2018 IEEE 34th International Conference on Data Engineering

Range Optimistic Concurrency Control for a Composite OLTP and Bulk Processing Workload

Jiahao Wang [1], Peng Cai [*1,2], Jinwei Guo [1], Weining Qian [1], Aoying Zhou [1]

[1] School of Data Science and Engineering, East China Normal University
{jiahaowang,guojinwei}@stu.ecnu.edu.cn, {pcai,wnqian,ayzhou}@dase.ecnu.edu.cn
[2] Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology

***Abstract*—This work addresses the need for efficient key-range validation for a composite OLTP and bulk processing workload characterized by modern enterprise applications. In-memory database system (IMDB), mostly adopting the optimistic concurrency control (OCC) mechanism, performs well if the contention of conventional OLTP workloads is low and each transaction only contains point read/write query with primary key. In this work we present the performance problem of IMDBs under mixed OLTP and bulk processing workloads. The reason is that existing OCC protocols take expensive cost to generate a serializable schedule if the OLTP workload contains bulk processing operations with key-range scan. To this end, we develop an efficient and scalable range optimistic concurrency control (ROCC) which uses logical ranges to track the potential conflicting transactions and to reduce the number of transactions to be validated. At the read phase, a transaction keeps a set of predicates to remember the version and precise scope of scanned ranges, which eliminates the cost of maintaining scanned records. Before entering the validation phase, if the transaction intends to update records in the logical range, it needs to register to the corresponding lock-free list implemented by a circular array. Finally, ROCC filters out unrelated transactions and validates the bulk operation at range level. Experimental results show that ROCC has good performance and scalability under heterogeneous workloads mixed with point access and bulk processing.**

***Index Terms*—Range concurrency control, OCC, Serializability**

I. Introduction

Online Transaction Processing (OLTP) systems have supported the business world over the past several decades. To deal with complicated business logic, the traditional database system, has gradually developed into a comprehensive enterprise-class data management software. However, nowadays the back-end database system is required to execute highly concurrent, short transactions and simultaneously to provide efficient bulk processing for complex business logic. For instance, the sales promotion of e-shopping malls needs the payment system to confront with massive payment requests in a short time. On the other hand, real-time fraud detection, end-of-day settlement, and other complex analyses require the payment system to efficiently process bulk data. The workload characterized by these kinds of applications is a composite OLTP and bulk processing transactions.

Modern database systems, such as Hekaton [1], HANA [2] and HyPer [3], have been optimized for multi-socket, multi-core servers with large memory. As the overheads of traditional disk I/O and buffer management have disappeared in in-memory database system (IMDB), the concurrency control scheme itself has become the main bottleneck of high-performance transaction processing [4], [5]. Recent researches focused on scaling the concurrency control scheme up to the increasingly more CPU cores. Compared with the well-known two-phase locking (2PL) protocol, the optimistic concurrency control (OCC) scheme allows transactions to execute concurrently in a lock-free manner, and takes much shorter time to detect transaction conflicts during the validation phase. Thus it has been revisited and adapted for in-memory database systems under multi-core hardware environment [1], [3], [2].

The serializability is the major criterion for correctness when transactions are executed concurrently. The concurrency control schemes of existing in-memory database systems are designed to effectively detect transaction conflicts at the record level. These schemes are inappropriate to the transaction workload containing bulk processing with key-range scan or updates. To guarantee a serial transaction schedule with scan queries, the concurrency control protocol must guarantee: (1) the scanned records are not changed, and (2) there are no inserted or deleted records in the scanned range. Classic locking based protocols such as key-range locking [6] and next-key locking [7] have been implemented in traditional DBMS products to provide a serializable schedule of the transaction with scan operations. In the setting of unbundling transaction processing and storage component, Levandoski [8] introduced a pessimistic concurrency control method for multi-version logical ranges to ensure that the key-range is unchanged. However, it's not precise for the locking on the physical or logical ranges to detect conflicts, and thus has the risk of detecting false conflicts [9]. Furthermore, frequently requesting and releasing locks consume much memory bandwidth, and introduce significant locking overhead in IMDBs [10].

A. Problem Analysis and Demonstration

Optimistic concurrency control detects transaction conflicts in the validation phase, but it's still inefficient under a mixed OLTP and bulk processing workload containing key-range scan. Recently proposed OCC variants for IMDBs adopted two categories of methods to guarantee the correctness of range scan. The first kind of key-range validation scheme

*Corresponding author

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00061

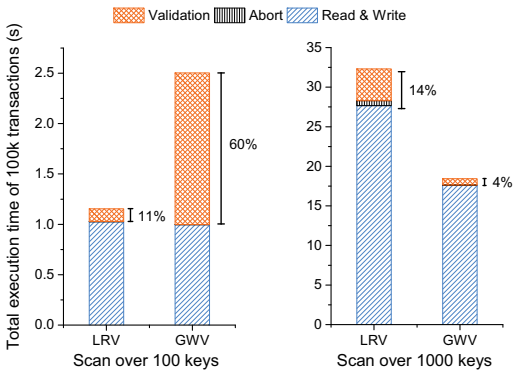


Fig. 1: Performance profile for Local Readset Validation and Global Writeset Validation under a hybrid YCSB workload.

requires the validating transaction to re-read records from the scanned key-range [1], [11], [12], [13], referred to as Local Readset Validation (LRV). Even in the setting of IMDBs, it has obviously negative impact on performance because of intensive memory accesses for a large amount of re-scanned records. Silo adopted LRV and optimized it for phantom detection by checking the version change of B-Tree nodes which cover the scanned key-range [12]. However, it still needs to maintain all scanned records and re-read them for validation. The other approach, instead of re-scanning the key-range, validates whether the writesets of all other concurrent transactions violate the predicates of the scanned key-range [14], [15], [16], referred to as Global Writeset Validation (GWV). The GWV approach is adopted by HyPer [3], [15]. Compared with LRV, GWV only needs to access the relatively small amount of written records under read-intensive workloads. However, in the case of multi-core setting and under write-intensive workloads, the performance of GWV would decline because it needs to validate a large set of writes generated by overlapping transactions.

In Fig. 1, we demonstrate performance profile for the validation schemes of LRV and GWV under a hybrid YCSB workload [17]. The YCSB table is initialized with 10 million records. The workload consists of 90% update transactions and 10% scan transactions. The update transaction contains five update operations, and the scan transaction contains four update operations and an additional key-range scan query. Both the update keys and the start keys of range scan queries are selected according to the same Zipfian distribution. The access pattern of the workload is low-skewed. The scan range is over 100 keys for the workload on the left side of Fig. 1, and it is over 1000 keys on the right side. The total execution time of each experiment is divided into three parts: the read and write, the validation and the abort. The part of read and write is the sum of CPU time used by all committed transactions for the read and write phases. The main cost of this part is to access the data from memory and to maintain the readset/writeset. The validation part is the sum of CPU time used by all committed transactions for the validation phase. The abort part sums up the execution time of all aborted transactions. For each experiment, we run 100k transactions and measure the execution times on each part.

On the left side of Fig. 1, GWV takes 60% of the total execution time in the validation phase. It is costly for GWV to validate all writes of the recently committed transactions, and GWV would suffer from wide concurrent updates. On the right side of Fig. 1, since the scan range under the workload is over 1000 keys, it is also costly for LRV to maintain a lot of scanned records in the readset and re-read them for validation. As a result, LRV takes more time than GWV not only on the part of read and write but also on the validation part. This experiment indicates that both LRV and GWV have the performance problem under a hybrid workload.

In this paper we introduce the design and implementation of range-based optimistic concurrency control to optimize the validation phase of transactions with key-range scan operations. We assume that all retrievals/updates are via index key in the hybrid workload. The basic idea of Range-based Validation (RV) is to partition the global key space into a number of logical ranges. And we use a lock-free list implemented by a circular array to track the transactions which have modified records in each logical range. During the validation phase, a transaction only needs to check the lists of its scanned logical ranges to detect the potential conflicts with other overlapping transactions.

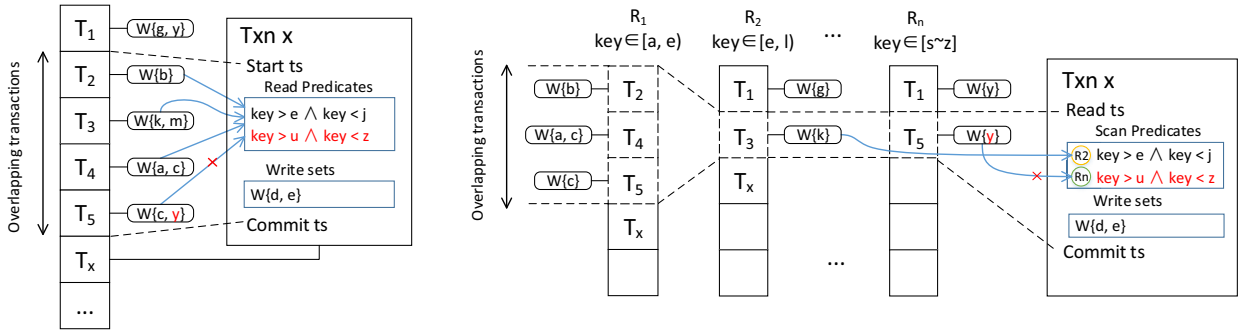
In this work, we make the following contributions :

- **Validation at Range Level**: This work introduces the Range-based OCC commit protocol to effectively support a composite OLTP and bulk processing workload. ROCC validates the scanned key-range at the logical key-range level instead of validating the record in the readset/writeset one by one.
- **Efficient Write Management**: We present an efficient implementation of transaction management via logical range.
- **Evaluation**: Experimental results show that this approach reduces the count of records to be validated and achieves good performance in the OLTP and bulk processing with key-range scan workload.

The paper is organized as follows: Section II introduces the background and the basic idea. We describe the implementation details in Section III and make a cost analysis on range validation in Section IV. We report extensive experiment results in Section V and make a discussion in Section VI. Section VII reviews related works and Section VIII concludes the paper.

II. Range Concurrency Control

Compared with generating a serializable schedule for transactions with operations only at the record level, it's harder to provide the serialization isolation for the transactions with range scan queries. The lock-based pessimistic concurrency control resolves this problem by using multiple granularity locking protocols to prevent the conflict writes or phantom records occurring at the specified range [18]. Although OCC has been regarded as a better choice to meet the requirements



(a) Validation via a global txn list.

(b) Validation via txn lists in logical ranges.

Fig. 2: Illustrating examples for comparing GWV (a) to RV (b). To validate no write violates scan queries, GWV maintains recent write transactions in a global list and RV maintains those in logical ranges.

of high-performance transaction processing, there is still a scalability problem under a heterogeneous workload mixed with write and range scan queries. In this section, we briefly review the original OCC protocol proposed by Kung and Robinson [14], and give an overview of ROCC model with a motivating example.

A. Optimistic Concurrency Control

OCC divides the transaction execution into three phases. In the read phase, a transaction optimistically executes its read/write operations without any blocking. The readset and writeset of a transaction are used to save its read and written records respectively. In the validation phase, conflict detection is carried out to guarantee the transaction schedule is serializable. In the write phase, if the transaction is successfully validated, its local writeset can be applied and visible to other transactions.

The original OCC assumes that the actual conflicts are rare among concurrently executed transactions and most transactions commit in a short time. However, this assumption is broken in the case of heterogeneous workloads including small updates (for the OLTP-like transaction) and short/long key-range scans (for bulk processing). This kind of mixed workloads often appears in the hybrid OLTP&OLAP database systems [3], [19], [2]. Since OCC needs to validate whether the records in the readset are changed by other overlapping transactions, the validation cost grows along with the size of readset. The time-consuming validation on range scan operation may delay committing the transaction and further increases the probability of violating serializability validation, which leads to higher abort rate.

B. Range based OCC Overview

To clearly demonstrate range based key-range validation, we give a motivating example of global writeset validation and range based validation in Fig. 2. It shows how the OCC protocol based on RV reduces the validation cost.

When using GWV, the transaction first obtains a start timestamp from a global counter before entering the read phase, and it obtains a commit timestamp from the same counter after completing its read phase [15]. Transactions are serialized according to the assigned commit timestamps and pushed into a sequenced *global list*. These two timestamps make it to find the transactions that have been committed during the read phase. As we can see from Fig. 2(a), transactions from T_2 to T_5 are committed during the read phase of T_x. Under the commit protocol of GWV, T_x passes the validation only if no committed write satisfies its read predicate from T_2 to T_5. In this example, we assume that the key space starts from a to z. Since the writeset of T_5 is $W\{c, y\}$ and the record with key y satisfies the predicate of $key > u \wedge key < z$, the transaction T_x should be aborted. According to the GWV-based OCC protocol, T_x needs to validate four overlapping transactions. However, there is only T_5 whose writes intersect the scanned range of T_x.

In order to accelerate validation, the RV-based OCC protocol adopts logical ranges to filter out *unrelated transactions*. First, the key space is partitioned into a set of logical and continuous key-ranges. Fig. 2(b) presents a toy example of partitioning range. Each range keeps the recently committed transactions, containing committed write operations in this range, into a associated list. Second, we transform the scan query of T_x into several predicates, where each predicate fully or partially matches a single logical key-range. In Fig. 2(b), a scan query of T_x is transformed into two predicates, both of which partially scan R_2 and R_n respectively. Thanks to detecting conflicts at range level, RV-based OCC finds write operations only in R_2 or R_n may change T_x's reads.

Instead of validating four transactions by GWV-based OCC, RV-based OCC just needs to validate two transactions for T_x. Furthermore, if the predicate fully covers the logical range R_2 or R_n, it is not necessary for the RV-based OCC to validate the write records from committed transactions one by one. Since any committed write by a committed transaction should conflict with the validating transaction, it only needs to check whether there exists any overlapping committed transaction.

III. ROCC IMPLEMENTATION

A. Logical Ranges Management

The concept of the logical range has been introduced in Deuteronomy, a transactional key-value store, where concurrency control component is clearly decoupled from the data storage and access module [20], [8]. To provide serializability for the key-range scan operation in the transactional component, they designed a pessimistic concurrency control by adding a read lock on the scanned logical range.

Logical ranges are partitioned according to the primary key, and each logical range contains the same number of records. Fig. 3 shows an example where the index keys (from a to z) are partitioned into n logical ranges from R_1 to R_n. Thus, the logical ranges are continuous and disjoint to the neighboring one. Each logical range contains keys in the interval of $[start\_key, end\_key)$.

ROCC creates a lock-free list for each logical range to maintain the recent transactions which have modified records in this range. The lock-free list is efficiently implemented by a circular array which contains the pointers to transactions and is operated by atomic instructions. The memory space of the circular array is preallocated. When registering writes to a logical range, the transaction pushes its pointer to the next available slot of the circular array. Since the array is circular, the content of the stale slots would be overwritten by the running transactions. A registered slot becomes stale when the commit timestamp of the registered transaction is earlier than those of all the other running transactions. Besides, each logical range maintains a counter to denote the current version of the logical range and a transaction registration will increment the counter by one atomically. We analyze and explain how to select the appropriate size of the circular array in Section IV.

B. Predicates for Scan Queries

The key-range query may scan the records across multiple logical ranges. ROCC uses a set of predicates to remember the ID of the scanned logical ranges and the scanned scope as well. For each scanned logical range, ROCC creates one predicate with four fields $\{rangeID$, rd_ts, $start\_key$, end_key, $cover\}$. The $rangeID$ is the index number of the logical range. The rd_ts is the logical start timestamp at the beginning of scan in this range. The precise scope of scanned range starts from $start\_key$ to end_key, where the end_key is not included. When the logical range is fully covered by the predicate, the $start\_key$ and the end_key is set to $\varnothing$. Instead, the $cover$ flag is set to true. Fig. 3 shows examples of the mappings between predicates and logical ranges for three scan queries.

C. Commit Protocol

The commit protocol of ROCC guarantees the serializability of transaction schedule. In order to detect read-write conflicts at range level, on the one hand, ROCC uses the mapped predicates for a scan query to remember the version of logical ranges scanned by this query. On the other hand, if

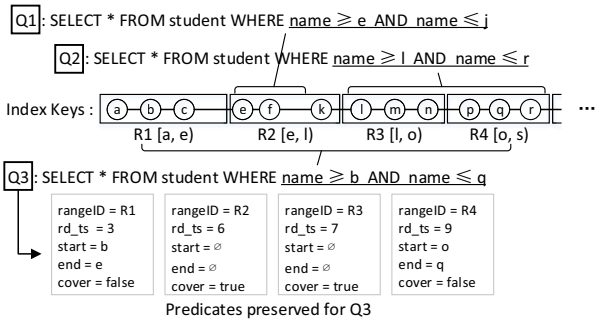


Fig. 3: Examples of the mappings between predicates and logical ranges.

a transaction has a write operation in a logical range, it will register to the range's lock-free transaction list. The ROCC commit protocol is presented in Algorithm 1. We prove the serializability guarantee of this protocol in Section III-D.

1) Timestamp Management: The life cycle of each transaction starts at start timestamp and finishes at commit timestamp. The start timestamp is the time of when the transaction begins. *The commit timestamp determines the order of committing transaction and is assigned before entering into the validation phase*. If the transaction has an assigned commit timestamp, it means this transaction is either committed or just in the validation phase. The commit timestamp is also used to represent the version of the record and the logical range. The version of a record would be updated by the committed transaction. For each logical key-range, ROCC uses a transaction list to track the recently committed or validating transactions which contain write operations in this key-range. The latest version of a logical range is represented by the last commit timestamp of the transaction in the corresponding transaction list.

2) Read Phase: In the read phase, the record returned for a non-scan operation (e.g., read with primary key) is copied to the transaction's readset, and the written record is copied to its writeset. ROCC treats locked records as dirty data. If the accessed records have been locked by other transactions, the transaction will be aborted immediately.

ROCC maintains predicates for a scan query in the predicate set P. Each scan query may read records from several consecutive logical ranges. To guarantee the consistency of scanning records over multiple logical key-ranges, ROCC constructs a predicate p for each key-range before scanning this key-range. The interleaving of constructing predicate and then scanning the key-range is equivalent to the procedure of adding a read lock on an object and then reading it. Besides, it will not block the write operations. At the time of constructing the predicate p, $p.rd\_ts$ denotes the current version of its corresponding logical range. $p.startkey$ and $p.endkey$, representing the precise scope of this predicate, are set after finishing the scan over a logical range. Finally, this predicate p is added into the predicate set P which is saved in the transaction's local memory.

Instead of copying all scanned records into the readset,

Algorithm 1: Commit Protocol (T)

Data: $ReadSet$, $WriteSet$, scan predicate set P

```
1  for record ∈ sorted(WriteSet) do
2      lock(record);
3      range_list = range_manager.find_range(record);
       range_list.register(T);
4  end
5  generate_commit_timestamp();
   /* Validation Phase                                  */
6  for record ∈ ReadSet do
7      if record.rd_ts ≠ reread_ts(record) or
        record is locked then
8          abort();
9      end
10 end
11 for p ∈ P do
12     p.v_ts = range_list[p.id].get_version();
13     if p.cover == True then
14         if p.v_ts ≠ p.rd_ts then
15             abort();
16         end
17     end
18     else
19         //get overlapping txns from the list with p.id
           ovlap_txns =
           txns_between(p.rd_ts, p.v_ts);
20         for recd in ovlap_txns.write_set do
21             if recd ∈ [p.start, p.end) then
22                 abort();
23             end
24         end
25     end
26 end
   /* Write Phase                                       */
27 for record ∈ W do
28     write(record);
29     unlock(record);
30 end
```

ROCC tracks the returned records of a scan query using predicates. Since ROCC avoids copying the scanned records and their versions into transactions' local memory, it reduces the overhead of maintaining the scanned records.

3) Validation Phase: Before validating the transaction, ROCC adds locks on all records in the writeset according to the key order. For each write, the transaction registers itself to the transaction list of the corresponding logical range at step 3 in Algorithm 1. The registration will change the version of a logical range indicating that the transaction has an intention to modify the records in this logical range. Other concurrent transactions which have scanned this range can detect the read-write conflict by checking the version of this list in the validation phase. When all the writes are locked and registered to the corresponding logical ranges, the transaction is assigned the commit timestamp as the serialization point.

The validation for non-scan operations is similar to the standard OCC protocol. From step 6 to 10, ROCC compares the version of the read records in the readset to check whether they have been modified by other transactions. ROCC validates the range scan operation by verifying all predicates in the predicate set P from step 11 to 26. Firstly, for each predicate p, ROCC reads the version of the transaction list with $p.rangeID$ and save it to $p.v\_ts$. If the flag $p.cover$ is true, instead of checking writes from the writeset of other concurrent transactions, ROCC only needs to check whether the version of the transaction list has been changed (i.e. $p.v\_ts \neq p.rd\_ts$), and then decides if the transaction should be aborted. Otherwise, if the flag $p.cover$ is false, ROCC retrieves the overlapping transactions from the transaction list between $p.rd\_ts$ and $p.v\_ts$. For each overlapping transaction committed before the commit timestamp of the validating transaction, if there is any write in the scanned key-range starting from $p.startkey$ to $p.endkey$, the transaction should be aborted.

4) Write Phase: If all the validations pass, the transaction commits the writes with its commit timestamp and releases write locks. Transactions do not release their resources whenever they commit or abort until these resources are no longer used in any logical range. Transaction resources are finally reused by asynchronous garbage collection.

D. Serialization Analysis

The validation phase of ROCC contains two parts: (1) validating in record level for the read/write on single records and (2) validating in range level for the scan. In record level, ROCC can be regarded as a kind of LRV method (similar to Silo [12]) which detects read-write and write-write conflict and ensures a serializable transaction schedule. In range level, we prove that validating the predicates of a scan query still guarantees the serializability of transactions.

The range validation mechanism of ROCC is equivalent to adding predicate locks to the scanned ranges. A predicate lock prevents other transactions from inserting, updating and deleting satisfied records. Thus, the predicate lock can be viewed as acquiring read locks for all possible (existing or non-existing) records that satisfy the predicate.

ROCC commit protocol is equivalent to acquiring predicate locks at the beginning of scan and releasing them after the transaction is committed/aborted. The reasons are as follows. (1) A transaction registers to the corresponding logical range after it has locked the record; (2) the version of the logical range (i.e., rd_ts) is acquired before scanning any records in the read phase; (3) the version of the logical range (i.e., v_ts) is acquired again after the commit timestamp is generated. The record would be successfully scanned only when it is not locked, otherwise, the transaction would be aborted. Thus, if the scan operation succeeds, all read operations in this scan obtain the committed version of records. We ensure that the scanned records are not changed before the validating

transaction is committed by examining all the transactions with a commit timestamp between rd_ts and v_ts for conflict detection.

Lemma 1: A transaction T can be committed iff no committed update, insert and delete of overlapping transactions satisfies any *predicate* in the predicate set of T.

Proof: We suppose that a record x_v scanned by T_i has been modified by another transaction T_j which is committed before T_i. We represent this happen-before relation as $Scan(x_v) \prec W(x_v)$. Assuming that the scanned range is $range_k$. There is a read-write conflict between T_i and T_j, and the commit order of the two transactions is $Commit(T_j) \prec Commit(T_i)$. According to the read phase of ROCC protocol, a predicate p will be created which maintains rd_ts before scanning x_v, i.e., $Get(rd\_ts) \prec Scan(x_v)$. It is certain that T_j should register itself to $range_k$ after $W(x_v)$ and before $Commit(T_j)$, denoted as $W(x_v) \prec Register(T_j) \prec Commit(T_j)$. Since $Commit(T_i) \prec Get(v\_ts)$, we can conclude that $Get(rd\_ts) \prec Register(T_j) \prec Get(v\_ts)$. Therefore, the T_j must exist in the list of $range_k$ between rd_ts and v_ts. Since T_i should validate p in the validation phase and check the transactions in $range_k$ between rd_ts and v_ts, T_j would be examined and T_i would be aborted. Accordingly, any modified record by the overlapping transactions in the scanned range would lead T_i to be aborted. ■

Lemma 2: A transaction T can be committed iff no updates, inserts or deletes which are committed by overlapping transactions intersect with the read set of T (guaranteed by the LRV).

Theorem 1: Our ROCC protocol is equivalent to a serial history in the commit timestamp order.

Proof: According to Lemma 1, if a transaction T is committed, no data committed by recent transactions intersect with any read predicates of T. Thus, this is equivalent to adding a long duration predicate read lock before the read phase and releasing the lock after the time of commit timestamp. According to Lemma 2, if a transaction T is committed, no data committed by recent transactions intersect with any read set of T. Thus, this is equivalent to adding a long duration read lock before the read phase and releasing the lock after the commit timestamp. For updates, ROCC exclusively locks the written records at the time of commit timestamp. Thus, ROCC generates the transaction schedule which is serially equivalent to the order of transaction's commit timestamp. ■

IV. Range Validation Cost Analysis

It is obvious that the validation cost is related to the consumed CPU cycles for validating records or transactions. We measure the cost by the number of records or transactions to be validated. LRV validates the scanned records by re-reading them and checking their versions. Thus, the validation cost of a scan transaction is linearly proportional to the number of scanned records when using LRV. In contrast, GWV examines all of its concurrent transactions to avoid read-write conflicts in the validation phase. Therefore, for each scan transaction, the validation cost of GWV is mainly affected by the average number of concurrent transactions. To optimize the validation, RV uses logical ranges to reduce the number of transactions to be validated. There are three major factors influencing the performance of RV, explained in the following section.

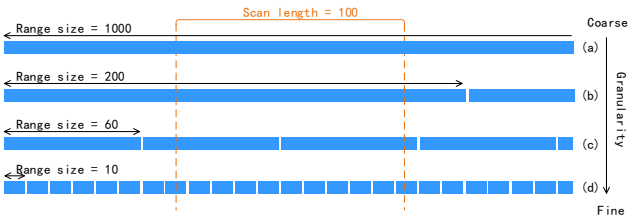


Fig. 4: A scan example under various partitioning granularity.

A. Influence Factors to RV

The first factor is the average number of scanned logical ranges. It is determined by *partition size* (i.e., number of records in a logical range) and *scan length*. We illustrate how partition size affects the performance in Fig. 4, which presents several cases of a scan operation covering a various number of ranges. In Fig. 4(a), it shows a coarse-grained partitioning where the range size is 10x larger than the scan length. The scan transaction needs to check all the transactions registered in its scanned logical ranges, though there might be only 10% transactions that perhaps have conflicts with it. Fig. 4(d) shows the other extreme case of partitioning granularity, where the range size is 10x smaller than the scan length. However, this fine-grained partitioning cannot further improve the performance by filtering out more unrelated transactions. In this extreme case, smaller partition size in Fig. 4(d) would only introduce the more overhead of maintaining predicates and transaction registrations. Two kinds of reasonable partition sizes are presented in Fig. 4(b) and (c), where the scan length is 0.5x-2x of the range size. We empirically demonstrate the effectiveness of the partitioning method in the experiment of Section V-F.

The second factor is the *skewness* of a workload. Since the contention is low in a non-skew workload, the accesses of a transaction are mostly not intersected with others. In this case, it is efficient for ROCC to filter out a large number of non-conflict transactions. In a high-skew workload, since a small amount of logical ranges receive most of the update requests, ROCC needs to validate a large number of transactions registered to these hot ranges. Therefore, the benefit of partitioning logical ranges will decrease. It should be noted that the performance of LRV, GWV, and RV suffers from high contentions. For example, if transactions access records only in a hot logical range, RV needs to register and validate all concurrent transactions in a single transaction list. In this case, RV behaves similarly to GWV as both of them use a centralized list to track the concurrent transactions. Thus, under the hybrid workload with high-skewed data access, RV is at least not worse than GWV. We evaluate and explain how the workload skewness impacts on the performance of

different validation schemes in the experiment of Section V-E.

The third factor is the size of the circular array. To ensure that no updates violate the scanned predicates, RV needs to maintain the pointer of concurrent transactions by the corresponding circular arrays. If using a small size of the circular array, highly concurrent transactions may be blocked because of no available slots for transaction registrations, especially in the high-skew workload. The problem is alleviated by allocating a large size of the circular array, but it consumes large memory space. As a result, there is a trade-off between memory usage and performance. In our experiment, we allocate 5000 slots for each array and avoid transactions being blocked at the time of registration. Each circular array needs 40KB memory space. The key space is divided into 16384 logical ranges for the YCSB workload, and the total space used by circular array is at most 600MB.

V. Experiments

A. Experimental setup

In this section, we present the evaluation of the ROCC protocol implemented in the DBx1000 codebase. DBx1000 is an in-memory DBMS prototype [21] which provides a pluggable framework to integrate different concurrency control protocols for performance comparison [22]. Existing integrated protocols include Silo, HEKATON, 2PL, etc. We compare the validation scheme of ROCC with those used by Silo and Hyper. In order to validate the key-range, Silo adopts Local Readset Validation mechanism and needs to re-read the scanned range. Hyper uses Global Writeset Validation method for checking the write sets from all other concurrent transactions. We set up the experiments on the machine described in Table I. The single machine equipped with 192GB DRAM has 20 physical cores, and each one hosts two hardware threads.

B. Workloads

YCSB. The workload E of YCSB benchmark contains read, write and scan operations. The degree of workload skew is tuned by changing the parameter (θ) of Zipfian distribution. The read and write keys are selected according to the predefined Zipfian distribution. For scan operations, the start key of a scanned range is also selected by the same Zipfian distribution. We create two types of transactions to generate the hybrid workload by configuring the YCSB workload generator. The first one is a simple transaction which consists of five read/write operations, and each operation accesses a single record. The second is a bulk processing transaction containing four read/write operations and one key-range scan operation with a fixed length. The hybrid workload is a mixture of 90% simple transactions and 10% bulk process transactions. We generate four kinds of workloads with different Zipfian distributions which are the characteristic of workload skew:

- No-skew: Uniform distribution.
- Low-skew: A Zipfian distribution where θ=0.7.
- Medium-skew: A Zipfian distribution where θ=0.88.
- High-skew: A Zipfian distribution where θ=1.04.

By default, the access pattern of operations is low-skew. The data in the YCSB table is initialized to contain 10 million records. The YCSB table is partitioned into 16384 equal-sized logical ranges, where each one covers 610 records. A logical range is associated with a lock-free circular list. For each experiment, we run 100k transactions five times and measure the average throughput, latency, and abort rate.

Modified TPC-C. TPC-C is a standard online transaction processing (OLTP) benchmark of mixed read-only and update-intensive transactions with a non-uniform data distribution. We modify the TPC-C according to the actual application requirement. In the recent online sales market, there is a growing demand for processing bulk scan operations in its workload. For instance, to promote consumption in the China Single's Day, the online shops often launch shoppers rewards program to reward the top shopper who has purchased the most in a certain period of time. According to the schema of TPC-C, we create a new bulk transaction that executes a scan operation in the customer table to figure out the top shopper in a randomly selected district who made the highest total payment from a certain start time, and this customer would be given the reward money to his balance. Thus, the bulk processing transaction of the modified TPC-C consists of one scan operation that randomly accesses a range in the customer table, and three updates on the customer, district and warehouse tables respectively.

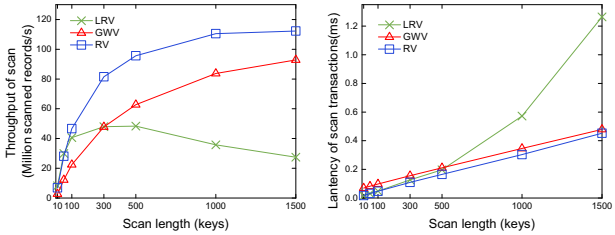
In the experiment, the number of processing threads is equal to the number of TPC-C warehouses. The original TPC-C transactions access its local warehouse and other warehouses with the probability of 85% and 15% respectively. All bulk processing transactions only scan the local warehouse. Therefore, the bulk transaction has conflicts with the cross-warehouse Payment transactions. The hybrid workload is a mixture of 40% Payment transactions, 40% New-Order transactions, 10% bulk processing transactions, 4% OrderStatus, 4% Delivery and 2% StockLevel transactions. With 40 warehouses, the customer table is partitioned into 2000 logical ranges with equal size where a range contains 600 records.

TABLE I: Details of experimental environment

OS	CentOS 7
CPUs	2-socket Intel(R) Xeon(R) E5-2630 v4 Clockspeed 2.2GHz; Turbospeed 3.1GHz 20 physical cores; 40 logical cores (threads) 25M L3 cache
Memory	12*16GB DDR4 2400 MHz

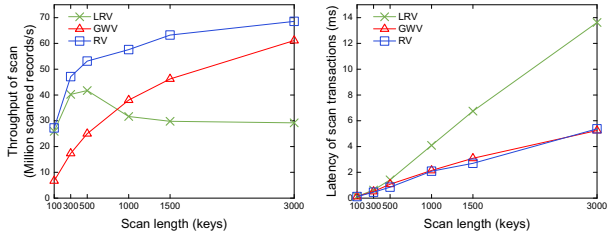
C. Scan Performance

In this experiment, we start 40 processing threads (bound to 40 CPU cores) and run YCSB and modified TPC-C workloads to evaluate the performance of three key-range validation schemes under hybrid workloads. We combine 90% simple transactions and 10% bulk scan transactions, where the scan length varies from 10 to 1500 keys for hybrid YCSB workload and from 100 to 3000 keys for the modified TPC-C workload.



(a) Throughput of scanned records (b) Latency

Fig. 5: Results of the hybrid YCSB.



(a) Throughput of scanned records (b) Latency

Fig. 6: Results of the modified TPC-C workload.

Fig. 5(a) presents the overall throughput of scan transactions and Fig. 5(b) shows the average latency of scan transactions. In Fig. 5(a), as the scan length increases, the performance of LRV, GWV and RV reflects a growing trend proportional to scan length at the beginning. However, LRV's performance appears to decrease while the scanned key-range covers more than 300 keys. In the workload with the longest scan length (1500 keys), the performance of RV is about 3 times more than LRV and about 22% better than GWV. The overall trend demonstrates that RV can adapt to different scan length and achieve the best scan performance. From Fig. 5(b), it's observed that LRV suffers from a sharp increase in the latency of scan transactions when the scan length is varied from 500 keys to 1500 keys. On the other hand, the latency of RV and GWV increases only slightly and the latency of RV is always lower than GWV.

The result of modified TPC-C workload is presented in Fig. 6. Fig. 6(a) and (b) show the performance and latency of scan transactions respectively. It's observed that the three validation schemes demonstrate similar results to that of hybrid YCSB workload.

Result Analysis: The LRV strategy maintains all the scanned records in transaction local memory and re-reads them for validation which doubles the execution time before committing the transaction. Accordingly, the validation cost of LRV is mainly affected by the number of keys of scanned range. It can be seen from Fig. 5(b) and Fig. 6(b) that the latency of LRV increases at a higher rate with a larger scan range. Therefore, it's inefficient for LRV in the case of transactions containing large range scan (e.g., 1500 keys). In contrast, both GWV and RV strategies validate the overlapping transactions that have modified its scanned logical ranges. Their validation costs are determined by the number of concurrent transactions. Under short range scan workload (e.g., 100 keys), most transactions would be executed and committed in a relatively short time. In this situation, GWV takes too much effort on validating a large amount of recently committed transactions and thus leads to the low performance. Compared to GWV, instead of validating all the concurrent transactions, RV only validates a small amount of transactions having writes in scanned logical ranges. In other words, RV uses logical ranges to filter out unrelated transactions in the validation phase which reduces the number of overlapping transactions to be validated. In addition, RV directly validates the key-range at range level if the scanned range is fully covered by the predicate. It can be seen that, compared to GWV, RV scheme significantly enhances the throughput under short range scan workloads and is also efficient for large range scan workloads. Compared to LRV, the main overhead of both RV and GWV is to track transactions by a list attached to each logical range or by a global list respectively. Because of this, it can be observed that the performance of RV is 10% worse than LRV under the very short range scan workload (e.g., 10 keys). However, under large range scan workloads, this registration overhead becomes insignificant compared to the gain from accelerating the validation. We make detailed overhead analysis for RV in Section V-H.

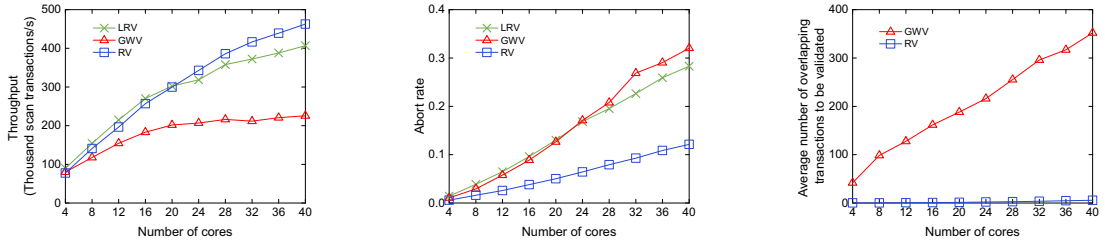
D. Scalability

In this section, we evaluate the scalability of three validation schemes under YCSB and modified TPC-C workloads. The number of threads is varied from 4 to 40 and each thread runs the workload on an individual CPU core. The length of range scan for the YCSB workload is fixed to 100 keys, and it's about 3000 keys for the modified TPC-C workload. The experimental results of YCSB and modified TPC-C are presented in Fig. 7 and Fig. 8 respectively.

Using more concurrent threads to run the workload can cause more data access collisions. Thus, it introduces transaction conflicts for scan transactions and leads to more aborts. As the performance of YCSB workload is shown in Fig. 7(a), LRV is better with less than 20 threads but the rising trend of the performance becomes slow with more than 24 threads. RV method achieves near-linear scalability and the performance is the best with 40 threads. GWV is always worse than RV and LRV. With 40 threads, the performance of RV is 53% higher than GWV and 18% higher than LRV.

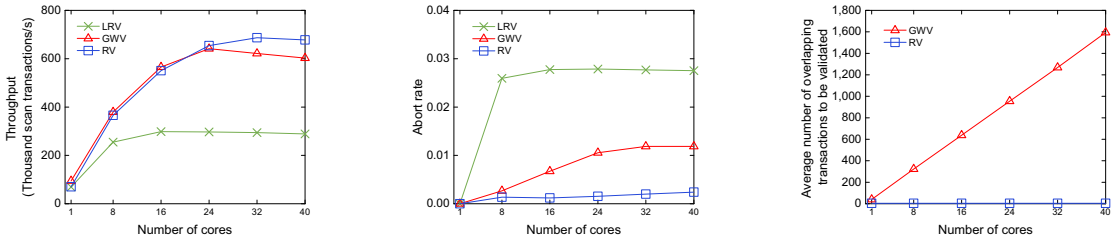
The modified TPC-C workload is configured to scan a large range which contains 3000 customers. Under the setting of 3000 scan keys, the performance in Fig. 8(a) reflects that LRV is poorly scaled up and almost stops growing by the 8 threads. RV achieves the optimal performance at the 32 threads and then declines slightly. GWV has the ascending curve of the performance similar to RV, but starts to decline at the 24 threads.

Result Analysis: We analyze the results by checking the average abort rate of scan transactions and the average number of overlapping transactions to be validated by GWV and RV. The average number of overlapping transactions determines the validation cost of RV and GWV. For LRV, it re-reads all



(a) Throughput of scan transactions (b) Abort rate of scan transactions (c) Validation cost

Fig. 7: Results of YCSB workload with increasing cores from 4 to 40. The scan length is over 100 keys.



(a) Throughput of scan transactions (b) Abort rate of scan transactions (c) Validation cost

Fig. 8: Results of the modified TPC-C workload with increasing cores from 1 to 40. The scan length is over 3000 keys.

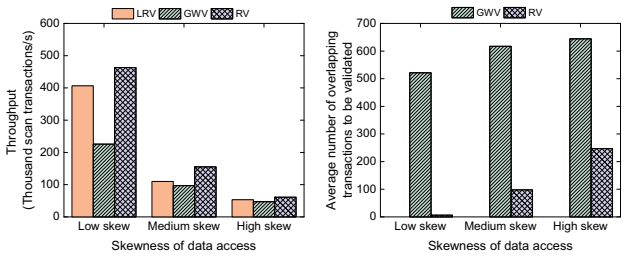
the scanned records in the validation phase which consumes almost the same CPU cycles for scanning them. Thus, the validation cost of LRV is proportional to the length of scanned keys.

Under the YCSB workload, LRV is required to validate 100 scanned records before committing a transaction. In the contrast, RV only needs to validate a few overlapping transactions containing writes. The number of writes is much less than that of scanned records validated by LRV. GWV needs to validate more than three hundred writes from all the overlapping transactions, and Fig. 7(c) demonstrates that its validation cost is for more than RV. In the modified TPC-C workload, the scalability of LRV is poor for its huge cost of maintaining a large amount of scanned records and validating them. This has further led to a rise abort rate for LRV in both workloads, which is shown in Fig. 7(b) and Fig. 8(b).

The validation cost of GWV depends on the count of recently committed transactions. Under the multi-core setting, GWV would take more time in the validation phase because of more concurrent transactions. In the YCSB workload, although the writes of concurrent transactions will register to 16k logical ranges, transactions only in 5 logical ranges need to be validated (where the 3000 scanned keys cover 5 logical ranges at the most and each range contains 610 records). In Fig. 7(c) and Fig. 8(c), it can be seen that GWV validates a large amount of concurrent transactions and RV only validates a very small amount of them. In addition, GWV uses a centralized list to maintain committed transactions which are likely to become a bottleneck in the environment of multi-cores.

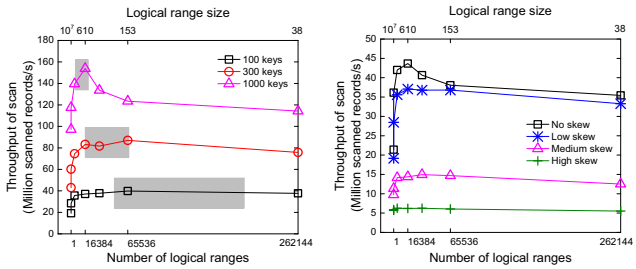
E. Impact of Skew Workloads

We start 40 threads to execute three skew workloads and make a comparison of the three validation strategies. The scan length is over 100 keys in the three workloads. The experimental results are presented in Fig. 9. As it is shown in Fig. 9(a), RV achieves the best performance for the low-skew workload. Under the medium-skew workload, RV only performs slightly better. The three validation schemes have comparable performance when the workload is high-skew. The skew workload causes high contention and thus leads to more abort for scan transactions. In the low-skew workload, each logical range has relatively balanced transaction registrations. For the RV, a transaction only needs to validate a small number of transactions containing writes on its scanned ranges. Thus, RV is more efficient in the low-skew workload because most unrelated transactions can be effectively filtered out. Fig. 9(b) shows the average number of transactions validated by GWV is much more than RV. However, under medium-skew and high-skew workloads, the update and scan operations fall into several logical ranges. It implies that partitioning ranges cannot help filter out unrelated transactions because most of the transactions are really related. Fig. 9(b) indicates that the validation cost of RV increases proportionally with the degree of skewness but is still lower than the validation cost of GWV.



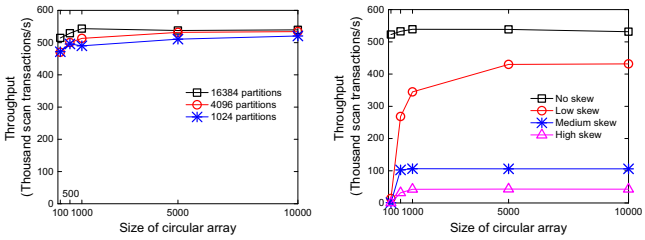
(a) Throughput (b) Validation cost

Fig. 9: Throughput of scan under skew workload.



(a) Workloads of various scan length. (b) Under various skew workloads.

Fig. 10: Scan throughput with various partitioning granularity.



(a) Varying the size of circular array (b) Under skew workloads

Fig. 11: Scan throughput under various sizes of circular array.

F. Partitioning Granularity

In this section, we evaluate the scan performance of RV under different partitioning granularity and various workload skew. The total number of keys is fixed to 10 million, and Table II gives five typical configurations of the number of logical ranges and the size of the logical range. The results are presented in Fig. 10. In Fig. 10(a), it shows how the scan performance varies with the number of partitioned logical ranges. The shadow area indicates a reasonable partition size for the workload according to the previous analysis in Section IV. It's observed that the scan performance get better with the increasing number of ranges from 1 to 16384 for all workloads with different scan length. The reason is that the proper partitioning granularity can effectively filter out the unrelated concurrent transactions and then reduce the validation cost. When the number of logical ranges exceeds 16384, for the workloads with the scan length over 100 and 300 keys, the performance has a stable trend. For the workload scanning 1000 keys, the performance drops nearly 30%. This result demonstrates that more fine-grained partitions are not useful for long scan query. The reason is that when the scan length is much longer than the partition size, smaller partitioning granularity cannot make further reduction on the validation cost but introduce much overhead on maintaining more predicates for scanned ranges. The empirical result shows that RV achieves the best performance when the scanned key-range covers 0.5-3 logical ranges which confirm our analysis in Section IV.

We also evaluate the performance of various partitioning granularity under different workload skew (with the fixed scan length over 100 keys), and the result is presented in Fig. 10(b). It's observed that the more skew workload causes the lower scan throughput. We also find that the performance of non-high skew workloads continues to improve by increasing the number of ranges at the beginning, and when the number exceeds 16384 the performance begins to decrease. For the high-skew workload, as most of the data accesses focus on a few records, coarse-grained or fine-grained partitions really do not matter.

TABLE II: Partitioning Granularity

range count	1	16	4096	16384	262144
range size	10^7	6×10^5	2.4×10^3	610	38

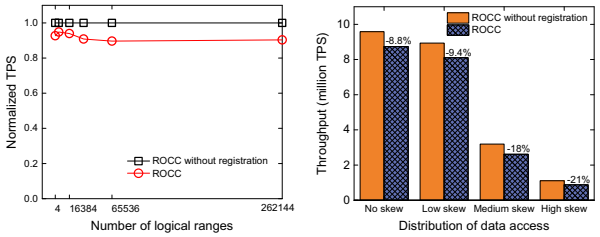
G. Circular Array

In this section, we evaluate how the size of circular array affects the scan performance. We start 40 threads and run hybrid YCSB workload with fixed scan length over 100 keys. We change the array size from 100 to 10000 and evaluate the performance under various partition granularity and different workload skew. The results are presented in Fig. 11. As we can see from Fig. 11(a), the size of the circular array has a very small impact on the performance under different partition granularity. In Fig. 11(b), we find that the small size of circular array has a negative influence on the performance under the skew workloads. It's always better to allocate a large circular array, but it may consume more memory. Tuning the array size is a trade-off between memory consumption and performance. We empirically allocate 5000 slots for each array. Under the setting of 16384 partitions, it approximately takes 600MB memory space.

H. Overhead analysis

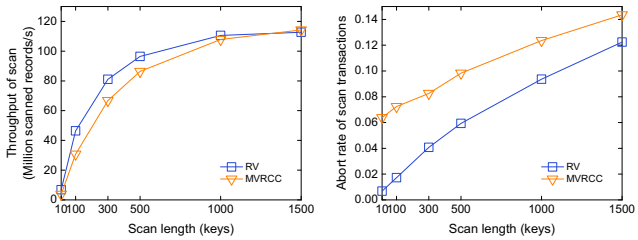
The main overhead of ROCC is caused by registering update transactions to the corresponding list if they have writes in a logical range. In each logical range, ROCC maintains recently committed transactions in an optimized list implementation using a lock-free circular array. The circular array is operated by atomic instructions to reduce the extra cost consumed on registering transactions. However, it still leads to performance overhead in a write-intensive workload which has no bulk process transactions. We use the YCSB workload A to evaluate the overhead of transaction registrations, and the ratio of read to write operations performed in the YCSB workload A is 50/50. Each transaction is configured to consist of five operations without any scan queries. We turn off the transaction registration, referred to as ROCC without registration, and start 40 threads to run the workload. The experimental results are presented in Fig. 12(a) and (b).

Fig. 12(a) shows the comparison with various partitioning granularity settings. The TPS is normalized according to that of ROCC without registration. Since transactions register to the same logical range only once, it would make many registrations in the setting of fine-grained partition. It can be seen that the cost of the registration increases when using more fine-grained logical ranges. In spite of this, the overall overhead is less than 10%.



(a) Varying partitioning granularity (b) Varying skewness of workloads

Fig. 12: Performance under non-scan workloads.



(a) Throughput (b) Abort rate

Fig. 13: Comparison with MVRCC.

Fig. 12(b) presents the performance comparison under different workload skew. The decline of TPS in no-skew and low-skew workloads is less than 10%. The overhead increases to between 18% and 21% in medium-skew and high-skew workloads. In the high-skew workload, it introduces the relatively high overhead caused by so many transactions competing for registering to a few logical ranges.

I. Summary

The RV achieves the best scan performance and scalability under a low-skew hybrid workload. Its advantage becomes more obvious especially when the workload contains larger range scan. In the medium-skew workload, RV performs slightly better and all of the three validation schemes are comparable in the high-skew workload. It should be noted that RV will introduce extra overhead in the case that the workload has no scan queries. In a write-intensive workload, the overhead is between 10% and 20% according to the degree of workload skewness.

VI. Discussion

Deuteronomy introduced a multi-version timestamp ordering concurrency control for logical ranges (MVRCC) [8]. To avoid the phantom problem, the read timestamp of a key-range scan is required to be earlier than that of all uncommitted writes in the scanned ranges. Similar to MVRCC, RV also registers update transactions to the corresponding lists of logical ranges. However, RV has two advantages over MVRCC. Firstly, MVRCC needs to check more transactions than RV does for detecting read-write conflicts in logical ranges. MVRCC adopts a timestamp ordering concurrency control under which transactions are committed in the order of their start timestamps. While updating records in a logical range, transactions register to the transaction list of this range. These transactions are not ordered by their start timestamps in the associated list of a logical range. Thus, before executing scan over a logical range, the transaction needs to examine the entire list to detect conflicts. In comparison, for each scanned range, RV maintains the start timestamp at the beginning of a scan operation and the validating timestamp at validation phase. Therefore, RV only validates the overlapping transactions between the two timestamps instead of all the transactions in the entire list. The second advantage is that the conflict detection of RV is more precise than MVRCC for the boundary of scanned ranges. A scan operation may cover several logical ranges, and the boundary ranges (e.g. the first or/and the last scanned range) are usually not fully covered. RV uses predicates with start and end keys to precisely describe the scanned scope of boundary ranges. At the validation phase, RV examines whether the updated keys satisfy the predicate. In MVRCC, the boundary ranges are treated as fully scanned ranges, and thus it causes more false aborts.

We implement MVRCC in DBx1000 and make a comparison with RV using the hybrid YCSB workload. The number of the logical ranges is 16384 for both MVRCC and RV. The scan performance is presented in Fig. 13(a). As we can see, the performance of RV is 51% higher than MVRCC when the scan length covers 100 keys, and 12% higher when the scan length covers 500 keys. When the scan length exceeds more than 1000 keys, RV and MVRCC have similar performance. Fig. 13(b) demonstrates that the abort rate of MVRCC is always higher than that of RV because of inaccurate detection of the transaction conflicts in boundary logical ranges.

VII. Related Work

A. Optimistic Concurrency Control

In the setting of multi-cores, OCC still has a challenge of scaling out the serializable committing protocol. Silo [12] is based on a single version OCC scheme. In the validation phase, it adopts time-based epochs to avoid centralized bottlenecks of generating serialization point. For the range query, Silo introduces a lightweight phantom detecting method by checking the version of B-Tree nodes which cover the scanned range, but it still needs to maintain all the scanned records and re-scan them before commit. To optimize the performance of OCC protocol under high contention workloads, the mostly-optimistic concurrency control (MOCC) [23] combines pessimistic locking schemes with OCC to avoid having to frequently abort a transaction accessing hot data. BCC [24] reduces the false aborts of transactions by detecting data dependency patterns with low overhead and improves the performance in high contention workloads. Instead of simply aborting a transaction when its validation fails, Wu [25] introduced a transaction healing mechanism to restore the non-serializable operations and heal inconsistent states caused by transaction conflicts.

B. Hybrid Transactional/Analytical Processing

A recent paper surveyed the systems supporting hybrid transactional/analytical processing (HTAP) [26]. A single

transaction under the true HTAP workload consists of both write operations as well as analytical queries which often require range scan operations to evaluate aggregation functions. Multiple version concurrency control (MVCC) has a particularly attractive feature that write and read operations don't block each other, and thus it was widely adopted by HTAP-enabled systems (e.g. Hekaton, Hyper and SAP HANA etc.).

Hekaton [11] designed multi-version OCC (MV-OCC) scheme and it's reported that MV-OCC has better performance compared to the scheme of combing multiple version storage with the lock-based concurrency control. To ensure serializability, in the validation phase Hekaton checks whether the versions of each record in the readset have been changed. For range queries, Hekaton re-scans the range to detect serializability violation and prevent phantom problems.

Hyper is a high-performance database system towards OLTP&OLAP mixed workloads [15]. It proposed a fast MVCC protocol to achieve the good scan performance and also guarantee the serializability for the heterogeneous workload. Each transaction keeps read predicates on its predicate tree. Instead of detecting conflicts by checking the records in the readset, Hyper validates the predicate tree against the writes from other recently committed transactions. In this paper, the validation strategy is referred to as Global Writeset Validation (GWV). GWV may not be applicable for the update-intensive workloads, where the validating transaction needs to detect conflicts by checking massive updates from a large number of concurrent transactions. In this situation, ROCC restricts the conflict detection within logical ranges and thus reduces the count of transactions be validated.

VIII. Conclusion

In this paper, we developed an efficient and scalable Range Optimistic Concurrency Control (ROCC) scheme for the emerging OLTP and bulk processing workload. ROCC uses the logical ranges to track the potential conflicting transactions and to reduce the number of transactions to be validated. It not only can avoid the prohibitive validation cost of re-scanning the large key-range, but also it does not require system-wide lookup for overlapping transactions. Our evaluation results demonstrate that ROCC can maintain high performance for short and large key-range scan transactions with serializability guarantee. In addition, ROCC provides better scalability and achieves comparable performance in high-contention workloads.

Acknowledgment

This work is partially supported by National Hightech R&D Program (863 Program) under grant number 2015AA015307, NSFC under grant number 61432006 and 61332006, and Guangxi Key Laboratory of Trusted Software (kx201602). The corresponding author is Peng Cai. We thank anonymous reviewers for their very helpful comments.

References

[1] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql servers memory-optimized oltp engine," in *SIGMOD*. ACM, 2013, pp. 1243–1254.

[2] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," *SIGMOD*, vol. 40, no. 4, pp. 45–51, 2012.

[3] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE*. IEEE, 2011, pp. 195–206.

[4] K. Ren, J. M. Faleiro, and D. J. Abadi, "Design principles for scaling multi-core oltp under high contention," in *SIGMOD*. ACM, 2016, pp. 1583–1598.

[5] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang *et al.*, "H-store: A high-performance, distributed main memory transaction processing system," *VLDB*, pp. 1496–1499, 2008.

[6] D. B. Lomet, "Key range locking strategies for improved concurrency," *VLDB*, pp. 655–664, 1993.

[7] C. Mohan, "Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes," in *VLDB*. Morgan Kaufmann Publishers Inc., 1990, pp. 392–405.

[8] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang, "Multi-version range concurrency control in deuteronomy," *VLDB*, vol. 8, no. 13, pp. 2146–2157, 2015.

[9] J. Jordan, J. Banerjee, and R. Batman, "Precision locks," in *SIGMOD*. ACM, 1981, pp. 143–147.

[10] R. Johnson, I. Pandis, and A. Ailamaki, "Improving oltp scalability using speculative lock inheritance," *VLDB*, pp. 479–489, 2009.

[11] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-performance concurrency control mechanisms for main-memory databases," vol. 5, no. 4. VLDB Endowment, 2011, pp. 298–309.

[12] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *SOSP*. ACM, 2013, pp. 18–32.

[13] B. Momjian, *PostgreSQL: introduction and concepts*. Addison-Wesley New York, 2001, vol. 192.

[14] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, pp. 213–226, 1981.

[15] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," in *SIGMOD*. ACM, 2015, pp. 677–689.

[16] M. Reimer, "Solving the phantom problem by predicative optimistic concurrency control," in *VLDB*. Morgan Kaufmann Publishers Inc., 1983, pp. 81–88.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *ACM Symposium on Cloud Computing*, 2010, pp. 143–154.

[18] J. N. Gray, R. A. Lorie, and G. R. Putzolu, "Granularity of locks in a shared data base," in *VLDB*, ser. VLDB. ACM, 1975, pp. 428–451.

[19] H. Plattner, "The impact of columnar in-memory databases on enterprise systems: implications of eliminating transaction-maintained aggregates," *VLDB*, vol. 7, no. 13, pp. 1722–1729, 2014.

[20] D. Lomet and M. F. Mokbel, "Locking key ranges with unbundled transaction services," *VLDB*, pp. 265–276, 2009.

[21] in *DBx1000. https://github.com/yxymit/DBx1000*.

[22] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "Tictoc: Time traveling optimistic concurrency control," in *SIGMOD*, vol. 8. VLDB Endowment, 2016, pp. 209–220.

[23] T. Wang and H. Kimura, "Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores," *VLDB*, vol. 10, no. 2, pp. 49–60, 2016.

[24] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang, "Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases," *VLDB*, vol. 9, no. 6, pp. 504–515, 2016.

[25] Y. Wu, C.-Y. Chan, and K.-L. Tan, "Transaction healing: Scaling optimistic concurrency control on multicores," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD, 2016, pp. 1689–1704.

[26] F. Özcan, Y. Tian, and P. Tözün, "Hybrid transactional/analytical processing: A survey," ser. SIGMOD, 2017.