Hybrid Concurrency Control Method in Firm Real-Time Databases

Jan Lindström
IBM Finland Lab
Laajalahdentie 23, 00101 Helsinki, Finland
jan.lindstrom@fi.ibm.com

Abstract

Real-time database system must meet time constraints in addition to the integrity constraints. Concurrency control is one of the main issues in the studies of real-time database systems. Traditional concurrency control methods use serializability as the correctness criterion when transactions are executed concurrently. However, strict serializability as the correctness criterion is not always suitable in real-time databases. Instead, correctness requirements vary from one type of transactions to another and from one data type to another. In this paper we propose a concurrency control method where used concurrency control is based on method defined on table creation. Transactions can freely use tables based on pessimistic concurrency control, optimistic concurrency control or nocheck concurrency control based on application needs. Proposed method is evaluated and tested in prototype implementation of real-time database system for telecommunications.

1. Introduction

Many real-word applications involve time-constrained access to data and access to data that has temporal validity. Consider for example a telephone switching system, network management, navigation systems, stock trading, and command and control systems. Additionally, consider the following operations within these environments: looking up the ”800 directory”, obstacle detection and avoidance, radar tracking and recognition of objects. These involve gathering data from the environment, processing of information, and contributing *timely* response. Additionally, these examples contain processing both temporal data, which loses its validity after a certain time intervals, as well as historical data.

Traditional databases, hereafter referred to as databases, deal with persistent data. Transactions access this data while maintaining its consistency. The overall goal of transaction and query processing in databases is to get a good throughput or response time. In *real-time systems* can also deal with temporal data, i.e., data that becomes outdated after a certain time. Therefore, tasks in real-time systems have deadlines. The important difference is that the goal of real-time systems is to meet the time constraints of the tasks [21].

Real-time does not just mean fast [21] and timing constraints that are in nanoseconds or microseconds. Instead, real-time means the need to manage *explicit* time constraints in a predictable fashion, that is, to use time-cognizant protocols to deal with deadlines or periodicity constraints associated with tasks [18].

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) [9]. 2PL has been studied extensively in traditional database systems and is being widely used in commercial databases.

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 [19].

Optimistic concurrency control protocols [15, 10] 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 [12, 11], 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 [14, 16, 7, 17, 8, 23].

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 Intelligent Network (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. In this paper two transaction types have been studied: service provision (IN) and service management (TMN) [3] transactions. In transaction scheduling, IN transactions are expressed as firm deadline transactions and TMN transactions are expressed as soft deadline transactions.

The rest of the paper is organized as follows. Section 2 presents the proposed method. Section 3 presents a experimental setup and results. Finally, the conclusion of the paper is presented in Section 4.

2. Hybrid Concurrency Control

There are three different concurrency control methods available for users: pessimistic, optimistic and nocheck. These have different behaviors to the user. These different methods are discussed more thoroughly in the following subsections.

2.1. Pessimistic Concurrency Control

Pessimistic concurrency control is also known as ”locking”. Locks allow multiple users to safely share a database as long as all users are updating different data at the same time. For example, you can update Ms. Smith’s record while I update Mr. Kumar’s record.

When locks are used, the locks are placed as soon as any piece of the row is updated. Thus it is impossible for two users to update a row at the same time. As soon as one user gets a lock, no one else can process that row. This is a safe, conceptually simple approach. The disadvantage is that it requires overhead for every operation, whether or not two or more users are actually trying to access the same record. This overhead is small, but adds up because every row that is updated requires a lock. Furthermore, every time that a user tries to access a row, the system must also check whether the requested row(s) are already locked by another user (or connection).

Pessimistic concurrency control is called ”pessimistic” because the system assumes the worst – it assumes that two users will want to update the same record at the same time, and then prevents that possibility by locking the record, no matter how unlikely conflicts actually are.

Here is an example of pessimistic concurrency control operation:

On T1, enter the statements below:

```
create table t2(a int) engine=soliddb
comment='MODE=PESSIMISTIC';
insert into t2 values (1),(2);
commit;
set autocommit  = 0;
update t2 set a = 5 where a = 1;
```

Figure 1. Using pessimistic tables.

On T2, enter the statements below:

```
set autocommit = 0;
update t2  set a = 7  where a = 1;
/* This query waits for T1 to release
locks on table t2 */
```

We have selected to use 2PL-HP [1] method in pessimistic concurrency control.

2.2. Optimistic Concurrency Control

An alternative approach to locking is called ”optimistic” concurrency control. Optimistic concurrency control assumes that although conflicts are possible, they will be very rare. Instead of locking every record every time that it is used, the software merely looks for indications that two users actually did try to update the same record at the same time. If that evidence is found, then one user’s updates are discarded (and of course the user is informed).

When using optimistic concurrency control, each time that the server reads a record to try to update it, the server makes a copy of the ”version number” of the record and stores that copy for later reference. When it’s time to write the updated data back to the disk drive, the server compares the original version number that it read with the version number that the disk drive now contains. If the version numbers are the same, then no one else changed the record and we can write our updated value. However, if the value we originally read and the current value on the disk are not the same, then someone has changed the data since we read it, and whatever operation we did is probably out-of-date, so we discard our version of the data and give the user an error message. Naturally, each time that we update a record, we also update the version number. Here is an example of optimistic concurrency control operation: On T1, enter the statements below:

```
create table t2(a int) engine=soliddb
comment='MODE=OPTIMISTIC';
insert into t2 values (1),(2);
commit;
set autocommit = 0;
update t2 set a = 5 where a = 1;
```

Figure 2. Using optimistic tables.

on T2, enter the statements below:

```
set autocommit = 0;
update t2 set a = 7 where a = 1;
```

On T2, you can see:

ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction.

2.3. Nocheck Concurrency Control

In this alternative approach no concurrency control checking is done at all. However, normal consistency checking is done, i.e. primary key, foreign key and check constraints are checked. This table mode need careful design and usage from the user because concurrent changes could cause unwanted results. When the application uses e.g. replace or insert semantics or when data in this table does not need always to be consistent this table mode provides best possible concurrency between transactions and increased performance because no concurrency checks are done in any phase of the transaction execution. Here is an example of creating nocheck table:

```
create table t3(a int) engine=soliddb
comment='MODE=NOCHECK';
```

Figure 3. Using nocheck tables.

2.4. Example of hybrid concurrency control

Lets assume that we have tables A, B, and C. Furthermore, assume that we have defined that the table A uses pessimistic concurrency control, B uses optimistic concurrency control and C nocheck concurrency control. Now, transaction:

```
BEGIN;
UPDATE A SET A.B = A.B + 1 WHERE A.ID BETWEEN 21 AND 56;
UPDATE B SET B.A = B.A - 1 WHERE B.ID BETWEEN 33 AND 99;
UPDATE C SET C.C = C.A WHERE C.ID BETWEEN 77 AND 212;
COMMIT;
```

Figure 4. Transaction accessing all three tables.

Now this transaction will use pessimistic locking when accessing rows in the table A, optimistic method when accessing rows in the table B and no concurrency control when accessing rows in the table C. Thus, transaction might wait locks to be granted while accessing rows in the table A and transaction might be rolled back at commit time if at validation phase of rows accessed the table B do not serialize.

2.5. Differences Between Optimistic and Pessimistic Concurrency Control

When you use optimistic concurrency control, you don't find out there's a conflict until just before you write the updated data. In pessimistic locking, you find out there's a conflict as soon as you try to read the data. To use our analogy with banks again, pessimistic locking is like having a guard at the bank door who checks your account number when you try to enter; if someone else (a spouse, or a merchant to whom you wrote a check) is already in the bank accessing your account, then you can't enter until that other person finishes her transaction and leaves. Optimistic concurrency control, on the other hand, allows you to walk into the bank at any time and try to do your business, but at the risk that as you are walking out the door the bank guard will tell you that your transaction conflicted with someone else's and you'll have to go back and do the transaction again.

Optimistic and pessimistic concurrency controls differ in another important way besides the time at which conflicts are detected and error messages are issued. Pessimistic locking allows one user to not only block another user from updating the same record, but even from reading that record. If you use pessimistic locking and you get an exclusive lock, then no other user can even read that record. With optimistic locking, however, we don't check for conflicts except at the time that we write updated data to disk. If user1 updates a record and user2 only wants to read it, then user2 simply reads whatever data is on the disk and then proceeds, without checking whether the data is locked. User2 might see slightly out-of-date information if user1 has read the data and updated it but has not yet "committed" the transaction.

solidDB actually implements optimistic concurrency control in a more sophisticated way than this. Rather than giving each user "whatever version of data is on the disk at

the moment it is read", solidDB can store multiple versions of each data row temporarily. Each user's transaction sees the database as it was at the time that the transaction started. This way, the data that each user sees is consistent throughout the transaction, and users are able to concurrently access the database. Data is always available to users because locking is not used; access is improved since deadlocks no longer apply. (Again, however, users run the risk that their changes will be thrown out if those changes conflict with another user's changes.)

Thus, for example, user1 might put an exclusive lock on a record and update it. When the record is updated, its version number changes. User2, who is using a read-only transaction, can read the previous version of the record even though the record has an exclusive lock on it.

Pessimistic locking allows you an option that optimistic locking does not offer. We said earlier that pessimistic locks fail "immediately" – that is, if you try to get an exclusive lock on a record and another user already has a lock (shared or exclusive) on that record, then you will be told that you can't get a lock. In fact, solidDB allows you the option of either failing immediately or of waiting a specified number of seconds before failing. You might specify a wait of 30 seconds; this means that if you initially try to get the lock and cannot, the server will continue trying to get the lock until either it gets the lock or until the 30 seconds has elapsed. In many cases, especially when transactions tend to be very short, you may find that setting a brief wait allows you to continue activities that otherwise would have been blocked by locks.

This wait mechanism applies only to pessimistic locking, not to optimistic concurrency control. There is no such thing as "waiting for an optimistic lock". If someone else changed the data since the time that you read it, no amount of waiting will prevent a conflict that has already occurred. In fact, since optimistic concurrency methods do not place locks, there is literally no "optimistic lock" to wait on.

Neither pessimistic nor optimistic concurrency control is "right" or "wrong". When properly implemented, both approaches ensure that your data is properly updated. In most scenarios, optimistic concurrency control is more efficient and offers higher performance, but in some scenarios pessimistic locking is more appropriate. In situations where there are a lot of updates and relatively high chances of users trying to update data at the same time, you probably want to use pessimistic locking. If the odds of conflict are very low (many records and relatively few users, or very few updates and mostly "read" operations), then optimistic concurrency control is usually the best choice. The decision will also be affected by how many records each user updates at a time.

2.6. Example database and transactions

To motivate semantic correctness in real-time database setting, consider network database that mimics a Home Location Register (HLR) [24] which is used to store information about users of the network. Operators use HLR databases to store subscriber data, location data, network access data, and about network services data, for example call forwarding. To simplify presentations schema presented below does not contain all the operations of the HLR. Instead database and transactions are simplified version of The Telecom One (TM1) benchmark designed for telecommunication applications [22]. Database consists three tables: Subscriber, SpecialFacility, and CallForwarding. The basic data, such as the location data, of all subscribers using the network is found in the Subscriber table. Network services accessible to a subscriber are stored in the SpecialFacility table. Each of those services might have a number of call forwarding's, which are stored in the CallForwarding table.

- SUBSCRIBER(*s_id*, cf_active, vlr_location, hlr_location)
- SPECIALFACILITY(*s_id*, sf_type, is_active)
- CALLFORWARDING(*s_id*, cf_number, sf_type, start_time, end_time)

From these tables we have selected that SUBSCRIBER table uses pessimistic concurrency control while SPECIALFACILITY uses optimistic concurrency control and CALLFORWARDING nocheck concurrency control. This is because subscriber information needs higher consistency and have potentially more conflicts. Call information is naturally temporal and have lower consistency requirements and has potentially less conflicts.

3. Experimental results

We have carried out a set of experiments in order to examine the feasibility of our prototype implementation, specifically the concurrency control mechanism. All experiments were executed in the prototype database running on a AMD Opteron Processor 146 containing 3 GB of main memory with the Linux operating system 2.6.28.

The test database represents a typical network database that mimics a Home Location Register (HLR) [24] which is used to store information about users of the network. Operators use HLR databases to store subscriber data, location data, network access data, and about network services data, for example call forwarding. To simplify presentations schema presented below does not contain all the operations of the HLR. Instead database and transactions are from The

Telecom One (TM1) benchmark designed for telecommunication applications [22]. Transaction parameters used in these test are listed in the Table 1.

Table 1. Transaction parameters.

Transaction name	Test # #1	#2	Deadline	Mode
GetNewDestination	90 %	85 %	50 ms	read-only
UpdateDestination	10 %	10 %	100 ms	update
UpdateLocation	0 %	5 %	150 ms	replace

Transactions are validated atomically. If the deadline of a transaction expires, the transaction is always aborted. Other test parameters include the exponentially-distributed arrival rate. The final deadline is calculated using EDF (Earliest Deadline First) [13] method.

New transactions are accepted up to a specified limit, which is the number of installed Transaction Processes. If there is no Transaction Process available when a new transaction arrives, the transaction is aborted. Transactions are validated atomically. If the deadline of a transaction expires, the transaction is always aborted. The workload in a test session consists of a variable mix of transactions. The fraction of each transaction type is a test parameter. Other test parameters include the exponentially-distributed arrival rate. The relative deadline of all firm real-time transactions is 50ms. The final deadline is calculated adding relative deadline to arrival time. Thus, deadlines are also exponentially distributed.

3.1. Results of our experiments

In the Figure 5 we have compared overall performance of different Multiversion concurrency control (MVCC) implementations in MySQL/InnoDB and MySQL/solidDB storage engines. MySQL/InnoDB implements traditional MVCC using locks while MySQL/solidDB uses method presented in this paper. This experiment clearly shows that proposed method provides a lot better overall performance because it allows more concurrency between different type of transactions.

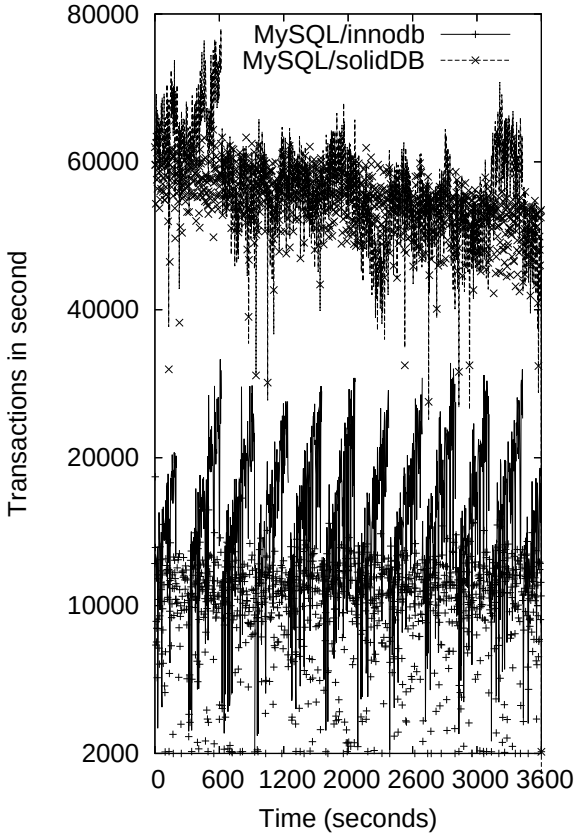

Figure 5. MVCC methods compared.

4. Conclusions

Although the optimistic approach has been shown to have a better performance than locking protocols in firm real-time database systems, it has problems of unnecessary restarts and a high restart overhead. In this paper, we have proposed a method where application developer can assign best suited concurrency control method to table based on application consistency needs and transaction conflict probability.

Experiment results using real-time database system for telecommunications clearly indicate that the proposed method is able to dynamically adapt changing workload situations.

4.1. Related Work

In [5] a hybrid concurrency control scheme that uses a deadlock-free approach was presented. This method ensures that transactions are re-started at most once. In the proposed scheme there are two types of transactions: first-run transaction and second-run transaction. First-run transactions use modification of broadcast optimistic concur-

rency control where if transactions are known to conflict, the transaction manager tries to re-order the transactions. If re-ordering fails transaction manager shifts to second-run state where transaction is restarted only after all locks to the data items can be granted. Another similar research can be found from [25] where pure optimistic concurrency control is used in first run and broadcast optimistic concurrency control method in second run.

Similarly [20] proposes an hybrid protocol where both optimistic and pessimistic concurrency control is used for the same transaction.

In [6] a hybrid concurrency control method for mobile computing is proposed that uses a pre-validation of the transactions that is implemented by use of weak locks. Our proposed scheme is not developed only for mobile system, instead our proposed method is usable in general transaction processing.

Our proposed method differs this method significantly because transactions are executed at most once. If transaction can't be executed because concurrency control failure, transaction is rolled back not re-executed. Additionally, concurrency control is based on table properties not transaction properties. In our proposed method if transaction access only optimistic tables then optimistic concurrency control is used and similarly if transaction uses pessimistic tables then pessimistic concurrency control is used. Finally, transaction can use optimistic to some tables and pessimistic to other tables.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of the 14th VLDB Conference*, pages 1–12, Los Angeles, California, USA, 1988. Morgan Kaufmann.

[2] I. Ahn. Database issues in telecommunications network management. *ACM SIGMOD Record*, 23(2):37–43, June 1994.

[3] M. Appeldorn, R. Kung, and R. Saracco. TMN + IN = TINA. *IEEE Communications Magazine*, 31(3):78–85, Mar. 1993.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] S. Cho. A hybrid concurrency control with deadlock-free approach. In *Computational Science and Its Applications – ICCSA 2003*, Lecture Notes in Computer Science, 2668. Springer Verlag, 2003.

[6] S. Cho, J. Lee, and C. Hwang. Hybrid concurrency control for mobile computing. In *Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia*, pages 478–483. IEEE, 1997.

[7] H.-J. Choi and B.-S. Jeong. A timestamp-based optimistic concurrency control for handling mobile transactions. In *ICCSA (2)*, pages 796–805, 2006.

[8] A. Climent, J. Navarro, M. Bertran, and F.-X. Babot. Optimistic concurrency control with partial replication design. In *IADIS AC (2)*, pages 181–186, 2009.

[9] 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, Nov. 1976.

[10] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[11] 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, 1990.

[12] 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.

[13] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, pages 232–242, San Antonio, Texas, USA, 1991. IEEE Computer Society Press.

[14] A. S. Jalal, S. Tanwani, and A. K. Ramani. Optimistic concurrency control in firm real-time databases. In *IWDC*, pages 487–492, 2005.

[15] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[16] G. Li, B. Yang, and J. Chen. Efficient optimistic concurrency control for mobile real-time transactions in a wireless data broadcast environment. In *RTCSA*, pages 443–446, 2005.

[17] A. Makni, R. Bouaziz, and F. Gargouri. Performance evaluation of an optimistic concurrency control algorithm ensuring strong consistency for transaction time relations. In *Enterprise Information Systems and Web Technologies*, pages 258–265, 2007.

[18] B. Purimetla, R. M. Sivasankaran, K. Ramamritham, and J. A. Stankovic. Real-time databases: Issues and applications. In S. H. Son, editor, *Advances in Real-Time Systems*, pages 487–507. Prentice Hall, 1996.

[19] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, Apr. 1993.

[20] S. H. Son, J. Lee, and Y. Lin. Hybrid protocols using dynamic adjustment of serialization order for real-time concurrency control. *The Journal of Real-Time Systems*, 4(2):269–276, June 1992.

[21] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, June 1999.

[22] T. Strandell. Open source database systems: Systems study, performance and scalability. Series of publications C, master of science thesis, University of Helsinki, 2003.

[23] W. Wang and C. Amza. On optimal concurrency control for optimistic replication. In *ICDCS*, pages 317–326, 2009.

[24] Wikipedia. Network switching subsystem. Technical report, http://en.wikipedia.org/wiki/Home_Location_Register.

[25] P. S. Yu and D. M. Dias. Analysis of hybrid concurrency control for a high data contention environment. *IEEE Transactions on Software Engineering*, SE-18(2):118–129, Feb. 1992.