Adaptive Optimistic Concurrency Control for Heterogeneous Workloads

Jinwei Guo[†], Peng Cai[†§*], Jiahao Wang[†], Weining Qian[†], Aoying Zhou[†]
[†]School of Data Science & Engineering, East China Normal University
[§]Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology
{guojinwei,jiahaowang}@stu.ecnu.edu.cn, {pcai,wnqian,ayzhou}@dase.ecnu.edu.cn

ABSTRACT

Optimistic concurrency control (OCC) protocols validate whether a transaction has conflicts with other concurrent transactions after this transaction completes its execution. In this work, we demonstrate that the validation phase has a great influence on the performance of modern in-memory database systems, especially under heterogeneous workloads. The cost of validating operations in a transaction is determined by two main factors. The first factor is the operation type. An OCC protocol would take much less cost on validating a single-record read operation than validating a key-range scan operation. The second factor is the workload type. Existing schemes in OCC variants for validating key-range scan perform differently under various workloads.

Although various validation schemes share the same goal of guaranteeing a transaction schedule to be serializable, there are remarkable differences between the costs they introduced. These observations motivate us to design an optimistic concurrency control which can choose a low-cost validation scheme at runtime, referred to as adaptive optimistic concurrency control (AOCC). First, at transaction-level granularity, AOCC can assign a validation method to a transaction according to the features of its operations. Furthermore, for each operation in a transaction, the validation method is selected according to not only the number of accessed records but also the instant characteristics of workloads. Experimental results show that AOCC has good performance and scalability under heterogeneous workloads mixed with point accesses and predicate queries.

PVLDB Reference Format:
Jinwei Guo, Peng Cai, Jiahao Wang, Weining Qian, Aoying Zhou. Adaptive Optimistic Concurrency Control for Heterogeneous Workloads. *PVLDB*, 12(5): 584-596, 2019.
DOI: https://doi.org/10.14778/3303753.3303763

1. INTRODUCTION

Concurrency control, as the fundamental technology of transaction processing, has been a topic of intensive study for several decades. The overhead of disk IO and buffer management has disappeared for the main-memory database systems which are usually deployed on multi-core and big-memory machines. The concurrency control itself—which guarantees the isolation property of ACID transactions—has become the major bottleneck. A great number of studies are focused on designing and implementing lightweight pessimistic or optimistic concurrency control protocols to achieve better scalability with more and more CPU cores [17, 22, 27]. In addition, different concurrency control schemes are combined to improve the OLTP performance under high-contention workloads [28].

Recently, a kind of emerging applications (such as online fraud detection and financial risk analysis) requires real-time analytics on transactional data. A transaction in these applications may mix OLTP-style operations and analytic queries, known as hybrid transactional and analytical processing (HTAP) [8, 15]. Lightweight concurrency control protocols mainly optimize conflict detection at the row level and lack efficient mechanisms to guarantee a serializable schedule for HTAP transactions. Under heterogeneous workloads, the cost introduced by concurrency control schemes has a significant impact on overall performance, and it has received surprisingly little attention. In this paper, we demonstrate that the validation methods should be adaptively selected according to the type of operation in a transaction and the characteristics of its current workload.

Optimistic concurrency control (OCC) protocol divides the execution of a transaction into three phases: read, validation and write. A transaction validates its reads and writes to check whether it breaks the serializable schedule in the validation phase. From a perspective of which records are validated, we group the validation mechanisms into two categories: local read-set validation (LRV) and global write-set validation (GWV). In LRV, a transaction uses a local read set to track all read records and their versions. In the validation phase, the committing transaction only checks all records in its local read set to find whether their versions are changed by other concurrent transactions during the read phase. Instead of keeping read records in the transaction context, a transaction using GWV only keeps a set of predicates with respect to WHERE clauses in all SQL queries. When a transaction enters the validation phase, GWV checks whether these predicates are satisfied by the write sets of other concurrent transactions, which are stored in a global recently-committed list.

*Peng Cai is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 12, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3303753.3303763

However, none of these validation mechanisms can be suitable for different kinds of workloads. It is clear that the performance of LRV is sensitive to the size of read-set. For instance, when executing a long scan query, the transaction needs to keep lots of scanned records in its read-set. This would increase the validation cost because of re-reading many tuples. Unlike LRV, the cost of GWV heavily depends on the characteristics of real-time workloads. With the increase of concurrent transactions and modified tuples, the cost of validating a predicate grows obviously.

Although OCC scales up nicely on modern hardware, it sustains high abort rates when the workload contains frequent read-write conflicts. Existing works about adaptive CC [21, 23] are mainly concentrated on the combination of OCC and pessimistic two-phase locking (2PL). To exploit the advantage of 2PL, they look for the hot-spot data raising conflicts between concurrent transactions, and adopt 2PL-like methods when a transaction accesses these hot data. However, they neglect the impact of validation cost in the OCC protocol.

In this paper, we propose adaptive optimistic concurrency control (AOCC), which incorporates two validation methods LRV and GWV. We first give a design of AOCC at the transaction-level granularity, which assigns the validation type for a transaction according to the features of its read queries. Furthermore, for each query in a transaction, AOCC at the query-level granularity can choose an appropriate validation method not only according to the number of accessed records, but also on the basis of the instant characteristics of workloads. The following is the list of our main contributions.

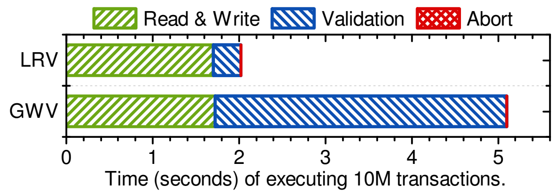
- Based on the analysis of validation strategies in existing OCC variants, we integrate different tracking mechanisms in the read phase in order to adaptively choose a validation method in our proposed AOCC.

- For each operation in a transaction, AOCC dynamically estimates and compares these validation costs incurred by different validation methods, and then uses an appropriate tracking mechanism and corresponding low-cost validation method. In essence, AOCC determines the validation scheme according to the characteristics of each operation in a transaction and the circumstance in which this transaction is running.

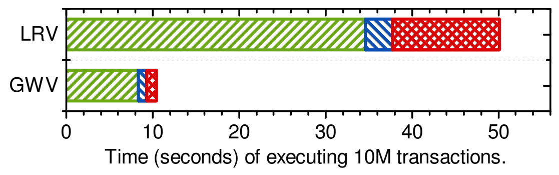
- Experimental results show that our approach effectively reduces validation costs and then achieves good performance under heterogeneous workloads.

2. MOTIVATION

The OCC protocol with fixed validation scheme could cause performance degradation in two kinds of heterogeneous workloads. The first workload allows scan queries for analytics and operational updates to exist in a single transaction, referred to as In-progress HTAP in a recent Gartner report [2]. The second is a mixture of dynamic workloads including an OLTP workload, a real-time/batching data ingestion workload, or an OLAP workload. In this section, we highlight the key requirements for OCC protocols, which aim to adaptively choose validation schemes for transactions under these heterogeneous workloads.



(a) 80% reads, 20% writes, low contention ($\theta = 0.6$), 32 worker threads.



(b) 80% reads, 10% writes, 10% scans (length = 800), high contention ($\theta = 0.9$), 32 worker threads

Figure 1: Performance profile of local read-set validation (LRV) and global write-set validation (GWV) under different YCSB workloads.

2.1 HTAP in a single Transaction

Fraud detection is a great example of HTAP for making real-time business decision. Traditionally, a bank needs to extract the data from OLTP systems to search for fraudulent activities, which means that a fraudulent activity can only be detected after several hours or days. In real-time fraud detection, a payment transaction first finds historical records as the input of a scoring model to calculate the possibility of fraudulent activity, and confirms this payment only after successful verification of this detection. In this case, the query should return strongly consistent records for the predictive model because the latest committed data have a significant impact on the predicted results. If someone had used a lost or stolen credit card one minute before, it's still expected that real-time fraud detection should be sensitive and prevent the fraudulent activity next time.

We simulate this kind of In-progress HTAP workloads by putting scan and update queries in a single transaction. In Figure 1, we profile the performance of both validation methods LRV and GWV using YCSB benchmark. We use a single table with 10 million records. Each transaction contains five queries, each read/write query accesses a single tuple based on a Zipfian distribution with a parameter θ that controls the contention level, and each scan query accesses a range of tuples where the first member follows the same Zipfian distribution. The total execution time of each experiment is divided into three parts: read&write, validation and abort. Figure 1(a) shows the results under a low-contention workload without scan queries. Since the size of read-set of each transaction is very small for the LRV method, its validation time accounts for a small part. The GWV method takes more than 60% of the total execution time in the validation phase. This is because validating all writes of the recently committed transactions is costly under this workload and GWV would suffer from highly concurrent updates. Figure 1(b) illustrates the results under a high-contention workload with scan queries, and each one accesses 800 tuples. Since it is costly for LRV to track and to validate a long scan query, LRV takes more time than GWV not only on the part of read&write but also on the validation part. What's worse, as the time of executing a transaction increases, the abort ratio grows obviously.

2.2 Mixed Workloads

In practical settings, modern enterprise applications based on HTAP-enabled DBMS often require database servers to run dynamically changing mixed workloads. The HTAP workload and data ingestion may co-exist when a database periodically loads data from other systems to support real-time decision making. In this case, if the OCC protocol takes GWV as its validation scheme, a transaction in the HTAP workload might take much time to check many writes generated by the data ingestion tasks. In Section 6.4, we demonstrate that adaptive OCC can perform better than conventional methods under mixed HTAP workloads.

3. VALIDATION SCHEMES

Optimistic concurrency control can adopt backward validation or forward validation. The basic idea of commonly used backward validation is to check the intersections between read&write sets of a committing transaction and those of overlapping, committed transactions. Recently proposed OCC variants are designed to resolve the bottleneck of timestamp allocation [22,27], to alleviate high abort rates for contended workloads [8, 23, 24, 28], and to adapt to analytical workloads [8, 14]. Although these OCC variants focus on addressing different issues, their validation strategies can be grouped into two categories.

3.1 LRV

The first kind of validation scheme requires a validating transaction to re-read tuples to compare their versions with those in the local read set of the transaction [5,10,11,13,22], referred to as local read-set validation (LRV). The key principle of this approach is to ensure that reads and writes of a transaction logically occur at the same time point, which is served as its serialization point. Even in the setting of main-memory database systems, it has an obviously negative impact on performance because of intensive memory accesses for a large number of read tuples. Silo adopted LRV and optimized it for phantom detection by checking the version change of B^+-Tree nodes which cover the scanned key-range [22]. However, it still needs to maintain all read tuples and to re-read them for validation.

3.2 GWV

Another approach, instead of re-reading tuples in local read set, validates whether the write sets of all other concurrent transactions violate the predicates of the transaction's read queries [9, 14, 16], referred to as global write-set validation (GWV). The GWV approach is adopted by HyPer [7, 14], which is designed for HTAP workloads. Compared with LRV, GWV only needs to access a relatively small number of modified tuples 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.

4. AOCC ON TRANSACTION LEVEL

As the total execution time of a transaction is tightly related to its queries, the straightforward approach for improving OCC adaptivity is to choose an efficient validation scheme for each transaction according to the type of its read queries. We have observed that LRV is appropriate for a transaction that only contains a set of single point read queries, and GWV is suitable for that with scan queries. Thus, an intuitive idea is to assign LRV to a transaction without scan queries, and GWV to a transaction having scans. In this section, we present tracking mechanisms used in the read phase and validation process in an unified OCC framework combining GWV and LRV on transaction level.

4.1 Tracking Mechanisms

For simplicity, and without loss of generality, we assume that there is only one table t in the database in this paper. Therefore, unless otherwise specified, all transactions' queries operate on the same table t.

Recall that LRV and GWV need to keep different tracking information for the later validation phase. To combine these validation methods in AOCC, a transaction needs to maintain a PredicateSet in addition to a ReadSet and a WriteSet. Besides, a global data structure gList is required to record pointers to transaction contexts.

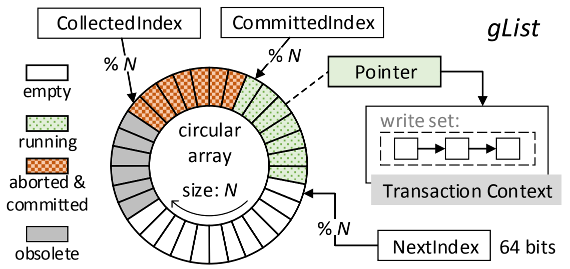


Figure 2: Design of gList. The transaction whose gList index is smaller than CommittedIndex is committed or aborted. In gList, the position with index smaller than CollectedIndex can be reused.

Specifically, gList is implemented by a lock-free circular array with fixed size N, which is illustrated in Figure 2. The space of gList is pre-allocated and can be reused to avoid the overhead of garbage collection. A transaction requests gList for a NextIndex as its commit timestamp by using the atomic compare-and-swap (CAS) instruction. Therefore, any two transactions have different commit timestamps. The CollectedIndex is used to track which slots in gList can be reused. More specific details are out of the scope of this paper and will be covered in a technical report.

Like other OCC protocols, AOCC adopts the WriteSet to maintain the uncommitted writes for each transaction. Each entry in the WriteSet is a quadruple data structure $\langle row, column, old, new\rangle$, where *row* is a pointer to the target tuple in the database, *column* is an updated field of the tuple, *old* and *new* represent before and after images of updated column, respectively. When a transaction enters the validation phase, the pointer of its transaction context that containing its WriteSet is added to gList. AOCC supports three tracking mechanisms:

- $\mathcal{R}$ (ReadSet): In this method, a transaction uses a ReadSet to store accessed tuples with their versions. When entering the validation phase, the transaction uses LRV to validate the ReadSet.

- $\mathcal{P}_{no\_readset}$ (a predicate without ReadSet): In this mechanism, a transaction maintains a PredicateSet for itself. When executing a read query, the transaction generates a predicate for the query and puts it into the PredicateSet. GWV is used to check each entry from the PredicateSet in the validation phase.

- $\mathcal{P}_{readset}$ (a predicate with ReadSet): Like $\mathcal{P}_{no\_readset}$, a transaction needs to maintain a PredicateSet. When executing a read query, the transaction not only generates a predicate for the query itself but also uses a ReadSet to store accessed tuples. It adds the predicate with its returned ReadSet into the PredicateSet. In the validation phase, the transaction can use LRV or GWV to validate each entry in the PredicateSet.

In the first scheme $\mathcal{R}$, in order to check whether an accessed tuple is modified, we need to record the tuple version. Therefore, each entry in the ReadSet is structured as a two-tuple $\langle row, ts \rangle$, where row is the pointer to the target tuple and ts is the timestamp copied from the tuple's timestamp field when the transaction accesses the tuple.

In the schemes $\mathcal{P}_{no\_readset}$ and $\mathcal{P}_{readset}$, we can store these two types of predicates in the same PredicateSet. One predicate in this set can be seen as the conjunction of propositional functions, each function describes the range condition in one column. In order to achieve this, an entry in the PredicateSet is structured as a triple $\langle type, COLs, set \rangle$, where $type$ denotes the tracking mechanism type for this predicate, $COLs$ contains the range conditions of columns and set is the ReadSet for this query. Note that if a predicate p belongs to $\mathcal{P}_{no\_readset}$ (i.e., $p.type = \mathcal{P}_{no\_readset}$), its own ReadSet is empty (i.e., $p.set = null$). Each $column$ in $COLs$ uses a triple $\langle col, start, end \rangle$ to denote the propositional function "$start \leq col \leq end$", where col is the target column id. Although the WHERE clause of a transaction SELECT statement may be complicated, it can be converted to an equivalent disjunctive normal form. Therefore, a WHERE clause can be tracked by multiple predicates in the PredicateSet. For example,

```
SELECT * FROM t WHERE (C_A>=a and C_B<=b) or C_D=d;

<P_no_readset, {<C_A,a,MAX>,<C_B,MIN,b>}>, null>
<P_no_readset, {<C_D,d,d>}, null>                  P_no_readset
```

The transaction decides to use $\mathcal{P}_{no\_readset}$ to track the SELECT statement. Since WHERE clause of the statement contains two conjunctive clauses, we add two predicates into the transaction's PredicateSet. In this paper, to avoid ambiguity, a read query is always seen as a single conjunctive clause, which can be represented by a predicate.

4.2 Validation Methods

Table 1 summarizes the three tracking mechanisms. The storage structures of these tracking methods are different from each other, and they record different information and adopt different validation methods to detect conflicts between concurrent transactions. In what follows, we introduce validation methods for each tracking mechanism. The pseudocode is illustrated in Algorithm 1.

The tracking mechanism $\mathcal{R}$ uses one type of LRV method—which is labeled as LRV1—to validate the ReadSet of a transaction. The execution of LRV1 is shown by the function valLR1. Recall that a transaction records the accessed tuples with their versions into its ReadSet. When validating the ReadSet, the transaction calls the function valLR1 to check whether the version of each entry in the ReadSet is modified by other concurrent transactions. If the latest timestamp in a tuple is not equal to the timestamp when the tuple is accessed, or the tuple is locked by another one, the result $ABORT$ is returned (lines 3 and 4).

The tracking mechanism $\mathcal{P}_{no\_readset}$ utilizes GWV to check whether a predicate conflicts with WriteSets of other concurrent transactions. When validating a predicate, a transaction txn calls the function valGW with the range [$sIndex$, $eIndex$] of gList, in which all of its overlapping transactions can be found. In particular, for each concurrent transaction in gList, txn checks whether the predicate intersects with each element in the concurrent transaction's WriteSet (lines 19–21). If yes, the result $ABORT$ is returned. It should be noted that if the status of a concurrent transaction in gList is $ABORT$, we just skip it (lines 17 and 18).

The tracking mechanism $\mathcal{P}_{readset}$ can choose one of both validation methods LRV and GWV to verify whether a transaction has conflicts with other concurrent transactions. Obviously, $\mathcal{P}_{readset}$ is a flexible mechanism. When validating a predicate p, the transaction is required to decide which validation method to use. If the GWV method is chosen, we can use the function valGW to validate the predicate p. If LRV is chosen, it should be noted that we use another type of LRV method, denoted by LRV2. The pseudocode of LRV2 is presented in the function valLR2. Unlike LRV1, LRV2 re-executes the predicate in order to get the latest ReadSet for the query (lines 8–10). We compare the new ReadSet with the old ReadSet stored in the predicate ($p.set$). If these two sets are different in size or the elements in the same position of these two sets are different, valLR2 will return the result $ABORT$ (lines 11 and 12).

Phantoms. The columns in a table can be classified into two types: primary key and non-primary key. Therefore, there are four types of queries: point/range query on primary/non-primary key column. To simplify the discussion, we use the term *single point* to indicate the operation

Algorithm 1: Validation Methods

```
   /* Validation Execution for LRV1                        */
1  Function valLR1(readSet)
2      for r in readSet do
3          if $r.ts \neq r.tuple.ts \vee r.tuple.isLocked()$ then
4              return ABORT;
5      return SUCCESS;
   /* Validation Execution for LRV2                        */
6  Function valLR2(predicate)
7      $tSet = \{\}$;
8      $tuples$ = getTuples($predicate$);
9      for tuple in tuples do
10         $tSet$.add($\langle tuple, tuple.ts \rangle$);
11     if $tSet \neq predicate.set$ then
12         return ABORT;
13     return SUCCESS;
   /* Validation Execution for GWV                         */
14 Function valGW(predicate, sIndex, eIndex)
15     for $i = sIndex$ to $eIndex$ do
16         $txn = gList$.get($i$);
17         if $txn.status == ABORT$ then
18             continue;
19         for w in txn.writeSet do
20             if $predicate \cap w \neq \emptyset$ then
21                 return ABORT;
22     return SUCCESS;
```

Table 1: Tracking mechanisms for a read query.

Mechanism	Storage Structure	Validation Strategy	Specified Column	Function
$\mathcal{R}$	ReadSet	LRV1	primary key	`valLR1`
$\mathcal{P}_{readset}$	a predicate with ReadSet	LRV2/GWV	primary/non-primary key	`valLR2/valGW`
$\mathcal{P}_{no\_readset}$	a predicate without ReadSet	GWV	primary/non-primary key	`valGW`

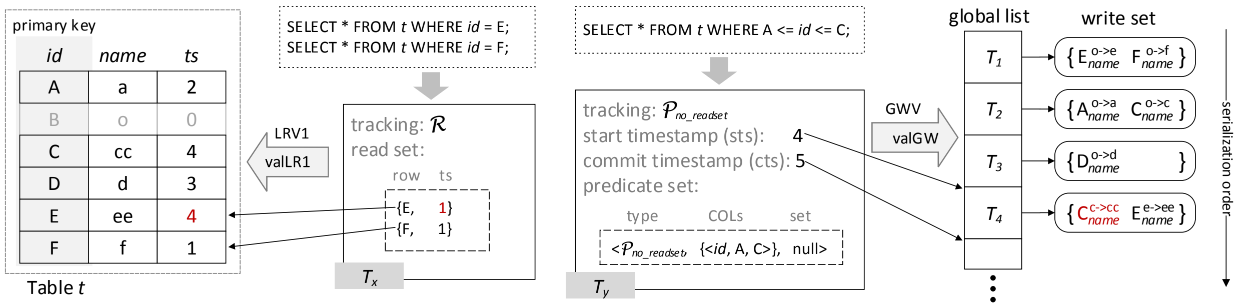


Figure 3: An example of transactions adopting the tracking mechanisms $\mathcal{R}$ and $\mathcal{P}_{no\_readset}$ at the transaction-level granularity. T_x, T_y and T_4 are concurrent transactions.

using primary keys. The tracking and validation schemes of $\mathcal{P}_{readset}$ and $\mathcal{P}_{no\_readset}$ can avoid the phantom anomalies when handling all types of queries [10, 14].

For a range query on primary key columns, the scheme $\mathcal{R}$ prevents the phantoms in a manner similar to Silo. $\mathcal{R}$ needs to record all scanned B^+-tree leaf nodes and their versions. If a single point query attempts to access an absent tuple, it adds an *virtual* entry $\langle tuple, ts = 0\rangle$ to the local ReadSet. Virtual entries denote non-existent tuples in the database. The scheme $\mathcal{R}$ suffers from phantom problem in the case of a query using non-primary keys. Since a non-primary key can map to any number of tuples, validating the ReadSet under the scheme $\mathcal{R}$ cannot detect a newly inserted tuple that satisfies the non-primary key of the executed read query. Consequently, the scheme $\mathcal{R}$ only guarantees no phantom problem for the queries using primary keys.

4.3 Illustrating Examples

Figure 3 illustrates an example of transaction-level AOCC. In this picture, we use $A_c^{o\to n}$ to denote an entry $\langle A, c, o, n\rangle$ in the WriteSets in gList. The table t has two columns *id* and *name*, where *id* is the primary key column. In the table t, there are six tuples with continuous primary keys. The initial values of the fields *name* and *ts* are *o* and 0. Note that the column *ts*—which is used for concurrency control—is invisible to database users. In this example, the transactions T_x, T_y and T_4 are concurrent and T_4 is finished first. We omit write operations from these transactions for more clearly demonstrating how AOCC works at the transaction level. Since T_x has only single point read queries, it is assigned the tracking mechanism $\mathcal{R}$. Thus, when executing the read queries, T_x is required to add $\langle E, 1\rangle$ and $\langle F, 1\rangle$ into its private ReadSet. When T_x enters into the validation phase, it uses LRV1 to detect the conflicts. It calls the function `valLR1` to re-read the tuples E and F and finds that the filed *ts* in E has been modified. This indicates that T_x may conflict with other transactions (i.e., T_4). As a result, T_x should be aborted. On the other hand, the transaction T_y is assigned $\mathcal{P}_{no\_readset}$ due to its scan query. Therefore, T_y records the predicate $\langle \mathcal{P}_{no\_readset}, \{\langle id, A, C\rangle\}, null\rangle$ of the query into its private PredicateSet. When entering the validation phase, T_y calls the function `valGW` (i.e., the validation method GWV used by $\mathcal{P}_{no\_readset}$). It uses the start timestamp ($sts = 4$) and commit timestamp ($cts = 5$) to find the recently-committed concurrent transactions (i.e., T_4) in gList. Then, T_y checks the T_4's WriteSet and finds that the tuple C modified by T_4 is in the range $[A, C]$ of its predicate. Consequently, we should abort T_y as well.

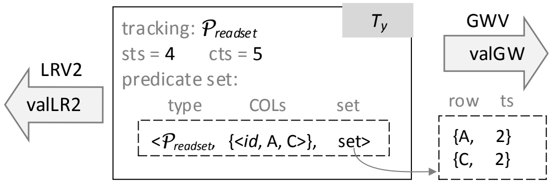


Figure 4: Alternatively, transaction T_y can adopt the tracking mechanism $\mathcal{P}_{readset}$.

Alternatively, AOCC can assign the tracking mechanism $\mathcal{P}_{readset}$ to the transaction T_y, which is illustrated in Figure 4. It should be noted that the members *type* and *set* in the predicate are different from those in the T_y's predicate in Figure 3. When entering the validation phase, T_y chooses one validation scheme from LRV2 and GWV to detect the conflicts with other concurrent transactions.

4.4 Implementation Rules

LRV and GWV can co-exist for one OCC protocol if the tracking mechanisms of LRV and GWV are maintained simultaneously. Therefore, the transaction-level AOCC would be relatively easy to implement. We briefly summarize implementation rules to combine the tracking mechanisms $\mathcal{R}$ and $\mathcal{P}_{no\_readset}$ for validating read queries as follows.

Rule 1: Each tuple in the database contains a timestamp (*ts*) field, which is invisible to the clients and is the commit timestamp (*cts*) of the last transaction updating the tuple.

Rule 2: A global data structure gList is required to store transactions that are being validated or finished. Since the logical index of the last added transaction is unique and monotonically increasing, we use the index to be the *cts* for the transaction in the corresponding position of gList.

Rule 3: Before a transaction is executed, it has to be assigned a validation type. If the transaction only contains single point queries, it is an LRV type using the tracking scheme $\mathcal{R}$; otherwise, it is a GWV transaction using $\mathcal{P}_{no\_readset}$.

Rule 4: In the read phase, the LRV transaction adds each read tuple with the corresponding *ts* into its ReadSet, and the GWV transaction adds the predicate of each query into its PredicateSet. Before executing the first read query, the GWV transaction needs to get the CommittedIndex from gList as its start timestamp (*sts*).

Rule 5: When a transaction enters the validation phase, it is added to gList. Then it checks its ReadSet if it is an LRV transaction; otherwise, it validates its PredicateSet.

4.5 Limitations

It may not be optimal for choosing a single validation method for a transaction. For instance, a transaction contains two key-range scan queries. One is a very short scan that only acquires three or fewer tuples, and another query scans a very large key range containing thousands of keys. It is obvious that LRV would be an efficient method for this short scan and GWV might be the best choice for this large range query. In addition, we summarize two limitations of transaction-level AOCC as follows.

- AOCC on transaction level is only useful in one-shot transaction execution model, where the database system knows the entire logic of a transaction before it starts to execute. In other words, transaction-level AOCC is not suitable for an interactive transaction execution model based on JDBC or ODBC API. This is because it is uncertain about whether a scan query exists at the beginning of a new transaction.
- Choosing a validation method for a transaction does not take the characteristics of real-time workload into consideration. In the case of write-intensive workload, the transaction-level AOCC is possible to perform worse than the OCC with LRV. The reason is that the cost of GWV grows rapidly with the increase of concurrent update transactions. On the other hand, the performance of LRV is not sensitive to how many update transactions execute concurrently.

5. AOCC ON QUERY LEVEL

In this section, we propose the query-level AOCC, which allows a transaction to choose a tracking mechanism and its corresponding validation method for each query. The key intuition is that a transaction can cherry-pick a validation strategy for each query to reduce its total validation cost.

5.1 Protocol Specification

5.1.1 Read phase

In the read phase of a transaction, the execution of a write is similar to other OCC protocols, i.e., each modified tuple is added to its own WriteSet. Thus, we focus on the execution of a read query in our query-level AOCC protocol. We first acquire present tuples from the dataset according to the query's predicate. Then we choose an appropriate tracking mechanism for the query according to the query's type and the feature of real-time workloads. We present the details of choosing a tracking and validation mechanism for a query in Section 5.2. In the following, we assume that the transaction has chosen a tracking method and then starts to record tracking information. Similar to AOCC on transaction level, there are three cases as follows:

Algorithm 2: Validate Execution

```
1  Function validate(txn)
2      txn.writeSet.sort();
3      for w in txn.writeSet do
4          w.tuple.lock(txn);
5      index = gList.put(txn);
6      txn.cts = index;
7      if ABORT == valLR1(txn.readSet) then
8          txn.status = ABORT;
9          return ABORT;
10     for p in txn.predicateSet do
11         if p.type == $\mathcal{P}_{no\_readset}$ then
12             if ABORT == valGW(p, txn.sts, txn.cts-1) then
13                 txn.status = ABORT;
14                 return ABORT;
15         else if $\mathcal{C}_{lrv2}(p) < \mathcal{C}_{gwv}(p)$ then
               /* Choose LRV2 for $\mathcal{P}_{readset}$ */;
16             if ABORT == valLR2(p) then
17                 txn.status = ABORT;
18                 return ABORT;
19         else
               /* Choose GWV for $\mathcal{P}_{readset}$ */;
20             if ABORT == valGW(p, txn.sts, txn.cts-1) then
21                 txn.status = ABORT;
22                 return ABORT;
23     return SUCCESS;
```

Case 1 ($\mathcal{R}$): This case indicates that the query is on primary key columns. For each accessed tuple, we get its pointer *tuple*, generate a read version $\langle tuple, tuple.ts\rangle$ and add it to the transaction's ReadSet. To avoid phantoms, $\mathcal{R}$ needs to record the version of B^+-tree leaf nodes overlapping with the key range if the query is a range scan.

Case 2 ($\mathcal{P}_{no\_readset}$): The transaction generates a predicate with type $\mathcal{P}_{no\_readset}$ for the query. Since it is not required to store the version of each accessed tuple, the member variable *set* of the predicate is set to *null*.

Case 3 ($\mathcal{P}_{readset}$): Besides generating a predicate with type $\mathcal{P}_{readset}$, the transaction needs to record the version of each accessed tuple. Specifically, it adds the $\langle tuple, tuple.ts\rangle$ of each accessed *tuple* to the predicate's ReadSet.

Note that if the tracking type is $\mathcal{P}_{readset}$ or $\mathcal{P}_{no\_readset}$ and the transaction's PredicateSet is empty, the transaction needs to get the CommittedIndex from gList as its start timestamp (*sts*). After the predicate is generated successfully, it is added to the PredicateSet of the transaction.

5.1.2 Validation phase

After a transaction executes all of its operations, it enters the validation phase, where the transaction needs to determine its serialization point and to check whether it conflicts with other concurrent transactions before its serialization point. The pseudocode of the validation execution is illustrated in Algorithm 2.

The first step is to lock all tuples in the transaction's WriteSet (lines 3 and 4). To avoid deadlocks, these tuples are locked according to the order of their primary keys. This approach is also used in Silo [22] and TicToc [27].

After all write locks are acquired, the transaction puts itself to the NextIndex of gList and then gets its position in the list. Note that the index of this position in gList is unique and monotonically increasing. Therefore, we use it as the transaction's commit timestamp, which also serves as its serialization point in all transactions.

AOCC first validates the queries adopting the tracking mechanism $\mathcal{R}$. Recall that all $\mathcal{R}$ queries of a transaction use the same ReadSet to record tracking information and $\mathcal{R}$ utilizes the method LRV1. Therefore, AOCC calls the function valLR1 to validate the ReadSet (lines 7–9). If the function returns the result *ABORT*, the transaction sets its status to *ABORT* and returns.

If the validation for the ReadSet is passed successfully, then we validate the transaction's PredicateSet. Recall that we use one PredicateSet to record the tracking information of queries adopting $\mathcal{P}_{readset}$ or $\mathcal{P}_{no\_readset}$ for a transaction. Therefore, for each entry p in the PredicateSet, AOCC first checks the tracking type of the predicate p. If $p.type = \mathcal{P}_{no\_readset}$, we use GWV method to validate the predicate (lines 11–14). Specifically, the transaction calls the function valGW with the parameters p, $t.sts$ and $t.cts - 1$, where $[t.sts, t.cts - 1]$ denotes the index range of concurrent transactions in gList. If the function valGW returns *ABORT*, the status of the transaction is set to *ABORT*.

If $p.type = \mathcal{P}_{readset}$, we first pick a validation method from LRV2 and GWV for the predicate. In order to reduce the validation cost, we compare both methods and choose one that costs less. In Section 5.2, we introduce the analysis of validation cost and how to choose a validation strategy. If LRV2 is chosen, we call the function valLR2 to validate this predicate; otherwise, valGW is called (lines 15–22). If the function returns *ABORT*, the transaction sets its status to *ABORT* and then aborts itself. For the method GWV, it should be noted that the status of a transaction in gList may not be *ABORT* or *COMMITTED*. This case indicates that the transaction in gList is in the validation phase. In this case, AOCC can wait until its status is refreshed. Alternatively, we can validate the predicate p along with the WriteSet of the unfinished transaction in gList directly.

Lastly, if all validations are passed successfully, the transaction returns *SUCCESS* result.

5.1.3 Write phase

If the validation of a transaction is passed successfully, the transaction will enter the write phase, where it updates and unlocks each tuple in its WriteSet. It should be noted that the lock of a tuple is not released until the tuple's data are refreshed and timestamp field is set to the transaction's commit timestamp. Finally, the transaction sets its status to *COMMITTED*.

5.2 Adaptive Strategy

There are two unsolved issues in AOCC on query level:

Issue 1: How to choose a tracking mechanism type for a read query in the read phase.

Issue 2: How to choose a validation method for the tracking mechanism $\mathcal{P}_{readset}$ in the validation phase.

In the following sections, we first present a simplified and formal cost analysis for each validation scheme.

5.2.1 Cost analysis

For the validation scheme LRV1, we use $\mathcal{S}_1$ to denote the number of tuples stored in the ReadSet for a query handled by $\mathcal{R}$, and let c_{r1} represent the cost of checking version change of a tuple in the ReadSet by LRV1. We use the cost of maintaining and checking the ReadSet of a read query as its validation cost. Therefore, if $\mathcal{R}$ serves a read query q, the cost of LRV1 for the query is estimated as follows:

$$\mathcal{C}_{lrv1}(q) = \mathcal{S}_1 \cdot c_{r1} \tag{1}$$

Similarly, we use $\mathcal{S}_2$ and c_{r2} to represent the number of tuples stored by $\mathcal{P}_{readset}$ and the cost of verifying one tuple for LRV2, respectively. Usually, c_{r2} is greater than c_{r1}. This is because LRV2 is required to re-execute the predicate to generate a new ReadSet in addition to checking each tuple from the predicate's ReadSet. When $\mathcal{P}_{readset}$ uses LRV2 to detect the violation of serialization, the estimated cost of LRV2 for a query q tracked by $\mathcal{P}_{readset}$ is as follows:

$$\mathcal{C}_{lrv2}(q) = \mathcal{S}_2 \cdot c_{r2} \tag{2}$$

The tracking mechanisms $\mathcal{P}_{readset}$ and $\mathcal{P}_{no\_readset}$ can utilize the validation method GWV to detect a conflict between a transaction and its concurrent transactions. For each query of a transaction, the GWV cost is related to two aspects: the number of its overlapping transactions and the average size of WriteSet of each overlapping transaction, which are denoted by $\mathcal{N}$ and $\mathcal{W}$, respectively. Let c_p represent the cost of checking whether an entry from the WriteSets of other transactions satisfies the predicate. Therefore, the cost of GWV for a query q is estimated as follows:

$$\mathcal{C}_{gwv}(q) = \mathcal{N} \cdot \mathcal{W} \cdot c_p \tag{3}$$

To calculate the costs $\mathcal{C}_{lrv1}$, $\mathcal{C}_{lrv2}$ and $\mathcal{C}_{gwv}$ for a query, we need to know the values of those parameters on the right hand of above equations. The factors of c_{r1}, c_{r2} and c_p can be regarded as configurable parameters. We demonstrate how to avoid configuring these parameters in Section 5.2.3. The values of parameters $\mathcal{N}$ and $\mathcal{W}$ can be estimated according to the statistics on the information of committed transactions for a recent period of time. In our implementation and experiments, AOCC does not calculate $\mathcal{N}$ and $\mathcal{W}$ for each computation of $\mathcal{C}_{gwv}$. Instead, AOCC takes $\mathcal{N}$ and $\mathcal{W}$ as global parameters which are updated in a preconfigured time interval. For example, we can compute and refresh $\mathcal{N}$ and $\mathcal{W}$ according to the transactions registered in gList every one second. As a result, $\mathcal{C}_{gwv}$ can be estimated for each query without introducing extra overheads. However, we still lack the values of $\mathcal{S}_1$ and $\mathcal{S}_2$ for estimating $\mathcal{C}_{lrv1}$ and $\mathcal{C}_{lrv2}$, respectively.

Recall from Issue 1 that we have to choose the tracking mechanism type for a read query in the transaction's read phase. For the parameter $\mathcal{S}_1$, we know that it only serves a query on primary key columns. Since $\mathcal{R}$ needs to record B^+-tree leaf nodes covering the query range, the value of $\mathcal{S}_1$ for the query is equal to the sum of the size of result set and the number of the corresponding leaf nodes.

For the parameter $\mathcal{S}_2$, it is used to calculate the LRV2 cost $\mathcal{C}_{lrv2}$. Recall from Section 4 that $\mathcal{P}_{readset}$—which generates a predicate to track a query on primary/non-primary columns—only needs to record the present tuples into the predicate's ReadSet. Thus, $\mathcal{S}_2$ is equal to the number of present tuples matching the predicate in the database. Since we can first get the present tuples before the tracking mechanism is chosen for the query in the read phase, $\mathcal{C}_{lrv1}$ and $\mathcal{C}_{lrv2}$ can be accurately estimated, which helps the decision of which tracking mechanism is selected.

5.2.2 The choice of validation scheme

Before a read query starts to execute, the straightforward method is to assign it a validation scheme with minimal

Algorithm 3: Adaptive Strategy

```
1  Function getTrackingType(query)
2      if isPredictable(query) then
3          if $\mathcal{C}_{lrv1}(query) < \min(\mathcal{C}_{lrv2}(query), \mathcal{C}_{gwv}(query))$ then
4              return $\mathcal{R}$;
5          else if $\mathcal{C}_{lrv2}(query) < \min(\mathcal{C}_{lrv1}(query), \mathcal{C}_{gwv}(query))$ then
6              return $\mathcal{P}_{readset}$;
7          else
8              return $\mathcal{P}_{no\_readset}$;
9      else
10         return $\mathcal{P}_{readset}$;
```

cost. Although the cost of each validation scheme can be estimated in the read phase, it would be better to defer the cost estimation until the validation phase in some cases. For example, a read query of a long-running transaction is executing under a workload with sudden variation, where the cost $\mathcal{C}_{gwv}$ is also changing dramatically. This leads to a result that the tracking mechanism chosen in the read phase may not have the minimal cost. To this end, we classify the read queries into two categories: predictable queries and non-predictable queries. For instance, a read query in auto-commit mode can be regarded as a predictable one, since it enters the validation phase once the query is done.

In Algorithm 3, we illustrate how to choose the tracking mechanism for a read query.

- For a predictable query q, if $\mathcal{C}_{lrv1}(q)$ has the minimal cost, we choose the tracking mechanism $\mathcal{R}$; if $\mathcal{C}_{lrv2}(q)$ costs the least, $\mathcal{P}_{readset}$ is chosen; otherwise, $\mathcal{P}_{no\_readset}$ is chosen (lines 3–8). Since $\mathcal{R}$ can not serve the query using non-primary keys, $\mathcal{C}_{lrv1}(q)$ is set to the *max* value (i.e., always greater than $\mathcal{C}_{lrv2}$ and $\mathcal{C}_{gwv}$) if the query q is on a non-primary key column.
- For a non-predictable query, it is not necessary to compare the validation costs (line 10), and we use $\mathcal{P}_{readset}$ to track. This is because $\mathcal{P}_{readset}$ can be validated by both methods LRV2 and GWV, and AOCC chooses one of them with minimal cost in the validation phase.

$\mathcal{R}$ adopts the validation method LRV1, and $\mathcal{P}_{no\_readset}$ uses GWV. Recall from Issue 2 that in the validation phase, we need to choose a validation method for each $\mathcal{P}_{readset}$ predicate. In most cases, a predicate represents a query. Therefore, for the same validation method, the cost of validating a query is equal to that of validating its corresponding predicate. According to Equation 2 and Equation 3 shown above, we calculate both costs $\mathcal{C}_{lrv2}$ and $\mathcal{C}_{gwv}$ for the $\mathcal{P}_{readset}$ predicate, and choose the validation method with minimal cost. As these costs are calculated in the validation phase, AOCC can give reasonably accurate estimates for selecting the best validation method.

5.2.3 Reducing adaption overhead

It is obvious that computing these costs $\mathcal{C}_{lrv1}$, $\mathcal{C}_{lrv2}$ and $\mathcal{C}_{gwv}$ for each query can consume enormous quantities of system resources, which may decrease the system performance. Therefore, we need an efficient mechanism, which does not estimate the costs for a query every time. c_{r1}, c_{r2} and c_p can be regarded as constants. Let $c_{r2} = a \cdot c_{r1}$. Thus, we can replace c_{r2} with c_{r1} to calculate the cost of LRV2 for a query, i.e., $\mathcal{C}_{lrv2} = \mathcal{S}_2 \cdot a \cdot c_{r1}$. Next, we introduce a new constant c, and set it to (c_p/c_{r1}). Then we define a threshold $\mathcal{T}$:

$$\mathcal{T} = \mathcal{N} \cdot \mathcal{W} \cdot c \tag{4}$$

The threshold $\mathcal{T}$ is a global variable, which is refreshed by transactions periodically. Specifically, when a transaction accesses $\mathcal{T}$ and finds that $\mathcal{T}$ is not updated in a fixed period of time, the transaction gets $\mathcal{N}$ and $\mathcal{W}$ according to the transactions registered in gList within the last period of time and updates $\mathcal{T}$ according to Equation 4. The transaction directly utilizes $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{T}$ to pick an appropriate tracking mechanism for a predictable query instead of computing and comparing different validation costs:

Case 1 ($\mathcal{S}_1 < \min(\mathcal{S}_2 \cdot a, \mathcal{T})$): $\mathcal{R}$ is chosen.
Case 2 ($\mathcal{S}_2 \cdot a < \min(\mathcal{S}_1, \mathcal{T})$): $\mathcal{P}_{readset}$ is chosen.
Case 3 ($\mathcal{T} \leq \min(\mathcal{S}_1, \mathcal{S}_2 \cdot a)$): $\mathcal{P}_{no\_readset}$ is chosen.

As a consequence, this optimization basically takes a few comparisons for the choice of a validation method for each query, and thus reduces the overhead of computation on validation costs.

5.3 Correctness Analysis

Now we analyze the correctness of AOCC, i.e., the serializability of transactions. Recall that a transaction gets a unique index from gList as its serialization point. In other words, the unique index, a monotonically increasing number, is used as the commit timestamp of the transaction. Owing to the centralized gList, any two transactions must have different commit timestamps.

THEOREM 1. *Any schedule in AOCC is equivalent to the serial order according to the transactions' commit timestamps.*

PROOF. Essentially, a transaction is serializable if all its reads and writes occur at its serialization point.

Since a transaction first locks all items in its WriteSet and then gets a monotonically increasing commit timestamp in the validation phase, our protocol ensures that all writes of the transaction can occur at the commit timestamp point.

Next, for a committed transaction, we show our protocol can also ensure that the execution of all its read queries is equivalent to the occurrence of these read queries at its commit timestamp point. In other words, a read operation from a committed transaction returns the values of the latest writes at the point in time of generating the transaction's commit timestamp. Specifically, a transaction T_a writes a tuple A and commits first with commit timestamp t_1. If a following committed transaction T_c with t_3 reads A written by T_a, there is no transaction T_b writing A and committed at the point t_2, such that $t_1 < t_2 < t_3$.

This can be proved by contradiction. Assuming that there exists the committed transaction T_b with t_2 such that $t_1 < t_2 < t_3$ in the above paragraph.

If the tuple A is tracked by $\mathcal{R}$, then the tuple A with its version t_1 must be added to the ReadSet of T_3. When T_3 enters the validation phase, T_2 is committed or being validated. In other words, the tuple A has been updated to the new version t_2 or it is being locked. Therefore, T_c must be aborted by the function valLR1 introduced in Section 4.2. This is a contradiction to the assumption that T_c is a committed transaction.

If the tuple A is tracked by $\mathcal{P}_{no\_readset}$, then the corresponding predicate p must be stored by T_c. Since $t_1 < t_2$,

T_b must not be committed before T_c accesses A, i.e., $T_c.sts \leq t_2$. In addition to $t_2 < t_3$, T_b must be in the range $[T_c.sts, t_3)$ in gList. Since A is in the WriteSet of T_b, it is covered by T_c's predicate p. Therefore, T_c must be aborted by the function GWV, which is a contradiction to the assumption that T_c is a committed transaction.

If the tuple A is tracked by $\mathcal{P}_{readset}$, the corresponding predicate p with ReadSet must be stored. In the validation phase, if GWV is chosen for checking A, we have proven this in the above case. If LRV2 is chosen, we re-execute the predicate p to get A. Since the new ReadSet containing A written by T_b is not equal to the original ReadSet stored by T_c, T_c must be aborted by the function valLR2, which is a contradiction to the assumption that T_c is committed.

Our tracking mechanisms and validation methods can ensure that the read queries of a transaction behave as if they were executed at the point in time of getting the transaction's commit timestamp. Hence, the theorem follows. □

5.4 Discussion

Secondary indexes: All of our discussion above assumes that there is always a secondary index on the non-primary key column. Therefore, we avoid traversing the whole table when a query is on a non-primary key. In real systems, it is common that we may not create a secondary index for a column. In this case, the cost of LRV2 may be not simply equal to the size of ReadSet in the corresponding predicate. The reason is that the transaction is required to scan the whole table when validating the $\mathcal{P}_{readset}$ predicate. Its validation cost grows with the increase of the table's size. Therefore, if a read query is on the non-primary key column without secondary index, we directly use the tracking mechanism $\mathcal{P}_{no\_readset}$ for this query.

Overhead: AOCC can choose the appropriate tracking mechanism for a query according to costs of different validation methods. However, to achieve the adaptive choice, AOCC incurs additional expenses over the system. Compared with the traditional LRV method, AOCC has to maintain an additional data structure: the global list (i.e., gList) to store the WriteSets of transactions that are finished or being validated. Recall from Section 4.1 that we adopt a lock-free circular array to implement gList which is cache-friendly by allocating a contiguous area of memory. In the aspect of memory footprint of gList, assuming that the size of each transaction's WriteSet is about 1KB and the capacity of gList is 1,000. Then AOCC consumes about at most 1MB more than the traditional LRV method in memory space. Thus, the extra memory consumption of AOCC is very small. Another concern is that the centralized gList may be the bottleneck of the system. To alleviate this problem, gList is operated in a lock-free manner and the read-only transactions do not have any burden on gList. The results of scalability experiment in Section 6.2 show that AOCC scales up nicely as the worker threads increases.

6. EVALUATION

In this section, we present our evaluation of the AOCC scheme, which is implemented in the DBx1000 codebase developed by Yu et al. [26, 27]. DBx1000 is an in-memory DBMS prototype [1] that stores all dataset in main memory in a row-oriented manner. It provides a pluggable framework to integrate different concurrency control protocols (e.g., Silo, Hekaton, 2PL, etc.) for performance comparison. Therefore, we compare the performance of AOCC with other OCC schemes in DBx1000:

- AOCC. This is our AOCC scheme on query level, which adopts adaptive strategy to choose an appropriate tracking mechanism and validation method for a read query.
- AOCC-TXN. This is AOCC on transaction level, which chooses a tracking and validation method for each transaction. In this scheme, we assume that for each transaction, its execution logic is known before it starts.
- LRV-OCC. This OCC scheme adopts the tracking mechanisms $\mathcal{R}$ and $\mathcal{P}_{readset}$. Since gList is not required to be implemented in this scheme, we use LRV1 and LRV2 to validate queries assigned $\mathcal{R}$ and queries assigned $\mathcal{P}_{readset}$, respectively. Since these validation methods LRV1 and LRV2 are used in Hekaton and Silo, we use this scheme to represent these OCC systems.
- GWV-OCC. This OCC scheme adopts the tracking mechanism $\mathcal{P}_{no\_readset}$ and the validation method GWV. Since it checks the WriteSets from all other concurrent transactions for validating a transaction, we need to maintain gList, which is similar to the *recentlyCommitted* list in HyPer. Therefore, we use the performance of this scheme to denote the results of HyPer.

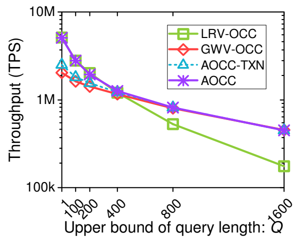
All experiments are executed on a 2-socket single machine with 160GB main memory and two Intel Xeon Silver 4100 processors (each with 8 physical cores) while running CentOS 7.4 64bit. With hyper-threading, there are a total of 32 logical cores at the operating system level. This hardware configuration is very common in modern data centers.

6.1 Workloads

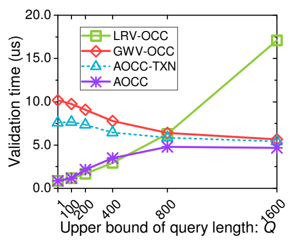
YCSB. The Yahoo! Cloud Serving Benchmark (YCSB) is representative of large-scale online services [4]. The workload E of YCSB benchmark contains read, write and scan queries. The degree of workload skew is tuned by changing the parameter (θ) of Zipfian distribution. The key of each read/write query is selected according to the predefined Zipfian distribution. For each scan operation, the start key of its range is also selected by the same Zipfian distribution.

By default, we run YCSB workloads in the low-contention scenario ($\theta = 0.6$). Each transaction contains five queries, each one follows the same access distribution. The YCSB dataset is a single table, which is initialized to 10 million records. Each tuple has a single primary key column and 10 additional columns.

Hybrid TPC-C. TPC-C is the most widely used OLTP benchmark, which is a mixture of read-only and update-intensive transactions. According to the requirement of practical applications, we create a new transaction, named Reward. To achieve an incentive, a Reward transaction first finds the top shopper who spent the most in the past period of time in a randomly specified district. Then, the top shopper is rewarded with a bonus to his/her balance. Thus, in our hybrid TPC-C workload, the Reward transaction contains four queries, where one is a scan operation randomly accessing a range of the customer table, and the others are three update operations in the customer, district and warehouse tables, respectively.



(a) Transaction throughput.



(b) Validation time of each transaction.

Figure 5: YCSB (Query length) – The performance of different OCC schemes under the YCSB workloads with different upper bounds of query length.

By default, the number of TPC-C warehouses is set to 4. We adopt the access model of the original TPC-C, i.e., the probabilities of accessing local warehouse and remote warehouses are 85% and 15%, respectively. Note that all Reward transactions only scan the local warehouse. Therefore, each Reward transaction may conflict with the cross-warehouse Payment transactions. Since Payment and NewOrder make up 88% of the original TPC-C, our hybrid workload is a mixture of 45% Payment transactions, 45% NewOrder transactions, 10% Reward transactions.

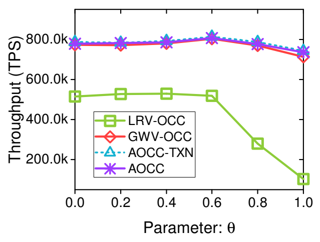
6.2 YCSB Results

A heterogeneous workload usually mixes read, write and scan operations in a transaction. Among these operations, a scan often takes much time on execution and validation. Query length is used to indicate how many records might be accessed by the scan query. By default, the percentages of reads, scans and writes are 80%, 10% and 10%, respectively. Next, we first analyze the performance of all OCC schemes under the YCSB workloads with various query lengths.

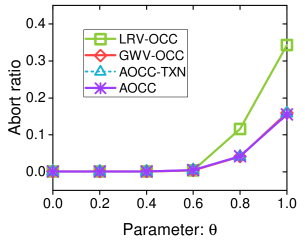
Query length. To investigate the effect of query length on different OCC schemes, we introduce a new variable: the upper bound of query length Q. For each scan operation, its query length is randomly set to a number from $[1, Q]$. We vary the variable Q to measure the performance of each OCC scheme under a low-contention workload, i.e., the parameter θ of Zipfian distribution is set to 0.6. In this experiment, we fix the number of worker threads to 32.

Figure 5 shows the performance of different OCC schemes in this experiment. When the query length is short (e.g., $Q < 400$), LRV-OCC outperforms GWV-OCC. In this scenario, it can be seen that LRV-OCC provides higher throughput and GWV-OCC spends more time in its validation phase of a transaction. This is because GWV-OCC needs to check the WriteSets of all concurrent transactions in gList for a read query, even if the query length is 1. On the other hand, when the query length is long (e.g., $Q > 400$), GWV-OCC has better performance than LRV-OCC. Recall from Section 5.2 that the cost of LRV is linearly proportional to the number of accessed tuples. Therefore, the validation time of a transaction in LRV-OCC grows linearly as the parameter Q increases, which is illustrated in Figure 5(b).

AOCC-TXN chooses GWV for a transaction containing scan queries, even if the result size of these queries is very small. Thus, AOCC-TXN does not perform well when Q is short. As Q increases, GWV starts showing its advantages. Therefore, AOCC-TXN can achieve good performance for workloads with large upper bound of query length Q.



(a) Transaction throughput.



(b) Ratio of aborted transactions.

Figure 6: YCSB (Contention) – The performance of different OCC schemes under the YCSB workloads with various contention rates ($Q = 800$).

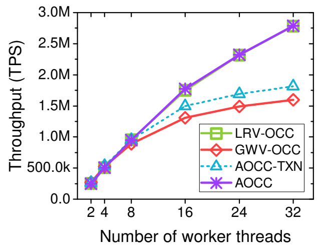
Owing to the adaptive strategy on query level, AOCC always has the best performance regardless of the query length in this experiment. In other words, AOCC always chooses the validation method with minimal cost for a read query. As Q increases, the cost of the LRV method for a scan query becomes higher. Therefore, a scan with large query length is likely assigned GWV by AOCC.

Contention. Next, we use YCSB to compare AOCC and other OCC schemes under different contentions by varying the parameter θ of Zipfian distribution from 0 to 1. Note that the access distribution is uniform when $\theta = 0$ and the contention is extremely high when $\theta = 1$. The number of worker threads is fixed to 32. We set the upper bound of query length Q to 800 in this experiment, and similar results were observed in other Q values.

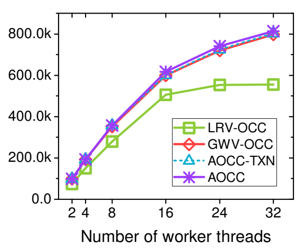
Figure 6 illustrates the throughput and abort ratio of each OCC scheme. When θ is small (≤ 0.6), the contention of the workload is at a low or medium level. In this case, the abort ratio of each OCC scheme is close to zero (< 0.005), and thus the throughput of each method keeps stable. This reflects that when $Q < 800$ and $\theta < 0.6$, the abort ratio of each method would also be kept at a very low level. Therefore, AOCC would have adaptation behavior similar to that in Figure 5 when the workload is less contended.

Once the parameter θ exceeds 0.6, the abort ratio of each OCC scheme increases, especially for LRV-OCC. Since LRV-OCC takes more time on executing and validating a transaction when the query length is large, its throughput significantly drops due to massive aborts: as many as 35% transactions are aborted at the highest conflict level ($\theta = 1$). AOCC and AOCC-TXN choose the GWV method for a long scan query, and they provide performance similar to GWV-OCC in this experiment. As a consequence, LRV-OCC is more sensitive to the high-contention workload containing scan queries with wide range. At runtime, AOCC can choose a validation method to reduce the time of validating a transaction. Therefore, it's able to achieve the best performance under different values of parameter θ.

Scalability. To investigate the scalability of different OCC schemes, we measure the throughput by varying the number of worker threads. The experimental results are presented in Figure 7. GWV-OCC does not scale well when the query length is short ($Q = 100$). As the number of workers increases, more transactions are running concurrently. It leads to increased validation cost of GWV-OCC as a transaction has to check more concurrent transactions in gList. When the query length is long ($Q = 800$), LRV-OCC does not scale well because of its high cost on validating a long scan query. AOCC-TXN is unable to scale well when trans-

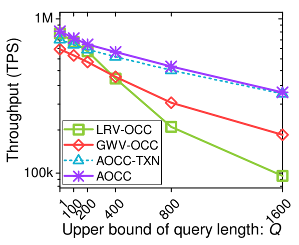


(a) $Q = 100$.

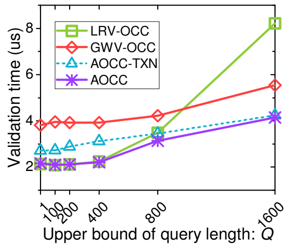


(b) $Q = 800$.

Figure 7: YCSB (Scalability) – The throughput of different OCC schemes with different number of worker threads under the YCSB workloads.



(a) Transaction throughput.



(b) Validation time of each transaction.

Figure 8: TPC-C – The performance of different OCC schemes under the hybrid TPC-C workloads with different upper bounds of query length Q.

actions in a workload have short scan queries ($Q = 100$). In this case, fine-grained adaptation mechanism is required to choose a validation method for each read query. Hence, AOCC has better scalability than other OCC methods.

6.3 TPC-C results

We now compare AOCC with other OCC schemes under hybrid TPC-C workloads. In this experiment, the number of worker threads is fixed to 32. Since we add a new Reward transaction that contains a scan query, we measure the performance of each OCC scheme by varying the upper bound of query length of the Reward's scan operation.

Figure 8 shows the performance of each OCC scheme in this experiment. Note that the results in Figure 8(b) are calculated by all finished transactions including aborted ones.

Like the results in the experiments under the YCSB workloads, LRV-OCC performs better than GWV-OCC when the query length of a scan operation is short (e.g., $Q < 400$), and GWV-OCC provides better performance when the query length is long (e.g., $Q > 400$). Again, this confirms that each validation method has its own advantages. As illustrated in Figure 8(b), the validation time of a transaction in LRV-OCC increases as Q grows, which explains that the long scan query has a great impact on LRV-OCC.

AOCC still performs the best among the four OCC schemes we tested in this experiment, due to its adaptive strategy. For each transaction in AOCC, its validation time is the least. Since transactions are more complex than those in the YCSB workload, the scan query can not dominate the execution time even the query length is long. It can be seen that AOCC achieves 1.9× better throughput than GWV-OCC even though the upper bound is set to 1600.

Since Payment and NewOrder do not contain any scan query, AOCC-TXN chooses LRV for these two types of transactions. On the other hand, it would assign GWV to a

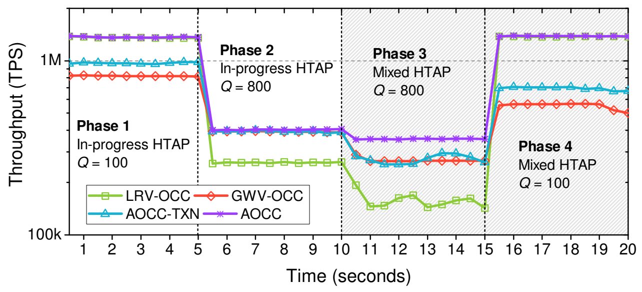


Figure 9: Performance of different OCC schemes in a dynamically changing workload mixture.

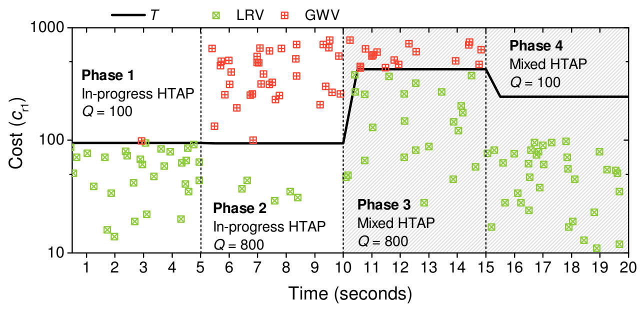


Figure 10: Selection of validation schemes in a dynamically changing workload mixture. ($\mathcal{T}$ is refreshed every 50 ms.)

Reward transaction. Since the GWV method is more suitable for a scan with large query length, AOCC-TXN performs closer to AOCC with the increase of Q.

6.4 Workload-Aware Adaption

In the practical applications, the running workloads are often dynamically changed and mixed. For instances such as real-time fraud detection, customer marketing and risk management, these In-progress HTAP applications require database systems to scan both latest committed records and historical trading data for aggregation. On the other hand, database systems often need to ingest data from other sources for immediate business decision-making. Thus, we now examine the ability of AOCC to adapt workload at runtime and compare it against other schemes.

In this experiment, we run a sequence of workloads with changing properties. To mimic the dynamic and time-varying character of the heterogeneous workloads, and to clearly demonstrate the impact of the validation schemes on the different workload cases, the workload sequence is divided into 4 consecutive phases. All phases use 16 specified worker threads to run the YCSB transactions with 80% reads, 10% scans and 10% writes. We set $Q = 100$ in the first and last phases and $Q = 800$ in the middle two phases. Phases 1 and 2 are to simulate In-progress HTAP workloads. The workloads in the last two phases are mixed with a data ingestion task, referred to as mixed HTAP. We use additional 16 worker threads to run the data ingestion transactions, where each one contains 10 write queries. Each phase takes 5 seconds to run a particular workload type, and then switches to another workload type in the next phase. We measure the throughput of workload in each phase for LRV-OCC, GWV-OCC, AOCC-TXN, and AOCC. It should be noted that the throughput is measured on the 16 YCSB worker threads in order to observe how the data ingestion tasks impact the performance of In-progress HTAP.

Figure 9 shows the performance of different OCC schemes in this experiment. In Phase 1, owing to the short scan queries, LRV-OCC performs better than GWV-OCC. Once the scan length becomes large in Phase 2, GWV-OCC has better performance. These results conform to our previous analysis. *In Phase 3,* AOCC *has minimal performance reduction incurred by data ingestion tasks* (AOCC $\downarrow$ 12%, AOCC-TXN $\downarrow$ 29%, GWV-OCC $\downarrow$ 32% and LRV-OCC $\downarrow$ 44%). Once the query length of an In-progress HTAP transaction becomes short in Phase 4, the data ingestion tasks have a very small impact on LRV-OCC. Since data ingestion can increase the validation cost of GWV, GWV-OCC performs worse in Phase 3/4 than Phase 2/1.

As workloads or transaction operations change, AOCC will react and adaptively choose validation schemes, and thus it achieves the best performance. Figure 10 presents the computation of validation costs for selecting the appropriate validation schemes. C_{lrv1}, C_{lrv2}, and C_{gwv} are normalized by c_{r1}. We show the change of the threshold $\mathcal{T}$ (i.e., normalized C_{gwv} by Equation 4) and the estimated LRV costs of randomly selected 40 scan queries in each phase. It should be noted that y-coordinates represent the $\max(C_{lrv1}, C_{lrv2})$ when a square is under the line of $\mathcal{T}$, otherwise y-coordinates denote the $\min(C_{lrv1}, C_{lrv2})$. Therefore, the green squares represent queries using LRV as validation method, and red ones represent queries using GWV. We observe that $\mathcal{T}$ is stable in Phases 1 and 2, and AOCC switches validation schemes for most of queries since their result sets have become large (i.e., Q is changed from 100 to 800). Once the workload is mixed with data ingestion tasks in Phase 3, the threshold $\mathcal{T}$ becomes large. As workload features change, AOCC re-selects the LRV method for some queries.

6.5 Summary

Our experiments confirm that the LRV method is not suitable for validating a large-range scan query and GWV does not perform well when the workload is write-intensive. AOCC-TXN chooses GWV for a transaction containing scan queries without considering the size of result set returned by the scan query and the workload characteristics. Thus, AOCC-TXN does not have good performance under the workload where the query length is small. However, experimental results show that AOCC-TXN can work well once the scan queries dominate the performance (e.g., $Q > 800$). Owing to the adaptive strategy on query level, AOCC always has the best performance under various workloads.

7. RELATED WORK

Optimistic Concurrency Control. Optimistic concurrency control (OCC) was first proposed by Kung and Robinson in 1981 [9]. In a multi-core setting, it suffers from scalability bottlenecks due to the serial validation. Silo [22] adopted periodically-updated epochs with the commit protocol to avoid bottlenecks of centralized serialization protection. TicToc [27] proposed a data-driven timestamp allocation method to further enhance the scalability of OCC. Yuan et al. [28] presented the balanced concurrency control (BCC) method to reduce false aborts by detecting data dependency patterns with low overhead. Since transaction healing [24] avoids blindly aborting a transaction once it fails in validation, it scales the conventional OCC towards dozens of cores under high-contention workloads.

Multiversion concurrency control (MVCC) improves the performance of analytical processing systems since read-only transactions are never blocked or aborted. Therefore, the multiversion scheme is widely used in the HTAP-friendly database systems, such as Hekaton [10], HyPer [7, 14], SAP HANA [18], etc. To ensure correctness, Hekaton re-executes each range query to detect serializability violation and to prevent phantoms for optimistic transactions. HyPer uses a variation of precision locking technique [6] to test discrete writes of recently committed transactions against predicate-oriented reads of the committing transaction.

Mixed Concurrency Control. A recent work claims that there is no single concurrency control that is ideal for all workloads [21]. Hence, it proposed an adaptive concurrency control (ACC) protocol for the partition-based in-memory database systems. CormCC [20], the successive work of ACC, is a general mixed concurrency control framework without coordination overhead across candidate protocols while supporting the ability to change a protocol online with minimal overhead. Callas [25] and its successive work Tebaldi [19] provided a modular concurrency control mechanism to group stored procedures and provided an optimized concurrency control protocol for each group based on offline workload analysis. Cao et al. [3] proposed a hybrid tracking method for choosing one from OCC and pessimistic CC to handle different conflict accesses. 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. Lin et al. [12] proposed an adaptive and speculative optimistic concurrency control (ASOCC) protocol for effective transaction processing under dynamic workloads.

All the mixed concurrency control protocols are mainly concentrated on exploiting the benefits of pessimistic method under high-contention workloads and optimistic mechanism under low-contention workloads. However, even under heterogeneous workloads with low contentions, the validation scheme used by an OCC protocol has a significant impact on the performance. AOCC aims to choose a low-cost validation scheme at runtime.

8. CONCLUSION

In this work, we introduce a simple yet effective OCC framework to integrate two widely used validation methods. According to the number of records read by a query and the size of write sets from concurrent update transactions, AOCC adaptively chooses an appropriate tracking mechanism and validation method to reduce the validation cost for this query. Our evaluation results demonstrate that AOCC can achieve good performance for a broad spectrum of heterogeneous workloads without sacrificing serializability guarantee.

Acknowledgments

This research is supported in part by National Key R&D Program of China (2018YFB1003303), National Science Foundation of China under grant numbers 61732014 and 61432006, and Guangxi Key Laboratory of Trusted Software (kx201602). Peng Cai is the corresponding author. The authors would like to thank the anonymous reviewers for their valuable feedback.

9. REFERENCES

[1] DBx1000. https://github.com/yxymit/DBx1000.

[2] Market Guide for HTAP-Enabling In-Memory Computing Technologies. https://www.gartner.com/doc/3599217/market-guide-htapenabling-inmemory-computing, 2017.

[3] M. Cao, M. Zhang, A. Sengupta, and M. D. Bond. Drinking from both glasses: combining pessimistic and optimistic tracking of cross-thread dependences. In *PPOPP*, pages 20:1–20:13, 2016.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[5] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Conference*, pages 1243–1254, 2013.

[6] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision locks. In *SIGMOD Conference*, pages 143–147. ACM Press, 1981.

[7] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[8] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *SIGMOD Conference*, pages 1675–1687, 2016.

[9] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[10] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[11] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD Conference*, pages 21–35, 2017.

[12] Q. Lin, G. Chen, and M. Zhang. On the design of adaptive and speculative concurrency control in distributed databases. In *ICDE*, pages 1376–1379, 2018.

[13] B. Momjian. *PostgreSQL: Introduction and Concepts.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[14] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD Conference*, pages 677–689, 2015.

[15] F. Özcan, Y. Tian, and P. Tözün. Hybrid transactional/analytical processing: A survey. In *SIGMOD Conference*, pages 1771–1775. ACM, 2017.

[16] M. Reimer. Solving the phantom problem by predicative optimistic concurrency control. In *VLDB*, pages 81–88. Morgan Kaufmann, 1983.

[17] K. Ren, A. Thomson, and D. J. Abadi. VLL: a lock manager redesign for main memory database systems. *VLDB J.*, 24(5):681–705, 2015.

[18] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD Conference*, pages 731–742, 2012.

[19] C. Su, N. Crooks, C. Ding, L. Alvisi, and C. Xie. Bringing modular concurrency control to the next level. In *SIGMOD Conference*, pages 283–297, 2017.

[20] D. Tang and A. J. Elmore. Toward coordination-free and reconfigurable mixed concurrency control. In *USENIX Annual Technical Conference*, pages 809–822, 2018.

[21] D. Tang, H. Jiang, and A. J. Elmore. Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *CIDR*, 2017.

[22] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[23] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.

[24] Y. Wu, C. Y. Chan, and K. Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *SIGMOD Conference*, pages 1689–1704, 2016.

[25] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via modular concurrency control. In *SOSP*, pages 279–294. ACM, 2015.

[26] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[27] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *SIGMOD Conference*, pages 1629–1642, 2016.

[28] 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. *PVLDB*, 9(6):504–515, 2016.