POMVCC: 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 proposes Partial Order Multi-Version Concurrency Control (POMVCC), which is a concurrency control technique based on partial order transaction processing. We claim that timestamp generation per transaction can be a critical section on multi-core for high-throughput DataBase Management Systems (DBMSs), and POMVCC can execute multiple transactions using the same timestamp without losing consistency. In this paper, we change the order of transaction processing from total to partial on Multi-Version Concurrency Control (MVCC), which allocates a timestamp on partial order per multiple transactions. It helps the DBMS reduce the overall number of increments to the timestamp; therefore, improving its overall performance. We claim that a POMVCC-based system achieves 1.74 times higher throughput than that of a conventional MVCC-based system. We implemented a lock-free version of POMVCC on MPDB, which is in-memory DBMS.**

Keywords – Partial order transaction processing; Multi-version concurrency control; Transaction; Timestamp; In-memory DB.

I. INTRODUCTION

We research to adapt new hardware technology or new software techniques to old DataBase Management Systems (DBMS) techniques [1][2][3]. For example, the number of Central Processing Unit (CPU) cores and memory size have recently increased due to the progress of hardware technology. For DBMSs, scalability technology [4][5][6] for multicore CPUs and large-scale and non-volatile in-memory technology [7][8] are advancing rapidly, and the performance of DBMSs is close to reaching one million Transactions Per Second (tps) [5][9].

A DBMS must guarantee the Atomicity, Consistency, Isolation and Durability (ACID) properties to maintain data consistency [10]. However, strictly doing so prevents a DBMS from improving performance because it needs to process Transactions (Tx) as serial processing in total order [11]. To improve performance, a DBMS generally uses the isolation level, which lessens ACID properties step by step; thus, improving parallel processing.

Multi-Version Concurrency Control (MVCC) has recently 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). Recent research has also clarified how SERIALIZABLE can be implemented. Therefore, DBMSs with MVCC are expected to become widespread in the near future [12][13].

There are two types of Timestamps (Ts) for MVCC, i.e., physical clock and 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 widely used as a protocol for synchronizing UTC among servers [14]. Implementation of a logical clock in DBMSs is common [15]. Spanner implemented a physical clock for DBMSs, but such an example is rare [16]. The larger the system is, the more difficult conventional timestamp management becomes using a logical clock. Because it is mandatory for timestamps to be numbered every 1 us to reach one million tps. In such an environment, large-scale mutual exclusion with a high CPU clock frequency may be problematic. In addition, the memory size and the number of CPU cores will increase, e.g., Hewlett Packard’s Memory-Driven Computing, will further increase [17].

Silo was proposed to solve this problem [9]. 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 conventional MVCC-based DBMSs because it requires lock processing and management of the Read-Set and Write-Set for concurrency control.

Moreover, we must better understand the partial order model and low isolation levels because a user requires two advanced points. The first point is high-performance and high-scalability. NoSQL is very fast and executes 80–120 million operations per second [18]. If we want to promote only DBMS to a data management system for simple management, the performance of DBMS needs to exceed the one of NoSQL. The second point is that we must understand the meaningless assumptions on industry, as shown in Figure 1 [19]. High isolation levels and the stored procedures are not needed on industry. Not all transactions are executed as stored procedures only 47% of users (excluded 0% and 1-10% on Figure 1.B), and almost all users do not set the isolation level of SERIALIZABLE. Read Committed is most frequently used; therefore, we need to develop a high-performance DBMS on a low isolation level.

From these reasons, we propose Partial Order Multi-Version Concurrency Control (POMVCC), which is the partial order transaction processing based on the reduction in the conflict rate, which is caused by a large-scale DB. It mitigates the problems with simultaneous executable transactions on each isolation level. Specifically, it increments a timestamp during the abortion phase of a transaction. Thus, multiple transactions can be processed at the same timestamp, and the number of timestamp updates can be reduced on any isolation levels.

In summary, our contributions are as follows.

1. We propose partial order transaction control based on reconsidering the isolation level of MVCC, called POMVCC. To update a timestamp during the abortion phase of a transaction, POMVCC can process multiple transactions at the same timestamp and reduce the number of timestamp updates. It is also easily implementable for DBMSs based on MVCC.
2. We show the cause and solution of a new anomaly called "HISTORICAL READ" caused by POMVCC.
3. We also show a lock-free implementation of POMVCC and discuss the implementation of mixed Pessimistic Concurrency Control (PCC) and Optimistic Concurrency control (OCC) to solve the problem of long-short transaction.
4. Finally, we discuss the implementation of POMVCC on an in-memory DBMS and the evaluation its performance.

The rest of this paper is organized as follows. In Section II, we introduce research on concurrency control for DBMSs. In Section III, we reconsider the requirement of concurrency control for DBMSs and present a problem with performance and scalability. In Section IV, we propose POMVCC and discuss a new anomaly called "HISTORICAL READ" caused by POMVCC and its solution. In Section V, we describe a method for implementing our developed MPDB, which is an MVCC-based, lock-free, in-memory DBMS characterized by parallel logs and mixed PCC/OCC. In Section VI, we describe a method for implementing POMVCC that is lock-free. In Section VII, we discuss the evaluation of POMVCC's performance and present the results. Finally, in Section VIII, we give concluding remarks and discuss our future work.

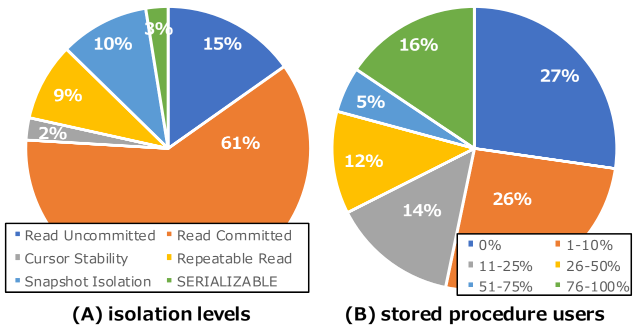


Figure 1. Survey on frequency of use on DBMS functions

II. Related Work

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

Algorithms for Recovery and Isolation Exploiting Semantics (ARIES) involve general persistence processing [20]. 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, and 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, faster technology that uses distributed logging with non-volatile memory has recently been proposed for WAL [7].

Research on the concurrency control of transactions has been conducted since the 1980s. There are two types of concurrency control, i.e., PCC and OCC [21][22][23]. For PCC, concurrency control with a 2-Phase Lock (2PL) is mainly used. DORA [24], PLP [25], and Shore-MT [26] have been proposed as lock-based DBMSs [27]. However, DBMSs with MVCC, which enables OCC, have recently been proposed because the processing cost of locks and latches is high [28][29][30].

It was stated that an isolation level for SERIALIZABLE is not possible [31]. However, the proposal of SERIALIZABLE SNAPSHOT ISOLATION (SSI) has made this possible [12][13]. Using this technology, H-Store/ VoltDB [32][33], Hekaton [4][6], and SAP HANA [8] were proposed as MVCC-based DBMSs. H-Store creates transaction sites, the number of which is the same as the number of CPUs, and transaction threads that stick to the logical sites execute Structured Query Language (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, the update efficiency of which is high, and column store, the reference efficiency of which is high. Many such MVCC-based DBMSs that have diverse characteristics have been proposed.

Moreover, a Silo in-memory DBMS that manages Epoch-based timestamps as a concurrency control has also been proposed [9]. In Silo, updates of timestamps are removed from the concurrency control of a transaction on Single-Version Consistency Control (SVCC). 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 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) [34]. Moreover, Silo-based transaction control is adopted by Intel's Rack-Scale Architecture, which has become popular, and in-memory DBMS Foedus [5], which uses Hewlett Packard's Memory-Driven Computing [17]. Therefore, Silo-based concurrency control has become popular.

Research on SVCC-based DBMSs is now advancing. Silo-like concurrency control enables faster than conventional MVCC-based DBMSs. 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. Therefore, we propose an easier implementation technique 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 reconsider the update conflict of timestamps, which is a problem in Silo, and solve this problem.

A DBMS must maintain ACID properties, but to do so strictly, transactions must be serialized, which degrades performance. To avoid this phenomenon, an isolation level, in which ACID properties are lessened 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 differs between lock-based control and MVCC-based control [31]. We outline the relationship of the isolation level for MVCC and anomalies and clarify the order of transactions and mitigate the problem with scalability.

We define B as the begin phase of a transaction, C as the commit phase of the transaction, A as the abort of the transaction, BTs as an allocated timestamp during the begin phase, CTs as an allocated timestamp during the commit phase, ATs as an allocated timestamp during the abort phase, R as the reference in the transaction, and W as the update/insert/delete in the transaction. We also define Tx.1, Tx.2, etc., as identifiers of transactions X, Y, etc. as a set of records and i, j, etc. as integers.

A. *Relationship between isolation level and anomalies*

The general anomalies are WRITE SKEW (WS), FUZZY READ (FR), READ SKEW (RS), and LOST UPDATE (LU) on MVCC [31]. Examples of these anomalies are listed in Table I.

For example, LOST UPDATE occurs when Tx.1 and Tx.2 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. For one-side failure (W1 W2 C2 A1), LOST UPDATE may occur when Tx.2 updates record X to X', then Tx.1 aborts and record X' is roll-backed to X.

The isolation level is defined as the allowable range for anomalies. SSI has the strictest requirement of consistency. The second strictest is READ COMMITTED and READ UNCOMMITED is the least strict. Table II lists the relationships between the isolation level and anomalies. For example, for 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 (LU)	W2[X → X'] W1[X → X'']
READ SKEW (RS)	W2[X → X', Y → Y'] R1[X', Y]
FUZZY READ (FR)	R1[X] W2[X → X'] R1[X']
WRITE SKEW (WS)	R1[X] R2[Y] W1[Y → Y'] W2[X → X']

TABLE II. ISOLATION LEVELS ON MVCC

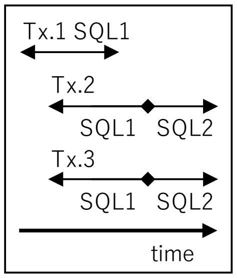
Isolation Level	LU	RS	FR	WS
SERIALIZABLE	-	-	-	-
SNAPSHOT ISOLATION	-	-	-	v
READ COMMITTED	-	-	v	v
READ UNCOMMITTED	v	v	v	v

B. *Concurrency control*

MVCC controls records and transactions by using a timestamp. MVCC manages the update history of records by allocating a timestamp at the commit to the records. Transactions refer to a timestamp at the begin phase or when SQL executes and to the latest record whose timestamp is smaller than BTs. The references of transactions maintain consistency with this method. How BTs is treated differs depending on the isolation level. SERIALIZABLE and SNAPSHOT ISOLATION use a timestamp that is referred to at the begin phase. READ COMMITTED uses a timestamp that is referred to at SQL execution. Figure 2 shows the difference between Tx.2 as SNAPSHOT ISOLATION and Tx.3 as READ COMMITTED. They execute the SQL at the same time. However, Tx.2.SQL2 reads X, but Tx.3.SQL2 reads X'. Such concurrency control protects SNAPSHOT ISOLATION from FUZZY READ. Similarly, READ SKEW is prevented.

The update conflicts at the validation of the commit process generally use First Committer Win (FCW), which is an OCC. It executes transactions in the order in which the commit command is executed. It maintains consistency by aborting subsequent conflicting transactions.

The concurrency control explained above cannot prevent WRITE SKEW from occurring. This occurs when references and updates of multiple transactions mutually conflict (RW-Conflict). SSI was proposed to find such a condition and avoid WRITE SKEW [12][13]. SSI adds a read flag and write flag to the conventional MVCC algorithm and detects RW-Conflict. It aborts at least one of the RW-Conflict transactions and avoids WRITE SKEW. Therefore, SERIALIZABLE is enabled. SSI enables SERIALIZABLE with the same performance of SNAPSHOT ISOLATION [12][13]. Thus, we can prevent anomalies from occurring by using these concurrency controls on MVCC.



SQL		Formula
Tx.1	SQL1	B1[BTs=10] W1[X→X'] C1[CTs=10, Ts=11]
Tx.2	SQL1	B2[BTs=10] R2[X]
	SQL2	R2[X] C2
Tx.3	SQL1	B3[BTs=10] R2[X]
	SQL2	B3[BTs=11] R2[X'] C3

Figure 2. Difference between SNAPSHOT ISOLATION (Tx.2) and READ COMMITTED (Tx.3)

C. Problem of scalability

To strictly maintain ACID properties, it is necessary for transactions to be processed in total order. Scalability is low in this case. Table III defines D1 as total order, D2 as weak order, and D3 as the order of transactions for MVCC.

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

As described above, MVCC increases scalability; however, it 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, we must mitigate the order of update transactions under D3. ii in order to improve the scalability for DBMS.

TABLE III. DEFINITION OF MVCC

D1. Total Order
i < j <==> i≦j and i≠j
D2. Weak Order
i ≦ j <==> i<j or i=j

D3. MVCC for write tx.	
CTs(Tx.i) < CTs(Tx.j) <==> i and ii	
i	CTs(Tx.i) ≦ CTs(Tx.j)
ii	CTs(Tx.i) ≠ CTs(Tx.j)

IV. PROPOSAL OF POMVCC

In this section, we propose POMVCC, which mitigates the order of update transactions and enables high-scalability. We also consider a new anomaly caused by POMVCC.

We define DBC as the content of a database, and the execution order of transactions is shown as (→).

A. Basic idea

Transactions can be controlled in partial order on the basis of the consistency of a DBC. For example, if the concurrency control of DBMS exchanges the execution order of one transaction with another transaction and the result does not change, these transactions can be executed in non-order, and consistency is maintained. Thus, we do not need to update the timestamp per transaction update and can share one timestamp among multiple update transactions. Therefore, we propose POMVCC as a new concurrency control focused on the partial order of transactions. POMVCC provides the same timestamp to two update transactions if they have no dependency. This technique mitigates condition D3. ii, so scalability can increase.

The concept and definition of POMVCC are shown in Figure 3 and Table IV. By controlling the partial order of transaction processing, POMVCC eliminates the need to update the timestamp every time transaction process is ended. POMVCC updates the timestamp when it detects an anomaly. For example, in Figure 3, since LOST UPDATE occurred between Tx.1 and Tx.3, POMVCC will update the timestamp. Even if the execution order of all transaction processes within the same timestamp is changed, POMVCC permits simultaneous execution if the content of the database is the same.

We show the allowable conditions of transaction processing on the same timestamp for MVCC and POMVCC in Table V, which shows that 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. We discuss the difference in isolation levels between MVCC and POMVCC, as shown in Figure 4.

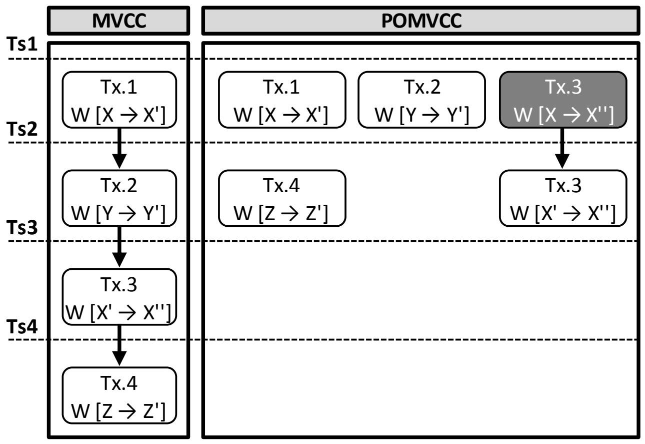


Figure 3. Difference between MVCC and POMVCC

TABLE IV. DEFINITION OF POMVCC

D4. POMVCC for write tx.	
CTs(Tx.i) ≦ CTs(Tx.j) <==> I or II	
I	CTs(Tx.i) < CTs(Tx.j)
II	CTs(Tx.i) = CTs(Tx.j) and DBC(Tx.i → Tx.j) = DBC(Tx.j → Tx.i)

TABLE V. ALLOWABLE CONDITIONS OF TRANSACTION PROCESSING ON SAME TIMESTAMP FOR MVCC AND POMVCC

Formula	MVCC	POMVCC
R1[X] R2[X]	**Success**	**Success**
R1[X] W2[X]	**Success**	**Success**
W1[X] R2[X]	**Success**	**Success**
W1[X] W2[Y]	Failure	**Success**
W1[X] W2[X]	Failure	Failure

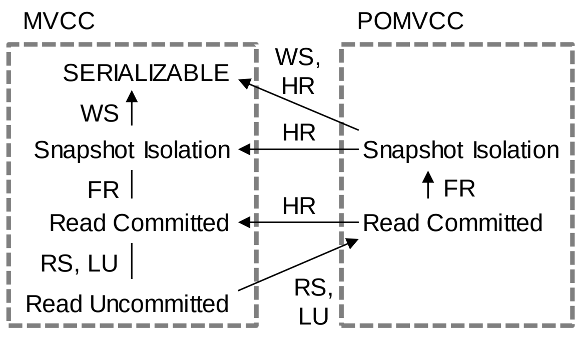


Figure 4. Diagram of isolation levels and relationships

B. *How to control POMVCC*

The trigger to update a timestamp of POMVCC differs from that of MVCC. MVCC updates a timestamp during the commit phase of a transaction, but POMVCC updates it during the abort phase of a transaction. Thus, multiple update transactions can be executed at the same timestamp on POMVCC.

The protocol of POMVCC is shown in Figure 5. The conflict of LOST UPDATE occurs between Tx.1 and Tx.3 on record X. In the case of MVCC, a timestamp is updated at the commit of Tx.1, but in the case of POMVCC, a timestamp is not updated. Therefore, Tx.3 refers to old record X, and conflict occurs. POMVCC updates a timestamp at the abort of Tx.3. Record X can be updated when Tx.3 is retried. Because a timestamp is updated at the abort of a transaction caused by an anomaly, partial order transaction control is possible.

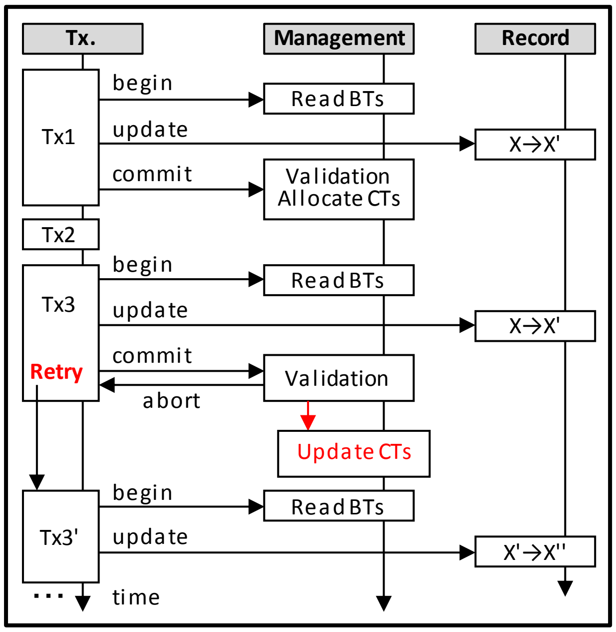


Figure 5. Concurrency control of POMVCC

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 an application or user. For example, consider that the succeeding transaction refers to the result of the preceding transaction. In this case, the HISTORICAL READ, 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 the application requires the result of the preceding transaction.

Table VI and Figure 6 provide the definition of HISTORICAL READ. The Tx.2 cannot refer to record X', which Tx.1 updates after the commit of Tx.1. This is a new anomaly. If Tx.1 and Tx.2 are independent transactions, such an anomaly does not occur. However, when the application assumes that the execution order is Tx.1 → Tx.2, an unexpected response occurs. This anomaly of HISTORICAL READ does not occur on MVCC.

TABLE VI. DEFINITION OF HISTORICAL READ

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

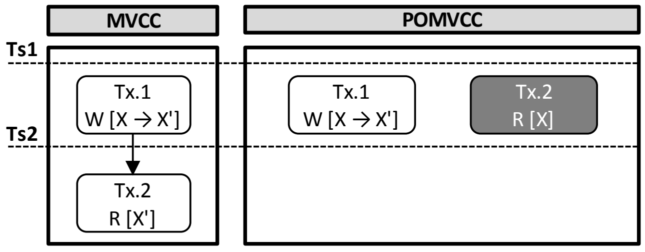


Figure 6. Anomaly of HISTORICAL READ

D. *How to avoid HISTORICAL READ*

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

The avoidance method for the same user (connection-based method) may include false positives. Figure 7 shows the solution of HISTORICAL READ for the connection-based approach. In the worst case, timestamps are updated at every commit. For example, timestamp updates are unnecessary in the independent transactions. However, in the connection-based method, timestamps are always updated during the begin phase of the transactions. As a result, performance degradation is a concern due to there being many false-positive cases.

With the avoidance method for the same application (request-based method), minimum increments of the timestamp, which would preferably be referred to, are set when the application issues transactions. This method can avoid HISTORICAL READ efficiently because false positives are excluded. However, the interface of a DBMS, such as begin and commit, must be modified, which is a disadvantage of this method. Figure 8 shows the solution of the connection-based method. POMVCC returns a CTs at the commit of Tx.1, and a BTs (= CTs) is set at the begin of Tx.2. As a result, Tx.1.CTs < Tx.2.BTs is established, and Tx.2 can refer to the execution result of Tx.1. We implemented the request-based method shown in Figure 8.

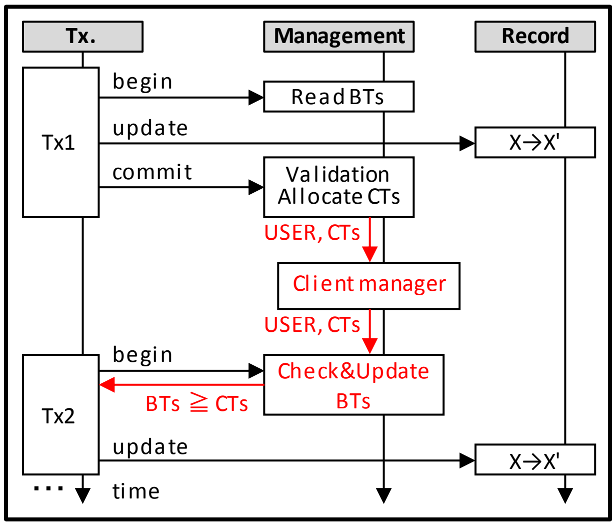


Figure 7. Solution of HISTORICAL READ on connection-based method

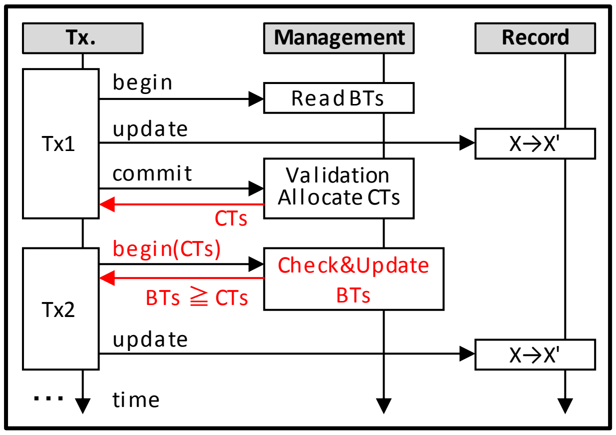


Figure 8. Solution of HISTORICAL READ on request-based method

V. IMPLEMENTATION OF MPDB

We developed an in-memory DBMS called "MPDB" to compare the performance of MVCC and POMVCC. We implemented MVCC and POMVCC on MPDB and evaluated their performance. MPDB is an MVCC-based, lock-free, in-memory DBMS characterized by parallel logs and mixed PCC/OCC [1][2][3]. In this section, we introduce the implementation of MVCC and POMVCC on MPDB.

A. *Technical issues*

From evaluating the breakdown of TPC-C to organize the DBMS issues on OLTP, buffering (30%), locking (29%) and logging (21%) accounted for 80% of the whole process [11] [26]. The buffering manages temporary data on a DBMS to achieve high-performance by reducing the number of storage accesses. Locking is mainly used for updating when maintaining DBMS consistency by transaction processing. Logging writes log sets to storage to make the transaction results persistent. We aimed to solve these problems on MPDB.

The number of CPU cores and memory size have been increasing. Although the number of cores per CPU has increased rapidly, the CPU frequency is converging to about 3 GHz [35]. Therefore, we must develop high-parallelism for improving performance of tps in line with the technical trend. The memory capacity is also increasing with the momentum exceeding DB size. The data set of OLTP is often several TB or less, and in-memory processing that does not acquire data from storage has become possible. Therefore, we developed an in-memory DBMS called "MPDB" for sustainable and high-performance DBMSs.

B. *Design overview*

MPDB implements MVCC-based architecture using lock-free on an in-memory DBMS for high-performance and high-scalability OLTP. We implemented lock-free control to avoid degradation of scalability on lock control due to the increased number of CPU cores.

Figure 9 shows a design overview of MPDB. Transaction processing is organized into three phases on MPDB. The first phase is the begin processing and the transaction processing of read and write. The DBMS allocates a timestamp for reference to the transaction during the begin phase and the transaction reads/writes the records using a BTs. The second phase is the validation phase during the commit phase. The processing details are given in Sections V.D and VI. The third phase is the durability phase during the commit phase. The DBMS writes log sets to storage to make the transaction results persistent.

During in-memory processing, client communication and log processing increase in proportion to performance, and interrupt handling becomes a bottleneck. However, load balancing is easy for client communication. The load of interrupt handling can be generally distributed by Receive Side Scaling (RSS) or "*irqbalance*". The number of interrupts can be reduced by changing the interrupt handling to polling processing. Log processing must manage the log file sequentially to guarantee the ACID. However, it is not necessary to manage log files physically in one dimension along the time series. A one-dimensional log file is sufficient to produce a logical log file at recovery. Therefore, MPDB implements a mechanism that allows log processing to be executed in parallel by the assigned TxID and timestamp to the transaction log. We implemented asynchronous input/output (I/O) using "*libaio*" for efficient log processing [36].

The group commit may be a cause of hindering the scalability of log management. We do not implement group commit since random write performance does not become a bottleneck due to the appearance of a high-performance storage such as a solid-state drive or storage class memory.

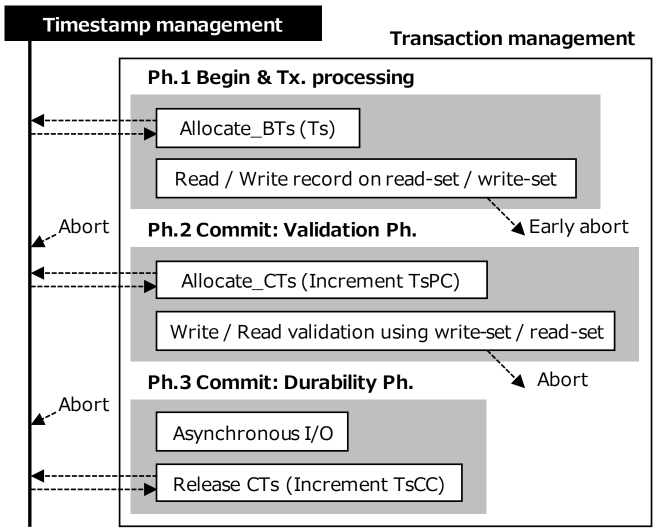

Timestamp management
Transaction management
Ph.1 Begin & Tx. processing
Allocate_BTs (Ts)
Read / Write record on read-set / write-set
Abort
Early abort
Ph.2 Commit: Validation Ph.
Allocate_CTs (Increment TsPC)
Write / Read validation using write-set / read-set
Ph.3 Commit: Durability Ph.
Abort
Abort
Asynchronous I/O
Release CTs (Increment TsCC)

Figure 9. Design orverview of MPDB. Ts = Timestamp

C. Data structure

We have to construct the data structure considering the non-uniformity of memory latency by Non-Uniform Memory Access (NUMA) [37]. We divide the data allocation into thread areas, i.e., local areas and a global area in consideration of NUMA on MPDB. Figure 10 shows the overview of data allocation on MPDB. As a premise, MPDB allocates threads of transaction processing to CPU cores.

The thread area manages a work area and log area for transaction processing. Each thread has its own thread area to execute transactions and references/updates another thread area when it executes the validation process, but this is infrequent. Therefore, the thread area should be built in the local area.

We assigned a local area for each CPU. A local area has log-management information to perform log processing for each CPU and is used to expand the thread area.

MPDB assigns common data, such as tables, indexes, and system information, with no locality in the global area. It creates the global area by the NUMA option of "—interleave" to allocate this area and multiple memory to load balance the memory access.

Figure 11 shows the detailed structure of the tables and B-tree index on MPDB. We adopted the linked list for all data structures to implement a lock-free DBMS. MPDB inserts records to update/delete/insert the records for MVCC. We define the rows of the table as a record and the record of update history as a row.

The B-tree index includes nodes and edges. The nodes are arranged in descending order, and edges are arranged in ascending order. MPDB enables bidirectional search by using this index structure. This structure is lock-free since it is made of the linked list.

MPDB also allows the possibility that the index does not refer to the latest record to enable early commit. As shown in Figure 11, transaction processing does not positively change the record pointer of the leaf edge to the pointer of new record when delete Row.1', so that index.col.2 does not necessarily indicate the latest Row.1''. We define this processing method as LATE UPDATE. Therefore, the thread can shorten the serialization point and improve scalability during the commit phase. However, the thread changes the pointer of the record to the latest pointer when referring with the index on LATE UPDATE. The thread can reduce the number of chains of the linked list and achieve fast record access.

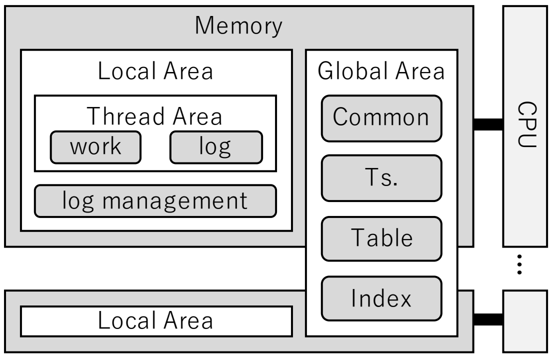


Figure 10. Overview of data structure on MPDB

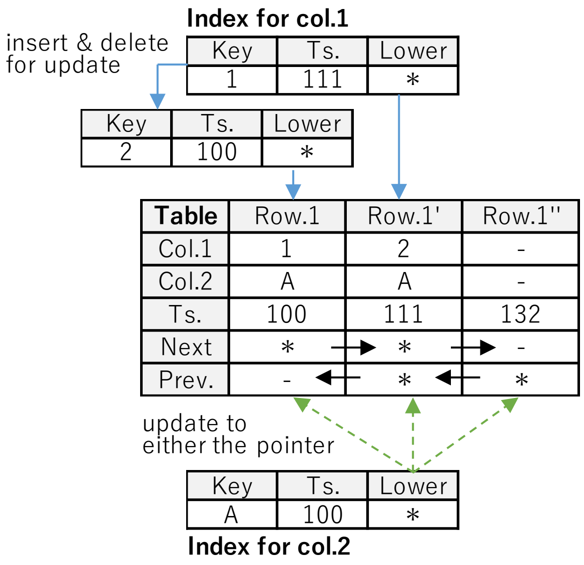


Figure 11. Detailed structure of tables and indexes

D. Transaction management

We now explain the procedure of transaction processing. The state of the transaction is illustrated in Figure 12. MPDB manages the four typical states of transactions. The transaction states can be classified into ACTIVE during the begin phase, PRE-COMMIT at the validation phase during the commit phase, COMMIT at the durability phase during the commit phase and ABORT during the abort phase, as shown in Figures 9 and 12.

Table VII shows the transaction state for each transaction method on MPDB. MPDB implemented mixed OCC/PCC to provide six transaction methods. Generally, long transactions are easily aborted by short transactions; therefore, long transactions can reduce the frequency of the abort when short transactions set OCC and long transactions set PCC.

OCC and PCC are illustrated in Figures 13 and 14, respectively. On OCC, the initial state of the transaction is ACTIVE and the database performs begin processing in the first phase and commit processing in the second–fifth phases. However, on PCC, the initial state of the transaction is PRE-COMMIT and the database performs begin processing in the first phase and the commit processing in the second and third phases. The processing equivalent to write lock is executed with the third phase on OCC and first phase on PCC. That is, since threads can execute write lock during transaction processing on PCC, it is possible to perform record update reservation earlier than OCC. Because of this, MPDB enables the coexistence of long and short transactions.

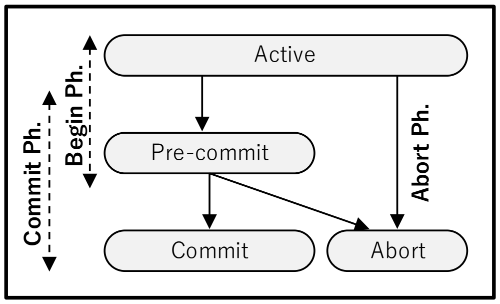


Figure 12. States of transaction

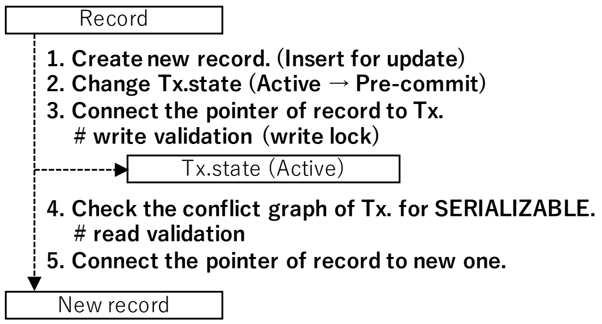

Record
1. Create new record. (Insert for update)
2. Change Tx.state (Active → Pre-commit)
3. Connect the pointer of record to Tx.
# write validation (write lock)
Tx.state (Active)
4. Check the conflict graph of Tx. for SERIALIZABLE.
# read validation
5. Connect the pointer of record to new one.
New record

Figure 13. Tx. processing on OCC

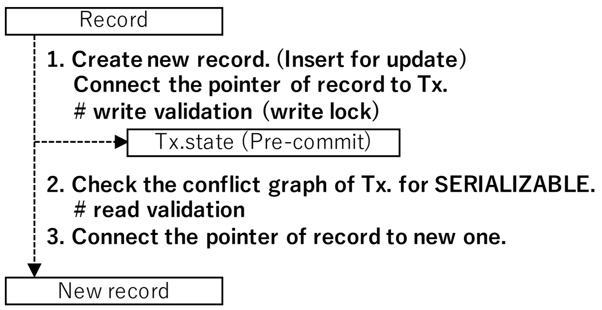

Record
1. Create new record. (Insert for update)
Connect the pointer of record to Tx.
# write validation (write lock)
Tx.state (Pre-commit)
2. Check the conflict graph of Tx. for SERIALIZABLE.
# read validation
3. Connect the pointer of record to new one.
New record

Figure 14. Tx. processing on PCC

TABLE VII. MEHTODS OF TRANSACTION PROCESSING

ISOLATION	BTs. Allocation	Tx.state
PCC.RC	each SQL	state(pre-commit) at begin
PCC.SI	each Tx.	state(pre-commit) at begin
PCC.SERIALIZABLE	each Tx.	state(pre-commit) at begin
OCC.RC	each SQL	state(active) at begin, state(pre-commit) at validation
OCC.SI	each Tx.	state(active) at begin, state(pre-commit) at validation
OCC. SERIALIZABLE	each Tx.	state(active) at begin, state(pre-commit) at validation

VI. IMPLEMENTATION OF POMVCC

The lock used in parallel processing may degrade scalability [6]. In this section, we introduce a lock-free implementation for scalable POMVCC to reduce this degradation.

A. Implementation

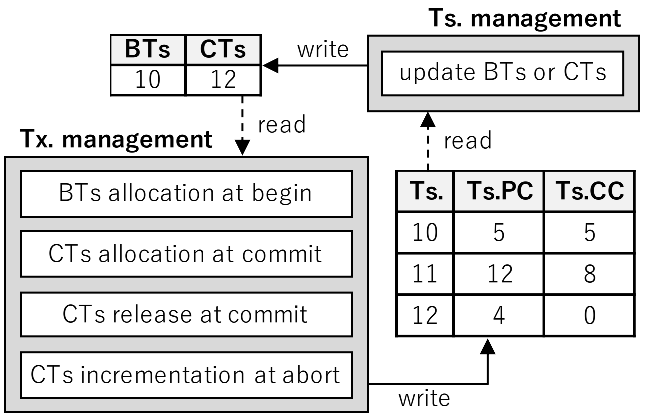
We implemented POMVCC to solve the problem of critical section. Previously, the critical section is that the transaction increments a CTs, adapts the CTs to the newest versions and unlocks it during the commit phase. Therefore, Tx.2 waits until the end of Tx.1 to allocate the CTs. Therefore, we divide a timestamp into a BTs and CTs to solve this problem. This is similar to speculative execution. A BTs is the timestamp used for referring to a record. This technique is very common. Table VIII and Figure 15 show the timestamp management and data structure on POMVCC.

We solve the problem of lock for scalability. Generally, transactions increment a CTs during the commit phase in parallel. Therefore, the lock is necessary to obtain the sequential and unique CTs on MVCC. However, POMVCC does not require a unique CTs. That is, a transaction does not increment a CTs during the commit phase on POMVCC. The transaction manager reads a CTs, and the timestamp manager updates it, as shown in Figure 15. On POMVCC, timestamp control is divided into a read process by the transaction manager and a write process by the timestamp manager for lock-free.

Finally, the commit phase is divided into pre-commit and commit. The DBMS must manage committed transactions at the same timestamp on partial order for consistency. Therefore, MPDB implements double counters to manage the state of many transactions at each timestamp. The double counters are Pre-commit Counter at each Timestamp (Ts.PC) and Commit Counter at each Timestamp (Ts.CC). The DBMS can determine the transaction state from the difference between the Ts.PC and Ts.CC. We show the commit process as follows. The Tx.1 reads a CTs, increments the Ts.PC of the CTs, and adapts the CTs to the newest versions of record during the pre-commit phase. It then increments the Ts.CC of the CTs when the log is completed during the commit phase. Then, Tx.2 does not wait for Tx.1 to execute the commit process. Therefore, POMVCC is highly scalable. Strictly, the atomic processing has critical section for incrementing the Ts.PC or Ts.CC; however, it is very short. The timestamp manager can increment a BTs or CTs anytime when it has detected an anomaly or requirement. For example, if the Ts.PC and Ts.CC are the same, the timestamp manager updates a RTs. That is, the record can be referred to by using this timestamp while maintaining consistency. Table VIII lists the timestamp-management rules on POMVCC.

TABLE VIII. TIMESTAMP-MANAGEMENT RULES

D5. CTs management	
CTs[a] → CTs[a+1] <==> i or any time	
i	DB(Tx.i → Tx.j) ≠ DB(Tx.j → Tx.i)
D6. BTs management	
BTs[b] → BTs[b+1] <==> I and II	
I	BTs[b+1] < CTs[b+1]
II	Ts.PC[b+1] = Ts.CC[b+1]



Ts.	Ts.PC	Ts.CC
10	5	5
11	12	8
12	4	0

Figure 15. Timestamp management and data structure on POMVCC

B. Log management

Table IX lists the general log-management rules. We define the I/O completion as Completion (Comp). Log management must complete the transaction processing of all CTs (10) if it can complete transaction processing of CTs (11), as shown in Table IX. This rule corresponds to the general cascading protocol for recovery processing.

As a result of separating timestamps into BTs and CTs, this rule became unnecessary on MPDB because the BTs manager guarantees that all readable records persisted, as shown in Table VIII. Log management does not need to control the log execution order, so it can maintain high-scalability.

TABLE IX. LOG-MANAGEMENT RULES

D7. Log management
Comp(Ts[a]) → Comp(Ts[a+1]) <==> ∀a (Logged(Ts[a]) ≦ Logged(Ts[a+1])

C. Interface of request-based approach

Figures 16, 17, 18 and 19 illustrate POMVCC. Figure 16 and 17 show the user interface with which a user requests begin, commit or abort to the DBMS, and Figure 18 and 19 show the timestamp interface with which the transaction thread requests any timestamp allocation.

The thread performs the initialization of the data structure and numbering of a BTs during the begin phase. At this time, if the user instructs a transaction to use a timestamp, the timestamp manager increments a CTs up to Ts + 1 and increments the BTs up to the timestamps at Allocate_BTs. The timestamp manager stores the transaction-history log when it increments a CTs.

The thread changes the transaction state from ACTIVE to PRE-COMMIT and allocates a CTs from the timestamp manager. When allocating the CTs, the thread increments the Ts.PC to determine the number of transaction processes in the CTs. After that, the thread changes the transaction state from PRE-COMMIT to COMMIT through the validation phase. If the transaction state is COMMIT, the thread stores the log and increments the Ts.CC. If the transaction state is ABORT, the thread decrements the Ts.PC through the abort phase. After completion of the commit phase, the thread provides the result and the CTs to the user.

The thread performs the initialization of the data structure for aborting and incrementing the CTs during the abort phase and provides the result and CTs to the user because the thread increments the CTs to avoid the abort due to refer/update conflict. A transaction must increment a BTs after incrementing a CTs to avoid conflict. Therefore, the user gives the CTs during the begin phase during the retry process. Thus, the transaction can at least avoid the same conflict problem as the previous one.

Finally, the timestamp manager updates the BTs and CTs periodically and asynchronously with transaction processing. This solves the problem in which a user cannot reference update records even after a long time.

```
// DBMS aborts the Tx.
AbortTx ( ) {
  ・・・ abort phase ・・・
  if ( /*DBMS identifies the cause of Ts. on abort.*/ )
    CTs = Update_CTs ( ) ;
  return ( CTs ) ;
}
```

Figure 16. POMVCC interface 1

```
// DBMS begins the Tx.
BeginTx ( Ts ) {
  ・・・ begin phase ・・・
  BTs = Allocate BTs ( Ts ) ;
  return () ;
}

// DBMS commits the Tx.
CommitTx ( ) {
  Change Tx.state ( Pre-commit ) ;
  CTs = Allocate CTs ( ) ;
  ・・・ write validation phase ・・・
  ・・・ read validation phase ・・・
  Change Tx.state ( Commit / Abort ) ;
  if ( Tx.state = Commit ) { // DBMS can commit the Tx.
    ・・・ durable phase ・・・
    Increment_TsCC ( CTs ) ;
  } else if ( Tx.state = Abort ) { // DBMS detects the Anomaly.
    CTs = AbortTx ( ) ;
    Decrement_TsPC ( CTs ) ;
  }
  return ( CTs ) ;  // Ts. for historical read
}
```

Figure 17. POMVCC interface 2

```
// Tx. is allocated the BTs. at begin for read.
Allocate BTs ( Ts ) {
  CTs = Read_CTs ( ) ;
  while ( CTs ≦ Ts ) {
    CTs = Update_CTs ( ) ;
  }
  do {
    BTs = Update_BTs ( ) ;
  } while ( BTs < Ts ) ;
  return ( BTs ) ;
}

// Tx. is allocated the CTs. at commit
Allocate CTs ( ) {
  atomic {
    CTs = Read_CTs ( ) ;
    Increment_TsPC ( CTs ) ;
  }
  return ( CTs )
}

// This function updates the CTs.
Update_CTs ( ) {
  CTs = Increment_CTs () ;
  Log_CTs ( CTs-1, Read_TsPC (CTs-1) ) ;
  return ( CTs ) ;
}
```

Figure 18. Ts-management interface 1

```
// This function checks & updates the BTs.
Update_BTs ( ) {
  BTs = Read_BTs ( ) ;
  CTs = Read_CTs ( ) ;
  // It reads the Ts.Pre-commit Counter (Ts.PC).
  PC = Read_TsPC ( BTs + 1 ) ;
  // It reads the Ts.Commit Counter (Ts.CC).
  CC = Read_TsCC ( BTs + 1 ) ;
  if ( BTs < CTs - 1 && PC = CC )
    BTs = Increment_BTs () ;
  return ( BTs ) ;
}
```

Figure 19. Ts-management interface 2

VII. EVALUATION OF PROTOTYPE IMPLEMENTATION

In this section, we compare the performance of MVCC and POMVCC. We implemented MVCC and POMVCC on MPDB and evaluated their performance. In this experiment, we used the industry standard benchmark TPC-C and repeatedly executed the stored procedure calls that model New Order [34].

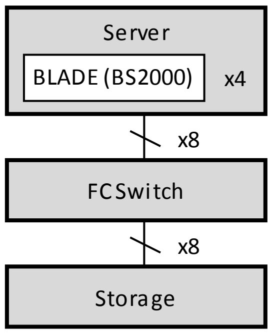
A. Experimental Environment

Figure 20 depicts the system configuration. Four blade servers were used, i.e., symmetric multiprocessors, and had 8 CPUs (80 cores), 1 TB of memory, and 8 ports of an 8-Gb Fiber Channel (FC). The servers and storage were connected via an FC switch and communicated 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.

For the MPDB settings, one thread was assigned to one core. This means that MPDB used a maximum of 80 threads. One log file was assigned to one CPU to load balance the 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 the DB was 0.72 GB. The item, stock, and order_line tables were used in TPC-C. Indexes were also 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 Memory
Disk	6.4TB (1.6TB x 4) Hitachi Accelerated Flash
RAID	RAID5 (3D + 1P)

Figure 20. System Configuration

B. Workload

The workload shown in Figure 21 was created on the basis of TPC-C's New Order. The workload simulates the repeatedly executing part of New Order. The processing in Figure 21 was repeated ten 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 21. Experiment Workload

C. Experimental Results and Consideration for *MPDB*

We evaluated MPDB before evaluating POMVCC. We did not use POMVCC to evaluate the basic performance of MPDB. MPDB has various mechanisms but the one most contributing to performance improvement is log parallelization. Therefore, we verified the effects of performance and scalability using parallel log processing. We compared single log processing and parallel log processing and measured the performance and scalability of DBMS with increasing CPU for each log processing.

We compared the performance of single log processing and parallel log processing corresponding to the number of threads. In Figure 22, the x-axis represents the number of threads, and the y-axis represents transactional performance (tps). The performance of parallel log processing increased as the number of threads increased. However, the performance of single log processing decreased as the number of threads increased more than 40 threads. We confirmed that parallel log processing can perform 5.02 times better than single log processing. We also found that I/O interrupt is focused on a specific CPU core by analyzing single log processing with "*mpstat*" of Linux, as shown in Figure 23. In this figure, the x-axis represents the id of CPU core (0–79), and the y-axis represents time. We confirmed that parallel log processing distributes the load of I/O interrupt.

We also compared the scalability of single log processing and parallel log processing corresponding to the number of threads. In Figure 24, the x-axis represents the number of threads, and the y-axis represents the performance rate on Figure 22 when the performance at 10 threads was assumed as that at 100. In single log processing, the scalability suddenly deteriorated at 40 threads. However, parallel log processing maintained scalability degradation at less than 15% even with 80 threads.

We confirmed that if the number of CPUs exceeds 2, it is necessary to parallelize log processing.

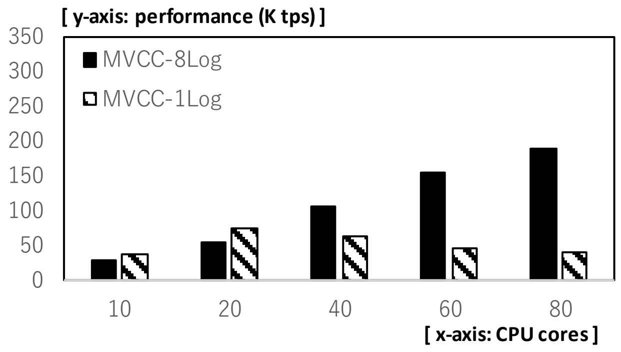
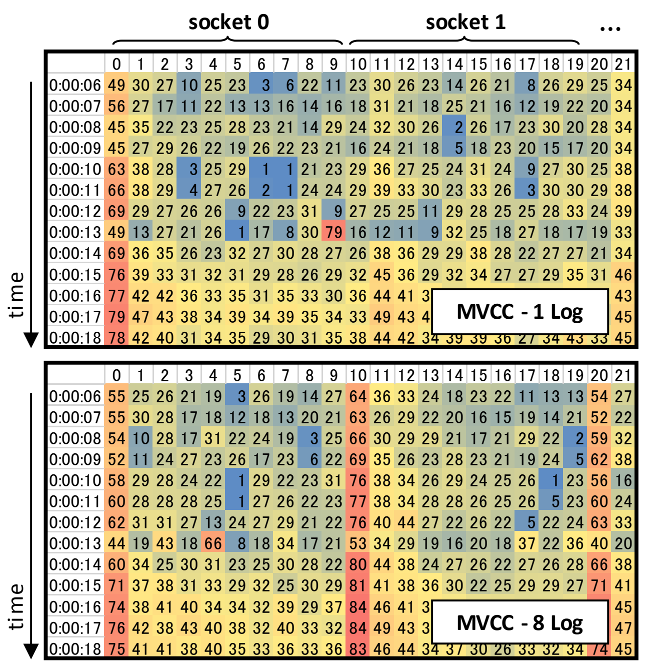


Figure 22. Performance evaluation for log processing


socket 0
socket 1
...
time
MVCC - 1 Log
MVCC - 8 Log

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0:00:06	49	30	27	10	25	23	3	6	22	11	23	30	26	23	14	26	21	8	26	29	25	34
0:00:07	56	27	17	11	22	13	13	16	14	16	18	31	21	18	25	21	16	12	19	22	20	34
0:00:08	45	35	22	23	25	28	23	21	14	29	24	32	30	26	2	26	17	23	30	20	28	34
0:00:09	45	27	29	26	22	19	26	22	23	21	16	24	21	18	5	18	23	20	15	17	20	34
0:00:10	63	38	28	3	25	29	1	1	21	23	29	36	27	25	24	31	24	9	27	30	25	38
0:00:11	66	38	29	4	27	26	2	1	24	24	29	39	33	30	23	33	26	3	30	30	29	38
0:00:12	69	29	27	26	26	9	22	23	31	9	27	25	25	11	29	28	25	25	28	33	24	39
0:00:13	49	13	27	21	26	1	17	8	30	79	16	12	11	9	32	25	18	27	18	17	19	33
0:00:14	69	36	35	26	23	32	27	30	28	27	26	38	36	29	29	38	28	22	27	27	21	34
0:00:15	76	39	33	31	32	31	29	28	26	29	32	45	36	29	32	34	27	27	29	35	31	46
0:00:16	77	42	42	36	33	35	31	35	33	30	36	44	41	3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	43
0:00:17	79	47	43	38	34	39	34	39	35	34	33	49	43	4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	45
0:00:18	78	42	40	31	34	35	29	30	31	35	38	44	42	34	39	39	36	27	34	43	33	45

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0:00:06	55	25	26	21	19	3	26	19	14	27	64	36	33	24	18	23	22	11	13	13	54	27
0:00:07	55	30	28	17	18	12	18	13	20	21	63	26	29	22	20	16	15	19	14	21	52	22
0:00:08	54	10	28	17	31	22	24	19	3	25	66	30	29	29	21	17	21	29	22	2	59	32
0:00:09	52	11	24	27	23	26	17	23	6	22	69	35	26	23	28	23	21	19	24	5	62	38
0:00:10	58	29	28	24	22	1	29	22	23	31	76	38	34	26	29	24	25	26	1	23	56	16
0:00:11	60	28	28	28	25	1	27	26	22	23	77	38	34	28	28	26	25	26	5	23	60	24
0:00:12	62	31	31	27	13	24	27	29	21	22	76	40	44	27	22	26	22	5	22	24	63	33
0:00:13	44	19	43	18	66	8	18	34	17	21	53	34	29	19	16	20	18	37	22	36	40	20
0:00:14	60	34	25	30	31	23	25	30	28	22	80	44	38	24	27	26	22	27	26	28	66	38
0:00:15	71	37	38	31	33	29	32	25	30	29	81	41	38	36	30	22	25	29	29	27	71	41
0:00:16	74	38	41	40	34	34	32	39	29	37	84	46	41	3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	45
0:00:17	76	42	38	43	40	38	32	40	33	32	84	49	43	3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	47
0:00:18	75	41	41	38	40	35	33	36	33	36	83	46	44	34	37	30	26	33	32	34	74	45

Figure 23. Load of I/O interrupt each CPU core

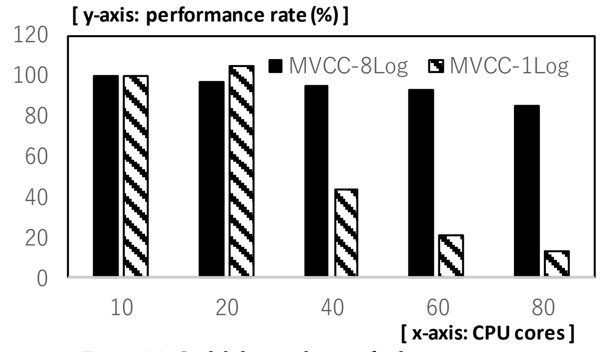


Figure 24. Scalability evaluation for log processing

D. *Experimental Results and Consideration for POMVCC*

We compared the performance of MVCC (8 logs) and POMVCC (8 logs) corresponding to the number of threads. In Figure 25, the x-axis represents the number of threads, and the y-axis represents tps on increasing conflict rate. 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 conducted an experiment in which the number of warehouses changed corresponding to the number of threads. That is, the number of warehouses was ten (DB size was 0.45 GB) when the number of threads was ten and the one was 80 (DB size was 3.61 GB) when the one was 80. The respective experimental results show Figures 26 and 27.

In Figure 26, the x-axis represents the number of threads, and the y-axis represents tps on a fixed conflict rate. The performance of POMVCC on a fixed conflict rate (Figure 26) is higher than the performance on increasing conflict rate (Figure 25). POMVCC on a fixed conflict rate was 1.34 times faster than one on increasing rate. However, MVCC exhibited almost the same performance regarding increasing conflict rate (Figure 25) and regarding the fixed conflict rate (Figure 26). Therefore, POMVCC was 1.63-1.74 times faster than MVCC.

We then compared the scalability of MVCC and POMVCC corresponding to the number of threads at a fixed conflict rate. In Figure 27, the x-axis represents the number of threads, and the y-axis represents the performance rate on Figure 26 when the performance at 10 threads was assumed as that at 100. The scalability of both MVCC and POMVCC slowly decreased as the number of threads increased. The scalability coefficient of MVCC was 87.98–97.96% and that of POMVCC was 94.02–98.32%. POMVCC improved by 6.87% compared with MVCC. This experiment suggests 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 number of threads. When 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.

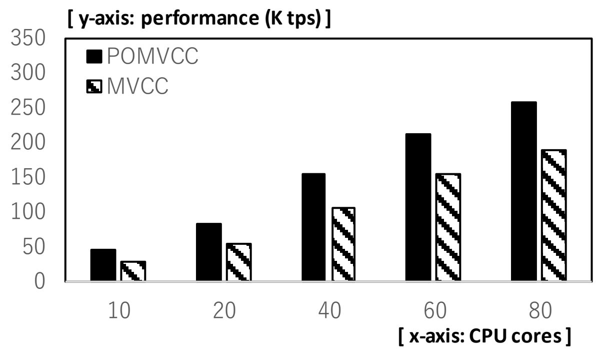


Figure 25. Performance evaluation regarding increasing conflict rate

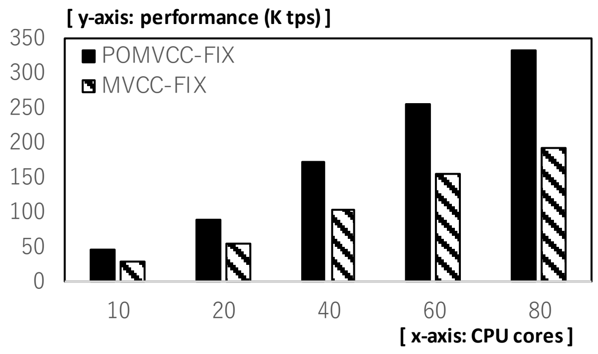


Figure 26. Performance evaluation regarding fixed conflict rate

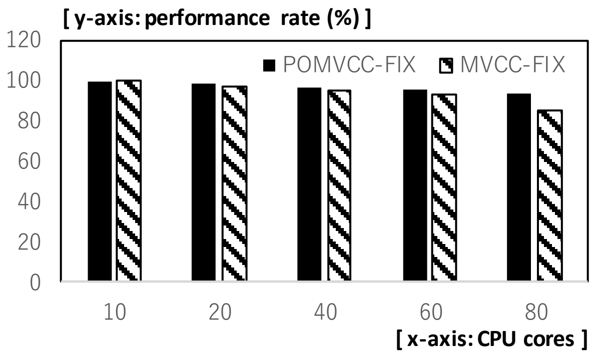


Figure 27. Scalability evaluation regarding fixed conflict rate

VIII. CONCLUSION AND FUTURE WORK

We proposed POMVCC, which maintains the protocol of MVCC and improves performance and scalability of DBMS. POMVCC is focused on the partial order of transactions. The conventional technique provides a timestamp to each transaction, but POMVCC provides a timestamp to multiple transactions. POMVCC reduces the number of timestamps that are updated and improves performance and scalability of DBMS. We discussed the difference in isolation levels between MVCC and POMVCC, as shown in Figure 4.

We implemented and evaluated POMVCC on an in-memory DBMS we developed called “MPDB”, which is an MVCC-based, lock-free, in-memory DBMS that is characterized by parallel logs and mixed PCC/OCC.

We first compared the performance and scalability of MPDB corresponding to the number of threads. The results indicate that the most contributing mechanism to performance improvement was log parallelization. Parallel log processing maintains scalability degradation of less than 15% even with 80 threads. We confirmed that if the number of CPUs exceeds 2, it is necessary to parallelize the log processing.

We then compared the performance and scalability of MVCC and POMVCC corresponding to the number of threads regarding an increasing conflict rate. The performance of POMVCC was 1.36–1.60 times better than that of MVCC. We also compared the performance of MVCC and POMVCC regarding a fixed conflict rate. POMVCC was 1.63–1.74 times faster than MVCC. The scalability coefficient of MVCC was 87.98–97.96% and that of POMVCC was 94.02–98.32%. The performance of POMVCC improved by 6.87 % compared with MVCC.

The scalability coefficients of POMVCC and MVCC depended on the size of the DB and number of threads. When 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.

We implemented POMVCC on MPDB and evaluated it by using SNAPSHOT ISOLATION, for which POMVCC performed better than MVCC. However, the performance trend was unclear because the probability of WRITE SKEW increased on SERIALIZABLE. 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. It is also possible that RW-CONFLICT GRAPH will increase and a large cyclic graph will be created. Therefore, our future work is to implement and evaluate POMVCC by using SERIALIZABLE.

REFERENCES

[1] Y. Isoda, A. Tomoda, T. Tanaka, and K. Mogi, "Partial Order Multi Version Concurrency Control," DBKDA 2018, The Tenth International Conference on Advances in Databases, Knowledge, and Data Applications, May 2018.

[2] 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).

[3] 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).

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] J. Gray, and A. Reuter, "Transaction Processing: Concepts and Techniques," Elsevier, 1992.

[11] 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.

[12] 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.

[13] 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.

[14] D. L. Mills, "Internet time synchronization: the network time protocol," IEEE Transactions on Communications, Volume 39, Issue 10, October 1991.

[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] 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.

[17] Hewlett Packard, "Memory-Driven Computing," https://news.hpe.com/content-hub/memory-driven-computing/, November 2018.

[18] H. Lim, D. Han, D. G. Andersen, and M. Kasminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage," NSDI '14, pp. 429-444, April 2014.

[19] A. Pavlo, "What are we doing with our lives?," SIGMOD '17 Keynote, http://www.cs.cmu.edu/~pavlo/slides/pavlo-keynote-sigmod2017.pdf, November 2018.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database System," 1987.

[28] ORACLE, "Oracle Database 12c Release 2," https://docs.oracle.com/en/database/oracle/oracle-database/12.2/index.html, November 2018.

[29] MySQL, "MySQL 5.7 Reference Manual," https://dev.mysql.com/doc/refman/5.7/en/, November 2018.

[30] PostgreSQL, "PostgreSQL 9.6.10 Documentation," https://www.postgresql.org/docs/9.6/static/index.html, November 2018.

[31] 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.

[32] 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.

[33] 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.

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

[35] J. L. Hennessy, and D. A. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers.

[36] E. P. C. Jones, D. J. Abadi, and S. Madden, “Low Overhead Concurrency Control for Partitioned Main Memory Databases,” SIGMOD '10 Proceedings, pp. 603-614, June 2010.

[37] D. Levinthal, “Tutorial: Intel Core i7 and Intel Xeon 5500 Microarchitecture, Optimization and Performance Analysis,” 2010 IEEE International Symposium on Performance Analysis of Systems and Software, White Plains, NY, 2010.