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, 19 Skyline Drive, Hawthorne, NY 10532, USA[1]
University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706, USA[2]
Teradata, 5752 Tokay Blvd Suite 400, Madison, WI 53719, USA[3]
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. 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.

Categories and Subject Descriptors
H.2.4 [Systems]: query processing, relational databases, H.2.7 [Database Administration]: data warehouse and repository
General Terms
Algorithms, Management, Measurement, Performance, Design, Experimentation
Keywords
Synchronized scans, transaction reordering, workload management, data warehouse

1. INTRODUCTION

Traditional workload management methods mainly focus on the current system status. 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), 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. [2] proposed the transaction reordering method for continuous data loading. 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 [1] 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. 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. [2]) 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

CIKM'08, October 26–30, 2008, Napa Valley, California, USA.
ACM 978-1-59593-991-3/08/10.

before and we show such an example here in a major commercial RDBMS.

2. GENERAL TRANSACTION REORDERING FRAMEWORK

The transaction reordering method was originally proposed in Luo et al. [2]. That method uses lock conflict analysis as the single reordering criterion and only works for continuous data loading. 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. [2] into a general transaction reordering framework. This framework can easily take different factors into consideration as various reordering criteria.

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. 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 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. [2], this criterion is that the desirable transaction has no lock conflicts with the transactions in S_r.

3. 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 [1] as a concrete example to illustrate our techniques.

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. To reduce the overhead of full table scans, people have developed the synchronized scan technique [1]. Its main idea 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.

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, after all 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.

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.

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 as many transactions in Q that does full table scan on relation R as the system permits to S_r for execution.

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, we run transaction T immediately so that it does not get blocked in Q.

In a typical scenario, most long-running transactions in the RDBMS are I/O-bound rather than CPU-bound. 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.

4. EXPERIMENTAL RESULTS

The experimental results are available in [3].

5. REFERENCES

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

[2] 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.

[3] G. Luo, J.F. Naughton, and C.J. Ellmann et al. Full version of this paper available as IBM research report RC24264 at pages.cs.wisc.edu/~gangluo/syncscan.pdf, 2008.