Accelerating Hybrid Transactional/Analytical Processing Using Consistent Dual-Snapshot

Liang Li[1], Gang Wu[1,3], Guoren Wang[2(✉)], and Ye Yuan[1]

[1] Computer Science and Engineering, Northeastern University, Shenyang, China
liliang@stumail.neu.edu.cn
[2] Computer Science and Technology, Beijing Institute of Technology, Beijing, China
wanggr@bit.edu.cn
[3] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Abstract. To efficiently deal with OLTP and OLAP workload simultaneously, the conventional method is to deploy two separate processing systems, *i.e.*, the transaction-friendly designed OLTP system and the analytic-friendly optimized OLAP system. To maintain the freshness of the data, extra ETL tools are required to propagate data from the OLTP to the OLAP system. However, low-speed ETL processing is the bottleneck in those business decision support systems. As a result, there has been tremendous interest in developing hybrid transactional/analytical processing (HTAP) systems. This paper proposes a wait-free HTAP (WHTAP) architecture, that can perform both OLTP and OLAP requests efficiently in a wait-free form. We develop and evaluate a prototype WHTAP system. Our experiments show that the system can obtain a similar OLTP performance as the TicToc system and a four to six times acceleration in analytical processing at the same time.

Keywords: HTAP · In-memory database systems · Transaction concurrency control

1 Introduction

From a database perspective, there are two types of workloads, *i.e.*, online transaction processing (OLTP) and online analytical processing (OLAP). It is well known that OLTP and OLAP are difficult to process uniformly for a given system. The conventional approach for dealing with such hybrid workloads is to maintain a data warehouse for OLAP that is independent of the OLTP system. The data generated by OLTP systems are periodically transferred into the OLAP systems for analytical processing in a batch-wise fashion (*i.e.*, ETL tools), where ETL tools provide both excellent performance isolation between

G. Li et al. (Eds.): DASFAA 2019, LNCS 11446, pp. 52–69, 2019.
https://doi.org/10.1007/978-3-030-18576-3_4

the two workloads and the ability to tune each system independently. However, the traditional methods are not always timely enough when making business decisions.

In the last few years, organizations have grown increasingly interests in conducting real-time analyses of, and hence making decisions based on, up-to-date sets of raw data. As a result, the freshness of the data used for analysis becomes increasingly important. Recently, there has been tremendous interest in developing hybrid transactional/analytical processing (HTAP) systems. Gartner coined the term "(HTAP)" to describe this new type of database [8]. A similar term used to describe this type of processing is **operational analytics** [1], which indicates that insight and decision-making occur instantaneously with a transaction. HTAP offers 3 advantages: **1. Lower implementation complexity.** The implementation of HTAP no longer requires the consideration of data movement between transactional and analytical databases. **2. Lower analytic latency.** Analytic queries can access the latest transactional data to any extent needed, guaranteeing freshness in decision-making activities. **3. Less data duplication.** Since data movement is avoided, it is possible to reduce data duplication.

Fortunately, the recently emerged database technologies, such as in-memory computing (IMC), have brought new opportunities for the support of hybrid workloads within one database instance. Currently, there are two main types of HTAP implementations: *true-HTAP* and *loose-form HTAP*.

1.1 true-HTAP

To handle the OLTP and OLAP workloads in one single system, multi-version concurrency control (MVCC) is the currently accepted approach [30]. MVCC guarantees a high degree of parallelism, since writes are never blocked by reads. If a tuple is updated, a new physical version of this tuple is created and stored alongside the old version by engaging a version chain, which allows the old version to remain available for the allowed readers.

Limitations. MVCC does not distinguish between transaction types. Both short-running OLTP transactions and long-running read-oriented OLAP queries are treated in the same way on the same database. Obviously, it is inefficient to treat both OLTP and OLAP workloads similarly. First, scan-heavy OLAP queries require a substantial amount of time to deal with long version chains [30]. Second, costly garbage collections are inevitable in the recycling of unattended data.

Therefore, uniform processing approaches, such as MVCC, cannot meet the requirements of HTAP workloads, which consist of transactions of inherently different natures.

1.2 Loose-Form HTAP

As mentioned above, *true-HTAP*, which treats OLTP and OLAP equally, is not a suitable solution for the unsatisfactory performance of this approach. To the best of our knowledge, recently published studies typically classify queries based on the query type and execute the queries in separate replicas within a single system (*loose-form*). Two methods are used to organize the replicas of OLTP and OLAP.

- **Copy on Write.** The system call of fork obtain a virtual snapshot of the OLTP data as an OLAP replica, as in Hyper [13]. However, OLTP and OLAP replica data own the same physical data layouts, which is not effective for optimizing these methods simultaneously. Furthermore, the granularity of the fork is too coarse, which significantly influences performance based on dataset size. In this sense, the fork is not a particularly flexible solution. Anker [37] proposed a fine-grained system-level virtual snapshot system call (similar to the fork), but this method must be applied at the column storage level.
- **Recording and merging delta snapshot.** The recently published AIM [2] and BatchDB [26] both employ the delta snapshot method for data propagation. These methods require additional memory space to record transactions, generate delta snapshots over certain intervals and then merge the snapshots into the OLAP datasets. Note that BatchDB exploits a special in-memory log to record delta updates as a kind of delta snapshot. The most typical industry system is SAP HANA [7,38]. HANA holds a write-optimized delta store to collect the insert and delete operations and then merges the stores into a read-optimized and immutable main store.

1.3 Motivation

Apparently, the delta snapshot scheme is more flexible. However, two challenges are encountered when using delta snapshots. (1) It is difficult to record delta snapshots without significantly affecting the OLTP's latency and throughput. (2) It is difficult to avoid merging snapshots without blocking OLAP queries. For interactive applications, it is better to run both OLTP and OLAP in a wait-free manner. To the best of our knowledge, the state-of-the-art systems both in academia and industry have not yet addressed the wait-free execution of OLTP and OLAP. Is it possible to run both transactions and analysis in a wait-free manner and to perform transactions and analysis simultaneously while ensuring good performance?

This paper presents a wait-free HTAP (WHTAP) architecture. The key feature of the WHTAP architecture is a primary-secondary replication plus a dual snapshot. The primary replica is dedicated for OLTP, whereas the secondary replica is dedicated to OLAP workloads. The OLTP replica and OLAP replica are then connected by a dual-snapshot. Hence, WHTAP can independently optimize OLTP and OLAP according to their workload characteristics while physically isolating their resources. WHTAP not only guarantees both short transactions and long-running analytical queries execute in wait-free manner, but also ensures data freshness besides the consistency and isolation.

In summary, this work results in the following contributions:

- **Dual snapshot-based engine.** We design a dual snapshot structure in the storage engine to ensure the data freshness and wait-free features of WHTAP (Sect. 3).
- **LSM-like query layer.** To ensure that the analytical queries can always access the latest data, we propose an LSM-like query algorithm with a state controller, which ensures that analysis transactions run in a wait-free mode and access the freshest data (Sect. 4).
- **High-performance wait-free HTAP (WHTAP) system.** We implement a prototype and open the source code on Github[1], which can ensure the serialization of OLTP and OLAP at the snapshot isolation level. Compared with the traditional single-engine (*true-HTAP*) method, WHTAP can obtain a similar performance as previously documented for OLTP while delivering an OLAP performance that is approximately 4 to 6 times better relative to values recorded when using single engine. (Sect. 5).

The remainder of the paper is organized as follows: Sect. 2 shows the architecture of the WHTAP system, and Sects. 3 and 4 give the details of the components of WHTAP. Section 5 evaluates several experiments with a modified YCSB benchmark. Related work is discussed in Sect. 6. Section 7 presents our conclusion.

2 Dual Snapshot-Based Architecture

In this paper, we propose a dual delta snapshot-based structure for managing mixed workloads. In this structure, both the transactions and analytical queries are handled in isolation mode. In other words, distinct data replicas are used by OLTP and OLAP.

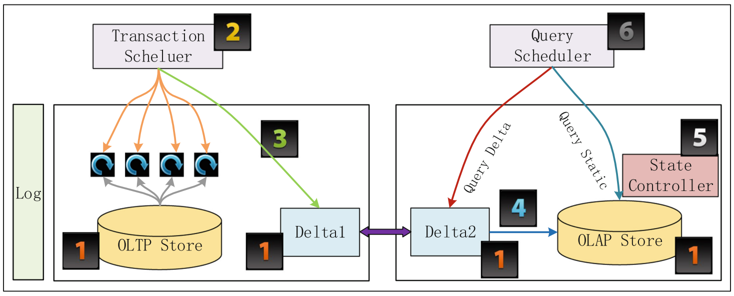


Fig. 1. Architecture of WHTAP

At the highest level, the system runs alternately between two periods. As shown in Fig. 1, we assume that in period one, the transaction data are recorded

[1] https://github.com/bombehub/WHTAP.git.

into delta snapshots $delta_1$ and $delta_2$, which will be merged into the OLAP data store. Then, in the next period, the roles of $delta_1$ and $delta_2$ are exchanged. That is, the delta snapshot remains $delta_2$, while $delta_1$ is merged into the OLAP data store. The data freshness of the WHTAP system can be guaranteed by minimizing the duration of each period.

As is known, taking a delta snapshot will lead to a throughput loss and latency spike, and merging a delta snapshot will break the OLAP's read-only feature. Therefore, the challenge in design is how to record a delta snapshot (Component 3) but not block the OLTP from running, followed by merging the delta snapshot (Component 4) but not blocking the OLAP running at the same time.

To overcome those challenge, we propose an architecture named WHTAP that possesses six total components: the logical components of OLTP are dual-snapshot storage engine and delta snapshot recording, and the OLAP part includes snapshot compaction, LSM-like query layer, and state controller.

- **Storage Engine.** OLTP and OLAP are stored in 2 replicas. Additionally, space to recode the delta data is necessary. This module is primarily responsible for the organization and storage of data.
- **Transaction Concurrency Control.** This control relates to scheduling of the running logic of the OLTP workload and guarantees that the structure is serializable and meets the desired performance.
- **Delta Snapshot.** To ensure the data freshness, the running data of the transaction must be transformed into an OLAP Store. To collect the snapshot, three difficulties must be overcome. First, the linear serializability between the OLTP and delta snapshot must be ensured; second, the litter performance must be disregarded as much as possible; and third, it is important not to cause a significant latency spike.
- **Compact Snapshot.** The dual snapshot structure must periodically merge the frozen snapshot data into the OLAP data. The difficulty that must be overcome here is merging of the snapshot without blocking the normal execution of the OLAP queries.
- **State Controller.** The WHTAP system maintains a state controller to ensure that the query runs correctly and is wait-free, and that the snapshot is merged within a suitable amount of time.
- **Query Execution.** The OLAP data are updated periodically by the delta snapshot data, thus ensuring the high performance of OLAP queries and guaranteeing the snapshot isolation level. Doing so while also maintaining the wait-free feature remains a challenge that must be addressed.

3 OLTP Components

3.1 Storage Engine

It is better for a system to expose one single schema while maintaining two data replicas for OLTP and OLAP. The storage engine should also employ two extra

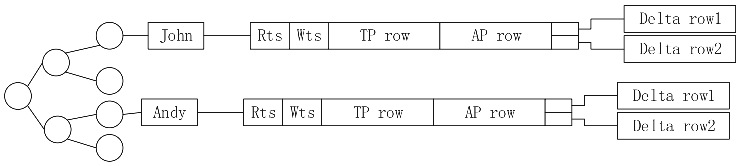


Fig. 2. Dual snapshot-based storage engine.

memory spaces to record the delta snapshots. For each tuple, we use two counters to represent the read and write timestamps for concurrency control, similar to the approach presented in TicToc [42]. The storage engine is organized in the form of a tuple level. Each logical tuple contains two static in-memory rows that store the OLTP and OLAP data, one set per row. Additionally, the logical tuple holds two copies of dynamic data, $delta_1$ and $delta_2$ (Fig. 2). The dynamic data are used to record increments in an alternating fashion. The use of dynamic memory can save memory footprint as desired, while static memory can help avoid memory allocation times.

Many indexes have been designed to organize in-memory row data, such as the Adaptive Radix Tree (ART) [18,19], BwTree [20], Masstree [27], and SkipList [34]. [40] gives an in-depth performance evaluation of the state-of-the-art indexes. How the appropriate index is chosen is beyond the scope of this paper. To access the table in a fast and easy manner, we can exploit a hash index or use a memory-optimized b+ tree for organizing the tuples. The OLTP, OLAP and dual delta snapshot components share the same index.

3.2 Transactions and Concurrency Control

Component 2 (Fig. 1) entails choosing an OCC-based protocol to schedule the OLTP workload. One of the largest advantages of using the optimistic concurrency control protocol for main memory DBMSs is that the contention period is short, because transactions write their updates to shared memory only at the commit time [15]. This outstanding feature could be combined with a recording snapshot, *i.e.*, each write operation occurring in the write phase will be accompanied by a **write_delta** (details are available in Sect. 3.3) operation synchronously. As a result, we chose OCC as WHTAP's transaction execution protocol. Our baseline OCC protocol is specified in TicToc [42], which details a new concurrency control algorithm that achieves higher concurrency than state-of-the-art T/O schemes. Another popular OCC protocol that can be used to replace TicToc is Silo [39]. Although our algorithms can also be adapted to the multiple-version concurrent control, we abandon MVCC in designing our WHTAP system due to its high overhead memory cost.

3.3 Delta Snapshot

Our concurrency control algorithm falls under the framework of TicToc, although we must record the delta snapshots in conjunction with transaction execution.

We integrate Pingpong [3] (or Hourglass [24]) with TicToc. Each OLTP's write instruction (only appearing in the write phase) should not only write to the OLTP data but also write to a particular delta snapshot according to the pointer, as introduced in Pingpong. Both Pingpong and Hourglass can be combined with a virtual snapshot approach [23,35], *i.e.*, when we take virtual snapshots into consideration, once the pointers have been exchanged, this approach only affects the behavior of the new transactions and has no effect on the current running transactions.

Algorithm 1. Transaction Execution Thread

```
   Input: Transaction $T$
1  Read Phase
2  if Validation Passed then
3      pointer = $p\_update$
4      for each request in $T.writeset$ do
5          write(index(request.key))
6          malloc_delta($pointer$)
7          write_delta(index(request.key), $pointer$)
8          KeySet.insert(request.key)
9          index(request.key).wts = commit_ts
10         index(request.key).rts = commit_ts
11         unlock(index(request.key))
```

Since TicToc accesses OLTP data only during the write phases, we can improve the algorithms by making WHTAP only determine to which period (odd or even period) the time point (at the beginning of the write phase) belongs. That is, pointer swapping (see Algorithm Sect. 2 Line 5) only affects the transactions that extend into the write phase after the pointer swapping moment.

Algorithm 1 presents the details of the transactions' concurrency control and the delta recording process. The read and validation phase is the same with TicToc, so the details regarding line 1 and line 2 can be found in [42]. The primary difference between WHTAP and TicToc exists in the write phase. Once deep into the write phase, the transaction first decides which period it belongs to (line 3) and then determines whether the delta snapshot should be recorded in $delta_1$ or $delta_2$. The thread commits the writeset data to OLTP data (line 5) and the corresponding periodic delta snapshot (lines 6 and 7). Memory needs to be allocated dynamically (line 6) before each delta recording operation. Considering that malloc is very time-consuming, we set $delta_1$ and $delta_2$ to be static memory spaces. Since the storage engine shares the index structure, our approach requires an additional KeySet to record the modified keys (line 8) during the current period. If we maintain the index structure separately for the delta structure, then the KeySet here is unnecessary.

4 OLAP Components

4.1 Compact Snapshot and State Controller

Section 3.3 describes an approach for freezing delta snapshots interchangeably. With the help of the dual snapshot scheme, it is possible to always maintain the data freshness of the OLAP component by frequently compacting the frozen data in the snapshot into the OLAP store. However, the approach that must be used to compact the delta snapshot into the existing OLAP data without blocking the execution of the OLAP query and while maintaining the model's high performance is a challenge.

We propose to solve the problem and make OLAP wait-free by distinguishing and scheduling five states/phases, *i.e.*, *NORMAL*, *FROZEN*, *WAITING*, *COMPACTION*, and *GC* phase, between consecutive pointer swapping events.

***NORMAL* Phase.** In the *NORMAL* phase, every update transaction that reaches the write phase should write to both the OLTP store and the delta snapshot (lines 5 and 7 in the Algorithm 1). Long-running OLAP queries must directly consume data in the OLAP store in a consistent manner. Therefore, the time duration of this phase becomes a key factor with regard to the data freshness of OLAP.

***FROZEN* Phase.** The *FROZEN* phase occurs after the *NORMAL* phase. The delta snapshot must be frozen in this phase. Accordingly, The first step is to swap the *p_update* and *p_delta* pointers, in other words, to exchange the dual snapshot roles. As an exception, the transactions with write phases that begin in the *NORMAL* phase but have not yet been committed at this point (called *dirty* transaction) should ignore the role change operations. The completion of all the dirty transactions denote the end time point of the *FROZEN* phase. At the end boundary of the *FROZEN* phase, the delta snapshot is frozen, meaning that no OLTP transaction should write data to this dataset. We can see that at the beginning of the *FROZEN* phase, all of the new OLTP transaction write data should be allowed to write to another delta snapshot.

A global counter is needed to identify the end times of all of the dirty transactions. The use of a global counter will affect performance, this element behaves as a bottleneck in a multi-core system [39]. Because the OLTP transaction is usually short, we assume that the OLTP transaction is less than 1 µs, we simply wait for 1 µs because the transaction time under the memory database will be very short. It should be noted that this time can be directly adjusted. The specific code corresponds to line 6 in the Algorithm 2.

***WAITING* Phase.** The Waiting phase begins at the completion of the *FROZEN* phase and includes all transactions started when the *NORMAL* phase commenced. At the beginning of the *WAITING* phase, the delta frozen snapshot is generated. The snapshot cannot be compacted into the OLAP data because several active OLAP queries are still running; hence, we must wait for all active OLAP queries to finish (*i.e.*, *static_counter* $= 0$). Because those transactions are queries from OLAP data, the data should not be able to write. Once those queries have finished, we reach the end point of the *WAITING* phase.

We innovatively introduce a Log Structure Merge tree [12] approach for query execution. Here, the frozen delta snapshot can be regarded as a *MemTable* like that in the LSM tree, while OLAP data storage can be regarded as an *SSTable*. Once the delta snapshot is generated (*i.e.*, has been frozen), the system compacts the snapshot into the OLAP data. Any query transaction must first search the delta and then access the OLAP data if the result is not found in the delta. The queries that begin in the *WAITING* phase should query the delta first. For the wait-free consideration, since the "MemTable" has been generated, we can query more fresh data from the delta and then can merge the data into the OLAP (SSTable).

***COMPACTION* Phase.** The *WAITING* phase ends when all queries begun in the *GC* and *NORMAL* and *FROZEN* phases have been committed. Next, the system enters the *COMPACTION* phase. Once this phase begins, the delta data are traversed sequentially according to the keyset, then compacted into the OLAP data set. If the OLAP storage engine is column-major designed, this approach will be more complicated.

For querying transactions, we still require an LSM-like query approach. First, we query the frozen delta, and if the result is found, we return it directly; otherwise, we continue the search in the OLAP storage. We can even add a bloom filter to the frozen delta.

Note that in this process, because our OLAP query looks up the delta first, if the query finds the result and hence does not need to look up the OLAP data, then the compaction work can be conducted in a lock-free manner and will not cause any blocks.

***GC* Phase.** GC is the last phase of the cycle and immediately follows the *COMPACTION* phase. Hence, any query beginning in the *GC* phase can read the latest data directly from the OLAP. As the frozen delta snapshot has been successfully compacted, the snapshot can be released just like the process of "Garbage Collection". The process remains in the *GC* phase until the completion of all queries issued during the Waiting and Compaction phases. When the *GC* phase finished, the system has now entered the next cycle of the *NORMAL* phase.

Algorithm Code. As shown in the Algorithm 2, the system alternates between five phases. From the normal to the frozen phase (line 3), we can control the duration of this phase. If the time duration is too long, the data of the OLAP will be out of date, because the data freshness will not be good enough. Once the system enters the *FROZEN* phase, we must exchange the roles of the dual-snapshot and wait for the dirty transaction to commit. Note that in line 6 of the code, for performance reasons, we can directly wait for a period of time (such as 1 µs) to replace the function to ensure that the dirty transactions are all committed. For the duration of the waiting phase, we need to use the counter

Algorithm 2. State Controller and Compaction Work

```
1  while true do
2      State = NORMAL
3      Waiting for the delta snapshot frozen signal
4      State = FROZEN
5      Swap(p_update, p_delta)
6      Wait Until Dirty Transaction Finished
7      State = WAITING
8      wait until static_counter = 0
9      State = COMPACTION
10     Compact(p_delta, AP)
11     State = COMPLETE
12     Gargage_Colection(p_delta)
13     wait until delta_counter = 0
```

to determine whether the transactions (which were queried directly from the OLAP) are all committed. The compact function (line 10) must be used to merge the data by engaging KeySet. In the GC phase, the main goal is to free useless data and wait until the LSM-like queries have finished.

4.2 LSM-Like Query Layer

In Sect. 3.2, we discuss approaches to handling the OLTP workload; in this part, we describe the OLAP workload running process.

Once a query request is accepted, the system detects the state phase in which it was begun. For performance reasons, only in the *WAITING* and *COMPACTION* phases do we execute the LSM-like query strategies; in contrast, in the *NORMAL*, *FROZEN* and *GC* phases, we query the OLAP data directly. As shown in the Algorithm 3, line 5 **Query_Static()** and line 11 **Query_DeltaFirst()** represent two different query strategies. The 2 counters, static_counter and delta_counter, are essential for identifying the state in which the system is functioning.

Algorithm 3. Query Execution Thread

```
   Input: Query Q
1  start_state = State
2  if start_state = GC||NORMAL||FROZEN then
3      fetch_and_add(static_counter)
4      for each request in Q do
5          Query_Static(index(request.key))
6      fetch_and_sub(static_counter)
7  else
8      if start_state = WAITING||COMPACTION then
9          fetch_and_add(delta_counter)
10         for each request in Q do
11             Query_DeltaFirst(index(request.key)) //LSM-alike query
12         fetch_and_sub(delta_counter)
```

The performance associated with looking up a delta first is worse (slower) than the performance achieved by querying OLAP data directly. Fortunately, **Query_DeltaFirst()** is a relatively rare event, because the frozen and compact phases are short. The next section elaborates on this idea.

4.3 Running Example

To enhance clarity in our explanation of the system, the Fig. 3 is presented, detailing a running example of our WHTAP algorithm. This figure fully illustrates the five phases in a cycle and shows several types of transactions running.

As shown in the Fig. 3, green represents each transaction that successfully enters the write phase within the current cycle. Write transactions for the previous period are shown in yellow. After the system triggers the frozen signal (at time t_1), the new transaction entering the write phase (green, T_2, T_3, T_4, T_5, T_6, T_7) records the delta into another data set. Transactions that have not been completed (T_2) continue to execute; once they are finished, the system exits the frozen phase.

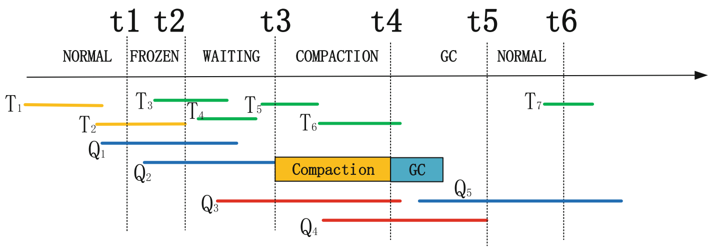


Fig. 3. Running example of WHTAP. (Color figure online)

At time t_2, the system has obtained a stable delta snapshot of the previous cycle. Next, we can compact the delta snapshot from the previous cycle into the OLTP storage. Unfortunately, several active OLAP queries (Q_1, Q_2) remain uncommitted. As a result, the delta snapshot may overwrite this part of the data during direct compact, causing a query error. Therefore, we must invoke a waiting phase to wait for the blue query transaction in the figure to complete. For queries that are started during the waiting phase (Q_3), we need to query the delta snapshot first to execute the OLAP queries, since this approach will ensure that the write phase of the *COMPACT* phase can merge in a lock-free manner. Once transactions Q_1 and Q_2 are committed, the system can start the compact operation. The yellow rectangle in the figure represents the *COMPACT* operation, which basically merges the incremental transaction data for the previous cycle into the OLAP store. Similarly, any query started in this phase is still queried in a similar way as those in LSM. Once this phase is over and the incremental data from the previous cycle have become useless, we can naturally delete these data (blue rectangle). At the same time, we must also wait for the end of the transaction for the LSM-style query (Q_4). Once that point has been reached, we can proceed to the next normal phase of the cycle and once more wait for the frozen signal to trigger.

5 Experimental Study

This section implements and evaluates the WHATP prototype. The prototype is based on DBx1000 OLTP DBMS [41]. To integrate our WHTAP algorithm into the DBx1000 system, we need to modify the storage engine and the concurrent processing of the OLTP transaction and then must separate the OLAP queries from the OLTP component, and add the state controller and LSM-like OLAP query methods. The prototype allows us to compare five approaches all within the same system: TicToc [42], SILO [39], HEKATON [5], Basic MVCC and WHTAP. All of the experiments are run on a high-end server, which is equipped with two E7-4820 CPU sockets each with 40 physical sockets, 512 GB of memory, and 1 TB of hard disk drive space. CentOS 7.3 X86_64 with Linux kernel 3.10 and g++ 4.8 is installed.

5.1 Benchmark Setup

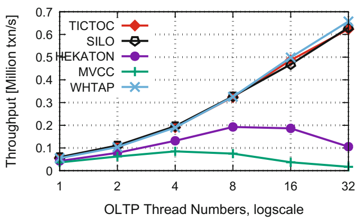
The experiments focus on OLTP and OLAP mixed transactions. The performance of the HTAP workloads was evaluated with the YCSB [4] benchmark. Because YCSB is used to evaluate the OLTP workload, we must modify the origin benchmark to support the evaluation of both transaction and long-running read-only queries. First, we prepared a single table database for the experiments. The schema of the database table was configured to own 11 columns. The first column is the primary key, and the following 10 columns store randomly generated string values. Initially, 10M records were bulk-loaded into the table. The accesses (reads or writes) of records follow a Zipfian distribution that can be controlled by parameter θ (reflecting the level of contention).

For OLTP transactions, 16 accesses per transaction (50% reads and 50% writes) with a hotspot of 10% tuples accessed by 75% of all queries ($\theta = 0.9$). For OLAP queries, 48 queries (100% reads) per transaction and a uniform access distribution ($\theta = 0$).

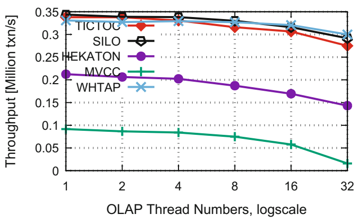
The first group evaluation fixes the number of threads in OLAP and tests the performance impact of the number of OLTP thread. In the second group evaluation, the number of threads in the OLTP is fixed, and the performance impact of the number of OLAP threads is tested.

5.2 Fixing OLAP Threads

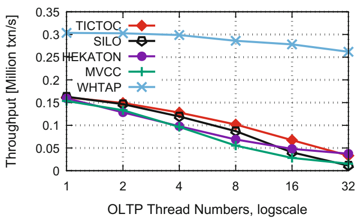
We fix the number of OLAP threads to 8 and then fix the length of each OLAP query to 48. Figure 4(a) shows the performance results of OLTP when the number of OLAP threads is fixed. The horizontal axis corresponds to the number



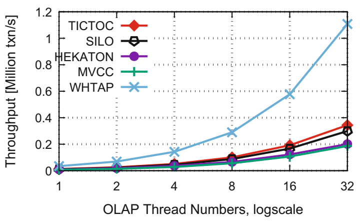
(a) OLTP performance (OLAP thread = 8)



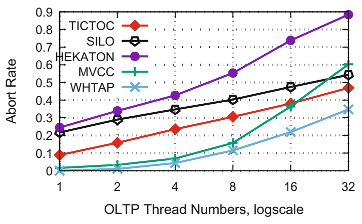
(b) OLTP performance (OLTP thread = 8)



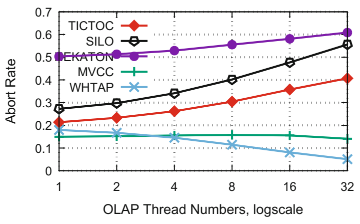
(c) OLAP performance (OLAP thread = 8)



(d) OLAP performance (OLTP thread = 8)



(e) Abort rate (OLAP thread = 8)



(f) Abort rate (OLTP thread = 8)

Fig. 4. Performance of HTAP

of threads in OLTP from 1 to 32, which shows that the OLAP workload has little impact on OLTP performance. Figure 4(c) shows the performance results of OLAP in which the number of OLAP threads is 8, and the number of OLTP threads is from 1 to 32. The performance of OLAP is 2–3 times that of other algorithms. Increasing the number of OLTP threads in the WHTAP algorithm has little impact on OLAP performance, which effectively does not decrease with the increase of OLTP threads. In the traditional concurrency control algorithm, as the number of OLAP threads is increased, the performance decreases of the algorithm decreases significantly. In particular, even MVCC and HEKATON are suitable for reading operations, but the performance of OLAP remains poor due to version chain scanning. Figure 4(e) gives the effect of the number of OLAP threads on the abort rate. As it can be seen, our algorithm has the lowest abort rate (substantially lower than that of TicToc), because our algorithm's OLAP thread is executed on the snapshot and does not require abort at all.

5.3 Fixing OLTP Threads

Figure 4(b) shows the performance results of OLTP when the number of OLTP threads is fixed. The horizontal axis corresponds to the number of threads of OLAP and ranges from 1 to 32. When the OLAP's workload is increased, the OLTP performance decreases accordingly. The performance of the OCC-based single-version concurrent control algorithm should be comparatively worse. As the number of OLAP threads increases, the performance of OLTP decreases. MVCC decreases significantly, while the OCC scheme decreases by a small amount. The OLTP workload performance of WHTAP is slightly worse than that of TicToc. Figure 4(d) shows the performance results of OLAP when the number of OLTP threads is fixed. We can see that the performance of WHTAP has increased significantly, mainly because the algorithm benefits from technologies that are processed separately from OLTP and OLAP. When the number of OLAP threads is 32, WHTAP has 3.2 times the performance of TicToc and 5.5 times that of HEKATON. Figure 4(f) shows the number of system rollback transactions when the number of OLTP threads is fixed. As the OLAP workload increases, the number of WHTAP rollback transactions is trending downward, because OLAP transactions are certain to execute and do not abort. The read-only queries of other algorithm can also result in abort events.

Combining the above results, we obtain the following findings:

- When both OLTP and OLAP workloads exist at the same time in a given scenario, WHTAP outperforms other algorithms. The performance of the OLTP part is close to that of TicToc (Fig. 4(a) and (b)), although WHTAP has an absolute advantage over the performance of the OLAP part (Fig. 4(c) and (d)).
- In the WHTAP algorithm, OLTP and OLAP workloads have the least impact on each other's performance (Fig. 4(c) and (b)).

6 Related Work

HTAP. Hybrid transaction/analytical processing (HTAP) was first defined by Gartner Inc. [8,9] and referred to a novel architecture that could effectively combine both types of processing to fulfill the emerging requests in informative and real-time decision making activities. A recent works [1,10,31] survey on this topic was conducted. HTAP emphasizes two main points: data freshness and unified data representation. Since the 2000s, many systems have targeted the HTAP market. Typical systems include SAP HANA [7,38], Hyper [30] and HYRIES [11], Peleton [32]. The other systems (including MemSQL [28], Hekaton [5] and Apollo [16]) can support OLTP and OLAP workload separately.

The effective design of an HTAP system remains an open problem that can be broadly divided into three categories. (1) One straightforward method is to use a single system to process the mixed workload. Hyper [30] proposed a novel MVCC implementation that can update in place and store prior versions before image deltas, enabling both an efficient scan execution and the fine-grained serializability validation needed for the rapid processing of point access transactions. From the NoSQL side, Pilman et al. [33] demonstrated how scans can be efficiently implemented on a key value store (KV store) to enable more complex analytics with large and distributed KV stores. (2) The second method is to create a copy of the main database, which is then used as an OLAP dataset. Specifically, Hyper [13] and Swingdb [29] use such an approach. (3) The 3rd method is to generate a delta snapshot, which is then periodically merged into the second replica. Specifically, BatchDB and SAP HANA do this.

In-Memory Concurrency Control. The traditional OCC [15] algorithm is found to be effective in the control of concurrency in OLTP with the advent of new high-performance DBMSs. For example, OCC variants are commonly used in in-memory database scenarios and include examples such as TicToc [42] and SILO [39].

MVCC and its variants constitute another type of concurrency control algorithm that has been widely adopted in in-memory databases. Such algorithms can be found in Hekaton [5,17], HyPer [30], Bohm [6], Deuteronomy [21,22] and ERMIA [14] and Cicada [25]. For a record, MVCC creates multiple version copies to reduce conflict between transactions and allow access to earlier versions of a record.

Frequent Snapshot. To obtain a consistent snapshot, Salles *et al.* [36] provides a comparison of several state-of-the-art snapshot algorithms. In this work, the authors concluded that Naive Snapshot is good for high-throughput workloads with small datasets, whereas Copy On Update is more widely applicable. Swingdb [29] and Hyper [13] work by modifying the Linux kernel to support a fine-grained fork-like system that is then called to generate the snapshot. Zigzag [3] was developed for use in MMO game scenarios, although it is suitable for use only with small datasets and is not a more generally applicable algorithm. Pingpong, Hourglass and Piggyback [24] all use a kind of pointer swapping technique to generate snapshots. Moreover, both Pingpong and Hourglass can be used to generate a delta snapshot, and therefore are useful with HTAP systems. It is important to note that all of these snapshot algorithms depend on a physical consistent time-point. In contrast, for more widely applicable cases, such as OLTP, to establish a physical consistent state in a running system, system blocks must be introduced. In the most recent work on this subject, CALC [35], the authors invent a virtual snapshot idea to solve the problem; in this case, it would be possible to integrate both Pingpong and Hourglass with this idea.

7 Conclusion

In this paper, we proposed a wait-free HTAP control protocol and implemented a prototype (WHTAP). We used WHTAP to exploit a dual-snapshot structure to isolate OLTP and OLAP workloads. The OLTP transaction was able to run in a serializable context, whereas OLAP was able to yield isolated snapshots. Our prototype enabled both OLTP and OLAP to simultaneously function in a high throughput manner, furthermore, both OLTP and OLAP could be executed in a wait-free manner. We evaluated the performance of our prototype using the modified YCSB benchmark to validate our performance with regard to the HTAP workload. Compared with the TicToc concurrency control protocol, WHTAP can achieve an OLTP performance that is similar to that of TicToc, and in contrast, the performance of OLAP with our algorithm is approximately four times that of TicToc. We strongly recommend that developers in scenarios in which significant data quantities are processed use the WHTAP architecture for scenarios that require real-time analysis and processing, such as “Amazon’s Prime Day”.

Acknowledgements. Guoren Wang is the corresponding author of this paper. Guoren Wang is supported by the NSFC (Grant No. U1401256, 61732003, 61332006 and 61729201). Gang Wu is supported by the NSFC (Grant No. 61872072) and the State Key Laboratory of Computer Software New Technology Open Project Fund (Grant No. KFKT2018B05).

References

1. Bohm, A., Dittrich, J., Mukherjee, N., Pandis, I., Sen, R.: Operational analytics data management systems. PVLDB **9**(13), 1601–1604 (2016)
2. Braun, L., et al.: Analytics in motion: high performance event-processing and real-time analytics in the same database. In: SIGMOD, pp. 251–264 (2015)
3. Cao, T., et al.: Fast checkpoint recovery algorithms for frequently consistent applications. In: SIGMOD, pp. 265–276 (2011)
4. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC, pp. 143–154. ACM (2010)
5. Diaconu, C., et al.: Hekaton: SQL server’s memory-optimized OLTP engine. In: SIGMOD, pp. 1243–1254. ACM (2013)
6. Faleiro, J.M., Abadi, D.J.: Rethinking serializable multiversion concurrency control. PVLDB **8**(11), 1190–1201 (2015)
7. Farber, F., Cha, S.K., Primsch, J., Bornhovd, C., Sigg, S., Lehner, W.: SAP HANA database: data management for modern business applications. Sigmod Rec. **40**(4), 45–51 (2012)
8. Gartner: hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. https://www.gartner.com/doc/2657815/hybrid-transactionanalytical-processing-foster-opportunities
9. Gartner: market guide for HTAP-enabling in-memory computing technologies. https://www.gartner.com/doc/3599217/market-guide-htapenabling-inmemory-computing
10. Giceva, J., Sadoghi, M.: Hybrid OLTP and OLAP. In: Sakr, S., Zomaya, A. (eds.) EBDT. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-63962-8
11. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., Madden, S.: HYRISE: a main memory hybrid storage engine. PVLDB **4**(2), 105–116 (2010)
12. Jagadish, H.V., Narayan, P.P.S., Seshadri, S., Sudarshan, S., Kanneganti, R.: Incremental organization for data recording and warehousing. In: VLDB 1997 (1997)

13. Kemper, A., Neumann, T.: HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE, pp. 195–206. IEEE (2011)
14. Kim, K., Wang, T., Johnson, R., Pandis, I.: ERMIA: fast memory-optimized database system for heterogeneous workloads. In: SIGMOD, pp. 1675–1687 (2016)
15. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6**(2), 213–226 (1981)
16. Larson, P.A., Birka, A., Hanson, E.N., Huang, W., Nowakiewicz, M., Papadimos, V.: Real-time analytical processing with SQL server. Proc. VLDB Endow. **8**(12), 1740–1751 (2015)
17. Larson, P., Blanas, S., Diaconu, C., Freedman, C.S., Patel, J.M., Zwilling, M.: High-performance concurrency control mechanisms for main-memory databases. PVLDB **5**(4), 298–309 (2011)
18. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: ICDE. pp. 38–49 (2013)
19. Leis, V., Scheibner, F., Kemper, A., Neumann, T.: The art of practical synchronization. In: DaMoN, pp. 1–8 (2016)
20. Levandoski, J.J., Lomet, D.B., Sengupta, S.: The Bw-Tree: a B-tree for new hardware platforms. In: ICDE, pp. 302–313 (2013)
21. Levandoski, J.J., Lomet, D.B., Sengupta, S., Stutsman, R., Wang, R.: High performance transactions in deuteronomy. In: CIDR (2015)
22. Levandoski, J.J., Lomet, D.B., Sengupta, S., Stutsman, R., Wang, R.: Multi-version range concurrency control in deuteronomy. PVLDB **8**(13), 2146–2157 (2015)
23. Li, L., Wang, G., Wu, G., Yuan, Y., Chen, L., Lian, X.: A comparative study of consistent snapshot algorithms for main-memory database systems. ArXiv e-prints, October 2018
24. Li, L., Wang, G., Wu, G., Yuan, Y.: Consistent snapshot algorithms for in-memory database systems: experiments and analysis. In: ICDE, pp. 1284–1287 (2018)
25. Lim, H., Kaminsky, M., Andersen, D.G.: Cicada: dependably fast multi-core in-memory transactions. In: SIGMOD, pp. 21–35 (2017)
26. Makreshanski, D., Giceva, J., Barthels, C., Alonso, G.: BatchDB: efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In: SIGMOD, pp. 37–50 (2017)
27. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: ACM European Conference on Computer Systems, pp. 183–196 (2012)
28. MemSQL: MemSQL. https://www.memsql.com/
29. Meng, Q., Zhou, X., Chen, S., Wang, S.: SwingDB: an embedded in-memory DBMS enabling instant snapshot sharing. In: Blanas, S., Bordawekar, R., Lahiri, T., Levandoski, J., Pavlo, A. (eds.) IMDM/ADMS -2016. LNCS, vol. 10195, pp. 134–149. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56111-0_8
30. Neumann, T., Muhlbauer, T., Kemper, A.: Fast serializable multi-version concurrency control for main-memory database systems. In: SIGMOD, pp. 677–689 (2015)
31. Ozcan, F., Tian, Y., Tozun, P.: Hybrid transactional/analytical processing: a survey. In: SIGMOD, pp. 1771–1775 (2017)
32. Pelotondb: Pelotondb. https://pelotondb.io/
33. Pilman, M., Bocksrocker, K., Braun, L., Marroquín, R., Kossmann, D.: Fast scans on key-value stores. PVLDB **10**(11), 1526–1537 (2017)
34. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. In: The Workshop on Algorithms & Data Structures, pp. 668–676 (1990)

35. Ren, K., Diamond, T., Abadi, D.J., Thomson, A.: Low-overhead asynchronous checkpointing in main-memory database systems. In: SIGMOD, pp. 1539–1551. SIGMOD (2016)
36. Salles, M.A.V., et al.: An evaluation of checkpoint recovery for massively multi-player online games. PVLDB **2**(1), 1258–1269 (2009)
37. Sharma, A., Schuhknecht, F.M., Dittrich, J.: Accelerating analytical processing in MVCC using fine-granular high-frequency virtual snapshotting. In: SIGMOD (2017)
38. Sikka, V., Farber, F., Goel, A.K., Lehner, W.: SAP HANA: the evolution from a modern main-memory data platform to an enterprise application platform. PVLDB **6**(11), 1184–1185 (2013)
39. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: SOSP, pp. 18–32 (2013)
40. Xie, Z., Cai, Q., Chen, G., Mao, R., Zhang, M.: A comprehensive performance evaluation of modern in-memory indices. In: ICDE, pp. 641–652. IEEE (2018)
41. Yu, X., Bezerra, G., Pavlo, A., Devadas, S., Stonebraker, M.: Staring into the abyss: an evaluation of concurrency control with one thousand cores. PVLDB **8**(3), 209–220 (2014)
42. Yu, X., Pavlo, A., Sanchez, D., Devadas, S.: TicToc: time traveling optimistic concurrency control. In: SIGMOD, pp. 1629–1642 (2016)