An Advanced Approach of Local Counter Synchronization to Timestamp Ordering Algorithm in Distributed Concurrency Control

Loc Nguyen

University of Science, Ho Chi Minh City, Vietnam
Email: ng_phloc@yahoo.com

Received 17 September 2015; accepted 4 October 2015; published 9 October 2015

Copyright © 2015 by author and OALib.
This work is licensed under the Creative Commons Attribution International License (CC BY).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

Concurrency control is the problem that database management system (DBMS) meets with difficulties, especially distributed DBMS. There are two main methods of concurrency control such as locking-based and timestamp-based. Each method gets involved in its own disadvantages, but locking-based approach is often realized in most distributed DBMS because its feasibility and strictness lessen danger in distributed environment. Otherwise, timestamp ordering algorithm is merely implemented in central DBMS due to the issue of local counter synchronization among sites in distributed environment. The common solution is broadcasting the message about the change of local counter (at one site) over distributed network so that all remaining sites "know" to update their own counters. However, this solution raises some disadvantages of low performance. So, an advanced approach is proposed to overcome such disadvantages by giving out another measure so-called active number that is responsible for harmonizing local counters among distributed sites. Moreover, another method is proposed to apply minimum spanning tree into reducing cost of broadcasting messages over distributed network.

Keywords

Distributed Concurrency Control, Timestamp Ordering Algorithm, Local Counter Synchronization

Subject Areas: **Distributed Computing, Information and Communication Theory and Algorithms**

1. Introduction

Concurrency control ensures the consistence and reliability attribute of transaction. Before discussing the main

How to cite this paper: Nguyen, L. (2015) An Advanced Approach of Local Counter Synchronization to Timestamp Ordering Algorithm in Distributed Concurrency Control. *Open Access Library Journal*, **2**: e982.
http://dx.doi.org/10.4236/oalib.1100982

topic, we should glance over some importance definitions. Note that the research is a proposed approach for improving a well-known algorithm. So, definitions and methods are mentioned in short; please see [1] for more details about distributed concurrency control. The schedule S is the set of transactions: $S = \{T_1, T_2, \cdots, T_n\}$; each transaction T_i has data operations O_{ij} (read or write). Two operations accessing the same data item x are in conflict if one of them is a write.

A complete schedule S_T^C is defined as a partial order $S_T^C = \{\Sigma_T, \prec_T\}$ where [1]:

1) The set $\Sigma_T = \bigcup_{i=1}^{n} \Sigma_i$ with note that Σ_i is the set of data operations O_{ij} in transaction T_i.

2) The relationship $\prec_T = \bigcup_{i=1}^{n} \prec_i$ with note that $\prec_i$ denotes the partial order relationship between two arbitrary operations $O_{ij}(x)$ and $O_{kl}(x)$ in transaction T_i.

3) Two arbitrary operations $O_{ij}(x)$ và $O_{kl}(x)$ in Σ_T are conflicting if either $O_{ij}(x) \prec_T O_{kl}(x)$ or $O_{kl}(x) \prec_T O_{ij}(x)$.

All concurrency control algorithms are based on serializability theory [1]. First, *serial scheduler* is the scheduler in which transactions are executed step-by-step, meaning that all operations of a transaction must be completed before starting a new transaction. Schedules S_1 and S_2 defined over the same set of transactions are conflict-equivalent [1] if for any pair of conflicting operations $O_{ij}(x)$ and $O_{kl}(x)$, $i \neq j$, whenever $O_{ij}(x) \prec_1 O_{kl}(x)$, and then $O_{ij}(x) \prec_2 O_{kl}(x)$.

A schedule is serializable if it is conflict-equivalent to a serial schedule and the serial execution is called serialization order [1]. All concurrency control algorithms focus on generating the serializable schedule to archive two goals:

- The order of executing transactions is a serialization order which ensures the sound result of transactions execution.
- Taking advantage of parallel processing.

There are two basic methods of concurrency control such as locking-based and timestamp-based. Both of them have strong and weak points but timestamp-based method is the key in this paper because this method does not meet deadlock problem and it is more effective than locking-based method in some situations. Now some main aspects about concurrency control methods are already introduced but it is very difficult to "move" such methods from non-distributed database to distributed database. Of course, locking-based method is modified to be conformable to distributed environment but I propose some advancement for timestamp-based method in distributed environment. Timestamp-based algorithm is discussed in Section 2 and its improvement is proposed in Section 3. Section 4 is the conclusion.

2. Timestamp Ordering Algorithm

Unlike locking-based approach which maintains *serializability* by mutual exclusion, timestamp-based concurrency control algorithm tries to uphold serializability by [1]:

- Assigning a unique timestamp to each transaction.
- Ordering transactions upon their timestamps. Consequently, there is a serializable order of transactions.
- Performing transactions according to this order.

Timestamp is an identifier of every transaction and used to permit ordering. Timestamp has two essential attributes: uniqueness and monotonicity. Monotonicity refers that timestamps generated by the same transaction manager (TM) are monotonically increased in values. This attribute permits to distinguish timestamp and transaction identifier.

There are two ways to generate timestamp:

- Using a monotonically increasing global counter, but it is very difficult to maintain global counter in distributed environment.
- Every site is free to assign timestamps by itself, upon the local counter. To keep the uniqueness, each site attaches its identifier to the local counter. Therefore, timestamp is in the two-part forms <*local-counter-value*, *site-identifier*> or <*local-system-clock*, *site-identifier*> [1]. Site identifier is put in the least significant position to avoid the situation that one timestamp from a site will be always larger or smaller than from another site.

When transactions have already been assigned timestamp, ordering the operations (read, write) of transactions must obey the timestamp ordering (TO) rule "Given two conflicting operations $O_{ij}(x)$ and $O_{kl}(x)$ belonging to transactions T_i and T_k respectively, $O_{ij}(x)$ is executed before $O_{kl}(x)$ if timestamp of T_i is smaller than timestamp of T_k" [1]. It means that the older transaction is executed first.

In general, TO algorithm through two main steps is described briefly below [1]:

1) The TM receives transactions coming from application and assigns timestamp values to them. After that, TM deliver operations (read, write) O_{ij} of these transactions to the scheduler (SC).

2) SC schedules data operations by TO rules. It checks each new operation against conflicting operations that have been scheduled. If the new operation belongs to a younger (later) transaction than all conflicting ones, then accept it; otherwise reject it and *restart* the entire transaction with a new timestamp.

It can be asserted that TO algorithm maintains the execution order according to the timestamps order.

3. An Advanced Approach of Local Counter Synchronization

Suppose that timestamp is generated upon local counter at each site, there is a problem of local counter synchronization. Note that SC will restart transaction if there is another younger transaction scheduled. Such situation occurs many times if there are sites which are not active or not receives any transaction in regular period. In this case, the local counters of such sites are so smaller than other sites and most transactions coming to these sites will be restarted many times until their counters are approximate to local counters of other sites. Hereafter, **Figure 1** [1] depicts an example about how to maintain timestamps among sites.

Transaction T at site 2 try to read the data item x which exists at site 1 and has read and write timestamps: $rts(x)$ and $wts(x)$. If timestamp of T denoted $ts(T)$ is smaller than $wts(x)$, the read is rejected and T is restarted and gets a new larger timestamp from counter at site 2. When counter at site 2 is too small, T has no chance to be executed.

It is necessary to synchronize local counter at all sites. Whenever TM at one site increases its own counter due to accepting transaction, it broadcasts a message about updating counter over all TM (s) at other sites. After that, other TM (s) will compare their local counters to coming counter and adjust themselves their counter one more than coming counter (= coming counter + 1). This is the mechanism of broadcasting information about updating local counters. It ensures that there is no local counter which runs too fast or too slowly. Two improvements of counter synchronization are proposed as follows:

- Improving the way of broadcasting message of updating local counters.
- Reducing expense of data transmission in broadcasting message of updating local counter

3.1. Improving the Way of Broadcasting Message of Updating Local Counters

To broadcast message of updating local counter on distributed network is the most considered problem in counter synchronization. Given transaction manager TM_i at site i increases its own counter when receiving a new transaction. So TM_i has the new counter *counter* $(TM_i) = 200$ and it sends three messages namely m_1, m_2 and m_3 to TM_1, TM_2 and TM_3, respectively.

- TM_1 has old local counter *counter* $(TM_1) = 1$.
- TM_2 has old local counter *counter* $(TM_2) = 150$.
- TM_3 has old local counter *counter* $(TM_3) = 170$.

Because *counter* (TM_i) is so larger than *counter* (TM_1), *counter* (TM_1) get the new value 201 = *counter* (TM_i) + 1. In usual way, both remaining *counter* (TM_2), *counter* (TM_3) are also re-assigned new value 201. At that time, *counter* (TM_i) is considered as the "milestone". Yet, in situation that TM_2 and TM_3 are active with note that the term "active" implicates that TM received more transactions. If we choose *counter* (T_i) as the "milestone" to increase counters of TM_2 and TM_3 then the frequency of rejecting transactions at other sites become high because TM_2 and TM_3 receive more and more transactions and their locks increase faster and faster. In consequences, the

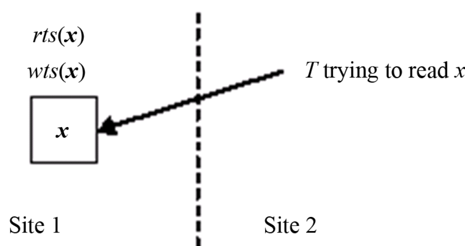

Figure 1. Maintain timestamps among sites [1].

timestamps of new transactions at site 2, 3 get so high values and so $rts(x_k)$ and $wts(x_k)$ of data item x_k at site 2, 3 raises suddenly. Transactions (with low timestamps at other sites) accessing x_k will be rejected more constantly. Of course the performance of distributed database management system is decreased significantly. However, it is nearly impossible to ignore the role "milestone" of *counter* (TM_i) because this will lead to miss results in other situations in which TM_2 and TM_3 are not active. What is the solution in this situation?

I propose that the timestamp is represented in three-part form <*site-identifier*, *local-counter*, *active-number*>. The positions of parts are corresponding to their importance, e.g. local-counter in the second position is more significant than active-number in the third position. Thus, each TM_i at site i has the new *active-number* which is responsible for measuring the degree of activeness at site i. The more transactions site i is received, the higher the active-number is and so the active-number of given site can be updated by frequency of transactions coming to such site. Suppose sites namely TM_1, TM_2 and TM_3 update their own local counters in situation that TM_i has the new *counter* (TM_i) and sends three messagesm_1, m_2, m_3 to TM_1, TM_2, TM_3 respectively, my proposal has two following steps given two integer parameters $\alpha > 0$ and $\beta > 0$:

1) If $\left|counter\left(TM_j\right) - counter\left(TM_i\right)\right| > \alpha$ then $counter\left(TM_j\right) = counter\left(TM_i\right) + 1$ where $j = 1, 2, 3$.

2) If $\left|counter\left(TM_j\right) - counter\left(TM_i\right)\right| \le \alpha$ then *active-number* (TM_j) is considered. If *active-number* (TM_j) < β then *counter* (TM_j) = *counter* (TM_i) + 1. Otherwise, *counter* (TM_j) is kept in original (not changed).

For example, we have *counter* (TM_i) = 200, *counter* (TM_1) = 1, *counter* (TM_2) = 150, *counter* (TM_3) = 170, *active-number* (TM_1) = 10, *active-number* (TM_2) = 60, *active-number* (TM_3) = 100, $\alpha = 50$ and $\beta = 80$. Applying above algorithm, we have:

- The *counter* (TM_1) = *counter* (TM_i) + 1 = 201 because $\left|counter\left(TM_1\right) - counter\left(TM_i\right)\right| = 199 > \alpha$.
- The *counter* (TM_2) = *counter* (TM_i) + 1 = 201 because *active-number* (TM_2) = 60 < β.
- The *counter* (TM_3) = 170 (not changed) because $\left|counter\left(TM_3\right) - counter\left(TM_i\right)\right| = 30 < \alpha$ and *active-number* (TM_3) = 100 > β.

Finally, the increasing rate of local counter at a site will be reduced if that site is active. In other words, that lock at active site is increased is delayed. It is easy to infer that the purpose of using active number is to harmonize an increase in local counters. In the case that the active-number of given site is lost, it is reset to be zero. Note that the triplet for timestamp <*site-identifier*, *local-counter*, *active-number*> is identified by only local-counter and site-identifier. Active-number is an auxiliary part. When transaction restarts, active-number is also reset to be zero.

3.2. Reducing Expense of Data Transmission in Broadcasting Message

Suppose all sites in distributed database compose the network including nodes which are respective to TM (s). Whenever one node adjusts its own counter, it issues the message all over the network. So this consumes much system resource. How to reduce the expense of message transmission but ensure that the message has to coming to every node?

We consider TM network as the graph in which every node is a TM and every edge is attached to the weight which represents the expense of the transmission from source TM node to destination TM node. It is not difficult to find the most minimum spanning tree [2] of such graph by applying some algorithms such as PRIM and Krussal [2]. Finally, message is broadcasted over this tree. So, all TM nodes are received message with lowest transmission expense. **Figure 2** depicts TM graph and minimum spanning tree drawn as bold line.

4. Conclusions

Local counter synchronization is the essence of timestamp-based method in concurrency control. In this paper, two improvements on this synchronization are proposed as follows:

- The timestamp is represented in three-part form <*site-identifier*, *local-counter*, *active-number*>. Whether local counters are updated depends on active numbers and two other parameters α and β. The ideology is to delay the increase of local counters in sites until the condition composed of active numbers, α and β, is met fully so that the increasing rate of local counter at a site will be reduced if that site is active.
- Reducing expense of message transmission by considering TM network as the graph and broadcasting the message over the minimum spanning tree of such graph.

Of course, my technique doesn't provide the thorough solution to the drawback of low performance when

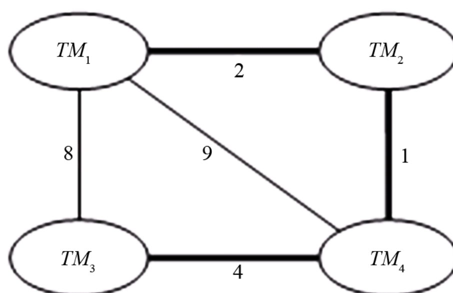

Figure 2. TM graph and minimum spanning tree (bold line).

broadcasting message over network. It only improves timestamp-based method. In general, this article is a work-in-progress research for improving well-known timestamp ordering algorithm. If compared with existing approaches, it requires a lot of evaluations which go beyond a proposal.

References

[1] Ozsu, M.T. and Valduriez, P. (2011) Principles of Distributed Database Systems. 3rd Edition, Springer, Berlin.

[2] Wikipeadia (2014) Minimum Spanning Tree. http://en.wikipedia.org/wiki/Minimum_spanning_tree