2018 IEEE 34th International Conference on Data Engineering

Consistent Snapshot Algorithms for In-Memory Database Systems: Experiments and Analysis

Liang Li [#1], Guoren Wang [*#2], Gang Wu [#2], Ye Yuan [#3]

[#] *Computer Science and Engineering, Northeastern University, China*
[*] *Computer Science and Technology, Beijing Institute of Technology, China*
[1] liliang@stumail.neu.edu.cn [2] {wanggr,wugang}@mail.neu.edu.cn [3] yuanye@ise.neu.edu.cn

***Abstract*—In-memory databases (IMDBs) are gaining increasing popularity in big data applications, where clients commit updates intensively. Consistent snapshot is a key step in backup and recovery of IMDBs, thus an important factor for system performance of IMDBs. 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 the academia to trade off throughput and latency, yet industrial IMDBs such as Redis still stick to the simple fork algorithm. As an understanding of 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 the academia. Finally, we have opensourced the implementation of all the above snapshot algorithms to facilitate practitioners to benchmark the performance of each algorithm and select proper methods for different application scenarios.**

I. 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. In these applications, clients usually commit updates intensively, which requires efficient backup and recovery support. A key enabler to support such backup and recovery in IMDBs is in-memory consistent snapshot, where the system regularly takes an in-memory consistent time-in-point snapshot with two constraints: 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, including Consistent Checkpoint [2], [3] and Hybrid OLTP&OLAP (HTAP) Systems [4], [5], etc.

The wide applications of in-memory consistent snapshot have attracted extensive research interests in academia. Some of the representative snapshot algorithms are Naive Snapshot (NS) [6], [7], Copy-on-Update (COU) [2], [8], [9], Zigzag (ZZ) [3] and PingPong (PP) [3]. Besides, the simple fork [10] function is used as common snapshot algorithm in industrial systems (Redis, Hyper, etc). 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, besides, we observe several interesting questions: 1) Why do popular industrial IMDBs, *e.g.*, Redis, utilize the simple fork function instead of the state-of-the-art snapshot algorithms? 2) Whether the state-of-the-art snapshot algorithms are inapplicable in update-intensive workload scenarios? 3) Can we provide unified implementation and benchmark studies for future studies?

The Contributions of this work are:

1. We find that the simple fork() function indeed outperforms the state-of-the-arts in update-intensive workload scenarios. We conduct large-scale experiments on five mainstream snapshot algorithms (NS, COU, ZZ, PP, Fork). Surprisingly, we observe that the simple fork algorithm indeed outperforms the rest of the algorithms.

2. We propose two simple yet effective modifications of the state-of-the-arts, which have better tradeoff among latency, throughput, complexity and scalability. Based on the aforementioned experiments about mainstream snapshot algorithms, we identify the drawbacks of existing research and propose two lightweight improvements based on state-of-the-art snapshot algorithms. In particular, extensive evaluations on synthetic data and Redis show that our lightweight improvements yield better performance than fork and the representative snapshot algorithms from the academia.

3. We opensource our implementations [11], 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. We further integrate the two improved algorithms into Redis, and investigate the scalability with the Yahoo! Cloud Serving Benchmark (YCSB) [12]. We envision our experiences as valuable guidance for future snapshot algorithm design, implementation, and evaluation.

II. In-Memory Consistent Snapshot Algorithms

Definition 1 (In-Memory Consistent Snapshot) *Given an In-Memory database D, the snapshotter thread aims to take an in-memory consistent time-in-point snapshot, simultaneously, the client threads must be able to read the latest data items and should not overwrite the snapshot.*

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00131

A. Representative Snapshot Algorithms

1) Naive Snapshot: Naive snapshot (NS) [6][7] takes a snapshot of dataset D during the snapshot taken phase when the client thread is blocked. Once the snapshot $\overline{D}$ is taken in memory, the client thread is then resumed. Meanwhile, 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.

2) Copy-on-Update and fork: Copy on Update (COU) [9] utilizes an auxiliary data structure $\overline{D}$ to shadow copy D and a bit array $\overline{D}_b$ to record 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 $\overline{D}$ and a setting to the corresponding bit of $\overline{D}_b$ to indicate the state before the page update. In COU, the snapshotter thread can utilize the $\overline{D}_b$ to access the snapshot. Note that COU has many variants and here we refer to the latency-spike-free implementation in [9]. The fork function [10] is also a system-level COU variant, many popular industrial systems such as Redis [13] and Hyper [4] exploit fork to take snapshots.

3) Zigzag: Zigzag (ZZ) [3] employs one shadow copy $\overline{D}$ (the same size as D) and two auxiliary bit arrays $\overline{D}_{br}$ and $\overline{D}_{bw}$. For a page i, $\overline{D}_{br}[i]$ and $\overline{D}_{bw}[i]$ are responsible for indicating which copy the client should read from or write to, respectively. Hence $\neg\overline{D}_{bw}[i]$ indicates the copy that the snapshotter thread should access since this copy cannot be written by the client. In this way, ZZ is able to avoid overwritten and keep an untouched snapshot data with the help of $\overline{D}_{bw}[i]$.

4) Ping-Pong: Ping-Pong (PP) [3] is proposed to completely eliminate the latency spikes. 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 kept in $\overline{D}_u$. PP attains an immediate swapping by exchanging the pointers $\overline{D}_u$ and $\overline{D}_d$.

B. Improved Snapshot Algorithms

Previous snapshot algorithms mainly trade off between latency and throughput. To simultaneously achieve low latency, high throughput, small time complexity and free of latency spikes, we propose two lightweight improvements, Hourglass and Piggyback, over existing snapshot algorithms.

1) Hourglass: One intuitive improvement over the above snapshot algorithms is to combine Zigzag (bit array marking) and Ping-Pong (pointers swapping) to avoid latency spikes while retaining a small memory footprint. We call this improvement Hourglass (HG). It maintains the 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 page in $D[i]$ or $\overline{D}[i]$ has been updated during the current period. Hourglass utilizes these bit arrays to record the incremental data updates in the current period. Pointers 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 pages. A zero of the bit $\overline{D}_{br}[i]$ means that the latest data locates in $D[i]$, and the value one means in $\overline{D}[i]$. Fig. 1 illustrates how Hourglass works during two successive snapshots.

(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

Figure 1. Running example for Hourglass (HG)

EXAMPLE 1 (Hourglass) *As shown in Fig. 1(a), assume that at time t_0, D=$\overline{D}$. $\overline{D}_{b1}$ 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 page i, $\overline{D}_{b1}[i]$ is set to 1 and $\overline{D}_{br}[i]$ is set to 0. $\overline{D}$ will be isolated 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 $\overline{D}$ has been used (*i.e.*, accessed or dumped), the jth position in $\overline{D}_{b2}$ is reset to 0. At the end of this period, all bits in $\overline{D}_{b2}$ are reset to zeros. Fig. 1(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 $\overline{D}$ are swapped as in Fig. 1(c). Then in the access phase, the snapshotter thread begins to access the incremental snapshot data (marked in yellow shadow in Fig. 1(c)) from D. Only those pages pointed by pD where the corresponding bits set to ones will be included in the snapshot. During this time, the client thread resumes to execute transactions. The state at the end of P_2 is shown in Fig. 1(d).*

2) Piggyback: Although pointers swapping (in Ping-Pong and HG) eliminates latency spikes, it is only applicable for incremental snapshots. To enable full snapshots with the pointers 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 pointers swapping.

To support piggyback copies (copy from pD to pU), the Piggyback algorithm leverages two techniques. 1) Piggyback defines a WriteToOnline() function which ensures the data pointed by pU will always be the latest at the end of each period. 2) Piggyback maintains a two-bit array $\overline{D}_b$. The value of $\overline{D}_b[i]$ is one of three states in $\{0, 1, 2\}$, which indicates from which dataset the client thread should read. When $\overline{D}_b[i]$=0, the client thread can read page i from either array because it means that $D[i]$=$\overline{D}[i]$. When $\overline{D}_b[i]$=1, the client thread should read page i from $D[i]$. When $\overline{D}_b[i]$=2, the client thread should read page i from $\overline{D}[i]$. The advantage of three states for $\overline{D}_b$ is that it is able to minimize the size of the data set that needs to be copied. For the space reason, pseudo code for HG and

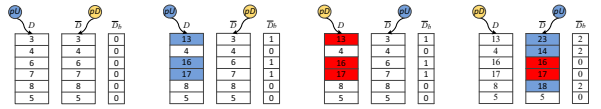
Table I
COMPARISON OF ALGORITHMS IN DIFFERENT METRICS; "(*)" REPRESENTS THE DRAWBACK

Algorithms	Average Latency	Latency Spike	Snapshot Time Complexity	Max Throughput	Is Full Snapshot	Max Memory Footprint
Naive Snapshot [6], [7]	low	(*) high	(*) O(n)	low	yes	2×
Copy-on-Update [2], [8], [9]	(*) high	(*) middle	(*) O(n)	middle	yes	2×
Fork [10]	low	(*) middle	(*) O(n)	high	yes	2×
Zigzag [3]	middle	(*) middle	(*) O(n)	middle	yes	2×
Ping-Pong [3]	(*) 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×

PB are offered on website.[1]

EXAMPLE 2 (Piggyback) *Initially, pU and pD point to D and $\overline{D}$, respectively. The bit array $\overline{D}_b$ is set to zeros as shown in Fig. 2(a). Fig. 2(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 $\overline{D}_b$ 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 $\overline{D}_b$. 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 of D and can be accessed. Meanwhile, $\overline{D}$ can be updated by the client thread. Note that there may be dirty pages in $\overline{D}$ in the P_3 period. For instance, $\overline{D}[0]$, $\overline{D}[2]$ and $\overline{D}[3]$ (red shadow) are older pages (Fig. 2(c)). To avoid dirty pages, Piggyback performs a piggyback copy of these pages from D to $\overline{D}$ in this period together with the client's normal updates on pages $\overline{D}[0]$, $\overline{D}[1]$ and $\overline{D}[4]$ (blue shadow). Hence at the end of P_2, all the pages in $\overline{D}$ are updated to the latest state as shown in Fig. 2(d).



(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

Figure 2. Running example for Piggyback (PB)

C. Comparison of Snapshot Algorithms

Table I 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 outperforms the rest in all the metrics.

III. EXPERIMENTAL STUDIES

This section comprehensively evaluates the performance of various snapshot algorithms from the previous section. We first present a thorough benchmark study (Sec. III-B). Then we implement two Redis variants by integrating HG and PB, respectively, to study the scalability in real-world IMDB systems (Sec. III-C). For the space reason, we only present a subset of our experimental results here.

[1]http://t.cn/RQXZZ7Y

A. Evaluation Environment

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 with Linux kernel 2.6.32 and GCC 5.1.0 was installed.

B. Benchmark Study

Setups: we benchmark all snapshot algorithms in checkpoint application and follow the setups in [3]. The update-intensive client is simulated by a Zipfian [14] distribution random generator. Since the Zipfian distribution parameter α has little impact on the experimental results [3], we set α to 2 by default. To carefully control the update frequency, the client thread runs in a tick-by-tick (a.k.a. time slice) way [3]. The tunable parameters are update frequency (***uf***) and ***dataset size***.

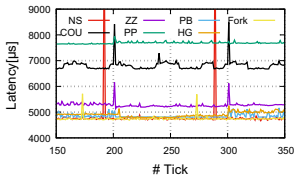


Figure 3. Latency distribution (***uf***=256K)

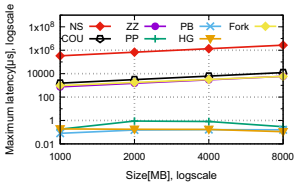


Figure 4. Data size vs. Maximum latency

Results: Fig. 3 plots the latency traces on a 1000MB dataset with uf=256K per tick. Fig. 4 shows the maximum latency with the increase of 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 magnitude orders lower than that of NS, COU, Fork, and ZZ. Moreover, the maximum latency of latter algorithms grows steadily with the increase of the dataset size. The good performance of PP, HG and PB owes to the pointers swapping technique. Our benchmark evaluations on synthetic workload also reveal the following findings: 1) Fork outperforms NS, COU, ZZ and PP in both latency and throughput, besides, fork has a simple engineering implementation. That is the reason why fork is adopted in several industrial IMDBs such as Redis. 2) The latency spike of PB and HG is not affected by the data size (see Fig. 4). In general, PB and HG are more scalable than the other algorithms including fork. 3) NS, Fork, COU, ZZ

and PP are fit for specific applications (*i.e.*, they perform well either on latency or throughput). PB and HG trade off latency, throughput and scalability, which are fit for a wider range of applications.

C. Performance in Industrial IMDB System

Redis is a popular In-Memory NoSQL system and it utilizes fork to persist data [15]. 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 [16]. In this performance study, we aim to harness proper snapshot algorithms to improve the scalability of snapshots in Redis.

Snapshot Algorithm Selection: To optimize the scalability of Redis, an option is to replace fork with snapshot algorithms of O(1) complexity. According to Table 1, HG and PB are candidates, we implement two Redis variants Redis-HG and Redis-PB using the HG and PB algorithms, respectively.

Dataset: We chose YCSB (Yahoo! Cloud Serving Benchmark) [12] to validate the practical performance of our prototypes. 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.

Results: Fig. 5 illustrates the change of throughput as the benchmark record count grows. We observe that Redis-HG and Redis-PB have similar throughput performance to the default Redis. The reason is that although Redis-HG and Redis-PB can avoid locks between data updating and dumping, they need additional checking for each read/write operation. Therefore the throughout improvement is marginal.

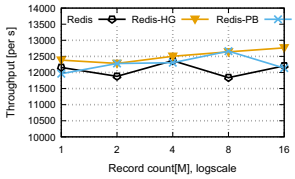


Figure 5. Redis: YCSB record count vs. Throughput

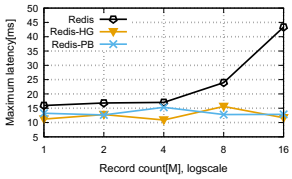


Figure 6. Redis: YCSB record count **vs.** Maximum latency

Fig. 6 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 dramatic increase in maximum latency when the record count reaches 8 million. This result is consistent with the Redis document that 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 pointers swapping technique employed in the snapshot taken phase, which only needs almost constant and 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 a weak scalability. Conversely, Redis-HG and Redis-PB can scale to larger datasets than the default Redis.

Summary. Redis with the built-in fork function is unscalable (see Fig. 6). By replacing the default fork with HG and PB, the two variants, Redis-HG and Redis-PB, exhibit better scalability.

IV. Conclusions

In this paper, we analyze, compare, and evaluate representative in-memory consistent snapshot algorithms from both academia and industries. 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 algorithms achieve low latency, high throughput, small time complexity, and no latency spikes simultaneously, yet these requirements are essential for update-intensive big data applications. Therefore, 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 investigations on consistent snapshot algorithms.

Acknowledgment: We would like to thank Wenbo Lang, Phillip Saenz and the anonymous reviewers. Guoren Wang is the corresponding author. This paper is partially supported by the NSFC (Grant No. 61332006, U1401256, 61370154, 61732003, 61729201, 61572119 and 61622202) and the Fundamental Research Funds for the Central Universities (Grant No. N150402005).

References

[1] H. Zhang, "In-memory big data management and processing: A survey," *TKDE*, vol. 27, no. 7, pp. 1920–1948, 2015.
[2] M. A. V. Salles, "An evaluation of checkpoint recovery for massively multiplayer online games," *PVLDB*, vol. 2, no. 1, pp. 1258–1269, 2009.
[3] T. Cao, "Fast checkpoint recovery algorithms for frequently consistent applications," in *SIGMOD*, 2011, pp. 265–276.
[4] A. Kemper, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE*. IEEE, 2011, pp. 195–206.
[5] Q. Meng, X. Zhou, S. Chen, and S. Wang, "Swingdb: An embedded in-memory dbms enabling instant snapshot sharing," in *IMDM*. Springer, 2016, pp. 134–149.
[6] G. Bronevetsky, "Recent advances in checkpoint/recovery systems," in *IPDPS*. IEEE, 2006, pp. 8–pp.
[7] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *JPCS*, vol. 78. IOP Publishing, 2007, p. 012022.
[8] A.-P. Liedes and A. Wolski, "Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases," in *ICDE*. IEEE, 2006, pp. 99–99.
[9] T. Cao, *Fault Tolerance For Main-Memory Applications In The Cloud*. Cornell University, 2013.
[10] "Wikipedia of Fork (system call)," https://en.wikipedia.org/wiki/Fork_(system_call).
[11] (2017) Source code. [Online]. Available: https://github.com/bombehub/FrequentSnapshot
[12] B. F. Cooper, "Benchmarking cloud serving systems with ycsb," in *SoCC*. ACM, 2010, pp. 143–154.
[13] "Redis," https://redis.io.
[14] J. Gray, "Quickly generating billion-record synthetic databases," in *ACM SIGMOD Record*, vol. 23. ACM, 1994, pp. 243–252.
[15] "Redis persistence," http://redis.io/topics/persistence.
[16] (2017) Best ec2 setup for redis server. [Online]. Available: https://stackoverflow.com/questions/11765502/best-ec2-setup-for-redis-server