2016 IEEE 23rd International Conference on High Performance Computing

Read Consistency in Distributed Database Based on DMVCC

Jie Shao[†§], Boxue Yin[§], Bujiao Chen[§], Guangshu Wang[§], Lin Yang[§]
Jianliang Yan[§], Jianying Wang[§], Weidong Liu[†]
[†]Tsinghua University [§]Baidu, Inc
[†]shao-j14@mails.tsinghua.edu.cn [†]liuwd@mail.tsinghua.edu.cn
[§]{yinboxue,chenbujiao,wangguangshu,yanglin05,yanjianliang,wangjianying}@baidu.com

***Abstract*—In a traditional distributed database system, the partitions use two-phase locking (2PL) as the concurrency control protocol to ensure distributed read consistency. But the read-lock acquired by a read operation is incompatible with a write-lock, which undermines the performance of the system. While in a system at the snapshot isolation level, where partitions use Multi-Version Concurrent Control (MVCC) as the concurrent control protocol, distributed read inconsistency may occur. To achieve read consistency and guarantee the performance at the same time, we propose Distributed Multi-Version Concurrent Control (DMVCC). With DMVCC, the system can support snapshot reads, which do not block write operations, and ensure distributed read consistency. In this protocol, a transaction obtains a set of consistent snapshot version numbers at the startup time. The transaction then uses those numbers to read the corresponding data stored on each partition. The correctness of the protocol is strictly proved.**

We conduct a series of experiments to compare the performance of the system when using and not using DMVCC with a scaled TPC-C benchmark. We observe that our DMVCC based system outperforms the system using 2PL at both medium (up to 1.53x speed up) and high contention (up to 2.0x speed up) levels. Furthermore, when read/write ratio goes up to 1:1, the throughput of the DMVCC based system is 290% higher than that of the system using 2PL. The scalability of the system is also presented.

I. Introduction

As the data increase, many large-scale services in Baidu such as Baidu Wallet can no longer store data in a single database. Being a Chinese counterpart of PayPal, Baidu Wallet relies on a distributed on-line transaction processing (OLTP) system as its storage backend. OLTP systems require concurrency control to guarantee consistency[6], [7], so that services running on top of them can function correctly. Without right concurrency control, Baidu Wallet could transfer more money than there is from the account, execute the transfer twice, transfer the wrong amount of money, or present the wrong balance after a transaction.

While concurrency control is a well-studied field concerning single databases, the performance of protocols such as two-phase locking (2PL)[6] is limited with high-contention workloads, especially when the database receives long read-only transactions. To solve this problem, Multi-Version Concurrency Control (MVCC)[7], [11], [14] is proposed. For read operations, a client is allowed to read historical data to avoid read-write conflicts. This improve the performance intensely[16].

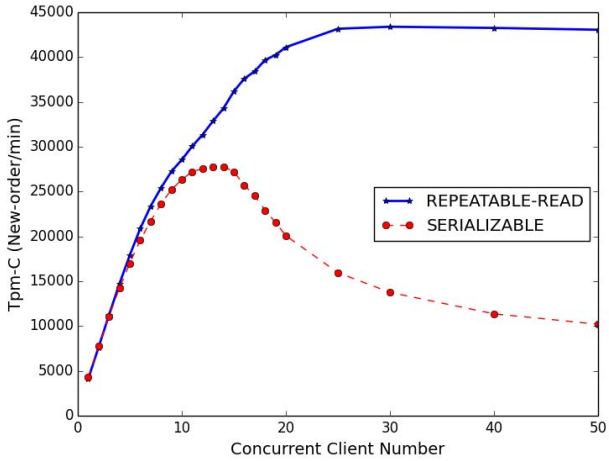
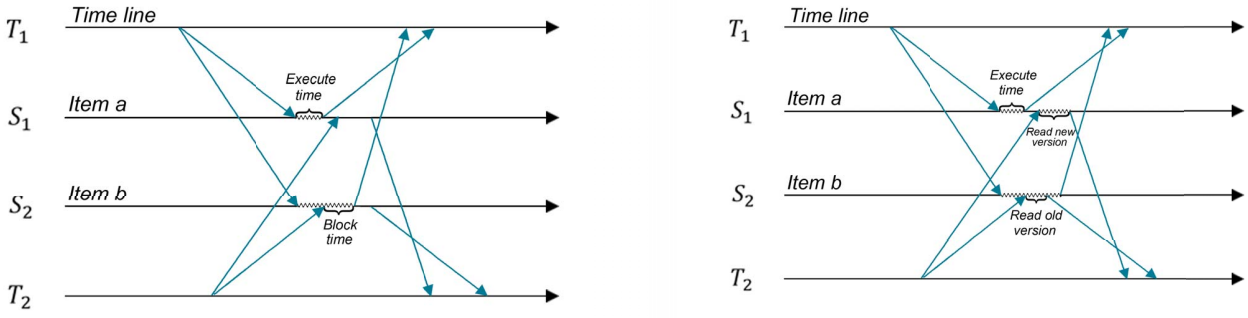


Figure 1: The throughput of a single database at different isolation levels as the client number increases

To prove that, we set up a simple experiment that compares the performances of a single database at different isolation levels. In this experiment, we use MySQL[1] as our database, which has different concurrency control methods. 2PL is used at the serializable isolation level, while MVCC is used at the repeatable read level. TPC-C[2] is used as our benchmark. The database contains 5 warehouses. The experimental setup remains the same in Section V. Figure 1 shows the results:

- When the number of clients is less than 10, the performance of the system remains almost unchanged at the repeatable read level and at the serializable level since there is little contention.
- As the number of clients increases, the throughput of the database at the serializable isolation level drops sharply. Meanwhile, the throughput of the system at the repeatable read level almost remains the same. That is to say the drop is caused by read and write conflicts instead of resource limitation.

978-1-5090-5411-4/16 $31.00 © 2016 IEEE
DOI 10.1109/HiPC.2016.11



(a) partitions using 2PL

(b) partitions using MVCC

Figure 2: How transactions read data using 2PL and using MVCC

As Figure 1 shows, if the distributed database uses MVCC as the concurrency control protocol, the system may have a better performance. Unfortunately, the distributed system requires that the transactions use 2PL in every partition[1][1].

A MVCC-based distributed database system at the non-serializable level leads to distributed read inconsistency, which refers to the circumstance where only some of the updates of a distributed transaction are visible to other transactions. We will explain that in detail in Section II. Many transactions in Baidu Wallet such as account checking and statistic analysis do not need to read the latest data. But a global consistent snapshot is still needed. So in this paper we propose DMVCC (Distributed Multi-Version Concurrency Control), a distributed concurrency protocol that ensures distributed read consistency and the performance at the same time.

DMVCC is a two-phase protocol based on two-phase commit (2PC)[6], [19]. A set of distributed transaction managers (DTMs) run the protocol on behalf of clients. At the beginning of a transaction, the DTM connects with the consistency coordinator to obtain global consistent snapshot version numbers. In the prepare phase, the transaction uses the snapshot version number to read data items stored on each partition. In the commit phase, the DTM collects the snapshot versions from each partition and sends them to the consistency coordinator. The consistency coordinator then calculates the new global consistent snapshot version. With DMVCC, the system guarantees that all or none of the updates of a distributed transaction is visible to other transactions.

The rest of the paper proceeds as follows. Section II presents a case of distributed read inconsistency and the way to avoid it. Section III elaborates the design of a DMVCC-based system. In Section IV, we prove the correctness of DMVCC. Section V sets up the experiments and presents the results of our experimental evaluation. Then we discuss the related work in Section VI and present the conclusion in section VII.

[1]In our design, we call each database a partition.

II. OVERVIEW

This section begins with a review of why we can not use MVCC in a distributed system. Then we will explain how the distributed multi-version concurrency control works.

A. A case of distributed read inconsistency

Application programmers usually prefer the highest isolation level to simplify the reasoning of correctness in the face of concurrent transactions[6], [17]. To guarantee distributed consistency, a distributed transaction runs standard concurrency control schemes such as standard 2PL combined with two-phase commit (2PC)[6], [8], [9].

To give a more intuitive explanation, we demonstrate a simplified process of money transferring in Baidu Wallet, during which the system reads the balance data from the account to ensure consistency. As is shown below, there are two distributed transactions T_1 and T_2 and two partitions S_1 and S_2, which contain data items a and b respectively. In transaction T_1, account *A* transfers 10 dollars to account *B*. Transaction T_2 then reads the data of the current account balance of *A* and *B*.

T_1:	UPDATA a = a - 10	UPDATA b = b + 10
T_2:	READ a	READ b

Firstly, let us see how distributed transactions work on each partition which uses standard 2PL as the concurrency control protocol and 2PC as the distributed commit protocol. Suppose partition S_1 and S_2 receive T_1 and T_2's subtransactions sequentially. The process is shown in Figure 2(a). We can see that T_2 arrives when T_1 is committed on partition S_1 but not yet committed on partition S_2 because of network latency or thread scheduling on S_2. Since T_1 is committed

on S_1, T_2 returns directly on S_1 without being blocked. However, T_2 will be blocked in S_2 until T_1 is committed because of the incompatibility of write locks and read locks. Therefore, the data read by T_2 on S_1 and S_2 have both been modified by T_1. That is to say, the data read by T_2 are consistent. On the other hand, when T_2 arrives ahead of T_1 on partition S_1 and S_2, neither of the data read by T_2 have been modified by T_1, which is also consistent. Transaction T_1 may be blocked in this case.

Now we consider another condition. If the partitions use MVCC as the concurrency control, a read operation sees a snapshot containing the committed data of each transaction. Figure 2(b) shows how transaction T_2 is executed on the partitions with MVCC. When T_2 arrives at partition S_1 after T_1 is committed, T_2 reads the latest version of data which have been modified by T_1. At the same time, T_2 arrives at S_2. Since T_1 is not yet committed on S_2, T_2 reads an old version of data b. In short, T_2 obtains the data on S_1 which have been modified by T_1 and the data on S_2 which have not been modified by T_1. This is when inconsistency occurs.

Based on the two situations discussed above, we can conclude that: 1) on partitions with 2PL, conflicts increase in the system due to the incompatibility of read locks and write locks, which impacts the performance; 2) on partitions with MVCC, the data read by clients may not be consistent. In this paper we propose DMVCC, a distributed protocol that avoids distributed read inconsistency. Using this protocol, we can free read locks for read operations and obtain a consistent snapshot version in the distributed database with evident performance improvement.

B. Distributed Multi-Version Concurrency Control

This section describes how DMVCC works to guarantee the properties around concurrency control, and how those properties are used to implement features such as transaction consistency and lock-free reads.

A read-only transaction shares the benefits of snapshot reads in performance[4]. And a snapshot read is a read operation that reads the historical data items without locking. In our design, a client does not need to specify a timestamp or a version of data items for a snapshot read. He only needs to determine whether the read operation is a snapshot read or not. If the read operation is a snapshot read, the system will assign a global consistent snapshot version to the operation. If not, the client should execute the operation with *SELECT ... FOR UPDATE*.

To understand the DMVCC, two key points need further elaboration: when to generate a snapshot version on each partition and when to use the version in the system.

- **Generating a snapshot:** Read and write operations in transactions use two-phase locking. As a result, the systems can generate a snapshot anytime after all locks are acquired and before any lock is released. When the partition generates a snapshot before a subtransaction is committed, the subtransaction can not see the modifications by itself. So for a given transaction, the partition generates a snapshot only when the system requires the partitions to commit the subtransaction. At the same time, a global snapshot version is generated.
- **Executing Reads with a version:** When a transaction arrives, the system assigns to it a global consistent snapshot version which involves all the partitions. To read data on a partition, a snapshot read in this transaction needs to refer to the version number related to this partition.

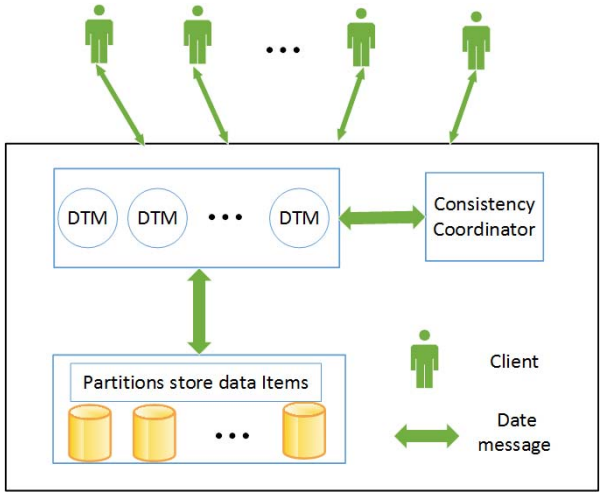


Figure 3: System architecture

Furthermore, a system using DMVCC not only reads a global consistent snapshot, but also reduces read-write conflicts and the chance of global deadlocks[6].

III. DESIGN

To realize DMVCC protocol, we design a system which can read global consistent snapshots. Figure 3 depicts the proposed architecture of our design. The distributed database system consists of three main components: the partitions, the DTMs and the consistency coordinator. The partitions are a number of local databases which store a portion of data items. They execute subtransactions from the DTMs and generate snapshots. Clients access the system with the DTMs. The DTMs break down the transactions into subtransactions and assign snapshot versions to them accordingly. The consistency coordinator is the center node that calculates the global consistent snapshot versions. Next we will describe the design of each component.

A. Partitions

The whole database is partitioned, with items stored across multiple servers and each partition storing only a portion of items. The partitions are independent from each

other, that is, a distributed transaction only contacts partitions which store the items it needs. So when a partition fails, it does not affect the data stored on others.

1) Snapshot: As is analyzed in Section II-B, the partition should generate a snapshot when the subtransaction is committed. The snapshot should have the following features:

- Transactions not yet committed cannot be seen from this snapshot. Since our partition is a multi-thread database, there might be many transactions being executed at the same time. A partition scans a transaction list to see whether a transaction is committed. If the transaction is not committed, its modifications on the data items should not be seen by other transactions.
- When a transaction starts after the snapshot is created, other transactions cannot see the modifications by that transaction from this snapshot.
- Conversely, when a transaction is committed before the snapshot is created, the snapshot can display the modifications by that transaction.
- A snapshot can display the modifications by the transaction that creates it.

Read-My-Write: Furthermore, when a snapshot read in a subtransaction requires to read a previous snapshot, it should not only read the version of data stored in the snapshot, but also the modifications made by the ongoing subtransaction that contains the snapshot read. To achieve that, the partition combines the data stored in the snapshot and the modifications made by the ongoing subtransaction.

2) Local Transaction IDentifier: Since the partitions have created a number of snapshots, the client should be able to determine which snapshot to read. One solution is to transmit the complete snapshot between the DTM and the partition[7]. But it is very hard to realize and the transmission may increase the network load. So the partition needs to generate a number for each snapshot. In this paper, we name this number LTID (Local Transaction IDentifier), which is also called snapshot version number. The generation of LTIDs is a serializable process and the numbers of LTIDs are sequential. When a subtransaction acquires a lock as it is to be committed, the partition generates a snapshot and the corresponding LTID during the lock time. Then the partition links the snapshot with the LTID by a hashtable. When the subtransaction ends, the LTID value is returned to the DTM.

3) Snapshot Reads With LTIDs: When a transaction arrives, the DTM gets a set of LTIDs which form a consistent snapshot (we will describe that process in the following subsection). If a read operation is a snapshot read, the DTM will specify LTIDs for it and sends them to the partitions. When a partition gets a read operation with an LTID, the partition looks up the hashtable using this LTID to get the corresponding snapshot. In this way, clients can read a distributed consistent snapshot version.

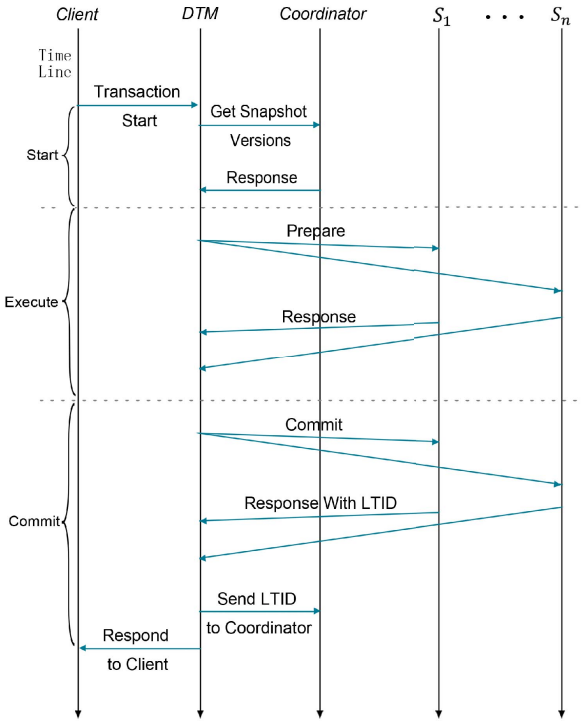


Figure 4: The process of the DTM executing a transaction

B. Distributed Transaction Manager

Clients connect the DMVCC based distributed database system with DTMs. DTMs split the transactions of clients into subtransactions and send them to the corresponding partitions.

DTMs execute distributed transactions with the two-phase commit (2PC) protocol[6], [19]. 2PC is one of the most common distributed control protocols regarding atomic communication in distributed systems. That is because it is quite simple and straightforward. To support DMVCC, we add more features to the 2PC protocol which are shown as follows. Figure 4 demonstrates the process of a transaction being executed.

- When a transaction arrives at a DTM, the DTM connects to the consistency coordinator to get the global consistent snapshot version. The consistent version is a set of LTIDs collected from the partitions which present the snapshots on the partitions.
- When a transaction is committed, the partition sends the LTID to the DTM. After the DTM collects all the LTIDs from the partitions, it sends the LTIDs to the consistency coordinator. The consistency coordinator then uses the LTIDs to calculate a global consistent snapshot version.

Our partitions have a mechanism to detect and recover from local dead locks[1], [7]. However, in a distributed

Table I: LTIDs of transactions involving all partitions

LTIDs / Partitions / Transactions	S_1	S_2	S_3
Initial	0	0	0
T_1	1	2	3
T_2	2	3	1
T_3	3	1	2

Table II: LTIDs of transactions involving portion partitions

LTIDs / Partitions / Transactions	S_1	S_2	S_3
Initial	0	0	0
T_1	1	2	-
T_2	-	1	2
T_3	2	-	1

database system, there might be a global dead lock which cannot be detected actively by the DTMs[6], [7]. The time-out mechanism is adopted in our system to ensure that other partitions do not need to wait too long when a global dead lock occurs. In future researches, active detection mechanism might be used, with which the system aborts the transactions actively when a dead lock is detected.

C. Consistency Coordinator

Being the core part of the system, the consistency coordinator enables the system to get a global consistent snapshot. Since we already have a good design in the partitions, the algorithm for the calculation of a consistent version is very simple and lightweight. It takes two steps:

I. Every time the coordinator receives the LTID value pairs (S_1:$LTID_1$, S_2:$LTID_2$, ..., S_n:$LTID_n$) sent from each DTM, it sorts the LTIDs by partition immediately, including the ones that have been stored on the coordinator. For example, when an LTID value of partition S_1 is sent to the coordinator, it compares and sorts the received LTID with others that are previously stored on it.

II. A global consistent state is achieved if the LTIDs generated by each partition are consecutive and holes-free. Therefore, snapshots related to the maximum LTID values of each partition form a global consistent state (S_1:$LTID_{1max}$, S_2:$LTID_{2max}$, ..., S_n:$LTID_{nmax}$). The correctness of this algorithm is proved in Section IV.

After the coordinator figures out the consistent snapshot version, another set of LTID values arrives. The coordinator then starts a new round of calculation. To alleviate the burden of calculation, the coordinator only needs to sort with the maximum LTID values of the previous round as the initial values in the new round. In this way the system gets a number of consistent snapshots. When a new transaction starts, the latest consistent snapshot version is sent.

Next we present two examples to explain how the algorithm functions in transactions involving all the partitions and in ones involving only a few.

The first example contains transaction T_1, T_2 and T_3 and partition S_1, S_2 and S_3. Each transaction involves all three partitions.

Since all partitions are multi-thread databases, a set of transactions may be committed in different orders on each partition, that is, they may generate LTID values in different orders. Now suppose the initial LTID values on each partition are 0, which means they are in a global consistent state in the beginning.

When the transactions are committed, the LTID values are generated on each partition. One of the possibilities is shown in Table I. The consistency coordinator receives the LTID values in the order of T_1, T_2 and T_3. The numbers next to them are the LTID values generated on each partition when transaction T_1, T_2, T_3 are committed.

- **Commission of transaction T_1:** The LTID values of T_1 are sent to the consistency coordinator. The consistency coordinator determines, with the LTID values of T_1, whether a consistent version can be achieved. As is shown in Table I, the sequence of LTIDs on S_1 is (0,1), which is consecutive. But the sequence of LTIDS on S_2 is (0,2), which is inconsecutive. This means a consistent version is not accessible on S_2 now and the next transaction is expected.
- **Commission of transaction T_2:** In this phase, the sequence of LTIDs on S_1 is (0,1,2), which is consecutive. But that of LTIDs on S_2 is (0,2,3), which is still inconsecutive and in need of the next transaction.
- **Commission of transaction T_3:** The sequences of LTIDs on S_1, S_2 and S_3 all become (0,1,2,3) when the LTID values of T_3 arrive. A global consistent version is now accessible.

With the maximum LTID values, a global consistent version (S_1:3,S_2:3,S_3:3) comes into being. Now the coordinator is ready to send the latest consistent snapshot versions to another transaction if there is a request. When the next set of LTID values arrives, the coordinator uses (S_1:3,S_2:3,S_3:3) as the initial LTID values to judge consecutiveness.

In the second example, all three partitions are not involved in each transaction. Suppose that T_1 only involves S_1 and S_2, that T_2 only involves S_2 and S_3, and that T_3 only involves S_1 and S_2, as is shown in Table II.

The coordinator still receives the LTID values in the order of T_1, T_2 and T_3 while the way it judges consecutiveness remains the same. Again, only when T_3 arrives do the LTID values on S_1, S_2 and S_3 become consecutive and holes-free. Now the coordinator is able to find the consistent version of (S_1:2, S_2:2, S_3:2).

The consistency coordinator will not be the bottleneck since it only interacts with the DTMs at transaction start

and commit time and conducts simple calculations which operate fast.

D. Garbage Collection

In our design, there are many historical versions of snapshots. So clients can read any consistent snapshot with the suitable version. But without an effective garbage collection mechanism, the storage of these snapshots will become a burden to the system and impair its performance. While in our design, when the system gets a new consistent snapshot, the consistency coordinator sends its version number to the DTM, which then sends it to the corresponding partition. Since the partition may have a long-lived transaction which requires older snapshot versions, the partition compares the received LTID with the each LTID in the long-lived transactions to determine a deletable LTID value (which is the smaller one of the two). The partition then deletes the snapshots whose numbers are smaller than that deletable value. After that, the partition refers to the snapshot to delete the historical data items. Eventually, there are only a few snapshots on each partition.

E. Fault Tolerance

To tolerate failure, each partition, DTM and the coordinator needs to persist its transaction log to disks. Each partition and DTM log the transaction statements and the decision of each phase in case it fails during execution. The consistency coordinator also logs the LTID values of each transaction. There are also two standby coordinators in case the leading one fails. All of coordinators replicate logs using Paxos algorithm[13].

If DTM fails, it will judge whether the transactions are in prepare state or committed state after it recovers. If the transaction has been prepared, it will request partitions to commit. If it has not been prepared or committed, the transaction will be executed from the beginning. If the partition fails, it needs to recover all the prepared transactions after it recovers and then responds to other requests. If a transaction has been decided to be committed, it should be committed after the partition recovers. If the coordinator fails, the system will switch to a standby coordinator which has the same records.

There are other failures, such as network partition and abnormal connection closure. We have methods to deal with all of them. But we cannot discuss them in detail because of the page limit.

IV. Correctness

Like transactions in a single database, distributed transactions also need to satisfy the consistency requirements, including write consistency and read consistency[4], [7], [8]. In this section, we will prove the effectiveness of our algorithm in ensuring read consistency and explain the transaction isolation level the system can achieve.

A. Distributed Read Consistency

The purpose of this paper is to design a system which supports a distributed read consistency protocol on partitions using MVCC as the concurrency control. We prove why consistency can be achieved when the LTIDs generated by each partition are consecutive and holes-free. But to do that, we have to prove a proposition first.

The proposition:

There are two subtransactions T_1 and T_2 on one partition. When they are committed, they get two LTID values $LTID_1$ and $LTID_2$, which correspond to two snapshots $snapshot_1$ and $snapshot_2$. If $LTID_2 >= LTID_1$, $snapshot_2$ is able to capture the modifications which have been made by transaction T_1.

Proof:

If $LTID_2 = LTID_1$, then T_1 and T_2 are the same transaction and the snapshot it creates is able to capture the modifications made by the transaction itself. Therefore, we mainly focus on $LTID_2 > LTID_1$.

Suppose that $snapshot_2$ cannot capture the modifications made by T_1. This might be caused by two conditions according to the definition of snapshot.

1. T_1 is an active transaction;
2. The start time of T_1 is later than the time when $snapshot_2$ is created.

In condition one, when T_2 is committed, T_1 has not been committed yet. So when T_1 is committed, the LTID value of T_1 is larger than that of T_2 because the LTID values are generated in a serializable way. But this contradicts with $LTID_2 > LTID_1$. In condition two, transaction T_1 has not started yet when T_2 is committed. So the LTID value of T_1 is larger than that of T_2, which leads to contradiction again.

Therefore, we can conclude that for any transaction, the snapshot with a larger LTID can capture the modifications made by the transaction with a smaller LTID.

On the other hand, it is also self-evident that if $snapshot_2$ can capture the modifications made by transaction T_1, $LTID_2$ is larger than $LTID_1$.

With the proposition proved, we will now move on to prove that a global consistent state can be formed when all the LTIDs generated by the partitions are consecutive.

Proof:

Suppose the amount of the distributed transactions is m (T_1,T_2,...,T_m) and the amount of partitions is n (S_1,S_2,...,S_n). $LTID_{i,j}$ represents the snapshot number of transaction T_i generated on partition S_j.

Without loss of generality, the committed order received by the consistency coordinator is T_1,T_2,...,T_m, with the initial snapshot version number as 0. Because the partition is a multi-thread database, the committed order on each partition may not be the same. When the consistency coordinator receives transaction T_i, the coordinator sorts the

LTID values. One situation is that the LTID values are consecutive on partition S_j but not consecutive on other partitions, such as S_k. This means transaction T_1,T_2,...,T_i are all committed on S_j, but at least the LTID value of one transaction $T_a(a \leq i)$ committed on partition S_k is larger than i. If now the consistency coordinator considers the system to be in a consistent state, the system uses the snapshots corresponding to the maximum LTID values on each partition (according to the above proposition, a transaction can read the modifications made by another transaction with a smaller LTID) to read data items when the next transactions arrive. On partition S_j, the new transaction can read the modifications by T_1,T_2,...,T_i. But on partition S_k, the new transaction can read the modifications by other transactions which might include but are not limited to T_1,T_2,...,T_i. That leads to inconsistency. Another situation is that the LTID values on every partition are consecutive. When the system uses the snapshots corresponding to the maximum LTID values on each partition, the new transaction can read the modifications by all transactions T_1,T_2,...,T_i on all the partitions, which is a consistent state.

Table I shows an example. The LTID value on S_1 is 1 after the results of T_1 are sent to the consistency coordinator, which means only T_1 is executed on S_1. But the LTID value on S_2 is 2, which means another subtransaction (T_3) has been executed on S_2 before T_1. Similarly, the LTID value on S_3 is 3, meaning another two subtransactions (T_2,T_3) have been executed on S_3 before T_1. Now suppose the consistency coordinator determines (S_1:1,S_2:2,S_3:3) as a consistent snapshot version. A new transaction T_4 involving S_1, S_2 and S_3 will use the above-mentioned LTIDs as version numbers to read data items on the partitions. With the proposition we have proved before, we can infer that the results T_4 reads on S_1 only involve data modified by T_1, while those on S_2 involve data modified by both T_1 and T_3 and those on S_3 involve data modified by T_1, T_2 and T_3. This leads to inconsistency presented in Section II-A.

Conversely, the LTIDs on each partition are consecutive after the consistency coordinator receives T_3, as is shown in Table I. Using (S_1:3,S_2:3,S_3:3), a new transaction T_4 is able to read all the modifications made by T_1, T_2 and T_3, which is similar to the proposition we have proved. So when the LTIDs on each partition are consecutive, a new transaction is able to read global consistent snapshots corresponding to the maximum LTIDs on each partition and the read statement using the consistent version is a distributed consistent read.

B. Snapshot Isolation Level

All snapshot read operations in a transaction can see a global consistent snapshot of the database. Read operations in one transaction are personal snapshots of the database. For write operations, they acquire write locks that are incompatible with other transactions which update the same data item. Instead of simply being aborted, the transactions wait until they acquire the locks and modify the data. This means the system has achieved the snapshot isolation level according to [4], [8].

V. EVALUATION

To gain a quantitative understanding of the benefits of using DMVCC in a distributed database system, we set up a series of experiments. The partition is realized by modifying MySQL[1], a popular open-source relation database management system. Many systems like Google's F1[18] and Salt[23] use MySQL as the backend database.

Our experiments explore three key questions:

1. How is the throughput of the distributed system using DMVCC comparing to that of the system using standard 2PL under varying levels of contention?
2. How does the performance improve as the proportion of read transactions increases?
3. Does the consistency coordinator become a bottleneck when the system scales out?

The results show that 1) the system with DMVCC has higher throughput than one with 2PL, especially when the level of contention is high; 2) as the read transactions increase, the performance of the system with 2PL drops sharply while the DMVCC based system only shows minor decrease in its performance; 3) the consistency coordinator does not become the bottleneck as the system scales out.

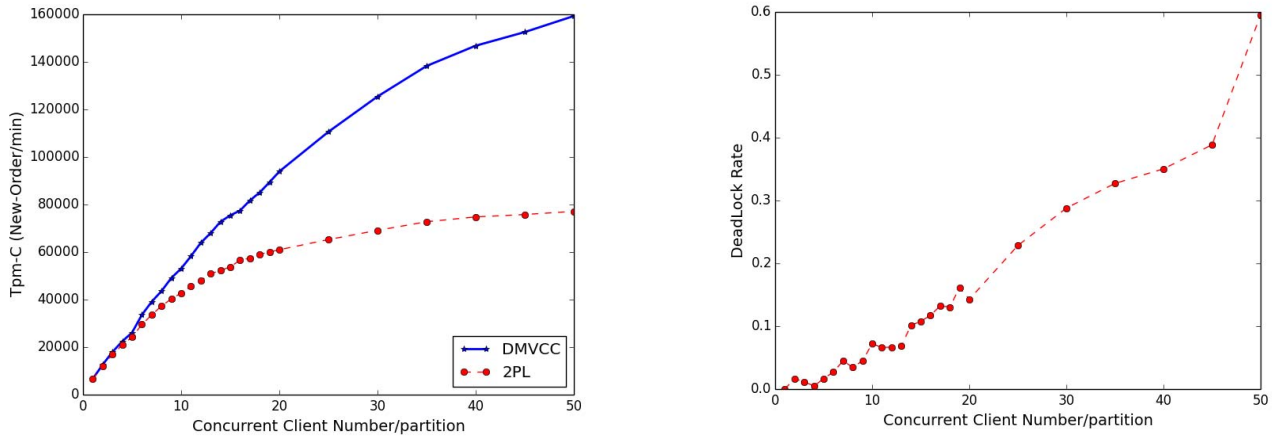
A. Experimental Setup

In our experiments, each machine is equipped with a 12-core Intel Xeon CPU E5-2620 running at 2.4GHz with 32G RAM and Gigabit Ethernet. We use TPC-C[2], a popular database benchmark that models on-line transaction processing, as our benchmark. The performance metric reported by TPC-C measures the number of new orders that can be fully processed per minute and is expressed in tpm-C.

In this experiment, clients and servers run on different machines. Each client machine runs 1 to 20 single-threaded client processes while each server machine runs a single server process. Like[20], [21], our experiments divide the TPC-C database by warehouse. To test the performance in all experiments, we assign five warehouses to each partition. Each data point in the figure represents the median of at least five trials. Each trial is run for over 120s with the first and last quarter of each trial elided to avoid start up and cool down artifacts.

B. Contention

In the contention experiment, we use 4 partitions, with each partition serving five warehouses. Then we vary the clients per partition from 1 to 50. As the number of clients increases, there are more requests per partition and thus higher contention. The results are shown in Figure 5. Figure 5(a) shows the throughput of new-order transactions



(a) Throughput

(b) Deadlock rate using 2PL

Figure 5: The characteristics of the system with 4 partitions as the client number increases

while Figure 5(b) shows the rate of deadlocks on the partitions using 2PL.

Low Contention. When the number of concurrent requests per partition is smaller than 10, the throughput of the system shows no difference when using and not using DMVCC. When the client number per partition is 10, the throughput of the system is 53056 new-orders/min if it uses DMVCC, while it is 42780 new-orders/min if it does not use DMVCC. The system using DMVCC does not have deadlocks and the CPU is never saturated, so the throughput of the new-order transactions can increase linearly as the client number goes up. Since a system using 2PL has almost no contention or deadlocks (less than 7%), the throughput of the system displays a similar growth pattern to one using DMVCC.

Medium Contention. When the number of concurrent requests per partition increases from 10 to 20, the contention level increases from low to medium. The system using DMVCC is less sensitive to the increase of client number, so the throughput keeps increasing in a linear fashion as in the previous case, which is faster than the system using 2PL. This is because the system using DMVCC only has write-write conflicts while the system using 2PL has both read-write and write-write conflicts, where read operations will block write operations and generate deadlocks easily. When the client number increases, the chance of deadlocks increases as well. As Figure 5(b) shows, the rate of deadlocks increases to 17%. At the same time, when the request number is 20, the throughput of the system using DMVCC is 93898, 1.53x larger than that of the system using 2PL, which is 60975.

High Contention. When the number of concurrent requests per partition is over 20, the benchmark reflects a high level of contention. The throughput displays almost no increase in the system using 2PL. At the same time, the rate of deadlocks increases sharply to about 60%. When the system has plenty of deadlocks, it needs to spend a lot of time dealing with it and retry multiple times. As a result, the increase of the throughput is very slow. On the contrary, the system using DMVCC is less sensitive to this increase. But the throughput does not increase as fast as before because of a larger amount of write-write conflicts and resource limitation. Now the throughput of the system using DMVCC is 152538 when there are 40 parallel clients per partition. This is 2.0x larger than that of the system using 2PL, which is 75780.

C. *Effects of Varying Read/Write Ratio*

To obtain more read operations, we vary the ratio of read-only transactions to read-write transactions. In the regular experiments, the ratio of read-only transactions to read-write transactions is about 1:10 [2]. So we increase the proportion of read-only transactions in this experiment.

In the previous experiment, we have learned that when the client number is small, the throughput of the system almost remains the same no matter whether it uses DMVCC or not. In this experiment, we have 10 clients per partition who only run read-write transactions. Then we have other clients running read-only transactions. The client number per partition increases from 0 to 10, so the ratio of read-only transactions to read-write transactions is 0 to 1.

To measure the performance in this experiment, we not only use tpm-C but also the throughput ratio in response to the ratio of read-only transactions to read-write transactions. The results are shown in Figure 6.

When there are few read-only transactions, the gap be-

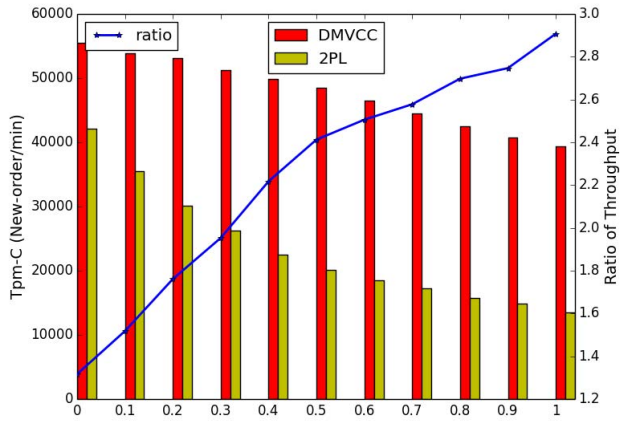


Figure 6: The throughput and the throughput ratio as the read/write varies

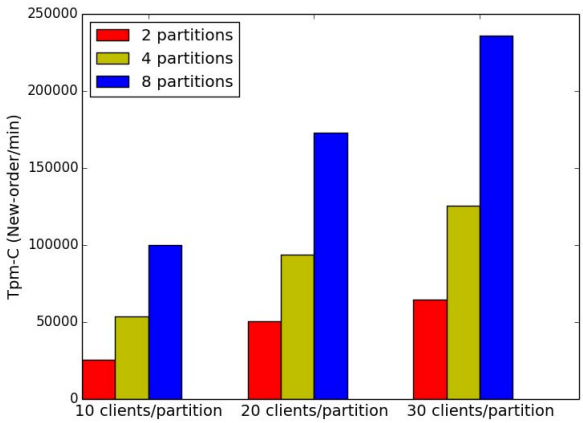


Figure 7: Throughput at different contention levels when partitions scale out

tween the throughput of the system using DMVCC and that of the system using 2PL is small. With the proportion of the read-only transactions increases, the gap becomes larger. We can see that the throughput of the system using 2PL decreases quickly, because as the read transactions increase, the chance of read operations blocking write operations increases and deadlocks occur. The throughput of the system using DMVCC decreases as well, because when the clients running read-only transactions increase, the system needs more resources to deal with them, which decreases the throughput of new-order transactions. However, the decrease is slower than that of the system using 2PL. When the ratio of read-only transactions to read-write transactions is 1:1, the throughput of the system using DMVCC is 2.9x larger than that of the system using 2PL.

D. Scalability

We evaluate the scalability by scaling out the number of partitions, each of which has five warehouses. Due to the limit of resources, we increase the number of machines from 2 to 8. We test the scalability of the system at low, medium and high contention levels, or, when each partition has 10, 20 and 30 clients. The result is shown in Figure 7.

We find that as the partitions scale out, the throughput of new-order transactions increases linearly at all contention levels. The existence of the consistency coordinator is the only possible cause of the system having a bottleneck. But since it only has simple interactions with the DTMs and conducts simple calculations, it is not likely to be a bottleneck in our system.

VI. RELATED WORK

The key contribution of this design using DMVCC is that an application is able to access a consistent snapshot version without read locks in a distributed database system. In distributed database systems, it is very hard to get a global consistent version if each partition is based on MVCC. Most distributed systems like Lynx[24], Calvin[20] do not support snapshot reads while systems like R*[15] and Gamma[10] use 2PL.

There are some attempts of supporting read consistency. First, timestamps are adopted in distributed database systems as a type of ordering concurrency control known as conservative T/O[5], [7], which allows transaction abort and reorder. But unlike our design, they do not support snapshot read. Spanner[9] designed by Google supports distributed consistent reads. To make the global time synchronous, the system uses a GPS and an atomic clock as clock references. When using an atomic clock, there is little mistiming on each machine. But there is also a problem: not all systems have atomic clocks because of their high cost. Another solution is presented in[17], which designs an additional protocol for read-only transactions. OCC[12], a lock-free concurrency control protocol, is used specially for read-only transactions in that system.

There are also serveral algorithms that allow the system using MVCC to have a distributed consistent read. Distributed commit list is proposed in [7]. Clients connect to each partition to get the commit list at the start of a transaction. As the transaction is committed, the partition generates a temporary commit list, in which case the next query may see an inconsistent view. Bailis introduces Read Atomic Multi-Partition transactions[3] which ensure that none or all of the transaction updates are visible to other transactions. But inconsistency may occur when a new transaction starts before the prepare instruction of a previous transaction arrives at all partitions. The last one which uses global snapshot is similar to our design. In ecStore[22],

the snapshots are detained by the center node. When a transaction is committed on one partition, the center node will add a number to it and send the number to all the other partitions. In our design, the version number is generated on the partitions while the consistency coordinator is a lightweight server which only conducts simple computation.

VII. CONCLUSION

This paper has designed the Distributed Multi-Version Concurrency Control protocol, which supports consistency and lock-free snapshot read operations. Not only in read-only transactions but also in read-write transactions, clients can read consistent snapshot without read-write conflicts. When using TPC-C benchmark, a DMVCC based system outperforms a conventional system at different contention levels, especially when read/write ratio is high. Finally, we design a system which adopts this protocol. The system has been used in Baidu's services such as Baidu Wallet, where both performance and scale-out ability are ensured..

REFERENCES

[1] Mysql. http://dev.mysql.com/doc/.

[2] Transaction processing performance council. http://www.tpc.org/tpcc/.

[3] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with ramp transactions. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 27–38. ACM, 2014.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[5] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the sixth international conference on Very Large Data Bases-Volume 6*, pages 285–300. VLDB Endowment, 1980.

[6] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[8] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):20, 2009.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[10] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):44–62, 1990.

[11] F. D. Hinshaw, C. S. Harris, and S. K. Sarin. Controlling visibility in multi-version database systems, Dec. 4 2007. US Patent 7,305,386.

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

[13] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[14] P.-Å. 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*, 5(4):298–309, 2011.

[15] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[16] C. Mohan, H. Pirahesh, and R. Lorie. *Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions*, volume 21. ACM, 1992.

[17] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, 2014.

[18] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, et al. F1: the fault-tolerant distributed rdbms supporting google's ad business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778. ACM, 2012.

[19] D. Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142. ACM, 1981.

[20] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[21] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[22] H. T. Vo, C. Chen, and B. C. Ooi. Towards elastic transactional cloud storage with range query support. *Proceedings of the VLDB Endowment*, 3(1-2):506–514, 2010.

[23] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, 2014.

[24] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.