Performance of Distributed Optimistic Concurrency Control in Real-Time databases

Jan Lindström

University of Helsinki, Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki,Finland
jan.lindstrom@cs.Helsinki.FI

Abstract. Concurrency control is one of the main issues in the studies of real-time database systems. In this paper different distributed concurrency control methods are studied and evaluated in real-time system environment. Because optimistic concurrency control is promising candidate for real-time database systems, distributed optimistic concurrency control methods are discussed more detailed way. We propose a new distributed optimistic concurrency control method, demonstrate that proposed method produces a correct results and proposed method is evaluated and tested in prototype implementation of real-time database system for telecommunications. ...

1 Introduction

Numerous real-word applications contain time-constrained access to data as well as access to data that has temporal validity. For example consider telephone switching system, network management, navigation systems, stock trading, and command and control systems. Moreover consider the following tasks within these environments: looking up the "800 directory", obstacle detection and avoidance, radar tracking and recognition of objects. All of these contains gathering data from the environment, processing of information in the context of information obtained in the past, and contributing *timely* response. Another characteristic of these examples is that they contain processing both temporal data, which loses its validity after a certain time intervals, as well as historical data.

Traditional databases, hereafter referred as databases, deal with persistent data. Transactions access this data while maintaining its consistency. The goal of transaction and query processing in databases is to get a good throughput or response time. In contrast, *real-time systems*, can also deal with temporal data, i.e., data that becomes outdated after a certain time. Due to the temporal character of the data and the response-time requirements forced by the environment, task in real-time systems have time constraints, e.g., periods or deadlines. The important difference is that the goal of real-time systems is to meet the time constraints of the tasks.

Concurrency control is one of the main issues in the studies of real-time database systems. With a strict consistency requirement defined by serializability [4], most real-time concurrency control schemes considered in the literature

are based on two-phase locking (2PL) [6]. 2PL has been studied extensively in traditional database systems and is being widely used in commercial databases. In recent years, various real-time concurrency control protocols have been proposed for single-site RTDBS by modifying 2PL (e.g. [10, 1, 14]).

However, 2PL has some inherent problems such as the possibility of deadlocks as well as long and unpredictable blocking times. These problems appear to be serious in real-time transaction processing since real-time transactions need to meet their timing constraints, in addition to consistency requirements [17].

Optimistic concurrency control protocols [11, 7] have the nice properties of being non-blocking and deadlock-free. These properties make them especially attractive for real-time database systems. Because conflict resolution between the transactions is delayed until a transaction is near to its completion, there will be more information available in making the conflict resolution. Although optimistic approaches have been shown to be better than locking protocols for RTDBSs [9, 8] , they have the problem of unnecessary restarts and heavy restart overhead. This is due to the late conflict detection, that increases the restart overhead since some near-to-complete transactions have to be restarted. Therefore in recent years numerous optimistic concurrency control algorithms have been proposed for real-time databases (e.g. [5, 12, 13]).

Telecommunication is an example of an application area, which has database requirements that require a real-time database or at least time-cognizant database. A telecommunication database, especially one designed for IN services [2], must support access times less than 50 milliseconds. Most database requests are simple reads, which access few items and return some value based on the content in the database.

This paper is organized as follows. Different distributed concurrency control techniques proposed in literature are presented in Section 2. We will propose a new distributed optimistic concurrency control method which is presented in Section 3. Evaluation of the proposed method is presented in Section 4. Finally, Section 5 concludes this study.

2 Distributed Concurrency Control Techniques

In this section we present basic concurrency control techniques and some results of their complexity. Thus we present different distributed schedulers. There are three basic schedulers which allow transactions to execute safely concurrently [3]:

1. Locking methods
2. Timestamp methods
3. Optimistic methods.

These methods have been mainly developed for centralized DBMS and then extended for the distributed case.

2.1 Distributed Optimistic Method

Optimistic Concurrency Control (OCC) [7, 11] is based on the assumption that a conflict is rare, and that it is more efficient to allow transactions to proceed without delays to ensure serializability. When a transaction wishes to commit, a check is performed to determine whether conflict has occurred. There are three phases to an optimistic concurrency control protocol:

- *Read phase*: The transaction reads the values of all data items it needs from the database and stores them in local variables. Updates are applied to a local copy of the data and announced to database system by operation named *pre-write*.
- *Validation phase*: The validation phase ensures that all the committed transactions have executed in a serializable fashion. For read-only transaction, this consists of checking that the data values read are still the current values for the corresponding data items. For a transaction that contains updates, validation consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained.
- *Write phase*: This follows the successful validation phase for update transactions. During the write phase, all changes made by the transaction are permanently stored into the database.

There are several ways to extend optimistic method to distributed case. One of the easiest is to use tickets. Others are based on optimistic locking, hybrid methods and backward validation.

Concurrency control method requires certain information in order to find and resolve conflicts. This information must be gathered from the data and from the transactions. This information is read and manipulated when some transaction arrives into system, validates or commits.

Every data item in the real-time database consists the current state of object (i.e. current value stored in that data item), and two timestamps. These timestamps represent when this data item was last committed transaction accessed. These timestamp are used in concurrency control method to ensure that transaction reads only from committed transactions and write after latest committed write.

There are certain problems that arise when using optimistic concurrency methods in distributed systems [18]. It is essential that the validation and the write phases are in one critical section. These operations do not need to be executed in one phase. It is sufficient to guarantee, that no other validating transaction uses same data items before earlier validated transaction has wrote them.

- **Problem 1**: *Preserving the atomicity of validating and write phases* [18]. One has to find a mechanism to guarantee that the validate-write critical section is atomic also for global transactions.
- **Problem 2**: *The validation of subtransactions is made purely on local basis* [18]. In the global validation phase, we are interested only in the order

between global transactions. However, the order between distributed transactions may result from indirect conflicts, which are not visible to the global serializability mechanism. Used method must be able to detect also these indirect conflicts. These indirect conflicts are caused by local transactions which access same data items as global transactions.

- **Problem 3**: *Conflicts that are not detectable at the validation phase.* Transaction may be non-existent in the system, active or validated. A conflict is always detected between two active transactions. Combining the phases of two transactions we can find three different combinations of states which describe different conflict detection situations.
  1. Both transactions are active during the first validation.
  2. Both transaction were active at the same time, but the conflict occurred after the first validation. This case means that remaining active transaction made read or prewrite operation to data item after validation of the other transaction.
  3. Transactions execute serially. Because serial execution is correct, this case is not a problem.

Because optimistic concurrency control is main research area of this paper we will present more detailed discussion in the next section.

3 Proposed Distributed Optimistic Concurrency Control Method

In this section we propose a new distributed optimistic concurrency control method DOCC-DATI (Distributed Optimistic Concurrency Control with Dynamic Adjustment of the Serialization order using Timestamp Intervals). This method is based OCC-DATI protocol [16]. We have added new features to OCC-DATI to achieve distributed serializability and to solve problems of the distributed optimistic concurrency control methods presented in section 2. Commit protocol is based on 2PC, but 3PC could be also used.

Every site contains directory where all objects are located. Additionally, every site contains data structures for keeping transaction and object information. Transaction data structure contains information of transactions identification, execution phase, read and write sets, and other administration information. These data structures are used to maintain information on operations of the transaction and to find out what operations transaction has executed, which transactions have performed operation on this data item and so on.

In the read phase if a transaction reads an object which is in the local node then only necessary bookkeeping to the data structures is done and the object is returned to the transaction. Firstly, transaction requesting the read operation must be active and not aborted. Secondly, requested data item must not be marked as an validating object. Thirdly, if object is not located in the local node, distributed read operation is requested in the objects home node. This node is found from the object directory. A *subtransaction* whith the same identification

is created in the remote site. An identical bookkeeping is done in the remote site as in local site. Requested object is returned to requesting site and the object is returned to the transaction.

In the read phase if a transaction writes an object which is in the local node then a *prewrite* operation is executed and only necessary bookkeeping is done to the data structures. Firstly, transaction requesting the prewrite operation must be active and not aborted. Secondly, requested data item must not be marked as an validating or preparing object. Thirdly, if object is not located in the local node, a distributed prewrite operation is requested in the objects home node. This node is found from the object directory. A *subtransaction* is created in remote site which has same identity as a requesting transaction. The identical bookkeeping is done in remote site as in transactions local site. Requested object is returned to requesting site and to the requested transaction.

In the validation phase if the transaction is local transaction, then only local validation is executed. On the other hand, if the validating transaction is global transaction, then global validation have to be done. First, a coordinator is selected to coordinate commit protocol (2PL used here).

Coordinator will be the node where first operation of a distributed transaction arrived. Coordinator sends a PREPARE message to all nodes where the validating transaction have operations. Every participating site executes local validation and returns the result of the validation to coordinator. In same time, coordinator also executes local validation. If validation is successful, then participant sends YES message to coordinator. Otherwise participant sends ABORT message. If all participants (coordinator included) voted YES, then the coordinator sends COMMIT message to all participants. Otherwise the coordinator sends ABORT message. If no vote arrives from participant in predefined time, then vote is ABORT (presumed abort). This predefined time can be the same as transactions deadline.

Local validation consists iterating all objects accessed by the transaction, finding conflicting operation, and resolving conflicts. The adjustment of timestamp intervals iterates through the read set (RS) and write set (WS) of the validating transaction. First is checked that the validating transaction has read from committed transactions. This is done by checking the object's read and write timestamp. These values are fetched when the read and/or write to the current object was made. Then the set of active conflicting transactions is iterated. When access has been made to the same objects both in the validating transaction and in the active transaction, the temporal time interval of the active transaction is adjusted. Thus deferred dynamic adjustment of the serialization order is used (for more information see [16]).

In local validation a new check is needed for distributed objects. This is because state of the distributed object can be changed between last operation of the validation transaction and the validation phase by some other concurrently executing transaction. If it is, validating transaction must be restarted. This restart could be unnecessary, but it is required to ensure distributed serializability. This new check must be done to all read-write transactions, even if transaction is not

writing to the distributed object. This is because, transaction is creating a new value based on old value read from database.

Time intervals of all conflicting active transactions are adjusted after the validating transaction is guaranteed to commit. If the validating transaction is aborted no adjustments are done. Non-serializable execution is detected when the timestamp interval of an active transaction becomes empty. If the timestamp interval is empty the transaction is restarted.

If the validating transaction has read a data item, then the validating transaction read must be after latest committed write to this data item. If the validating transaction has announced intention to write (prewrite) a data item, then the validating transaction read must be after latest committed write an read to this data item. If there is active transaction which is announced intention to write (prewrite) to same data item which the validating transaction has read, then the active transactions write must be after the validating transaction. Therefore, the active transaction is forward adjusted in case of read-write conflict. If there is active transaction which has read the same data item which the validating transaction will write, then the active transactions read must be before the validating transaction. Therefore, the active transaction is backward adjusted in case of write-read conflict. If there is active transaction which is announced intention to write (prewrite) to same data item which the validating transaction will write, then the active transactions write must be after the validating transaction. Therefore, the active transaction is forward adjusted in case of write-write conflict.

If local transaction validation is successful or global transaction commit is successful in all participant sites, then final commit operation is executed. For all objects in the validating transactions write set a validate bookmark is requested. Then current read and write timestamps of accessed objects are updated and changes to the database are committed.

Finally, we present solution to all problems of distributed optimistic concurrency control method that were presented in section 2.

- **Problem 1**: Preserving the atomicity of validating and write phases [18].
  Solution: In the beginning of the validation phase PREPARE bookmark is set to all data items updated by the validating transaction in the data structures of the concurrency controller. Other transactions are allowed to read data items marked by PREPARE bookmarks but not update them. If another transaction enters in the validation phase and requests PREPARE bookmark for data item already marked with PREPARE bookmark, then this validating transaction is restarted. When the prepare section is finished, the node sends it's answer to the coordinator. Then the coordinator sends COMMIT-message (or ABORT-message in which case all bookmarks are released), which can be considered as global validate. VALIDATE bookmarks are set to data items updated by the validating transaction in the data structures of the concurrency controller. Reads to these data items are not allowed. The VALIDATE bookmarks are released after the transaction is written the data item to the database.

- **Problem 2**: The validation of subtransactions is made purely on local basis [18].
  Solution : In global validation first local validation is done by checking all active transactions. Therefore, also indirect conflicts are detected.
- **Problem 3**: Conflicts that are not detectable at the validation phase.

  Example 1 *Consider transactions* T_1 *and* T_2 *and history* H_1*. Transaction* T_1 *is started in the node* S_1 *and transaction* T_2 *is started in the node* S_2*.*
  $T_1 : r_1[X]\ w_1[Y]$
  $T_2 : r_2[Y]\ w_2[X]$
  $H_1 : r_1[X]\ r_2[Y]\ w_1[Y]\ w_2[X]\ c_1\ c_2$
  When executing c_1 *operation on both nodes proposed algorithm will see serializable history. But order of the distributed transactions is not same in all nodes. In the node* S_1 *order is* $T_1 \rightarrow T_2$ *and in the node* S_2 *order is* $T_2 \rightarrow T_1$*. Therefore, distributed serialization graph has cycle. This case cannot be found using OCC-DATI with 2PC, because 2PC is only an agreement algorithm. But this case can be find, if current state of the data items are used. OCC-DATI uses state of data item which was stored in the data item when read or write operation was executed. In the proposed method an extra check is done if the transactions is distributed and updated some data item. All data items are rechecked using current state of the data item. Therefore, in example the transaction* T_1 *is committed and the transaction* T_2 *is aborted. The transaction* T_2 *is aborted because state of the data item* Y *has been changed. Therefore, proposed method provides solution to problem 3.* □

  Solution : In distributed update transactions use current state of the data items.

4 Evaluation

The prototype system used in evaluations is based on the *Real-Time Object-Oriented Database Architecture for Intelligent Networks* (RODAIN) [15] specification. RODAIN Database Nodes that form one RODAIN Database Cluster are real-time, highly-available, main-memory database servers. They support concurrently running real-time transactions using an optimistic concurrency control protocol with deferred write policy. They can also execute non-real-time transactions at the same time on the database. Real-time transactions are scheduled based on their type, priority, mission criticality, or time criticality. All data in the database is stored in the main-memory database. Data modification operations are logged to the disk for persistence.

In order to increase the availability of the database each Rodain Database Node consists of two identical co-operative units. One of the units acts as the Database Primary Unit and the other one, Database Mirror Unit , is mirroring the Primary Unit. Whenever necessary, that is when a failure occurs, the Primary and the Mirror Units can switch their roles.

The database server was running on an Intel Pentium 450 MHz processor with 256 MB of main memory. A similar computer was used for the client. The computers were connected using a dedicated network, the speed of which was controlled by changing the hub connecting the computers. To avoid unnecessary collisions, there was no other network traffic while the measurements were performed.

Used database is based on a GSM model and transactions are simple trasactions accessing Home Location Register (HLR) and Visitor Location Register (VLR). Database size is 30000 items. The used transactions and they ratios are presented in table 1.

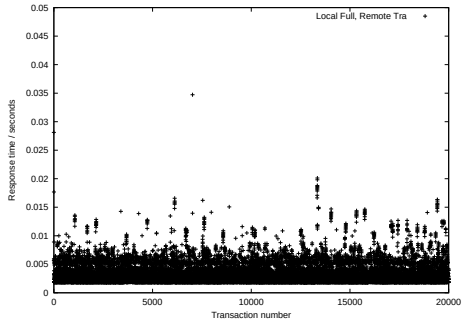
Table 1. Transactions used in the evaluation.

Transaction	type	ratio
GetSubscriber (HLR)	local read	70 %
GetSubscriber (VLR)	remote read	20 %
UpdateSubscriber	remote write	10 %

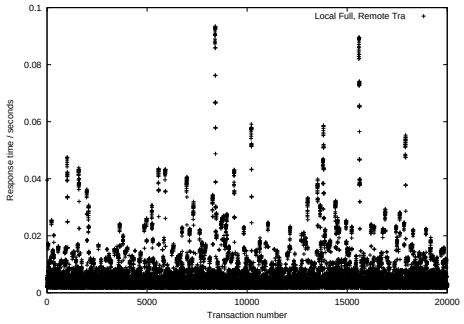
All time measurements were performed on the client computer using the gettimeofday function, which provides the time in microseconds. The client sends the requests following a given plan, which describes the request type and the time when the request is to be sent. When the request is about to be sent the current time is collected and when the reply arrives the time difference is calculated.

Linux provides static priorities for time-critical applications. These are always scheduled before the normal time-sharing applications. The scheduling policy chosen was Round-robin (SCHED_RR) using the scheduler function sched_setscheduler. The database was also avoiding swapping by locking all the processes pages in the memory using mlockall function. The swap causes long unpredictable delays, because occasionally some pages are sent and retrieved from the disk. Because in our experiment environment our database system was the only application running no swapping occurred during the tests.

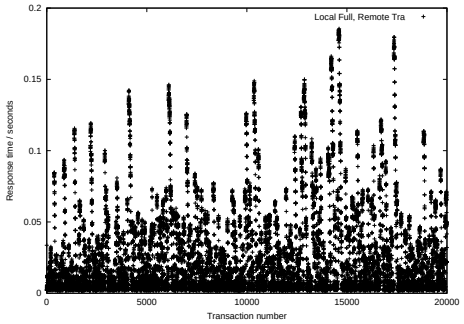
With low arrival rate the system can serve all requests within the deadlines. The single highest response time with 600 transactions per second is nearly 35 milliseconds (see Figure 1(a). A moderate arrival rate, 1000 tps (see figure 1(b)), which was usable in many precious tests here creates occasional situations, when the response time temporarily is higher than the 50 milliseconds. The transaction are treated similar in the measurements, because the service sequence does threat the differently. In the overload situation (arrival rate 1600 tps, see Figure 1(c)), the system is capable of serving most requests still within the 50 milliseconds. Unfortunately, there is no trend to predict which requests are served fast enough. Only a bit less than 20% (3400 requests out of the studies 20000) of all requests have response times over the 50 milliseconds. All kinds of requests belong to this 'over the deadline' group. The ratios of the served requests in this group are similar to the ratios of the original requests in the whole set.



(a) Arrival rate 600 tps



(b) Arrival rate 1000 tps



(c) Arrival rate 1600 tps

Fig. 1. Two database nodes. Local using Primary, Mirror and SSS units. Remote using only Transient unit

5 Conclusion

In this paper we have reviewed different distributed concurrency control techniques. Study have focused on distributed optimistic concurrency control methods, because optimistic concurrency control has been shown to be applicable to real-time database systems.

Our study has shown that there is no distributed optimistic concurrency control method, which is clearly suitable for real-time database system whitout modification. Therefore, a new distributed optimistic concurrency control method is proposed. Proposed method should be evaluated with some well known and widely used method. Therefore, we have selected 2PL-HP as reference method. With 2PL-HP we will use the 2PC method. We will in near future implement a simulation model for testing proposed method against 2PL-HP with 2PC.

References

1. D. Agrawal, A. E. Abbadi, and R. Jeffers. Using delayed commitment in locking protocols for real-time databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 104–113. ACM Press, 1992.
2. I. Ahn. Database issues in telecommunications network management. *ACM SIGMOD Record*, 23(2):37–43, June 1994.
3. D. Bell and J. Grimson. *Distributed Database System*. Addison-Wesley, 1992.
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
5. A. Datta, I. R. Viguier, S. H Son, and V. Kumar. A study of priority cognizance in conflict resolution for firm real time database systems. In *Proceedings of the Second International Workshop on Real-Time Databases: Issues and Applications*, pages 167–180. Kluwer Academic Publishers, 1997.
6. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.
7. T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.
8. J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 94–103. IEEE Computer Society Press, 1990.
9. J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 331–343. ACM Press, 1990.
10. S.-L. Hung and K.-Y. Lam. Locking protocols for concurrency control in real-time database systems. *ACM SIGMOD Record*, 21(4):22–27, December 1992.
11. H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.
12. K.-W. Lam, K.-Y. Lam, and S. Hung. An efficient real-time optimistic concurrency control protocol. In *Proceedings of the First International Workshop on Active and Real-Time Database Systems*, pages 209–225. Springer, 1995.
13. K.-W. Lam, K.-Y. Lam, and S. Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Proceedings of the IEEE Real-Time Technology and Application Symposium*, pages 174–179. IEEE Computer Society Press, 1995.
14. K.-Y. Lam, S.-L. Hung, and S. H. Son. On using real-time static locking protocols for distributed real-time databases. *The Journal of Real-Time Systems*, 13(2):141–166, September 1997.
15. J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Databases in Telecommunications*, Lecture Notes in Computer Science, vol 1819, pages 158–173, 1999.
16. J. Lindström and K. Raatikainen. Dynamic adjustment of serialization order using timestamp intervals in real-time databases. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 13–20. IEEE Computer Society Press, 1999.
17. K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1:199–226, April 1993.
18. G. Schlageter. Problems of optimistic concurrency control in distributed database systems. *ACM SIGMOD Record*, 13(3):62–66, April 1982.