

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 open-sourced 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.

- **Hybrid Transactional/Analytical Processing Systems (HTAP):** Hybrid OLTP&OLAP in-memory systems are gaining increasing popularity [3], [4], [5], [6], [7], [8], [9], [10], [11]. 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 [5], SAP HANA [10], [11] and SwingDB [9] have

already applied in-memory snapshot algorithms in Hybrid Transactional/Analytical Processing Systems.

- **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 2.8TB memory data were cleared [12]. Consistent checkpoints are important to avoid long-time system failures and support rapid recovery; in-memory systems such as Hekaton [13] and Hyper [14] 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.

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 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) [15], [16], Copy-on-Update (COU) [2], [17], [18], Zigzag (ZZ) [19] and PingPong (PP) [19]. In addition, the simple fork [20] 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

- *Liang Li, Ye Yuan are with the Department of Computer Science, Northeastern University of China, 100819.
E-mail: {liliang@stunmail,yuanye@ise}.neu.edu.cn*
- *Guoren Wang is with the Department of Computer Science, Beijing Institute of Technology, China, CN, 100081.
E-mail: wanggr-bit@126.com*
- *Gang Wu is with the Department of Computer Science, Northeastern University of China, 100819 and with the Department of State Key Lab. for Novel Software Technology, Nanjing University, P.R. China.
E-mail: wugang@mail.neu.edu.cn*
- *Lei 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*
- *Xiang Lian is with the Department of Computer Science, Kent State University, USA.
E-mail: xlian@kent.edu*

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). Fig. 1 shows illustrative latency traces of five mainstream snapshot algorithms. 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. This is more programming friendly.

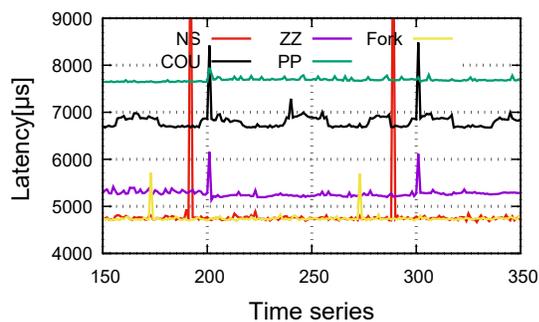


Figure 1. Comparison of State-of-art algorithms

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 GitHub¹. We further integrate the two improved algorithms into Redis and investigate the scalability with the Yahoo! Cloud Serving Benchmark (YCSB) [21]. The implementations and evaluations are also publicly accessible². 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 [22]. 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 [23].

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:*

- **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 must fulfill the following requirements.

- **Consistent and Full Snapshots.** “Dirty” (i.e., inconsistent) snapshots are intolerable. 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 snapshot overhead [2]. Therefore, lock-free and copy-optimized snapshot algorithms are more desirable.

1. <https://github.com/bombehub/FrequentSnapshot>

2. <https://github.com/bombehub/RedisPersistent>

- **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 D as a page array. Each page contains multiple data items, and the size of each page is 4 KB as in typical operating systems. To simplify illustration, we assume only one item per page in the running examples throughout this paper.

Interface 1 shows the snapshot algorithm framework. We assume two kinds of threads: the *client* thread and the *snapshotter* thread. The client thread continuously performs large amounts of **Read()** and **Write()** function requests, as in update-intensive applications. The snapshotter thread is responsible for taking snapshots periodically. **Trigger()** is periodically called to check if the previous snapshot process has completed. If yes, it invokes **TakeSnapshot()**. Then, **TraverseSnapshot()** is invoked to traverse the generated snapshot. Each interval (*a.k.a.*, *period*) consists of two phases, *i.e.*, the *taken phase* and the *access phase*. It should claim that **Trigger()** is always invoked at a physical consistent time point when no transactions are uncommitted.

Interface 1 Snapshot Algorithm Framework.

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

In the rest of this paper, we illustrate the process of representative snapshot algorithms via examples, and we start with an example of Naive Snapshot.

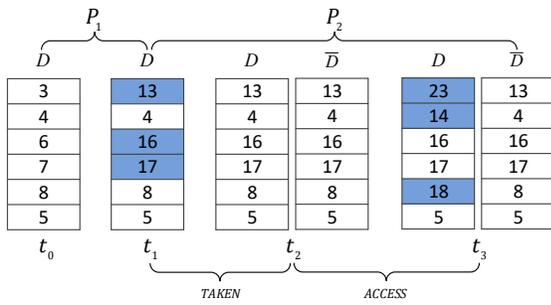


Figure 2. Running example for Naive Snapshot (NS)

Table 1
Transaction Data

Period	Transactions	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	$\langle 1, 14 \rangle, \langle 4, 18 \rangle$

EXAMPLE 1 (Naive Snapshot) Assume an initial dataset $D = \{3, 4, 6, 7, 8, 5\}$ at time t_0 . Table 1 shows the client data streams to be updated, and Fig. 2 shows the data state. We further assume periodic snapshot taking. In the first period P_1 ($t_0 \rightarrow t_1$), there are two transactions T_1 and T_2 , where each update is represented by an $\langle index, value \rangle$ pair. At the end of P_1 (*i.e.*, at 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 \bar{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 \bar{D} . Note that the client can read the latest data from D during the entire period.

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 also propose modifications and improvements of existing snapshot algorithms.

3.1 Representative Snapshot Algorithms

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

3.1.1 Naive Snapshot

Naive snapshot (NS) [15] [16] takes a snapshot of data state D during the taken phase when the client thread is blocked. Once the snapshot \bar{D} is taken in memory, the client thread is then resumed. Meanwhile, the snapshotter thread can access or traverse the snapshot data \bar{D} asynchronously. Clients can read the latest data from D during the entire process. EXAMPLE 1 shows an example.

3.1.2 Copy-on-Update and Fork

Copy on Update (COU) [18] utilizes an auxiliary data structure \bar{D} to shadow copy D and a bit array \bar{D}_b for recording the page update states of D . Any client write on a page of D for the first time leads to a shadow page copy to the corresponding page of \bar{D} and a setting to the corresponding bit of \bar{D}_b to indicate the state before the page update. In COU, the snapshotter thread can utilize the \bar{D}_b to access the snapshot. We refer readers to [18] for more details.

Note that COU has many variants [2] [17], and here, we refer to the latency-spike-free implementation in [18]. The fork function [20] is also a system-level COU variant. Many popular industrial systems such as Redis [24] and Hyper [5] exploit fork to take snapshots.

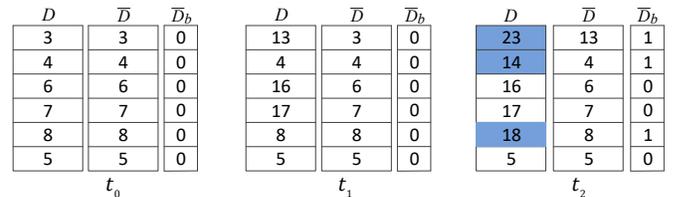


Figure 3. Running example for Copy on Update (COU)

EXAMPLE 2 (Copy-on-Update) As with EXAMPLE 1, dataset D update from $\{3, 4, 6, 7, 8, 5\}$ to $\{13, 4, 16, 17, 8, 5\}$. To take a

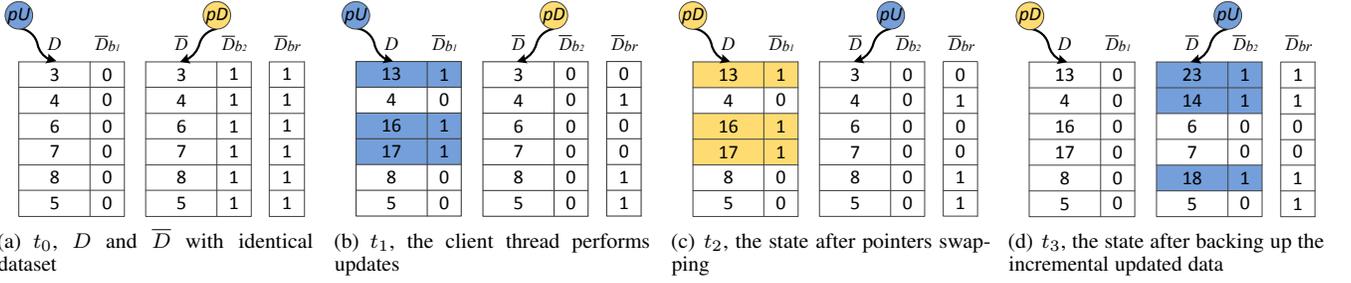


Figure 6. Running example for Hourglass (HG)

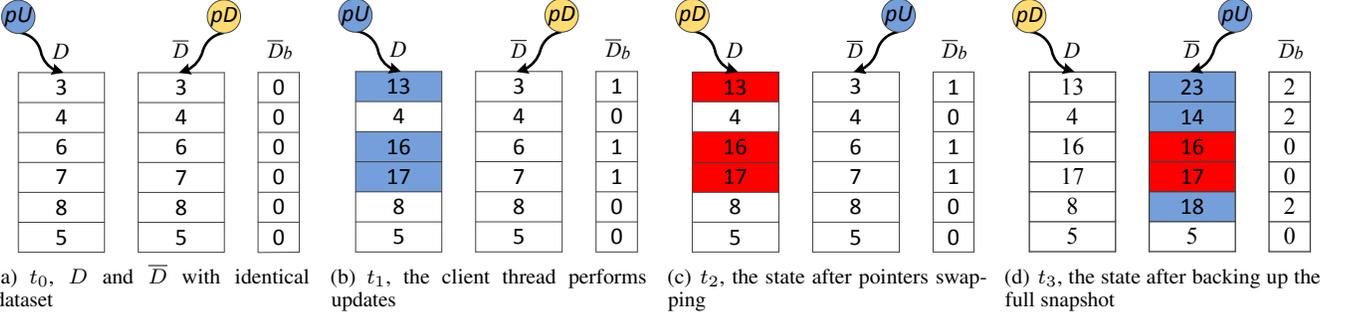


Figure 7. Running example for Piggyback (PB)

is set to 0. \bar{D} will be kept away from the client thread, so that it can be accessed by the snapshotter thread in the lock-free manner. At the same time, once a page j in the dataset \bar{D} has been accessed, the j th position in \bar{D}_{b2} is reset to 0. At the end of this period, all bits in \bar{D}_{b2} are reset to zeros. Fig. 6(b) shows the changes to the memory pages at the end of period P_1 . The updated pages are marked in blue shadow. Next, in the snapshot taken phase, the pointers of pU and pD between D and \bar{D} are swapped as in Fig. 6(c). Then, in the access phase, the snapshotter thread begins to access the incremental snapshot data from D . Only those pages pointed by pD where the corresponding bits are set to zeros are included in the snapshot. In our example, $D[0]$, $D[2]$, and $D[3]$ (marked in yellow shadow Fig. 6(c)) are accessed. During this time, the client thread resumes executing the transactions. The state at the end of P_2 is shown in Fig. 6(d).

Algorithm 2 describes the main idea of Hourglass.

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 copy the out-of-date data from pD to pU . Consequently, 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 \bar{D}_b . The value of $\bar{D}_b[i]$ is one of three states from $\{0, 1, 2\}$, which indicates from which dataset the client thread should read. When $\bar{D}_b[i]=0$, the client thread can read page i from either array because it means that $D[i]=\bar{D}[i]$. When $\bar{D}_b[i]=1$, the client thread should read page i from $D[i]$. When $\bar{D}_b[i]=2$, the client thread should read page i from $\bar{D}[i]$. (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 \bar{D} , respectively. The bit array \bar{D}_b is set to zeros as shown in Fig. 7(a). Fig. 7(b) shows the situation at time t_1 . The client thread updates pages $D[0]$, $D[2]$, and $D[3]$ (blue shadow) during the first period. The corresponding two-bit elements in \bar{D}_b are then set to ones by the client thread at the same time. This ensures that the client thread always reads the latest data based on the information in \bar{D}_b . Concurrently, \bar{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, \bar{D} can be updated by the client thread. Note that there may be dirty pages in \bar{D} in the P_3 period. For instance, $\bar{D}[0]$, $\bar{D}[2]$ and $\bar{D}[3]$ (red shadow) are older pages (Fig. 7(c)). To avoid dirty pages, Piggyback performs a piggyback copy of these pages from D to \bar{D} in this period together with the client's normal updates on pages $\bar{D}[0]$, $\bar{D}[1]$ and $\bar{D}[4]$ (blue shadow). Hence, at the end of P_2 , all the pages in \bar{D} are updated to the latest state as shown in Fig. 7(d).

3.3 Comparison of Snapshot Algorithms

Table 2 compares the advantages and drawbacks of the 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 Sec. 5.4).

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 [15], [16]	low	(*) high	(*) O(n)	low	yes	2×
Copy-on-Update [2], [17], [18]	(*) high	(*) middle	(*) O(n)	middle	yes	2×
Fork [20]	low	(*) middle	(*) O(n)	high	yes	2×
Zigzag [19]	middle	(*) middle	(*) O(n)	middle	yes	2×
Ping-Pong [19]	(*) 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×

Algorithm 2 Hourglass
Input:

DataSet $D, \bar{D} \leftarrow$ initial data source
 DataSet $*pU, *pD$
 BitArray $\bar{D}_{b1} \leftarrow \{0, 0, \dots, 0\}$
 BitArray $\bar{D}_{b2} \leftarrow \{1, 1, \dots, 1\}$
 BitArray $*pU_b, *pD_b$
 BitArray $\bar{D}_{br} \leftarrow \{1, 1, \dots, 1\}$
 $D \leftarrow pU, \bar{D}_{b1} \leftarrow pU_b, \bar{D} \leftarrow pD, \bar{D}_{b2} \leftarrow pD_b$
 $PageNum \leftarrow |D|$

```

1: function CLIENT::WRITE(index, newValue)
2:    $pU_b[index] \leftarrow 1$ 
3:    $pU[index] \leftarrow newValue$ 
4:    $\bar{D}_{br}[index] \leftarrow (pU == \&D)?0 : 1$ 
5: end function
1: function CLIENT::READ(index)
2:   return  $(\bar{D}_{br}[index] == 0)?D[index] : \bar{D}[index]$ 
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::TAKE_SNAPSHOT
2:   lock Client
3:   swap( $pU, pD$ )
4:   swap( $pU_b, pD_b$ )
5:   unlock Client
6: end function
1: function SNAPSHOTTER::TRAVERSE_SNAPSHOT
2:   for  $i = 1$  to  $PageNum$  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

```

Algorithm 3 Piggyback
Input:

DataSet $D, \bar{D} \leftarrow$ initial data source
 DataSet $*pU, *pD$
 $D \leftarrow pU, \bar{D} \leftarrow pD$
 FlagArray $\bar{D}_b \leftarrow \{0, 0, \dots, 0\}$
 $PageNum \leftarrow |D|$

```

1: function CLIENT::WRITE(index, newValue)
2:    $pU[index] \leftarrow newValue$ 
3:    $\bar{D}_b[index] \leftarrow (*pU \neq D)?2 : 1$ 
4: end function
1: function CLIENT::READ(index)
2:   return  $(\bar{D}_b[index] \neq 2)?D[index] : \bar{D}[index]$ 
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::TAKE_SNAPSHOT
2:   lock Client
3:   swap( $pU, pD$ )
4:   unlock Client
5: end function
1: function SNAPSHOTTER::WRITE_TO_ONLINE
2:    $bit = (pD == \&D)?1 : 2$ 
3:   for  $k = 1$  to  $PageNum$  do
4:     if  $\bar{D}_b[k] = bit$  then
5:        $\bar{D}_b[k] = 0$ 
6:        $pU[k] \leftarrow pD[k]$ 
7:     end if
8:   end for
9: end function
1: function SNAPSHOTTER::TRAVERSE_SNAPSHOT
2:   for  $k = 1$  to  $PageNum$  do
3:     Dump-All  $pD[k]$ 
4:   end for
5: end function

```

4 VIRTUAL SNAPSHOT

This section discusses the recent work [23] 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 the trigger function is invoked. Such a situation has been discussed frequently in applications, such as frequent consistent application [2], [19], actor-oriented database systems [25], and partition-based single thread running database, *i.e.*, H-Store [26], Redis, Hyper [5], 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 [23].

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 [23] proposes the concept of virtual consistent snapshot, for which snapshot is not captured at the point in trigger time but delayed until the end of all active transactions. 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 \bar{D} , as well as a bit array \bar{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.

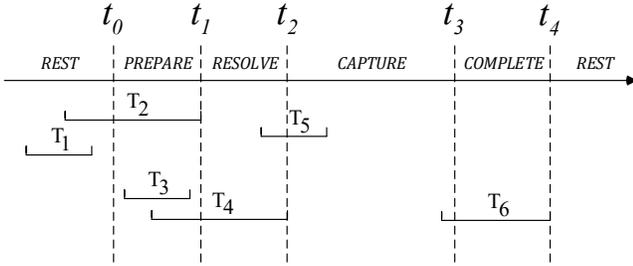


Figure 8. Running example for CALC

EXAMPLE 7 (CALC) As shown in Fig. 8, 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 $t_0 \rightarrow t_1$ is labeled the resolve phase. The snapshot is taken during $t_2 \rightarrow t_3$, 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.

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.

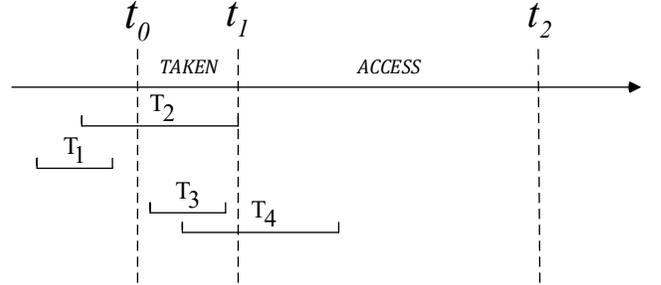


Figure 9. Running example for vHG

EXAMPLE 8 (vHG) As shown in Fig. 9, 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.*, \bar{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. We first present a thorough benchmark study on latency, throughput and snapshot overhead (Sec. 5.2). In addition, we briefly evaluate the performance of virtual snapshot algorithms (Sec. 5.3). Then, we implement two Redis variants by integrating HG and PB, respectively, to study the scalability in real-world IMDB systems (Sec. 5.4).

Algorithm 4 vHG

```

1: function CLIENT::TRANSACTIONEXECUTION(txn)
2:   if pU equals to D then
3:     for index, newvalue in txn do
4:        $\overline{D}_b[index] = 1$ 
5:        $\overline{D}[index] = \text{newvalue}$ 
6:     end for
7:   else
8:     for index, newvalue in txn do
9:        $\overline{D}_b[index] = 1$ 
10:       $\overline{D}[index] = \text{newvalue}$ 
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

```

5.1 Infrastructure

All the experiments are conducted on a server, HP ProLiant DL380p Gen8, which is equipped with two E5-2620 CPUs and 256GB main memory. CentOS 6.5 X86_64 operating system with Linux kernel 2.6.32 and GCC 5.1.0 os installed. Using the micro-benchmark ³, the evaluation environment has the following performance parameters: memory bandwidth = 2.72 GB/s, memory write startup overhead = 37.38 ns, lock overhead = 87.19 ns, and atomic bit check overhead = 0.94 ns.

5.2 Benchmark Study of Physical Snapshot

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

5.2.1 Setups

We benchmark all snapshot algorithms in checkpoint applications to reveal performance and follow the setups in [19]. The update-intensive client is simulated by a Zipfian [27] distribution random generator. It generates a stream of update data (in the form of $\langle \text{page_index}, \text{value} \rangle$), which will be consumed by the client thread. The generator ensures only a small portion of data to be “hot”, *i.e.*, frequently updated. Since the Zipfian distribution parameter α has little impact on the experimental results [19], we set α to 2 by default. To simulate a heavy updating workload, all the synthetic data are pre-generated and kept in a trace file. The trace file is loaded into the main memory before performing updates.

To carefully control the update frequency, the client thread runs in a tick-by-tick (a.k.a. time slice) way [19], and we divide each tick into two stages. One is the *update* stage in which *uf* times updates should be accomplished. The length of the update stage is defined as the tick latency (*latency* for short). Latency will be used as one of the evaluation metrics for performance comparison. 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 consistent tick duration of 100ms. We can control the update frequency by adjusting the proportion of the two stages. For

example, assume a memory page size of 4KB. Each page contains 1024 data items, and the data item is 4 bytes in size. Then, *D* has 1,000,000 pages, approximately 4000MB, and the data are updated at a rate of 0.128% per second. Hence, the update frequency is $\frac{4000MB}{4B} \times 0.128\% = 1280K$ per second, *i.e.*, *uf* = 128K per tick.

The checkpoint interval is defined to be at least 10 seconds. For each experiment, we monitor 10 successive checkpoints.

Table 3 summarizes the parameters of the synthetic workloads. The initial parameter settings are shown in bold. Two tunable parameters, *dataset size* and *uf*, can notably affect the performance of the algorithms. Intuitively, the size of the dataset directly relates to the checkpointing overhead and the time of the snapshotter taken phase. Furthermore, during each tick interval, we perform accurate times of operations, which represents the intensity of the client workload. Apparently, the workload has very strong influence on the latency performance. Hence, we conduct experiments to quantify the impacts of these two parameters in the following.

Table 3
Parameters of synthetic workloads

Parameters	Setting
Checkpoint count	10
Data item size	4B
Memory page size	4KB
Tick length	100ms
Update frequency	16k , 32k, 64k, 128k, 256k
Dataset size	1000MB , 2000MB, 4000MB, 8000MB

5.2.2 Performance

We mainly evaluate the latency (average, distribution and maximum), maximum throughput and checkpoint overhead of different snapshot algorithms.

Average Latency. Fig. 10 shows the average latency with the increase of update frequency (16k, 32k, 64k, 128k, and 256k per tick) on a 1000MB dataset. The average latencies of all algorithms exhibit similar increasing trends. 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 there are synchronization locks on pages to be updated between the client and the snapshotter. Page locking and duplicating increase latency. Unlike [19], we observe that PP incurs a large latency, as PP 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 [19] only consider one of the writes, while we also take the redundant writes into account. The same result can be found in [23]. The latency of PB, HG, Fork and ZZ is relatively small. In the case of PB, HG, and ZZ, they only need an extra bit operation rather than the costly page replication. The slightly larger latency of ZZ is because each update operation may occur at either dataset (*D* or \overline{D}), while the other two methods (HG and PB) have an exclusive dataset for updating during a checkpointing period. The accumulative cost of such interlaced memory addressing cannot be ignored in the case of high updating frequencies. As for Fork, the OS kernel function *fork()* is called to create a copy of the updating process context including the memory space, which is more efficient than the original COU.

Fig. 11 shows the average latency on datasets of different sizes at a fixed updating frequency of 256K. We observe that the data size has almost no impact on average latency.

Referring to Fig. 10 and Fig. 11, we conclude that HG, PB and Fork exhibit similar performance to NS in average latency.

3. http://www.cs.cornell.edu/bigreddata/games/recovery_simulator.php

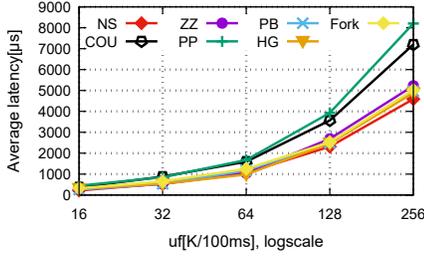


Figure 10. uf vs. Average latency

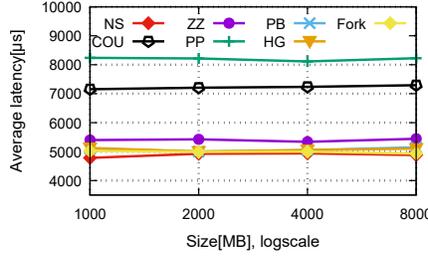


Figure 11. Data size vs. Average latency

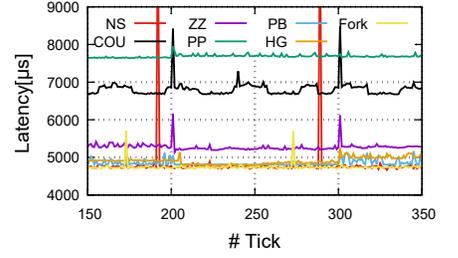


Figure 12. Latency distribution ($uf=256K$)

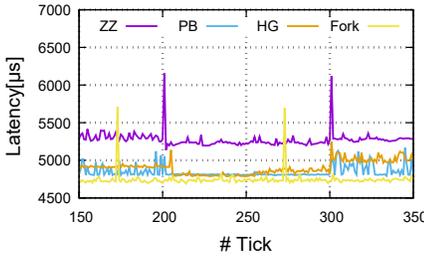


Figure 13. Part of Figure 12 without NS, COU, and PP

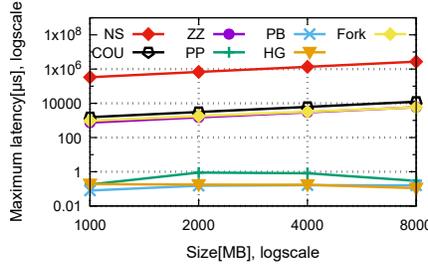


Figure 14. Data size vs. Maximum latency

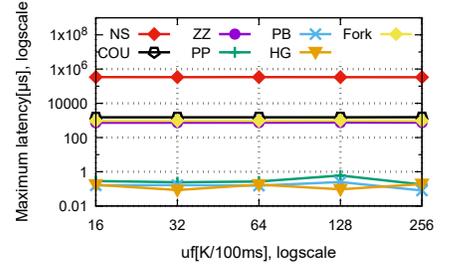


Figure 15. uf vs. Maximum latency

Latency Distribution. Fig. 12 plots the latency traces on a 1000MB dataset with $uf=256K$ per tick. Only the latency traces between the 150th and 350th ticks are plotted, which include a full checkpointing period. Other fragments of the traces show a similar pattern. Fig. 13 zooms in on the details of the differences between Fork, ZZ, HG, and PB.

We observe that the latency of each algorithm is relatively stable. A latency spike usually appears at the beginning of a checkpointing when the snapshotter enters the snapshot taken phase. NS, COU, Fork and ZZ show notable latency spikes because of the $O(n)$ time complexity. The dramatic latency spikes of NS can cause trouble in practical applications.

Maximum Latency. The taken phase time of the snapshotter dominates the maximum latency of the client thread. Fig. 14 shows the maximum latency with the increase in dataset size. The update frequency is set to 256K per tick in this experiment. We can observe that the maximum latency of PP, HG, and PB are several orders of magnitude lower than that of NS, COU, Fork, and ZZ. Moreover, the maximum latency of the latter algorithms becomes steadily larger with the increase in dataset size. The good performance of PP, HG and PB is due to the pointer swapping technique.

Fig. 15 further shows the impact of uf on the maximum latency on a 1000MB dataset. All curves remain horizontal because the maximum latency is only influenced by data size.

In sum, comparing average latency, latency distribution and maximum latency, our improved algorithms, HG and PB, exhibit better performance than other algorithms.

Maximum Throughput. Maximum throughput is one metric to assess the maximum load capacity. Fig. 16 shows the maximum throughput per millisecond on a 8000MB dataset. Unlike the previous experiments, the length of the idle stage in a tick is set to zero. Thus, the client can update as fast as possible in a full-update (no-wait) mode. Further, we turn off the Dump operation to observe the throughput limit.

HG, PB and Fork exhibit better performance in maximum throughput. In fact, the maximum throughput is a comprehensive reflection of the results in Fig. 10 and Fig. 15. For instance, although

NS has the shortest average latency, its latency spike is the most obvious, which leads to a bad maximum throughput.

Checkpointing Overhead. Checkpoint overhead is the traverse and dump overhead of the the snapshotter thread. All algorithms perform a full snapshot for fair comparison. For incremental snapshot algorithms, a full snapshot is obtained by merging the incremental dumped data with the last snapshot. This can be achieved by using the *Copy* and *Merge* proposed by [19]. *Merge* is more efficient than *Copy* in terms of the memory maintenance cost. Therefore, for Algorithm 2 Line 7 of Snapshotter::TraverseSnapshot(), we apply *Merge* to construct a new full snapshot.

Fig. 17 shows the trend in overhead on varying dataset sizes. The update frequency is fixed to 256K per tick. 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 written to the external memory. Fig. 18 shows the overheads with different update frequencies on a dataset of 1000MB. The overheads remain almost constant across all the algorithms because the dataset size is fixed.

5.2.3 Does Fork really good enough?

Fig. 19 shows the performance comparison of Fork, HG, and PB with data size ranging from 1000 MB to 8000 MB. In particular, the results reflect the limitations of Fork. It is no longer comparable with HG and PB in the case of maximum latency. As stated in [28], although it is an OS kernel function, fork is still expensive on most Unix-like systems. The costs are the result of copying the memory page table from the parent process to the child process. Let the memory page size be 4 KB for a Linux x86_64 system whose pointer size is 8 Bytes. Thus, to address a 50 GB memory space, the size of the memory page table should be $\frac{50GB}{4KB} \times 8B \approx 100$ MB. This will result in the allocation and copying of 100 MB memory whenever a checkpoint occurs. Suppose that the memory bandwidth is 2.5 GB/s; the latency spike is about 40 ms, which accords with the results in Fig. 14 and Fig. 19.

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

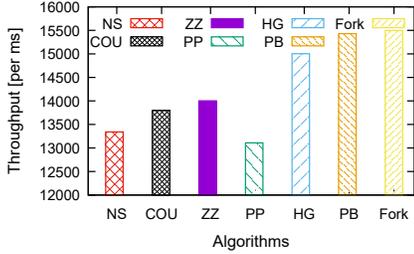


Figure 16. Maximum throughput

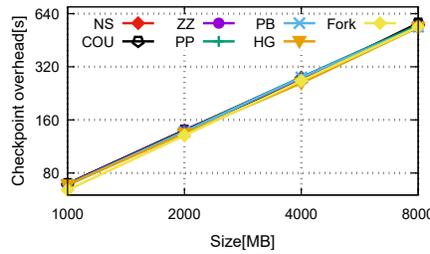


Figure 17. Data size vs. Checkpointing overhead

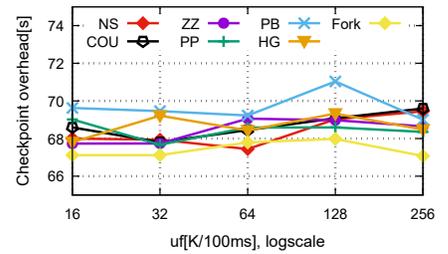


Figure 18. *u* vs. Checkpointing overhead

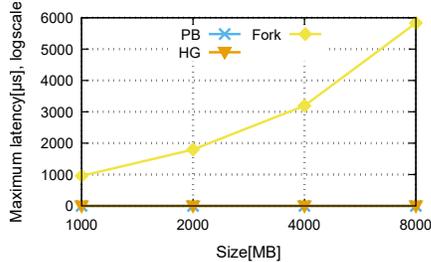


Figure 19. fork performance

- For applications where only backend performance is a concern, the snapshot algorithms should have low average latency. NS, Fork, HG and PB all are applicable (see Fig. 10 and Fig. 11).
- 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 number of spikes and the value of spikes (*i.e.*, maximum latency), while NS performs the worst (see Fig. 12, Fig. 14 and Fig. 15).
- Fork outperforms NS, COU, ZZ and PP in terms of both latency and throughput (see Fig. 12, Fig. 14 and Fig. 16); in addition, fork has a simple engineering implementation; fork is therefore adopted in several industrial IMDBs such as Redis.
- The latency performances of PB and HG are not affected by the data size (see Fig. 14 and Fig. 19). In general, PB and HG are more scalable than the other algorithms including fork.
- 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, which are fit for a wider range of applications.

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) [29] as in [23]. Note that any other concurrent control methods (e.g., MVCC, OCC, etc.) are also applicable as long as they are under the same concurrency control protocol.

Fig. 20 compares the throughput of CALC, vHG, and vPB on a 100MB dataset. Since vHG and vPB do not require page 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 [30].

5.4 Performance in Industrial IMDB System

Redis is a popular In-Memory NoSQL system and it utilizes fork() to persist data [31]. To generate the persistent image (a.k.a. RDB file) in the background, Redis has to invoke the system call *fork()* to start 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 latency and throughput. However, we also suspect that fork() will incur dramatic latency on large datasets, which limits the scalability of Redis. In fact, database users usually restrain the data size of a running Redis instance in practice [32]. 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 [31] 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, as a key-value database, Redis internally maintains a large hash table in the main memory. Alternatively, for each key, we build the array data model as its corresponding value. Although the arrays in the model are physically broken up into fragments, they are still logically traversable through the hash table. To save memory (see Sec. 3.3), we introduce the *garbage collection* and *dynamic memory allocation* techniques.

5.4.2 Dataset

We chose YCSB (Yahoo! Cloud Serving Benchmark) [21] to validate the practical performance of our prototype. A workload of YCSB is a dataset plus a set of read and write operations,

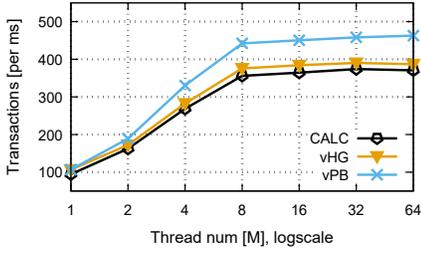


Figure 20. Thread num vs. Transaction throughput

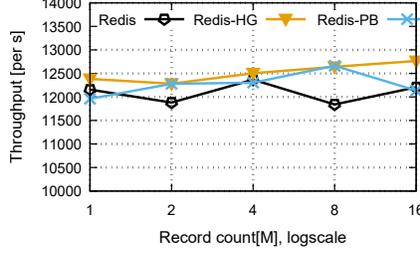


Figure 21. Redis: YCSB record count vs. Throughput

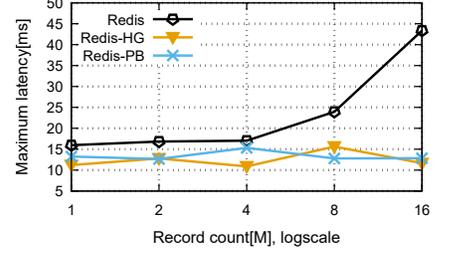


Figure 22. Redis: YCSB record count vs. Maximum latency

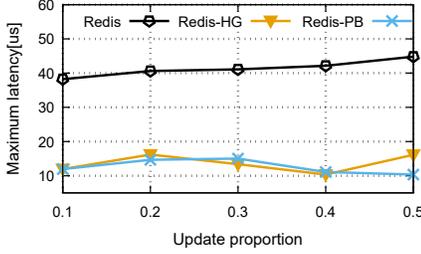


Figure 23. Redis: Update proportion vs. Maximum latency

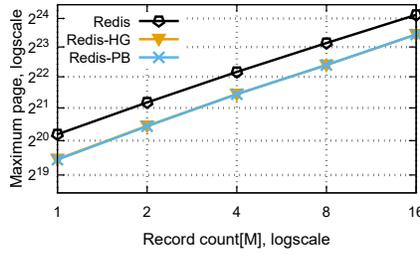


Figure 24. Redis: YCSB record count vs. Maximum memory cost

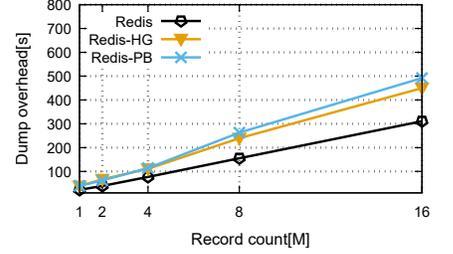


Figure 25. Redis: YCSB record count vs. Dump overhead

i.e., a transaction set. The dataset is loaded into the database and then consumed by the transaction set. YCSB predefines six main workloads. We implement our own workloads based on *Workload A* (update heavy workload) and *Workload B* (read mostly workload), which accord with our aim to evaluate performance for frequent consistent checkpointing. Table 4 shows the detailed workload setups. Those defined by YCSB 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.

Table 4
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

5.4.3 Performance

We mainly evaluate the performance of the two Redis variants in terms of throughput, latency, effect of updates, memory and snapshot overhead.

Throughput. Fig. 21 illustrates the change of throughput as the benchmark record count grows. The trends are consistent with those shown in Fig. 11. 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 throughput improvement is marginal.

Maximum Latency. Fig. 12 compares the differences in latency spikes and Fig. 14 shows the maximum latency with the

increase of the dataset size. Fig. 22 plots the maximum latency with the record count from 1 million to 16 million, approximately up to 50GB (with update proportion = 0.1).

The default Redis incurs a dramatic increase in maximum latency when the record count reaches 8 million. 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. Conversely, Redis-HG and Redis-PB can scale to larger datasets than the default Redis.

Effects of Updates on Latency. Fig. 23 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 of Redis grows with more updates, while those of Redis-HG and Redis-PB still remain relatively stable. We conclude that Redis-HG and Redis-PB are more suitable for 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) [33]. As explained in Redis FAQ [34], 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 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. 24 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.

Checkpointing Overhead. Fig. 25 presents the checkpointing (*i.e.*, RDB) overhead. The results are similar to those in Fig. 17. The scale of the record count ranges from 1 million to 16 million. The checkpointing 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. 22). 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 Sec. 1, consistent snapshots are not only used for consistent checkpoints but are also employed in HTAP systems [3], [4], [5], [6], [7], [8], [9], [10], [11], *i.e.*, Hyper, HANA and SwingDB. Although there are many studies about concurrency control protocols for OLTP systems [30], [35], [36], [37], [38], [39], 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.

ACKNOWLEDGMENTS

The authors would like to thank Wenbo Lang, Phillip Saenz and the anonymous reviewers. Guoren Wang is the corresponding author of this paper. 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. 61370154). 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 Transactions on Knowledge and Data Engineering (TKDE)*, vol. 27, no. 7, pp. 1920–1948, 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," *Proceedings of The VLDB Endowment (PVLDB'09)*, vol. 2, no. 1, pp. 1258–1269, 2009.
- [3] F. Özcan, Y. Tian, and P. Tözün, "Hybrid transactional/analytical processing: A survey," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017, pp. 1771–1775. [Online]. Available: <http://doi.acm.org/10.1145/3035918.3054784>
- [4] 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 *Proceedings of The VLDB Endowment (PVLDB'07)*, 2007, pp. 1150–1160.
- [5] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *2011 IEEE 27th International Conference on Data Engineering (ICDE'11)*. IEEE, 2011, pp. 195–206.
- [6] H. Plattner, "A common database approach for oltp and olap using an in-memory column database," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD'09)*. ACM, 2009, pp. 1–2.
- [7] 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 *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD'16)*. SIGMOD, 2016, pp. 311–326.
- [8] F. Funke, A. Kemper, and T. Neumann, "Benchmarking hybrid oltp&olap database systems," in *Datenbanksysteme fr Business, Technologie und Web (BTW'11)*, 2011, pp. 390–409.
- [9] Q. Meng, X. Zhou, S. Chen, and S. Wang, "Swingdb: An embedded in-memory dbms enabling instant snapshot sharing," in *International Workshop on In-Memory Data Management and Analytics (IMDB'16)*. Springer, 2016, pp. 134–149.
- [10] F. Farber, S. K. Cha, J. Primsch, C. Bornhovd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," *Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.
- [11] 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," *Proceedings of The VLDB Endowment (PVLDB'13)*, vol. 6, no. 11, pp. 1184–1185, 2013.
- [12] (2010) More details on today's outage. [Online]. Available: https://www.facebook.com/note.php?note_id=431441338919
- [13] 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 *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, 2013, pp. 1243–1254.
- [14] H. Mühe, A. Kemper, and T. Neumann, "How to efficiently snapshot transactional data: Hardware or software controlled?" in *Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN'11)*. ACM, 2011, pp. 17–26.

- [15] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Recent advances in checkpoint/recovery systems," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS'2006)*. IEEE, 2006, pp. 282–289.
- [16] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Conference Series (JPCS'07)*, vol. 78. IOP Publishing, 2007, pp. 012022–012032.
- [17] A.-P. Lienes and A. Wolski, "Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases," in *2006 IEEE 22th International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 99–99.
- [18] T. Cao, *Fault Tolerance For Main-Memory Applications In The Cloud*. Cornell University, 2013.
- [19] 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 *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*, 2011, pp. 265–276.
- [20] (2017) Wikipedia of Fork (system call). [Online]. Available: [https://en.wikipedia.org/wiki/Fork_\(system_call\)](https://en.wikipedia.org/wiki/Fork_(system_call))
- [21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10)*. ACM, 2010, pp. 143–154.
- [22] G. W. Y. Y. Liliang Li, Guoren Wang, "Consistent snapshot algorithms for in-memory database systems: Experiments and analysis," in *34rd IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*.
- [23] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson, "Low-overhead asynchronous checkpointing in main-memory database systems," in *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD'16)*. SIGMOD, 2016, pp. 1539–1551.
- [24] (2017) Redis. [Online]. Available: <https://redis.io>
- [25] P. Bernstein, "Actor-oriented database systems," in *34rd IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*.
- [26] 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," *Proceedings of The VLDB Endowment (PVLDB'08)*, vol. 1, no. 2, pp. 1496–1499, 2008.
- [27] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *ACM SIGMOD Record*, vol. 23. ACM, 1994, pp. 243–252.
- [28] (2017) Redis latency problems troubleshooting. [Online]. Available: <https://redis.io/topics/latency>
- [29] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison- Wesley, 1987.
- [30] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *PVLDB*, vol. 8, no. 3, pp. 209–220, 2014. [Online]. Available: <http://www.vldb.org/pvldb/vol8/p209-yu.pdf>
- [31] (2017) Redis persistence. [Online]. Available: <https://redis.io/topics/persistence>
- [32] (2017) Best ec2 setup for redis server. [Online]. Available: <https://stackoverflow.com/questions/11765502/best-ec2-setup-for-redis-server>
- [33] (2017) Redis memory and cpu spikes. [Online]. Available: <https://stackoverflow.com/questions/16384436/redis-memory-and-cpu-spikes>
- [34] (2017) Redis faq. [Online]. Available: <https://redis.io/topics/faq>
- [35] K. Ren, A. Thomson, and D. J. Abadi, "Lightweight locking for main memory database systems," vol. 6, no. 2, pp. 145–156, 2012.
- [36] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, 2013, pp. 18–32. [Online]. Available: <http://doi.acm.org/10.1145/2517349.2522713>
- [37] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas, "Tictoc: Time traveling optimistic concurrency control," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 1629–1642. [Online]. Available: <http://doi.acm.org/10.1145/2882903.2882935>
- [38] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, "An empirical evaluation of in-memory multi-version concurrency control," *PVLDB*, vol. 10, no. 7, pp. 781–792, 2017. [Online]. Available: <http://www.vldb.org/pvldb/vol10/p781-Wu.pdf>

- [39] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017, pp. 21–35. [Online]. Available: <http://doi.acm.org/10.1145/3035918.3064015>



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



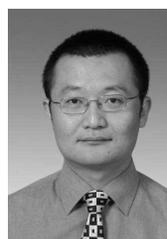
Guoren Wang received his BSc, MSc and PhD degrees in Computer Science from Northeastern University, China in 1988, 1991 and 1996, respectively. Currently, he is a Professor in the Department of Computer Science at Beijing Institute of Technology, China. His research interests include XML data management,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



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



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



Lei Chen received his BS degree in Computer Science and Engineering from Tianjin University, China in 1994, his MA degree from the Asian Institute of Technology, Bangkok, Thailand in 1997, and his PhD degree in Computer Science from the University of Waterloo, Canada in 2005. He is currently an Associate Professor in 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 in the Department of Computer Science, Kent University. His research interests include probabilistic/uncertain data management, probabilistic RDF graphs, inconsistent probabilistic databases, and

streaming time series.