Partial Order Multi Version Concurrency Control

Yuya Isoda, Atsushi Tomoda, Tsuyoshi Tanaka, Kazuhiko Mogi

Hitachi, Ltd. Research & Development Group
1-280, Higashi-koigakubo, Kokubunji-shi, Tokyo, Japan
email: { yuuya.isoda.sj, atsushi.tomoda.nx, tsuyoshi.tanaka.vz, kazuhiko.mogi.uv } @ hitachi.com

Abstract — **This paper presents the Partial Order Multi Version Concurrency Control (POMVCC), which is a concurrency control technique based on partial ordering of transactions. We claim that timestamp generation can be a bottleneck in multicore, high-throughput systems and POMVCC can execute multiple transactions using same timestamp without losing the consistency level. In this paper, we change the ordering of transaction processing from total order to partial order and propose partial order transaction processing on Multi Version Concurrency Control (MVCC), which numbers a timestamp in partial order per N transactions. This helps the system to reduce the overall number of increments to the timestamp and therefore improves the overall performance of the system. We claim that POMVCC achieves as high as 1.74 times the throughput of the conventional MVCC based system. We implemented a lock-free version of POMVCC in MPDB, which is their under development database system.**

Keywords – Partial Order Transaction Proccessing; In-memory DB; timestamp; Concurrency Control.

I. Introduction

In recent years, the number of CPU cores and the size of memory have increased owing to the progress of hardware technology. In the case of DataBase Management Systems (DBMSs), scalability technology for multicore CPUs [7] [12][15] and large-scale and non-volatile in-memory technology [14][16] are advancing rapidly, and the performance of DBMS is close to reaching one million Transactions Per Second (tps) [3][12].

DBMS must guarantee ACID properties to maintain data consistency [22]. However, strictly doing so prevents a DBMS from improving performance because it needs to process Transactions (Tx) as serial processing in strict total order [13]. To increase performance, a DBMS generally uses the isolation level, which mitigates ACID properties step by step, performance in parallel processing is improved.

Recently, Multi Version Concurrency Control (MVCC) has been used for controlling the isolation level. It manages timestamps of both before and after updating a record and enables records to be referenced and updated simultaneously. As a result, it increases the performance of OnLine Transaction Processing (OLTP) and OnLine Analytical Processing (OLAP). Also, recent research has clarified how SERIALIZABLE can be implemented. Therefore, DBMSs with MVCC are thought to prevail in the near future [23][24].

There are two types of Timestamps (Ts) for MVCC, that is, a physical clock and a logical clock. The physical clock is the time used in the real world, such as Coordinated Universal Time (UTC). The Network Time Protocol (NTP) is prevailingly used as a protocol for synchronizing UTC among servers. However, the logical clock is not related to time in the real world, such as UTC and is a counter that determines the order, in which events occur. The Lamport method is known as a mechanism for sharing this counter among servers [28].

The logical clock implementation in DBMSs is common [5]. Spanner implemented a physical clock for DBMSs, but such an example is rare [29]. In recent years, the performance of DBMSs is close to reaching one million tps owing to in-memory technology, multicore technology, and improved transaction management methods [3][12]. In addition, the size of memory and the number of CPU cores, e.g., Hewlett Packard's Memory-Driven Computing, will be increasing more and more [32]. The bigger the system is, the more difficult the conventional timestamp management becomes. For example, in recent computers, it is mandatory for timestamps to be numbered every 1 us. In such an environment, large-scale mutual exclusion with a high CPU clock frequency may be problematic.

Silo is proposed for this problem [3]. Silo is the timestamp based on Epoch. It periodically updates the high-order bits of the timestamp. Transaction threads update low-order bits under the condition that they satisfy the order of dependence. As a result, Silo can reduce the number of updates for the timestamp. However, it cannot be easily adapted for the conventional MVCC-based DBMS because it needs lock processing and management of the Read-Set and Write-Set for concurrency control.

In this paper, we propose Partial Order Multi Version Concurrency Control (POMVCC). POMVCC is technology based on the reduction of the conflict rate, which is caused by a large-scale DB. It mitigates the problems with simultaneous executable transactions. Specifically, it updates the timestamp at an abort. Thus, multiple transactions can be processed at the same timestamp, and the number of timestamp updates can be reduced.

In summary, our contributions are the following.

1. We propose partial order transaction control based on reconsidering the isolation level of MVCC. To update a timestamp at an abort, POMVCC can process multiple transactions at the same timestamp and reduce the number of timestamp updates. In addition, POMVCC is easily implementable for DBMS based on MVCC.

ISBN: 978-1-61208-637-8

2. We show the cause and the solution of a new anomaly named "HISTORICAL READ" caused by POMVCC.
3. We also show a lock-free implementation of POMVCC.
4. Finally, we implement POMVCC on an in-memory DBMS and evaluate the performance.

The rest of this paper is organized as follows. In Section 2, we introduce research on concurrency control for DBMS. In Section 3, we reconsider the requirement of concurrency control for DBMS and show a problem with performance and scalability. In Section 4, we propose POMVCC. We also show the cause and the solution of a new anomaly named "HISTORICAL READ" with POMVCC. In Section 5, we describe a method for implementing POMVCC that is lock-free. In Section 6, we evaluate the performance and consider the results. Finally, in Section 7, we give concluding remarks and our future work.

II. Related Work

In this section, we show work related to concurrency control for DBMSs. The most notable viewpoint of concurrency control is the persistence of an execution result and the concurrency control of transactions.

Algorithms for Recovery and Isolation Exploiting Semantics (ARIES) is a general persistence processing [17]. ARIES is composed of analysis, REDO, and UNDO. Analysis pinpoints the starting point of a recovery. REDO re-executes a transaction on the basis of a REDO log. UNDO deletes an uncommitted transaction on the basis of an UNDO log. During logging, Write-Ahead Logging (WAL), which can restore logs safely in the case of failure, is used. WAL has a problem in that the speed of writing a log to a storage device is slow. However, in recent years, speedup technology that uses distributed logging with non-volatile memory has been proposed for WAL [14].

Research on the concurrency control of transactions has been made since the 1980s. There are two types of concurrency control, that is, Pessimistic Concurrency Control (PCC) and Optimistic Concurrency Control (OCC) [4][6][1]. For PCC, concurrency control with a 2 Phase Lock (2PL) is mainly used. DORA, PLP, and Shore-MT are proposed as lock-based DBMSs [8][9][11][19]. However, in recent years, DBMSs with MVCC, which enables OCC, have been proposed because the processing cost of locks and latches is high [13][25][26][27].

In past research, it was stated that an isolation level for SERIALIZABLE cannot be realized [2]. However, the proposal of SERIALIZABLE SNAPSHOT ISOLATION enabled this [23] [24]. Using this technology, H-Store/VoltDB [10][18], Hekaton [7][15], and SAP HANA [16] were proposed as MVCC-based DBMS. H-Store creates transaction sites whose number is the same as the number of CPUs, and transaction threads that stick to the logical sites execute SQL. Such a mechanism enables in-memory and lock-free fast processing. To reduce the number of responses between interfaces, Hekaton compiles stored procedures into native codes. SAP HANA manages both the row store whose update efficiency is high and column store whose reference efficiency is high. A lot of MVCC-based DBMSs whose characteristics are diverse are proposed like these examples.

In addition, a Silo in-memory DBMS that manages Epoch-based timestamps as a concurrency control was proposed [3]. In Silo, updates of timestamps are removed from the concurrency control of a transaction. Silo uses a special-purpose thread for managing timestamps. As a result, it achieves high performance. In addition, it creates temporary areas per transaction for references (Read-Set) and updates (Write-Set). Concurrency control with the Read-Set and Write-Set can use cache and memory efficiently. Using these technologies, Silo achieves 700,000 tps for the industry standard benchmark TPC Benchmark^{TM} C (TPC-C) [20]. Moreover, Silo-based transaction control is adopted by Intel's Rack-Scale Architecture, which has become popular these days, and in-memory DBMS Foedus [12], which supposes Hewlett Packard's Memory-Driven Computing [32]. Silo-based concurrency control has become popular.

Research on MVCC-based DBMSs is now advancing. Silo-like concurrency control enables further speedup. However, it is difficult to adopt it for MVCC-based DBMSs because many components, such as thread management, transaction control, and data management must be modified. There, we propose an easier method that is equivalent to Silo's concurrency control for MVCC-based DBMSs.

III. Reconsidering Anomalies and Concurrency Control on MVCC

In this section, we outline concurrency control on MVCC, and we reconsider the update conflict of timestamps, which are a problem in Silo, and clarify the problem.

A DBMS must keep ACID properties, but to do so strictly, transactions must be serialized, and this degrades performance. To avoid this phenomenon, an isolation level, in which ACID properties are mitigated gradually is used. The isolation level is defined as the allowable range for an anomaly, which occurs when transactions are executed in parallel. This mitigation achieves high scalability enabled by the highly parallel and high performance transactions of DBMSs.

The isolation level is different between lock-based control and MVCC-based control [2]. In this paper, we outline the relationship of the isolation level for MVCC and anomalies, and we clarify the order of transactions and the problem with scalability.

In the following, we define Begin (B) as the start of a transaction, Commit (C) as the commit of the transaction, Abort (A) as the abort of the transaction, Read (R) as the reference in the transaction, and Write (W) as the update in the transaction. We also define TX1, TX2, etc., as identifiers of the transaction, X, Y, etc., as a set of records, and i, j, etc., as integers. The time at which commit is completed is the Committed Time (CT). The attribute of transaction type is defined as Type. For Type, Read represents read only, and Write includes write.

A. *Relationship between Isolation Level and Anomalies*

WRITE SKEW (WS), FUZZY READ (FR), READ SKEW (RS), and LOST UPDATE (LU) are general anomalies [2]. Examples of anomalies are shown in Table 1.

For example, LOST UPDATE happens when Tx1 and Tx2 update record X simultaneously and both are successful. This is a problem because the value of the record is either X' or X", and the update history of the record is not uniquely determined. In the case of one-side failure (W1 W2 C2 A1), LOST UPDATE may occur when Tx2 updates record X to X' and then Tx1 aborts and the record X' is roll-backed to X.

The isolation level is defined as the allowable range for anomalies. SERIALIZABLE SNAPSHOT ISOLATION has the strictest requirement of consistency. The second strictest is READ COMMITTED. READ UNCOMMITED is the least strict. Table 2 shows the relationship between the isolation level and anomalies. For example, in the case of READ COMMITED, WRITE SKEW or FUZZY READ may occur. READ UNCOMMITTED is hardly used because user-unallowable anomalies occur.

TABLE I. ANOMALIES ON MVCC

Anomaly	Formula
LOST UPDATE	W2[X→X’] W1[X→X”]
READ SKEW	W2[X→X’, Y→Y’] R1[X, Y’]
FUZZY READ	R1[X] W2[X→X’] R1[X’]
WRITE SKEW	R1[X] R2[Y] W1[Y→Y’] W2[X→X’]

TABLE II. ISOLATION LEVEL ON MVCC

Isolation Level	Anomaly
SERIALIZABLE	-
SNAPSHOT ISOLATION	WS
READ COMMITTED	WS, FR
READ UNCOMMITTED	WS, FR, RS, LU

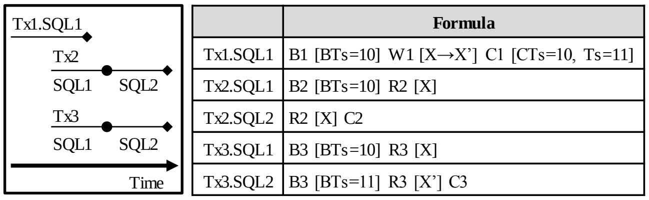
B. *Concurrency Control*

MVCC controls records and transactions by using timestamps. MVCC manages the update history of records by giving Timestamps at Commit (CTs) to the records. Transactions refer to Timestamps at Begin (BTs) or when SQL executes. They update timestamps at Commit. They refer to the latest record whose timestamp is smaller than BTs. The references of transactions maintain consistency with this method. How BTs are treated is different among the isolation level. SERIALIZABLE and SNAPSHOT ISOLATION use a timestamp that is referred to at Begin. READ COMMITTED uses a timestamp that is referred to at SQL execution. Figure 1 shows the difference between SNAPSHOT ISOLATION and READ COMMITTED. Tx2 and Tx3 are assumed to be SNAPSHOT ISOLATION and READ COMMITTED, respectively. They execute the SQL at the same time. However, Tx2.SQL2 sees record X, but Tx3.SQL2 sees record X'. Such concurrency control protects SNAPSHOT ISOLATION from FUZZY READ. Similarly, READ SKEW is prevented.

Update conflicts at the Commit of transactions generally use First Committer Win, which is optimistic concurrency control. It executes transactions in the order, in which Commits are executed. It keeps consistency by aborting subsequent conflicting transactions.

The concurrency control explained above cannot prevent WRITE SKEW from occurring. It happens when references and updates of multiple transactions mutually conflict (RW-Conflict). Serializable Snapshot Isolation (SSI) was proposed to find such a condition and avoid WRITE SKEW [23][24]. SSI adds a read flag and write flag to the conventional MVCC algorithm and detects RW-Conflict. SSI aborts at least one of the RW-Conflict transactions and avoids WRITE SKEW. Therefore, SERIALIZABLE is enabled. SSI can realize SERIALIZABLE with the same performance of SNAPSHOT ISOLATION [23][24].

We can prevent anomalies from occurring by using these concurrency controls on MVCC.



	Formula
Tx1.SQL1	B1 [BTs=10] W1 [X→X’] C1 [CTs=10, Ts=11]
Tx2.SQL1	B2 [BTs=10] R2 [X]
Tx2.SQL2	R2 [X] C2
Tx3.SQL1	B3 [BTs=10] R3 [X]
Tx3.SQL2	B3 [BTs=11] R3 [X’] C3

Figure 1. Difference between SNAPSHOT ISOLATION (Tx2) and READ COMMITTED (Tx3)

C. *Problem of Scalability*

To keep ACID properties strictly, it is necessary for transactions to be processed in strict total order. In this case, scalability is low. To the contrary, MVCC enables high scalability by parallel execution in total order. Table 3 defines D1 as strict total order, D2 as the total order, and D3 as the order of transactions for MVCC.

The CTs of MVCC must be different between the two transactions shown in D3.I, or one of the transactions must be the reference transaction shown in D3.II. That is, multiple update transactions cannot be committed at the same time due to D3.II. Thus, the transactions of MVCC are in strict total order in the case of update transactions only, or it is in total order when transactions include reference transactions.

As described above, MVCC allows D3.II, instead of D1 only, and scalability increases. However, D3.II is applicable only for transactions including reference transactions. In the case of update transactions only, scalability is low, because the conditions of the order are the same as D1. Therefore, mitigating the order of update transactions under D3.II is a problem.

TABLE III. DEFINITION OF MVCC

D1. Definition of Strict Total Order	
i < j < = = > i ≦ j AND i ≠ j	
D2. Definition of Total Order	
i ≦ j < = = > i < j OR i = j	
D3. Definition of Committed Tx. Order for MVCC	
CTs (Tx i) ⊴ CTs (Tx j) < = = > Ⅰ OR Ⅱ	
Ⅰ	CT (Tx i) < CT (Tx j)
Ⅱ	CT (Tx i) = CT (Tx j) AND Type (Tx i) = Read

ISBN: 978-1-61208-637-8

IV. PROPOSAL OF POMVCC

In this section, we propose POMVCC, which mitigates the order of update transactions and realizes high scalability. In addition, a new anomaly caused by POMVCC is considered.

In the following, DB is defined as the content of a database, and the execution order of transactions is shown as →.

A. Basic Idea

On the basis of the consistency of a database, transactions can be controlled in partial order. For example, if the concurrency control of DBMS exchanges the execution order of one transaction with other transaction and the result is not changed, these transactions can be executed in non-order, and consistency is kept. Thus, we do not need to update timestamps per update transaction, and we can share one timestamp among multiple update transactions. Therefore, we propose POMVCC as new concurrency control focused on the partial order of transactions. POMVCC gives a same timestamp to two update transactions if they have no dependency. This method mitigates condition D3.II, so scalability can increase.

The concept and definition of POMVCC are shown in Figure 2 and Table 4. By controlling the partial order of transaction processing, POMVCC eliminates the need to update the timestamp every time Tx. process is ended. POMVCC updates the timestamp when it detects Anomaly. For example, in Figure 2, since LOST UPDATE occurred between Tx1 and Tx3, POMVCC will update timestamp. Even if the execution order of all Tx. processes within the same timestamp is changed, POMVCC permits simultaneous execution if the contents of the database same. Then we also show the allowable conditions of transaction processing on the same timestamp for MVCC (D3.II) and POMVCC (D4.II) in Table 5, which shows POMVCC has more conditions that can be executed simultaneously than MVCC. Therefore, POMVCC can reduce the update frequency of timestamps. This means that the scalability of POMVCC is better than that of MVCC.

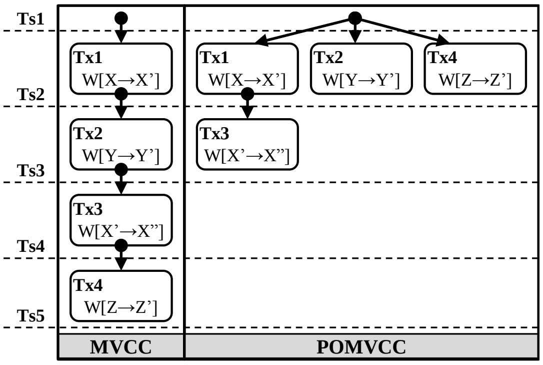


Figure 2. Difference between MVCC and POMVCC

TABLE IV. DEFINITION OF POMVCC

D4. Definition of Committed Tx. Order for POMVCC	
CTs (Tx i) ≤ CTs (Tx j) <===> I OR Ⅱ	
I	CT (Tx i) < CT (Tx j)
Ⅱ	CT (Tx i) = CT (Tx j) AND DB(Tx i → Tx j) = DB(Tx j → Tx i)

TABLE V. ALLOWABLE RANGE OF TRANSACTIONS FOR D3.II AND D4.II ON THE SAME TIMESTAMP

#	Formula	D3. Ⅱ	D4. Ⅱ
1	R1[X] R2[X]	Success	Success
2	R1[X] W2[X→X']	Success	Success
3	W1[X→X'] R2[X]	Success	Success
4	W1[X→X'] W2[Y→Y']	Failure	Success
5	W1[X→X'] W2[X→X'']	Failure	Failure

B. How to Control POMVCC

The trigger to update a timestamp of POMVCC is different from that of MVCC. MVCC updates a timestamp at the Commit of a transaction, but POMVCC updates it at the Abort of a transaction. Thus, multiple update transactions can be executed at one timestamp.

A schematic diagram of POMVCC is shown in Figure 3. Tx1 and Tx3 have the update conflict of record X. In the case of MVCC, a timestamp is updated at the Commit of Tx1, but in the case of POMVCC, a timestamp is not updated. Therefore, Tx3 refers to old record X, and an update conflict happens. POMVCC updates a timestamp at the Abort of Tx3. Record X can be updated when Tx3 is re-executed. Because a timestamp is updated at the Abort of a transaction caused by an anomaly, partial order transaction control can be realized.

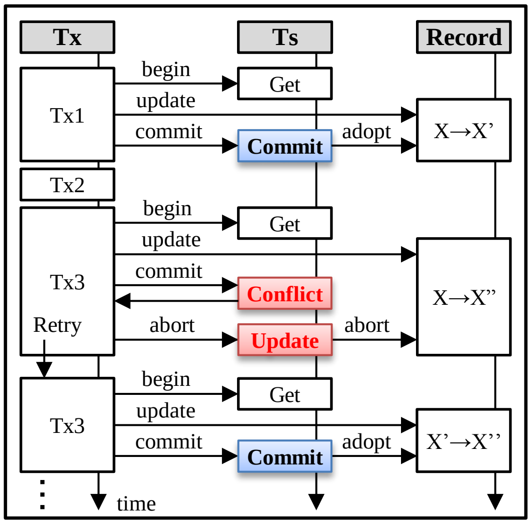


Figure 3. Concurrency Control of POMVCC

C. New Anomaly: HISTORICAL READ

The partial order transactions of POMVCC enable highly scalable concurrency control. However, the execution order of transactions is limited by the APplication (AP) or user. For example, consider that the succeeding transaction refers to the result of the preceding transaction. In this case, a HISTORICAL READ (HR), in which the succeeding transaction cannot refer to the result of the preceding transaction, occurs. It is necessary for POMVCC to provide the result of the preceding transaction to the succeeding transaction when AP requires the result of the preceding transaction.

Table 6 shows the definition of HISTORICAL READ. Tx2 cannot refer to record X', which Tx1 updates after the Commit of Tx1. This is the anomaly. If Tx1 and Tx2 are independent transactions, this does not happen. However, when AP assumes that the execution order is Tx1→Tx2, such an unexpected response occurs.

TABLE VI. DEFINITION OF HISTORICAL READ

Anomaly	Formula
Historical Read	W1[X→X'] C1 B2 R2[X]

D. How to Avoid HISTORICAL READ

HISTORICAL READ is avoidable if the BTs of a succeeding transaction is bigger than the CTs of the preceding transaction. That is, when the same user (DB connection) or the same AP executes transactions, the value that is bigger than the CTs of the preceding transaction is given to the BTs of the succeeding transaction. Therefore, HISTORICAL READ can be avoided.

The avoidance method for the same user (User Approach) may include false positives. In the worst case, timestamps are updated at every Commit. For example, the independent transactions that the same user issues do not need timestamp updates. However, in the User Approach, timestamps are always updated at the Begin of the transactions. As a result, performance degradation is a concern due to there being a lot of false positive cases.

In the avoidance method for the same AP (AP Base Method), minimum timestamps which would preferably be referred to, are set when the AP issues transactions. This method can avoid HISTORICAL READ efficiently because false positives are excluded. However, the DB interface, such as Commit and Begin, must be modified, and this is a downside of this method.

Figure 4 shows the solution of the AP Base Method. POMVCC returns CTs at the Commit of Tx1, and BTs (=CTs) is set at the Begin of Tx2. As a result, Tx1.CTs < Tx2.BTs is established, and Tx2 can refer to the execution result of Tx1.

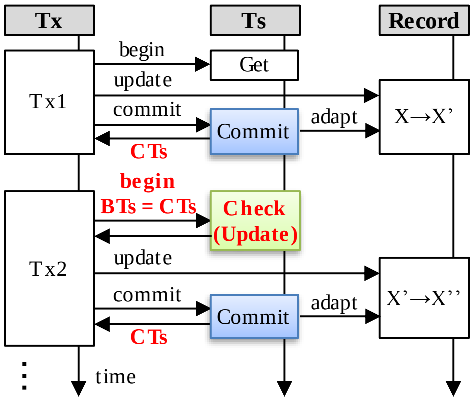


Figure 4. Solution for HISTORICAL READ

V. IMPLEMENTATION OF POMVCC

The lock used in parallel processing may degrade scalability [15]. In this section, to avoid this degradation, we introduce a lock-free implementation for scalable POMVCC.

However, in POMVCC, the implementation related to general DMBS is not different from the MVCC implementation of other pieces of literature [3][7][10][12][15][18][24]. Therefore, in the following, we focus on the extension of MVCC, that is, the concurrency control of transactions and timestamps.

In POMVCC, timestamps are divided into reference timestamps (RTs) and commit timestamps (WTs). Figure 5 shows the data structure of POMVCC. It has RTs, WTs, monotonically increasing timestamps, and the number of transactions at commit per timestamp. RTs are the timestamps that are used for referring to a record. WTs are the timestamps for a Commit. In addition, the state of Commit processing is divided into PreCommit and Commit. The PreCommit state includes the success of solving a conflict and the transfer to the Commit state. The Commit state includes giving CTs to all updated records and the completion of issuing a log. That is, if the timestamp of a PreCommit Counter and the Commit Counter is the same, the record can be referred to by using this timestamp while keeping consistency.

Figure 6 describes the control of POMVCC. In POMVCC, after the state is transferred to the PreCommit, CTs (= WTs) is obtained, and the PreCommit Counter of the CTs is incremented. After the state is transferred to the Commit, the Commit Counter of the CTs is incremented, and the transaction is completed. In case of Abort, WTs is incremented. If the timestamp (ATs) that causes the abort is known, RTs is incremented to ATs+1. At the re-execution of the transaction, this prevents the next abort, which has the same abort reason as the previous abort. RTs can be incremented if the PreCommit Counter and Commit Counter are the same and RTs < WTs. Finally, at Begin, RTs tries to be updated. If BTs is specified as the Begin interface, RTs is incremented till BTs < RTs is satisfied.

These controls enable POMVCC. They can be implemented without a lock by using atomic instructions, such as Compare-And-Swap (CAS).

RTs	WTs
10	12

Ts	PreCommit Counter	Commit Counter
9	14	14
10	3	3
11	6	2
12	1	0
13	0	0

Figure 5. Data Structure of POMVCC

```
Tx.Begin ( TmpTs = CTs or ATs) {
  WTs = Get.WTs ( ) ;
  while ( WTs ≦ TmpTs ) {
    WTs = Increment.WTs ( ) ;
  }
  do {
    BTs = Check.RTs ( ) ;
  } while ( BTs ≦ TmpTs ) ;
  return ( ) ;
}

Tx.Commit ( ) {
  // PreCommit Process
  if ( Tx.Judgement = Success ) {
    CTs = GetWTs ( ) ;
    Increment.PreCommitCnt ( CTs ) ;
    ･･･Commit Completion･･･
    Increment.CommitCnt ( CTs ) ;
  } else if ( Tx.Judgement = Failure) {
    Tx.Abort ( ) ;
  }
  return ( CTs or ATs) ;
}

Tx.Abort ( ) {
  // Abort Process
  IncrementWTs ( ) ;
  return ( ) ;
}

Check.RTs ( ) {
  RTs = Get.RTs ( ) ;
  WTs = Get.WTs ( ) ;
  if ( Diff.Commit.Counter ( RTs ) = 0 and RTs < WTs )
    RTs = Increment.RTs ( ) ;
  return ( RTs ) ;
}
```

Figure 6. Schematic Timestamp Control

VI. Evaluation of Prototype Implementation

In this section, we compare the performance of MVCC and POMVCC. We implemented MVCC and POMVCC on an in-memory DBMS named “MPDB”, which we are developing, and evaluated their performance. MPDB is an MVCC-based, lock-free, in-memory DBMS that is characterized by parallel logs and PCC/OCC mixed control [30] [31].

In this experiment, we use the industry standard benchmark TPC-C and repeatedly execute stored procedure calls that model NewOrder [20].

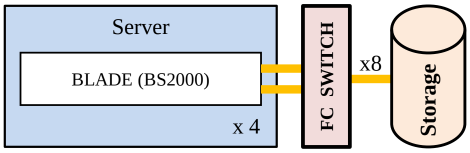
A. Experimental Environment

Figure 7 depicts the system configuration. Four blade servers were used. They were Symmetric MultiProcessors (SMP) and had 8 CPUs (80 cores), 1 TB of memory, and 8 ports of an 8-Gb Fiber Channel (FC). Servers and storage were connected via an FC switch and communicate with FC communication.

In the OS (CentOS 6.5) settings, FC ports were assigned to each CPU to distribute the interrupt overhead of FC communication. Hyper-threading was disabled.

In the MPDB settings, one thread was assigned to one core. This means that MPDB uses a maximum of 80 threads. One log file is assigned to one CPU to load balance logs. The isolation level was SNAPSHOT ISOLATION.

The DB was created on the basis of TPC-C. The number of warehouses was 16 and the size of database was 0.72 GB. The item table, stock table, and order_line table were used in TPC-C. In addition, indexes were created for the i_id of the item table, s_w_id and s_i_id of the stock table, and ol_o_id and ol_w_id of the order_line table.



System Configuration

Blade	BS2000
CPU	Xeon(R) E7 8870 x 2
Memory	256GB (16GB x 16)
PCIe	2 Port HBA (8Gb)

Storage	Hitachi Unified Storage VM (HUS-VM)
Cache	54GB
Disk	6.4TB (1.6TB x 4) Hitachi Accelerated Flash
RAID	RAID5(3D + 1P)

Figure 7. System Configuration

B. Workload

The workload shown in Figure 8 was created on the basis of TPC-C’s New Order. The workload simulates the repeatedly executing part of the New Order. The processing in Figure 8 was repeated 10 times per transaction on average.

1	SELECT	i_price, i_name, i_data
	INTO	:i_price, :i_name, :i_data
	FROM	item
	WHERE	i_id = :ol_i_id
2	SELECT	s_quantity, s_data, s_dist_...
	INTO	:s_quantity, :s_data, :s_dist_...
	FROM	stock

	WHERE	s_i_id = :ol_i_id AND
		s_w_id = :ol_supply_w_id

3	UPDATE	stock
	SET	s_quantity = :s_quantity

	WHERE	s_i_id = :ol_i_id AND
		s_w_id = :ol_supply_w_id

4	INSERT	
	INTO	order_line (,,,,,)
	VALUES	(,,,,,,)

While (Repeats 5 ~ 15 times, Ave. 10)

Figure 8. Experiment Workload

C. *Experimental Results and Consideration*

The experiments were done to compare the performance of MVCC and POMVCC corresponding to the number of threads. In Figure 9, the x-axis means the number of threads, and the y-axis means the transactional performance (tps). The performance of both MVCC and POMVCC increased as the number of threads increased. POMVCC ran 1.36-1.60 times faster than MVCC.

To investigate scalability more precisely, we made an experiment, in which the number of warehouses changed corresponding to the number of threads. That is, the number of warehouses was 10 (DB size was 0.45 GB) when the number of threads is 10. The number of warehouses was 80 (DB size was 3.61 GB) when the number of threads was 80. Figures 10 and 11 are the experimental results. From Figure 10, the performance of POMVCC was 1.63 - 1.74 times better than that of MVCC. From Figure 11, the scalability coefficient of MVCC was 87.98 - 97.96 [%], and that of POMVCC was 94.02 –98.32 [%]. This experiment says that the scalability coefficient of POMVCC is greater than that of MVCC.

From these experiments, the scalability coefficients of POMVCC and MVCC depended on the size of the DB and the number of threads. If the size of the DB was large and the conflict rate of the transaction was low, the scalability coefficient of POMVCC was high, and in all experiments, POMVCC ran faster than MVCC.

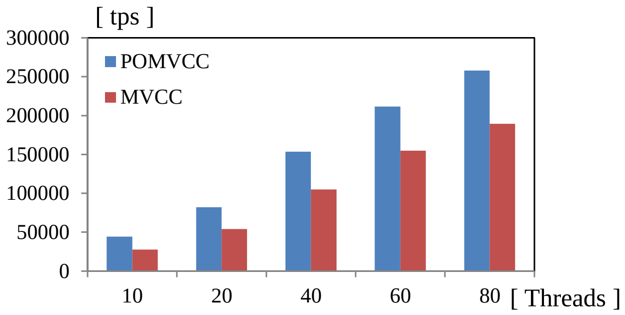


Figure 9. Performance Evaluation

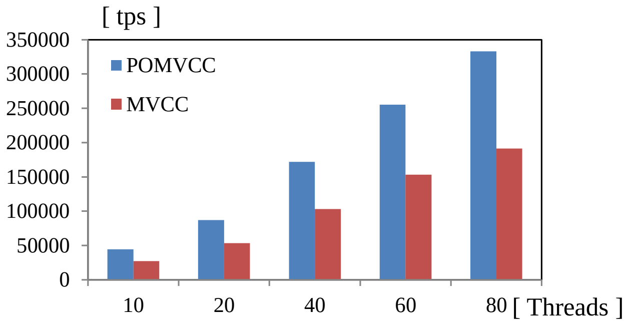


Figure 10. Performance when Number of Warehouses Changes

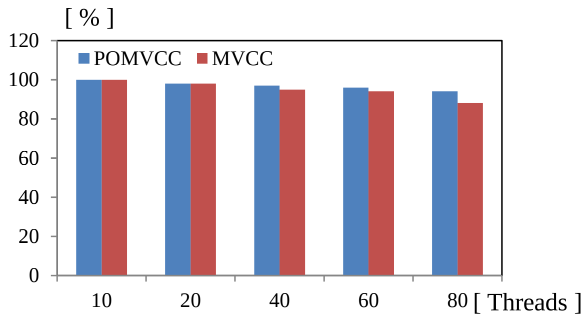


Figure 11. Scalability Coefficient when Number of Warehouses Changes

VII. CONCLUSION

In this paper, we proposed and evaluated POMVCC, which keeps the consistency of MVCC and improves performance and scalability. POMVCC is technology that focuses on the partial order of transactions. The conventional method gives a timestamp to a transaction, but POMVCC gives a timestamp to multiple transactions. POMVCC reduces the number of timestamps that are updated and improves performance and scalability. We show the difference of Isolation Level between MVCC and POMVCC in Figure 12.

We implemented and evaluated POMVCC on an in-memory DBMS named “MPDB” that we are developing. From experiments, the performance of POMVCC was 1.30 - 1.74 times better than that of MVCC. The scalability of the POMVCC was higher than that of the MVCC. Every experiment showed that the performance of POMVCC was 1.30 - 1.74 times higher than that of the MVCC.

We implemented the POMVCC on the MPDB and evaluated it by using SNAPSHOT ISOLATION, for which POMVCC had higher performance than MVCC. However, with SERIALIZABLE, the performance trend was unclear because the probability of WRITE SKEW increased. This occurs when reference and update transactions are executed at the same timestamp. POMVCC increases the number of transactions at the same timestamp. As a result, the number of WRITE SKEWs increases. In addition, it is possible that RW-CONFLICT GRAPH will grow and a large cyclic graph will be created. Therefore, our future work is to implement and evaluate POMVCC by using SERIALIZABLE.

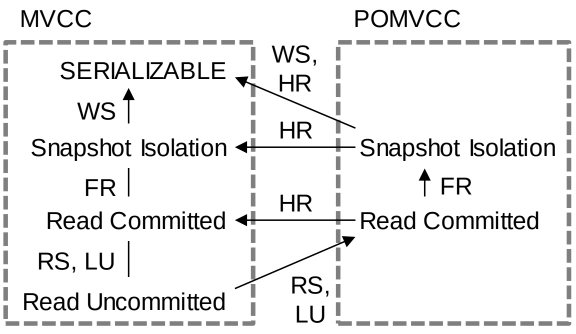


Figure 12. A Diagram of the Isolation Levels and Relationships

REFERENCES

[1] D. A. Menascé, and T. Nakanishi, “Optimistic versus pessimistic concurrency control mechanisms in database management systems,” Information Systems Volume 7, Issue 1, pp. 13-27, 1982.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil and, P. O'Neil, “A Critique of ANSI SQL Isolation Levels,” ACM SIGMOD '95 Proceedings, pp. 1-10, San Jose, CA, 1995.

[3] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, “Speedy Transactions in Multicore In-Memory Databases,” SOSP '13 Proceedings, pp. 18-32, Farmington, Pennsylvania, USA, 2013.

[4] H. T. Kung, and J. T. Robinson, “On optimistic methods for concurrency control,” ACM Transactions on Database Systems, Volume 6 Issue 2, pp. 213-226, 1981.

[5] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, “High-performance concurrency control mechanisms for main-memory databases,” Proceedings of the VLDB Endowment Volume 5 Issue 4, pp. 298-309, 2011.

[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, Volume 19 Issue 11, pp. 624-633, 1976.

[7] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, “Hekaton: SQL server's memory-optimized OLTP engine,” SIGMOD '13 Proceedings, pp. 1243-1254, 2013.

[8] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamak, “Data-oriented transaction execution,” Proceedings of the VLDB Endowment, Volume 3 Issue 1-2, pp. 928-939, 2010.

[9] I. Pandis, P. Tozun, R. Johnson, and A. Ailamaki, “PLP: page latch-free shared-everything OLTP,” Proceedings of the VLDB Endowment, Volume 4 Issue 10, pp. 610-621, 2011.

[10] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, “The end of an architectural era: (it's time for a complete rewrite),” VLDB '07 Proceedings, pp. 1150-1160, 2007.

[11] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, “Shore-MT: a scalable storage manager for the multicore era,” EDBT '09 Proceedings, pp. 24-35, 2009.

[12] H. Kimura, “FOEDUS: OLTP Engine for a Thousand Cores and NVRAM,” SIGMOD '15 Proceedings, pp. 691-706, 2015.

[13] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, “OLTP through the looking glass, and what we found there,” SIGMOD '08 Proceedings, pp. 981-992, 2008.

[14] T. Wang, and R. Johnson, “Scalable logging through emerging non-volatile memory,” Proceedings of the VLDB Endowment, Volume 7 Issue 10, pp. 865-876, 2014.

[15] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, “High-performance concurrency control mechanisms for main-memory databases,” Proceedings of the VLDB Endowment, Volume 5 Issue 4, pp. 298-309, 2011.

[16] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, “Efficient transaction processing in SAP HANA database: the end of a column store myth,” SIGMOD '12 Proceedings, pp. 731-742, 2012.

[17] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, “ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging,” ACM Transactions on Database Systems, Volume 17 Issue 1, pp. 94-162, 1992.

[18] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, and et al., “H-store: a high-performance, distributed main memory transaction processing system,” Proceedings of the VLDB Endowment, Volume 1 Issue 2, pp. 1496-1499, 2008.

[19] P. A. Bernstein, V. Hadzilacos, and N. Goodman, “Concurrency Control and Recovery in Database System,” 1987.

[20] The Transaction Processing Council, “TPC-C Benchmark (Version 5.11.0),” http://www.tpc.org/tpcc/, March 2018.

[21] G. Weikum, and G. Vossen, “Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery,” Elsevier, 2001.

[22] J. Gray, and A. Reuter, “Transaction Processing: Concepts and Techniques,” Elsevier, 1992.

[23] M. J. Cahill, U. Röhm, and A. D. Fekete, “Serializable isolation for snapshot databases,” ACM Transactions on Database Systems, Volume 34 Issue 4, Article No.20, 2009.

[24] A. Fekete, D. Liarokapis, P. O'Neil, and D. Shasha, “Making snapshot isolation serializable,” ACM Transactions on Database Systems, Volume 30 Issue 2, pp. 492-528, 2005.

[25] ORACLE, “Oracle Database 12c Release 2,” https://docs. oracle.com/en/database/oracle/oracle-database/12.2/index. html, March 2018.

[26] MySQL, “MySQL 5.7 Reference Manual,” https://dev.mysql. com/doc/refman/5.7/en/, March 2018.

[27] PostgreSQL, “PostgreSQL 9.6.8 Documentation,” https:// www.postgresql.org/docs/9.6/static/index.html, March 2018.

[28] L. Lamport, D. Malkhi, and L. Zhou, “Reconfiguring a state machine,” ACM SIGACT News, Volume 41 Issue 1, pp. 63-73, 2010.

[29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, and et al., “Spanner: Google’s Globally Distributed Database,” ACM Transactions on Computer Systems, Volume 31 Issue 3, Article No.8, 2013.

[30] Y. Isoda, A. Tomoda, K. Ushijima, T. Tanaka, T. Uemura, T. Hanai, and et al., “In-Memory Database Engine for Scale-up System,” Forum on Information Technology '15, D-035, 2015 (in Japanese).

[31] Y. Isoda, K. Ushijima, T. Tanaka, T. Hanai, and K. Mogi, “Proposal of Multi Version Concurrency Control for Partial Order Transaction,” Forum on Information Technology '16, D-015, 2016 (in Japanese).

[32] Hewlett Packard, “Memory-Driven Computing,” https://news. hpe.com/content-hub/memory-driven-computing/, March 2018.