Transaction Reordering

Gang Luo[a] Jeffrey F. Naughton[b] Curt J. Ellmann[b] Michael W. Watzke[c]
[a]IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
[b]University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706, USA
[c]Teradata, 5752 Tokay Blvd Suite 400, Madison, WI 53719, USA
luog@us.ibm.com naughton@cs.wisc.edu ellmann@wisc.edu michael.watzke@teradata.com

Abstract

Traditional workload management methods mainly focus on the current system status while information about the interaction between queued and running transactions is largely ignored. This paper proposes using transaction reordering, a workload management method that considers both the current system status and information about the interaction between queued and running transactions, to improve the transaction throughput in an RDBMS. Our main idea is to reorder the transaction sequence submitted to the RDBMS to minimize resource contention and to maximize resource sharing. The advantages of the transaction reordering method are demonstrated through experiments with three commercial RDBMSs.

Keywords — Data warehousing, workload management, transaction reordering, continuous data loading, synchronized scans

Corresponding author: Gang Luo

Phone: 914-784-6932

Fax: 914-784-6040

Contact email: gangluo@cs.wisc.edu

1. Introduction

Traditional workload management methods mainly focus on the current system status [1, 2]. For example, in a typical RDBMS, the load controller only allows a certain number of complex queries to run concurrently. Also, if the system is in the danger of thrashing (i.e., admitting more transactions for execution will lead to excessive overhead and severe performance degradation [1]), the load controller may choose not to run any new transactions.

To support modern applications, users are continually requiring higher performance from RDBMSs. To meet this requirement, it is natural to ask whether or not we can use information about the interaction between queued and running transactions to improve the existing workload management methods. The answer to this question is “yes.” In fact, in many instances, it is possible to improve the throughput of an RDBMS through the utilization of such information. More specifically, we can improve the throughput of an RDBMS that is processing a sequence of transactions by reordering these transactions before submitting them for execution. This is due to opportunities for either resource sharing among multiple transactions (e.g., sharing data in the buffer pool, or perhaps even sharing intermediate computations common to several transactions) or lowering resource contention (e.g., avoiding lock conflicts). Information about the interaction between queued and running transactions is essential in capturing such opportunities.

There are two main reasons why transaction reordering might be effective. The first is system independent – for example, it might be that a reordering of a transaction sequence truly eliminates some intrinsic lock conflicts between adjacent transactions and/or makes resource sharing possible. The second is system dependent – for example, a system may have a particular implementation of buffer management or concurrency control that renders one order of transactions superior to another. Even reordering to exploit system dependent opportunities is useful. Commercial RDBMSs are large, complex pieces of code, and changes in functionality can require a very long design-implement-test-release cycle. In many cases it may be far simpler to do some reordering of transactions outside of the RDBMS before submitting them to the RDBMS for execution than it would be to change, say, the concurrency control subsystem of the RDBMS. This is especially true for database application developers who are unable to change the database engine.

This paper presents a general transaction reordering framework, which utilizes both the current system status and information about the interaction between queued and running transactions. The basic concept is simple and shown in Figure 1. In an RDBMS, generally, at any time there are M_1 transactions waiting in a FIFO transaction admission queue Q to be admitted to the system for execution, while another M_2 transactions forming a set S_r are currently running in the system. Such a transaction admission queue Q is commonly used for load control purpose [1, 2].

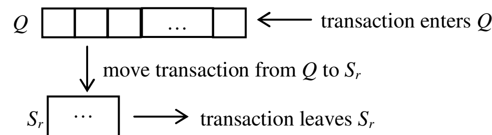


Figure 1. The general transaction reordering framework.

Those transactions in the transaction admission queue Q are the candidates for reordering. That is, the reorderer reorders the transactions waiting in Q so that the expected throughput of the reordered transaction sequence exceeds that of the original transaction sequence. In its reordering decisions, the reorderer exploits properties it deduces about the blocked transactions in Q and the properties it knows about the active transactions in S_r. The improvement in overall system throughput is a function of (a) the number of factors considered for reordering transactions, and (b) the quality of the original transaction sequence. The more factors considered, the better quality the reordered transaction sequence has. However, the time spent on reordering cannot be unlimited, as we need to ensure that the reordering overhead is smaller than the benefit we gain in throughput. Also, we need to ensure acceptable transaction response time in the sense that no transaction is subject to starvation.

There are a wide range of reordering algorithms that could be used. At the extremes, we could:

(1) Do no analysis. Run all the transactions in the order that they arrive at the RDBMS.

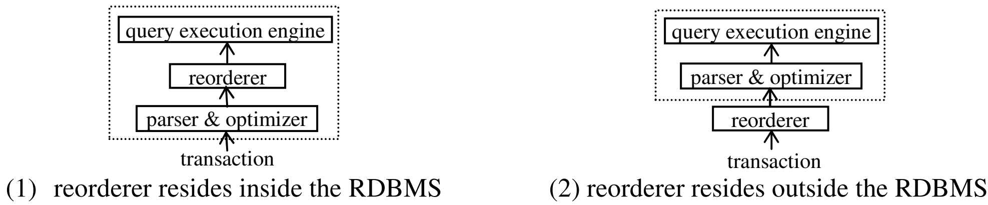
(2) Take a snapshot of the system. Analyze every possible order of the transactions and record the corresponding throughput. Pick the optimal order to run all the transactions.

The first extreme may be undesirable if some amount of reordering can improve the throughput. The second extreme is obviously unrealistic due to the exponential analysis overhead. Our goal is to find a good

compromise between these two extremes. That is, under the constraint of acceptable transaction response time, we want to maximize the difference between the gain in throughput and the reordering overhead.

Reordering transactions requires CPU cycles. However, the increasing disparity between CPU and disk performance renders trading CPU cycles for disk I/Os more attractive as a way of improving DBMS performance [3]. As shown in detail in Section 4 below, some forms of transaction reordering can be regarded as a way to trade CPU cycles for disk I/Os. Also, our experiments in three commercial RDBMSs show that with minor overhead, our transaction reordering method greatly improves the throughput of a targeted class of transactions while it has only a minor impact on the throughput of other classes of transactions.

There are many resource allocation factors that can be considered for transaction reordering. In this paper, due to space constraints, we only consider two factors: lock conflicts (with an application to materialized view maintenance [4]) and buffer pool performance (with an application to exploiting synchronized scans [5]).



(1) reorderer resides inside the RDBMS (2) reorderer resides outside the RDBMS

Figure 2. Transaction reordering architecture.

Transaction reordering can be implemented in two places: (1) inside the RDBMS, or (2) outside the RDBMS as an add-on module. These two choices are shown in Figure 2, where the dotted rectangle denotes the RDBMS. The inside-RDBMS choice affords more opportunities for reordering, as the reorderer is tightly integrated with the RDBMS and can use detailed information about the current state of the system. Also, certain reordering policy (such as the one described in Section 4 for exploiting synchronized scans) can only be implemented using the inside-RDBMS choice if it requires support of other modules in the RDBMS. The outside-RDBMS choice has the advantage of not needing to change the database engine and is especially suitable for database application developers. However, putting the reorderer outside the system means that it might have to treat the system as a black box and certain opportunities for reordering will be missed. It also requires an additional parsing of each transaction (once in the reorderer, once in the system). Section 5.3 gives an example of the outside-RDBMS choice.

Reordering transactions itself is not a new idea. RDBMS users sometimes order their transactions themselves before submitting those transactions to the DBMS [6]. However, to our knowledge the published literature has not considered a transaction reordering module that attempts to increase concurrency and share resource utilization.

In related work, the operating system community has explored the approach of adding a module outside of a system to reorder web server requests based on the knowledge of OS buffer contents [7, 8]. The database community has proposed multi-query optimization [9, 10] for resource sharing. The traditional multi-query optimization approach work in a batch fashion, as the optimizer needs to wait for a sufficient number of incoming queries with common sub-expressions to arrive, and then before executing them, changes their query plans to share common sub-expressions. Our transaction reordering method is dynamic and online: there is no need for either changing the query plans or waiting.

This paper is an extended version of our previous workshop papers [11, 12], and includes new material on deadlock probability computation in Section 3.3 and on the corresponding performance results in Section 5.1. The rest of this paper is organized as follows. Section 2 presents our general transaction reordering framework. Section 3 describes how to use lock conflict analysis as the reordering criterion. Section 4 discusses how to use buffer pool analysis as the reordering criterion. Section 5 investigates the performance of the transaction reordering method through an evaluation in three commercial RDBMSs. We conclude in Section 6.

2. General Transaction Reordering Framework

Transaction reordering is a general technique to improve RDBMS performance. It can be applied to multiple applications. In our discussion, we assume that all the transactions have the same priority. We assume that the strict two-phase locking protocol is used so that serializability is maintained. In this section, we discuss our general transaction reordering framework. In Sections 3 and 4, we show how to use lock conflict analysis and buffer pool analysis as the reordering criterion, respectively.

As mentioned in the introduction, in our model there are two sets of transactions in the RDBMS (Q and S_r). Q is the transaction admission queue that keeps the transactions waiting for execution. S_r is the set of active transactions that are currently running. Transactions enter Q in the order that they are submitted to the

RDBMS. Our goal is to reorder the transactions in Q so that the transaction throughput in the RDBMS is improved. We identify two specific methods to increase the transaction throughput: (1) increasing concurrency by preventing conflicting transactions from executing concurrently and (2) sharing resources among the (running) transactions.

In the general transaction reordering framework, we reorder transactions in the follow way:

(1) **Operation 1**: Suppose we want to schedule a transaction for execution. We scan Q sequentially until a desirable transaction T is found or we scan all the transactions in Q. A desirable transaction T is chosen according to some reordering criteria. If such a transaction is found, it is moved from Q to S_r and executed.

(2) **Operation 2**: Once a transaction is committed or aborted, it leaves S_r.

The basic transaction reordering framework needs to be extended when we take different factors into consideration (see Sections 3 and 4 for details).

When we search for the desirable transaction, we are essentially looking for a transaction that is compatible with the running transactions in S_r. That is, we implicitly divide transactions into different types and only concurrently execute the transactions that are of compatible types. The idea of using transaction types to improve database performance has been investigated previously [13, 14]. However, those methods are mainly used for concurrency control purpose rather than for reordering transactions. Also, their classification methods are different from ours:

(1) In Bernstein *et al*. [13], two transactions are of the same type if they have similar access patterns, conflict heavily, and cannot be interleaved. Here, in our classification, transactions of the same type ideally do not conflict and can be interleaved.

(2) The purpose of the classification in Garcia-Molina [14] is to allow non-serializable schedules which preserve consistency and which are acceptable to the users. In our transaction reordering method, we still preserve serializability. This is because we assume that the strict two-phase locking protocol is used and transaction reordering is done outside of the query execution engine.

3. Using Lock Conflict Analysis as the Reordering Criterion

In this section, we show how to use lock conflict analysis as the reordering criterion. Specifically, we use continuous data loading in the presence of materialized views as a concrete example to illustrate our techniques. We first provide some background on continuous data loading.

3.1 Continuous Data Loading

Today, an enterprise often has to make real-time decisions about its daily operations in response to the fast changes happening all the time in the world [15]. As a result, enterprises are starting to use operational data warehouses to provide fresher data and faster queries [16, 17]. In an operational data warehouse, the stored information is updated in real time or close to it. Also, materialized views are used to speed query processing.

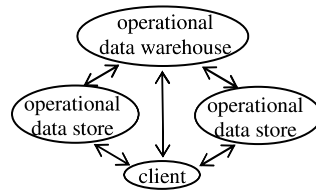


Figure 3. Operational data warehouse architecture.

Figure 3 shows the architecture of a typical operational data warehouse [16] (Wal-Mart's data warehouse uses this architecture [18]). Clients store new data into operational data stores in real time, where an operational data store is an OLTP database, a message queue [19], or anything else that is suitable for an OLTP workload. The purpose of these operational data stores is to acknowledge the clients' input immediately while ensuring the durability of this data. As quickly as feasible, this new data is transferred by continuous load utilities from operational data stores into a centralized operational data warehouse, where it is typically managed by an RDBMS. Then clients can query this operational data warehouse, which is the only place that global information is available.

Note: The continuous load utilities are not used for arbitrary applications. Rather, they are used to synchronize the centralized operational data warehouse with the operational data stores. As a result, existing commercial continuous load utilities (e.g., Oracle [20], Teradata [6]) have certain characteristics that are invalid in some applications and will be described below.

Since loading data into a database is a general requirement of database applications, most commercial RDBMS vendors provide load utilities, each of which have (somewhat) different functionality. Some are

continuous load utilities, while others only support batch bulk load. The functionality of certain load utilities can be implemented by applications. However, since a large number of applications need such functionality, RDBMS vendors typically provide this functionality as a package for application developers to use directly. In the rest of this paper, we do not differentiate between the load utilities provided by RDBMS vendors and the applications that are written by application developers and provide data loading functionality. We refer to both of them as load utilities, and our discussion holds for both. In this section, we describe how existing continuous load utilities typically work (minor differences in implementation details will not influence our general discussion).

3.1.1 Workload Specification

Figure 4 shows a typical architecture for loading data continuously into an RDBMS [6, 21]. Data comes from multiple data sources (files, OLTP databases, message queues, pipes, etc.) in the form of modification operations (insert, delete, or update). Then a continuous load utility loads the data into the RDBMS using update transactions. Each update transaction contains one or more modification operations. As is the case in data stream applications, the system has no control over the order in which modification operations arrive [22].

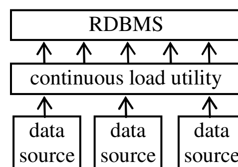


Figure 4. Continuous data loading architecture.

To decide which transformations are valid on the stream of load transactions, we discuss the semantics of continuous data loading. The state-of-the-art two popular commercial continuous load utilities (Oracle [20], Teradata [6]) make the following assumptions for continuous data loading:

(a) The RDBMS is running with standard ACID properties for transactions. The continuous load utility looks to the RDBMS like a series of transactions, each containing a single modification operation (insert, delete, or update) on a single relation. Hence, load transactions submitted by continuous load utilities will not cause inconsistency for transactions submitted by other applications.

(b) The RDBMS neither imposes nor assumes any particular order for these load transactions – indeed, their order is determined by the (potentially multiple) external systems feeding the load process. Hence, the load process is free to arbitrarily reorder these transactions.

(c) The RDBMS has no requirement on whether multiple modification operations can or cannot commit/abort together. Hence, for efficiency purposes, the load process is free to arbitrarily group these single-modification-operation transactions.

In this paper, we make the same assumptions. Hence, in our techniques, we can do reordering and grouping arbitrarily.

The alert reader may notice that arbitrary reordering can cause certain anomalies. For example, such an anomaly arises if the deletion of a tuple t is moved before the updating of tuple t. In practice, some applications tolerate such anomalies [personal communication with S. Brobst]. In other cases, the application ensures that the order in which modification operations arrive at the continuous load utility will not allow such anomalies. For example, before the continuous load utility acknowledges the completion of updating tuple t, the operation of deleting tuple t is not submitted to the continuous load utility. In either case, the continuous load utility does not need to worry about these anomalies.

To increase concurrency, a continuous load utility typically opens multiple sessions to the RDBMS (at any time, each session can have at most one running transaction [23, page 320]). These sessions are usually maintained for a long time so that they do not need to be re-established for each use. For efficiency, within a transaction, all the SQL statements corresponding to modification operations are usually pre-compiled into a stored procedure whose execution plan is stored in the RDBMS. This not only reduces the network overhead (transmitting a stored procedure requires a much smaller message than transmitting multiple SQL statements) but also eliminates the overhead of repeatedly parsing and optimizing SQL statements.

3.1.2 Grouping Modification Operations

Continuous load utilities usually combine multiple modification operations into a single transaction rather than applying each modification operation in a separate transaction [6, 21]. This is because of the per transaction overhead. Using a large transaction can amortize this overhead over multiple modification

operations. In the rest of this paper, we refer to the number of modification operations that are combined into a single transaction as the *grouping factor*.

3.1.3 The Partitioning Method

In this section, we review the standard approach used to avoid deadlock in continuous load operations in the absence of materialized views. Suppose the continuous load utility opens $k \geq 2$ sessions S_i ($1 \leq i \leq k$) to the RDBMS. If we randomly distribute the modification operations among the k sessions, transactions from different sessions can easily deadlock on X lock requests on the base relations. This is because these transactions may modify the same tuples concurrently [6]. A simple solution to this deadlock problem is to partition (e.g., hash on some attribute) the tuples among different sessions so that modification operations on the same tuple are always sent through the same session [6]. In this way, the deadlock condition (transactions from different sessions modify the same tuple) no longer exists and deadlocks will not occur. (Note: the partitioning method may change the order that the tuples arrive at the RDBMS. However, as mentioned in Section 3.1.1, such reordering is allowed in existing continuous load utilities.)

3.2 Impact of Immediate Materialized View Maintenance

In this section, we consider the general case in which materialized views are maintained in the RDBMS, and show that in this case the partitioning method of Section 3.1.3 is not sufficient to avoid deadlocks. We focus on an important class of materialized views called *join views*. In an extended relational algebra, by a join view *JV*, we mean either an ordinary join view $\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_h))$ or an aggregate join view $\gamma(\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_h)))$, where γ is an aggregate operator. SQL allows the aggregate operators *COUNT*, *SUM*, *AVG*, *MIN*, and *MAX*. However, because *MIN* and *MAX* cannot be maintained incrementally (the problem is deletes [24]), we restrict our attention to the three aggregate operators that make the most sense for materialized aggregates: *COUNT*, *SUM*, and *AVG*.

In continuous data loading, we allow data to be loaded into multiple base relations concurrently. This is necessary if we want to keep the data in the RDBMS as up-to-date as possible. However, if a join view is defined on multiple base relations, deadlocks are likely to occur. This is because a join view *JV* links different

base relations. When a base relation of *JV* is updated, to maintain *JV*, all the other base relations in the definition of *JV* are read. That is, the introduction of the join view changes the update transactions into update-read transactions. These reads can conflict with concurrent writes to the other base relations of *JV*. For example, consider the following two base relations: $A(a, c)$ and $B(d, e)$. Suppose a join view $JV=A \bowtie B$ is defined on *A* and *B*, where the join condition is $A.c=B.d$. Consider the following two modification operations:

(1) O_1: Modify a tuple t_1 in base relation *A* whose $c=v$.

(2) O_2: Modify a tuple t_2 in base relation *B* whose $d=v$.

These modification operations require the following tuple-level locks on base relations *A* and *B*:

O_1: (L_{11}) A tuple-level X lock on *A* for tuple t_1.

(L_{12}) Several tuple-level S locks on *B* for all the tuples in *B* whose $d=v$ (for join view maintenance purpose).

O_2: (L_{21}) A tuple-level X lock on *B* for tuple t_2.

(L_{22}) Several tuple-level S locks on *A* for all the tuples in *A* whose $c=v$.

Suppose operation O_1 is executed by transaction T_1 through session S_1, while operation O_2 is executed by transaction T_2 through session S_2. If transactions T_1 and T_2 request the locks in the order

(1) **Step 1**: T_1 requests L_{11}.

(2) **Step 2**: T_2 requests L_{21}.

(3) **Step 3**: T_1 requests L_{12}.

(4) **Step 4**: T_2 requests L_{22}.

a deadlock occurs. This is because L_{11} (L_{22}) contains a tuple-level X (S) lock on *A* for tuple t_1. Also, L_{21} (L_{12}) contains a tuple-level X (S) lock on *B* for tuple t_2.

A simple solution to the above deadlock problem is to do materialized join view maintenance in a deferred manner rather than immediately. That is, an update is inserted into the base relation as soon as possible; but the materialized join views that refer to that base relation only see the update at some later time, when the materialized join views are updated in a batch operation. Unfortunately, this makes the materialized join views at least temporarily inconsistent with the base relations. The resulting semantic uncertainty may not be acceptable to all applications. This observation has been made elsewhere. For example, Graefe *et al.* [25, 26]

emphasizes that consistency is important for materialized views that are used to make real-time decisions. As another example, in the TPC-R benchmark, maintaining materialized views immediately with transactional consistency is a mandatory requirement [27], presumably as a reflection of some real world application demands. As a third example, as argued in Graefe and Zwilling [25], materialized views are like indexes. Since indexes are always maintained immediately, immediate materialized view maintenance should also be desirable in many cases.

The reader might wonder whether using a multi-version concurrency control method can solve the above deadlock problem. In general, a multi-version concurrency control method can avoid conflicts between a pure read transaction and a write transaction (or a transaction that does both reads and writes) [25, 28]. However, in our case, the immediate materialized join view maintenance transactions do both reads and writes. As a result, a multi-version concurrency control method cannot avoid the conflicts between these transactions [25, 28]. In fact, Graefe and Zwilling [25] proposed a multi-version concurrency control method to avoid conflicts between pure read transactions on materialized join views and immediate materialized join view maintenance transactions. For this reason, in this paper, we do not discuss pure read transactions on materialized join views.

Allowing dirty reads is a standard technique to improve the concurrency of read-only queries. Since materialized join view maintenance has at its heart a join query, it is natural to wonder if dirty reads can be used here. Unfortunately, in the context of materialized view maintenance, allowing dirty reads is problematic. This is because using dirty reads to maintain join views makes the results of these dirty reads permanent in the join views [29]. Thus, although dirty reads would avoid the deadlock problem, they cannot be used.

It is also natural to question whether some extension of the partitioning method described in Section 3.1.3 can be used to avoid deadlocks in the presence of materialized join views. In certain cases, the answer is yes. For example, suppose we use the same partitioning function to partition the tuples of A and B among different sessions according to the join attributes $A.c$ and $B.d$, respectively. Then for immediate materialized view maintenance, the deadlock problem will not occur. This is because in this case, conflicting transactions are always submitted through the same session. Also, at any time, one session can have at most one running transaction [23, page 320]. Unfortunately, in practice, such an appropriate partitioning method is not always possible:

(1) In continuous data loading, modification operations on a base relation *R* usually specify some (e.g., the primary key) but not all attribute values of *R* [6]. We can only partition the tuples of base relation *R* among different sessions according to (some of) those attributes whose values are specified by the modification operations on *R*. This is because we use the same attributes to partition the modification operations on base relation *R* among different sessions. Suppose that base relation *R* is a base relation of a join view. Also, suppose the join attribute of *R* is not one of those attributes whose values are specified by the modification operations on *R*. Then we cannot partition the tuples of base relation *R* among different sessions according to the join attribute of *R*.

(2) If multiple join views with different join attributes are defined on the same base relation *R*, then it is impossible to partition the tuples of base relation *R* among different sessions according to these join attributes simultaneously.

(3) If within the same join view (e.g., $JV=A \bowtie R \bowtie B$), a base relation *R* is joined with multiple other base relations (e.g., *A* and *B*) on different join attributes, then it is impossible to partition the tuples of base relation *R* among different sessions according to these join attributes simultaneously.

3.3 Deadlock Probability

In this section we show that, contrary to the situation in the absence of materialized join views, in the presence of materialized join views, the probability of deadlock can easily be very high. For example, suppose

(1) There are $k>1$ concurrent transactions.

(2) Each transaction contains *n* modification operations and modifies either *A* or *B* with probability *p* and $1-p$, respectively.

(3) Within a transaction, each modification operation modifies a random tuple in *A* (*B*) and each of the *n* tuples to be modified has a distinct (and random) *A.c* (*B.d*) value.

(4) There are totally *s* distinct values for *A.c* (*B.d*).

(5) $s>>kn$.

Then following a reasoning that is similar to Gray and Reuter [23, page 428-429], we can show that the probability that any particular transaction deadlocks is approximately $p(1-p)(k-1)n^2/(2s)$. (If we do not have

$s>>kn$, then this deadlock probability is essentially 1. Hence, no matter whether $s>>kn$ or not, we can use a unified formula $min(1, p(1-p)(k-1)n^2/(2s))$ to roughly estimate the probability that any particular transaction deadlocks.)

This probability can be derived as follows. Consider a particular transaction T of the k transactions. There are two cases:

(1) Case 1: Transaction T modifies base relation A. From transaction T's perspective, there are $k-1$ other transactions, where a $1-p$ fraction of them modify base relation B. Each of these $(k-1)(1-p)$ transactions holds approximately $n/2$ sets of locks of the form L_{21} and L_{22}. Hence, these $(k-1)(1-p)$ transactions hold $(k-1)(1-p)n/2$ sets of locks. For any modification operation MO_1 of transaction T, the probability that it deadlocks with some modification operation MO_2 of another transaction T' is $PW_1=(k-1)(1-p)n/(2s)\times(1/2)=(k-1)(1-p)n/(4s)$. This is because:

   (a) The probability that the tuples modified by MO_1 and MO_2 have the same value for $A.c$ ($B.d$) is approximately $(k-1)(1-p)n/(2s)$.

   (b) In the case that the tuples modified by MO_1 and MO_2 have the same value for $A.c$ ($B.d$), the probability that MO_1 and MO_2 deadlock is ½ (depending on whether or not step 2 occurs before step 3).

   Transaction T contains n modification operations. Therefore, the probability that transaction T deadlocks is $PW_1(T)=1-(1-PW_1)^n\approx n\times PW_1=(k-1)(1-p)n^2/(4s)$.

(2) Case 2: Transaction T modifies base relation B. In this case, following a reasoning that is similar to Case 1, we can show that the probability that transaction T deadlocks is $PW_2(T)\approx(k-1)pn^2/(4s)$.

Case 1 happens with probability p. Case 2 happens with probability $1-p$. Hence, for any particular transaction T, the probability that transaction T deadlocks is $PW(T)=p\times PW_1(T)+(1-p)\times PW_2(T)=p(1-p)(k-1)n^2/(2s)$.

For reasonable values of k, n, and s, this deadlock probability is unacceptably high. This is mainly due to the following reasons:

(1) For efficiency purposes, n could be large (e.g., 600 [6]).

(2) The deadlock probability formula that is in Gray and Reuter [23, page 429] is of the form $kn^4/(4s^2)$, where s is the number of distinct tuples. Unlike that formula, the s in the denominator of our formula:

(a) is the number of distinct attribute values, which is usually smaller than the number of distinct tuples.

(b) has an exponent of 1 rather than 2.

As an example, if *p=50%*, *k=8*, *n=32*, and *s=10,000*, this deadlock probability is approximately 9%. Doubling *n* to 64 raises this probability to 36%. For a larger *n*, the deadlock probability could easily get close to 1. Note: we are not trying to use the above formula to precisely predict the deadlock probability. Rather, we use the formula to give a rough estimate of the deadlock probability and demonstrate how severe the deadlock problem could be. In Section 5.1 below, we validate this formula through a study of the deadlock problem in a commercial RDBMS.

3.4 Solution with Reordering

The deadlock problem occurs because we allow data to be concurrently loaded into multiple base relations of the same join view. Hence, a natural question is if this were not allowed, would the deadlock problem still occur? Luckily, the answer is "no" if we set the following rules:

(1) **Rule 1**: At any time, for any join view *JV*, data can only be loaded into one base relation of *JV*.

(2) **Rule 2**: Modification operations (insert, delete, update) on the same base relation use the partitioning method discussed in Section 3.1.3.

(3) **Rule 3**: The system uses a high concurrency locking protocol (e.g., the V locking protocol [30], or the locking protocol in Graefe and Zwilling [25]) on join views so that lock conflicts on the join views can be avoided.

The reason is as follows.

(1) Using rules 1 and 2, all deadlocks resulting from lock conflicts on the base relations are avoided.

(2) Using rule 3, all deadlocks resulting from lock conflicts on the join views can be avoided (e.g., in the V locking protocol [30], V locks are compatible with themselves; in the locking protocol in Graefe and Zwilling [25], E locks are compatible with themselves).

Since all possible deadlock conditions are eliminated, deadlocks no longer occur.

We now consider how to implement rules 1-3. It is easy to enforce rules 2 and 3. To enforce rule 1, we can use the following reordering method to reorder the modification operations. Recall in Section 3.1.1, the

semantics of the workload allows us to reorder modification operations arbitrarily. Consider a database with d base relations R_1, R_2, …, and R_d and e join views JV_1, JV_2, …, and JV_e. We keep an array J that contains d elements J_i ($1 \le i \le d$). For each i ($1 \le i \le d$), J_i records the number of transactions that modify base relation R_i and are currently being executed. Each J_i ($1 \le i \le d$) is initialized to zero. For each m ($1 \le m \le k$), we maintain a queue Q_m recording transactions waiting to be run through session S_m. Each Q_m ($1 \le m \le k$) is initialized to empty. During grouping (see Section 3.1.2), we only combine modification operations on the same base relation into a single transaction.

If base relations R_i and R_j ($1 \le i, j \le d$, $i \ne j$) are base relations of the same join view, we say that R_i and R_j conflict with each other. Two transactions modifying conflicting base relations are said to conflict with each other. We call transaction T a desirable transaction if it does not conflict with any currently running transaction. Consider a particular base relation R_i ($1 \le i \le d$). Suppose R_{s_1} , R_{s_2} , …, and R_{s_w} ($w \ge 0$) are all the other base relations that conflict with base relation R_i. At any time, if either $w=0$ or all the $J_{s_u} = 0$ ($1 \le u \le w$), then a transaction T modifying base relation R_i ($1 \le i \le d$) is a desirable transaction.

We schedule transactions as follows:

(1) **Action 1:** For each session S_m ($1 \le m \le k$), as discussed in Section 3.1.2, whenever the continuous load utility has collected n modification operations on a base relation R_i ($1 \le i \le d$), we combine these operations into a single transaction T and insert transaction T to the end of Q_m. Here, n is the pre-defined grouping factor that is specified by the user who sets up the continuous load utility. If session S_m is free, we try to schedule a transaction to the RDBMS for execution through session S_m.

(2) **Action 2:** When some transaction T modifying base relation R_i ($1 \le i \le d$) finishes execution and frees session S_m ($1 \le m \le k$), we do the following:

    (a) We decrement J_i by one.

    (b) If Q_m is not empty, we schedule a transaction to the RDBMS for execution through session S_m.

    (c) Suppose J_i is decremented to zero (so that some waiting transaction possibly becomes desirable). For each g ($1 \le g \le k$, $g \ne m$), if session S_g is free and Q_g is not empty, we try to schedule a transaction to the RDBMS for execution through session S_g.

(3) **Action 3:** Whenever we try to schedule a transaction to the RDBMS for execution through session S_m ($1 \leq m \leq k$), we do the following:

   (a) We search Q_m sequentially until either a desirable transaction T is found or all the transactions in Q_m have been scanned, whichever comes first.

   (b) In the case that a desirable transaction T modifying base relation R_i ($1 \leq i \leq d$) is found, we increment J_i by one and send transaction T to the RDBMS for execution.

Reordering transactions may cause slight delays in the processing of load transactions that have been moved later in the load schedule. On balance, these delays will be offset by the corresponding transactions that were moved earlier in the schedule to take the place of these delayed transactions. For some applications, this reordering is preferable to the inconsistencies that result from deferred materialized view maintenance. These are the target applications for our reordering technique.

The above discussion does not address starvation. There are several starvation prevention techniques that can be integrated into the transaction reordering method. We list one of them as follows. The idea is to use a special header transaction to prevent the first transaction in any Q_g from starvation ($1 \leq g \leq k$). We keep a pointer r whose value is always between 0 and k. r is initialized to 0. If every Q_m ($1 \leq m \leq k$) is empty, $r=0$. At any time, if $r=0$ and a transaction is inserted into some Q_m ($1 \leq m \leq k$), we set $r=m$. If $r=m$ ($1 \leq m \leq k$) and the first transaction of Q_m leaves Q_m for execution, r is incremented by one (if $m=k$, we set $r=1$). If Q_r is empty, we keep incrementing r until either Q_r is not empty or we discover that every Q_m ($1 \leq m \leq k$) is empty. In the later case, we set $r=0$. We make use of a pre-defined timestamp TS determined by application requirements. If pointer r has stayed at some v ($1 \leq v \leq k$) longer than TS, the first transaction of Q_v becomes the header transaction. Whenever we are searching for a desirable transaction in some Q_m ($1 \leq m \leq k$) and we find transaction T, if the header transaction exists, we ensure that either T is the header transaction or T does not conflict with the header transaction. Otherwise transaction T is still not desirable and we continue the search.

4. Using Buffer Pool Analysis as the Reordering Criterion

In this section, we show how buffer pool analysis can be used as the reordering criterion. When we mention a transaction T that does full table scan on relation R, we mean that transaction T only reads

relation R and executes no other operations. We use synchronized scan [31] as a concrete example to illustrate our techniques. We first show in Section 4.1 that the existing buffer management methods cannot utilize the synchronized scan technique efficiently when the RDBMS is heavily loaded. Then in Section 4.2, we provide a solution to this problem using transaction reordering.

4.1 Synchronized Scans and Load Management

In a typical data warehouse, there are a few very large relations with multiple queries submitted against them simultaneously. Some of these queries involve expensive full table scans. Such full table scans are unavoidable, as it is impossible to predict every possible access path into these large relations and/or afford the disk space and maintenance overhead to create all the indices that might be needed [32]. As ad hoc querying of data warehouses is becoming more common [27, 33, 34], the number of full table scans is greatly increased. Such a large number of expensive full table scans will consume a large portion of the disk I/O capability in the RDBMS and significantly decrease the amount of disk I/O capability available to the other transactions. To attack this problem, people have developed the synchronized scan technique that is available in at least four commercial database systems: Teradata [31], Red Brick [5], Microsoft SQL Server [35], and IBM DB2 [33, 34]. The main idea of the synchronized scan technique is that if two transactions are scanning the same relation, then we can group them together so that I/Os can be shared between them. This reduces the cumulative number of I/Os required by the scans while additionally saving CPU cycles that would otherwise have been required to process the extra I/Os.

Synchronized scans are typically implemented in the following way (minor differences in implementation details will not influence our transaction reordering algorithm). Consider a relation R containing K_1 pages in total. When a transaction T_1 starts a full table scan on relation R, we add some information recording this fact into an in-memory data structure DS (this information is dropped out of DS when transaction T_1 finishes the scan). Also, transaction T_1 keeps K_2 buffer pages (a predefined number) as a cache to hold the most recent K_2 pages that it just accessed in relation R. When a second transaction T_2 starts a full table scan on relation R, we first check the in-memory data structure DS to see whether some transaction is currently scanning relation R or not. If so, e.g., suppose transaction T_1 is processing the J-th page of relation R, then transaction T_2 starts

scanning relation R from the J-th page. In this way, transactions T_1 and T_2 can hopefully share the K_1-$J+1$ I/Os when they scan relation R. When transaction T_2 finishes processing the last page of relation R, it goes back to the beginning of relation R to make up the previously omitted first J-1 pages (other transactions may do synchronized scan with transaction T_2 for these J-1 pages). Note transactions T_1 and T_2 are not always locked together, but may drift apart if the required processing time for the scans differ too much from each other. For example, as long as the two scans are separated by less than K_2 pages, the K_2 buffer pages are used to save the intermediate blocks until the slower scan catches up. However, if the divergence exceeds K_2 pages, the two scans are separated and run independently, and no caching is performed. This prevents us from experiencing large response times due to a fast scan waiting for a slow scan to catch up.

From the above description, we can see that after transaction T_2 joins transaction T_1 for synchronized scan, transaction T_2 does not consume many extra buffer pages (except for a few buffer pages to temporarily store the query results) unless sometime later the two scans drift too far away from each other. However, the latter situation does not occur frequently. This is because the fast scan needs to be in charge of doing the time-consuming operation of fetching pages from disk into the buffer pool. During this period, the slow scan can catch up, as all the pages that it is currently working on reside in the buffer pool. Also, Lang *et al*. [33, 34] proposed a few techniques to reduce the likelihood that two scans drift too far away from each other.

The state-of-the-art buffer management algorithms cannot utilize the synchronized scan technique efficiently when the RDBMS is heavily loaded. This is because in a typical buffer management algorithm [2, 36, 37, 38], after all the buffer pages in the buffer pool are committed, no new transactions are allowed to enter the RDBMS for execution. (In fact, in a typical implementation, after a large percent (not all — this is mainly for the purpose of safety) of the buffer pages in the buffer pool are committed, no new transactions are allowed to enter the RDBMS for execution.) That is, after all the buffer pages are used up, even if some transaction T_1 is currently doing a full table scan on relation R, a new transaction T_2 scanning relation R is not allowed to enter the system to join transaction T_1 for synchronized scan. However, in this case, synchronized scan would be desirable (i.e., we should push transaction T_2 to enter the system for execution), as it usually does not consume many extra buffer pages (except for a few buffer pages to temporarily store the query results). Later, when transaction T_2 is finally allowed to enter the system, transaction T_1 may have already

finished execution so that transaction T_2 cannot utilize synchronized scan any more. Rather, transaction T_2 needs to reread all the pages of relation R from disk into the buffer pool. This leads to the waste of a large number of disk I/Os and CPU cycles.

4.2 Applying Transaction Reordering

To address the above problem, we use buffer pool analysis as another reordering criterion. This is to maximize the chance that the synchronized scan technique can be utilized. In the discussion below, we only apply synchronized scan to transactions (queries) that do full table scan on a single relation. The case with more complex transactions (e.g., queries including joins) is left for future work.

Technique 1: We maintain an in-memory hash table *HT* that keeps track of all the full table scans in the transaction admission queue Q. Each element in *HT* is of the following format: (relation name, list of transactions in Q that does full table scan on this relation). Each time we find a desirable transaction T in Q, if transaction T does full table scan on relation R, we move some (or all) of the transactions in Q that does full table scan on relation R to S_r for execution. Note we may not be able to move all such transactions in Q to S_r for execution. For example, the system may not have enough threads to run all such transactions in Q. However, as long as the system permits, we move as many such transactions to S_r as possible.

Technique 2: When a new transaction T that does full table scan on relation R arrives, before it is blocked in Q, we first check the data structure *DS* to see whether some transaction in S_r is currently doing a full table scan on relation R. If so, and if we have threads available and the system is not on the edge of thrashing due to a large number of lock conflicts [1], we run transaction T immediately so that it does not get blocked in Q. Note in this case, transaction T does not have table-level lock conflict (on relation R) with any transaction in S_r, otherwise it is impossible to have a transaction in S_r that is currently doing a full table scan on relation R.

Multiple scans in the same synchronized scan group may occasionally get separated if their scanning speeds differ too much from each other (as explained in Section 4.1, such chance is very low). This would cause synchronized scan to consume (possibly a large number of) extra buffer pages so that the system may be running out of buffer pages (in this case, the system may abort some running transactions). If this happens, or

if the system is running out of threads or on the edge of thrashing due to a large number of lock conflicts, we stop using Technique 1 and Technique 2 until the system returns to normal state.

In a typical scenario, most long-running transactions in the RDBMS are I/O-bound rather than CPU-bound [3]. Our transaction reordering method for exploiting synchronized scans requires a few CPU cycles and can be regarded as a way to trade CPU cycles for disk I/Os. It can greatly improve the throughput of a targeted class of transactions that can share synchronized scans and reduce the processing load on the database engine, while it has only a minor impact on the throughput of other classes of transactions. This is because the extra transactions that are scheduled to run by our transaction reordering method use synchronized scans and basically do not compete with existing transactions on I/Os.

5. Performance Evaluation

In this section, we describe experiments that were performed in three commercial parallel RDBMSs: IBM DB2, Teradata, and another commercial RDBMS. We investigated the performance of the transaction reordering method when either lock conflict analysis or buffer pool analysis was considered. In either case (except in Section 5.3), we focus on the throughput of a targeted class of transactions (i.e., transactions that have lock conflicts or may share table scans). This is because in a mixed workload environment, our method would greatly improve the throughput of the targeted class of transactions while the throughput of other classes of transactions would remain much the same. Our measurements were performed with the database client application and server running on an Intel x86 Family 6 Model 5 Stepping 3 workstation with four 400MHz processors, 1GB main memory, six 8GB disks, and running the Microsoft Windows 2000 operating system. We allocated a processor and a disk for each data server, so there were at most four data servers on each workstation.

5.1 Experiments in IBM DB2

In IBM DB2, we investigated the performance of the transaction reordering method when lock conflict analysis was considered for continuous data loading.

5.1.1 Experiment Description

The relations used for the tests model a real world scenario. Customers interact with a retailer via phone/web to make a purchase. The purchase involves browsing available merchandise items and possibly selecting an item to purchase. The following events occur:

(1) Customer indicates desire for a specific item and event is recorded in the *demand* relation.

(2) The *inventory* relation is checked for item availability.

(3) If the desired item is on hand, a customer order is placed and the *inventory* relation is updated; otherwise a vendor order is placed.

Table 1. Test data set in IBM DB2.

	number of tuples	total size
demand	8M	910MB
inventory	2M	77MB

The schemas of the *demand* and *inventory* relations are listed as follows:

demand (partkey, date, quantity, custkey, comment),

inventory (partkey, date, quantity, extended_cost, extended_price).

The underscore indicates the partitioning attributes. For each relation, we built an index on the partitioning attribute(s). In our tests, each *inventory* tuple matches 4 *demand* tuples on the attributes *partkey* and *date*. Also, different *demand* tuples have different *custkey* values. In practice, there can be a large number of different parts. However, for any given day, most transactions only focus on a small portion of them (the active parts). In our testing, we assume that *s* parts are active today. We only consider today's transactions that are related to these active parts. We believe that our conclusion would remain much the same if all transactions related to both active and inactive parts were considered. This is because in this case, the number of deadlocks caused by the transactions that are related to the active parts would remain much the same.

Suppose that the *demand* and *inventory* relations are frequently queried for sales forecasting, lost sales analysis, and assortment planning applications, so a join view *onhand_demand* is built as the join result of *demand* and *inventory* on the join attributes *partkey* and *date*:

create join view onhand_demand as select d.partkey, d.date, d.quantity, d.custkey, i.quantity

from demand d, inventory i where d.partkey=i.partkey and d.date=i.date partitioned on d.custkey;

There are two kinds of modification operations that we used for testing, both of which are related to today's activities:

(1) O_1: Insert one tuple (with today's *date*) into the *demand* relation. This new tuple matches 1 *inventory* tuple on the attributes *partkey* and *date*.

(2) O_2: Update one tuple in the *inventory* relation with a specific *partkey* value and today's *date*.

We created an auxiliary relation for the *demand* relation that is partitioned on the (*partkey*, *date*) attributes to change expensive all-node join operations for join view maintenance to cheap single-node join operations [39].

We evaluated the performance of the reordering method and the naive method in the following way:

(1) We tested the largest available hardware configuration with four data server nodes.

(2) We executed a stream of modification operations. A fraction p of these modification operations are O_1. The other $1-p$ of the modification operations are O_2. Each O_1 inserts a tuple into the *demand* relation with a random *partkey* value. Each O_2 updates a tuple in the *inventory* relation with a random *partkey* value.

(3) In both the reordering method and the naive method, we only combine modification operations on the same base relation into a single transaction. Each transaction has the same grouping factor n.

(4) In the naive method (without reordering), if a transaction deadlocked and aborted, we automatically re-executed it until it committed.

(5) We performed three tests:

    (a) **Concurrency test**: We fixed $p=50\%$ and the number of active parts $s=10,000$. In both the reordering method and the naive method, we tested four cases: $k=2$, $k=4$, $k=8$, and $k=16$, where k is the number of sessions. In each case, we let the grouping factor n vary from 1 to 128.

    (b) **Data ratio test**: We fixed $k=16$, $n=64$, and $p=50\%$. In both the reordering method and the naive method, we let the number of active parts s vary from 5,000 to 20,000.

    (c) **Transaction ratio test**: We fixed $k=16$, $n=64$, and $s=10,000$. In both the reordering method and the naive method, we let p vary from 12.5% to 87.5%.

5.1.2 *Concurrency Test Results*

We first discuss the deadlock probability and throughput testing results from the concurrency test.

5.1.2.1 Deadlock Probability

As mentioned in Section 3.3, for the naive method, we can use the unified formula $min(1, p(1-p)(k-1)n^2/(2s))$ to roughly estimate the probability that any particular transaction deadlocks. We show the deadlock probability of the naive method computed by the unified formula in Figure 5. (Note: all figures in Sections 5.1.2.1 and 5.1.2.2 use logarithmic scale for the x-axis.)

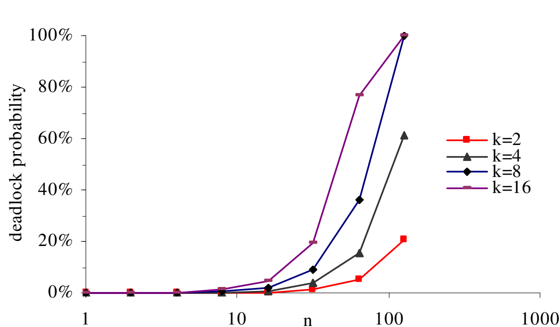


Figure 5. Predicted deadlock probability of the naive method (concurrency test).

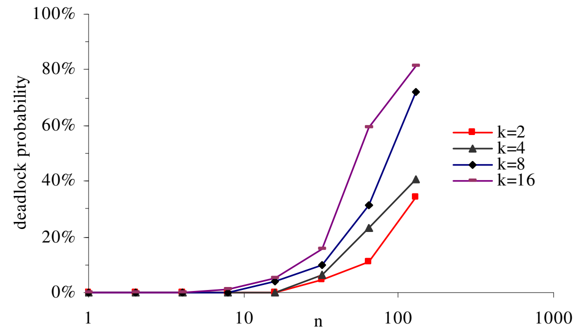


Figure 6. Measured deadlock probability of the naive method (concurrency test).

For the naive method, the deadlock probability is linear in the number of sessions k and quadratic in the grouping factor n. When both k and n are small, this deadlock probability is small. However, when either k or n becomes large, this deadlock probability approaches 1 quickly. For example, consider the case with $n=64$. When $k=2$, this deadlock probability is only 5%. However, when $k=16$, this deadlock probability becomes 77%. The larger k, the smaller n is needed to make this deadlock probability become close to 1.

We show the deadlock probability of the naive method measured in our tests in Figure 6. Figures 5 and 6 roughly match. This indicates that our unified formula is fairly good for the purpose of giving a rough estimate of the deadlock probability of the naive method.

5.1.2.2 Throughput

The throughput (number of modification operations per second) is an important performance metric of the continuous load utility. For the naive method, to see how deadlocks influence its performance, we investigated the relationship between the throughput and the deadlock probability.

By definition, when the deadlock probability becomes close to 1, almost every transaction will deadlock. Deadlock has the following negative influences on throughout:

(1) Deadlock detection/resolution is a time-consuming process. During this period, the deadlocked transactions cannot make any progress.

(2) The deadlocked transactions will be aborted and re-executed. During re-execution, these transactions may deadlock again. This wastes system resources.

Hence, once the system starts to deadlock, the deadlock problem tends to become worse and worse. Eventually, the throughput of the naive method deteriorates significantly.

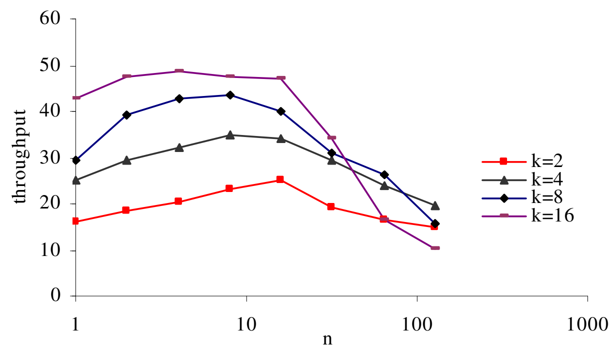


Figure 7. Throughput of the naive method (concurrency test).

We show the throughput of the naive method in Figure 7. For a given number of sessions *k*, when the grouping factor *n* is small, the throughput of the naive method keeps increasing with *n*. This is because executing a large transaction is more efficient than executing a large number of small transactions, as discussed in Section 3.1.2. (In our testing, the performance advantages of having a large grouping factor *n* are not very large. This is mainly due to the fact that due to software restrictions, we could only run the database client application and server on the same computer. In this case, the overhead per transaction is fairly low. Amortizing such a small overhead with a large *n* cannot bring much benefit.) When *n* becomes large enough, if the naive method does not run into the deadlock problem, the throughput of the naive method approaches a constant, where the system resources become fully utilized. The larger *k*:

(1) the higher concurrency in the RDBMS and the larger the constant.

(2) the easier it becomes to achieve full utilization of system resources and the smaller *n* is needed for the throughput to achieve that constant.

When n becomes too large, the naive method runs into the deadlock problem. The larger k, the smaller n is needed for the naive method to run into the deadlock problem. Once the deadlock problem occurs, the throughput of the naive method deteriorates significantly. Actually, it decreases as n increases. This is because the larger n, the more transactions are aborted and re-executed due to deadlock.

For a given n, before the deadlock problem occurs, the throughput of the naive method increases with k. This is because the larger k, the higher concurrency in the RDBMS. However, when n is large enough (e.g., $n=128$) and the naive method runs into the deadlock problem, due to the extreme overhead of repeated transaction abortion and re-execution, the throughput of the naive method may decrease as k increases.

We show the throughput of the reordering method in Figure 8. The general trend of the throughput of the reordering method is similar to that of the naive method (before the deadlock problem occurs). That is, the throughput of the reordering method increases with both n and k. For a given k, as n becomes large, the throughput of the reordering method approaches a constant. However, the reordering method never deadlocks. For a given k, the throughput of the reordering method keeps approaching that constant no matter how large n is. Once the naive method runs into the deadlock problem, the reordering method exhibits great performance advantages over the naive method, as the throughput of the naive method in this case deteriorates significantly.

In both the $k=8$ case and the $k=16$ case, when n becomes large enough, the throughput of the reordering method approaches (almost) the same constant. This is because in these two cases, all data server nodes (e.g., disk I/Os) become fully utilized. In our testing, if we had a larger hardware configuration with more data server nodes, the constant for the $k=16$ case would be larger than that for the $k=8$ case.

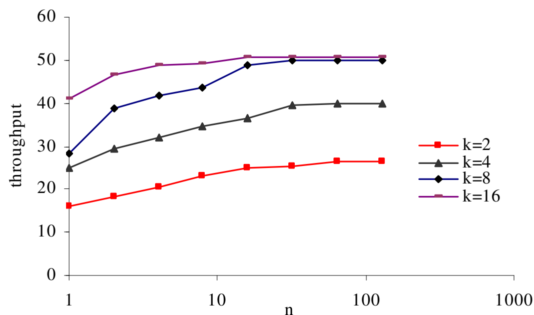


Figure 8. Throughput of the reordering method (concurrency test).

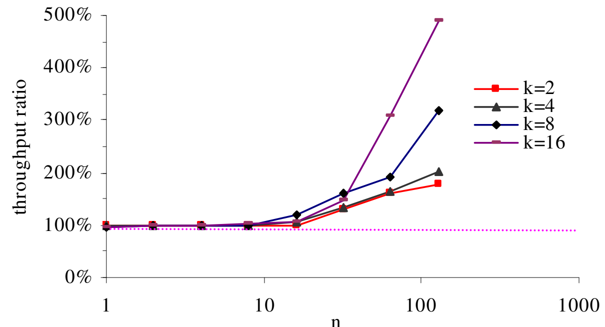


Figure 9. Throughput improvement gained by the reordering method (concurrency test).

We show the ratio of the throughput of the reordering method to that of the naive method in Figure 9. Before the naive method runs into the deadlock problem, the throughput of the reordering method is smaller than that of the naive method. This is because the reordering method has some overhead in performing reordering and synchronization (i.e., switching from executing one type of transactions (say, transactions updating the *inventory* relation) to executing another type of transactions (say, transactions updating the *demand* relation)). However, such overhead is not significant. In our tests, the throughput of the reordering method is never lower than 96% of that of the naive method.

When the naive method runs into the deadlock problem, the throughput of the reordering method does not drop while the throughput of the naive method is significantly worse. In this case, the ratio of the throughput of the reordering method to that of the naive method is greater than 1. For example, when $n=32$, for any k, this ratio is at least 1.3. When $n=64$, for any k, this ratio is at least 1.6. In the extreme case when $k=16$ and $n=128$, this ratio is 4.9. In general, when the naive method runs into the deadlock problem, this ratio increases with both k and n. This is because the larger k or n, the easier the transactions deadlock in the naive method. The extreme overhead of repeated transaction abortion and re-execution exceeds the benefit of the higher concurrency (efficiency) brought by a larger k (n). However, there are two exceptions. When $n=16$ or $n=32$, the ratio curve for $k=16$ is below the ratio curve for $k=8$. This is because in these two cases, for the reordering method, all data server nodes (e.g., disk I/Os) become fully utilized and the throughput is almost independent of both k and n. By comparison, in the naive method, as there are not enough transaction aborts, the throughput for the $k=16$ case is higher than that for the $k=8$ case.

5.1.3 Data Ratio Test Results

In this section, we discuss the deadlock probability and throughput testing results from the data ratio test. Recall that in the data ratio test, we fixed $k=16$, $n=64$, $p=50\%$, and let the number of active parts s vary from 5,000 to 20,000. We show the deadlock probability of the naive method computed by the unified formula and measured in our tests in Figure 10. The two curves in Figure 10 roughly match. This indicates that our unified formula roughly reflects the real world situation. For the naive method, the deadlock probability increases linearly as s decreases.

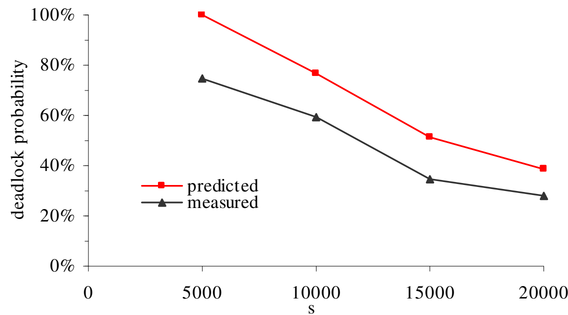


Figure 10. Deadlock probability of the naive method (data ratio test).

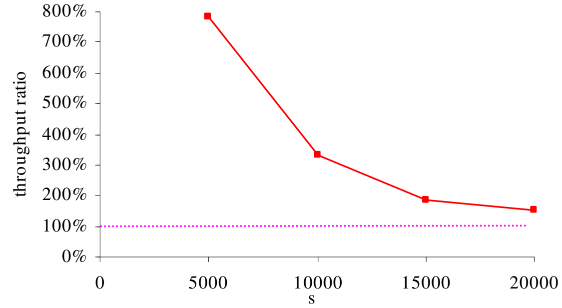


Figure 11. Throughput improvement gained by the reordering method (data ratio test).

We show the ratio of the throughput of the reordering method to that of the naive method in Figure 11. In all our testing cases, the naive method runs into the deadlock problem and the ratio is greater than 1. The smaller the number of active parts *s*, the more severe the deadlock problem of the naive method and the greater the ratio. That is, the smaller *s*, the greater performance advantages the reordering method exhibits over the naive method.

5.1.4 Transaction Ratio Test Results

In this section, we discuss the deadlock probability and throughput testing results from the transaction ratio test. Recall that in the transaction ratio test, we fixed $k=16$, $n=64$, $s=10,000$, and let p vary from 12.5% to 87.5%. We show the deadlock probability of the naive method computed by the unified formula and measured in our tests in Figure 12. The two curves in Figure 12 roughly match. This indicates that our unified formula roughly reflects the real world situation. For the naive method, the deadlock probability is proportional to $p(1-p)$. Note the value of $p(1-p)$ (and thus the deadlock probability of the naive method) is symmetric around $p=50\%$: it reaches the maximum when $p=50\%$ and keeps decreasing as p tends to either 0 or 1. This is easy to understand:

(1) In the extreme case when either $p=0$ or $p=1$, all modification operations are of the same type (either O_2 or O_1) and do not cause deadlocks.

(2) When $p=50\%$, the two kinds of modification operations (O_1 and O_2) have the same mixture ratio. This case causes the most deadlocks.

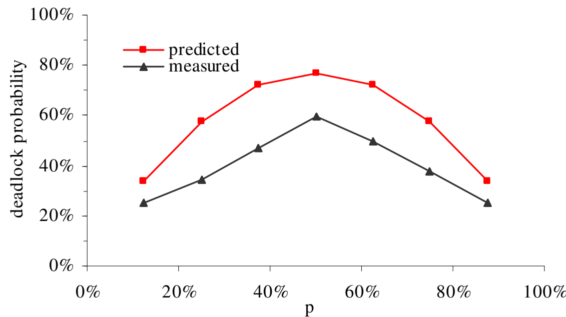


Figure 12. Deadlock probability of the naive method (transaction ratio test).

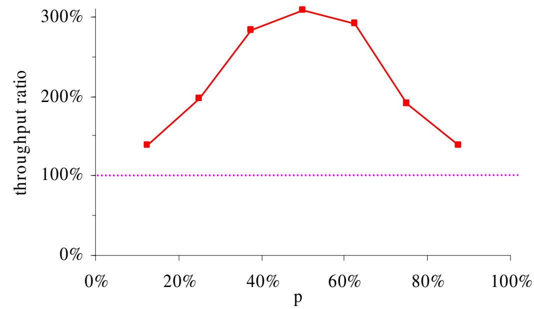


Figure 13. Throughput improvement gained by the reordering method (transaction ratio test).

We show the ratio of the throughput of the reordering method to that of the naive method in Figure 13. In all our testing cases, the naive method runs into the deadlock problem and the ratio is greater than 1. The closer *p* is to 50%, the more severe the deadlock problem of the naive method and the greater the ratio. That is, the closer *p* is to 50%, the greater performance advantages the reordering method exhibits over the naive method.

5.1.5 Comments

In our tests, we used no more than $k=16$ sessions. In a typical real world scenario, the number of sessions would be much larger than 16. Hence, we would expect the reordering method to perform better. (In a typical real world scenario, even if the number of active parts *s* could be larger than what we used in the tests, we would expect the effect coming from a large *s* to be compensated by a large *k*.) It would be desirable to test the cases with larger *k*'s. However, again due to software restrictions, we could not open more than $k=16$ sessions.

In our tests, we have two base relations and one join view in the database. In a typical real world scenario, there would be a large number of base relations and join views in the database. Base relations belonging to different join views are independent of each other. Hence, we would expect our conclusion to remain much the same if such independent base relations are added into the database. (In this case, we would interpret *k* as the number of concurrent load transactions on those dependent base relations, which should still be large enough to significantly deteriorate the performance of the naive method. This is because in a typical real world scenario, the total number of sessions would be quite large.) It would be desirable to test the cases with

more base relations and join views. Unfortunately, such an experiment was not possible in our testing environment due to the limited number of sessions.

5.2 Experiments in Teradata

In Teradata, we investigated the performance of the transaction reordering method when buffer pool analysis was considered to exploit synchronized scans.

5.2.1 Experiment Description

We created w relations R_i ($1 \le i \le w$) and a set S_t of other relations. All the relations R_i ($1 \le i \le w$) are of the same schema and contain the same number of tuples (and thus are of the same size), as shown in Table 2. w is an arbitrarily large number. Its specific value does not matter, as we only focus on the transaction throughput of the RDBMS.

Table 2. Test data set in Teradata.

	number of tuples	total size
R_i ($1 \le i \le w$)	8.4M	408MB

There are two kinds of transactions that we used for the testing:

(1) $\boldsymbol{T_i}$ ($1 \le i \le w$): Perform a full table scan on relation R_i. All the full table scans use the same query (except for the relation name). Such kind of full table scan queries are frequently encountered, as adhoc querying of real-time data warehouses is becoming increasingly common [17, 27, 33, 34].

(2) $\boldsymbol{U}$: Execute some query on the relations in S_t.

We evaluated the performance of the transaction reordering method and the baseline method in the following way:

(1) We tested the system configurations with four data server nodes ($L=4$) and eight data server nodes ($L=8$).

(2) We ran multiple long running U's so that most buffer pages in the buffer pool were committed. The remaining free buffer pages in the buffer pool only allowed the database to run y different T_i's.

(3) For each i ($1 \le i \le w$), we ran z T_i's. That is, we ran $w \times z$ T_i's in total.

(4) In the baseline method, we sent all the $w \times z$ T_i's to the database simultaneously (so that the original transaction sequence arriving at the RDBMS was in a random order).

(5) In the transaction reordering method, we used a centralized reorderer to reorder all the transactions.

5.2.2 Test Results

We measured the throughput of the $w \times z$ T_i's. The transaction throughput achieved by the transaction reordering method is shown in Figure 14. As long as $w>>z$, the transaction throughput of the baseline method does not depend on the specific value of z and is fairly close to that of the transaction reordering method in the $z=1$ case. This is because in this case, no matter how large z is, the probability that in the baseline method, the database runs multiple T_i's with the same i value concurrently is low. That is, the probability that the baseline method uses the synchronized scan technique is low.

When $z>1$, the transaction reordering method schedules the z T_i's with the same i value to run concurrently using the synchronized scan technique. Hence, the throughput of the transaction reordering method increases with z and becomes higher than that of the baseline method. When z becomes large enough (e.g., $z=8$), the CPU becomes the bottleneck. Because of this, the CPU speed approximately bounds the throughput achieved by the transaction reordering method. In more detail, as z increases, the throughput achieved by the transaction reordering method approaches a constant, where all CPUs are fully utilized. Since the 8-node configuration has twice the number of data server nodes than the 4-node configuration, and the sizes of the relation R_i's remain the same, the throughput of the transaction reordering method in the 8-node configuration case is close to twice that of the 4-node configuration.

The larger the y is, the more different T_i's compete for the disk I/O capability of Teradata. This will cause the disk heads to continuously oscillate among the different tracks where different R_i's are located. The effective disk I/O capability available for each T_i decreases as y increases. Hence, before all CPUs are fully utilized (i.e., when z is small), for a fixed z, the throughput achieved by the transaction reordering method decreases as y increases. However, when all CPUs become fully utilized (i.e., when z is large enough), the throughput achieved by the transaction reordering method approaches a constant that is independent of y, as that constant is almost solely determined by the CPU speed.

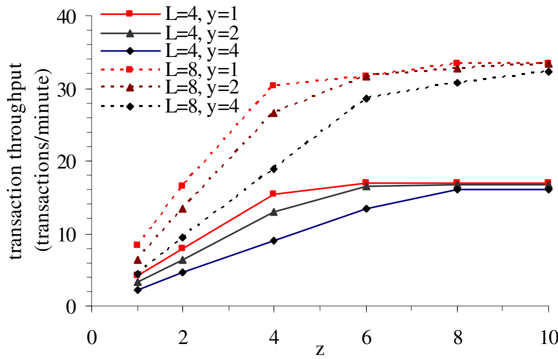


Figure 14. Throughput achieved by the transaction reordering method (with buffer pool analysis).

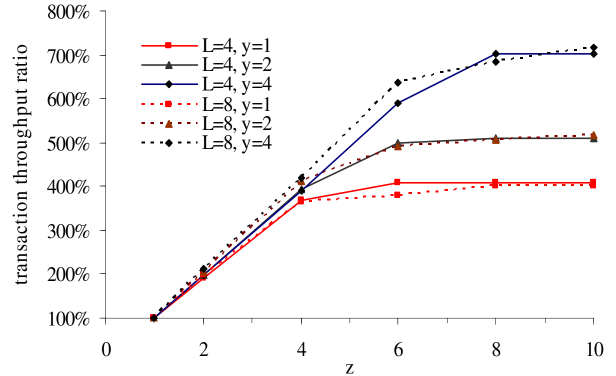


Figure 15. Throughput improvement gained by the transaction reordering method (with buffer pool analysis).

We show the ratio of the transaction throughput of the transaction reordering method to that of the baseline method in Figure 15. As explained above,

(1) The throughput of the transaction reordering method approaches a constant as *z* increases while the throughput of the baseline method is almost independent of *z*. Hence, the ratio approaches a constant as *z* increases.

(2) The throughput of the transaction reordering method (the baseline method) in the 8-node configuration case is close to twice that of the 4-node configuration. Hence, the ratio in the 8-node configuration is close to the ratio in the 4-node configuration.

(3) As *z* increases, the throughput achieved by the transaction reordering method approaches a constant that is independent of *y*. The throughput of the baseline method (which is close to the throughput achieved by the transaction reordering method in the $z=1$ case) decreases as *y* increases. Hence, as *y* increases, so does the constant that the ratio approaches as *z* increases.

In our testing, we never observed that once joined for synchronized scan, two scans drifted too far away from each other and got separated.

5.3 Experiments in Another RDBMS

To demonstrate the wide applicability of the transaction reordering method, we conducted experiments in the latest version of another commercial RDBMS from a major vendor and used the transaction ordering method with lock conflict analysis to address certain system dependent issue without changing the database

engine. This RDBMS uses a different concurrency control mechanism than Teradata. In this system, if multiple transactions run concurrently, each updating a base relation that has a materialized view defined on it, only one transaction can commit successfully while all the other transactions are aborted. This holds true no matter whether the base relations updated by these transactions are the same or not. It would be desirable to reorder transactions for this system so that at any time, at most one such transaction runs in the database updating a base relation that has a materialized view defined on it.

We analyzed the performance of the transaction reordering method in this RDBMS when lock conflicts were considered. We created a set S_1 of relations and another set S_2 of relations. Each relation in S_1 is a base relation of some materialized join view. Different relations in S_1 may have different materialized join views defined on them. No materialized view is defined on any relation in S_2. There are two kinds of transactions that we used for the testing:

(1) $\boldsymbol{T_1}$: Insert multiple tuples into some relation in S_1.

(2) $\boldsymbol{T_2}$: Execute some query/update on the relations in S_2. No T_2 conflicts with either a T_1 or another T_2.

We evaluated the performance of the transaction reordering method and the baseline method in the following way:

(1) We used a uni-processor database configuration. At any time, at most n transactions were allowed to run concurrently in the database (n is a large number whose specific value does not matter).

(2) We ran x transactions in total. x is an arbitrarily large number. Its specific value does not matter, as we only focus on the transaction throughput of the RDBMS. u% of these x transactions were T_1. The remaining $(100-u)$% of these x transactions were T_2. If some transaction aborted, we automatically re-executed it until it committed.

(3) In the baseline method, we sent all the transactions to the database simultaneously (so that the original transaction sequence arriving at the RDBMS was in a random order). In this case, as multiple T_1's may run concurrently in the database, some of them were aborted and re-executed. This decreased the transaction throughput of the RDBMS.

(4) In the transaction reordering method, we used a reorderer to reorder all the transactions so that at any time, at most one T_1 was running.

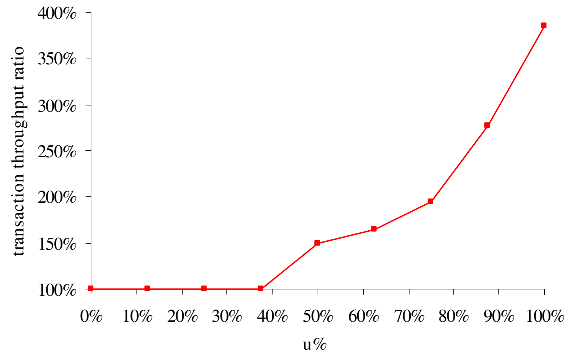


Figure 16. Throughput improvement gained by the transaction reordering method in another RDBMS (with lock conflict analysis).

We show the ratio of the transaction throughput of the transaction reordering method to that of the baseline method in Figure 16. In the baseline method, if multiple T_1's run concurrently, all but one of these T_1's are aborted and re-executed. The probability that multiple T_1's run concurrently increases with u. When u is small, such probability is small. In this case, almost no transaction is aborted. Even if a transaction gets aborted, its first-time execution has already fetched the necessary pages into memory. Re-executing the same transaction a second time is quick. Hence, the throughput of the transaction reordering method is the same as that of the baseline method. However, when u becomes large, the probability that multiple T_1's run concurrently also becomes large. This will cause a substantial percentage of the T_1's to get aborted and re-executed in the baseline method. Some of those re-executed T_1's may run concurrently with other (either first-time or re-executed) T_1's and get aborted and re-executed again. That is, in the baseline method, a T_1 may be aborted and re-executed multiple times before it is finally committed. The average number of times that a T_1 is aborted and re-executed increases with u. Hence, when u becomes large enough, the performance advantage of the transaction reordering method, i.e., the throughput ratio, becomes significant and keeps increasing with u. In the extreme case, when $u=100$ (i.e., when all the transactions are T_1), the throughput of the transaction reordering method is 3.85 times that of the baseline method.

6. Conclusions

This paper proposes the use of transaction reordering to improve the performance of an RDBMS. The general idea underlying transaction reordering is that by combining knowledge about the currently running

transactions and the transactions waiting to be run, a system can improve performance by selecting for running those transactions that fit best with those that are already running. In this paper we explored two different techniques, the first based upon reducing lock conflicts, the second upon increasing buffer pool hit rates. Our experiments with three commercial systems are promising, showing that this technique can significantly improve throughput for certain workloads.

For continuous data loading, there are several interesting directions that we intend to pursue in future work:

(1) We would like to give a fair comparison of the transaction response time in different methods: our reordering method, the naive immediate materialized view maintenance method (without reordering), and the deferred materialized view maintenance method. This is not a trivial task, since the overhead of load transactions in the deferred materialized view maintenance case (no materialized view maintenance is needed) is different from that in the immediate materialized view maintenance case. Also, the throughput of the naive immediate materialized view maintenance method (without reordering) is close to zero.

(2) As mentioned at the end of Section 3, reordering transactions may slightly delay the processing of some load transactions while speeding up the processing of other load transactions. It is a challenge to compare the data staleness caused by these slight delays with the data staleness caused by deferred materialized view maintenance. In fact, even giving a precise definition of data staleness that is comparable in both cases will be a challenge. For example, by some metrics the staleness of the reordering method will be zero (the transactions that are delayed will be balanced by those that are run early.) Also, it is difficult to define a standard base on which data staleness can be measured, since without reordering, the throughput of the naive immediate materialized view maintenance method is close to zero.

It is an open question whether immediate materialized view maintenance or deferred materialized view maintenance is more desirable. The answer to this question depends on how efficiently immediate materialized view maintenance can be done. Also, it depends on how well the semantic discrepancy between materialized views and base relations can be minimized for deferred materialized view maintenance. We hope that the techniques in this paper can contribute to the discussion in this regard.

Developing and exploring ways to define and detect which transactions fit best is another rich area for future work. Such future work can either seek to exploit intrinsic properties of sequences of transactions, or it

can seek to exploit performance problems that arise due to idiosyncrasies of specific commercial systems. Both approaches are interesting – as commercial RDBMSs continue to grow in complexity, the difficulty of making major changes to their functionality also grows, to the point where it is interesting in some cases to view them as artifacts to be studied rather than as programs to be modified. Transaction reordering research is one example of this approach.

Acknowledgements

We would like to thank Henry F. Korth for helpful discussions.

References

[1] M.J. Carey, S. Krishnamurthi, M. Livny, Load control for locking: the 'half-and-half' approach, PODS (1990) 72-84.

[2] C. Faloutsos, R.T. Ng, T.K. Sellis, Predictive load control for flexible buffer allocation, VLDB (1991) 265-274.

[3] R. Ramamurthy, D.J. DeWitt, Q. Su, A case for fractured mirrors, VLDB (2002) 430-441.

[4] A. Gupta, I.S. Mumick, Materialized Views: Techniques, Implementations, and Applications, MIT Press, 1999.

[5] P.M. Fernandez, Red brick warehouse: a read-mostly RDBMS for open SMP platforms, SIGMOD (1994) 492.

[6] Teradata parallel data pump reference, http://www.info.ncr.com/eDownload.cfm?itemid=023390001, 2002.

[7] A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Information and control in gray-box systems, SOSP (2001) 43-56.

[8] N.C. Burnett, J. Bent, and A.C. Arpaci-Dusseau et al., Exploiting gray-box knowledge of buffer-cache contents, USENIX (2002) 29-44.

[9] P. Roy, S. Seshadri, and S. Sudarshan et al., Efficient and extensible algorithms for multi query optimization, SIGMOD (2000) 249-260.

[10] T.K. Sellis, Multiple-query optimization, TODS 13(1) (1988) 23-52.

[11] G. Luo, J.F. Naughton, and C.J. Ellmann et al., Transaction reordering and grouping for continuous data loading, BIRTE (2006) 34-49.

[12] G. Luo, J.F. Naughton, and C.J. Ellmann et al., Transaction reordering with application to synchronized scans, DOLAP (2008) 17-24.

[13] P.A. Bernstein, D.W. Shipman, J.B. Rothnie, Concurrency control in a system for distributed databases (SDD-1), TODS 5(1) (1980) 18-51.

[14] H. Garcia-Molina, Using semantic knowledge for transaction processing in distributed database, TODS 8(2) (1983) 186-213.

[15] A. Dver, Real-time enterprise, Business week Dec. 2, 2002 issue.

[16] S. Brobst, J. Rarey, The five stages of an active data warehouse evolution, http://www.ncr.com/online_periodicals/brobst.pdf, 2001.

[17] G. Klaus, Real-time data warehousing and data mining for E-commerce, http://ids.csom.umn.edu/faculty/wanninger/lectures/DataMining-6204Sp00.html, 2000.

[18] A. Zimmerman, E. Nelson, In hour of peril, Americans moved to stock up on guns and TV sets, Wall Street Journal Newsletter, Sep. 18, 2001, http://www.swcollege.com/econ/street/html/sept01/sept18_2001.html.

[19] P.A. Bernstein, M. Hsu, B. Mann, Implementing recoverable requests using queues, SIGMOD (1990) 112-122.

[20] Oracle Streams, http://otn.oracle.com/products/dataint/htdocs/streams_fo.html, 2002.

[21] K. Pooloth, High performance inserts on DB2 UDB EEE using Java, http://www7b.boulder.ibm.com/dmdd/library/techarticle/0204pooloth/0204pooloth.html#overview, 2002.

[22] B. Babcock, S. Babu, and M. Datar et al., Models and issues in data stream systems, PODS (2002) 1-16.

[23] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, 1993.

[24] J. Gehrke, F. Korn, D. Srivastava, On computing correlated aggregates over continual data streams, SIGMOD (2001) 13-24.

[25] G. Graefe, M.J. Zwilling, Transaction support for indexed views, SIGMOD (2004) 323-334.

[26] Y. Zhuge, H. Garcia-Molina, J.L. Wiener, Multiple view consistency for data warehousing, ICDE (1997) 289-300.

[27] M. Poess, C. Floyd, New TPC benchmarks for decision support and Web commerce, SIGMOD Record 29(4) (2000) 64-71.

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

[29] Y. Zhuge, H. Garcia-Molina, J.L. Wiener, The strobe algorithms for multi-source warehouse consistency, PDIS (1996) 146-157.

[30] G. Luo, J.F. Naughton, and C.J. Ellmann et al., Locking protocols for materialized aggregate join views, TKDE 17(6) (2005) 796-807.

[31] Y. Zhao, P. Deshpande, and J.F. Naughton et al., Simultaneous optimization and evaluation of multiple dimensional queries, SIGMOD (1998) 271-282.

[32] R.D. Sloan, A practical implementation of the data base machine-Teradata DBC/1012, HICSS (1992) 320-327.

[33] C.A. Lang, B. Bhattacharjee, and T. Malkemus et al., Increasing buffer-locality for multiple relational table scans through grouping and throttling, ICDE (2007) 1136-1145.

[34] C.A. Lang, B. Bhattacharjee, and T. Malkemus et al., Increasing buffer-locality for multiple index based scans through intelligent placement and index scan speed control, VLDB (2007) 1298-1309.

[35] SQL Server 2005 books online, http://www.microsoft.com/technet/prodtechnol/sql/2005/downloads/books.mspx, 2007.

[36] H. Chou, D.J. DeWitt, An evaluation of buffer management strategies for relational database systems, VLDB (1985) 127-141.

[37] T. Johnson, D. Shasha, 2Q: a low overhead high performance buffer management replacement algorithm, VLDB (1994) 439-450.

[38] G.M. Sacco, M. Schkolnick, A mechanism for managing the buffer pool in a relational database system using the hot set model, VLDB (1982) 257-262.

[39] G. Luo, J.F. Naughton, and C.J. Ellmann et al., A comparison of three methods for join view maintenance in parallel RDBMS, ICDE (2003) 177-188.

Gang Luo received the BSc degree from Shanghai Jiaotong University, P.R. China, in 1998, and the PhD degree from the University of Wisconsin-Madison in 2004. He is currently a research staff member at IBM T.J. Watson Research Center. He has broad interests in various parts of relational database systems. Recently, he has been working on healthcare informatics and information retrieval.

Jeffrey F. Naughton earned a bachelor's degree in Mathematics from the University of Wisconsin-Madison, and a PhD in Computer Science from Stanford University. He served as a faculty member in the Computer Science Department of Princeton University before moving to the University of Wisconsin-Madison, where he is currently Professor of Computer Science. His research has focused on improving the performance and functionality of database management systems. He has published over 100 technical papers, and received the National Science Foundation's Presidential Young Investigator award in 1991, and was named an ACM Fellow in 2002.

Curt J. Ellmann earned a master’s degree in Computer Science from the University of Wisconsin-Madison. He works for the Division of Information Technology at the University of Wisconsin-Madison. His current focus is on "open source" software projects that the University is involved with. Prior to that, Curt was a manager for the optimizer group for the Teradata Database system, and the site manager for the Teradata software development lab in Madison, Wisconsin.

Michael W. Watzke received the bachelor’s degree in electrical engineering and computer science from the University of Wisconsin-Madison. He has worked in the database software and information technology consulting fields for twenty years. He is currently working for Teradata as a database architect focusing on the areas of application integration and performance.