SURVEY ON TRANSACTION REORDERING

Sneha S. Menon  
Computer science and engineering  
Karunya University, Tamil Nadu  
E-Mail: sneha.menon86@gmail.com

Mrs. G. Hemalatha  
Assistant Professor, CSE Dept  
Karunya University, Tamil Nadu, India  
E-Mail: hema_latha207@yahoo.com

ABSTRACT

Transactions in database are logical unit of work containing one or more sql statements. Transactions can contain operations that might insert, update, delete, or select data. If several transactions try to access the same data at a time concurrency issues occur and which is when one of the transactions has to be aborted for the process to continue. Contention manager takes care of aborting the needed transactions. But with increase in number of aborts, performance and response time of the system decreases. Thus in order to minimize number of aborts, the transactions must be reordered. Reordering the transactions before submitting them to the execution unit would also increase the throughput of the system. This paper presents a survey on some of the transaction reordering done so far on the basis of resource conflict aborts and resource sharing.

Keywords- Transaction reordering, Resource sharing, Transaction aborts

1. INTRODUCTION

Transactions can contain one or more sql statements which would try to access same data at a time. If conflicting data accesses are detected between any two transactions, one of them is aborted and usually restarted immediately. The immediate restart of the aborted transaction might again conflict with the original transaction leading to repeated aborts. Increase in the number of aborts leads to performance degradation and also the response time of the system decreases. Thus the order in which transactions are
executed, if reordered, could minimize number of aborts. The users expect higher throughput from the RDBMS for supporting their applications. Reordering the transactions before submitting them to the execution unit would increase the throughput of the system. The research focus of Transaction Reordering is to improve transaction throughput in a relational database management system by increasing resource sharing and decreasing resource contention. In this paper we present a survey on some of the transaction reordering done so far on the basis of resource conflict aborts and resource sharing. Transaction reordering given by Mohammad Ansari[1], Fernando Pedone[2] and Gang Luo[3] are based on resource conflicts where [1] presents a technique called Steal on Abort which is used when there are aborts when two transaction of different threads access same data. Similarly [2] shows the reordering used in replicated database systems when aborts occur. While [3] gives the case of continuous data loading which leads to lock conflicts in the presence of materialized join views. The reordering methods given by Gang Luo, Jeffrey F. Naughton [4] and Gang Luo [5] talks about the resource sharing. [4] presents transaction reordering which does synchronized scans using buffer pool analysis as reordering criterion and [5] proposes a approach which makes use of both lock conflict analysis and buffer pool analysis as reordering criterion. [5] Combines the methods of [3] and [4].

2. RESOURCE CONFLICT ABORTS

2.1 Steal-on-abort

In [1], Mohammad Ansari and Mikel Lujan proposed that in Transactional Memory (TM) if any two concurrently executing transactions perform conflicting data accesses, one of them is aborted. Aborted transactions waste computing resources, and reduce performance. This paper presents a technique called steal-on- abort, which aims to improve transaction ordering at run-time which would minimize the number of aborts. The aborted transaction is stolen by the non-aborted transaction and queued behind it preventing the two transactions from conflicting again.

Transactional Memory (TM) is a parallel programming model which reduces programming effort, while improving execution performance. The TM system compares each transaction’s data accesses against all other transactions for conflicts, also known as
conflict detection or validation. Once a conflict between two transactions is observed, it is not executed concurrently again. Steal-on-abort does not restart the aborted transaction immediately; it gives the aborted transaction to the opponent transaction, and a new transaction is started in place of the aborted transaction.

![Figure 1 Steal-on-abort in action](image)

This strategy aims to reduce the amount of temporally local aborts, and expects that minimizing temporally local aborts minimizes the global number of aborts. During implementation, a number of threads concurrently execute transactions. If one transaction conflicts with another, it is removed from thread. Thread gets another transaction to execute. In Figure 1 Thread A is executing a transaction based on Job 2, and Thread B is executing a transaction based on Job 6. In step 1, thread A’s transaction conflicts with, and aborts, Thread B’s transaction. In step 2, thread A steals thread B’s job, and places it in its own stolen Deque. In step 3, after thread A finishes stealing, thread B gets a new job, and starts executing it immediately.

### 2.2 Transaction Reordering in Replicated Databases

In this paper Fernando Pedone, Rachid Guerraoui Andre Schiper [2] presents a fault-tolerant lazy replication protocol. This protocol enables local transaction execution and does not lead to any deadlock situation. It reduces the abort rate of transactions. Protocol first executes transactions locally, and then broadcasts a transactions certification message to all replication managers.

Replication increases availability, but designing an efficient distributed protocol that provides fault-tolerance and ensures replica consistency is a difficult issue. Lazy protocols enable local transaction processing (i.e., do not require any remote
communication during transaction execution), but before committing, the transactions have to be certified. A transaction $T_i$ is first executed locally at a replica server closest to client that initiates transactions. Then, using non-blocking atomic broadcast, a certification message for $T_i$ is broadcast to all replica servers. Every server that delivers $T_i$’s certification message executes a certification procedure, deciding for the commit or the abort of $T_i$. Certification procedure checks if $T_i$, can be serialized after committed transactions. If not, instead of aborting $T_i$, a reordering technique can be used to find out whether $T_i$ can be serialized somewhere before some committed transactions. If $T_i$ cannot be reordered, it is aborted. Figure 2 shows the states in which the transaction can be in.

![Figure 2 Transaction states](image)

The reordering technique is based on the fact that transaction serialization order does not need to be the same as the transaction certification order. By reordering a transaction to a position other than the current one, the protocol increases the possibilities of committing. A transaction $T_i$ may be reordered to any position between two transactions that executed concurrently with it but that have already committed. Let $\{T(l), T(l+1), \ldots, T(n)\}$ be concurrent transactions. Although $T_i$ reaches the certify state after transactions $\{T(l), T(l+1), \ldots, T(n)\}$ have committed, $T_i$ cannot read any value that was updated by transactions ordered before it and must not update items that are read after it.

### 2.3 Transaction Reordering and Grouping for Continuous Data Loading

In this paper Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke [3] have shown that in the presence of materialized join views, loading data
concurrently into multiple base relations of the same materialized join view can cause a severe deadlock problem. To solve this problem, reordering is done to the data to be loaded for execution so that at any time, for any materialized join view, data is only loaded into one of its base relations. System uses high concurrency locking protocol. Since data comes from multiple data sources in the form of modification operations (insert, delete, or update), load utility opens multiple sessions which increases the concurrency. This is called partitioning method. Partitioning method allows the modification to same relation through same session. Also, for load transactions on the relations that contain “aggregate” attributes, pre-aggregation is used to reduce the number of SQL statements in the load transactions.

2.3.1 The Pre-aggregation Method

A large number of data warehouses have relations with certain attributes representing aggregate information. Updates to these relations increment or decrement the aggregate attribute values. When we load data continuously into these relations, we combine multiple modification operations into a single load transaction. This creates an opportunity for optimization: by pre-aggregation, we can reduce the number of SQL statements in the load transactions on these relations. Suppose the following two modification operations $O_1$ and $O_2$ are combined into a single load transaction $T$:

1. $O_1$: update $R$ set $R.b=R.b+b_1$ where $R.a=v$;
2. $O_2$: update $R$ set $R.b=R.b+b_2$ where $R.a=v$;

If we let $b_3=b_1+b_2$, then transaction $T$ can be transformed into an equivalent transaction $T'$ that contains only a single SQL statement:

update $R$ set $R.b=R.b+b_3$ where $R.a=v$;

Compared to transaction $T$, transaction $T'$ saves one SQL statement. Hence, transaction $T'$ is more efficient. Advantage of this technique is that the transaction execution/response time is reduced. This may further improve database concurrency, as the period that transactions hold lock is reduced.
3. RESOURCE SHARING

3.1 Transaction Reordering with Application to Synchronized Scan

The transaction reordering method is turned into a general transaction reordering framework by Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke in [4] that can incorporate various factors as the reordering criteria. With 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.

Main idea is that if two transactions are scanning the same relation, then we can group them together so that I/O can be shared between them. Example: Relation R contains K1 pages. When we start full table scan, we add some information in in-memory data structure. When T2 starts, it searches in data structure if any other transaction is scanning same relation. If yes then it shares those pages with T1.

3.2 Transaction Reordering

In this paper Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke[5] have shown that throughput of an RDBMS that is processing a sequence of transactions can be improved by reordering the transactions before submitting them for execution. In this paper, two factors are considered: lock conflicts and buffer pool performance which the Figure 3 shows.

![Figure 3 Transaction reordering classification](Image)
3.2.1 Using Lock Conflict Analysis as the Reordering Criterion

Continuous load utilities make the following assumptions for continuous data loading:

1) The RDBMS is running with standard ACID properties for transactions.
2) The RDBMS neither imposes nor assumes any particular order for these load transactions.
3) The RDBMS has no requirement on whether multiple modification operations can or cannot commit/abort together.

To increase concurrency, a continuous load utility typically opens multiple sessions to the RDBMS. Continuous load utilities usually combine multiple modification operations into a single transaction to avoid per transaction overhead.

3.2.2 Deadlock Probability

In the presence of materialized join views, the probability of deadlock can easily be very high. Suppose the following cases stand true:

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 the probability that any particular transaction deadlocks is approximately

\[ p \frac{(1-p)(k-1)n^2}{2s} \]

Rules for reordering:

(i) At any time, for any join view JV, data can only be loaded into one base relation of JV.
(ii) Modification operations on the same base relation use the partitioning method
(iii) The system uses a high concurrency locking protocol.
3.2.3 Using Buffer Pool Analysis as the Reordering Criterion

Main idea is that if two transactions are scanning the same relation, then we can group them together so that I/O can be shared between them. Two techniques are used:

**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 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 $Sr$ 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 $Sr$ is currently doing a full table scan on relation $R$. If so, we run transaction $T$ immediately so that it does not get blocked in $Q$.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Steal-on Abort</th>
<th>Transaction Reordering in Replicated Databases</th>
<th>Transaction Reordering and Grouping for continuous Data Loading</th>
<th>Transaction Reordering with Application to Synchronized scan</th>
<th>Transaction Reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock conflict Analysis</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Resource sharing</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Cost</td>
<td>High</td>
<td>Very high</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Moderate</td>
</tr>
<tr>
<td>System</td>
<td>Standalone</td>
<td>Replicated</td>
<td>Standalone</td>
<td>Standalone</td>
<td>Standalone</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Overhead</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Throughput</td>
<td>Increased</td>
<td>Increased</td>
<td>Increased</td>
<td>Increased</td>
<td>Increased</td>
</tr>
<tr>
<td>Average response Time</td>
<td>Improved</td>
<td>Improved</td>
<td>Improved</td>
<td>Improved</td>
<td>Improved</td>
</tr>
</tbody>
</table>

4. ANALYSIS

The Table1 above shows the comparison of the various transaction reordering done so far. The analysis is on the basis of some of the parameters like throughput, average response time, overhead incurred and cost. The analysis also shows whether
system does lock conflict analysis, allows resource sharing and is dynamic or not. Type of system is also taken as one of the parameters.

5. CONCLUSION

This paper has presented a survey on some of the transaction reordering done so far on the basis of resource conflict aborts and resource sharing. A study of runtime approaches, which dynamically re-orders the transactions with the aim of reducing the number of aborted transactions has been done. We also came to know that by knowing the currently executing transaction and the ones waiting for execution, transaction reordering can be done such that the resource sharing is achieved while reducing the lock conflicts. In future work we would like to explore ways to detect transactions that are appropriate for reordering and also would like to analyse the delays caused due to these reordering.

REFERENCES