Concurrent queries and updates in summary views and their indexes[1]

Goetz Graefe

Hewlett-Packard Laboratories

Abstract

Materialized views have become a standard technique in decision support databases and for a variety of monitoring purposes. In order to avoid inconsistencies and thus unpredictable query results, materialized views and their indexes should be maintained immediately within user transactions just like ordinary tables and their indexes. Unfortunately, the smaller and thus the more effective a materialized view is, the higher the concurrency contention between queries and updates as well as among concurrent updates. Therefore, we have investigated methods that reduce contention without forcing users to sacrifice serializability and thus predictable application semantics. These methods extend escrow locking with snapshot transactions, multi-version concurrency control, multi-granularity (hierarchical) locking, key range locking, and system transactions, i.e., with multiple proven database implementation techniques. The complete design eliminates all contention between pure read transactions and pure update transactions as well as contention among pure update transactions; it enables maximal concurrency of mixed read-write transactions with other transactions; it supports bulk operations such as data import and online index creation; it provides recovery for transaction, media, and system failures; and it can participate in coordinated commit processing, e.g., in two-phase commit.

1 Introduction

Timely business decisions depend on rapid information propagation from business operations to summary views for decision support. For example, productivity gains through inventory reduction depend on just-in-time supply chain management and thus require that e-commerce orders be summarized quickly for business analytics applications. In the ideal case, a single database management system is used for both business operations and decision support, and materialized views and their indexes summarizing customer orders are maintained immediately as user transactions update base tables and their indexes.

Consider, for example, the *lineitem* table in the well-known TPC-H database [TPC]. Among others columns, this table includes *orderno*, *lineno*, *partno*, *commitdate*, and *shipdate*, as illustrated in Figure 1 with some example values. Assume that a business using such a table wishes to monitor or audit its current and recent adherence to shipping deadlines and thus defines the view *select commitdate, shipdate, count (*) as shipments from lineitem where shipdate > '2000-01-01' group by commitdate, shipdate*, as illustrated in Figure 2 using some example values. For efficiency of both queries and updates, this view might be materialized and a clustered B-tree index may be constructed for the view with the grouping columns as search key, i.e., *commitdate, shipdate*, and all other columns as payload, i.e., *shipments*. While the main table might contain 100,000,000 rows, the materialized summary view might contain only 10,000 rows; and while insertions and updates in the table might affect different rows such that traditional locking is entirely adequate, most of the update activity in the view is focused on only a few rows. Moreover, in the assumed business scenario as in any real-time monitoring cases, much of the query activity also focuses on the part of the view that includes the rows with the most updates.

1.1 Serializable view maintenance

A premise of our research is that indexes should improve performance but should not alter the semantics of queries and updates. Thus, we believe that all indexes should be maintained immediately as part of user transactions, such that there is never a possibility that a transaction or query has different semantics

[1] A shorter report on this subject appears in Proc. ACM SIGMOD Conf. 2004. Beyond more related work, details, and examples, the present paper adds coverage of save points and partial rollback (including release of recently acquired or upgraded locks), lock escalation and de-escalation (including their treatment during partial rollback), two-phase commit (in local and distributed systems), and commit processing for optimistic concurrency control (which turns out to be similar to commit processing with escrow locks and virtual log records).

depending on the query optimizer's plan choices. Even if some users and some businesses are willing and able to accept query results that are somewhat out-of-date, i.e., accept asynchronous maintenance of materialized views, a general-purpose database management system should at least offer instant maintenance of views. This is contrary to designs that only support deferred view maintenance and thus force the user or application developer to trade off high performance against stale or even inconsistent data. Strict serializability of concurrent transactions provides users and businesses by far the most predictable application semantics, and we therefore believe that database management systems should provide efficient and correct implementations of serializable queries and updates for indexed views.

Order No	Line No	…	Part No	Commit Date	Ship Date
4922	…	…	…	2003-11-29	2003-11-27
4922	…	…	…	2003-11-29	2003-11-27
4939	…	…	…	2003-11-29	2003-11-27
4956	…	…	…	2003-12-31	2003-12-28
4961	…	…	…	2003-12-31	2003-12-29

Figure 1. Fact table with detail rows.

If maintenance of materialized views is deferred, i.e., materialized views are permitted to "fall behind" the base tables and are brought up-to-date only at convenient times, e.g., during nights or other periods of low system activity, it might seem that the techniques presented here are not required. However, if there is any activity at all against those views, concurrency and consistency must be considered. Such activity can result from sporadic queries due to around-the-clock or around-the-world operations, or it can result from concurrent updates because view maintenance is split into multiple concurrent batches. In all those cases, the techniques described in the present research can guarantee serializability, consistency, and high concurrency.

Commit Date	Ship Date	Shipments
2003-11-29	2003-11-27	3
2003-12-31	2003-12-28	1
2003-12-31	2003-12-29	1

Figure 2. Materialized summary view.

In fact, unless materialized views and their indexes are maintained immediately as part of user transactions, their existence can alter the semantics of queries and thus applications. For example, assume a view is indexed in support of some important, frequent query, and that a second query also happens to use this view, but with only moderate gain. After some updates, an automatic utility refreshes histograms and other statistics, whereupon the query optimizer recompiles the second query into a plan that does not employ the materialized view. The question now is: Did the automatic statistics refresh modify the transaction semantics and affect the query results? If the indexes on the materialized view are maintained immediately just like indexes on traditional tables, the answer is "no", and indexes on materialized views are purely a choice in physical database design just like traditional indexes on tables. On the other hand, if maintenance of materialized views is deferred or delegated to future maintenance transactions, the answer is "yes," since the query optimizer's choice affects which data are produced as query result.

If the view is joined with a table that is maintained immediately by all user transactions, even inconsistent query results are easily possible. For example, consider a query that lists orders for which the committed ship date was missed (comparing the *commitdate* and *shipdate* columns in the example table), specifically those orders that were the only ones for a specific ship date. The specific restriction is computed most efficiently using the materialized view; however, if the view is out-of-date, incorrect query results may be presented to the monitoring or auditing application.

Traditional concurrency control methods use shared and exclusive locks on tables, indexes, pages, records, and keys. For any locked data item, these lock modes limit update activity to a single update transaction at a time, which can be too restrictive for indexed views. Escrow locks and other commutative "increment" locks permit multiple concurrent update transactions to lock and modify a single data item, but they have their own restrictions. First, modification operations must be commutative with each other; fortunately, most summary views rely on counts and sums that can be incremented and decremented with commutative operations. Second, since multiple concurrent uncommitted updates imply that there is no single definitive current value, escrow locks are incompatible with share locks and therefore with read transactions, thus defeating the benefit of materialized views for monitoring ongoing operations.

1.2 Contributions

Mechanisms for concurrency control and recovery in summary views and their indexes are the primary subject of this paper. Prior research on materialized views has focused on choosing the views most desirable to materialize, on incremental maintenance of such views, and on query optimization exploiting materialized views. The present research is the first to focus on concurrency control and recovery for materialized and indexed views. Thus, it complements prior research on materialized views.

A second subject is integration of concurrency control and recovery with the entire operation of a database management system, e.g., multi-granularity locking, lock escalation and de-escalation, transaction save points, distributed two-phase commit, online index creation of materialized views. The latter cover not only creating an index for a view that is already materialized but specifically creating the view's initial materialization and thus its first index while permitting concurrent update transactions against the tables over which the view is defined.

In a side note, we also explore parallels between commit processing in optimistic concurrency control and commit processing with delayed transaction steps, e.g., in IBM's IMS/FastPath product, in difficult-to-reverse traditional index operations such as a drop operation, and in our design for concurrent updates of materialized and indexed views.

Since the proposed design originated in a product development organization, its goals are completeness of coverage, exploiting and complementing existing transaction implementation methods, and simplicity and robustness to the extent possible. It is this reliance on existing techniques that leads us to believe in the correctness and performance of the proposed methods. A complete analysis and sound proof of all aspects of this design seem an unrealistic undertaking.

To the best of our knowledge, our design is the first to address concurrent high read traffic and high update traffic over the same summary rows, to combine concurrent updates permitted by escrow locking and the linear history required for multi-version concurrency control and snapshot isolation, to consider multi-granularity escrow locks as well as concurrent small and large increment transactions using escrow locks, to integrate escrow locking with two-phase commit as well as save points and partial rollback with lock release, or to address concurrent update operations while a view is initially materialized and indexed, i.e., online materialization and index creation for summary views.

1.3 Overview

Building upon a wealth of existing research and experience, the design presented here fundamentally relies on four techniques, three of which are adaptations and extensions of existing research whereas the last one is a new idea that unifies multiple prior techniques. The contribution of our research effort is to combine these techniques into a coherent whole that enables concurrent execution and commit processing for small and large read and update transactions.

The first technique we adapted is to run pure read transactions in snapshot isolation, i.e., their commit point is their start time or logically their position in an equivalent serial transaction schedule. Moreover, we describe a novel design that combines concurrent update transactions with multi-version snapshot isolation, which in its traditional form supports only linear histories with only one update transaction at a time (for each data item).

Second, counts and sums in indexed views are maintained within user transactions using extensions of escrow locks, e.g., multi-granularity (hierarchical) escrow locking. Our extensions support not only small update transactions but also large operations such as bulk import and index creation, both with concurrent small update transactions using escrow locks.

Third, system transactions are employed not only to erase pseudo-deleted or "ghost" records but also to create such records that can then be maintained by user transactions using ordinary increment and decrement operations protected by escrow locks. This implementation technique ensures maximal concurrency when new summary rows are created in materialized views and their indexes.

The fourth component of our design requires that all stored records include a delta count, which unifies the roles of the count of duplicate rows, the size of the group, ghost bit, and anti-matter bit (explained later). Traditional ghost bit and anti-matter bit are no longer required, and an implicit predicate on the delta count in any query supplants today's implicit predicate on the ghost bit. The value of this unification will become apparent later, in particular in the section on online index operations.

After reviewing the wealth of prior research, we consider readers and updaters, multiple updaters, logging and recovery, multi-granularity locking, creation and removal of summary rows, and finally online index creation. We end with arguments about correctness and performance as well as a summary and our conclusions from this research.

2 Prior work

Our research is based on many prior research and development efforts, which we survey now in some depth because our design is primarily a novel combination of those existing techniques. No system has integrated all of them to-date. While some of these techniques might seem unrelated to each other or even to the topic at hand, they all contribute to the design. Knowledgeable readers may prefer to skip this section or refer to specific topics only as needed during later sections.

2.1 Deltas for materialized views

Multiple research efforts have focused on deriving the minimal delta needed for incremental maintenance of materialized views, e.g., [BCL 86, GM 99]. While important, those techniques are not reviewed here because they are complementary and orthogonal to our research, which focuses on transaction techniques such as locking and logging while applying such deltas to indexes on materialized views.

We also do not concern ourselves here with concurrency control needs during computation of the delta, e.g., reading additional tables while computing the delta for a join view. Concurrent updates of multiple tables contributing to the same join view may lead to contention in traditional locking system. Some of that contention may be mitigated by multi-version concurrency control, which is discussed below.

2.2 Concurrency control for materialized views

Other research also started with the importance of serializability of tables and materialized views, but assumed that instant maintenance imposes unacceptable overhead on update transactions, and then derived theories for detecting inconsistencies as part of read operations and resolving them using a postulated operation that brings a single row in the summary view up-to-date [KLM 97]. In contrast, we assume that indexes on views are maintained as part of user update transactions just like indexes on ordinary tables.

Instead of locking rows, records, key ranges, and keys in materialized views and their indexes, it is possible to always lock the underlying tables, their rows and their indexes, even if the chosen query execution plan does not employ them at all [LNE 05]. The advantage of this approach is that no special lock modes are needed; in fact no locks are needed at all for the views and their rows. The disadvantage, however, is that a single row in a materialized view might derive from tens, hundreds, or thousands of rows in an underlying table, forcing a choice whether to acquire that many individual row locks or to lock that table in its entirety.

We are aware of only one prior effort that specifically focuses on the same problems as our research. This prior design [LNE 03] employs special V and W locks that are, in a first approximation, equivalent to escrow locks and to short-term exclusive locks. In fact, the W locks and their duration are similar to our design choice of running system transactions that create ghost records, including their key-range locking behavior. In a follow-on publication [LNE 05], the same authors introduce multi-granularity or intention locks (IV locks) and replace the W locks with latches, i.e., short-term concurrency control mechanisms that do not participate in deadlock detection and thus impose other restrictions. In the specific design, these latches are allocated in a fixed-size pool, with each individual key value mapped to a specific latch using a hash function; in other words, these latches are somewhat detached from the views being updated. The authors argue that the appropriate size of the lock pool depends on the number of concurrent updates rather than their size or the size of the views. Transaction rollback is not discussed, e.g., whether transaction rollback erases records created under the protection of such a latch or what happens if another transaction has acquired a V lock on such a row right after it had been created.

For small update transactions, the performance effects reported in [LNE 03, LNE 05] and the design here are quite comparable, primarily because V locks and our E locks are very similar. The main difference in the two designs is that our design integrates concurrency control for materialized views with write-ahead logging and recovery as well as with multi-version snapshot isolation, transaction save points, partial rollback, and two-phase commit; and that our design covers additional application scenarios including readers concurrent with writers as well as bulk load and online index operations for materialized views.

2.3 Multi-level transactions and Aries

Prominent among the prior work are multi-level transactions [WS 84, WHB 90] as well as the Aries family of techniques [MHL 92], including appropriate logging and recovery techniques, e.g., [L 92]. The essence of multi-level transaction *undo* is that a higher-level action can be compensated in multiple ways. In fact, it is often not even possible to undo the precise physical actions invoked during the original forward processing. For example, if inserting a new key into a B-tree requires inserting a new record into a specific leaf page, undoing the key insertion might remove a record from a different leaf page, since the original leaf page was split or merged between the initial insertion and its compensation during transaction rollback.

Another important characteristic of multi-level transactions and Aries is that *redo* and *undo* of higher-level actions are not necessarily idempotent. The logging and recovery protocol guarantees that a higher-level action is redone only if none of its effects are reflected in the database, and it is undone only if all its effects are captured in the database. Lowest-level actions are undone or redone as in traditional recovery methods; higher-level actions rely on the fact that lower-level actions provide transaction semantics in their own right: higher-level actions are never redone (only the lowest-level actions are redone after a system restart), and they are undone by logical compensation.

When a transaction starts and invokes another transaction, that other transaction can commit its locks and its results either into the invoking transaction or to the system at large. The former case is a classic nested transaction. The latter case is called a "nested top-level action" in the Argus language [LS 82] and in Aries, and a "restructuring operation" elsewhere [S 85]; we call it a "system transaction" because it is a transaction invoked by the system itself for its own purposes. A typical example splits or recombines a page within a B-tree index. For those transactions, special commit optimizations apply, as discussed later. Moreover, a system transaction can be lock-compatible with its invoking transaction, which usually does not resume processing the user transaction until the system transaction commits.

2.4 Concurrent update techniques

The second very important foundation of our work is escrow locking [O 86]. As originally defined, escrow locks imply not only concurrent increment and decrement operations but also holding minimum values in escrow. However, we ignore this second aspect of escrow locks here and use the name generically for the family of locking schemes that exploit the commutativity of operations such as increment and decrement. Alternative commutative sets of actions could also be supported based on escrow locks, but incrementing and decrementing are the prototypical operations for escrow locking.

By definition, multiple concurrent, uncommitted update transactions imply that there is no single correct current value. Hence, an escrow lock does not permit reading the value or to set it to a specific value, and an escrow lock ("E" locks from now) implies neither a traditional "share" ("S") nor an "exclusive" ("X") lock. If a transaction holds an E lock on a data item and then needs to read the item, it must also acquire an S lock, which can only be granted after all other transactions have released their E locks and thus resolved all uncertainly about the correct current value.

Interestingly, the combination of S and E lock is practically equivalent to an X lock, not because it ensures the same privileges but because it has the same lock conflicts. Thus, we assume that the combination of S+E is not explicitly modeled; instead, E locks are upgraded to an X lock when read privileges are required in addition to increment privileges.

The original recovery scheme for escrow locks is quite complex, and multi-level transactions or Aries can be employed for a significantly simpler one [MHL 92]. The essence of this simplification is that recovery actions need not be idempotent – specifically, in a multi-level recovery scheme, *undo* actions at higher levels are not necessarily idempotent in their effects at lower levels. Specifically, *undo* actions are invoked only for data items that reflect the effect of the original *do* action, and *redo* actions are invoked only for data items that do not. Thus, *undo* and *redo* actions are increment and decrement actions practically equal to the original *do* actions. Log records must carry increment values, not the *before* and *after* images well known from traditional exclusive locking that permits setting a specific value but restricts update activity to a single uncommitted transaction at a time.

Somewhat similar to escrow locks are the concurrency and recovery techniques of IBM's IMS/FastPath product [GK 85]. During forward processing, conditions and increment operations are collected in an intention list associated with a transaction; during commit processing, the intention list is processed and values in the database are updated, using traditional after-image logging for durability and recovery. Commit processing for multiple transactions is serialized using the single latch synchronizing access to the recovery log. Careful optimization of the code path executed while holding the latch ensures that serial commit processing does not restrict overall system throughput [G 03].

Intention lists of deferred actions are often implemented but usually reserved for large destructive operations such as dropping an index or a table. For example, if an index is dropped within a multi-statement transaction, the actual structure is erased on disk only when the entire multi-statement transaction commits, because otherwise transaction rollback could be extremely expensive and might fail, e.g., due to lack of auxiliary disk space while reconstructing an index. Of course, transaction rollback must always be possible and succeed if attempted, or all transaction semantics are in jeopardy. It is not sufficient to deallocate the space allocated by the old index yet keep that space locked until transaction commit. For example, if an online index rebuild operation replaces one copy with a new one (which then has the proper on-disk layout, per-page free space, etc.), the old copy must remain accessible. In fact, it even must be maintained by concurrent update transactions, because otherwise the index would be out-of-date if the index builder aborts and rolls back.

The implementation of such deferred actions attaches an intention list (containing deferred actions) to each uncommitted transaction. Each such action is quite similar to a *redo* action, although there is no need to include such actions in the recovery log on stable storage. Instead, the recovery log on stable storage will document the actual physical actions performed while processing such “virtual” *redo* log records during commit processing, i.e., immediately before writing the transaction’s commit record to stable storage, or before the pre-commit record in a distributed transaction.

An interesting aspect is that recovery in multi-level transactions traditionally relies on physical *redo* and logical *undo* (usually called “compensation”), whereas virtual log records employed for deferred actions represent a logical *redo* operation. Note that idempotent *redo* actions are not required, because virtual log records are used only when execution of the *do* action is omitted during ordinary forward processing and because virtual log records are never found in the recovery log on stable storage, i.e., they are never processed during possibly multiple restart attempts.

Alternative to the IMS/FastPath method, conditions can be verified immediately against the entire range of uncertainty of a data value [R 82]. For example, if one uncommitted transaction has incremented a data item by 2 and another one has decremented it by 3, given an initial value of 10 the current range of uncertainty is 7 to 12. While it is possible to test a range of uncertainty against a condition, e.g., whether the value is greater than 5, it is not possible to read such a value, e.g., in order to display or print it.

2.5 Snapshot isolation and multi-version concurrency control

The third foundation for our work is the combination of snapshot isolation [BBG 95] and multi-version concurrency control [BHG 87]. Together, they can reduce or eliminate concurrency problems between readers and writers. In a correct serializable transaction schedule, each transaction must “see” for each database item the value most recently committed in any equivalent serial schedule, and a transaction must not overwrite any value other than those that it is permitted to “see.” A standard technique is to set the commit point (determining the position in an equivalent serial schedule) of a pure read transaction at the beginning of the transaction’s execution and to set the commit point for update transactions to the end of the transaction’s execution.

Thus, no matter how long a read transaction runs, the database management system must present a snapshot of the database as it existed at the transaction’s start, even if subsequent updates may have overwritten some database items. Read transactions have no need for locks, since they only read committed data that cannot be changed by concurrent or subsequent update transactions. Note that if a read transaction and a write transaction run concurrently, the commit point of the read transaction necessarily precedes the commit point of the write transaction, and the two transactions cannot possibly conflict.

A limitation of this technique affects transactions that both read and write, because a transaction cannot have two commit points in a serial schedule or in a correct serializable schedule. A seemingly clever implementation is to run such a transaction as if it had two commit points, an early one for read actions and a late one for write actions, which of course violates the rules for well-formed transactions in serializable schedules and thus does not result in proper transaction isolation [BBG 95].

There are multiple implementation alternatives for snapshot isolation. Without supporting multiple versions, queries must fail rather than read a database value that is “too new.” If multiple versions are supported, since the main database reflects the most current state of the information stored in the database, earlier versions can either be retained or recreated on demand.

For the latter choice, i.e., recreating earlier version on demand, the recovery log is the main data source. The buffer pool must be designed to manage multiple versions of the same disk page, and an older version is derived from a newer one (the current one) by rolling back the page image [BJK 97]. For efficient navigation within the recovery log, the standard backward chain (linked list) of log records per transaction is not useful, i.e., the traditional mechanism for transaction rollback. In addition, a recovery log may also embed a backward chain per page. In fact, the recovery logs in some commercial systems already do. Thus, it is possible to roll back a single page to an earlier point in time, assuming the transaction log has not been truncated to reclaim space on the log disk.

For the former choice, i.e., retaining earlier images of records or fields, some additional temporary space must be allocated. In a database management system using B-tree indexes, the free space in each B-tree node is a natural place to retain earlier records. Note that the average space utilization in B-tree pages is about 70%, i.e., there is about 30% free space, and that accessing a single record's earlier image within the same page seems less expensive than rolling back an entire page to an earlier time using the recovery log and an auxiliary buffer. If earlier record images must be retained for a long time due to long running queries, spill-over space must be provided. A possible design, in particular if long-running transactions are rare, is to employ node splits in the B-tree using standard mechanisms and code. Alternative designs are possible but might increase code complexity beyond the proposed scheme without improving overall system efficiency. Since multiple commercial database management systems already include support multiple versions, we do not discuss these tradeoffs further.

Interestingly, no data locks are required to prevent premature clean-up of data structures used to support multiple versions. If the system keeps track of the oldest running snapshot transaction, a simple decision suffices to decide whether any active read transaction is currently reading an old version or might need to do so in the future. This idea, its efficiency and its implementation, are quite similar to keeping track of the oldest active update transaction in the *Commit_LSN* method [M 90b], and have been employed in commercial database systems [AJ 98].

2.6 Derived lock modes

The fourth foundation is the derivation of auxiliary lock modes from primitive lock modes, including intention locks, combination locks, and upgrade ("U") locks. Intention locks ("IS," "IU," and "IX") enable multi-granularity or hierarchical locking, e.g., such that one transaction can lock a few individual records individually, another transaction can lock an entire file with all records with one large lock, yet conflicts are detected correctly and reliably. Combination locks bestow the privileges of multiple locks for a locked resource, e.g., an "S+IX" lock on a file provides a shared access to the entire file plus the right to obtain exclusive locks on pages or records within the file.

Upgrade locks reserve exclusive upgrade rights from one lock mode to another, i.e., convert from one lock mode to another one that encompasses the prior one. The upgrade component of the lock does not grant any data access privileges; thus, it can be relinquished prior to transaction completion if lock conversion turns out not to be required. "U" locks are often called "update" locks, because update processing often upgrades traditional "S" locks to "X" locks in order to prevent deadlocks among multiple transactions attempting to convert an "S" lock to an "X" lock [GR 93]. The concept of upgrade locks is much broader, however. In his thesis, Korth developed a comprehensive theory of intention locks, combination locks, and upgrade locks as well as algorithms for designing sets of lock modes and their compatibility matrix [K 83].

2.7 Lock escalation and de-escalation

An entirely different kind of lock upgrade is lock escalation, i.e., replacing a set of fine-granularity locks with a single coarse lock covering all of the original locks. For example, an existing IS lock is converted to an S lock and the S locks acquired at the finer granularity are subsumed by it. The primary reasons for lock escalation are lack of memory for the lock manager [CLW 02, CWL 05], excessive run-time overhead for lock acquisition at the finer level, and adaptation after an initial error in choosing the appropriate locking granularity, e.g., after a predicate's selectivity was estimated at 10% during query optimization but turned out to be 90% during query execution.

The opposite of lock escalation is lock de-escalation, e.g., from an S lock at a coarse granularity to an IS lock plus individual S locks at the next finer granularity. The primary reasons for lock de-escalation are errors in anticipated selectivity and unexpected contention for resources. If implemented well, it even seems reasonable to always attempt locking at a coarse granularity but to resort to a finer granularity in case of conflicts, either if lock acquisition fails or if another transaction requests a lock later on.

Lock de-escalation during initial lock acquisition is fairly simple to implement. For lock de-escalation during execution, i.e., after some data access, the needed finer-granularity locks must be recorded from the start [J 91, LC 89]. The difference to locking immediately at the finer granularity is that the finer-granularity locks are guaranteed by the initial coarse-granularity lock. Thus, there is no need to record these locks in a shared lock manager; they can be recorded in thread- or transaction-private memory. Of course, during lock de-escalation, they must be propagated to the shared lock manager. However, these lock requests are guaranteed to succeed, because they cannot conflict with other transactions while the transaction still holds the initial coarse-granularity lock. In fact, due to fewer searches in the lock manager, recording fine-granularity locks and propagating them later can be faster than acquiring these fine-granularity locks individually [GL 92].

Even in private space, it might be a good idea to organize fine-granularity locks like the hash table in the shared lock manager, in order to avoid acquiring and recording the same detail lock twice. Moreover, it quite possibly simplifies merging these locks into the shared lock manager during lock de-escalation, and it enables in private memory precisely the same operations as are needed in shared memory for efficient lock management, e.g., during rollback to transaction save points.

As a side note, it might be interesting in some cases to perform lock escalation or de-escalation while a prepared transaction is waiting to learn the global transaction outcome. Such escalation and de-escalation are possible and correct, and may be worthwhile if lock contention changes due to other local transactions starting or ending. The essence of the preparation phase in two-phase commit is that the local transaction can abide by the global decision. Thus, the local transaction must retain sufficient locks to maintain that guarantee, but the granularity of locking may be modified at any time.

Similarly, if recovery after a system crash resumes transaction processing immediately after log analysis, i.e., during redo and undo (compensation) processing, then log analysis phase must obtain locks for all transactions that need recovery actions. If forward processing logs the acquisition of locks, lock acquisition during the log analysis pass may be guided by these log entries. Lock escalation and de-escalation during recovery can be employed to replace the locks chosen during pre-crash forward processing with locks more appropriate to the contention during post-crash recovery and concurrent transaction processing.

2.8 Transaction save points

When a save point is requested by a user or an application, it is marked in the transaction's chain of locks and possibly in the transaction log. Rollback to a save point, also known as partial rollback of a transaction, means that all database changes since that time are undone or compensated [KJP 99]. Typical uses of transaction save points include interactive transactions in which a failed statement cleans up all its changes without aborting the entire transaction. Each individual statement execution starts with a new save point implicitly created by the system for the purpose of statement rollback.

An interesting aspect of transaction save points is that locks can be released during partial transaction rollback. Thus, another use case for partial rollback is deadlock resolution. For that to happen correctly and efficiently, the lock manager's data structures must support not only efficient insertion and search but also deletion as well as enumeration of a transaction's locks in the reverse order of lock acquisition.

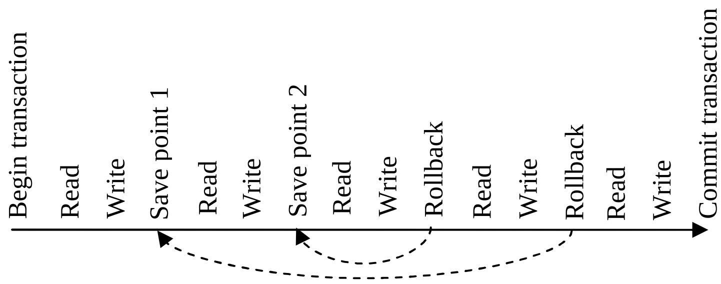

Figure 3. Complex partial rollbacks.

While aborting a single statement rolls back only the most recent save point, locking and logging must support any sequence of forward processing and partial transaction rollback. Figure 3 illustrates a complex transaction. In the first run through the transaction logic, a short rollback to save point 2 is needed, perhaps to abort the current statement. A second attempt of the same logic succeeds, but later a rollback to save point 1 is requested, perhaps to resolve a deadlock with minimal effort for transaction rollback and retry. In third attempt, the entire transaction succeeds and commits. While probably not a typical or frequent case, the locking and logging logic must support complex cases such as this one.

If read locks are kept in a separate lock chain, additional concurrency during one transaction's partial rollback can be gained by releasing read locks immediately (e.g., "S" and "IS"), before rollback of any write locks and update operations. This is quite similar to releasing read locks early in ordinary commit processing and during two-phase commit. In addition, upgrade locks can be downgraded (e.g., "U" to "S") and often released early together with the read locks. For maximal concurrency, write locks can be released incrementally as the rollback progresses.

Lock upgrades are special, e.g., from S or U to X as well as from IS or IU to IX, in particular if initial acquisition and upgrade are separated by a save point. In that case, either lock release during rollback must exclude upgrades and may include only original lock acquisitions, or initial lock acquisition and subsequent upgrade are represented as two separate data structures in the lock manager, even if these locks pertain to the same database resource.

Lock escalation, e.g., from an IS lock and multiple dependent S locks to a single S lock, can be rolled back quite readily, because this rollback actually releases locked access rights, namely those implied in the single large S lock but not included in the original small S locks. This assumes, of course, that appropriate detail information has been retained, i.e., the lock escalation was initiated in order to reduce the number of lock requests, not to reduce the amount of memory dedicated to locks and their data structures. Rolling back lock de-escalation, on the other hand, needs to re-acquire locks. If the original lock de-escalation happened upon demand, i.e., due to contention with a new lock request by a concurrent transaction, re-acquiring these locks is likely to fail. Moreover, even if rollback succeeds, renewed lock de-escalation is likely to be needed or wanted again soon. Thus, attempting to roll back lock de-escalation may not be worth the effort. Note also that rollback must never fail in such a way that a transaction can go neither forward nor backward; thus, the implementation of rollback must not rely on rolling back lock de-escalation. Fortunately, correctness of transaction rollback and continuation of that transaction do not depend on rolling back escalation and de-escalation, because the database resources required for the transaction will remain locked whether or not such rollbacks succeed.

Incremental lock release applies not only to rollback to a save point but also to rollback of an entire transaction. In fact, rollback of an entire transaction can be modeled and implemented using an implicit save point at the start of every transaction, and thus both rollback situations can use the same code. Neither rollback of lock escalation and nor rollback of lock de-escalation seems worthwhile if the entire transaction is aborted.

2.9 Local commit processing

A transaction's commit point is defined by writing a commit record to the recovery log on stable storage. However, contrary to common belief, traditional shared and exclusive locks can be released prior to that point, and the same is desirable for high-performance transaction processing with materialized and indexed views.

Shared locks can be released immediately after user or application request the transaction commit, i.e., after the transaction reaches its "high point" in two-phase locking. "Pre-commit" [DKO 84] (not related to preparation or pre-commit in distributed transactions) permits release of write locks to other transactions as soon as commit record has been written to the log buffer, even before it is written to stable storage and before the transaction's durability is truly guaranteed. If a subsequent transaction immediately locks the same resources, that transaction requires only durability if its own commit record is in the log on stable storage, and sequential log writing implies that the first transaction's log record is written earlier or simultaneously. Only notification of user or application must be delayed until the commit record is safely on stable storage.

This sequence of commit activities is used in IBM's IMS/FastPath. IMS/FastPath employs *group commit* and *fast commit* [GK 85]. The former attempts to always write full log pages rather than forcing the current log buffer to stable storage; the main mechanism is to introduce small delays with timeouts. The latter completes the commit process including lock release without flushing the log buffer containing the commit record. Correctness is ensured by delaying acknowledgement of the commit to the user (and thus delivery of the transactional durability promise) until the log buffer has indeed been written to stable storage. In other words, IMS/FastPath links user communication to the transaction log and its flush actions. This is quite similar to the link in traditional write-ahead logging between ordinary data pages in the I/O buffer and the transaction log. Both I/O buffer and communication buffer employ the same “high water mark” mechanism.

System transactions and their commit sequence are a special case of this optimization. All locks are released as soon as the transaction commit record is in the log buffer in memory. However, there is no user or application waiting for a notification; thus, there is no need to ever force the commit record to stable storage. Instead, a system transaction becomes durable only after the log buffer containing the transaction’s commit record is written to stable storage, e.g., because commit processing forced a log write for a subsequent user transaction.

A system transaction may modify only the physical data representation but not any logical database contents, because its results remain in the database even if the user transaction rolls back. It must be impossible based on the result of an ordinary “select” query to discern whether such a system transaction actually happened. Therefore, users and applications have no need for durability, one of the four traditional ACID properties of transactions [HR 83], so committing a system transaction does not require forcing buffered blocks of the recovery log to stable storage. Thus, committing a system transaction is substantially less expensive than committing an ordinary user transaction.

2.10 Two-phase commit

Two-phase commit is needed in many situations, as evidenced by many implementations and optimizations [SBC 93]. In addition to distributed query processing, two-phase commit is needed if a single transaction spans multiple databases within a single database server or if a database employs multiple files each with its own transaction log. If two or more transactions logs are involved in a transaction, reliable commit processing requires two-phase commit or an even more advanced technique such as a three-phase commit [S 81].

From an observer’s point of view, the essence of a two-phase commit is that the commit-or-abort decision is centralized to a single location and a single point in time [GR 93]. From a participant’s point of view, the essence of a two-phase commit is the separation of preparation and final commit or abort. During the preparation phase, the participants have a last chance to veto global transaction commit; if they don’t veto the commit, they subjugate the outcome of their local transaction to the global decision. Part of this promise is to protect the local transaction from any failure or abort. For example, such a transaction cannot be chosen as the victim in deadlock resolution. Interestingly, in addition to being the initiator, the user or the application also is an important participant that is often overlooked in discussions of commit coordination: in exchange for the user’s right to abort as well as the user’s promise to accept the transaction outcome whatever it may be, the transaction manager will attempt to commit the transaction as requested.

Write locks can only be released when the final transaction outcome is known and enforced within each participant, including both “X” and “IX” locks. Read locks, on the other hand, including both “S” and “IS” locks, can be released at the start of the preparation phase, without violation of two-phase locking [GR 93]. As a corollary, read-only transaction participants do not need to participate in the second phase of the commit coordination. After receiving the initial request to prepare and vote on the outcome of the global transaction, nothing of the local read-only transaction remains, neither locks nor log records. Traditional “U” and “IU” locks can be downgraded to “S” and “IS,” respectively. This relinquishes their implied exclusive permission to upgrade to an X or IX lock, which is no longer needed once the user or application requests transaction commit or abort. Thus, traditional U and IU locks can be released, too, during the preparation phase of two-phase commit.

2.11 Managing individual records and keys

Another important foundation of our research and of our design is found in the implementation mechanisms for key deletion under key-range locking [L 92, M 90a]. When a key is deleted, a witness for the actual key value remains in the index, because this permits retaining the key-value lock until the end of the user transaction. This witness usually is the deleted record with its old search key intact and with a special invalidation marker in the record header. While this idea has been known and used for a long time, e.g., during the System R project [B 03] and in Aries [M 90b], we call this record a “ghost record” and its marker a “ghost bit.” (The term “ghost” for logically deleted records seems to originate from [JS 89], albeit in a somewhat different meaning.)

Turning a valid record into a “logically deleted” or “pseudo-deleted” or ghost record instead of erasing it immediately from its storage location simplifies not only lock management but also *undo* processing, e.g., because space allocation cannot fail during rollback. After locks have been released during commit processing, the ghost record may be left behind for clean-up by a subsequent transaction, either a future insertion or an asynchronous clean-up utility. If the original transaction does not erase the ghost record during commit processing, all queries and scans must implicitly ignore records marked as ghosts. Thus, the convenience of ghost records implies that all queries have an implied predicate on the “ghost bit” or other marker within each record.

Note that a ghost record remains a resource that can be locked until it is actually erased, and that it defines a key range for the purposes of key range locking. Note also that erasing a ghost record is contents-neutral, that this action therefore is another typical use of system transactions. This system transaction may be part of commit processing for the original transaction, of an asynchronous clean-up utility, or of a future insert transaction that needs to reclaim and reuse record space within the disk page.

2.12 Online index operations

In addition to the ghost bit in each record header, some database implementations employ an “anti-matter” bit during online index creation, i.e., when permitting updates in a table while an index is still being created. There are two basic designs for online index operations, called “side file” and “no side file” [MN 92]. The former design builds the new index without regard to concurrent update operations and applies those in a “catch-up” phase guided by a side file that describes the missed updates, typically the recovery log. The latter design requires that concurrent update operations be applied immediately to the index being built, even if the index builder itself is still sorting and thus has not yet inserted any data into the new index. If a concurrent update operation attempts to delete a key in the new index that does not yet exist there, it instead inserts a special marker (“anti-matter”) that indicates to the index builder that a key must be suppressed.

Anti-matter is employed only within an index still being built, and all anti-matter is resolved by the index builder. Thus, no anti-matter remains when the index is complete, and no query or scan must ever encounter a record with anti-matter. Note that anti-matter can exist both in valid records and in ghost records due to multiple transactions in sequence deleting and inserting the same key while the index builder is still scanning and sorting. The semantics of the anti-matter bit are that the history of a given key in a given index began with a deletion. Note also that the traditional method using a single Boolean value (a bit) is not sufficient in indexes on views with a “group by” clause, where each record summarizes a group of rows and where multiple deletions might affect a non-existing row in the view.

2.13 Duplicate records and keys

A seemingly unrelated seventh foundation are the well-known mechanisms for representing duplicate records in intermediate query results and in stored tables. Duplicates can be represented using multiple records or by a single record with a counter. The former representation is easier to produce as it is the natural output of projection execution, whereas the latter is often desirable for data compression as well as for table operations with multi-set semantics. For example, the SQL “intersect all” clause applied to two input tables with duplicate rows requires finding, for each distinct row, the minimum duplication count between the two input tables. Combinations of the two representations are also possible, i.e., multiple copies each with a counter. Converting from one representation to the other is trivial in one direction and in the other direction requires any one of the well-known algorithms for executing “group by” queries.

RID	Old Ship Date	New Ship Date
[14:2]	2003-12-27	2003-12-26
[14:3]	2003-12-27	2003-12-26
[15:6]	2003-12-28	2003-12-27
[16:9]	2003-12-29	2003-12-28

Figure 4. Delta stream for the details table.

Duplicate counts are often used in materialized views and serve a function similar to reference counts in memory management. If the view definition includes a projection that renders the view result without a relational key, the duplicate counts assist in deciding when to remove a row from the view result. In fact, one approach to supporting materialized views in a commercial database management system is to require a key in the view result or a “group by” clause and a “count(*)” expression in the view result. The operation in the update plan maintaining the materialized view must delete the row when the duplicate count is decremented to zero.

Duplicate counts can also be employed in an intermediate result to represent the difference (the “delta stream”) that must be applied to a materialized view. Figure 4 through Figure 6 show intermediate results in an update execution plan that modifies both the table and the materialized view introduced in Figure 1 and Figure 2. Figure 4 shows the delta stream for the *line item* table upon the request *update lineitem set shipdate = shipdate – 1 day where orderno in (4922, 4956, 4961)*. Moreover, we assume that records in the table are identified by record identifiers (RIDs), which permit fetching any column not included in the delta stream. Figure 5 captures the same change preprocessed for maintaining an index on *shipdate*. Large updates can be applied to B-tree indexes more efficiently if the changes are sorted in the index order. Note that value changes for indexed columns imply insertions and deletions in the index, shown as *action* column.

Ship Date	RID	Action
2003-12-26	[14:2]	Insert
2003-12-26	[14:3]	Insert
2003-12-27	[14:2]	Delete
2003-12-27	[14:3]	Delete
2003-12-27	[15:6]	Insert
2003-12-28	[15:6]	Insert
2003-12-28	[16:9]	Insert
2003-12-29	[15:6]	Delete

Figure 5. Delta stream for an index on the details table.

2.14 Incremental view maintenance

Figure 6 shows the delta stream for the materialized view and its index. Rows in the view are identified here by their relational key, which might also be the search key in the view's clustered index. A second optimization has the query processor collapse multiple updates to the same row in the materialized view such that the storage engine applies and logs only one update for each record in the view's index. Positive and negative counts represent insertions and deletions and thus generalize the *action* column of Figure 5. When multiple changes apply to the same row in the materialized view, positive and negative values are simply added up to a single value (the "delta count"). This applies not only to counts but also to sums. If all counts and sums are zero, insertions and deletions cancel each other out such that no change is required to the stored record in the indexed view. Note that all counts and all sums must be equal to be zero in a delta record; it is not sufficient if only the count is zero, as it is to determine whether a stored record is a ghost record.

Commit Date	Ship Date	Delta Count
2003-12-24	2003-12-27	−2
2003-12-24	2003-12-26	+2
2003-12-31	2003-12-27	+1
2003-12-31	2003-12-28	±0
2003-12-31	2003-12-29	−1

Figure 6. Delta stream for an index on the view.

Among the standard aggregation functions, only *count*, *sum*, and *avg* can be maintained incrementally using the standard techniques for incremental view maintenance [BCL 86, BLT 86], the last one as *sum / count*. Sums of squares, sums of products, etc. can also be maintained efficiently in order to support variance, standard deviation, covariance, correlation, regression, etc. *Min* and *max* can be maintained incrementally during insertions but not always during updates and deletions, because a new extreme value must be found when the old one is deleted, whereupon summary rows for the affected groups must be recomputed [Q 96]. However, counts can be used to maintain reference counts for the current extreme value, thus ensuring that recomputation is triggered only when truly necessary. Thus, the design presented here applies not only to counts and sums but also in a limited way to indexed views with *min* and *max* expressions.

An alternative to instant maintenance is to mark affected summary rows in the materialized view as "out of date" without recalculating them, and to let queries invoke system transactions that apply all recent changes or recompute individual summary rows for groups marked "out of date." While possibly an interesting alternative, in particular for views with *min* and *max* expressions, we do not explore it here, instead focusing on techniques that incrementally keep all indexes on tables and views completely up-to-date at all times using serializable transactions.

2.15 Expanded "group by" clauses

Since escrow locking is not supported in most of today's database management systems, database administrators can work around lock contention among multiple update transactions by enlarging the view, e.g., augmenting the view definition such that multiple records together are equivalent to each summary row in the view of Figure 2. Each such record can be locked independently by the update transactions. Queries, however, pay the additional cost of summarizing results across all such records in order to derive a single row of Figure 2. Therefore, we consider these methods viable alternatives for database users to avoid contention, but they do not solve the contention problem to the same extent as our design does as described in the following sections.

In the example of Figure 2, assume that there is a small set of warehouses and each warehouse has only one process (no concurrent transactions) that set commit dates and ship dates. In that case, the "group by" clause in the view definition for Figure 2 can be expanded with a warehouse identifier, such that for each summary row in the unmodified view there is precisely one row for each warehouse in the modified view. Thus, updates can never interfere with each other or wait for record locks in the modified view. Of course, this work-around does not solve the problem of conflicting readers and updaters.

Alternatively, if all update transactions are small, i.e., affect only one or two rows in the base table and therefore only one or two summary rows, the additional grouping column does not have to be a real-world entity but can be the result of a randomizing function. Instead of a warehouse identifier, one could add "hash (OrderKey) % 10" to the view in Figure 2. In this case, each summary row in the unmodified view would be represented by up to 10 records, each update transaction would modify and lock typically only one of them, and queries would process them in order to produce the view of Figure 2. Unfortunately, this method works only for small update transactions that affect only a few orders; a bulk load operation that affects most summary rows in the original view will probably also affect most rows in the modified view, and thus block all small transactions that capture and summarize the on-going business activity.

Finally, one could invent syntax that directs each transaction to pick just one summary record at random and to increment only that summary row, e.g., "group by …, rand() %10". Quite obviously, such syntax does not exist in SQL. In fact, such a language extension would violate the spirit of ANSI/ISO SQL, which is designed to be a language about data contents rather than its representation. Just as the SQL standards contain syntax for table definition but not for index creation, such syntax would inappropriately mix information semantics and storage tuning. Moreover, the number of expected concurrent update transactions should determine the number of records per summary row. Unfortunately, the number of concurrent transactions varies quickly and widely in most systems, whereas the modulo calculation above is part of the database schema design and thus fixed.

3 Multi-version snapshot isolation

Combining and extending these prior techniques enables highly concurrent read and update operations on indexed views with aggregation. Focusing on the separation of readers and writers, let us now consider how pure read transactions and pure update transactions interact.

First, we consider whether running queries with a commit point equal to their start time is a reasonable and justifiable design decision. To do so, let us review whether a database server truly guarantees that query results reflect the most recent database state. The argument here covers multi-statement transactions processed in a single round-trip between client application and server. Interactive transactions with multiple user interactions as well as transactions including both queries and updates will be discussed later, since we believe they are the exception rather than the rule in monitoring applications.

After a query has been submitted to a database server as a single statement running as an implicit transaction, the server parses, compiles, and runs the actual query, and then assembles and sends network packets with result data. In traditional concurrency control schemes (assuming, as always in this paper, serializability, which implies repeatable read), traditional share locks are held until query execution completes; thus, the commit point is at the end of plan execution.

A user submitting a query and interpreting its result cannot determine the duration of each phase within the entire interaction with the database server. For example, it might be that query execution takes 90% of the time or it might be that 90% of the time is spent returning result data from the database server to the user. Thus, even traditional concurrency control methods guarantee only that a query's commit point is after submission of the query to the server.

Setting a query's commit point consistently at the beginning of query execution provides precisely the same guarantees with respect to currency of the query's result data. Note that query compilation, including insertion of the optimized query plan into a plan cache shared by all users and all transactions, should be a system transaction that does not acquire locks for the user transaction and thus does not affect that transaction's commit point.

Figure 7 illustrates the point. A user cannot, except with special performance management tools, distinguish among the three schedules, i.e., determine the time spent in each of the phases. Thus, any execution that is equivalent to one of these schedules is viable and acceptable to a reasonable user. Compilation may be omitted if the database management system employs pre-compiled query evaluation plans, or it may be modeled as very brief because query start-up requires finding a precompiled plan and validating it against the current data, e.g., using a schema version number. Computation might also be very brief if indexed views are readily available, possibly even in the buffer pool such that no physical I/O is required. Our model for concurrency control for read-only queries suggests that a user be presented with query results as if an extreme version of the right schedule in Figure 7 had been used.

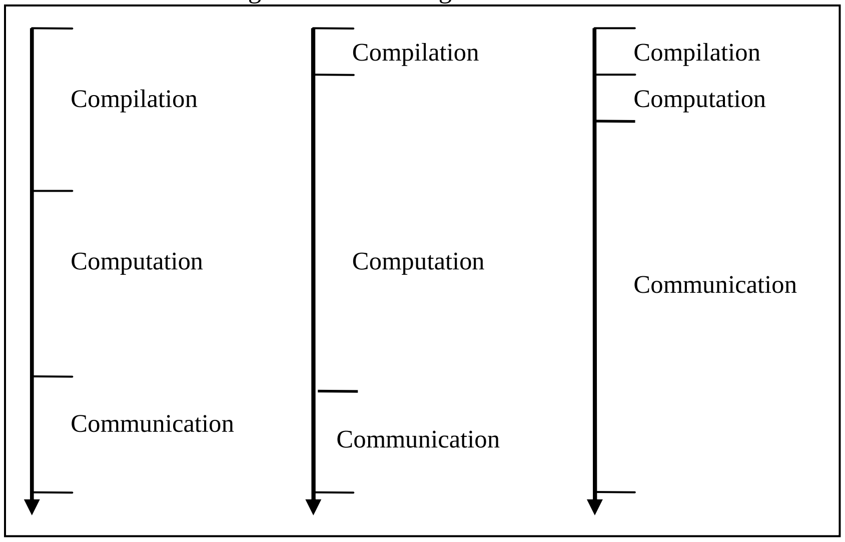

Figure 7. Timing in query execution.

The guarantees with respect to resource contention are much improved in this schedule compared to the other schedules. As queries "see" only data already committed before their execution, queries do not require locks. More importantly, the values for items in the database are actually decidable, whereas the outcomes of any uncommitted concurrent update transactions are, by definition, undecided. Thus, we believe that the proposed isolation method for queries is not only reasonable but in fact provides the best possible concurrency of query and update transactions without violating serializability.

It might be tempting to set a read transaction's commit point slightly later than the start of its plan execution. For example, not the query's start of execution but its first lock conflict could set the transaction's commit point. While this scheme might seem appealing, it requires read locks, i.e., items read without lock conflict must be locked in order to prevent subsequent updates that would otherwise undermine "repeatable read" transaction isolation. In other words, this scheme might create precisely the type of lock conflicts in materialized views between read transactions and update transactions that we set out to prevent.

In general, in order to process read transactions in serializable isolation yet without read locks, their commit point must be no later than their first read action. Given that there is practically no time between a query plan's start of execution and its first read action, our choice of commit points for read-only transactions is effectively the best possible to guarantee both serializability and maximal concurrency between read transaction and update transactions.

For transactions that both read and update an indexed view, serializability requires a single commit point, as discussed earlier. If the commit point is chosen as for read-only transactions above, i.e., when a transaction begins, a transaction must abort when attempting to modify an item that has been modified already by another transaction since the first transaction's commit point. If the commit point is chosen as for traditional transactions, i.e., when the transaction actually attempts to commit, all items read must be locked against updates in order to ensure repeatable reads. This dilemma is not avoided by escrow locks or any other standard method for concurrency control; thus, the present design does not offer any new technique for transactions that both read and update an indexed view.

In fact, escrow locks and thus multiple concurrent update transactions are in conflict with traditional multi-version snapshot isolation, because snapshot isolation requires a single linear history, which at first sight requires a single update transaction at a time (for each data item), i.e., traditional exclusive locks. The next section describes our solution to this ostensible contradiction between snapshot isolation and escrow locking.

4 Concurrent updates and linear version history

Prior schemes for snapshot isolation and multi-version concurrency control require a single linear sequence of consecutive values, and that each item's versions thus represent this one history of committed values of a data item. Note that a single history implies that a new version is not created until the most recent one has been committed, i.e., that multiple concurrent updates are not permitted. If multiple updates (to the same item) can be processed yet remain uncommitted at the same time, which is the essential value of escrow locking and of equivalent techniques, the assumption of a single sequential history is not justified.

Step	Transaction 1	Transaction 2	Transaction 3
1	Begin	Begin	
2	Insert two new orders	Insert a new order	
3	E lock, increment by 2		
4		E lock, increment by 1	
5		Commit	
6			Begin
7			Read summary row

Figure 8. Sample transaction schedule.

For example, consider the schedule shown in Figure 8 and based on the materialized view of Figure 2. The initial state includes one order with CommitDate = 2003-12-31 and ShipDate = 2003-12-29. Transaction 1 inserts two more orders and Transaction 2 inserts one more order with the same dates, and both transactions increment the count of shipments in the appropriate summary row. With its snapshot point set in step 6, Transaction 3 needs to see the value 2 as the current shipment count, but nowhere in the versions generated by the schedule is the value 2 found.

Thus, escrow locking and multi-version concurrency control have an inherent incompatibility, for which this section offers three alternative resolutions. While the first approach is likely easiest to implement, because actually it does not require escrow locks, we recommend the third approach, because it retains all update processing and overhead within update statements and thus will likely result in the best query performance for indexed views and data warehouses.

Following discussion of these three alternative methods, we discuss details and variations of the third method. We introduce new commit-time-only locks, develop appropriate commit processing and transaction abort, and finally integrate escrow locks and commit-time-only locks with transaction save points and with two-phase commit. Subsequent sections complement the third method with logging and recovery, multi-granularity lock, online index operations, etc.

4.1 Multiple records per summary row

Our first approach employs, when necessary, multiple records to represent a single row in a summary view. Recall that duplicate rows in an intermediate query result can be represented by multiple copies, by a single copy with a duplicate counter, or by multiple copies each with a duplicate counter, and that it is then left to subsequent query operations to interpret these records appropriately. For example, an "intersect all" query might include a preparatory operation that forces multiple copies, with or without counter, into a single copy with a counter.

The same idea applies to aggregation views with counts and sums, because materialized views are quite literally intermediate query results. This approach permits multiple records, each with counts and appropriate partial sums, rather than strictly requiring that a single row in the view is represented by a single record in each index on the view. A query over this kind of index is always obligated to perform the final summation of counts and sums. Note that multiple records for a single summary row are not the default, and that in an ordered index, multiple records contributing to a single row will typically be immediately next to each other, thus reducing the processing burden on queries.

In the example of Figure 8, Transaction 1 would X lock and update the shipment count for the existing row in the view, changing the count from 1 to 3. The old value of 1 remains available as a prior version of that row. Transaction 2 would find the lock held on the row and instead insert a new row with a shipment count of 1. Transaction 3 would then add the shipment counts from the newly inserted row by Transaction 2 and from the original row, i.e., prior to the update by Transaction 1, and produce the correct output value 2.

Update operations for any specific row within the view attempt to lock any one of the existing records for that row, typically the one-and-only existing record. Only if this lock request fails due to concurrency contention, a new record is created and locked. Thus, contention among multiple updaters for any one specific record can always be avoided, even if contention for a specific row within the view is high. Since concurrent increment and decrement operation on the same record are not required, these updates can actually employ traditional X locks; escrow locks are not required. Thus, each record has a single sequential history that can be represented in traditional multi-version concurrency control, and a query can determine a count or sum in a row at a specific time by adding up counts and sums in all records representing that row at that time.

At most times, each row is represented by a single record. When and where this is not the case, system transactions attempt to collapse multiple records into one. This process is reminiscent of traditional ghost clean-up, with comparable and thus overall acceptable overheads. One important difference is that clean-up must distinguish between increments older than the oldest current read transaction and newer ones; it may collapse all value changes due to the older increments but it must preserve the precise history represented in the newer increments because these might still be required in queries. Thus, it might be pragmatic to clean up only records and their versions when they are older than the oldest current read transaction.

To summarize this first approach, multiple records for each row in the summary view are employed for two purposes: sequential histories and concurrent partial counts and sums. This approach eliminates the need for escrow locks and thus is compatible with traditional versioning schemes, enabling snapshot readers and multiple concurrent updaters for any row or all rows in a summary view. Its costs are an additional asynchronous consolidation activity as well as residual consolidation within retrieval queries. The novelty of this approach is that it dynamically adjusts the number of records representing a single summary row in response to increasing or decreasing contention for that specific summary row, and that it thus sidesteps all contention without requiring unconventional lock modes such as escrow locks.

4.2 Multiple delta records per summary row

A second approach to the problem represents each summary row by a single "base record" that holds an absolute value for each count and sum, but relies on an alternative method to capture the actual increments and decrements ("delta records") rather than storing earlier values in the multi-version store. This scheme is similar to the recovery scheme described for the original escrow locks [O 86]. Just as a version store permits clean-up and consolidation when update transactions commit or abort, delta records can be erased after committed changes in the version store have been consolidated into the base record. This consolidation is not required within each user transaction; an asynchronous clean-up process can perform these actions. Queries against the version store must aggregate the base record and all delta records left behind by committed transactions.

Again referring to the example of Figure 8, the update by Transaction 1 would leave the prior record (with count 1) in place yet add another delta record with value 2. Similarly, Transaction 2 adds another delta record with value 1. Transaction 3 would read all committed records and adds the counts, producing the correct value 2. Alternatively, a consolidation clean-up after the commit of Transaction 2 and before the start of Transaction 3 might already have combined these two records, making the correct value 2 immediately available to Transaction 3.

4.3 Deferred actions and virtual log records

A third approach relies on intention lists of deferred actions and serial commit processing applying these deferred actions to guarantee a single linear history of committed values for each row in the view as well as a single linear history for the entire database. Each row in the materialized summary view is represented by a single record in each index on the view. A virtual log record is created when, under the protection of an escrow lock, an increment or decrement operation is permitted but not yet applied. The virtual log record contains the delta by which to increment or decrement counts and sums in a row in the indexed summary view, and is therefore quite similar to a *redo* log record, albeit a *redo* of a logical rather than a physical operation. The increment operation captured in the virtual log record is applied in the commit phase of the transaction – after the application requests the commit but before the final commit record is generated. The virtual log records are applied using ordinary low-level transactions that include writing physical log records. Thus, only commit processing applies increment and decrement operations to records in a materialized view and its indexes.

Using the example schedule of Figure 8 for a third time, the stored value for the count remains unchanged in steps 3 and 4, and is incremented from 1 to 2 only in step 5, during commit processing of Transaction 2. Transaction 3, with its snapshot time set in step 6, will correctly see the value 2 in step 7.

Logic similar to commit processing for a single summary row is required when a transaction upgrades an E lock to an X lock. Recall that this lock upgrade is required if a transaction attempts to read a summary row it just incremented or attempts to overwrite it with a specific new value, i.e., an assignment rather then an increment operation. Upgrading an E lock to an X lock requires applying the change protected by the E lock as well as waiting until any other E locks are released, in either order. In order to speed processing in those cases, each virtual log record should be attached to the data structure representing the associated lock. If this design is employed, it is also possible to ensure that each summary row and each escrow lock has only one associated virtual log record, even if multiple statements within a transaction increment or decrement that summary row.

If commit operations are serial as in IMS/FastPath, increment operations performed during commit processing will give each record as well as the entire database a serial history, which can readily be supported by traditional version stores and which is compatible with multi-version snapshot isolation. The experience of IMS/FastPath [GK 85] indicates that serial, one-transaction-at-a-time commit processing is practical and can be implemented extremely efficiently, and in fact can scale to very high transaction rates, assuming the typical FastPath environment in which many small transactions modify only very few records each. If each insert, update, or delete operation affects an individual row in a table underlying a summary view, this assumption is definitely justified. Even updates in the view's grouping columns, which force a decrement operation in one group and an increment operation in another group, still justify this assumption. Even the additional effort imposed by versioning, i.e., copying the pre-increment value to a version record, probably still does not invalidate the assumption. Thus, serial commit processing most likely suffices for many applications, databases, tables, views, and their indexes.

4.4 Commit-time-only locks

Large transactions with many virtual *redo* log records, however, clearly violate the assumption that each transaction modifies only very few records. A typical example is a bulk operation that affects a large fraction of the view, e.g., a large data import (load) into one of the underlying tables. Note that even if the fraction of updated rows in the underlying table is relatively small, the fraction of affected rows in the view might be quite large. Another example is "catching up" a materialized view that has fallen behind due to deferred maintenance, i.e., in systems that fail to keep tables and materialized views over those tables synchronized. If bulk operations were to be supported with many virtual log records, IMS/FastPath could not use the single latch protecting the recovery log and still achieve high transaction throughput. Serial commit processing for a single large transaction would prevent and delay commit processing for many small transactions, which directly contradicts the purpose and value of escrow locking.

	S	X	E	C
S	√	–	–	√
X	–	–	–	√
E	–	–	√	√
C	√	√	√	–

Figure 9. Lock compatibility matrix including E and C locks.

In order to support large transactions against indexed summary views, we extend the design above such that multiple transactions can process their commit activities concurrently. To that end, we introduce commit-time-only exclusive locks that permit serializable instead of serial commit processing. Importantly, these commit-time-only locks do not conflict with locks acquired during normal transaction processing, specifically with escrow locks. Thus, transactions not yet in their commit process are not affected by these commit-time-only locks. Serializing data access during normal transaction execution and serializing concurrent commits are two separate concerns with separate locks. Note that these commit-time-only exclusive locks are needed in addition to the escrow locks acquired during normal transaction execution.

Figure 9 illustrates the new commit-time-only exclusive locks, labeled “C” in the compatibility matrix. They are compatible with all other locks type except other C locks. Using this definition and compatibility matrix, a transaction must acquire a C lock during commit processing in addition to the E lock acquired while updating the materialized and indexed view.

	S	X	E	C
S	√	−	−	−
X	−	−	−	−
E	−	−	√	√
C	−	−	√	−

Figure 10. Lock compatibility with upgrade from E to C.

Alternatively, the C lock can be defined as a combination lock, i.e., as E+C in Figure 9. This is illustrated in Figure 10. Most importantly, C locks remain compatible with E locks. The main difference to the prior lock compatibility matrix is that a transaction avoids adding new locks; instead, it converts existing E locks to C locks. Other than that, semantics are not changed. In particular, concurrency among transactions is not changed. Very importantly, C locks continue to be compatible with E locks, i.e., one transaction may process its commit while other transactions still hold E locks on the same data.

Upgrading from one lock type to another might benefit from an intermediate upgrade lock, analogous to common “U” lock between S and X locks. While true in general, the upgrade from E to C locks is special, for two reasons. First, C locks are typically held only a short time, and contention for C locks is likely low. Recall that IMS/FastPath has, in effect, only a single C lock for the entire system, i.e., the exclusive latch protecting access to the transaction log. Second, acquisition of C locks can be organized in such a way that it is guaranteed to be free of deadlocks. This is possible because C locks are acquired only during commit processing, when the set of required C locks is known and can be sorted.

4.5 Commit processing

Commit processing proceeds in multiple steps, starting with the user’s request to commit. First, all read locks are released as in traditional commit processing. Second, E locks are upgraded to C locks. Third, virtual log records are applied and their effects logged like updates covered by ordinary X locks. Finally, the transaction is truly committed using a traditional commit record in the transaction log, and all remaining locks are released.

These basic steps require some further comments. Most importantly, in order to avoid deadlocks while upgrading E locks to C locks, the existing E locks should be sorted. The sort order does not really matter, as long as it is the same for all transactions trying to commit. The sort key typically will include index identifier and index key, and it might include the hash value with which each lock is stored in the lock manager’s hash table. The hash bucket number within the lock manager’s hash table could be used as artificial leading sort column, a technique that avoids most of the sort effort by sorting once per hash bucket rather than sorting all necessary lock requests in one large sort.

In addition to sorting the E locks prior to their upgrade to C locks, it might make sense to sort the virtual log records, e.g., by index identifier and index key, and to aggregate multiple virtual log records pertaining to the same summary row, if any, into a single change. This optional sort operation ensures that, if multiple virtual log records apply to the same page or even the same summary row in an indexed view, these changes are applied efficiently with beneficial effect on I/O buffer, the CPU caches, transaction log size, etc. This effect, of course, is quite similar to optimized index maintenance described elsewhere [GKK 01] and also implemented in some products, and not really unique to deferred actions or virtual log records.

Traditionally, acquiring locks during commit processing has been an anathema just like acquiring locks during transaction rollback. However, because deadlocks are avoided due to sorting the required lock requests, it is not required that these commit-time-only locks are full traditional locks, with deadlock detection, etc. Instead, much less expensive latches will suffice. For record-level or key-value latching, these latches can be attached to data structures in the lock manager, namely those used to coordinate escrow locks and other locks for a record or key. Moreover, it is not required to employ record-level locking during commit processing; depending on the desired achievable degree of concurrency, records can be grouped into pages, page extents, key ranges, etc. such that a single latch covers application of multiple virtual log records, although this introduces new challenges.

If commit-time-only exclusive locks are obtained on precisely the same resources as the escrow locks, then absence of deadlocks is guaranteed. If, however, escrow locks are obtained in fine granularity (e.g., individual keys) and commit-time-only exclusive locks are obtained in coarser granularity (e.g., large key ranges), then there might be conflicts, namely if some other transaction holds an X lock within the larger granule. Alternatively, the commit-time-only exclusive lock could be defined and interpreted to cover only those resources for which the transaction also holds an escrow lock, e.g., the individual keys within the large key range. In other words, this case must employ the lock compatibility matrix shown in Figure 9, not the one in Figure 10.

One might ask what needs to happen if an E lock has been combined with an S lock, i.e., it has been upgraded to an X lock? In that case, the X lock and its traditional commit processing prevail. Note that there shouldn't be any remaining virtual log records for the item upgraded from an E lock to an X lock, because any remaining virtual *redo* log records must be applied at the time of the upgrade to an X lock and no further virtual log records will be generated after the transaction holds an X lock.

Also, can E locks be released after C locks have been acquired? That depends on the compatibility of C locks. If C locks conflict only with other C locks, as shown in Figure 9, E locks must be retained. If these C locks imply all conflicts of E locks, as shown in Figure 10, then E locks can be upgraded to C locks. Note that deadlocks are still impossible because a transaction holding an E lock guarantees that no other transaction can hold an S or X lock.

4.6 Transaction abort

If a transaction fails to commit and instead aborts, the sequence of actions is much simpler. Interestingly, E locks can be released immediately together with S locks, because the actions protected by E locks have not been applied to the database yet. Similarly, deferred actions described in virtual log records are simply ignored, because their effect is no longer desired. Only X locks need to be retained and their actions compensated by traditional transaction rollback, including any X lock that resulted from adding an S lock to an E lock. C locks are not required when processing a transaction rollback.

4.7 Transaction save points

Rolling back to a save point, however, deserves special attention. Releasing recently acquired locks when rolling back to a save point is an effective technique for improving concurrency and even resolving deadlocks already employed in some commercial database management systems, and should remain possible in the presence of E locks. Rollback of C locks does not make sense, since these exist only during commit processing, i.e., the user or application has already relinquished their right to control the transaction.

Similar to transaction abort, recently acquired S and E locks can be released immediately, recently deferred actions and their virtual log records can be ignored, and recently acquired X locks are rolled back as in traditional partial rollback.

The situation is more complex if an E lock has been augmented with an S lock since the save point, i.e., the E lock has been upgraded to an X lock. In this case, the X lock must be downgraded back to an E lock. This is not strictly required for the purposes of isolation, given that the X lock provides more protection and isolation than the E lock. For the purpose of logging and recovery, however, in particular if the transaction is to support subsequent rollback to even earlier save points as illustrated earlier in Figure 3, the increment operations and their virtual log records must be re-established as part of the downgrade from X to E, as will be discussed shortly in the section on logging and recovery.

4.8 Two-phase commit

The goal of the preparation phase in two-phase commit is to ensure that the local transaction can abide by the global decision to commit or to abort. In traditional two-phase commit, this implies that all actions are complete except writing the final commit to the transaction log. Shared locks and upgrade locks can be released during the preparation phase, whereas exclusive locks must be retained until the commit record is safely on stable storage.

Applying this logic to the commit processing sequence discussed above, a transaction must retain its X and C locks until the global transaction outcome is confirmed and logged. E locks must be upgraded to C locks or must also be retained, i.e., if the design follows Figure 9 rather than Figure 10. The C locks are retained until the global transaction outcome is decided and propagated to the participating resource managers. S locks can be released as they are not affected by escrow locking, whether or not escrow locking is employed by a transaction, and X locks must be retained just as in traditional transaction processing and two-phase commit.

In addition to considerations about locks, two-phase commit also affects logging. Thus, the following section about logging and recovery will resume discussion of two-phase commit processing.

5 Logging and recovery

The original description of escrow transactions included somewhat complex methods for recovery from failures of individual transactions, storage media, or the entire database management system. In fact, one might surmise that the complexity of logging and recovery prevented the adoption of escrow locking in mainstream database management systems. Fortunately, multi-level transactions and Aries, suitably adapted to E and C locks as described above, can substantially simplify the required protocols and their implementation.

Because transaction abort is simpler than transaction commit, abort and rollback are covered prior to commit processing and recovery from system and media failures, including those that catch a transaction already in commit processing. Two additional sections cover two-phase commit and a side note on optimistic concurrency control.

5.1 Transaction abort

Rolling back an individual transaction to its beginning is actually less expensive than in traditional update processing using X locks, assuming that the actual application of increment operations is deferred until commit processing using virtual log records. This is because neither *undo* action nor compensation is required for deferred updates. The updates deferred using virtual log records have not been reflected yet in either the transaction log or the database, neither in the buffer pool nor on disk. Thus, no corrective action is required when a transaction aborts. The virtual log records are discarded without further action, quite similar to the corresponding E locks as discussed above.

5.2 Partial rollback for save points

Support for partial rollback is a bit more complex. If there are multiple updates to the same row in the summary view and thus the same record in the view's index, and if save points are interspersed with these updates, it is necessary to retain multiple virtual log records. During commit processing, sorting all virtual log records permits aggregating those such that each database record is actually updated and logged only once. Until then, however, aggregation must not cross save points that might still be the target of a partial transaction rollback. Aggregation within each interval between save points is acceptable, and might well be sufficient to capture most of the performance improvement of aggregating virtual log records. Aggregation across anonymous save points is also possible after those save points have become irrelevant. For example, if a save point is established to enable statement abort without transaction abort after, that save point can be ignored after the statement is completed successfully.

Rollback of a lock upgrade from E to X mode needs to consider the case that there might have been multiple save point intervals with increment operations under the E lock in each interval. If so, all those increment operations must be applied when the E lock is upgraded to an X lock, such that a read operation by the current transaction indeed reflects all the updates.

In the discussion so far, it has been implied that the pertinent virtual log records can be dropped once they have been applied to the database. Unfortunately, this would not be correct if subsequent rollback to a save point is to be supported. Recall that after rollback to a save point and subsequent processing, rollback to an even earlier save point might be requested.

If a lock upgrade from E to X is rolled back, the individual increment operations must be re-established within their original save point intervals, in order to recreate the situation prior to the lock upgrade. Perhaps the simplest implementation retains the virtual log records during the lock upgrade but marks them "already applied," such that rollback can revoke those markings. More precisely, the update that applied the cumulative increment during the lock upgrade is undone or compensated, and the relevant virtual log records are either deleted or their marking is revoked, depending on these virtual log records' time of creation relative to the save point targeted by the current rollback.

5.3 Transaction commit

Virtual log records cannot contribute to the durability of transactions after their commit. Instead, they remain attached to the transaction's in-memory data structure when they are created during increment and decrement operations under the protection of E locks, and they are not written into the transaction log on stable storage.

Durability is guaranteed by the log records written while an actual change is applied to the database under the protection of commit-time-only exclusive locks. These C locks, together with the database updates applied during commit processing and their log records, resemble very closely the changes applied in traditional concurrency control and recovery under traditional X locks. The difference is in the locking behavior and in the deferment of actual updates using virtual log records. In other words, the difference is in the timing when these updates are applied, the difference is not in the mechanisms how these updates are applied, logged, and made durable. Thus, the same rules about write-ahead logging, buffer management, etc. apply, with the same effects.

Specifically, the log records created for updates protected by C locks must obey the standard rules of write-ahead logging. They must be written to the transaction log on stable storage before the modified data pages may overwrite old database contents on disk.

For all transactions other than those at the lowest level in the multi-level transaction hierarchy, it is not required that *redo* and *undo* actions be idempotent, since the logging and recovery protocol ensures that a *redo* action is applied only when the database does not contain any effects of the action being redone, and that an *undo* action is applied only when the database contains all effects of the action being undone. As implied earlier in connection with virtual log records, the increment and decrement operations should be higher-level actions that are realized by lower-level transactions that actually assign specific new values within the records on disk. Thus, lowest-level “physical” log records contain absolute values just as in traditional transaction processing using X locks, whereas higher-level “logical” log records contain deltas.

If deemed desirable, it is not even required that entirely new formats for log records be designed. In fact, a traditional log record describing the update of a database field is sufficient for logging and recovery of escrow updates, with only a single bit indicating that the operation is a relative or escrow operation. The old and new values in a traditional log record must be interpreted as a numeric difference, positive or negative, and this difference must be applied when compensating an action during transaction rollback. In fact, formatting a log record of this type could be leveraged twice, first for the virtual log record that defers forward processing and then as for the log record on stable storage while applying the virtual log record during commit processing.

Copying the virtual log records to the persistent transaction log is not required. Logging the virtual log records could, however, enable an interesting optimization, albeit possibly with some risk. As soon as these virtual log records are logged, the entire transaction can be reported completed and committed to the user or application, which therefore do not need to wait until all changes and virtual log records have been applied to the individual records and summary rows in all indexed views affected by the transaction. Should the system fail after reporting completion to the user, the changes can be applied during restart, very similar to a global commit decision by a distributed transaction coordinator. This technique might seem promising, but the implied risks require further research. For example, should a failure occur while the logical log records are applied to the database, what are the appropriate next steps for the transaction and for the database affected?

5.4 System and media recovery

For the most part, checkpoints as well as system and media recovery follow the well-known Aries techniques [MHL 92]. Logical *redo* log records are used during recovery only for unfinished distributed transactions. If the recovery log contains a pre-commit or prepare log record but no final log record for that transaction, and if the transaction coordinator logged a commit record, all logical *redo* log records are processed as in normal commit processing, i.e., they are applied to the data records using low-level transactions that write standard physical log records. *Redo* processing for distributed transactions is interleaved with *redo* processing of local transactions based on the sequence in the recovery log, i.e., there is only one *redo* pass even if the system was finishing distributed transactions at the time of the server crash.

Other than that, the *redo* pass relies not on virtual log records but on the physical log records written to the recovery log by low-level transactions while processing virtual log records. The usual recovery optimizations apply: For example, if locks are taken during the analysis pass, new transactions can be accepted and processed concurrently with both the *redo* and *undo* passes. If locks are taken only during the *redo* pass, e.g., because log analysis is integrated and interleaved with the *redo* pass, concurrent transactions must wait until the *redo* pass is complete, i.e., until *undo* or compensation processing has begun. In general, if locks are reacquired before the *undo* pass, transactions can be allowed to enter the system as soon as the *redo* pass is complete.

Checkpoint processing is also not affected by escrow locks, multiple concurrent updates, multiple concurrent commits, deferred updates, or virtual log records. In fact, the techniques discussed in this paper do not interfere with checkpoints during the analysis, *redo*, and *undo* phases of restart recovery, with fuzzy checkpoints (e.g., second chance checkpoints that write out only those pages that have already been dirty during the prior checkpoint), or with checkpoint intervals (with separate *begin checkpoint* and *end checkpoint* log records, and active transaction processing while taking the checkpoint [MHL 92]). In Aries and multi-level transactions, checkpoints only affect (i.e., shorten) the *redo* effort to apply traditional physical log records.

Media recovery also employs Aries mechanisms, using fuzzy image copies obtained during active transaction processing, combined with a suitable period of the recovery log. During media recovery, the log is applied to the database restored from the image copy, including transaction *undo* and compensation [MHL 92]. Even single-page recovery (e.g., after a "bad block" disk error) or continuous database mirroring ("log shipping") are compatible with escrow locks, multiple concurrent updates, deferred updates, and virtual log records.

5.5 Failures and transaction abort during commit processing

If a transaction aborts prior to commit processing, the discussion above applies – escrow locks and virtual log records are not applied to the database and instead are simply ignored, and traditional locks and log records are processed in the traditional way. What wasn't covered above was transaction failure during commit processing, e.g., if lock conversion from E to C locks fails for some unexpected reasons (e.g., out of memory) or logging a change fails for some unexpected reasons (e.g., out of log space).

It turns out that the problem is simpler than perhaps expected, due to the similarity of C locks to traditional X locks and the parallel similarity of logging increment operations during commit processing to logging traditional update operations. If failure occurs during commit processing, some virtual log records have already been applied including generation of traditional log records, and some have not. For the former, abort processing follows the same rules as abort processing described earlier: the virtual log records are ignored and simply deallocated. For the latter, abort processing follows traditional lines: using the traditional log record generated from the virtual log records and applied under protection of a C lock, the update is compensated and the compensation action logged.

In summary, the commit-time-only exclusive locks ensure that any index entry can be updated by only one transaction currently in its commit processing phase. Thus, even if a transaction is forced to roll back after some of its commit processing has completed, it can do so with ordinary transaction mechanisms. Special, non-traditional logging and rollback mechanisms are not required.

5.6 Two-phase commit

In addition to upgrading E locks to C locks, two-phase commit needs to apply the changes protected by the C locks and log them. The first or "preparation" phase of two-phase commit performs all operations of commit processing up to yet excluding writing the commit record. Instead, as in ordinary two-phase commit, it writes a "prepared" record to the transaction log. When the global commit coordinator's final commit decision is known, commit processing resumes as discussed above. Alternatively, abort processing takes over as if the local transaction had failed at the end of commit processing, just before the final commit record was to be written – just as in traditional two-phase commit processing.

The notion of virtual log records, which more accurately can be called virtual *logical redo* log records, opens another possible optimization that might improve concurrency as well as response time for the preparation request. In spirit, this optimization is similar to the "vote reliable" optimization implemented in some IBM transaction processing products [SBC 93]. In general, an operation is considered durable (in the sense of transactional "ACID" guarantees) when its effects and updates are safely captured in the transaction log (on "stable storage"), whether or not these effects and updates are already reflected in the database. If a failure happens, redoing the actions described in the transaction log will recover their effects and thus ensure their durability.

Thus, it could be argued that capturing the virtual logical redo log records not only in the transaction's in-memory state but also in the transaction log is sufficient to report a local transaction ready to abide by the global decision about transaction commit or abort. In fact, this idea might be applicable in additional situations and cases. Of course, care must be taken to avoid unacceptable additional risks to transaction completion, e.g., out-of-memory or out-of-disk-space errors that might be encountered while performing the actions described in these *logical redo* log records, and that would have caused the local transaction to veto against global transaction commit if these errors had happened during the local transaction's preparation phase.

5.7 Optimistic concurrency control

Interestingly, and added here only as a side note, many of the commit processing techniques described above also apply to transactions governed by optimistic concurrency control [KT 79, KT 81].

A transaction under optimistic concurrency control proceeds with its work even if it might interfere with other active transactions. Employing temporary copies of data or temporarily accepting inconsistent data, end-of-transaction validation ensures that only valid transactions commit, i.e., all transaction schedules are serializable, by comparing a transaction's "read set" and "write set" with other transactions' read sets and write sets. In addition to maintenance of these read sets and write sets, the major downside of optimistic concurrency control is that work might be wasted even after it would have been possible to detect the interference with other transactions and to resolve that interference by waiting or by an earlier transaction abort.

Implementation of end-of-transaction validation uses some form of locks or latches to ensure consistency and validity of the validation step; the original research [KT 79] relied on critical sections. However, it could be argued that optimistic concurrency control simply defers lock acquisition until commit processing. A significant difference to pessimistic concurrency control is that the set of required locks is known at commit time in optimistic concurrency control, meaning the lock set can be sorted. As in commit processing for escrow locks above, this knowledge permits avoiding deadlocks with other transactions committing under optimistic concurrency control [HD 91]. Moreover, because commit processing takes only a fraction of the time needed for processing an entire transaction, coarser lock granularity is feasible without excessive contention.

After all locks have been acquired and transaction validation is complete, the transaction can release all read locks immediately, then perform and log the updates, finish by writing the commit record, and finally release the write locks. A transaction under optimistic concurrency control switches into a commit processing mode very much like pessimistic concurrency control and very much like transactions converting escrow locks to exclusive locks during commit processing. In both cases, i.e., optimistic concurrency control and transactions with escrow locks, these exclusive locks are limited to commit time only, and they interfere only with other commit-time-only locks held by transactions equally in their commit processing phase.

Maintenance of the transaction's read set and write set, i.e., maintenance of the set of required locks, can be implemented using a data structure very much like the data structure used in transaction-private memory for possible lock de-escalation as well as like the data structure for the lock manager. Thus, bookkeeping in transactions under optimistic concurrency control is more similar to bookkeeping for transactions under pessimistic concurrency control than usually understood. At least one research effect, however, suggested locks and a lock manager to determine the intersection between read sets and write sets, using a "dummy" lock mode that does not conflict with any other lock and only serves as a witness for a transaction's read operation [HD 91]. A refinement of that approach might employ notification locks [GR 93] instead of dummy locks.

Interestingly, not only concurrency control but also logging and recovery for optimistic concurrency control might be surprisingly similar to the mechanisms discussed above for escrow locks. Specifically, virtual log records seem a promising implementation technique to defer updates until they have been validated during commit processing, with its potential for performance and scalability proven in IMS/FastPath.

Given these similarities, it seems that mixing optimistic and pessimistic concurrency control might be easier to realize than one might suspect, and the choice of concurrency control methods can be made for each transaction rather than for the entire system and its implementation. It is even conceivable to choose the concurrency control method for individual resources, if desired. A candidate might be schema stability versus data operations. Protecting a transaction against changes in the database schema or at least the schema of the relevant tables could use traditional locking (with hardly any contention and waiting for locks) whereas queries and updates may employ optimistic concurrency control for the tables' contents. In the extreme, a transaction may lock an individual data items after accessing it repeatedly under optimistic concurrency control; or a database management system may learn which resources to manage with optimistic concurrency control and which ones to manage with pessimistic concurrency control.

Lausen investigated very similar ideas on formal level [L 82] and also came to the conclusion that optimistic concurrency control and pessimistic concurrency control can be combined, even to the point that a single transaction locks some resources and validates others at the end of the transaction. Managing an optimistic transaction's read set and write set in transaction-private memory using the same type of data structures employed by a traditional lock manager might be the implementation technique needed to make Lausen's ideas realistic and perhaps address some of the concerns about optimistic concurrency control [H 84, M 92].

Two-phase commit after optimistic concurrency control, i.e., optimistic concurrency control participating in distributed commit coordination, can also benefit from techniques borrowed from pessimistic concurrency control. In particular, the preparation phase requires that the transaction acquire the appropriate locks, such that the local transaction is entirely subjugated to the global commit decision. Read locks can be released at the end of validation and write locks must be retained until the global commit decision is known and reflected in the local database. In other words, in the commit sequence above for optimistic concurrency control, writing the commit record is replaced by writing the pre-commit record, voting, receiving the global decision, and then committing or aborting.

6 Multi-granularity locking

One of the proven database implementation techniques found in all commercial database management systems is multi-granularity (hierarchical) locking. When updating all rows in a table or all records in an index, rather than taking row locks or key locks for each one, a single "large granularity" lock for the table or index is taken. In order to detect conflicts correctly, a transaction that modifies only a single row (and thus intends to take a row lock) must obtain an "intention" lock at the larger granularity. Note that two intention locks never conflict, because two intention locks indicate that both participating transactions intend to take locks at a smaller granularity where actual concurrency conflicts will be detected if they indeed exist. At least one absolute lock (as opposed to an intention lock) is required for any lock conflict. Note also that a reasonable interpretation of an absolute lock at a larger granularity is equivalent to that same lock on each instance at the smaller granularity, including phantom instances. For example, a large read transaction holding an S lock on a table implicitly holds an S lock on all rows, including ones to be inserted after the initial S lock on the table has been granted – since an insert transaction must hold an X lock on the new row, which is incompatible with the S lock held implicitly by the large read transaction even on the new row, the insert must be prevented, which is why S and IX locks on a table conflict with each other.

For escrow locks and escrow transactions, two obvious questions are: If summary records in an indexed view are usually modified using escrow locks, can other transactions still employ large granularity locks when appropriate? Second, if a large transaction, e.g., a bulk insertion into a table of detail records, modifies practically all summary records in an indexed view, is it possible to obtain a single escrow lock on the entire materialized view or the entire index of the materialized view?

6.1 Derived lock types

Korth has described a fairly straightforward procedure for deriving auxiliary lock modes from a set of basic operational locks [K 83], including intention locks, combination locks, and upgrade locks. Thus, the answer is "yes" for both questions above. For the first question, we must define a new "intent to escrow" ("IE") lock mode that all escrow transactions must obtain for the materialized view or its index before requesting an escrow lock for a specific row in a summary table or for a specific key in an indexed view. This IE lock does not conflict with other intention locks, but it conflicts with both shared and exclusive locks on the view or index. Similarly, we ought to define a new "intent-to-commit" ("IC") lock mode that permits efficient multi-granularity locking during commit processing.

Figure 11 shows a traditional lock compatibility matrix extended with escrow locks ("E"), intent-to-escrow ("IE") locks, commit-time-only exclusive locks ("C"), and intent-to-commit locks ("IC"). The assumption in Figure 11 is that E locks are upgraded to C locks, i.e., C locks are obtained in place of E locks rather than in addition to E locks.

Note that the extension from S and X locks to S, X, E, and C locks might warrant multiple kinds of upgrade locks. For example, the traditional U lock is an S lock with the exclusive permission to upgrade to an X lock. After the introduction of E locks, there must be two upgrade locks, namely from E to X in addition to the traditional upgrade from S to X. Upgrade from S to E is not required because an E lock does not imply an S lock, and because an S+E combination lock is equivalent to an X lock. Update or upgrade locks

("U" and "IU") are not included to limit size and complexity of the diagram, even if they are a standard technique in commercial database management systems.

Traditional combination locks ("S+IX", etc.) are also not shown in Figure 11 as they can easily be inferred: a requested combination lock is compatible with a prior lock if both components of the combination are compatible, and a new lock request is compatible with a prior combination lock if the new lock is compatible with both components of the combination lock. Some combination locks might not increase concurrency and should thus be omitted: just as an S+X lock does not make any sense, an S+E lock is like an X lock in its conflict patterns, and transactions should obtain an X lock rather than separate S and E locks on the same resource or their combination. Similarly, IS+IE is equivalent to IX, which implies that locking a large granularity of locking in IX mode grants permission to lock at the next smaller granularity of locking not only in traditional X, S, IX, and IS modes but also in E or IE modes.

	S	X	E	C	IS	IX	IE	IC
S	√	–	–	–	√	–	–	–
X	–	–	–	–	–	–	–	–
E	–	–	√	√	–	–	√	√
C	–	–	√	–	–	–	√	–
IS	√	–	–	–	√	√	√	√
IX	–	–	–	–	√	√	√	√
IE	–	–	√	√	√	√	√	√
IC	–	–	√	–	√	√	√	√

Figure 11. Lock compatibility including intention locks.

6.2 Interpretation of the lock table

The compatibility matrix of Figure 11 shows the usual patterns seen in lock compatibility matrices that include intention locks – no conflicts among intention locks and the same pattern repeated in the three remaining quadrants. Each quadrant is mirrored over the diagonal, as is the entire matrix. The top-left quadrant applies to both larger and smaller granularities in the locking hierarchy, e.g., to both tables and rows or to both indexes and keys. If the lock hierarchy includes more than two granularities, all quadrants apply to multiple granularities. Thus, not only traditional S and X locks but also E and C locks apply to entire tables, views, or indexes.

For example, a bulk insert transaction may update 1% of the rows in a table and thus obtain X locks on individual rows in the table, but it may touch 90% of the rows in a view summarizing the table. Thus, the bulk insert operation can obtain an E lock on an entire materialized view, upgrading to a C lock on the entire materialized view during commit processing. It is even possible that a second, concurrent bulk update transaction also obtains an E lock on the entire view, with both bulk insertions updating the same set of summary rows in the materialized and indexed view concurrently. The results is that these two bulk operations conflict neither in the table nor in the view – the former due to locking disjoint sets of rows or keys in the table, the latter due to escrow locking in the materialized and indexed view. These bulk operations do, however, conflict during commit processing, i.e., they serialize their final updates to the summary rows in the materialized and indexed view.

The matrix also indicates that, even while a large transaction holds an E lock on the entire index (and thus implicitly on all keys in the index), another small transaction may obtain an IE lock on the index and then an E lock on one or more specific keys. This is entirely consistent with the interpretation of large granularity locks given earlier, because the small transaction may increment and decrement counts and sums in existing records. However, another transaction cannot insert or delete keys or records while the entire index is locked in E mode, because those actions would require an IX lock on the index and an X lock on the affected keys or records. Insertion and deletion operations will be discussed shortly.

6.3 Commit processing with intention locks

In order to avoid deadlocks during commit processing, E locks must be sorted before being upgraded to C locks. Similarly, IE locks must be upgraded to IC locks. In general, the sort cost can be minimized if the sort key begins with the hash bucket number in the lock manager's hash table, i.e., a single large sort operation is replaced by one small sort operation per hash bucket. If intention locks are involved, however, it is imperative that IC locks be acquired before their dependent C locks. Thus, the sort order and its implementation must ensure that IC locks sort earlier than their dependent C locks, and similarly that IE locks sort earlier than their dependent E locks.

It seems impractical to require that all C locks within a given index fall in the same hash bucket as the IC lock for the entire index – thus, the small per-bucket sort operations cannot guarantee that the IC locks sort earlier than their dependent C locks. A better alternative seems to employ either dedicated hash buckets for different granularities, and to ensure the proper sequence of lock upgrades by processing the hash buckets in the appropriate order.

7 Update and upgrade locks

Not only intention locks but also upgrade locks are similar yet different with escrow locks. In the original design for "U" locks for relational databases, a U lock implies an S lock and puts the transaction into favorite position to convert to an exclusive X lock [GR 93]. Because a transaction can convert an existing S lock to a U lock without waiting for other readers to finish, converting from S to U is very different from an immediate request for an X lock. Nonetheless, converting the U lock to an X lock will like succeed quickly and without deadlock because no second U lock can exists and because no further S locks can be granted once a U lock is in place.

In a system with escrow locks, not only S locks but also E locks may require conversion to an X lock. In fact, as discussed earlier, adding an S lock to an E lock is tantamount to converting the E lock to an X lock. Similarly, adding an E lock to an S lock does not really make sense and the lock should instead be converted to an X lock.

In the traditional interpretation of lock conversions, upgrade locks are intermediate lock modes between two primary lock modes, e.g., S and X. This interpretation leads to two separate upgrade modes, one from S to X and one from E to X. These two lock modes ought to conflict, because only one transaction can actually obtain the desired X lock. Thus, these two intermediate upgrade locks are more similar than different, and it seems that an alternative interpretation of upgrade locks might be more appropriate.

Rather than tying an intermediate upgrade lock to both the original and the desired lock modes, it might be tied only to the desired lock mode. Thus, it is not really an intermediate lock mode anymore; it is a really a preparatory lock mode. It does not give any rights to access data, it merely gives rights to acquire a desired lock. If such preparatory locks are indicated by P subscripted by the desired lock mode, a traditional U lock must then be written and thought of as $S+P_X$.

Moreover, there is use for a preparatory lock mode even if a transaction does not hold any lock at all on the desired resource, i.e., in situations in which an intermediate lock mode between two other locks modes does not really apply. For example, consider a transaction that employs a non-clustered index to determine which rows in a table to update, and assume that records in non-clustered indexes and in clustered indexes are locked separately by their respective index identifier and key value. After the initial search, this transaction holds locks in the non-clustered index but no locks yet in the clustered index. It may be beneficial to place preparatory locks P_X on all rows to be modified before attempting to acquire X locks on any of those rows. Since the preparatory P_X locks do not conflict with concurrent readers and their S locks, setting the P_X locks will likely succeed quickly. The transaction may wait when converting P_X locks to X locks, but while it is waiting for one row, no other transaction can acquire any new locks on any of the remaining rows.

In this specific example, there might not be much difference between a transaction acquiring a traditional U locks, i.e., $S+P_X$ in our notation, or merely a P_X lock. Modifying the example a little makes the difference more clear. Imagine that the updating transaction intends to lock not rows but entire pages or partitions in the clustered index. Other transactions may hold S, IS, or IX locks on those pages or partitions. None of these conflicts with a P_X lock, and P_X locks can be acquired on all needed pages or partitions without waiting. If, on the other hand, the updating transaction were forced to acquire X locks without preparatory P_X locks, it would likely wait for some of these pages or partitions. While the transaction is waiting for the first one, another transaction would be able to "sneak in" and acquire locks on one of the remaining ones.

By separating the upgrade lock mode from the original lock mode, any transaction attempting to acquire an exclusive lock may first acquire a P_X lock. The same considerations about lock compatibility, waiting, and deadlocks apply to P_X locks as to U locks. A transaction may acquire a P_X lock immediately even if other E, S, IS, or IX locks are already in place. The primary effect of the P_X lock is that no further E, S, IS, or IX locks will be granted. Thus, a transaction may be able to place multiple P_X locks without waiting, and then convert those into X locks in a second pass.

In summary, because there are multiple lock modes that can be converted to an exclusive lock, the traditional upgrade lock as an intermediate step between two lock modes might be replaced by a preparatory lock that is specific to the desired lock mode regardless of the existing lock mode.

8 Insert and delete

Our next subject is how new rows are created in materialized views, how new keys are inserted into indexes on materialized views, or how rows and keys are deleted in materialized views and their indexes. As an example, imagine a clustered index on the date columns in Figure 2, i.e., the grouping columns of the view form the search key in a B-tree that contains all columns of the table. The crucial difficulty is that record insertions and deletions require exclusive locks rather than escrow locks. If such locks are retained until end-of-transaction and thus potentially for a long time, lock contention might disrupt the flow of updates in an indexed view. Considering Figure 2, once order processing advances to a new day of shipping, i.e., once the first transaction creates a new summary row in the materialized view, many transactions need to increment counters and sums in that row. Thus, immediately after a new row in a materialized view or a new record in the view index is created, contention for the new row or record can be just as high as for prior rows and records, and holding an exclusive lock until the end of the shipping transaction might not be acceptable.

8.1 Escrow locks and key range locking

Clearly, key value locks and key range locks are important for high concurrency in B-tree indexes. Key range locking has introduced some new lock modes, e.g., “IS-S” [L 93], which have proven practical and efficient in some commercial implementations. These lock modes are combination locks of a special kind, combining two locks at different granularities in the locking hierarchy yet mapped to the same lockable resource, e.g., a single key value, in order to reduce the number of lock manager invocations. Each range between two existing keys (including ghost records) in an index is considered a lockable resource. The range must be locked if any component within is to be locked, specifically the one actual key within (and at the end) of the range. For example, “IS-S” means that the range between two keys is locked in “IS” mode and the key is locked in “S” mode. However, these lock modes must be understood as a performance optimization; they do not affect concurrency control theory or the locking protocol.

As a concrete example, assume a B-tree’s only leaf contains valid records with keys 4 and 9 as well as a ghost record with the key value 6. Assume further that the interval associated with each key ranges from a lower key exclusively to the next higher key inclusively. Thus, the ranges that can be locked are $(-\infty, 4]$, $(4, 6]$, $(6, 9]$, and $(9,\infty)$, which are represented in the lock manager by the actual key values 4, 6, and 9 as well as the artificial key value ∞, respectively. Before locking and reading a key and its record, say key 9, a transaction first requires an intent-to-share (IS) lock on the range $(6,9]$. Before inserting a new key, e.g., 8, a range lock is required for the range that includes 8, because the insertion of a new key could reduce the value of a range lock already held, i.e., from $(6,9]$ to $(8,9]$. Thus, a read-only query might hold an “IS-S” lock precisely to prevent insertion of phantoms [L 93].

It helps to consider each combined lock mode as two separate locks when deciding how these lock modes interact with escrow locks and intent-to-escrow locks. Specifically, update transactions that need to increment or decrement a summary row must obtain an IE lock on the range followed by an E lock on the key or record. Using the same optimization, an “IE-E” lock mode must be provided for key range escrow locking. Note that, as shown in Figure 11 and its discussion, intention locks never conflict. Thus, other locks can obtain other intention locks on the range, although they can only obtain escrow locks on the specific key or record, because escrow locks are compatible only with other escrow locks.

The most interesting case are insertions of new keys into a B-tree index. For those, it is actually not required to lock the boundary of the gap into which a new key is inserted. Instead, only the gap itself is locked, and only briefly, because once the new key is inserted, it is a resource that can be locked in its own right. Insertions are the main topic of the next section.

8.2 Record creation by system transactions

Our solution for this problem relies on two concepts discussed earlier, namely system transactions and the unification of ghost bit and duplicate count into a delta count in each stored record. Recall that a query scanning an indexed view must ignore records with a delta count equal to zero, because this value is equivalent to the traditional ghost bit indicating an invalid record left behind by a prior deletion. Thus, when a deletion of a row in a database table decrements a delta count in an indexed view to zero, there is actually no need to delete this record – the record automatically is semantically equivalent to a deleted record. Leaving it behind may even speed up a future insertion in the table and increment of the count in the materialized view. If that does not happen, erasing the record and reclaiming the space can safely be left to an asynchronous clean-up utility or a future operation that reorganizes the page contents in order to insert a new record. Just as for traditional ghost records, the actual deletion should be performed by a system transaction, which must obtain a suitable lock on the ghost record in order to ensure that no other transaction still retains a lock on it, e.g., because the transaction that turned the record into a ghost has not committed yet or because another transaction has obtained an escrow lock on the record in order to increment its counts and sums.

Inserting a new summary row in the materialized view and thus a new record in each index on the view can work quite similarly, also using a system transaction. The main difference is that a system transaction is invoked to insert a record that, due to a delta count equal to zero, is semantically not there in the sense that no query will include it in its result. This system transaction is triggered when an update attempts to modify a row that does not exist. In other words, inserting a new key in an indexed view with delta count equal to zero (as well as all fields that represent sums that can be incremented and decremented) does not modify the contents of the database, and thus is a good candidate for the short system transaction that can quickly commit and release its locks that were needed for creating the record.

It might be helpful to compare this procedure with traditional key insertion based on key-range locking protocols, which acquire and release locks on the range between keys when inserting a new key into an existing index. Basically, the insert must first lock the range into which a new key is inserted such that no other concurrent transaction can insert precisely the same key. Once the new key is inserted, it represents a database resource that can be locked in its own right. After an exclusive lock is obtained on the new key and registered in the system's lock manager, the danger of another transaction inserting precisely the same key is averted, and the lock on the range between the two pre-existing neighboring keys can be released. Note that this lock is released with a transaction still progressing, even though releasing a lock prior to transaction commit is usually dangerous.

8.3 Locking and logging in system transactions

In the proposed design, it is a system transaction that inserts a new ghost record using key range locking; the user transaction merely modifies the ghost bit or its replacement, the delta count. The user transaction locks only a specific row or index key and only in E mode, whereas the system transaction requires a key-range lock on the entire gap between the two pre-existing neighbors of the new key. Quite obviously, when the system transaction commits, it can release its locks, in particular the one on the key range. However, the user transaction that invoked the system transaction may acquire an E lock on the new record or key while the system transaction still holds an X lock on it, exploiting the lock compatibility between the two transactions. This E lock then permits the user transaction to increment the newly created record or row.

In the traditional locking protocol, a user transaction retains its IX lock on the materialized view or its index until it commits. Thus, if another transaction needs to increment many or most of the rows in the materialized view, it cannot obtain an E lock on the entire view or its index. In the proposed scheme, the IX lock is held only very briefly (for the duration of the system transaction), thus enabling higher concurrency among large and small update transactions. Note that a bulk insert transaction with an E lock on the entire index can still spawn a system transaction that inserts a new key for a new summary row using IX and X locks: while these locks ordinarily conflict, system transactions are lock-compatible with their invoking transaction.

Insertion and commit of ghost records might be objected to because a ghost record remains in the index if the user transaction rolls back. However, it must be considered that the ghost record only occupies space otherwise unused in its B-tree leaf. If the insertion of the ghost record triggers a leaf split, the traditional scheme would also require a leaf split. As discussed earlier, that page split would also be performed using a system transaction that is similarly not rolled back during rollback of the user transaction.

Moreover, it can be argued that the ghost of a summary row remaining in an indexed view can serve good purposes. First, since it represents a group in which there is at least some data and some update activity, the summary row might soon be needed in any case. This includes, of course, restarting the same insertion or update in the base table after the initial user transaction's successful rollback. Retaining the ghost record saves creating a ghost record during this second attempt. Second, since even a ghost record defines a range boundary in key range locking, it permits more fine-grained locks within the indexed view. In fact, one might consider even read-only transactions that insert ghost records into B-tree indexes (using system transactions) solely for the purpose of locking precisely the key range defined by the current query predicate. While maybe not a good idea in general, it can be a good idea when parallel threads create a new index, with each thread responsible for one or more key ranges in the new index.

9 Correctness

The large design presented here in substantial detail combines existing techniques and novel extensions, e.g., multi-granularity escrow locks and non-serial but serializable commit processing. The design's correctness relies on the proven correctness of its components.

Specifically, IE locks and their compatibility with other locks are derived using sound methodology [K 83]. Deadlocks during concurrent commit processing are avoided using the proven method of acquiring locks in a prescribed order. These commit-time-only exclusive locks guarantee that commit processing is serializable; therefore, it is equivalent to serial commit processing. As serial commit processing guarantees a traditional linear history for each summary row and for the entire database, versioning and snapshot isolation are made compatible with escrow locking even if multiple transactions commit concurrently.

Contents-neutral keys and records in B-tree indexes do not affect the semantics of a database, and system transactions may create them just as they erase them in commercial systems today. Delta counts in summary rows can subsume the roles of ghost bits in pseudo-deleted records and of anti-matter bits during online index operations because they can represent more information than those two separate bits can.

Transaction, media, and system failures and their recovery rely on traditional logging in stable storage (not on virtual log records), and our design calls for traditional logging and X locks at the time when changes are actually applied to the database. Thus, changes can be replayed for media and system recovery, and any transaction can be rolled back at any time until the final commit record is logged. In other words, even a failure during commit processing can be rolled back or compensated based on the recovery log.

10 Performance

The performance advantages of locking methods that permit multiple concurrent increment operations have been demonstrated in a prototype based on a commercial product [LNE 03], although that design supports neither concurrent read and update transactions (over the same data) nor concurrent execution and commit processing for small and large update transactions.

Further performance effects of the presented design must be considered for two different cases. First, assume a database system's only current user attempts to perform a large update of tables and of materialized views over those tables. In this case, any and all concurrency control is wasted effort, and performance will not improve in the present design and might even deteriorate due to additional overheads. Second, if high concurrency is required around the clock, and if both data analysis and database maintenance operations are required, our design creates entirely new opportunities not supported by prior work.

For the first case, consider the overhead of saving earlier versions in each update. Clearly, update processing will slow down due to creating and copying old versions of records. Alternatively, some versioning techniques do not retain prior versions during the update but recreate them on demand when needed by a query transaction; or creating and copying old record versions can be suppressed if no other transaction is active at the time of the update. Either one of these alternatives applies to any multi-version concurrency control scheme, and would also apply to and improve the design presented here. Successful commercial database management system products demonstrate that multi-version snapshot isolation can perform very well.

With respect to virtual log records, it is worthwhile to consider that IMS/FastPath performs very well using intention lists. Thus, with careful implementation and tuning effort, the cost difference should be quite moderate between the presented design and a traditional design based entirely on shared and exclusive locks.

For the second case, the presented design enables new scenarios, which must be considered both in terms of concurrency and in terms of availability of tables, views, and their indexes. In a traditional design, materialized views and their indexes are maintained either instantly within the user transaction or periodically in large batches. In the former case, contention for summary rows might render real-time information analysis and monitoring infeasible. In the latter case, large (table or index) locks are required during view maintenance, leading to periods during which the database is not available for queries or update. During those periods, traditional locking schemes would simply defer any incoming query or update transaction, whereas the presented design can process updates instantly with full use of all tables, views, and indexes; and it can be employed to support concurrent update transactions while a separate large batch transaction refreshes or re-creates the materialized view. Given the current interest in stream processing, continuous full availability is very desirable, whereas periods of high load and large locks are very undesirable. If the waiting time is considered part of a query or update transaction's performance, the presented design should compare extremely favorably to traditional designs.

11 Summary and conclusions

In summary, the presented combination of snapshot isolation and escrow locking, the latter generalized to exploit and complement proven database implementation techniques, is a relatively simple yet robust and efficient solution for concurrency control and recovery of materialized summary views and their indexes. Combining these techniques, including some novel techniques to make their combination possible, achieves several goals not addressed in prior research. Foremost among those is contention-free concurrent high read traffic and high update traffic over the same summary data. Second is the ability to combine read and update traffic with large bulk operations, e.g., data import and even online index creation.

The novel techniques presented include multi-granularity (hierarchical) escrow locking and serializable (instead of serial) commit processing. The latter is desirable when deferred increment operations based on virtual log records increase commit processing time for large operations such as bulk data loading. It is achieved using commit-time-only exclusive locks, and deadlocks between concurrent commit activities are proactively avoided by sorting the keys for which commit-time-only locks are required. Such commit-time-only locks generalize the serial commit processing employed in IBM's FastPath product, and can also serve as a template for installing validated updates in optimistic concurrency control methods.

Moreover, the presented design considers not only concurrency control but also logging and recovery. Slight modifications in the format of log records describing record updates and moderate extensions in the interpretation during *undo* processing extend the Aries recovery method for transaction rollback, media recovery, and system recovery. Transaction rollback to save points requires some additional consideration for virtual log records, akin to use of virtual log records in today's commercial systems for large operations such as dropping an index or a table. Checkpoints are not affected by multiple concurrent updates, new transactions can be accepted before restart recovery is complete, and checkpoints during all phases of restart recovery are possible. Virtual (logical *redo*) log records processed during transaction commit ensure a serial history for each record and thus enable versioning support even when multiple update transactions hold concurrent locks on the same data item.

Finally, the design unifies ghost bits, anti-matter bits, duplicate counts, reference counts, group size, and delta counts to enable increment and decrement operations by multiple concurrent uncommitted user transactions. Queries ignore records with counts equal to zero (similar to queries today ignoring records with a ghost bit), and contents-neutral system transactions create and erase records with counts equal to zero such that user update transactions indeed only increment and decrement values but never insert and delete records or keys in a materialized view or any of its indexes. By melting multiple concepts (including the anti-matter used in online index operations) into a single delta count, the presented design is the first to permit online creation of indexed views that require grouping or that might include duplicate rows.

In conclusion, the presented design is a novel combination of implementation techniques that are already proven in database transaction processing, and we hope that it will prove successful in supporting real-time business analysis. Having completed the high-level design, our next steps are to prototype, implement, tune, evaluate, document, release, and support these techniques in a commercial database management system.

Acknowledgements

Mike Zwilling participated in the initial publication of this research and contributed to the presentation of the recent extensions. Phil Bernstein and David Lomet pointed out relevant work we had not been familiar with, Per-Åke Larson focused us on the need for a single sequential history, Wei Xiao helped us explore alternative approaches to combining multi-version snapshot isolation with multiple concurrent update operations, Wey Guy contributed to our design of online index operations, and Steve Lindell provided valuable feedback during initial discussions leading to this design. Phil Bernstein, Roger Barga, Steve Lindell, David Lomet, Tengiz Kharatishvili, and the reviewers offered many helpful suggestions on earlier drafts of this paper.

References

[AJ 98] Gopalan Arun, Ashok Joshi: KODA - The Architecture and Interface of a Data Model Independent Kernel. VLDB Conf. 1998: 671-674.

[B 03] Michael Blasgen: Personal communication. 2003.

[BBG 95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. ACM SIGMOD Conf. 1995: 1-10.

[BCL 86] José A. Blakeley, Neil Coburn, Per-Åke Larson: Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. VLDB Conf. 1986: 457-466.

[BHG 87] Philip A. Bernstein, Vassco Hadzilacos: Nathan Goodman, Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1987.

[BJK 97] William Bridge, Ashok Joshi, M. Keihl, Tirthankar Lahiri, Juan Loaiza, N. MacNaughton: The Oracle Universal Server Buffer Manager. VLDB Conf. 1997: 590-594.

[BLT 86] José A. Blakeley, Per-Åke Larson, Frank Wm. Tompa: Efficiently Updating Materialized Views. ACM SIGMOD Conf. 1986: 61-71.

[CLW 02] Ji-Woong Chang, Young-Koo Lee, Kyu-Young Whang: Global lock escalation in database management systems. Inf. Process. Lett. 82(4): 179-186 (2002).

[CWL 05] Ji-Woong Chang, Kyu-Young Whang, Young-Koo Lee, Jae-Heon Yang, Yong-Chul Oh: A formal approach to lock escalation. Inf. Syst. 30(2): 151-166 (2005).

[DKO 84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood: Implementation Techniques for Main Memory Database Systems. ACM SIGMOD Conf. 1984: 1-8.

[G 03] Dieter Gawlick: Personal communication. 2003.

[G 03a] Goetz Graefe: Sorting and Indexing with Partitioned B-tree. Conf. on Innovative Data Systems Research, Asilomar, CA, January 2003; http://www-db.cs.wisc.edu/cidr.

[GK 85] Dieter Gawlick, David Kinkade: Varieties of Concurrency Control in IMS/VS Fast Path. IEEE Data Eng. Bulletin 8(2): 3-10 (1985).

[GKK 01] Andreas Gärtner, Alfons Kemper, Donald Kossmann, Bernhard Zeller: Efficient Bulk Deletes in Relational Databases. IEEE ICDE 2001: 183-192.

[GL 92] Vibby Gottemukkala, Tobin J. Lehman: Locking and Latching in a Memory-Resident Database System. VLDB Conf. 1992: 533-544.

[GM 99] Ashish Gupta, Inderpal Singh Mumick (eds): Materialized Views: Techniques, Implementations, and Applications. The MIT Press, Cambridge, MA, 1999.

[GR 93] Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, CA, 1993.

[H 84] Theo Härder: Observations on optimistic concurrency control schemes. Inf. Syst. 9(2): 111-120 (1984).

[HD 91] Ugur Halici, Asuman Dogac: An Optimistic Locking Technique for Concurrency Control in Distributed Databases. IEEE Trans. Software Eng. 17(7): 712-724 (1991).

[HR 83] Theo Härder, Andreas Reuter: Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys 15(4): 287-317 (1983).

[J 91] Ashok M. Joshi: Adaptive Locking Strategies in a Multi-node Data Sharing Environment. VLDB Conf. 1991: 181-191.

[JS 89] Theodore Johnson, Dennis Shasha: Utilization of B-trees with Inserts, Deletes and Modifies. PODS Conf. 1989: 235-246.

[K 83] Henry F. Korth: Locking Primitives in a Database System. JACM 30(1): 55-79 (1983).

[KJP 99] Sun Hwan Kim, Mi Suk Jung, Jun Hyun Park, Young Chul Park: A Design and Implementation of Savepoints and Partial Rollbacks Considering Transaction Isolation Levels of SQL2. DASFAA Conf. 1999: 303-312.

[KLM 97] Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, Dallan Quass, Kenneth A. Ross: Concurrency Control Theory for Deferred Materialized Views. ICDT 1997: 306-320.

[KT 79] H. T. Kung, John T. Robinson: On Optimistic Methods for Concurrency Control. VLDB Conf. 1979: 351.

[KT 81] H. T. Kung, John T. Robinson: On Optimistic Methods for Concurrency Control. ACM TODS 6(2): 213-226 (1981).

[L 82] Georg Lausen: Concurrency control in database systems: A step towards the integration of optimistic methods and locking. ACM Conf. 1982: 64-68.

[L 92] David B. Lomet: MLR: A Recovery Method for Multi-level Systems. ACM SIGMOD Conf. 1992: 185-194.

[L 93] David B. Lomet: Key Range Locking Strategies for Improved Concurrency. VLDB Conf. 1993: 655-664.

[LC 89] Tobin J. Lehman, Michael J. Carey: A Concurrency Control Algorithm for Memory-Resident Database Systems. FODO 1989: 490-504.

[LNE 03] Gang Luo, Jeffrey F. Naughton, Curt Ellmann, Michael Watzke: Locking Protocols for Materialized Aggregate Join Views. VLDB Conf. 2003: 596-607.

[LNE 05] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, Michael Watzke: Locking Protocols for Materialized Aggregate Join Views. IEEE Trans. Knowl. Data Eng. 17(6): 796-807 (2005).

[LS 82] Barbara Liskov, Robert Scheifler: Guardians and Actions: Linguistic Support for Robust, Distributed Programs. POPL Conf. 1982: 7-19.

[M 90a] C. Mohan: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. VLDB Conf. 1990: 392-405.

[M 90b] C. Mohan: Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. VLDB Conf. 1990: 406-418.

[M 92] C. Mohan: Less optimism about optimistic concurrency control. RIDE-TQP 1992: 199-204.

[MHL 92] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM TODS 17(1): 94-162 (1992).

[MN 92] C. Mohan, Inderpal Narang: Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. ACM SIGMOD Conf. 1992: 361-370.

[O 86] Patrick E. O'Neil: The Escrow Transactional Method. ACM TODS 11(4): 405-430 (1986).

[PSC 02] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, Hamid Pirahesh: Incremental Maintenance for Non-Distributive Aggregate Functions. VLDB Conf. 2003.

[Q 96] Dallan Quass: Maintenance Expressions for Views with Aggregation. Workshop on Materialized Views: Techniques and Applications (VIEW 1996). Montreal, Canada, June 7, 1996: 110-118.

[R 82] Andreas Reuter: Concurrency on High-traffic Data Elements. ACM PODS Conference 1982: 83-92.

[S 81] Dale Skeen: Nonblocking Commit Protocols. ACM SIGMOD Conf. 1981: 133-142.

[S 85] Dennis Shasha: What Good are Concurrent Search Structure Algorithms for Databases Anyway? IEEE Database Eng. Bull. 8(2): 84-90 (1985).

[SBC 93] George Samaras, Kathryn Britton, Andrew Citron, C. Mohan: Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. IEEE ICDE 1993: 520-529.

[TPC] Transaction Processing Performance Council, http://www.tpc.org/tpch.

[WS 84] Gerhard Weikum, Hans-Jörg Schek: Architectural Issues of Transaction Management in Multi-Layered Systems. VLDB Conf. 1984: 454-465.

[WHB 90] Gerhard Weikum, Christof Hasse, Peter Brössler, Peter Muth: Multi-Level Recovery. ACM PODS Conf. 1990: 109-123.