moreover, inasmuch as B-tree operations log changes of multiple data pages within a single log record, for example, splitting and merging B-tree nodes, asynchronous

prefetch of all affected pages seems an obvious approach and a particularly good fit for B-tree indexes.

5.2. Parallel Recovery

In addition to exploiting multiple I/O devices concurrently, recovery from a system failure can also exploit multiple CPUs, processing cores, and threads.

During the "redo" pass, the foundation for parallelism is partitioning of data pages (e.g., by hashing the page identifier). All log records pertaining to the same data page are processed by the same recovery thread, and they are applied in accordance with the order during transaction processing before the failure.

Partitioning of transactions (e.g., by transaction identifier) might lead to recovery actions being applied in an order different from the processing sequence before the failure. In an unfortunate case, a deletion (or ghost removal) might be required prior to an insertion, with the correct sequence before the failure but an impossible sequence during recovery due to partitioning transactions to multiple threads. Partitioning log records (e.g., by LSN) suffers from the same danger.

During the "undo" (compensation) pass, the foundation for parallelism is partitioning of transactions. Partitioning of data pages is not possible since some compensation actions apply to pages never mentioned in the recovery log. For example, an insertion into one page may require deletion in another page if the record has been moved in the meantime, for example, when a B-tree node was split or merged.

Some additional considerations apply to parallel recovery. First, multiple thread issuing I/O requests may lead to thrashing I/O devices. Thus, it might be appropriate to assign all data pages on a device to the same recovery thread.

Second, as discussed earlier as an optimization for logging system transactions, multiple log records might be fused together and with the commit record. For example, the log records for splitting a B-tree node might be fused into one, even though a split affects three data pages and one page in the free space information. For parallelism during recovery, specifically the "redo" pass, it might be necessary to split such a fused log record, with one copy or component per recovery thread or even per affected data page.

The design of ARIES [Mohan et al. 1992] includes parallel "redo" and "undo" (compensation). The "redo" is designed as described before, whereas parallel "undo" for a single transaction splits creation of compensation log records, which is done sequentially, and applying them, which is done in parallel.

5.3. Log Analysis

Recovery after a system failure (crash) requires two passes over the recovery log, first forward from the most recent checkpoint to the end of the log and then backward for those transactions that require rollback. Curiously, recovery performance can improve with the introduction of an additional pass prior to the "redo" and "undo" passes.

Log analysis scans the log forward from the most recent checkpoint without touching any data pages. It requires I/O only for the recovery log and thus very little time. It executes some of the logic that otherwise is part of the first pass in two-pass recovery. In addition, it permits several other optimizations and performance improvements for recovery from a system failure or a media failure.

During two-pass recovery without prior log analysis, random I/O for data pages imposes most of the delay. For each logged page modification, recovery reads the data page in order to verify that the page modification is indeed reflected in the database. The test is very simple; it focuses on the LSN included in the page header. If the LSN on the page is lower than the LSN of the log record being verified, recovery must reapply the logged change, write the current LSN into the page header, and mark the data

page dirty in the buffer pool. Asynchronous write-behind, replacement in the buffer pool, or a checkpoint at the end of recovery will write the page back to the database.

In most cases, however, the data page read during recovery satisfies the comparison of LSNs, because the data page has been written due to replacement in the buffer pool and continuous flushing of dirty pages from the buffer pool. Thus, the random read operation during recovery has taken much time but not resulted in any recovery applied to the database. Recovery performance improves if all or most useless read operations can be eliminated, for example, by noting in the recovery log when a dirty data page in the buffer pool has been written successfully to the database.

If each completed write operation is logged, a log analysis pass can achieve this purpose, except perhaps for a few data pages referenced at the very end of the recovery log. If the analysis pass matches a log record describing a page modification to a subsequent log record describing the completed write operation, the "redo" pass can avoid a useless random read operation. In systems that permit additional changes to a dirty page in the buffer pool while a write action is pending, the log record describing a complete write operation must include the page LSN written. Mohan et al. [1992] considered logging completed write actions and exploiting them during recovery but did not include it in the design.

Both B-tree creation algorithms ("allocation-only logging") and some structural modifications of B-trees (e.g., node split and merge operations) write their data pages in a careful sequence. Logging the completion of such write sequences, together with log analysis ahead of "redo" recovery, eliminates possible bottlenecks in recovery and failover of data stores-based B-trees, because it eliminates structural B-tree modifications and the affected pages from the recovery process.

A system failure indicates some root cause that is probably not resolved by system restart and recovery. Thus, another system failure might occur soon. If another system failure occurs during recovery, the same logic and I/O cost apply again. Thus, completed write actions should be logged both during normal transaction processing and during recovery. Moreover, checkpoints should be taken during and after recovery from a system or media failure. A checkpoint immediately following the log analysis can save the results of the log analysis and thus speed a subsequent system recovery should it become necessary.

Log analysis prior to any recovery action may save not only I/O during recovery but it may also eliminate some recovery actions altogether. For example, is it truly required to "redo" actions of aborted transactions? Mohan and Pirahesh [1991] proposed restricted repeating of history in ARIES-RRH. Only those operations with later dependencies are repeated during recovery, with a transaction commit being one form of dependency. Other dependencies can be other updates to the same pages. For example, if one transaction inserts a new record into a B-tree node and thus modifies the slot number of other records, and then another transaction and its log records refer to a record within that page by slot number, then the history of these transactions is repeated during "redo" recovery even if the first transaction eventually is aborted and its updates compensated with "undo" recovery.

5.4. Sorting Log Records

A log analysis pass also permits preprocessing log records. In the simplest form of preprocessing, log records might be sorted by the data page to which they pertain. The sort algorithm must ensure that, for each data page, the sequence of log records after sorting (and thus during recovery) is the same as in the log (and thus during transaction processing before the failure). This type of sort is known as stable sort; any sort algorithm can be made a stable sort by appending an appropriate minor sort key to the major sort, for example, the LSN in this case.

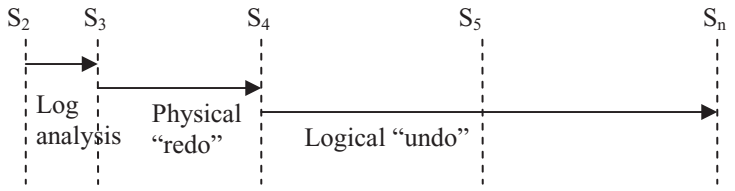

Fig. 6. Recovery phases.

The most important benefit of sorting log records during recovery is that each data page is recovered only once. In particular, each data page is read and written only once during recovery from a media or system failure. After a data page reflects all pertinent log records, the buffer pool may evict and replace it immediately. An additional benefit is that checkpoints taken during or after recovery can be very fast, because hardly any dirty data pages are lingering in the buffer pool.

Instead of sorting on page identifier alone, the sort order may consider device identifiers (and thus recover devices in priority sequence or all devices concurrently), tables (and thus recover tables in priority sequence), or transaction types (and thus recover all incomplete system transactions first such that the database reaches a consistent physical state quickly). The sort order may also include index identifier; for example, indexes being built at the time of the system or media failure might be recovered last. Within ordered indexes such as B-trees, the sort order may include key values such that an index becomes available to new user transactions in a controlled, incremental way. The latter technique might prove particularly valuable for partitioned B-trees, with key ranges defined by the artificial leading key field or partition identifier. For example, if one small, in-memory partition is dedicated to absorbing all new insertions and deletions (by insertion of deletion markers), then immediate recovery of that partition may enable quick resumption of database activities, for example, capture of sensor data or of business processes.

The sort operation may include both "redo" actions and "undo" (compensation) actions. If "undo" actions are included, more than the log analysis pass is required to produce the input for the sort. Specifically, for transactions that remained undecided at the time of the system or media failure, log records requiring compensation actions are included in the sort. Of course, log records for "undo" actions must follow all log records for "redo" actions. Note that, in some cases, the same log record might require initial "redo" and subsequent "undo." Moreover, log records for "redo" actions must be sorted by increasing LSN and log records for "undo" (compensation) actions by decreasing LSN of the original log record.

5.5. Early New Transactions

Perhaps the most significant opportunity enabled by a log analysis pass is transaction processing during recovery from a system failure. The traditional design for recovery permits new transactions only after all recovery is complete. ARIES [Mohan 1992] includes an option that permits new transactions during the "undo" (compensation) phase. Using a log analysis pass, however, can enable new transactions during the "redo" recovery phase already.

Figure 6 illustrates the three recovery phases. Log analysis is very short as it reads only log records but no database pages. The database administrator can limit the duration of the physical "redo" phase at the expense of frequent checkpoints during transaction processing. The duration of the logical undo phase is unpredictable, because it depends on the duration and activity of the oldest transactions active during the system failure, for example, transaction T_3 in Figure 5. Checkpoints during recovery,

in particular after each recovery phase such as S_3, S_4, and S_n but also during a long-running recovery phases such S_5, are a precaution that speeds up recovery in case of another failure during the initial recovery.

If the log analysis pass acquires locks, all parts of the database requiring recovery can be locked at the end of the log analysis pass. This requires that checkpoints record the set of locks held by undecided transactions. Starting from the most recent checkpoint, log analysis can acquire and release locks very much like standard transaction processing. After log analysis is complete, transaction processing may resume, including both retrievals and updates. The retained locks prevent new transactions from interfering with actions required in the "redo" and "undo" passes of the recovery.

While lock acquisition and release during recovery imposes an additional effort, some optimizations apply. First, log analysis during recovery may acquire locks without checking for conflicts. Transaction processing prior to the system failure has already detected all concurrency control conflicts among all recovering transactions, including both those that committed prior to the failure and those that need to roll back. Queues of waiting locks are also not required.

Second, the granularity of locking can differ during transaction processing and recovery. For example, recovery might lock an entire B-tree when transaction processing before the failure used key range locking. Multiple transactions in recovery may hold the same exclusive index lock since locks acquired during recovery do not conflict. Of course, large locks might interfere with transaction processing during the "redo" and "undo" passes of the recovery. Fortunately, the standard rules for lock escalation (from a small granularity of locking to a large one) and deescalation apply. For example, log analysis may lock an entire B-tree but conflicts with new transactions may force deescalation to key range locking. This requires that log records contain key values. Recovery will complete with key range locking or, after all conflicting new transactions commit, may request lock escalation back to an index lock. However, since recovery does not acquire any additional locks after log analysis, there does not seem any advantage in lock escalation in such a situation.

The design of ARIES [Mohan et al. 1992] includes lock acquisition during the "redo" pass and permits early transactions during the "undo" (compensation) pass. At least one commercial system has copied this design. Since lock acquisition is delayed to the "redo" pass, however, it is not complete until the "redo" pass with all its I/O and log application actions is complete. Thus, lock acquisition during a dedicated log analysis pass enables much earlier transaction processing during a restart recovery from a system failure.

An alternative perspective on the same techniques merges all precrash transactions into a single one, extending incomplete transactions with appropriate compensation actions and then committing the resulting transaction. Obviously, the actions within that single transaction do not conflict; therefore, lock acquisition during log analysis does not require conflict checking. Moreover, while this single transaction (i.e., failure recovery) is proceeding, it can employ lock escalation and deescalation just like other transactions. This single transaction may invoke new system transactions as appropriate, for example, in order to execute compensating actions.

5.6. Forward Recovery

After a media or system failure, standard database recovery erases or compensates the database changes of all transactions unfinished at the time of the failure. Taking database changes back in this way gives rise to the terms "rollback" and "backward recovery." For most transactions, this is appropriate, but perhaps not for all. For a long-running database utility such as index creation, a lot of work may be lost due to backward recovery. Moreover, it is quite likely that the database administrator

will restart the index operation as soon as recovery is complete and the database is back to a stable state. Thus, "forward recovery" would be very desirable, that is, the recovery process should, in appropriate cases, attempt to finish some of the incomplete transactions rather than roll them back.

The term "forward recovery" can have multiple meanings. For example, recovery from media failure by restoring an old database backup and then rolling it forward by replaying a series of log backups has been called forward recovery [Cabrera et al. 1995]. Do et al. [1998] describe it as: "all long-lived tasks . . . should incrementally write checkpoints to a persistent storage. In case of failure, the system only needs to have partial roll-back to previous checkpoint and restart."

The definition by Zou and Salzberg [1996] of the term "forward recovery" in index reorganization is the one employed here. In their design for B-tree reorganization, incomplete load balancing among sibling nodes can be recovered not by undoing work already done but by finishing it. Their work considered a specific operation, index reorganization, forward recovery applies more broadly and more generally.

Forward recovery is liable to fail in some cases, for example, if the system is short of data space. Even if reservations during initial forward processing can prevent many such contingencies, they might not cover all possibilities, for example, if a database is migrated to a new server after a severe hardware malfunction. Thus, a user cannot reliably predict whether a transaction will, during recovery from a system or media failure, complete or roll back.

Given this lack of guarantees, it seems most appropriate to focus on cases in which users should not even care about completion or rollback. Fortunately, physical data independence creates a substantial class of such cases, namely modifications in the physical database representation that do not affect the logical database contents. In other words, system transactions are perfect candidates for forward recovery.

System transactions are usually very focused on a single task or change, for example, splitting a B-tree node, merging a pair of nodes, or creating or removing a single ghost record. Even large system transactions have only a single purpose, for example, merging space allocation information of a dropped index into the appropriate global space allocation information ("free space map"). Such large system transactions often invoke multiple small system transactions to achieve their overall purpose incrementally.

This restriction to a single task permits logging the intent of a system transaction very concisely, such that forward recovery can determine and follow through on the intent. For example, adding the purpose and required parameters (such as "split the B-tree node in disk page 4711") is very little information to add to the system transaction's first log record. In other words, with little additional information in the recovery log, system transactions may succeed in their first execution, even if a system failure interrupts that execution and recovery resumes it.

Long-running system and utility transactions can usually be broken into a long sequence of many small system transactions. As each system transaction leaves a consistent database state, it is not truly required that the entire sequence be executed before another transaction accesses or even modifies the affected data. Equivalently, a long-running transaction can often terminate and commit the changes achieved until that time. For example, if a defragmentation utility encounters contention in terms of processing load or concurrency control, defragmentation can terminate. Rollback is not required. It might be desirable to resume the operation in a separate transaction.

Some of this logic also applies to the recovery of large system transactions. Instead of complete rollback or complete forward recovery, recovery of a database utility active at the time of a system failure may leave the database in any consistent state, including the one that is attained most efficiently from the database state found during recovery.

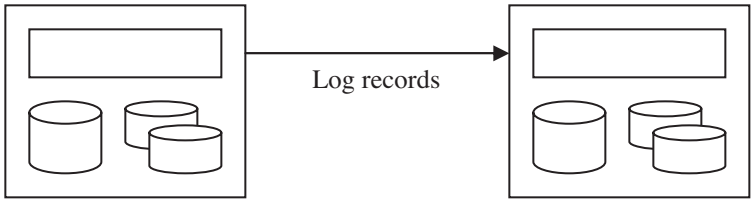

Fig. 7. Log shipping between primary and secondary sites.

5.7. Pulling the Concepts Together: Log Shipping in Replicated Key-Value Stores

An example might illustrate how these recovery techniques and optimizations combine to permit efficient B-tree operations. This example focuses on key-value stores for cloud computing, where multiple copies of all data enable high availability and scalability. Note that copies cannot substitute for logging and recovery, for example, in case of transaction failure and rollback (compensation), but they may eliminate the need for stable storage for the recovery log.

In a cloud server, there is little difference between failover, load balancing, workload management after server consolidation, etc. In other words, log shipping and failover techniques apply directly to all kinds of data management and indexing services deployed in the cloud. Log shipping means that log records drive maintenance of one or multiple copies of the data; failover means that a secondary copy can become the primary copy when appropriate.

Figure 7 illustrates log shipping from a primary site to a secondary site. The primary site includes a database and a recovery log (shown as mirrored disks as in Figure 3). The secondary site holds and maintains a copy of the database and of the recovery log. It does not generate this recovery log; it merely receives it from the primary site, applies it to its copy of the database, and saves it.

As non-logged page and index operations avoid detailed log records, the information contained in a log stream may not capture all database updates in all detail. Thus, log shipping requires the non-logged page and index operations be avoided on the primary site. Alternatively, the stream of log records can be augmented by copies of the affected database pages. This happens at the same time as these pages are forced to the primary database in order to ensure that subsequent log records describing updates at the primary site can be applied correctly at the secondary site.

In the design described here, any copy can answer any query, whereas updates in user transactions modify only the primary copy. The resulting recovery log drives maintenance of the secondary copies. In this design, commonly called log shipping, maintenance operations of the secondary copies save the effort for B-tree searches, for managing disk space, for monitoring and automatic tuning such as defragmentation, etc. In other words, this design minimizes the maintenance effort for secondary copies and thus optimizes scalability for read-intensive workloads. Moreover, since all copies are truly copies of each other, a secondary copy can instantly become the primary copy in the case of a failure, and the former primary copy can catch up eventually based on the recovery log generated at the former secondary copy.

The focus of the discussion here is maintenance of a secondary copy by application of the recovery log from the current primary copy. In a sense, each secondary copy is perpetually in recovery mode, at least until it is called upon to take over for a failed primary copy. Efficient application of the recovery log contributes to performance and scalability of the overall key-value store and its viability as a cloud service.

The design described here applies physical “redo” to each secondary copy. One advantage of using physical “redo” is its use of standard recovery code. Frequent or continuous

usage of this code raises the probability of rigorous testing and careful tuning. The alternative is logical "redo," in which the same logical operations modify different data structures. For example, Sears et al. [2008] use a log-structured merge-tree [O'Neil et al. 1996] for the secondary copy, which sequentially writes pages full of compressed data and therefore enables very efficient absorption of replicated operations. It seems, however, that physical "redo" and log-structured merge-trees or partitioned B-trees [Graefe 2003] in a primary copy will achieve the best performance, because this combination writes complete pages in both copies and it avoids some efforts inherent in logical recovery, for example, in-memory sorting or root-to-leaf B-tree traversals. The exceptions to physical "redo" are minimally logged operations such as B-tree creation, specifically if repeating the index creation is more efficient than shipping all pages of the new index.

Most data stores employ asynchronous prefetch and write-behind during request processing as a matter of course. Therefore, the secondary copies must do the same during recovery, that is, application of the log. Otherwise, they might not be able to keep up with the primary copy. Thus, the secondary copies must employ asynchronous prefetch driven by the recovery log, in addition to the asynchronous write-behind that is part of most designs for buffer pools and their page replacement mechanisms.

Most data stores also employ multiple concurrent read operations if multiple storage devices are used. Therefore, the secondary copies must gain further recovery performance by applying recovery actions to multiple database partitions in parallel, for example, partitioned by storage devices. Sorting log records might improve the access patterns on those devices and thus contribute to maintenance and recovery performance. Of course, traditional stop-and-go sorting consuming the entire input before producing the first output is not appropriate. Instead, this sort might be limited to run generation, for example, by replacement selection quite similar to the "elevator" algorithm for scheduling storage accesses.

Multiple passes over the recovery log, in particular a preliminary analysis pass, would not be appropriate in the context of log shipping. It is appropriate, however, that log analysis precede "redo" recovery by a sufficient amount to enable asynchronous prefetch of data pages requiring maintenance.

Finally, the design crucially depends on a technique equivalent of early new transactions, in particular lock acquisition and release during recovery processing. Specifically, while continuous recovery applies updates based on the incoming stream of log records, locks must ensure proper isolation of these updates with concurrent local B-tree searches. For example, update transactions may employ key range locking to prevent concurrent queries from reading inconsistent result sets.

Note that update transactions cannot conflict with one another in a secondary copy due to concurrency control at the primary copy, that is, during creation of the stream of log records. Thus, it might be possible to employ a lock compatibility matrix in which updates do not conflict. This may permit a large granularity of locking for updates and thus lower effort for lock acquisition and release during continuous recovery of the secondary B-tree copies.

Update transactions may conflict with local queries, however. Resolution of such lock conflicts must abort the local query transaction such that the update transaction can proceed. Otherwise, an update transaction would fail at a secondary copy and a secondary copy of the key-value store would differ from the primary copy.

If the primary copy fails (e.g., due to hardware failure) and a secondary copy is chosen to fail over, the secondary copy may employ forward recovery. This ensures that it can process the database updates as soon as they are redirected to the secondary copy and restarted there. If, however, the failure is not a node failure but a network disruption, independent decisions by the nodes about forward recovery will lead to differences in

the data representations and thus to very difficult reintegration after network repair. Moreover, if there are multiple secondary copies, all of them must take the same actions with respect to forward and backward recovery. If that cannot be ensured, traditional backward recovery is required.

5.8. Summary

In summary, recovery after a failure can be implemented in many ways. While the basics of forward and backward interpretation of the recovery log remain the same in all techniques, recovery performance can be improved by avoiding work (e.g., exploiting logged writes), by speeding up the required recovery actions (e.g., asynchronous prefetch and parallel recovery), by shortening the unproductive delay imposed by recovery (e.g., early transaction processing), and by making recovery itself productive (e.g., forward recovery of index operations). Some of these optimizations transcend B-trees even if invented and implemented first or primarily in the context of B-trees.

Key-value stores for cloud applications are an excellent use case for B-tree indexes, because B-trees offer many operational advantages such as efficient retrieval, efficient maintenance, built-in space management for small and large index entries, efficient creation and defragmentation, proven techniques for concurrency control and recovery, etc. Log shipping in replicated key-value stores applies almost all of the recovery techniques discussed in this section to B-tree indexes in a contemporary environment.

6. FUTURE DIRECTIONS

B-tree indexes will remain central in information storage and management. Therefore, their reliability and performance will remain an active area of research and development, including write-ahead logging and log-based recovery from transaction, media, and system failure. With many of the basics well understood, it will be mostly external forces that trigger new improvements, in particular new hardware technologies, unprecedented data volumes, shifting economics of hardware and data center operations, and the resulting new system architectures.

The advent of persistent (nonvolatile) semiconductor storage (such as flash memory) is one such change. For the time being, traditional disks will remain cost effective for cold data, although the break-even point guiding provisioning will grow from minutes [Graefe 2009] to hours and perhaps eventually days. Semiconductor storage devices will remain much faster (in latency) but smaller (in capacity) than disks. With a lower price per access (calculated via accesses per second) than disks, semiconductor storage will continue to command a higher price per capacity (measured in GB or soon TB). In other words, disks will remain cost effective for historical data.

The challenge for B-tree implementations is threefold. First, B-tree software including logging and recovery must be adapted to more than two layers in the memory hierarchy, that is, beyond the traditional model based on volatile memory and nonvolatile disk. For example, what precisely does “write-ahead” logging mean if an item is written twice, first to semiconductor storage and eventually to disk? Second, B-tree software must avoid local failures that are possible due to the durability issues that still plague new semiconductor storage devices. For example, can wear leveling be built into data structures such as B-trees in order to avoid another level of indirection implicit in today’s flash translation layers? Third, B-tree software must detect and remedy local failures online. This likely implies that incremental consistency checking and automatic repair of B-tree pages will gain importance. For example, is full media recovery required when a single page fails in a flash device?

In addition to transaction, media, and server failure and recovery, single-page failure could be a fourth failure class with its own recovery techniques. Single-page failures can occur in traditional disk drives, although those devices often can mask them with

"bad block mapping." Semiconductor storage or "storage-class memory" is known to have limited write endurance, with individual pages failing long before others. Wear leveling with techniques similar to log-structured file systems may ease or delay the issue. Nonetheless, failure detection and recovery in higher-level software such as database storage software might further improve the reliability and practical value of such semiconductor memory. Failure detection methods may include parity calculations within pages as well as consistency checks specific to the data structures such B-trees. Recovery method may include restoring a single page from a backup or snapshot and carrying it forward based on the recovery log.

As semiconductor storage enables low-latency I/O and a lower multi-programming level suffices to hide all I/O latencies. On the other hand, many-core processors increase the multi-programming level required to exploit all available resources for maximal throughout and efficiency. The result might be smaller units of work assigned to separate threads. Software latches may impose a limit on efficient concurrency, but hardware-supported transactional memory might provide a very inexpensive alternative and thus change many implementation details at the lowest level of B-tree code. Hardware transactional memory might be limited to a few cache lines but that might be sufficient for critical sections protected by software latches in today's data management software.

This redesign and implementation of lowest-level code might require a more systematic use of system transactions to enable more delayed, asynchronous, and incremental work based on committed logical actions. For example, incremental index creation may become the standard implementation technique instead of today's monolithic offline or online operations. Other examples include novel forms of deferred updates including maintenance of materialized and indexed views as well as histograms and other statistics.

In addition to bulk insertion (load), efficient bulk deletion from B-tree indexes is important in two contexts. First, most data warehouses not only load new data but also purge old data. The latter may occur as part of data movement from more expensive, performance-optimized storage to less expensive, capacity-optimized storage, for example, from flash storage to traditional disks, from RAID-1 storage to RAID-5 or RAID-6 storage, or from fiber-channel devices to SATA devices. Second, data streams with "punctuation" [Tucker et al. 2007] require efficient index insertion and deletion. These problems are similar to each other, albeit at different scales. Both cases demand concurrent query processing while indexes grow and shrink. Concurrency control, logging, and recovery for loading and purging remain opportunities for research and advanced development.

Finally, as mentioned earlier, transaction semantics and appropriate implementation techniques apply not only to database state but also to server state. Instead of rebooting a server process and developing its internal data structures as part of the recovery process, it may become more efficient to recover those data structures quite similarly to an in-memory database. It is an open question whether this is a viable approach and whether future hardware parameters and economics will encourage it.

7. SUMMARY AND CONCLUSIONS

In summary, logging and recovery for B-tree indexes can easily appear more complex than truly necessary. A systematic approach enables both understanding and implementation. It relies on physical data independence, specifically separation of logical database contents and physical representation, and the division of work into user transactions and system transactions. System transactions are similar to top-level actions in ARIES [Mohan et al. 1992] and well anticipated in the theory of multilevel transactions [Weikum 1991]. Their execution overhead is very low, because

they require neither a new execution thread nor a synchronous forced write to the recovery log upon transaction commit, and an implementation of B-tree indexes should therefore use them liberally.

Logging optimizations focus on space and bandwidth requirements by reducing both the number of log records as well as their level of detail. Proven optimizations pertain to operations on leaf records (user updates), nodes and pages (B-tree structure modifications), and entire B-trees (utilities). Many recovery optimizations separate log analysis from the "redo" pass over the log. These optimizations may reduce the recovery effort, speed up the remaining effort, reduce the impact of ongoing recovery on concurrent transaction processing, or recover the database and its indexes more directly towards a desirable state, for example, with new indexes completed rather than rolled back.

In conclusion, logging and recovery are not mysterious. Many of the techniques discussed are applicable to many other index types beyond B-trees. Thus, it will be possible to adapt and optimize them for future applications, storage formats, and hardware. Without doubt, the economics of hardware, software, maintenance, that is, the total cost of ownership of data and of data management software, will continue to evolve, and so will techniques for indexing in general and for B-trees specifically as well as techniques for concurrency control, logging, and recovery.

ACKNOWLEDGMENTS

Many thoughtful suggestions by Gary Smith, Harumi Kuno, the anonymous reviewers, and editor Anastasia Ailamaki have greatly improved the survey. Their help is gratefully acknowledged.

REFERENCES

Bayer, B. and McCreight, E. M. 1972. Organization and maintenance of large ordered indices. *Acta Inf. 1*, 173–189.

Bayer, R. and Unterauer, K. 1977. Prefix B-trees. *ACM Trans. Databa. Syst. 2*, 1, 11–26.

Bender, M. A., Farach-Colton, M., and Mosteiro, M. A. 2006. Insertion sort is O(n log n). *Theory Comput. Syst. 39*, 3, 391–397.

Bender, M. A. and Hu, H. 2007. An adaptive packed-memory array. *ACM Trans. Datab. Syst. 32*, 4.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Cabrera, L.-F., Rees, R., and Hineman, W. 1995. Applying database technology in the ADSM mass storage system. In *Proceedings of the International Conference on Very Large Database*. 597–605.

Do, L., Drew, P., Jin, W., Jumani, V., and Van Rossum, D. 1998. Issues in developing very large data warehouses. In *Proceedings of the International Conference on Very Large Database*. 633–636.

Graefe, G. 2003. Sorting and indexing with partitioned B-trees. In *Proceedings of the CIDR Conference*.

Graefe, G. 2004. Write-optimized B-trees. In *Proceedings of the International Conference on Very Large Databases*. 672–683.

Graefe, G. 2006. B-tree indexes, interpolation search, and skew. DaMoN: 5.

Graefe, G. 2009. The five-minute rule 20 years later and how flash memory changes the rules. *Comm. ACM 52*, 7, 48–59.

Graefe, G. 2010. A survey of B-tree locking techniques. *ACM Trans. Datab. Syst. 35*, 3.

Graefe, G. and Larson, P.-A. 2001. B-Tree indexes and CPU caches. In *Proceedings of the International Conference on Data Engineering*. 349–358.

Gray, J. 1978. Notes on data base operating systems. In R. Bayer, R. M. Graham, G. Seegmüller Eds., *Operating System—An Advanced Course*. Lecture Notes in Computer Science, vol. 60. Springer-Verlag.

Gray, J., McJones, P. R., Blasgen, M. W., Lindsay, B. G., Lorie, R. A., Price, T. G., Putzolu, G. R., and Traiger, I. L. 1981. The recovery manager of the System R database manager. *ACM Comput. Surv. 13*, 2, 223–243.

Gray, J. and Reuter, A. 1992. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Härder, T. and Reuter, A. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4, 287–317.

HELLAND, P., SAMMER, H., LYON, J., CARR, R., AND GARRETT, P., REUTER, A. 1987. Group commit timers and high volume transaction systems. In *Proceedings of the HPTS Conference*. 301–329.

KUMAR, V. AND HSU, M. 1998. *Recovery Mechanisms in Database Systems*. Prentice Hall.

LEHMAN. P. L. AND YAO, S. B. 1981. Efficient locking for concurrent operations on B-trees. *ACM Trans. Datab. Syst. 6*, 4, 650–670.

LESLIE, H., JAIN, R., BIRDSALL, D., AND YAGHMAI, H. 1995. Efficient search of multi-dimensional B-trees. In *Proceedings of the International Conference on Very Large Databases*. 710–719.

LOMET, D. B. 1998. Advanced recovery techniques in practice. In *Recovery Mechanisms in Database Systems*, Kumar and Hsu, 697–710.

LOMET, D. B. 2001. The evolution of effective B-tree page organization and techniques: A personal account. *SIGMOD Rec. 30*, 3, 64–69.

LOMET, D. B. AND SALZBERG, B. 1992. Access method concurrency with recovery. In *Proceedings of the ACM SIGMOD Conference on Database Management.* 351–360.

LOMET, D. B. AND SALZBERG, B. 1997. Concurrency and recovery for index trees. *VLDB J. 6*, 3, 224–240.

LOMET, D. B. AND TUTTLE, M. R. 1995. Redo recovery after system crashes. In *Proceedings of the International Conference on Very Large Databases*. 457–468.

LOMET, D. B. AND TUTTLE, M. R. 2003. A theory of redo recovery. In *Proceedings of the ACM SIGMOD Conference on Database Management.* 397–406.

MOHAN, C. 1990. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *Proceedings of the International Conference on Very Large Databases*. 392–405.

MOHAN, C. 1999. Repeating history beyond ARIES. In *Proceedings of the International Conference on Very Large Databases*. 1–17.

MOHAN, C., HADERLE, D. J., LINDSAY, B. G., PIRAHESH, H., AND SCHWARZ P. M. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *Trans. Datab. Syst. 17*, 1, 94–162.

MOHAN, C. AND PIRAHESH, H. 1991. ARIES-RRH: Restricted repeating of history in the ARIES transaction recovery method. In *Proceedings of the International Conference on Data Engineering.* 718–727.

MOSS, J. E. B. 1992. Working with persistent objects: To swizzle or not to swizzle. *Trans. Softw. Engin. 18*, 8, 657–673.

MOSS, J. E. B. 2006. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues (WMPI)*.

O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf. 33*, 4, 351–385.

OBERMARCK, R. 1998. Logging and recovery in commercial systems. In *Recovery Mechanisms in Database Systems*, Kumar and Hsu, 711–720.

SEARS, R., CALLAGHAN, M., AND BREWER, E. A. 2008. Rose: Compressed, log-structured replication. *Proc. VLDB 1*, 1, 526–537.

SUN, X., WANG, R., SALZBERG, B., AND ZOU, C. 2005. Online B-tree merging. In *Proceedings of the ACM SIGMOD Conference on Management of Data.* 335–346.

TUCKER, P. A., MAIER, D., SHEARD, T., AND STEPHENS, P. 2007. Using punctuation schemes to char-acterize strategies for querying over data streams. *IEEE Trans. Knowl. Data Engin. 19*, 9, 1227–1240.

WEIKUM, G. 1991. Principles and realization strategies of multilevel transaction management. *ACM Trans. Datab. Syst. 16*, 1, 132–180.

WEIKUM, G. AND SCHEK, H.-J. 1992. Concepts and applications of multilevel transactions and open nested transactions. In A. K. Elmagarmid. Ed., *Database Transaction Models for Advanced Applications*, Morgan-Kaufmann, 515–553.

WEIKUM, G. AND VOSSEN, G. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann.

ZOU, C. AND SALZBERG, B. 1996. On-Line reorganization of sparsely populated B^+-trees. In *Proceedings of the ACM SIGMOD Conference on Management of Data.* 115–124.

Received April 2011; accepted June 2011