Low-Cost Compensation-Based Query Processing

Øystein Grøvlen* ClustRa Trondheim, Norway Oystein.Grovlen@clustra.com Svein-Olaf Hvasshovd Norw. Univ. of Sci. and Tech. Trondheim, Norway Svein-Olaf.Hvasshovd@idi.ntnu.no

Øystein Torbjørnsen ClustRa Trondheim, Norway Oystein.Torbjornsen@clustra.com

Abstract

Compensation-based query processing has been proposed in order to avoid lock contention between updating transactions and ad-hoc queries. This paper presents an algorithm based on undo/no-redo compensation. A query will read an inconsistent version of the database, but updates made by concurrent transactions are later undone to make the query result transaction-consistent. By processing the database internal log to obtain information on concurrent updates, queries impose no extra work on updating transactions. A simulation study shows that response times for query execution is significantly improved compared to the earlier compensation-based algorithms. Compared to executing queries with no consistency requirements, the algorithm gives only a small increase in query response times, while the effects on transaction response times are negligible.

1 Introduction

A current trend in database management is increased demand for large and complex queries performed on near real-time data. Such applications often access

Proceedings of the 24th VLDB Conference New York, USA, 1998 data produced by business-critical real-time transactions. Introducing ad-hoc queries in such systems present several unsolved problems. One problem is that long-lived transactions tend to acquire many locks and hold them for a relatively long time. This prevents concurrent updates by real-time transactions [6].

In order to avoid that ad-hoc queries slow down real-time transactions, many organizations keep a separate copy of the database (e.g., a data warehouse) for query processing. Maintenance of the copy will have to be done during low-activity hours (e.g., each night) in order to avoid lock contention with on-line transactions and with queries [14]. Hence, a problem with this approach will be data staleness. In addition, due to increased globalization, many organizations have no off-peak hours.

Another approach is to run the ad-hoc queries with reduced degree of consistency. One example is cursor stability [7], where queries only lock a tuple while it is actually being read. However, many applications require that queries see data that is consistent.

This paper proposes a method for running transaction-consistent queries¹ in an OLTP system without delaying the OLTP transactions. That is, queries should not hold locks for which transactions would have to wait. At the same time, the method should also ensure that all queries get a transaction-consistent view of the database.

1.1 Compensation-Based Query Processing

Compensation-based query processing has been proposed by Srinivasan and Carey in order to reduce lock contention when running queries in an OLTP system [16]. In the first phase of their two-phased approach, queries scan the base relations using cursorstability locking, and a set of temporary relations is created. Concurrently, transactions that update the base relations, append a compensation record for each

^{*}Most of this work was performed while the author was at Norwegian University of Science and Technology. The work was partially supported by a grant from the Norwegian Research Council.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

¹The term *query* will in this paper refer to a long-running read-only transaction.

update to an *update-list*. In the second phase, the compensation records are applied to the temporary relations making the final result reflect updates made by concurrent transactions. The information entered in the update-list can be tailored to each specific query.

The two-phased approach proposed by Srinivasan and Carey has some drawbacks with respect to query efficiency. Intermediate storage, possibly on disk, is required to store the result of the first phase. Also, the two-phased approach prevents efficient pipelining of relational algebra operations. No tuples can be emitted from the query before the entire base relations have been scanned. In addition, the method requires that the query process must either execute under cursor stability or wait for the termination of all transactions that have updated the update-list. This may significantly increase the response times of queries in the presence of long transactions.

Another drawback of the method is that it adds extra work to transactions in order to maintain the update-list. This could possibly increase transaction response times. In addition, the method is not applicable to transaction-consistent execution of read-only transactions that consist of several queries [2].

This paper presents a more efficient method for compensation-based query processing. In Section 2, it is described how the query can use the database internal log to obtain information about concurrent updates. Section 3 presents the new method for compensation-based query processing, and a simulation experiment that evaluates the performance of the method is presented in Section 4. Related work is discussed in Section 5, and Section 6 concludes the paper.

2 Log-Oriented Compensation

Compensation-based query processing reduces lock contention between a query and concurrent transactions without compromising on consistency requirements. A query reads inconsistent versions of its base relations, but returns a transaction-consistent result by compensating for updates made by concurrent transactions.

The compensation-based query-processing method presented in this paper, bases its compensation on the database internal log. In this way, no extra work is imposed on transactions in order to inform queries about concurrent updates. In order to achieve nonblocking execution of queries with respect to transactions, queries will not set any locks on the tuples they access. This will give a query an inconsistent view of the database. However, the log will be used to bring the query result to a transaction-consistent state. This is done by redoing and undoing operations recorded in the log. A query process performs three main operations: It (1) scans the base relation(s) of the query, (2) processes the log and extracts relevant information from log records, and (3) performs the necessary undo/redo-operations before emitting the tuples.

The scan is performed by reading the relation(s) tuple-by-tuple without setting any read locks. Thus, a query will not have to wait for transactions to commit before reading a tuple, and transactions will not have to wait for the query to finish before updating a tuple. The equivalent of a latch will be set on each tuple only while it is being read to protect the read operation from other operations. The scan does not necessarily need to be a sequential file-scan; other access methods (e.g., an index) may be used.

Concurrently with the scan, transactions that update the relations being queried will have entered their updates into the log. The query process will extract relevant information from these log records.

For each query, the set of all concurrent transactions is divided into two disjoint subsets, the BEFORE set and the AFTER set. The query result will reflect all updates by transactions in the BEFORE set and no updates by transaction-consistency, no transaction in the BEFORE set may be dependent on a transaction in the AFTER set. The task of the compensation activity will be to redo operations of members of the BEFORE set and to undo operations performed by members of the AFTER set.²

Approaches to compensation-based query processing may be classified into three categories based on which criteria is used when establishing the BEFORE set and the AFTER set:

- Undo/No-Redo Transactions that are active during the scan are included in the AFTER set. Thus, the query's view of the database will only include updates by transactions that committed before the start of the query.
- No-Undo/Redo Transactions that are active during the scan are included in the BEFORE set. In addition, transactions that are started after the end of the scan, may have to be included in the BEFORE set because other members of the BE-FORE set may depend on them. Using this approach, the query's view will be more up-to-date than for undo/no-redo compensation. However, the query process will have to wait for all transactions that are active during the scan to termi-

 $^{^2}$ Note that the undo and redo-operations are not applied to the database but to the query result. Hence, the compensation will not affect the correctness of concurrent transactions and queries.

nate. The method proposed by Srinivasan and Carey [16] uses this approach.

Undo/Redo The only restriction on the establishment of the BEFORE set and the AFTER set is that that no transaction in the BEFORE set may depend on any transaction in the AFTER set. One way to ensure this is to include all transactions that have committed before a certain point in time in the BEFORE set, while all transactions committed after that point are included in the AFTER set.

In this paper, we present an algorithm for transaction-consistent execution of queries using undo/noredo compensation. It is assumed that the statechanging operations performed by transactions are restricted to update, insert, and delete. Transactions execute using strict two-phase locking (2PL), and log their state-changing operations using a tuple logging policy [9]. That is, all log records contain the primary key and the relation identifier of the corresponding tuple. This is necessary in order to be able to relate the log records to the tuples read by the scan thread. Compensation oriented logging is assumed (i.e., undo operations are logged using compensation log records). Both partial and complete tuple logging may be used, however, where not otherwise stated, partial tuple logging can be assumed.³ It is also assumed that a tuple is contained within a single data block, and that each block includes a state identifier that contains the log sequence number (LSN) of the log record for the most recent update to the block. Each tuple may also have its own state identifier.

3 Undo/No-Redo Compensation

One of the main advantages of undo/no-redo compensation is that all log records needed for performing the compensation on a tuple are already available when the tuple is read. For each tuple read during scan, the log records for this tuple can be fetched and the necessary operations undone before the next tuple is read. By interleaving scanning and compensation in such a manner, intermediate storage of the scanned tuples, possibly on disk, is avoided.

This interleaved execution requires direct access to the log records of a tuple. In order to support such access, the log processing activity will enter the relevant information found in the log records in an *update-table* for later use. This update-table will support direct access (hash-based) on primary key. Relevant information from all log records created by members of the AFTER set (i.e., all transactions that are active after

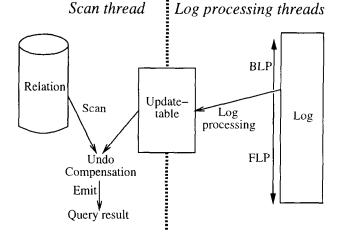


Figure 1: Undo/no-redo compensation.

the start of the query) must be entered in the updatetable. It will in this paper be assumed that a separate update-table is maintained for each query, and that the update-tables are entirely stored in main memory. Efficient methods for storing update-tables on disk are presented in [8].

A query process executing using undo/no-redo compensation consists of one scan thread and two log processing threads (Figure 1). The scan thread will also perform the compensation since this will be interleaved with the scanning. In addition to the log processing thread that processes log records created during the scan, a separate thread will be used to process the log records that have been created by transactions in the AFTER set before the start of the query.

3.1 The Log Processing Threads

The log processing threads process all log records⁴ produced before the end of the scan by transactions in the AFTER set. When a query starts, its *forward log processing thread* (FLP) start processing all new log records. In order to avoid reading the log records from disk, FLP should process log records before they are removed from main memory.

The log records produced by members of the AF-TER set before the start of the query, must also be processed. This is done by a *backward log processing thread* (BLP). At the start of the query, BLP will insert into its AFTER set all transactions that are recorded as active. It will then go backwards in the log processing all log records created by members of this set. The shaded log records in Figure 2 represent the records that are processed by the log processing threads.

When a log processing thread processes a log record,

³That is, only before-images and after-images of attributes that are modified are recorded in a log record.

 $^{^{4}\,\}mathrm{All}$ log records referring to relations that are accessed by the query.

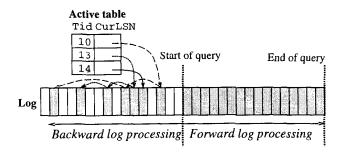


Figure 2: Backward and forward log processing.

it does a hash-based lookup into the update-table using the primary key found in the log record. BLP will enter the before-images of all attributes found in a log record into the update-table, substituting possible previous values. FLP will only enter a before-image into the update-table if no previous value exists for this attribute. BLP and FLP will be executed in parallel. When BLP is finished, the update-table will contain the committed values at the start of the query for all attributes that have so far been changed by members of the AFTER set.

The entries in the update-table could be of three different types: write, insert, or delete. Write entries only contain before-images of attributes that have been changed by members of the AFTER set. During compensation, these before-images will be substituted for the attributes read by the scan thread. For delete entries, the before-images of all relevant⁵ attributes of the deleted tuple are included in the entry. Delete entries represent tuples that must be included in the query result even if they are not read by the scan thread. Insert entries contain no before-images and represent tuples that should be ignored by the scan thread. Details on rules for determining the type of an entry can be found in [8].

3.2 The Scan Thread

When BLP is finished, the scan thread can start scanning the base relation(s) of the query. How the scan thread is executed depends on the operations of the query. Figure 3 shows the basic algorithm for producing a snapshot of a relation. The scan thread scans the entire relation, and checks for each tuple the corresponding state identifier.⁶ If the state identifier is smaller than the LSN of the oldest log record processed by BLP, the tuple can be emitted as it is since it could not possibly have been changed by members of the AFTER set.

SCAN	$-\mathrm{Thread}(R)$		
1 for each tuple t in base relation R do			
2	Read t		
3	if state-identifier[t] < LSN of oldest log rec.		
	processed by BLP then		
4	$\operatorname{emit}t$		
5	else		
6	Wait for necessary log processing		
7	$k \leftarrow \text{Update-Table-Entry}(primkey[t])$		
8	if $k = NIL$ then		
9	$\mathbf{emit} t$		
10	else		
11	if $type[k] \neq \text{INSERT then}$		
12	$\hat{t} \leftarrow \text{COMPENSATE}(t,k)$		
13	$\operatorname{emit} \hat{t}$		

Figure 3: Basic algorithm for the scan thread.

Before checking the update-table, the scan thread must make sure that the necessary log records have been processed by FLP. This synchronization between the scan thread and FLP is further described in Section 3.3. When the scan thread has made sure that the necessary information for the current tuple has been entered in the update-table, it does a hash-based lookup in the update-table on the primary key of the current tuple. If an insert entry is found, the tuple is not emitted. If an update or delete entry is found, compensation is performed. That is, the values of each attribute found in the update-table are substituted for the corresponding attribute values of the tuple. If no entry is found in the update-table, the tuple is emitted in the form it was read.

When the entire relation has been read by the scan thread using the algorithm of Figure 3, the result will include all tuples of a transaction-consistent snapshot, except possibly some tuples that have been deleted during the scan. These tuples could be emitted at the end of the scan by searching the update-table for delete entries that have not been visited by the scan thread. However, query evaluation algorithms often exploit that the scan sequence is sorted on a combination of attributes. In Section 3.4 it will be discussed how the scan order can be preserved in the output of the query.

In general, queries could be executed by first obtaining a transaction-consistent snapshot as described above, and then run the queries on this copy. However, queries could be more efficiently executed by integrating them with the scan thread and the log processing threads. For example, selection predicates can be evaluated by the scan thread after the compensation has been performed. Aggregation can be performed by letting the scan thread and the log processing threads directly update the aggregated result. An in depth

 $^{^{5}}$ An attribute is relevant if it is either part of the query's projection or is needed for processing the query (e.g., used in the selection predicate).

 $^{^{6}\}mathrm{A}$ state identifier could be maintained for each tuple or just for each block.

discussion on this topic can be found in [8].

3.3 Synchronizing the Scan Thread and the Forward Log Processing Thread

Before the scan thread performs compensation, all log records of operations that are reflected in the current tuple must have been processed by FLP. If not, the scan thread will wait for the FLP thread to process more log records.⁷

One way to ensure that sufficient log has been processed, is to process all new log records before compensation is performed. However, a less eager strategy can be used by taking advantage of the state identifiers included in each tuple/block. If the state identifier of the current tuple/block is smaller than the LSN of the last log record processed by FLP, all operations needed for doing the compensation are already entered into the update-table. Thus, no more log records need to be processed before the compensation is performed. Otherwise, the scan thread is suspended until the necessary log records have been processed. In other words, line 6 of the algorithm in Figure 3 should be changed to:

while state-identifier[t] > LSN of last log rec. processed by FLP do Wait

Using the synchronization described above instead of processing all new log records, the amount of log processing could normally be reduced. However, FLP should still make sure to process all log records before they are removed from main memory.

3.4 Preserving Scan Order in the Query Output

Query evaluation algorithms often exploit that the scan sequence is sorted on a combination of attributes, the *sort key*. In order to be able to preserve the scan order, the log processing threads will also insert the sort key of deleted tuples into a priority queue together with a reference to the corresponding entry in the update-table. For each tuple it reads, the scan thread will check the priority queue for tuples that should be emitted before the current tuple. If it is not necessary to preserve the scan order, tuples in the queue could be emitted at any time during the scan.

When synchronizing the scan thread and FLP, the state identifier of the current block can be used to check whether all possible deletions of tuples with a smaller sort key than the current tuple have been entered in the priority queue.⁸

3.5 Space Optimization of the Update-Table

In order to minimize memory usage, the update-table should be kept as small as possible. The following optimizations will reduce the size of the update-table:

- Only before-images of relevant attributes are entered into the update-table.
- If the scan order is predefined, FLP should, if possible, check whether the tuple referred in the current log record has already been read by the scan thread. If so, it is not necessary to enter information from this log record into the update-table. In order to be able to decide whether a tuple lies behind or ahead of the scan thread, the log record must contain the attributes determining scan order.
- When the scan thread has finished processing a tuple, the entry for this tuple in the update-table can be removed. However, if it is not guaranteed that FLP will be able to decide whether the tuple of a later log record lies behind or ahead of the scan thread, a new entry may be made for this tuple later. In order to avoid this, only the attribute values are deleted, while the primary key is kept and the entry is marked *processed*.

3.6 Complete Tuple Logging

So far, it has been assumed that partial tuple logging is used. If, on the other hand, complete tuple logging is used, the before-image of the entire tuple is stored in a log record. This way, the log processing threads will have access to all attributes of the tuple, and this can be exploited to optimize query execution. Moreover, FLP will always be able to decide whether the tuple of a log record lies behind or ahead of the scan thread when base relations are scanned in a predefined order.

Complete tuple logging also simplifies the maintenance of the update-table. All attributes of a tuple will be available in the first log record processed by one of the log processing threads. Hence, only BLP needs to process more log records for this tuple.

4 Performance Study

This section presents an evaluation of the performance of compensation-based query processing. The evaluation is based on simulation experiments. The sim-

 $^{^{7}}$ Note that if before-images of all relevant attributes are available in the update-table, the compensation can be done without synchronizing with the FLP. Processing more log records will in this case never change the entry in the update-table.

⁸If a relation is scanned using a secondary index, the state identifiers of the index blocks must be used. In addition, an update of the index key must be treated as a delete/insert pair by the log processing threads.

ulation model was implemented in the C++-based CSIM18 simulation language [11]. The performance of the algorithm presented in Section 3 was compared to the performance of other query algorithms. The simulation model is briefly described before presenting experiments and results. A more detailed description of the model and the experiments can be found in [8].

4.1 Simulation Model

The simulation model presented below is divided into two main parts, the *system model* and the *application model*. The first part models the behavior of the DBMS and its resources, while the latter part models the database and transaction and query workloads. A separate section presents the part of the system model that is related to query execution.

4.1.1 System Model

The system model encapsulates the logical and physical resources of a DBMS and its underlying operating system and hardware. The system model consists of a single CPU, a single disk manager which administers several disks, a buffer manager, a checkpoint manager, a lock manager, and a log manager. In addition, the system has several transaction managers and query managers, each executing a single transaction or query, respectively, at a time.

The CPU module models the behavior of the CPU scheduler. The scheduler is priority-based and nonpreemptive. It is assumed that the DBMS is run as a collection of light-weight threads in a single process. The CPU scheduler will assign the CPU to the requesting thread with the highest priority. In case of ties in priorities, the scheduler uses a first-come, first-served (FCFS) policy. Threads executing queries are given lower priority than threads executing transactions.

The disk manager will receive requests from the buffer manager to transfer a certain number of blocks starting with a given block ID between the disk and the database buffer. Asynchronous I/O towards raw disk devices is assumed. The model for the service time of a disk request is based on [15], and the settings of the disk parameters are based on the data sheet for the Seagate Cheetah 4LP disks and on measurements presented in [18].

The buffer manager handles the database buffer using an LRU replacement policy. A data page can be accessed either through the buffer hash table, or the page holding a particular tuple can be found using a B-tree index. For simplicity, it is assumed that the B-tree has a fixed size, and that index pages are never updated. To avoid that pages from large sequential scans fill the entire buffer, such pages are inserted at the front of the LRU list. All flushing of dirty pages to

Table 1: Parameters for the system model.

Parameter	Description
CPURate	Instruction rate of CPU
BufSize	Number of pages in the DB buffer
DiskBlockSize	Disk block size
DiskMaxLatency	Maximum rotational delay
DiskTransfer	Disk transfer rate
MaxSeqIO	Max. number of pages in a se-
	quential read/write
LogBufSize	Number of pages in log buffer
TransMaxTime	Max. lifetime for transactions
SchedCPU	Overhead for thread switching
BufCPU	Cost for a lookup in the DB buffer
BuffMissCPU	Extra cost for an unsuccessful
	lookup in the DB buffer
DiskX ferCPU	Cost for an asynch. disk request
DiskPollCPU	Cost to poll for a disk request
LockReqCPU	Cost to request a lock
LockRelCPU	Cost to release a lock
LogCreRecCPU	Cost to create a log record
TransInitCPU	Cost to initiate a transaction
TransCommitCPU	Cost to commit a transaction
TransAbortCPU	Cost to abort a transaction
ReadCPU	Cost for a read operation
UpdateCPU	Cost for a write operation
DeleteCPU	Cost for a delete operation
InsertCPU	Cost for an insert operation

disk is normally done by the checkpoint manager which is activated when the number of dirty pages gets high.

The lock manager implements key range locking in addition to ordinary tuple-level locking. Transactions are executed using strict 2PL. No deadlock detection is implemented. Deadlocks are resolved by the transaction managers which abort transactions that have not terminated within a given time limit.

The log manager maintains a buffer of the most recent log pages. Complete tuple redo/undo logging is used, and it is assumed that a log record uses twice as much space as the corresponding tuple. Log records are flushed to disk either upon request by the checkpoint manager or when transactions are committed. In order to reduce the work associated with log flushing, a group commit policy similar to the method used by the Oracle DBMS [13] is applied.

The main parameters of the system model are presented in Table 1.

4.1.2 Query Execution Model

Each query manager runs a single query at a time. The query manager can use four different algorithms to execute a query:

UNDO compensation. The algorithm presented in Section 3. The update-table is assumed to be

stored as a hash table in a temporary main memory buffer separate from the database buffer. Thus, the size of the update-table will not affect the hit ratio of the database buffer. Since the simulation model assumes complete tuple logging, information is never entered into the update-table more than once for each tuple during forward log processing. Where not otherwise stated, the space optimization techniques presented in Section 3.5 are not applied. A priority queue is used to preserve the scan order for deleted tuples. The priority queue is implemented as a partially ordered binary tree.

The implementation of UNDO compensation requires that log records are processed before they are removed from the log buffer. If a large part of the log buffer has not been processed, forward log processing is given transaction priority until the entire log buffer has been processed. A query is aborted if an unprocessed log record is removed from the log buffer. Only backward log processing will may log records from disk.

- **REDO Compensation.** A slight modification of the algorithm by Srinivasan and Carey. The updatelist is implemented as a hash table instead of as a sequential list. This way, sorting the updatelist between the scan phase and the compensation phase is not necessary. The hash-table will contain maximum one entry per tuple. In order to avoid reading uncommitted data, a read lock is acquired before a tuple is read. The lock is released before locking the next tuple. Information on updates made by a transaction is entered into the update-list at commit time. The only extra CPU cost modeled for transactions is the cost associated with inserting an entry into the update-list. Neither the cost to access information on concurrent queries nor the cost to keep a list of all updates of transactions until commit time are modeled.
- **Two-phased locking.** All tuple accesses by queries are covered by primary-key range locks. Before accessing a new page, a query extends its key range lock to cover all tuples of the page. When the query is finished, the range lock is released.
- **GO-processing.** No locks are waited for or set by the query, and all tuples are emitted in the form they are read. In other words, the query result will not be transaction-consistent. The performance of GO-processing will be used as a reference to how much query performance must be sacrificed in order to achieve transaction-consistency.

Table 2:	Query	execution	parameters.
----------	-------	-----------	-------------

Parameter	Description
TupProcCPU	Cost to process a single tuple
TupEmitCPU	Cost to emit a single tuple
LogProcCPU	Cost to process a log record
Up Tab In CP U	Cost for an insertion into the update-
	table
UpTabOutCPU	Cost for a lookup in the update-table

The execution of a query involves at least three separate threads. In addition to the scan thread, a query will have a read and a write thread. The read thread reads one batch of pages at a time from disk. When the scan thread starts processing such a batch, the read thread will request a new batch from the buffer manager. The query result is stored in a temporary buffer, and the write thread writes batches of pages from the temporary buffer to disk. If UNDO compensation is used, a query process will also contain a forward log processing thread. Backward log processing is performed by the scan thread. In order to minimize the effects on transaction processing, a scan thread or a log processing thread will only process a single tuple or log record, respectively, each time it is scheduled for the CPU. The CPU cost parameters used to model query execution are presented in Table 2.

4.1.3 Application Model

The application model is based on the TPC-B specification [17], modeling a bank with branches, tellers, and accounts and reflecting the history of recent transactions. Each transaction contains three updates, one for each of the Account, Teller, and Branch relations, and one insert in the History relation. In the experiments, the database size is fixed (1 million accounts) and not scaled to the transaction load as required by the TPC-B specification. In order to be able to support a higher number of concurrent transactions without scaling the database, the number accounts per branch was set to 2000. For each transaction, the teller and account is chosen by a uniform distribution. Each relation is stored as a clustered B+-tree indexed on primary key. A sequence number is used as primary key in the History relation. Thus, the tuples will be inserted in physical order.

Tuples in the History relation is 50 bytes, all other tuples are 100 bytes. *NumDisks* disks is used to store the Account relation. One disk is used for each of the other relations, and three additional disks are used for indexes, temporary tables, and the log.

The workload of the system is modeled as a fixed set of terminals, each either requesting the execution of transactions or queries. Each terminal only submits one job at a time, and a new job is submitted at once

Table 3: Application model parameters

Parameter	Description
NumDisks	Number of disks for the Account relation
MPL	Number of transaction terminals
QMPL	Number of query terminals
Sel	Selectivity factor for queries
KeysPP	Number of keys per index page

its previous job has terminated. Queries are executed by scanning a given fraction, a specified primary key range, of the Account relation in primary key order. The selectivity factor of the query determines the size of the key range.

The parameters of the application model are listed in Table 3.

4.2 Experiments and Results

Below, the results of the simulation experiments are presented. The main performance metrics of the experiments are query response time and transaction throughput.⁹ In addition, the storage needed for the update-table and the time used for backward log processing are studied.

For all measurements, statistical validity was ensured by verifying that size of the 90% confidence intervals were, if not otherwise stated, within 1% of its mean. The confidence intervals were computed using the replication method for steady-state simulations [1].

The settings of the simulation model parameters for the experiments are shown in Table 4. The values for the CPU cost parameters are based on measurements done on the ClustRa DBMS [10]. The experiments were run over a range of multiprogramming levels (MPL) and query selectivity factors (Sel).

4.2.1 Query Response Time

Figure 4 compares the query response times for the different query algorithms used in the experiments. Executing the query using 2PL gives the lowest response times. However, as will be shown below, using 2PL also results in a significant lower transaction throughput. In fact, the reason for the good query performance is that the system utilization by transactions is reduced due to lock contention.

The experiments show that UNDO compensation only gives a slight increase in response times compared to GO-processing. This increase represent the work associated with processing the log, maintaining the update-table, and compensating for concurrent updates. REDO compensation give more than twice as

Table 4: Parameters settings.

Parameter	Setting
CPURate	300 MIPS
BufSize	4096 pages
DiskBlockSize	4 kBytes
DiskMaxLatency	6.0 msec
DiskTransfer	13.7 MBytes/sec
MaxSeqIO	32 pages
LogBufSize	40 pages
TransMaxTime	5 secs
SchedCPU	150 instr.
BufCPU	1500 instr.
BuffMissCPU	6000 instr.
DiskXferCPU	50000 instr.
DiskPollCPU	1500 instr.
LockReqCPU	2000 instr.
LockRelCPU	300 instr.
LogCreRecCPU	7000 instr.
TransInitCP U	35000 instr.
TransCommitCPU	40000 instr.
TransAbortCPU	40000 instr.
ReadCPU	8000 instr.
UpdateCPU	15000 instr.
DeleteCPU	6000 instr.
InsertCPU	12000 instr.
TupProcCPU	8000 instr
TupEmitCPU	7500 instr.
LogProcCPU	1400 instr.
UpTabInCPU	7500 instr.
UpTabOutCPU	1000 instr.
NumDisks	8
MPL	1, 3, 6, 10, 15, 21, 28
QMPL	1
Sel	2%, 5%, 10%, 25%, 50%, 100%
KeysPP	150

high query response times as UNDO compensation. This is as expected since the two-phased approach requires intermediate storage of data on disk. For the compensation-based algorithms, the ratios of their average query response times to that of GO-processing, were independent of the selectivity of the query.

Figure 5 compares the performance of UNDO compensation and GO-processing. At low system utilization by transactions, the compensation overhead is small. At higher utilization, the overhead increases up to about 30% when MPL is 21 (about 80% CPU utilization by transactions). When MPL was increased to 28 (about 87% CPU utilization), all queries were aborted because the log processing thread was not given enough CPU time to process all log records before they were removed from the log buffer.

Note that the starvation of the log processing thread does not imply that UNDO compensation is only practical for a small number of concurrent transactions. The limiting factor for log processing is CPU time.

⁹Since for each simulation run transactions are generated by a constant number of terminals, transaction throughput is really a measure of average transaction response time. The only difference is that throughput is affected by the abort rate.

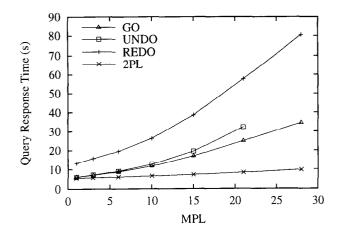


Figure 4: Query response times for various MPL. (Sel = 10%)

There is no limitation on the number of concurrent transactions as long as the CPU has spare capacity for log processing.

4.2.2 Transaction Throughput

The reduction in transaction throughput caused by concurrent execution of queries is small except when two-phase locking is used (Figure 6). Transaction throughput is slightly higher for UNDO compensation than for REDO compensation and GO-processing. This is mainly a consequence of the non-preemptive CPU scheduling. Since the log processing thread holds the CPU for a shorter time period than the scan thread, the average time transactions have to wait for the CPU is reduced by introducing the log processing thread.

The extra work imposed on transactions by REDO compensation did not significantly increase transaction response times. The main reason is that in the

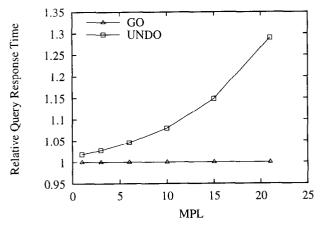


Figure 5: Relative performance of UNDO compensation compared to GO-processing. (Sel = 10%)

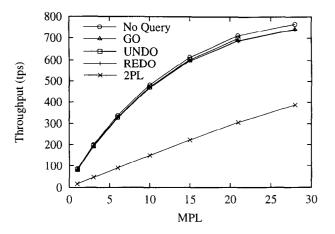


Figure 6: Transaction throughput when running queries. (Sel = 10%)

experiments response times were dominated by disk access time. In addition, during the compensation phase queries only access disks holding temporary tables. Thus, transactions do only have to wait for disk requests made by queries during the scan phase. Also note that the transaction throughput of REDO processing will be actually somewhat lower since not all the cost of maintaining the update-list has been modeled.

When running queries using UNDO compensation, the distribution of transaction response times is not significantly changed. For MPL = 10 and Sel =10%, the response times of transactions in the 90th percentile increased with only 3%. In other words, transaction-consistent query execution can be achieved without any significant effect on concurrent transaction processing.

4.2.3 Size of Update-Table

In order to apply UNDO compensation, the main memory requirements for the update-table should not be too large. The storage needed is dependent on both MPL and the length of the query. Figure 7 shows the storage requirement for the update-table both when using and not using the space optimization techniques presented in Section 3.5. The figure shows that for a given MPL, the maximum size of the update-table is proportional to the selectivity (length) of the query. As long as MPL is low, only a small fraction of the tuples have entries in the update-table. When MPL increases, the storage needed for the update-table increases both due to increased update rates and longer query execution times.

Figure 7 also shows that the space requirements for the update-table could be significantly reduced if space optimization is applied. Assuming uniform tuple access and constant scan rate, avoiding insertions of en-

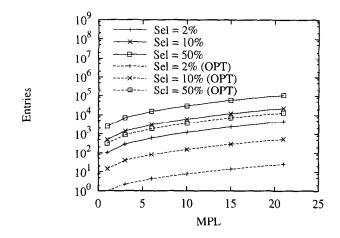


Figure 7: Non-optimized and optimized (OPT) space requirements for the update-table. (Logarithmic scale on the vertical axis.)

tries for already processed tuples and deleting entries of processed tuples, reduce the space requirements to one fourth of the non-optimized case.¹⁰ In addition, entries need only be made for tuples within the selectivity range of the query. Thus, optimization may reduce the space requirements to Sel/4 of the nonoptimized case, where Sel is the selectivity factor of the query. However, in order to apply the optimization, the log processing thread need to be able to determine whether a tuple lies behind or ahead of the scan. Thus, for partial tuple logging, the non-optimized and the optimized results in Figure 7 can be viewed as the upper and lower limits, respectively, for the space requirements of the update-table.

The optimizations also lead to increased query performance due to reduced work for the log processing thread. For long running queries at high MPL, the query response times were reduced by 10%. Transaction throughput was not significantly affected.

4.2.4 Transaction-Mix Experiment

While TPC-B transactions reflect a simple updateintensive OLTP application, an OLTP system is often characterized by multiple transaction types of varying complexities. Therefore, another experiment was performed where transactions of various lengths and operations were used. The number of operations of a transaction were Poisson distributed with an average of five operations, and the mix of operations used was 40% reads, 30% updates, and 15% each of inserts and deletes. The same database was used for this experiment, and for each operation the relation was ran-

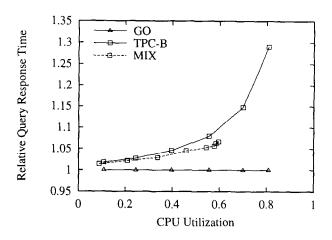


Figure 8: Comparing the overhead of UNDO compensation relative to GO-processing in the two experiments. (Sel = 10%)

domly determined. Within each relation an uniform access pattern was used.

Figure 8 compares the relative query performance with respect to GO-processing of this experiment to that of the TPC-B experiment. The overhead of UNDO compensation for a given CPU utilization by transactions were lower than in the TPC-B experiment. This is because TPC-B transactions have a higher frequency of modifying operations than the transactions used in the second experiment. In other words, more log processing and compensation are needed for running queries concurrently with TPC-B transactions.

The reduction in transaction throughput when running queries was similar to the TPC-B experiment. The space requirements for the update-table were smaller due to the lower update-rate.

4.2.5 Time Used for Backward Log Processing

For the TPC-B experiment, the time used for backward log processing (BLP) seldom exceeded 1 ms. The reason for this is the profile of TPC-B transactions. Most transactions will only have to fetch one page from disk, the page for the Account tuple. Since the Account relation is accessed by the first operation, most of the active TPC-B transactions will not have performed any operations at the start of the query, and these transactions need not be handled by BLP.

The execution time of BLP for the transaction-mix experiment is shown in Figure 9. As expected, the time is dependent on the multiprogramming level (MPL) since this directly determines the size of the AFTER set. The reason for the more than linear growth in execution time with increasing MPL, is that an increasing update rate will increase the probability of having to access log pages that is no longer in main memory. By

 $^{^{10}{\}rm The}$ update-table will reach its maximum half way through the scan. At that time, half of the entries have not yet been made, and the update-table will only contain entries for unread tuples.

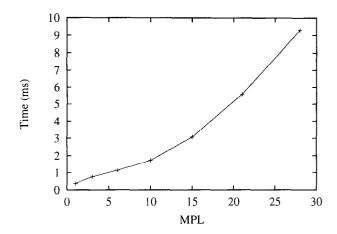


Figure 9: Time to perform backward log processing. (Sel = 2%)

using a larger buffer for log pages, less pages would have to be fetched from disk by BLP. Accessing log records on disk, may potentially slow down transactions. However, no significant effects on transaction throughput was observed in this experiment.

4.3 Discussion

The simulation experiments show that it is possible to efficiently perform compensation-based query processing without significantly affecting the performance of transactions. As expected, UNDO compensation outperforms REDO compensation with respect to query response times and 2PL with respect to transaction throughput.

The extra work required to achieve transactionconsistency only slightly increases the response times for queries in a system with low to medium CPU utilization by transactions. The maximum overhead compared to GO-processing was 30% when running a query concurrently with TPC-B transactions. For less update-intensive transactions, the overhead to achieve transaction-consistency will be smaller. At very high CPU utilization by transactions, the log processing thread was not able to keep up with the production of new log records. The starvation of the log processing thread can be avoided by letting it process all new log records each time it is scheduled. However, this may potentially increase transction response times.

The increase in transaction response time by introducing queries is negligible as long as threads executing the query are given lower priority than other threads. The experiments also showed that the extra work required by transactions when doing REDO compensation is not significant as long as transactions need to read pages from disk. For transactions which do not need to access disk, this extra work may possibly affect transaction response times, especially for queries that require more complex processing of concurrent updates.

The experiments show that the critical part for when using UNDO compensation is the size of the update-table. For short queries or low to medium update rates, the size will not represent a problem. For long queries and very high update rates, the space requirements for the update-table may exceed the available main memory even when using the space optimization techniques.

In all experiments, the time needed to perform backward log processing (BLP) was negligible compared to the entire execution time of the query. The BLP time is generally dependent on the length and number of concurrent transactions. For most queries the BLP time will be a small fraction of the execution time even for much longer transactions than those used in these simulation experiments.

5 Related Work

Transient versioning is an alternative approach to avoid lock contention while still achieving transactionconsistency for queries in an OLTP system [5, 3, 12, 19]. In transient versioning algorithms, transactions create a new physical version of a data item when performing an update. Queries may access an older version in order to get a transaction-consistent view.

In order to maintain data clustering, transient versioning algorithms should perform in-place updates. Hence, the previous version of a data item will be copied before the data item is updated. Hence, transient versioning will increase the response times of transactions. In the original transient versioning algorithms, prior versions are stored in a separate version pool [5]. This will potentially reduce query performance since the data clustering is disrupted with respect to queries. In addition, several disk accesses may be needed in order to locate the correct version of a data item. To reduce this problem, on-page caching of prior version has been proposed [3]. However, on-page caching will decrease the buffer hit ratio for transactions since the database will occupy more pages.

Unlike transient versioning, compensation-based query processing will only materialize versions that are actually needed by queries in the system. This means that there will be no overhead when there is no active queries. On the other hand, transient versioning will never maintain more than one instance of a prior version while update-tables for concurrent queries may each contain a copy of the same version of a tuple. However, this can also be achieved with compensationbased query processing by letting a single forward log processing thread and a single update-table serve multiple concurrent queries. The update-table may then contain several versions of the same tuple, and each version will be tagged with the query IDs of the queries that should access this particular version.

Compensation-based query processing also has the advantage of being able to tailor the content of the update-table to specific queries [8]. In addition, if the update-table is (partly) stored on disk, its organization could be optimized on I/O cost given the access pattern of the query. This is not possible for the general version pool used in transient versioning.

Transient versioning has been implemented in the Oracle DBMS which uses its rollback segments as the version pool [4].

6 Conclusions

This paper has presented a novel method for compensation-based query processing that overcomes most of the disadvantages of the method presented by Srinivasan and Carey [16]. By using the log to communicate updates, negligible extra load is put on updating transactions. Hence, all work related to the compensationbased execution of a query could be performed by a separate query process. A two-phased approach is avoided by using undo/no-redo compensation. Thus, there is no need for temporary storage of base relations. In addition, queries can emit tuples at once they are read, making it possible to exploit efficient pipelining of relational operations. Several queries may also see the same transaction-consistent state by using a common update-table.

The simulation experiments show that compensation-based query processing can be efficiently performed without significantly affecting response times of concurrent transactions. This is achieved by giving higher priority to transactions than to query processes. A minimum of spare capacity must be available for query execution in order to keep the update-table at a moderate size. The main-memory requirements for the update-table could be reduced by storing parts of the update-table on disk. Efficient methods for a disk-based update-table are presented in [8].

The method has already been extended to distributed execution of queries [8]. The plan is to implement the distributed version in the ClustRa DBMS.

Acknowledgements

The authors would like to thank Rune Humborstad and Maitrayi Sabaratnam for constructive comments on earlier drafts of this paper.

References

 J. Banks, J. S. Carson, II, and B. L. Nelson. Discrete-Event System Simulation. Prentice Hall, second edition, 1996.

- [2] P. M. Bober. Towards Practical Multiversion Locking Techniques for On-Line Query Processing. PhD thesis, University of Wisconsin - Madison, 1993.
- [3] P. M. Bober and M. J. Carey. On mixing queries and transactions via multiversion locking. In Proc. Int. Conf. Data. Eng., pp. 535-545, Feb. 1992.
- [4] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton. The Oracle Universal Server buffer manager. In Proc. Int. Conf. VLDB, pp. 590– 594, Aug. 1997.
- [5] A. Chan, S. Fox, W. Lin, A. Nori, and D. Ries. The implementation of an integrated concurrency control and recovery scheme. In *Proc. ACM SIGMOD*, pp. 184-191, 1982.
- [6] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. Commun. ACM, 35(6):85-98, June 1992.
- [7] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.
- [8] Ø. Grøvlen. Concurrent Query and Transaction Processing — A Compensation-Based Approach. Dr.ing. thesis, Norwegian University of Science and Technology, Trondheim, Norway, Aug. 1998.
- [9] S.-O. Hvasshovd. Recovery in Parallel Database Systems. Vieweg, 1996.
- [10] S.-O. Hvasshovd, Ø. Torbjørnsen, S. E. Bratsberg, and P. Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In Proc. Int. Conf. VLDB, pp. 469-477, Sept. 1995.
- [11] Mesquite Software, Austin, TX. User's Guide CSIM18 Simulation Engine (C++ Version).
- [12] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proc. ACM SIGMOD*, pp. 124–133, June 1992.
- [13] Oracle Corporation, Redwood City, CA. Oracle7 Server Concepts, Release 7.2, 1995.
- [14] D. Quass and J. Widom. On-line warehouse view maintenace. In Proc. ACM SIGMOD, pp. 393–404, May 1997.
- [15] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, pp. 17–28, Mar. 1994.
- [16] V. Srinivasan and M. J. Carey. Compensation-based on-line query processing. In Proc. ACM SIGMOD, pp. 331-340, June 1992.
- [17] Transaction Processing Performance Council, San Jose, CA. TPC Benchmark B Standard Specification, Revision 2.0, June 1994.
- [18] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *Proceedings of the ACM SIGMETRICS Conference*, pp. 146–156, May 1995.
- [19] K.-L. Wu, P. S. Yu, and M.-S. Chen. Dynamic finite versioning: An effective versioning approach to concurrent transaction and query processing. In Proc. Int. Conf. Data. Eng., pp. 577-586, Apr. 1993.