A Comparative Study of Consistent Snapshot Algorithms for Main-Memory Database Systems

Liang Li, Guoren Wang, Gang Wu, Ye Yuan, Lei Chen, and Xiang Lian, *Member, IEEE*

Abstract—In-memory databases (IMDBs) are gaining increasing popularity in big data applications, where clients commit updates intensively. Specifically, it is necessary for IMDBs to have efficient snapshot performance to support certain special applications (e.g., consistent checkpoint, HTAP). Formally, the in-memory consistent snapshot problem refers to taking an in-memory consistent time-in-point snapshot with the constraints that 1) clients can read the latest data items and 2) any data item in the snapshot should not be overwritten. Various snapshot algorithms have been proposed in academia to trade off throughput and latency, but industrial IMDBs such as Redis adhere to the simple fork algorithm. To understand this phenomenon, we conduct comprehensive performance evaluations on mainstream snapshot algorithms. Surprisingly, we observe that the simple fork algorithm indeed outperforms the state-of-the-arts in update-intensive workload scenarios. On this basis, we identify the drawbacks of existing research and propose two lightweight improvements. Extensive evaluations on synthetic data and Redis show that our lightweight improvements yield better performance than fork, the current industrial standard, and the representative snapshot algorithms from academia. Finally, we have opensourced the implementation of all the above snapshot algorithms so that practitioners are able to benchmark the performance of each algorithm and select proper methods for different application scenarios.

Index Terms—In-memory database systems, snapshot algorithms, checkpoints, HTAP

1 Introduction

In-memory databases (IMDBs) [1] have been widely adopted in various applications as the back-end servers, such as e-commerce OLTP services, massive multiple online games [2], electronic trading systems (ETS) and so on. For these applications, it is common to support both intensively committed updates and efficient consistent snapshot maintenance. Here, we use in-memory consistent snapshot to emphasize taking an in-memory consistent time-in-point snapshot with the constraints that (1) clients can read the latest data items, and (2) any data item in the snapshot should not be overwritten. In-memory consistent snapshot can be applied in diverse real-life applications. Representative examples include but are not limited to the following.

- *Consistent Checkpoint:* System failures are intolerable in many business systems. For instance, Facebook was out of service for approximately 2.5 hours in 2010. There was a worldwide outage, and 2.8 TB memory data were cleared [3]. Consistent checkpoints are important to avoid long-time system failures and support rapid recovery; in-memory systems such as Hekaton[4] and Hyper [5] typically perform *consistent checkpoint* frequently. Checkpoint works by taking a "consistent memory snapshot" of the runtime system and dumping the snapshot asynchronously. The key step is to take a consistent snapshot efficiently. Inefficient snapshot algorithms may accumulatively lead to system performance degradation and thus unacceptable user experience in update-intensive applications.
- *Hybrid Transactional/Analytical Processing Systems (HTAP):* Hybrid OLTP&OLAP in-memory systems are gaining increasing popularity [6], [7], [8], [9], [10], [11], [12], [13], [14], [15] In traditional disk-resident database systems, the OLTP system needs to extract and transform data to the OLAP system. That is, OLTP and OLAP are usually separated in two systems. Due to the high performance of in-memory database systems, it becomes viable to exploit OLTP snapshot data as an OLAP task and build a hybrid system. In fact, database vendors including Hyper [8], SAP HANA [13], [14], SwingDB [12] and ANKER [16] have already applied in-memory snapshot algorithms in Hybrid Transactional/Analytical Processing Systems.

However, the unavoidable fact is that the accumulated latency brought by the snapshot maintenance may have significant impacts on system throughput and response time. Improper handling of snapshot may result in latency spikes

- L. Li and Y. Yuan are with the Department of Computer Science, Northeastern University of China, Shenyang 100819, China. E-mail: liliang@stumail.neu.edu.cn, yuanye@mail.neu.edu.cn.
- G. Wang is with the Department of Computer Science, Beging Institute of Technology, Haidian, Beijing 100081, China. E-mail: wanggr-bit@126.com.
- G. Wu is with the Department of Computer Science, Northeastern University of China, Shenyang 100819, China, and also with the Department of State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing Shi, Jiangsu Sheng 210008, P.R. China. E-mail: wugang@mail.neu.edu.cn.
- L. Chen is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. E-mail: leichen@cse.ust.hk.
- X. Lian is with the Department of Computer Science, Kent State University, Kent, OH 44240. E-mail: xlian@kent.edu.

Manuscript received 28 Aug. 2018; revised 31 May 2019; accepted 16 July 2019. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding authors: Guoren Wang, Gang Wu and Ye Yuan.)
Recommended for acceptance by by P. K. Chrysanthis, B. C. Ooi, and J. Dittrich.
Digital Object Identifier no. 10.1109/TKDE.2019.2930987

and even system stalls. Thus, pursuing a fast snapshot with low and uniform overhead, or one that is *lightweight*, is the focus of in-memory snapshot algorithms.

The wide applications of in-memory consistent snapshot have attracted the interest of academia. Some representative snapshot algorithms are Naive Snapshot (NS) [17], [18], Copy-on-Update (COU) [2], [19], [20], Zigzag (ZZ) [21] and PingPong (PP) [21]. In addition, the simple fork [22] function is used as a common snapshot algorithm in industrial systems. However, it is often difficult for practitioners to select the appropriate in-memory snapshot algorithm due to the lack of a unified, systematic evaluation on existing snapshot algorithms. This work is primarily motivated by this absence of performance evaluation, which is described in more detail as follows.

1.1 Motivation

1. Why do popular industrial IMDBs, e.g., Redis/Hyper, utilize the simple fork() function instead of state-of-the-art snapshot algorithms? As mentioned above, various in-memory consistent snapshot algorithms have been proposed in academia to trade off between latency and throughput. However, it is interesting that popular industrial IMDBs such as Redis/Hyper still apply the simple fork() function as the built-in algorithm for consistent snapshot. It is worth investigating whether this is due to the simplicity of fork()'s engineering implementation or its good system performance (e.g., high throughput and low latency).

2. Are state-of-the-art snapshot algorithms inapplicable to update-intensive workload scenarios? Many modern in-memory applications are highly interactive and involve intensive updates. The performance of the state-of-the-arts from academia and industry in large-scale update-intensive workload scenarios is not known. If no existing algorithms fit, can we modify and improve the state-of-the-arts for this scenario?

3. Can we provide unified implementation and benchmark studies for future studies? A frustrating aspect of snapshot algorithm research is the lack of a unified implementation for fair and reproducible performance comparisons. Since new application scenarios are continually emerging, researchers would benefit by making unified implementation and evaluation of existing snapshot algorithms accessible to all.

1.2 Contributions

1. We find that the simple fork() function indeed outperforms the state-of-the-arts in update-intensive workload scenarios. Snapshot algorithms for update-intensive workloads should have consistently low latency. This requirement can be assessed by average latency and latency spikes. We conduct large-scale experiments on five mainstream snapshot algorithms (NS, COU, ZZ, PP, Fork). NS has low average latency but also high-latency spikes, meaning high latency when taking snapshots. In contrast, PP has no latency spikes but incurs higher average latency. Surprisingly, we observe that the simple fork algorithm indeed outperforms the remaining algorithms. That is, fork() has low average latency and almost no high latency spikes. These experimental results can explain why popular industrial IMDBs prefer the simple fork algorithm rather than state-of-the-art algorithms from academia.

2. We propose two simple yet effective modifications of the state-of-the-arts that exhibit better tradeoff among latency, throughput, complexity and scalability. Based on the aforementioned experiments with mainstream snapshot algorithms, we identify the drawbacks of the existing research and propose two lightweight improvements based on state-of-the-art snapshot algorithms. In particular, extensive evaluations on synthetic data and Redis, the popular industrial IMDB, show that our lightweight improvements yield better performance than fork, the current industrial standard, and the representative snapshot algorithms of academia. In addition, the algorithms can not only easily adapt to widely used cases but also maintain good performance with the snapshot technique.

3. We opensource our implementations, algorithmic improvements, and benchmark studies as guidance for future researchers. We implement five mainstream snapshot algorithms and two improved algorithms and conduct comprehensive evaluations on synthetic datasets. The implementations and evaluations have been released on GitHub.[1] We further integrate the two improved algorithms into Redis and investigate the scalability with the Yahoo! Cloud Serving Benchmark (YCSB) [23]. The implementations and evaluations are also publicly accessible.[2] We envision our experiences as providing valuable guidance for future snapshot algorithm design, implementation, and evaluation.

This paper is a complete description of a previous brief version of this work [24]. The main additions include a number of examples in the background and motivation, the theoretical foundation and implementation of our algorithms, and presentation and analysis of extensive experimental results. Furthermore, we adapt the proposed algorithms to the more general concurrent transaction-execution case for comparison with the CALC algorithm [25].

The rest of the paper is organized as follows. In Section 2, we define and model the problem of consistent snapshot. Existing algorithms and two proposed algorithms are detailed in Section 3. We discuss a more general case in Section 4. To show the feasibility of the algorithms, we first evaluate them with a synthetic dataset in Section 5.2 and then integrate them into Redis and benchmark them with YCSB in Section 5.4. We conclude in Section 6.

2 Preliminaries

2.1 Problem Statement

In this work, we compare, analyze and improve snapshot algorithms designed for in-memory databases, particularly in update-intensive scenarios. First, we formally define the in-memory consistent snapshot problem as follows.

Definition 1 (In-Memory Consistent Snapshot). *Let D be an update intensive in-memory database. A consistent snapshot is a consistent state of D at a particular time-in-point, which should satisfy the following two constraints:*

1. https://github.com/bombehub/FrequentSnapshot.git
2. https://github.com/bombe-org/RedisPersistent.git

TABLE 1
Update Data for the Single *Client* Thread

Period	Updates	Data to be Updated
P_1	T_1	$\langle 0, 13 \rangle$
	T_2	$\langle 2, 16 \rangle$, $\langle 3, 17 \rangle$
P_2	T_3	$\langle 0, 23 \rangle$
	T_4	$\langle 1, 14 \rangle$, $\langle 4, 18 \rangle$

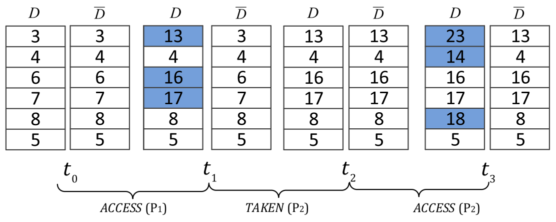


Fig. 1. Running example for Naive Snapshot (NS).

- Read constraint: *Clients should be able to read the latest data items.*
- Update constraint: *Any data item in the snapshot should not be overwritten. In other words, the snapshot must be read-only.*

An in-memory consistent snapshot algorithm for update-intensive applications should fulfill the following requirements.

- *Consistent and Full Snapshots*. Inconsistent snapshots are intolerable. In other words, snapshots should be a consistent view of the data which contains a particular time-point data instead of a duration data. Furthermore, since we do not consider applications such as incremental backups, full snapshots that materialize all the application data states are indispensable.
- *Lock-free and Copy-Optimized*. Locking and synchronous copy operations are the main causes of performance loss(i.e., latency increase) [2]. Therefore, lock-free and copy-optimized snapshot algorithms are more desirable.
- *Low Latency and No Latency Spikes*. Latency spikes (i.e., periodic sharp surges in latency) lead to system quiescing, which degrades user experience.
- *Small Memory Footprint*. The snapshot algorithms should incur low overhead and memory to support large-scale update-intensive applications.

2.2 Model and Framework

We model the in-memory dataset (a.k.a., table) D as a row array. Each row contains multiple data items, and the size of each row is a tunable parameters. To simplify illustration, we assume only one item per row in the running examples throughout this paper.

Interface 1 shows the snapshot algorithm framework. We assume that the framework runs with two kinds of threads: the *client* thread and the *snapshotter* thread. The *snapshotter* thread is responsible for taking snapshots periodically. **Trigger()** is periodically invoked to check if the previous snapshot process has completed. If yes, then generating a new version snapshot interval, and each interval consists of two phases, i.e., the *taken phase* (**TakeSnapshot**()) and the *access phase* (**TraverseSnapshot()**). **TraverseSnapshot()** is invoked to traverse or query the generated snapshot. With the same time, the *client* thread continuously performs large amounts of **Read()** and **Write()** function requests, as in update-intensive applications. **Read()/Write()** only invokes during *access* phase. If *taken* phase is very long, which means an obvious latency spike for the *client* thread.

Interface 1. Snapshot Algorithm Framework

```
1: Snapshotter::Trigger();
2: Snapshotter::TakeSnapshot();
3: Snapshotter::TraverseSnapshot();
4: Client::Read(index);
5: Client::Write(index,newValue);
```

3 In-Memory Consistent Snapshot Algorithms

In this section, we review the mainstream snapshot algorithms for in-memory database systems. Based on in-depth analysis on the drawbacks of existing algorithms, we propose modifications and improvements for existing snapshot algorithms.

3.1 Representative Snapshot Algorithms

This section describes four mainstream snapshot algorithms (NS, COU, ZZ, PP) proposed by academia.

3.1.1 Naive Snapshot

Naive snapshot (NS) [17], [18] takes a snapshot of data state D during the *taken* phase by copying the whole dataset synchronously, meanwhile blocking the *client* thread. Once the snapshot $\overline{D}$ is taken in memory, the *client* thread is then resumed. In *access* phase, the *snapshotter* thread can access or traverse the snapshot data $\overline{D}$ asynchronously. Clients can read the latest data from D during the entire process. Example 1 shows a running example of NS.

Example 1 (Naive Snapshot). Assume an initial dataset $D = \{3, 4, 6, 7, 8, 5\}$ at time t_0. At this time, $\overline{D} = D$, $\overline{D}$ maintains the latest snapshot of D. Table 1 shows the client data streams to be updated, and Fig. 1 shows the data state. In the first period P_1 ($t_0 \rightarrow t_1$), there are two updates T_1 and T_2, and each update is represented by an $\langle index, value \rangle$ pair. At the end of P_1 (time t_1), the updated data state $D = \{13, 4, 16, 17, 8, 5\}$. We need to take a snapshot of D at time t_1. First, the client is blocked during the snapshot taken phase ($t_1 \rightarrow t_2$), and the snapshotter thread duplicates and bulk copies all the data D to snapshot $\overline{D}$, i.e., *memcpy*($\overline{D}$, D). Next, in the access phase ($t_2 \rightarrow t_3$), the client thread writes T_3 and T_4 to D, and the snapshotter thread can access the snapshot from $\overline{D}$ asynchronously. Note that the client can read the latest data from D during the entire period, but the write will be blocked during taken phase.

3.1.2 Copy-on-Update and Fork

Copy-on-Update (COU) [20] utilizes an auxiliary data structure $\overline{D}$ to shadow copy D and employs a bit array $\overline{D}_b$ to

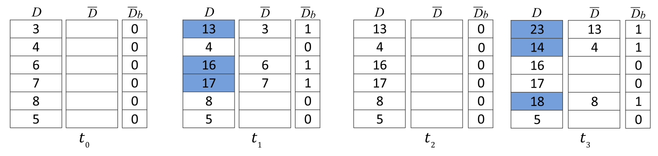


Fig. 2. Running example for Copy on Update (COU).

record the row update states of D. Any client writing on a row of D for the first time will lead to a shadow row copy to the corresponding row of $\overline{D}$ and a setting to the corresponding bit of $\overline{D}_b$ to indicate the state before the row update. In COU, the *snapshotter* thread can utilize the $\overline{D}_b$ to access the snapshot. We refer readers to [20] for more details. Note that COU has many variants [2], [19], and here, we refer to the latency-spike-free implementation in [20]. The fork function [22] is also a system-level COU variant, the main difference is that fork organizes the data in page, not in a row layout; meanwhile the bit array is changed to page table. The detail analysis about fork can refer to Section 5.2.3. Many popular industrial systems such as Redis [26] and Hyper [8] exploit fork to take snapshots.

Example 2 (Copy-on-Update). As with Example 1, $D = \{3, 4, 6, 7, 8, 5\}$ at time t_0, and $\overline{D}$ is empty. $\overline{D}_b$ are all zeros which means all data are not updated and copied. To take a snapshot at time t_0, the incoming updates T_1 and T_2 should not overwrite the snapshot data D. In the first period P_1 $(t_0 \rightarrow t_1)$, when updating data D, the snapshot data should be first copied to $\overline{D}$. COU copies the snapshot data to the shadow data $\overline{D}$ and sets the bit flag $\overline{D}_b$ to keep track of the "old" data. e.g., T_1 updates $D[0]$, then the old value 3 should be copied to $\overline{D}[0]$, and $\overline{D}_b$ should be set to mark that $\overline{D}[0]$ is the old data. Therefore, the snapshotter thread could traverse the snapshot of timepoint t_0 according to $\overline{D}_b$. It is noteworthy that the snapshotter thread and client thread have contention access to the data, therefore mutex lock is needed. To take snapshot of timepoint at t_1, COU will reset all the bit in $\overline{D}_b$ in the taken phase $(t_1 \rightarrow t_2)$. $(t_2 \rightarrow t_3)$ is similar to $(t_0 \rightarrow t_1)$. The snapshotter thread is able to access the snapshot data through the bit flag, as shown in Fig. 2. However, there must maintain exclusive locks between the client thread and the snapshotter thread, which leads to performance loss.

3.1.3 *Zigzag*

Zigzag (ZZ) [21] retains untouched snapshot data with the help of two bit arrays. ZZ employs one shadow copy $\overline{D}$ (of the same size as D) and two auxiliary bit arrays $\overline{D}_{br}$ and $\overline{D}_{bw}$. For a row i, $\overline{D}_{bw}[i]$ is responsible for indicating which copy ($D[i]$ or $\overline{D}[i]$) the client should write to. Note that ZZ keep $\overline{D}_{bw}[i]$ unchanged during the *access* phase, thus, $\neg\overline{D}_{bw}$ indicates the snapshot data since this copy cannot be written by the client. $\overline{D}_{br}[i]$ indicates which copy is the latest version and from which the client should read. At timepoint t_1, to get the snapshot, the data being marked in $\overline{D}_{br}$ should not be overwritten. Therefore, on the next *taken* phase, ZZ just need to let $\overline{D}_{bw} = \neg\overline{D}_{br}$, which ensures that the next *access* phase has an untouched snapshot. In short, the data updated by the client is tracked through $\overline{D}_{bw}$, and the "untouched" snapshot data is stored in the set indicated by $\neg\overline{D}_{bw}$.

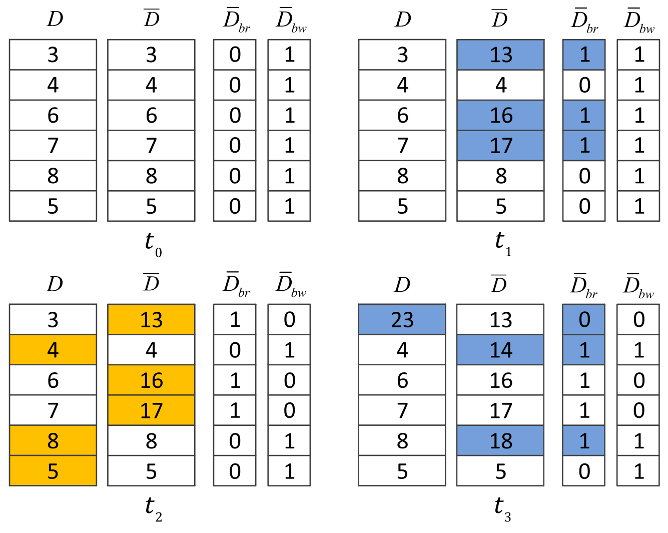


Fig. 3. Running example for Zigzag (ZZ).

Example 3 (Zigzag). Assume Example 3 is in the same setting as Example 1. At the initial time t_0, $D = \overline{D} = \{3, 4, 6, 7, 8, 5\}$. $\overline{D}_{br}$ are all zeros, and $\overline{D}_{bw}$ are all ones. During the first period, updates T_1 and T_2 are written to $\overline{D}$, and D has the time-in-point snapshot data of time t_0. For each write, Zigzag sets $\overline{D}_{br}[i] = 1$, which means the latest version of row i is in $\overline{D}[i]$. At the end of P_1 (at time t_1), the latest data can be tracked by the $\overline{D}_{br}$ array. To take the snapshot, we should ensure that updates T_3 and T_4 are not written to the data tracked by $\overline{D}_{br}$ (marked in yellow in the Fig. 3). On the contrary, T_3 and T_4 should be written according to $\neg\overline{D}_{br}$. So, we set $\overline{D}_{bw} = \neg\overline{D}_{br}$. During the second period P_2, we can access the snapshot with the help of $\neg\overline{D}_{bw}$.

3.1.4 *Ping-Pong*

Ping-Pong (PP) [21] is proposed to eliminate the latency spikes completely. It leverages one copy $\overline{D}_u$ to collect updates and the other copy $\overline{D}_d$ to record the incremental snapshot. During each period, the *client* thread reads from D and writes to both D and $\overline{D}_u$. The *snapshotter* thread can asynchronously access the incremental snapshot $\overline{D}_d$. At the end of each period, all the updated data for constructing the upcoming incremental snapshot are held in $\overline{D}_u$, PP attains an immediate swap by exchanging the pointers $\overline{D}_u$ and $\overline{D}_d$, which can ensure almost no latency spikes.

Example 4 (Ping-Pong). Assume it is in the same setting as Example 1. At time t_0, $D = \overline{D}_d = \{3, 4, 6, 7, 8, 5\}$. During P_1, we execute T_1 and T_2 to D and $\overline{D}_u$. Meanwhile, $\overline{D}_d$ holds the snapshot of time t_0. At the end of P_1 (at time t_1), the data in $\overline{D}_u$ hold the incremental data regarding the updated data during P_1. In the taken phase $(t_1 \rightarrow t_2)$, $\overline{D}_u$ and $\overline{D}_d$ are exchanged to freeze the snapshot data (marked in yellow in Fig. 4). During P_2, we can write data to $\overline{D}_u$ and access the incremental snapshot in $\overline{D}_d$.

3.2 Improved Snapshot Algorithms

So far, we have discussed four state-of arts algorithms. Due to *memcpy*, it is inevitable for NS to cause an obvious latency spike because of . COU can reduce the latency spike by introducing the copy-on-write technique, but mutex lock

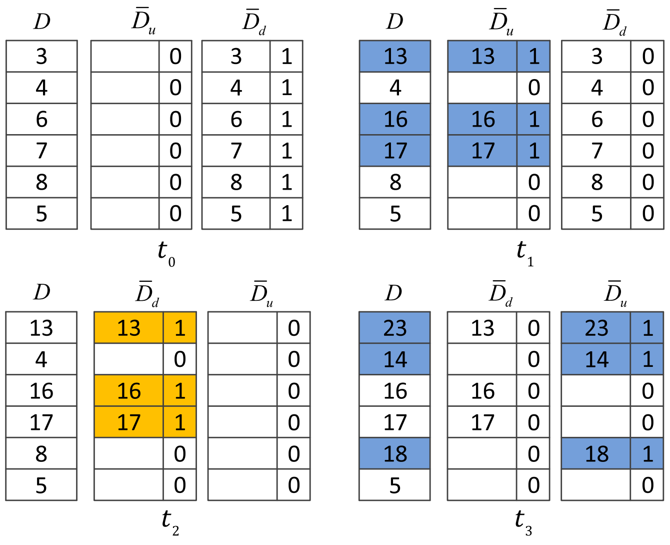


Fig. 4. Running example for Ping-Pong (PP).

which increases the average latency is indispensable. ZZ can eliminate the mutex lock by applying two extra bit arrays, however, ZZ is only suitable for small dataset. PP seems perfect, but the data need to be updated by each **write()** twice (D and $\overline{D}_u$). The above mentioned algorithms will be validated in the evaluation parts. To achieve low latency, zero latency spikes, high throughput and small time complexity simultaneously, two lightweight improvements, namely Hourglass and Piggyback, are proposed according to existing snapshot algorithms.

3.2.1 Hourglass

One intuitive improvement over the above snapshot algorithms is the combination of Zigzag (bit array marking) and Ping-Pong (pointers swapping) to avoid latency spikes while at the same time retaining a small memory footprint. We call this improvement Hourglass (HG). It maintains dataset D and a shadow copy $\overline{D}$, which are accessed by pointers "pU" and "pD", respectively, as in Ping-Pong. D and $\overline{D}$ are accompanied by bit arrays $\overline{D}_{b1}$ and $\overline{D}_{b2}$, where $\overline{D}_{b1}[i]$ and $\overline{D}_{b2}[i]$ indicate whether the row in $D[i]$ has been updated during the current period. Hourglass utilizes these bit arrays to record the incremental data updates in the current period. Pointer swapping happens at the end of the period. An additional bit array $\overline{D}_{br}$ is set up to indicate the locations (either in D or in $\overline{D}$) from which the *client* thread can read the latest rows. A zero for the bit $\overline{D}_{br}[i]$ indicates that the latest data locate in $D[i]$, and a value of one indicates they are located in $\overline{D}[i]$. The following example illustrates how Hourglass works during two successive snapshots.

Example 5 (Hourglass). As shown in Fig. 5a, assume that at time t_0, $D = \overline{D} = \{3,4,6,7,8,5\}$. $\overline{D}_{b1}$ and $\overline{D}_{b2}$ are initialized with zeros and ones, respectively. $\overline{D}_{br}$ is initialized with ones. During P_1, when an update occurs on row i, $\overline{D}_{b1}[i]$ is set to 1, and $\overline{D}_{br}[i]$ is set to 0. $\overline{D}$ will be isolated with the client thread, so that it can be accessed by the snapshotter thread in the lock-free manner. At the end of P_1 period, all bits in $\overline{D}_{b2}$ are reset to zeros. Fig. 5b shows the changes to the memory rows at the end of period P_1. The updated rows are marked in blue shadow. Next, in the snapshot taken phase, the pointers of pU and pD between D and $\overline{D}$ are swapped as in Fig. 5c. Then, in the access phase, the snapshotter thread begins to access the incremental snapshot data from D. Only those rows pointed by pD where the corresponding bits are set to ones are included in the snapshot. In our example, $D[0]$, $D[2]$, and $D[3]$ (marked in yellow shadow Fig. 5c) are accessed. During access phase($t_2 \rightarrow t_3$), the client thread resumes executing the updates. The state at the end of P_2 is shown in Fig. 5d. We can read the latest data with the help of $\overline{D}_{br}$.

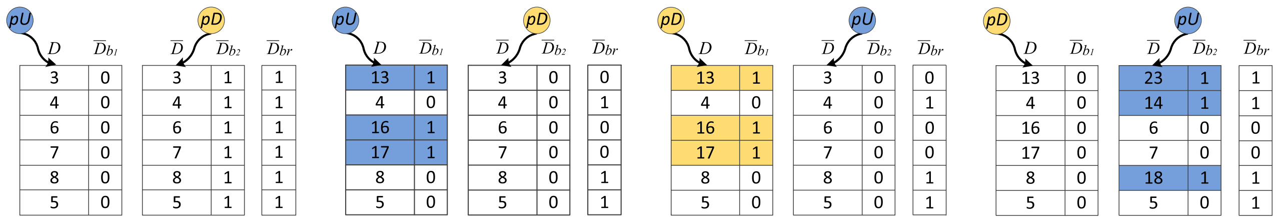
Algorithm 2 describes the main idea of Hourglass.

Algorithm 2. Hourglass

```
Input:
  DataSet $D, \overline{D} \leftarrow$ initial data source
  DataSet $*pU, *pD$
  BitArray $\overline{D}_{b1} \leftarrow \{0,0,\dots,0\}$
  BitArray $\overline{D}_{b2} \leftarrow \{1,1,\dots,1\}$
  BitArray $*pU_b, *pD_b$
  BitArray $\overline{D}_{br} \leftarrow \{1,1,\dots,1\}$
  $D \leftarrow pU$, $\overline{D}_{b1} \leftarrow pU_b$, $\overline{D} \leftarrow pD$, $\overline{D}_{b2} \leftarrow pD_b$
  $RowNum \leftarrow |D|$
1: function CLIENT::WRITE($index, newValue$)
2:   $pU_b[index] \leftarrow 1$
3:   $pU[index] \leftarrow newValue$
4:   $\overline{D}_{br}[index] \leftarrow (pU == \&D)?0 : 1$
5: end function
1: function CLIENT::READ($index$)
2:   return $(\overline{D}_{br}[index] == 0)?D[index] : \overline{D}[index]$
3: end function
-----
1: function SNAPSHOTTER::TRIGGER
2:   if previous snapshot done then
3:     $TakeSnapshot()$
4:     $TraverseSnapshot()$
5:   end if
6: end function
1: function SNAPSHOTTER::TAKESNAPSHOT
2:   lock Client
3:   swap($pU, pD$)
4:   swap($pU_b, pD_b$)
5:   unlock Client
6: end function
1: function SNAPSHOTTER::TRAVERSESNAPSHOT
2:   for $i = 1$ to $RowNum$ do
3:     if $pD_b[i] = 1$ then
4:       $pD_b[i] \leftarrow 0$
5:       write $pD[i]$
6:     else
7:       copy-from last snapshot
8:     end if
9:   end for
10: end function
```

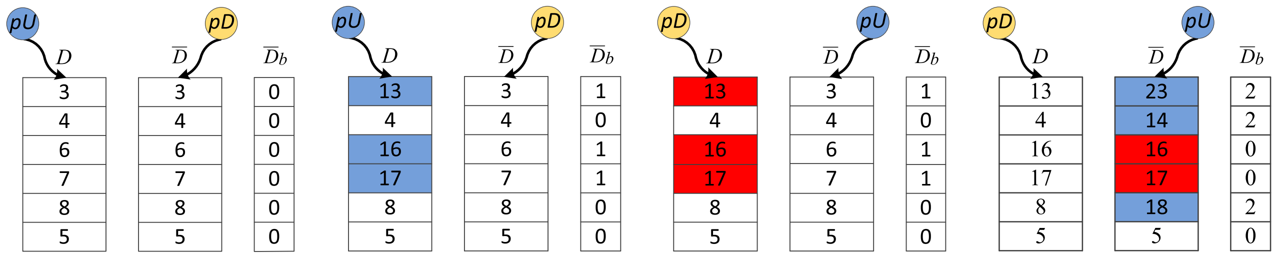
3.2.2 Piggyback

Although pointer swapping (in Ping-Pong and Hourglass) eliminates latency spikes, it is only applicable for incremental snapshots. To enable full snapshots with the pointer swapping technique, we propose another improvement called Piggyback (PB). The idea is to introduce a background thread copying the fresh data from pD to pU. In



(a) t_0, D and $\overline{D}$ with identical dataset (b) t_1, the client thread performs updates (c) t_2, the state after pointers swapping (d) t_3, the state after backing up the incremental updated data

Fig. 5. Running example for Hourglass (HG).



(a) t_0, D and $\overline{D}$ with identical dataset (b) t_1, the client thread performs updates (c) t_2, the state after pointers swapping (d) t_3, the state after backing up the full snapshot

Fig. 6. Running example for Piggyback (PB).

TABLE 2
Comparison of Algorithms in Different Metrics; "(*)" Represents the Drawback

Algorithms	Average Latency	Latency Spike	Snapshot Time Complexity	Max Throughput	Is Full Snapshot	Max Memory Footprint
Naive Snapshot [17], [18]	low	(*) high	(*) O(n)	low	yes	2×
Copy-on-Update [2], [19], [20]	(*) high	(*) middle	(*) O(n)	middle	yes	2×
Fork [22]	low	(*) middle	(*) O(n)	high	yes	2×
Zigzag [21]	middle	(*) middle	(*) O(n)	middle	yes	2×
Ping-Pong [21]	(*) high	almost none	O(1)	low	no	(*) 3×
Hourglass	low	almost none	O(1)	high	no	2×
Piggyback	low	almost none	O(1)	high	yes	2×

other words, the data pointed by pU will always be the latest at the end of each period, i.e., pD holds the full snapshot data after pointer swapping.

To support piggyback copies, the Piggyback algorithm leverages two techniques. *(i)* Piggyback maintains a two-bit array $\overline{D}_b$. The value of $\overline{D}_b[i]$ is one of three states from $\{0, 1, 2\}$, which indicates from which dataset the *client* thread should read. When $\overline{D}_b[i] = 0$, the *client* thread can read row i from either array because it means that $D[i] = \overline{D}[i]$. When $\overline{D}_b[i] = 1$, the *client* thread should read row i from $D[i]$. When $\overline{D}_b[i] = 2$, the *client* thread should read row i from $\overline{D}[i]$. *(ii)* Piggyback defines another function **Snapshotter::WriteToOnline()** which is called in **Snapshotter::Trigger()** as in Algorithm 3. **Snapshotter::WriteToOnline()** ensures the data pointed by pU will always be the latest at the end of each period, so that **Snapshotter::TraverseSnapshot()** can access the full snapshot in pD.

Example 6 (Piggyback). Initially, pU and pD are pointed to D and $\overline{D}$, respectively. The bit array $\overline{D}_b$ is set to zeros as shown in Fig. 6a. Fig. 6b shows the situation at time t_1. The client thread updates rows $D[0]$, $D[2]$, and $D[3]$ (blue shadow) during the first period. The corresponding two-bit elements in $\overline{D}_b$ are then set to ones by the client thread at the same time, which ensures that the client thread always reads the latest data based on the information in $\overline{D}_b$. Concurrently, $\overline{D}$ has the full snapshot data of time t_0. At the beginning of P_2, pointers pU and pD are exchanged. A full snapshot about time t_1 is held in this copy in D and can be accessed. Meanwhile, $\overline{D}$ can be updated by the client thread. Note that there may be older rows in $\overline{D}$ in the P_3 period. For instance, $\overline{D}[0]$, $\overline{D}[2]$ and $\overline{D}[3]$ (red shadow) are out-of-date rows (Fig. 6c). To make $\overline{D}$ maintains the latest data as the next period snapshot data, Piggyback performs a piggyback copy of these rows from D to $\overline{D}$ in this period together with the client's normal updates on rows $\overline{D}[0]$, $\overline{D}[1]$ and $\overline{D}[4]$ (blue shadow). Hence, at the end of P_2, all the rows in $\overline{D}$ are updated to the latest state as shown in Fig. 6d.

3.3 Comparison of Snapshot Algorithms

Table 2 compares the advantages and drawbacks of these four snapshot algorithms. Although fork is a variant of COU, we list it separately since it is the standard method in many industrial IMDBs. In theory, Piggyback, our modification over Zigzag and Ping-Pong, outperforms the rest in all metrics.

Note that the $2\times$ memory consumptions of HG and PB are only for the abstract array model (static memory allocation). Their memory footprints can be further reduced in the production environment thanks to the dynamic memory allocation technique (see Section 5.4).

Algorithm 3. Piggyback

```
Input:
  DataSet $D, \overline{D} \leftarrow initial\ data\ source$
  DataSet $*pU, *pD$
  $D \leftarrow pU, \overline{D} \leftarrow pD$
  FlagArray $\overline{D}_b \leftarrow \{0, 0, \dots, 0\}$
  $RowNum \leftarrow |D|$
1: function CLIENT::WRITE($index, newValue$)
2:   $pU[index] \leftarrow newValue$
3:   $\overline{D}_b[index] \leftarrow (*pU \neq D)?2:1$
4: end function
1: function CLIENT::READ($index$)
2:   return $(\overline{D}_b[index] \neq 2)?D[index] : \overline{D}[index]$
3: end function
---
1: function SNAPSHOTTER::TRIGGER
2:   if previous snapshot done then
3:     $TakeSnapshot()$
4:     $WriteToOnline()$
5:     $TraverseSnapshot()$
6:   end if
7: end function
1: function SNAPSHOTTER::TAKESNAPSHOT
2:   lock Client
3:   swap($pU, pD$)
4:   unlock Client
5: end function
1: function SNAPSHOTTER::WRITETOONLINE
2:   $bit = (pD == \&D)?1:2$
3:   for $k = 1$ to $RowNum$ do
4:     if $\overline{D}_b[k] = bit$ then
5:       $\overline{D}_b[k] = 0$
6:       $pU[k] \leftarrow pD[k]$
7:     end if
8:   end for
9: end function
1: function SNAPSHOTTER::TRAVERSESNAPSHOT
2:   for $k = 1$ to $RowNum$ do
3:     Dump-All $pD[k]$
4:   end for
5: end function
```

4 VIRTUAL SNAPSHOT

This section discusses the recent work [25] in designing virtual snapshot algorithms that are independent of a physically consistent state. We also modify our Hourglass and Piggyback algorithms to meet this new requirement.

4.1 Physical Snapshot Algorithms with Physically Consistent State

The above snapshot algorithms from academia and industry rely on a physically consistent state. That is, the in-memory data must remain consistent at a point in time once **Trigger ()** is invoked. Such a situation has been discussed frequently in applications, such as frequent consistent application [2], [21], actor-oriented database systems [27], and partition-based single thread running database, i.e., H-Store [28], Redis, Hyper [8], etc. However, for a broader application situation (e.g., concurrent transaction based database), to maintain such a physically consistent state, system quiescing is inevitable until all active transactions have been committed. This is the cause of latency spikes [25].

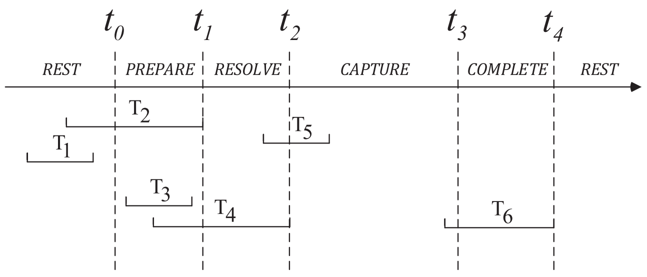


Fig. 7. Running example for CALC.

4.2 Virtual Snapshot Algorithms without Physically Consistent State

4.2.1 CALC

To avoid blocking transactions *during the trigger*, one recent pioneering work called CALC [25] proposes the concept of virtual consistent snapshot, for which snapshot is not captured at the point in trigger time but delayed until all active transactions are committed. CALC is a concurrent variant of COU. In CALC, each cycle (period) is divided into 5 phases. Similar to COU, CALC maintains two copies of data D and $\overline{D}$, as well as a bit array $\overline{D}_b$. CALC can obtain a virtual consistent view of the snapshot data by carefully performing COU during specific phases. We interpret the idea through the following example.

Example 7 (CALC). As shown in Fig. 7, the trigger is invoked at time t_0. The time before t_0 is the rest phase. At time t_1, all the transactions started in the rest phase are committed. The time interval $t_0 \rightarrow t_1$ is called the prepare phase. At time t_2, all the transactions started in the prepare phase are committed, and the corresponding time interval $t_1 \rightarrow t_2$ is labeled the resolve phase. The snapshot is traversed or dumped during t_2 to t_3, which is called the capture phase. At time t_4, all the transactions started in the capture phase are committed. The time interval $t_3 \rightarrow t_4$ is labeled as the complete phase.

For transactions (T_1, T_2) started during the rest or the complete phase, CALC only needs to update D. For transactions (T_3, T_4, T_5, T_6) started during the prepare, the resolve or the capture phase, CALC performs the COU strategy. Finally, a virtual consistent view snapshot is generated at time point t_1. The virtual consistent view of the snapshot data should contain T_1, T_2, T_3, and we can start accessing the view of data after t_2.

4.2.2 vHG and vPB

Although our improved snapshot algorithms Hourglass and Piggyback are primarily designed to be dependent on a physically consistent state, we find such a dependency can be easily eliminated. We call the new versions of Hourglass and Piggyback vHG and vPB, respectively.

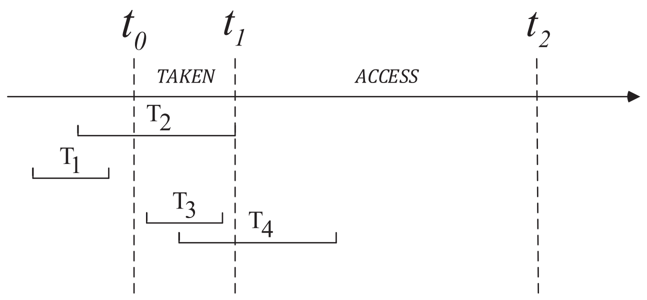


Fig. 8. Running example for vHG.

We describe the main idea of vHG as follows. The trick here is that once the trigger function is invoked, the pointers are swapped immediately. The new transactions (i.e., those started after the trigger) should update the data pointed by pU while the active transactions (i.e., those uncommitted when the trigger is invoked) will keep updating the data pointed by pD. In other words, pointer swapping does not influence the writing strategies of active transactions. The access operation of the *snapshotter* thread should wait until all active transactions are committed. Note that vPB shares the similar idea with vHG.

Example 8 (vHG). As shown in Fig. 8, the structure of vHG is the same as HG. The main difference lies in the trigger function. At time t_0 when the trigger is invoked, the pointers of pU and pD are swapped immediately. The transactions started before t_0 (T_1, T_2) are updated to pU (i.e., D) regardless of the pointer swapping. In contrast, the transactions started after t_0 (T_3, T_4) are updated after swapping pU (i.e., $\overline{D}$). Once T_1 and T_2 are committed at time t_1, the data in pD hold the virtual consistent view of data. Then, the snapshotter thread invokes the TraverseSnapshot() function.

Algorithm 4 shows the pseudo code of vHG, which shares the same framework with Algorithm 2. The difference lies in the fact that data should be updated to the same dataset within a transaction lifecycle, and snapshot should be postponed by detecting and waiting for the end of all active transactions rather than being performed immediately after the trigger is invoked. In this way, incoming transactions cannot be blocked as shown in line 4 of **Snapshotter::Trigger()** function.

5 Experimental Studies

This section comprehensively evaluates the performance of various snapshot algorithms from the previous section. First, a thorough benchmark study on latency, latency spike and snapshot overhead is presented in Section 5.2. In addition, the performance of virtual snapshot algorithms is evaluated in Section 5.3. Then, two Redis variants are implemented by integrating HG and PB, respectively, to study the scalability of all algorithms in real-world IMDB systems (Section 5.4).

5.1 Infrastructure

All the experiments are conducted on HP ProLiant DL380p Gen8, which is equipped with two E5-2620 v2 @ 2.10 GHz CPUs and 256 GB main memory. CentOS 7.3 X86_64 operating system with Linux kernel 3.10.0 and GCC 4.8 was installed.

Algorithm 4. vHG

```
1: function Client::TransactionExecution($txn$)
2:   if $pU$ equals to $D$ then
3:     for $index, newvalue$ in $txn$ do
4:       $D_b[index]$ = 1
5:       $D[index]$ = newvalue
6:     end for
7:   else
8:     for $index, newvalue$ in $txn$ do
9:       $\overline{D}_b[index]$ = 1
10:      $\overline{D}[index]$ = newvalue
11:    end for
12:  end if
13: end function
1: function Snapshotter::Trigger
2:   if previous snapshot done then
3:     $TakeSnapshot()$
4:     $Detect\_and\_Waiting()$
5:     $TraverseSnapshot()$
6:   end if
7: end function
```

5.2 Benchmark Study of Physical Snapshot

This part of experiments evaluates snapshot algorithms with synthetic update-intensive workloads.

5.2.1 Setups

DataSet. We benchmark all snapshot algorithms in checkpoint applications to reveal performance and follow the setups in [21]. For example, assume that the size of a row is 64 B, each row contains 8 fields, and each field is 8 bytes in size. Then, D has 16,000,000 rows, approximately 1 GB. For each experiment, we monitor 5 successive checkpoints. All algorithms perform a full checkpoint for fair comparison. For incremental snapshot algorithms(PP and HG), a full checkpoint is obtained by merging the incremental dumped data with the last snapshot. This can be achieved by using the *Copy* and *Merge* proposed by [21]. *Merge* is more efficient than *Copy* in terms of memory maintenance cost. Therefore, for Line 7 of Algorithm 2 of Snapshotter:: TraverseSnapshot(), we apply *Merge* to construct a new full checkpoint.

Workload. To carefully control the update frequency, the single *client* thread runs in a tick-by-tick (a.k.a. time slice) way [21], and we divide each tick into two stages. One is the *update* stage in which uf times of updates should be accomplished. The time duration of the update stage is defined as the *tick latency*. Tick latency will be used as one of our evaluation metrics. The remaining duration of a tick is regarded as the *idle* stage, which aims to idle the *client* thread until the start of the next tick to guarantee a constant tick duration of 10 ms. We can control the update frequency by adjusting the proportion of the two stages. To simulate a heavy updating workload, all the synthetic data are pre-generated and kept in memory. The update-intensive client is simulated by a Zipfian [29] or Uniform distribution random generator. It generates a stream of updated data (in the

TABLE 3
Parameters of Synthetic Workloads

Parameters	Setting
Dataset size	1 GB, 2 GB, **4 GB**, 8 GB, 16 GB
Row size	**64B**, 128B, 256B, 512B, 1024B
Update frequency	16k, **32k**, 64k, 128k, 256k
Access distribution	**uniform**, zipf (0.1, 0.3, 0.5, 0.7, 0.9)

form of ⟨*row_index*, *value*⟩), which will be consumed by the *client* thread. Note that zipfian generator ensures only a small portion of data to be "hot", i.e., frequently updated.

Parameters. Table 3 summarizes the parameters of the synthetic workloads. Four tunable parameters, *row size*, *dataset size*, *uf* and *access distribution*, can notably affect the performance of the algorithms. The default parameters are all in bold font. Experiments are conducted to quantify the impacts of these four parameters in the following.

Metrics. Here, we will demonstrate some details about those metrics.

- *Tick latency distribution* records the time-serials of the latency in update stage.
- *Average tick latency* is the average consuming time of each tick in update stage.
- *Average tick latency spike* is the taken phase time of the *snapshotter* thread (**TakeSnapshot()**).
- *Checkpoint Overhead* records the cost time of dumping phase.

5.2.2 Performance

Latency Distribution. Fig. 9 plots the latency traces with default parameters (16 GB dataset, data row size of 64 B, uf = 32 K/tick and following the uniform distribution). Only the latency traces between the 5,000th and 15,000th ticks are

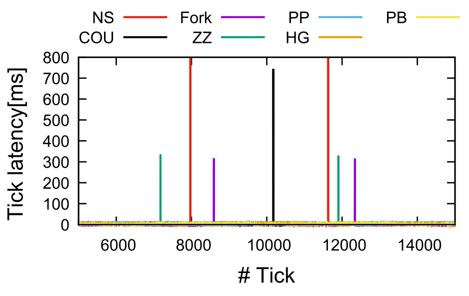
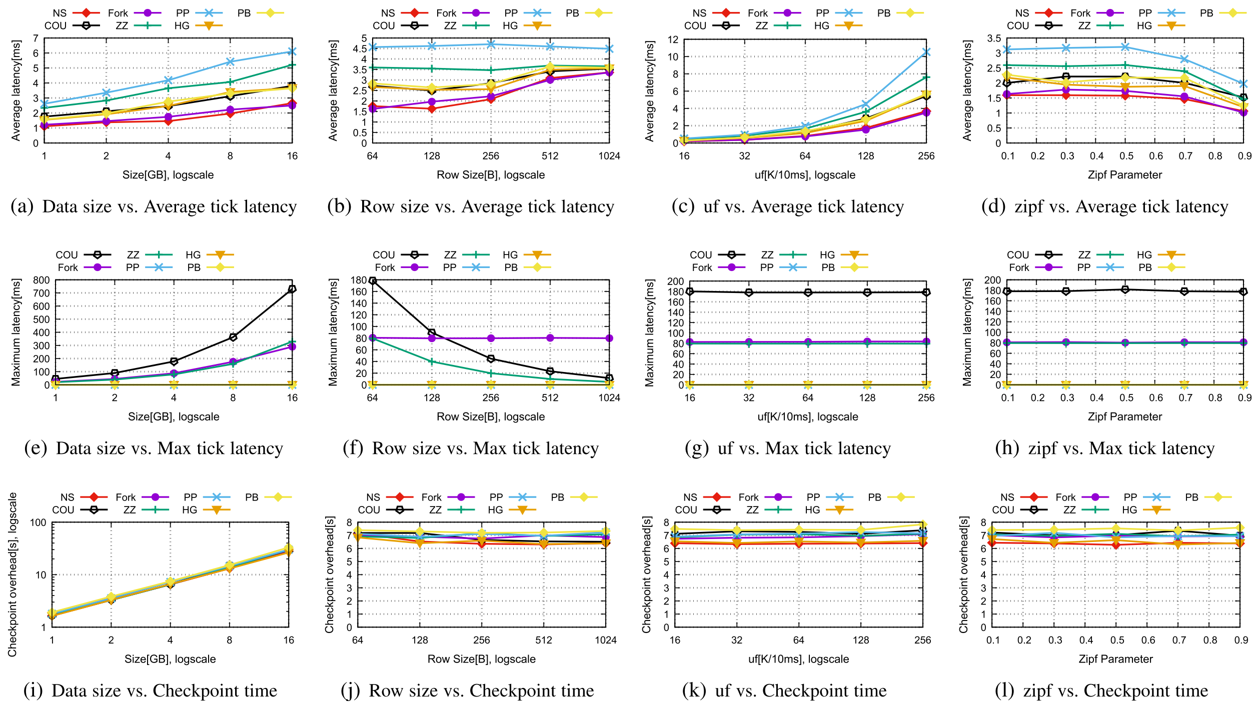


Fig. 9. Tick latency distribution (***uf*** = 32K).

plotted, which include a complete checkpoint period. Other fragments of the traces show a similar pattern. The latency spike usually appears at the beginning of a checkpoint, at which time the *snapshotter* enters the snapshot taken phase. On the one hand, we find that NS, COU, Fork and ZZ all show notable latency spikes, especially NS. The dramatic latency spikes of NS can cause trouble in practical applications. On the other hand, Pingpong, Hourglass and Piggyback all remain stable during the whole time.

Average Tick Latency. Fig. 10a shows the average latency on datasets of different sizes. It is significant that the average latencies of all algorithms exhibit a similar increasing trend and the average tick latency increases with the size of the data set. It takes more time to access larger databases. Fig. 10c shows the average latency with the increase of update frequency. We observe that the average latencies of all algorithms increase with the update frequency. The reason is that more time will be required to process more update. Fig. 10b shows the average latency on datasets of different row sizes. It is observed that the row size almost has no impact on average tick latency with ZZ and PP, but the latency increases for NS, COU, HG and PB. Fig. 10d shows the average tick latency with the increase of zipf



(a) Data size vs. Average tick latency (b) Row size vs. Average tick latency (c) uf vs. Average tick latency (d) zipf vs. Average tick latency

(e) Data size vs. Max tick latency (f) Row size vs. Max tick latency (g) uf vs. Max tick latency (h) zipf vs. Max tick latency

(i) Data size vs. Checkpoint time (j) Row size vs. Checkpoint time (k) uf vs. Checkpoint time (l) zipf vs. Checkpoint time

Fig. 10. Evaluations on synthetic update-intensive workloads.

TABLE 4
Cache Misses for Each Algorithms

	NS	COU	Fork	ZZ	PP	HG	PB
cache misses	711,270,200	1,226,185,063	648,628,216	**1,517,131,653**	1,367,856,967	864,672,521	933,946,167
average tick latency[ms]	1.475	2.455	1.745	3.646	4.183	2.468	2.758

distribution's parameter. As is shown, the average latencies of all algorithms decrease with the zipf's parameter. The phenomenon of Figs. 10b and 10d can be explained by cache misses. The cache misses of each algorithms are measured under default parameters in Table 4.

The first row of Fig. 10 illustrates that NS has the shortest average latency *because the normal read and update show no interference by additional copy or bit checking operations.* COU has a long latency because row locking (there are synchronization locks on rows to be updated between the client and the *snapshotter*)and row duplicating. We can find that, because of the interleave write schema, ZZ has the largest cache misses (Table 4), which leads to a relatively high latency. Different with [21], we notice that PP may generate a large latency, because it exploits the redundant update mode for the client, that is, the *client* thread of PP has to update both D and $\overline{D}_u$ during each operation. The experiments in [21] only considered one of the writes, while we take the redundant writes into account. The same result can be found in [25]. Fork exhibit a similar performance in average latency as NS. The latency of PB, HG is relatively small. Compared with COU, HG and PB only need an extra bit operation instead of the costly row replication. Compared with ZZ, the cache misses of HG and PB are relatively low.

Maximum Tick Latency. The taken phase time of the *snapshotter* dominates the maximum latency of the *client* thread. Fig. 10e shows the maximum latency with the increase of dataset size. Note that we do not display the results of NS because NS is of several orders of magnitude larger than the other algorithms. We can observe that the maximum latency of COU, fork and ZZ algorithms increases with the increase of data size, but the maximum latency of PP, HG and PB algorithms is not affected by the size of data sets, which means COU, fork, ZZ are only suitable for small datasets. Fig. 10f shows the impact of row size to maximum latency. We find that with the increase of row size, the max latency of COU and ZZ deceases. That is because of the larger the row size, the smaller of the bit array length, which is closely related with the max latency. Fork is not influenced by the size of row, because fork works on account of the size of system level memory page size, which is always 4 KB. Fig. 10g further shows the impact of uf on the maximum latency. We find that all curves remain horizontal, and the maximum latency is not influenced by *uf*. Fig. 10h shows the impact of zipf on the maximum latency. We find that the maximum latency is not influenced by the distribution.

According to the second row of Fig. 10, it can be concluded that the maximum latency of PP, HG, and PB is very small. The excellent performance of PP, HG and PB is due to the pointer swapping technique. In sum, comparing latency distribution, average latency and maximum latency, our improved algorithm HG and PB exhibit better performance than other algorithms.

Checkpoint Overhead. Checkpoint overhead is the traverse and dump overhead of TraverseSnapshot(). Fig. 10i shows the trend of overhead on varying dataset sizes. The checkpointing overheads of all the algorithms increase linearly, and there is little difference between their overheads. The overheads are primarily dominated by the dataset size which is written to the external memory. Figs. 10j, 10k and 10l respectively show the checkpoint overhead on different row sizes, *uf* and zipf parameter. The overheads of all the algorithms almost remain constant, because the dataset size is fixed.

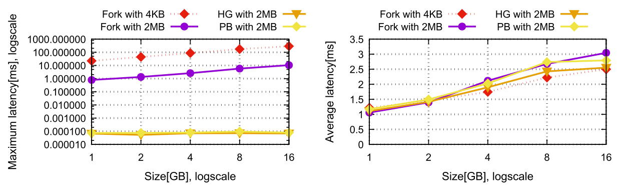
Summary. Our benchmark evaluations on the synthetic workload reveal the following findings.

- For applications whose only concern is the backend performance, the snapshot algorithms should have a lower average latency. It is applicable for NS, Fork, HG and PB all are applicable (see first row of Fig. 10).
- For interaction-intensive applications (i.e., frequent updates), latency spikes should be included to assess the snapshot algorithms. PP, HG and PB outperform the others in terms of the value of spikes (i.e., maximum latency), while NS performs the worst.
- NS, Fork, COU, ZZ and PP are fit for specific applications (i.e., they perform well either on latency or throughout). PB and HG trade off latency, throughput and scalability, and they are fit for a wider range of applications.
- The latency spike of PB and HG are not affected by the data size (see Fig. 10e). In general, PB and HG are more scalable than the other algorithms including fork.
- Fork outperforms NS, COU, ZZ and PP in terms of both average tick latency and maximum tick latency (see Figs. 9 and 10e); in addition, fork has a simple engineering implementation. Therefore, fork is adopted in several industrial IMDBs such as Redis, Hyper, etc.

5.2.3 More Discussion on Fork?

As stated in [30], although it is an OS kernel function, fork is still very time consuming, because it needs to copy the whole page table entries (PTEs) from the parent process to the child process synchronously. If the dataset is large-scale, fork will have a large maximum latency spike. The size of page table entry is influenced by the page size. For example, given 200 MB data, for regular small page (4 KB), it has 51,200 pages and 51,200 page table entries; for huge page (2 MB), it just has 100 pages and 100 page table entries. Copying 100 entries is much faster than copying 512,000 entries. Therefore, maximum latency can be decreased simply by running fork with huge page.

However, every coin has two sides. Huge page may increase the average latency, since larger page leads to



(a) Effect of huge page on maximum tick latency (b) Effect of huge page on average tick latency

Fig. 11. Effects on huge page.

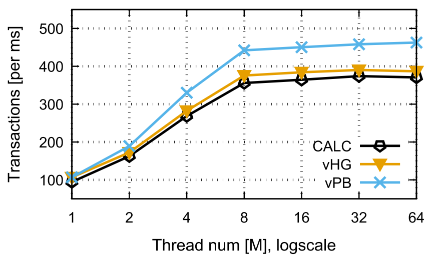


Fig. 12. Thread num *versus* transaction throughput.

larger copy time when copy-on-write occurs. Actually, the redis official document [30] mentioned that the administrator should turn off the huge page parameter when boot the Linux kernel. Besides, Oracle [31], MySQL TokuDB [32], VoltDB [33], NuoDB [34], MongoDB [35], couchbase [36] also suggest to turn off the huge page.

In order to validate that huge pages will increase the copy-on-write latency, a micro evaluation is conducted. Fig. 11 shows the performance comparison of Fork, HG, and PB with data size ranging from 1 to 16 GB, and the results show the limitations of huge pages. On the one hand, compared with regular pages, the huge page version of fork has a relatively small maximum latency as it will be (Fig. 11a), but it is still larger than HG and PB. On the other hand, huge pages will inevitably lead to a higher latency because of the page copy (Fig. 11b). In summary, the huge page is not suitable for fork, especially for the database environments.

Technically, we can organize the hot data with the regular small page (4 KB), and cold data with the huge page (2 MB). However, how to identify the real workload of the hot/cold data is beyond the scope of this paper, it should be improved in the future work.

What's more, the most recent work [16] needs to be mentioned. It invents a $vm\_snapshot$ system call within a custom Linux kernel. Fork targets to the whole virtual memory space. *vm_snapshot* is a fine-grained style "fork", and it just takes snapshot for the data you need and only need to copy a smaller page table entry, which means a better performance for some specific scenarios. Nevertheless, for the database checkpoint scenario, the system needs to snapshot the whole memory space. Once exploiting $vm\_snapshot$ in the checkpoint applications, it must snapshot the whole data and copy the all page table entries , and the performance will be similar to fork. The performance of **vm_snapshot** is not evaluated in this paper due to the space reason and moreover, the source code of [16] is not open.

5.3 Benchmark Study of Virtual Snapshot

The above experiments were conducted under the update intensive physical snapshot scenario. In this section, we present a comparison of CALC, vHG and vPB under the virtual snapshot scenario. The transaction concurrent control method used here is the general Strict two-phase Locking Protocol (S2PL) [37] as in [25]. Note that any other concurrent control methods (e.g., MVCC [38], OCC, etc.) are also applicable as long as they are under the same concurrency control protocol.

Fig. 12 compares the throughput of CALC, vHG, and vPB on a 100 MB dataset. Since vHG and vPB do not require row copy operations, they have greater workload capacity than CALC for all the multi-thread cases. Interestingly, we observe that the throughput tends to be stable when the thread number is larger than 8. Even if the number of threads in the system continues to increase, the performance will not always improve. This phenomenon can be explained by the heavy lock contentions among threads. The result is similar to that of the DBx1000 project [38].

5.4 Performance in Industrial IMDB System

Redis is a popular In-Memory NoSQL system and it utilizes fork() to persist data [39]. To generate the persistent image (a.k.a. RDB file) in the background, Redis has to invoke the system call *fork*() to spawn a child process to execute snapshot and dumping work. From the above benchmark study, we see that fork() indeed performs better than mainstream snapshot algorithms including NS, COU, ZZ and PP in terms of average latency and maximum latency. However, we also suspect that fork() will incur dramatic latency on large datasets, which limits the scalability of Redis. In fact, database administrators usually restrain the data size of a running Redis instance in practice [40]. In this performance study, we aim to harness proper snapshot algorithms to improve the scalability of snapshots in Redis.

5.4.1 Snapshot Algorithm Selection

Fork has weak scalability due to its O(n) time complexity. It is posted in the official website [39] that "*Fork() can be time consuming if the dataset is large, and as a result, Redis may stop serving clients for milliseconds or even one second if the dataset is large and the CPU performance not great*".

To optimize the scalability of Redis, an option is to replace fork with snapshot algorithms of O(1) complexity. Here, we implement two Redis variants Redis-HG and Redis-PB using the HG and PB algorithms, respectively. Both variants are a single-process with double-thread (*client* and *snapshotter*). We do not choose ZZ, for it is only suitable for small datasets; during the taken phase, ZZ needs to operate all the bit flags. Traversing all the keys is time consuming and is almost equal to executing the "keys *" directive. PP is also excluded due to the three copies of the memory footprint. In addition, the Redis architecture is unfit for integrating the PP algorithm.

Note that, for the real-world database systems, the row data is not stored in an array model, like in Fig. 5. For $\forall i$, we can organize the data $D[i], \overline{D}_{b1}[i], \overline{D}[i], \overline{D}_{b2}[i], \overline{D}_{br}[i]$ into a group, called *snap_row*. All the snap_rows are indexed by a global hash index in redis. To save memory (see Section 3.3), we introduce the *garbage collection* and *dynamic memory allocation* techniques.

TABLE 5
Parameters of YCSB Workloads

Parameters	Setting
Loading thread	256
Distribution	Zipfian
Operation count	4M
Update proportion	**0.1**, 0.2, 0.3, 0.4, 0.5
Record count	1M, 2M, 4M, **8M**, 16M

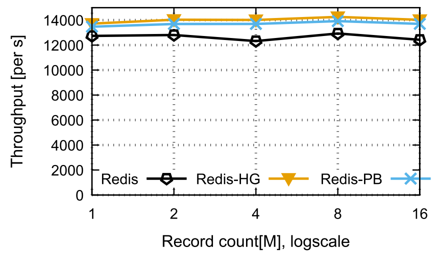


Fig. 13. Redis: YCSB record count versus throughput.

5.4.2 Setups

We chose Yahoo! Cloud Serving Benchmark (YCSB) [23] to validate the practical performance of our prototype. A workload of YCSB is a dataset plus a set of read and write operations. The dataset is loaded into the database and then consumed by those operations. *The system is evaluated from the heavily-updating workload to the intensive-reading workload,* which accords with our aim to evaluate performance for frequent consistent checkpointing. Table 5 shows the detailed workload setups, the default parameters are in bold.

The Redis configuration file *"redis.conf"* contains a number of directives. We use directive *"save 10 1"* to configure Redis to automatically dump the dataset to disk every 10 seconds if there is at least one change in the dataset. Following the redis official recommendation, we turn off the linux huge page mode.

5.4.3 Performance

The metrics evaluated in this part include throughput, maximum latency and dump overhead. The throughput is about how many operations the redis system can deal per second. Note that redis is a single thread model to handle read and update, and the maximum latency reveals the approximate time of the taken phase, while the dump overhead is about the time to generate the RDB file.

Throughput. Fig. 13 illustrates the change of throughput as the record count grows. The trends are consistent with those shown in Fig. 10a. We make two observations. *(i)* The throughput of all the algorithms is insensitive to the dataset size. *(ii)* Redis-HG and Redis-PB have similar throughput performance to the default Redis; although Redis-HG and Redis-PB can avoid locks between data updating and dumping, they need additional checking through the hash table for each read/write operation. Therefore, the throughout improvement is marginal.

Maximum Latency. Fig. 14 plots the maximum latency with the record count from 1 million to 16 million, approximately up to 35 GB (with update proportion = 0.1). The default Redis incurs a dramatic increase in maximum

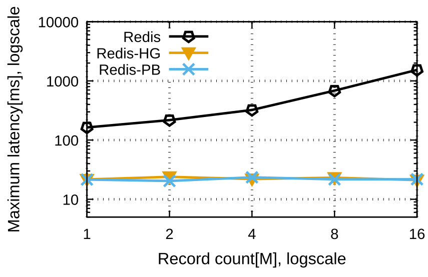


Fig. 14. Redis: YCSB record count *versus* maximum latency.

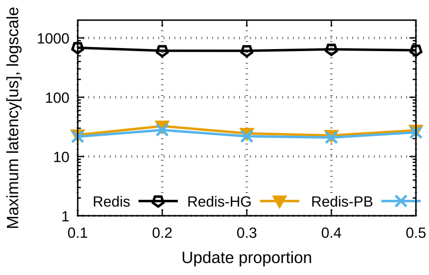


Fig. 15. Redis: Update proportion *versus* maximum latency.

latency. This result is consistent with the Redis document for which the maximum latency becomes huge because of the invocation of fork(). Redis-PB and Redis-HG have similar maximum latencies, and both remain stable with the growth of the record count. This can be explained by the pointer swapping technique employed in the snapshot taken phase, which only needs to be almost constant and incur a small cost. We expect that the maximum latency of official Redis implementation will grow rapidly with the record count until eventually quiescing the system, which leads to weak scalability. The trends are consistent with Fig. 10e. Conversely, Redis-HG and Redis-PB can scale to larger datasets than the default Redis.

Effects of Updates on Latency. Fig. 15 shows the maximum latency with fixed record counts of 8 million and a varying proportion of updates from 0.1 to 0.5. As shown, the maximum latency keeps stable with more updates, we conclude that Redis, Redis-HG, Redis-PB are insensitive to update-intensive applications.

Maximum Memory Footprint. Since the default Redis persistence strategy depends on forking a child process to dump the snapshot, the additional application dataset size of the memory footprint is inevitable. Although at the beginning of a fork, the parent and child processes share a single data region in memory, the actual size of the memory consumed will increase with frequent data updates (i.e., page duplication). That is, the memory footprint depends on the workload. In the worst case (update intensive), fork will lead to a memory spike (almost double the memory footprint) [41]. As explained in Redis FAQ [42], the fork may fail when the Redis memory size is larger than half of system memory. Although the fork failure can be avoided by setting parameter *overcommit_memory* to 1, there still exists the risk of being killed by the OS' OOM killer. Based on experience, the case where the redis instance is larger than half of the local physical memory is dangerous.

In principle, Redis-HG and Redis-PB need a memory footprint that is twice the size. To reduce memory usage, we leverage the *dynamic memory allocation* and the *garbage*

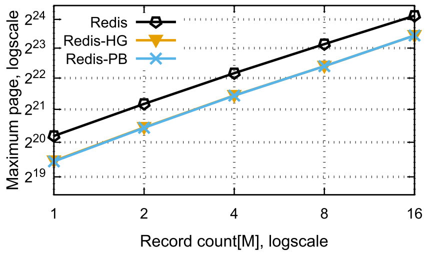


Fig. 16. Redis: YCSB record count *versus* maximum memory cost.

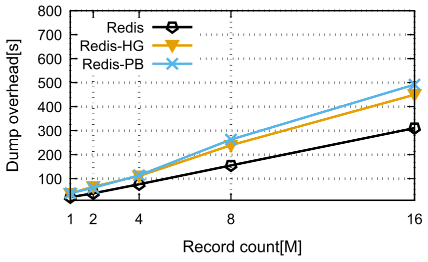


Fig. 17. Redis: YCSB record count *versus* dump overhead.

collection technologies. Once a value has been dumped to the disk and the value is not up-to-date, the corresponding value portion should be identified as garbage that can be destroyed and reused by the system now or later. Fig. 16 shows the comparison of the maximum memory cost. All comparisons are linear to the dataset size. The memory cost of Redis-HG and Redis-PB are similar and far smaller than the original Redis.

Dump Overhead. Fig. 17 presents the dump (i.e., RDB) overhead. The results are similar to those in Fig. 10i. The scale of the record count ranges from 1 to 16 million. The dump overhead grows linearly with the dataset size. For small datasets, the dump overheads of Redis-HG and Redis-PB are close to that of the original Redis. The gap increases slowly with the increase in dataset size. Note that the two variants need additional state checking to determine the appropriate copy of data for dumping while the default Redis' child thread only needs to traverse the hash table to flush all the key-value pairs. Fortunately, the double-thread design effectively separates the updating and dumping tasks and induces only a slightly longer background dumping period. Furthermore, the overhead gap can be reduced by leveraging high-speed disks and large memory buffers.

Summary. Redis with the built-in fork() function is unscalable (see Fig. 14). By replacing the default fork with HG and PB, the two variants, Redis-HG and Redis-PB, exhibit better scalability.

6 Conclusions

In this paper, we analyze, compare, and evaluate representative in-memory consistent snapshot algorithms from both academia and industry. Through comprehensive benchmark experiments, we observe that the simple fork() function often outperforms the state-of-the-arts in terms of latency and throughput. However, no in-memory snapshot algorithm achieves low latency, high throughput, small time complexity, and no latency spikes at the same time; however, these requirements are essential for update-intensive in-memory applications. We propose two lightweight improvements over existing snapshot algorithms, which demonstrate better tradeoff among latency, throughput, complexity and scalability. We implement our improvements on Redis, a popular in-memory database system. Extensive evaluations show that the improved algorithms are more scalable than the built-in fork() function. We have made the implementations of all algorithms and evaluations publicly available to facilitate reproducible comparisons and further investigation of snapshot algorithms.

7 Future Work

This work discusses leveraging snapshot algorithms to perform checkpoints. As described in Section 1, consistent snapshots are not only used for consistent checkpoints but are also employed in HTAP systems [6], [7], [8], [9], [10], [11], [12], [13], [14], i.e., Hyper, HANA and SwingDB. Although there are many studies about concurrency control protocols for OLTP systems [38], [43], [44], [45], [46], [47], few works address HTAP's concurrency control; thus, we plan to build a prototype based on snapshot concurrency control to fill this gap in the future. In addition, apart from exploit our proposed algorithm to improve the latency spike of Redis, recognizing hot/cold data and managing the cold data with 2 MB huge page also can be used to solve the latency spike problem. It could be an another future work.

Acknowledgments

The authors would like to thank Wenbo Lang, Phillip Saenz and the anonymous reviewers. Lei Chen is supported by the Hong Kong RGC GRF Project 16214716 , National Grand Fundamental Research 973 Program of China under Grant 2014CB340303, the National Science Foundation of China (NSFC) under Grant No. 61729201, Science and Technology Planning Project of Guangdong Province, China, No. 2015B010110006, Webank Collaboration Research Project, and Microsoft Research Asia Collaborative Research Grant. Guoren Wang is supported by the NSFC (Grant No. U1401256, 61732003, 61332006 and 61729201). Gang Wu is supported by the NSFC (Grant No. 61872072). Ye Yuan is supported by the NSFC (Grant No. 61572119 and 61622202) and the Fundamental Research Funds for the Central Universities (Grant No. N150402005).

References

[1] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, Jul. 2015.

[2] M. A. V. Salles, T. Cao, B. Sowell, A. J. Demers, J. Gehrke, C. Koch, and W. M. White, "An evaluation of checkpoint recovery for massively multiplayer online games," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 1258–1269, 2009.

[3] More details on today's outage. 2010. [Online]. Available: https://www.facebook.com/note.php?note_id=431441338919

[4] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1243–1254.

[5] H. Mühe, A. Kemper, and T. Neumann, "How to efficiently snapshot transactional data: Hardware or software controlled?" in *Proc. 7th Int. Workshop Data Manage. New Hardware*, 2011, pp. 17–26.

[6] F. Özcan, Y. Tian, and P. Tözün, "Hybrid transactional/analytical processing: A survey," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1771–1775. [Online]. Available: http://doi.acm.org/10.1145/3035918.3054784

[7] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (It's time for a complete rewrite)," in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 1150–1160.

[8] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 195–206.

[9] H. Plattner, "A common database approach for OLTP and OLAP using an in-memory column database," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2009, pp. 1–2.

[10] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper, "Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 311–326.

[11] F. Funke, A. Kemper, and T. Neumann, "Benchmarking hybrid OLTP&OLAP database systems," in *Proc. Datenbanksysteme Bus. Technologie Web*, 2011, pp. 390–409.

[12] Q. Meng, X. Zhou, S. Chen, and S. Wang, "SwingDB: An embedded in-memory DBMS enabling instant snapshot sharing," in *Proc. Int. Workshop In-Memory Data Manage. Analytics*, 2016, pp. 134–149.

[13] F. Farber, S. K. Cha, J. Primsch, C. Bornhovd, S. Sigg, and W. Lehner, "SAP HANA database: Data management for modern business applications," *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, 2012.

[14] V. Sikka, F. Farber, A. K. Goel, and W. Lehner, "SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1184–1185, 2013.

[15] D. Šidlauskas, C. S. Jensen, and S. Šaltenis, "A comparison of the use of virtual versus physical snapshots for supporting update-intensive workloads," in *Proc. 8th Int. Workshop Data Manage. New Hardware*, 2012, pp. 1–8.

[16] A. Sharma, F. M. Schuhknecht, and J. Dittrich, "Accelerating analytical processing in MVCC using fine-granular high-frequency virtual snapshotting," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 245–258.

[17] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Recent advances in checkpoint/recovery systems," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, 2006, pp. 282–289.

[18] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *J. Physics: Conf. Series*, vol. 78, pp. 012 022–012 032, 2007.

[19] A.-P. Liedes and A. Wolski, "SIREN: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases," in *Proc. IEEE 22th Int. Conf. Data Eng.*, 2006, pp. 99–99.

[20] T. Cao, *Fault Tolerance for Main-Memory Applications in the Cloud*. Ithaca, NY, USA: Cornell Univ., 2013.

[21] T. Cao, M. A. V. Salles, B. Sowell, Y. Yue, A. J. Demers, J. Gehrke, and W. M. White, "Fast checkpoint recovery algorithms for frequently consistent applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 265–276.

[22] Wikipedia. "Wikipedia of Fork (system call)," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Fork_(system_call)

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[24] L. Li, G. Wang, G. Wu, and Y. Yuan, "Consistent snapshot algorithms for in-memory database systems: Experiments and analysis," in *Proc. 34th IEEE Int. Conf. Data Eng.*, 2018, pp. 1284–1287. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00131

[25] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson, "Low-overhead asynchronous checkpointing in main-memory database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1539–1551.

[26] Antirez. Redis. 2017. [Online]. Available: https://redis.io

[27] P. Bernstein, "Actor-oriented database systems," in *Proc. 34rd IEEE Int. Conf. Data Eng.*, 2018, pp. 13–14.

[28] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al., "H-store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[29] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," *ACM SIGMOD Rec.*, vol. 23, pp. 243–252, 1994.

[30] Antirez. "Redis latency problems troubleshooting," 2017. [Online]. Available: https://redis.io/topics/latency

[31] Oracle. "Disabling transparent hugepages," 2019. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/18/ladbi/disabling-transparent-hugepages.html#GUID-02E9147D-D565-4AF8-B12A-8E6E9F74BEEA

[32] M. TokuDB "Tokudb installation," 2019. [Online]. Available: https://www.percona.com/doc/percona-server/5.6/tokudb/tokudb_installation.html

[33] VoltDB. "Configure memory management," 2019. [Online]. Available: https://docs.voltdb.com/AdminGuide/adminmemmgt.php

[34] NuoDB. "Notes on using transparent huge pages," 2019. [Online]. Available: http://doc.nuodb.com/Latest/Content/Note-About-%20Using-Transparent-Huge-Pages.htm

[35] MongoDB. "Disable transparent huge pages (THP)," 2019. [Online]. Available: https://docs.mongodb.com/v3.2/tutorial/transparent-huge-pages/

[36] Couchbase. "Disabling transparent huge pages (THP)," 2019. [Online]. Available: https://docs.couchbase.com/server/6.0/install/thp-disable.html

[37] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA, USA: Addison-Wesley, 1987.

[38] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proc. VLDB Endowment*, vol. 8, no. 3, pp. 209–220, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol8/p209-yu.pdf

[39] antirez. "Redis persistence," 2017. [Online]. Available: https://redis.io/topics/persistence

[40] Best EC2 setup for redis server. 2017. [Online]. Available: https://stackoverflow.com/questions/11765502/best-ec2-setup-for-redis-server

[41] Redis memory and CPU spikes. 2017. [Online]. Available: https://stackoverflow.com/questions/16384436/redis-memory-and-cpu-spikes

[42] antirez. "Redis faq," 2017. [Online]. Available: https://redis.io/topics/faq

[43] K. Ren, A. Thomson, and D. J. Abadi, "Lightweight locking for main memory database systems," *Proc. VLDB Endowment*, vol. 6, no. 2, pp. 145–156, 2012.

[44] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. ACM SIGOPS 24th Symp. Operating Syst. Principles*, 2013, pp. 18–32. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522713

[45] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas, "TicToc: Time traveling optimistic concurrency control," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1629–1642. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882935

[46] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, "An empirical evaluation of in-memory multi-version concurrency control," *Proc. VLDB Endowment*, vol. 10, no. 7, pp. 781–792, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p781-Wu.pdf

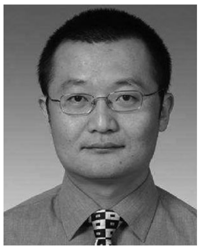
[47] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 21–35. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064015

Liang Li received the BSc degree from the College of Computer Science and Engineering, Northeastern University, China, in 2014. Currently, he is working toward the PhD degree in computer science and engineering at Northeastern University. His main research interests include in-memory database systems, distributed systems, and database performance.

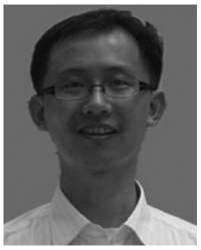
Guoren Wang received the BSc, MSc, and PhD degrees in computer science from Northeastern University, China, in 1988, 1991, and 1996, respectively. Currently, he is a professor with the Department of Computer Science, Beijing Institute of Technology, China. His research interests include XML data management, query processing and optimization, bioinformatics, high-dimensional indexing, parallel database systems, and P2P data management.

Gang Wu received the BS and MS degrees in computer science from Northeastern University, China, in 2000 and 2003, respectively, and the PhD degree in computer science from Tsinghua University, in 2008. He is now an associate professor with the College of Information Science and Engineering, Northeastern University. His research interests include in-memory databases, graph databases, and knowledge graphs.

Ye Yuan received the BS, MS, and PhD degrees in computer science from Northeastern University, China, in 2004, 2007, and 2011, respectively. He is now a professor with the College of Information Science and Engineering, Northeastern University. His research interests include graph databases, probabilistic databases, data privacy-preserving, and cloud computing.



Lei Chen received the BS degree in computer science and engineering from Tianjin University, China, in 1994, the MA degree from the Asian Institute of Technology, Bangkok, Thailand, in 1997, and the PhD degree in computer science from the University of Waterloo, Canada, in 2005. He is currently an associate professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. His research interests include crowdsourcing over social media, social media analysis, probabilistic and uncertain databases, and privacy-preserved data publishing.



Xiang Lian received the BS degree from the Department of Computer Science and Technology, Nanjing University, in 2003, and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong. He is now an assistant professor with the Department of Computer Science, Kent State University. His research interests include probabilistic/uncertain data management, probabilistic RDF graphs, inconsistent probabilistic databases, and streaming time series. He is a member of the IEEE.