RC24264 (W0705-106) May 22, 2007
Computer Science

IBM Research Report

Transaction Reordering with Application to Synchronized Scans

Gang Luo
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Jeffrey F. Naughton, Curt J. Ellmann
University of Wisconsin
Madison

Michael W. Watzke
Teradata

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Transaction Reordering with Application to Synchronized Scans

Gang Luo[1] Jeffrey F. Naughton[2] Curt J. Ellmann[3] Michael W. Watzke[3]
IBM T.J. Watson Research Center[1] University of Wisconsin-Madison[2] Teradata[3]
luog@us.ibm.com naughton@cs.wisc.edu ellmann@wisc.edu michael.watzke@ncr.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. An exception to this is the transaction reordering method, which reorders the transaction sequence submitted to the RDBMS and improves the transaction throughput by considering both the current system status and information about the interaction between queued and running transactions. The existing transaction reordering method only considers the reordering opportunities provided by analyzing the lock conflict information among multiple transactions. This significantly limits the applicability of the transaction reordering method. In this paper, we extend the existing transaction reordering method into a general transaction reordering framework that can incorporate various factors as the reordering criteria. We show that by analyzing the resource utilization information of transactions, the transaction reordering method can also improve the system throughput by increasing the resource sharing opportunities among multiple transactions. We provide a concrete example on synchronized scans and demonstrate the advantages of our method through experiments with a commercial parallel RDBMS.

1. Introduction

Traditional workload management methods mainly focus on the current system status [CKL90, FNS91]. 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 [CKL90]), 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, Luo et al. [LNE06] proposed the transaction reordering method for continuous data loading [Ter]. This workload management method uses information about the interaction between queued and running transactions to improve the throughput of an RDBMS by reordering the transactions before submitting them for execution. Although the general idea is interesting, the applicability of the existing transaction reordering method is limited, as that method only considers the reordering opportunities provided by analyzing the lock conflict information among multiple transactions.

In this paper, we extend the existing transaction reordering method into a general transaction reordering framework that can incorporate various factors as the reordering criteria for different applications. We show that the resource utilization information of transactions can provide another opportunity for the transaction reordering method to improve the throughput of an RDBMS. Our idea is to reorder transactions to increase the likelihood that they can share resources (e.g., sharing data in the buffer pool, or perhaps even sharing intermediate computations common to several transactions). As a concrete example, we show how to exploit synchronized scans [Fer94, LBM07] to reorder transactions so that buffer pool performance can be improved.

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 [RDS02]. Our transaction reordering method for exploiting synchronized scans can be regarded as a way to trade CPU cycles for disk I/Os. Our experiments in a commercial parallel RDBMS show that with minor overhead, our proposed 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 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 (as discussed in Luo et al. [LNE06]) 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 system dependent issue has never been discussed before and we show such an example here in a major commercial RDBMS.

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 [AA01, BBA02]. The database community has proposed multi-query optimization [RSS00, Sel88] 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.

The rest of this paper is organized as follows. Section 2 extends the existing transaction reordering method into a general transaction reordering framework. Section 3 discusses our transaction reordering method for exploiting synchronized scans. Section 4 investigates the performance of the transaction reordering method through an evaluation in two commercial RDBMSs. We conclude in Section 5.

2. General Transaction Reordering Framework

The transaction reordering method was originally proposed in Luo et al. [LNE06]. That method uses lock conflict analysis as the single reordering criterion and only works for continuous data loading [Ter]. Actually, transaction reordering is a general technique to improve RDBMS performance. It can be applied to multiple applications. In this section, we extend the existing transaction reordering method in Luo et al. [LNE06] into a general transaction reordering framework. This framework can easily take different factors into consideration as various reordering criteria. In Section 3, we show how to use buffer pool analysis as the reordering criterion to exploit synchronized scans. In our discussion, we assume that the strict two-phase locking protocol is used and all the transactions have the same priority.

The basic concept of transaction reordering is simple. 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 [CKL90, FNS91].

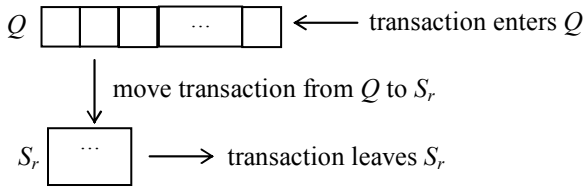


Figure 1. The general transaction reordering framework.

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.

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.

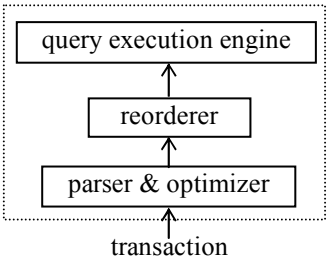
When we search for the desirable transaction, we are essentially looking for a transaction that is "compatible" with the running transactions in S_r. That is, we implicitly divide transactions into different types and only concurrently execute the transactions that are of "compatible" types. In Luo et al. [LNE06], this criterion is that the desirable transaction has no lock conflicts with the transactions in S_r. The idea of using transaction types to improve database performance has been investigated previously [BSR80, GM83]. 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 [BSR80], 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.

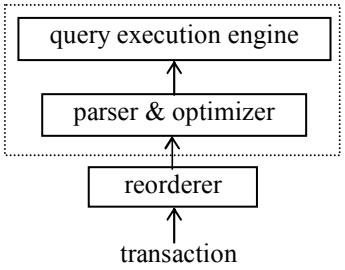
(2) The purpose of the classification in [GM83] is to allow non-serializable schedules which preserve consistency and which are acceptable to the users. In our transaction reordering method, we still preserve serializability. This is because we assume that the strict two-phase locking protocol is used and transaction reordering is done outside of the query execution engine.

During transaction reordering, we need to prevent transactions from starvation. We refer the reader to Luo et al. [LNE06] for a detailed discussion of the starvation prevention algorithm and its impact on transaction response time.

The above discussion assumes that transaction reordering is implemented inside the RDBMS. Transaction reordering can also be implemented 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 3 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 4.2 gives an example of the outside-RDBMS choice.



(1) reorderer resides inside the RDBMS



(2) reorderer resides outside the RDBMS

Figure 2. Transaction reordering architecture.

3. A Transaction Reordering Algorithm for Exploiting Synchronized Scans

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 [ZDN98] as a concrete example to illustrate our techniques. We first show in Section 3.1

that the existing buffer management methods cannot utilize the synchronized scan technique efficiently when the RDBMS is heavily loaded. Then in Section 3.2, we provide a solution to this problem using transaction reordering.

3.1 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 [Slo92]. As ad hoc querying of data warehouses is becoming more common [PF00, LBM07], 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 [ZDN98], Red Brick [Fer94], Microsoft SQL Server [SQL07], and IBM DB2 [LBM07]. 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.

As discussed below, 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. [LBM07] 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 [FNS91, JS94, CD85, SS82], 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.

3.2 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.

(1) **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, as shown in Figure 3. 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.

(2) **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 [CKL90], 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.

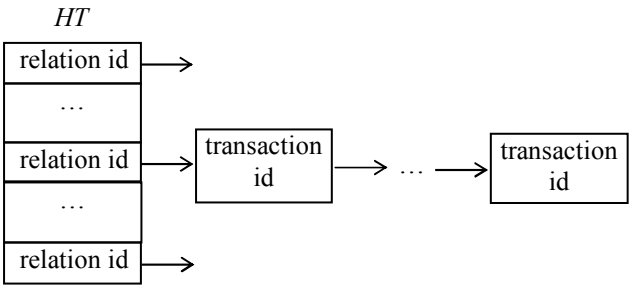


Figure 3. Hash table *HT*.

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 3.1, 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 [RDS02]. 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.

4. Performance Evaluation

In this section, we describe experiments that were performed in two commercial RDBMSs: Teradata and another commercial RDBMS. In Teradata, we investigated the performance of the transaction reordering method when buffer pool analysis was considered. In the other commercial RDBMS, we use the transaction ordering method with lock conflict analysis to address certain system dependent issue without changing the database engine.

In the first case, we focus on the throughput of a targeted class of transactions (i.e., transactions that 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.

4.1 Experiments in Teradata

We first conducted experiments in the commercial parallel RDBMS Teradata Release V2R5. We allocated a processor and a disk for each data server, so there were at most four data servers on each workstation.

We created w relations R_i ($1 \le i \le w$) and a set of other relations S_i. 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 1. w is an arbitrarily large number. Its specific value does not matter, as we only focus on the transaction throughput of the RDBMS.

Table 1. Test data set.

	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 [Kla, PF00, LBM07].

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

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.

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 4. 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.

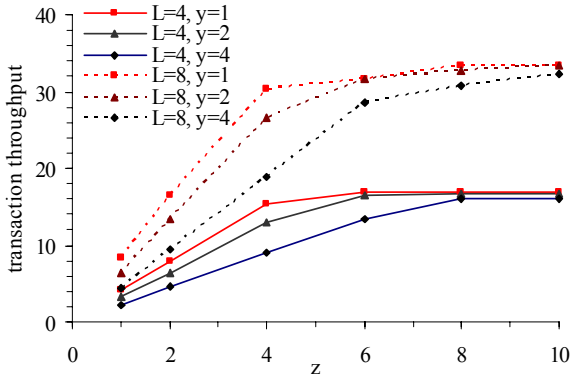


Figure 4. Throughput achieved by the transaction reordering method (with buffer pool analysis).

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.

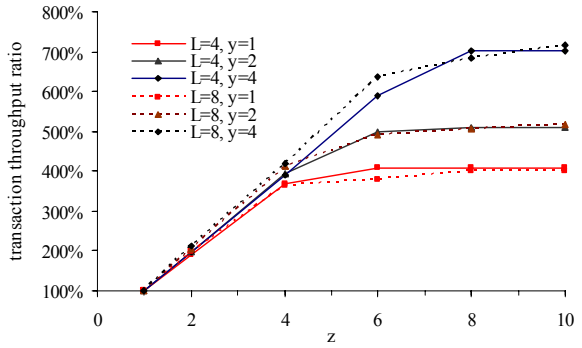


Figure 5. Throughput improvement gained by the transaction reordering method (with buffer pool analysis).

We show the ratio of the transaction throughput of the transaction reordering method to that of the baseline method in Figure 5. 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. This confirms our theory in Section 3.1.

4.2 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. 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 (see Luo et al. [LNE06] for details of the lock conflict analysis). We created a set of relations S_1 and another set of relations S_2. 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 are 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.

We show the ratio of the transaction throughput of the transaction reordering method to that of the baseline method in Figure 6. 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 [FRT92]. 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.

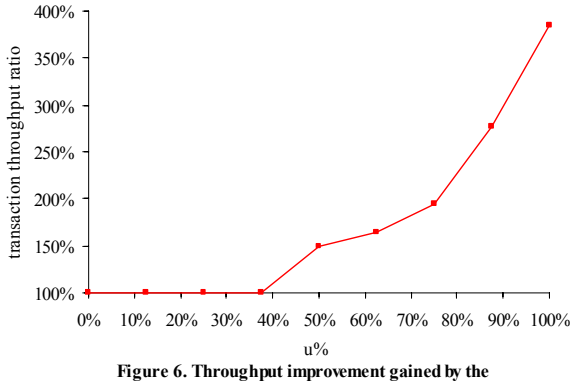


Figure 6. Throughput improvement gained by the transaction reordering method in another RDBMS (with lock conflict analysis).

5. Conclusion

This paper proposes a general transaction reordering framework to improve the performance of an RDBMS. The basic 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 increasing buffer pool hit rates, the second upon reducing concurrency control conflicts. Our experiments with two commercial systems are promising, showing that the transaction reordering method can significantly improve throughput for certain workloads.

Developing and exploring ways to define and detect which transactions "fit best" is a 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.

References

[AA01] A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. SOSP 2001: 43-56.

[BBA02] N.C. Burnett, J. Bent, and A.C. Arpaci-Dusseau et al. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. USENIX 2002: 29-44.

[BSR80] P.A. Bernstein, D.W. Shipman, J.B. Rothnie. Concurrency Control in a System for Distributed Databases (SDD-1). TODS 5(1): 18-51, 1980.

[CD85] H. Chou, D.J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. VLDB 1985: 127-141.

[CKL90] M.J. Carey, S. Krishnamurthi, and M. Livny. Load Control for Locking: The 'Half-and-Half' Approach. PODS 1990: 72-84.

[Fer94] P.M. Fernandez. Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms. SIGMOD Conf. 1994: 492.

[FNS91] C. Faloutsos, R.T. Ng, and T.K. Sellis. Predictive Load Control for Flexible Buffer Allocation. VLDB 1991: 265-274.

[FRT92] P.A. Franaszek, J.T. Robinson, and A. Thomasian. Concurrency Control for High Contention Environments. TODS 17(2): 304-345, 1992.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in Distributed Database. TODS 8(2): 186-213, 1983.

[JS94] T. Johnson, D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. VLDB 1994: 439-450.

[Kla] G. Klaus. Real-time Data Warehousing and Data Mining for E-Commerce. http://ids.csom.umn.edu /faculty/wanninger/lectures/DataMining-6204Sp00.html.

[LBM07] C. Lang, B. Bhattacharjee, and T. Malkemus et al. Increasing Buffer-Locality for Multiple

Relational Table Scans through Grouping and Throttling. ICDE 2007: 1136-1145.
[LNE06] G. Luo, J.F. Naughton, and C.J. Ellmann et al. Transaction Reordering and Grouping for Continuous Data Loading. BIRTE 2006: 34-49. Springer Lecture Notes in Computer Science 4365. Full version available as IBM research report RC24087.
[PF00] M. Poess, C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. SIGMOD Record 29(4): 64-71, 2000.
[RDS02] R. Ramamurthy, D.J. DeWitt, and Q. Su. A Case for Fractured Mirrors. VLDB 2002.
[RSS00] P. Roy, S. Seshadri, and S. Sudarshan et al. Efficient and Extensible Algorithms for Multi Query Optimization. SIGMOD Conf. 2000: 249-260.
[Sel88] T.K. Sellis. Multiple-Query Optimization. TODS 13(1): 23-52, 1988.
[Slo92] R.D. Sloan. A Practical Implementation of the Data Base Machine-Teradata DBC/1012. Hawaii Int. Conf. on System Sciences 1992: 320-327.
[SS82] 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.
[SQL07] SQL Server 2005 Books Online. http://www.microsoft.com/technet/prodtechnol/sql/2005/downloads/books.mspx, 2007.
[Ter] Teradata Parallel Data Pump Reference. http://www.info.ncr.com/temp/3021-122A94552.pdf.
[ZDN98] Y. Zhao, P. Deshpande, and J.F. Naughton et al. Simultaneous Optimization and Evaluation of Multiple Dimensional Queries. SIGMOD Conf. 1998: 271-282.