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


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

[1] College of Computer Science and Engineering, Northeastern University, China

[2] College of Computer Science and Technology, Beijing Institute of Technology, China

{liliang@stumail, wanggr@mail, wugang@mail, yuanye@ise}.neu.edu.cn


## Introduction

- In-memory consistent snapshot have attracted extensive research interests in academia.
- Consistent snapshot can be applied in diverse real-life applications, including **Consistent Checkpoint** and Hybrid OLTP&OLAP (**HTAP**) Systems, etc.
- Some of the **representative snapshot algorithms** are Naive Snapshot (NS) , Copy-on-Update (COU) , Zigzag (ZZ) and PingPong (PP) . Besides, the simple **fork()** function is used as common snapshot algorithm in industrial systems (Redis, Hyper, etc).

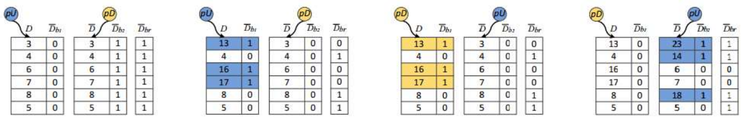
- **Motivations:**
  1. Why do popular industrial IMDBs, e.g., Redis and Hyper, utilize the simple fork() function instead of the state-of-the-art snapshot algorithms?
  2. Whether the state-of-the-art snapshot algorithms are inapplicable in update-intensive workload scenarios?
  3. Can we provide unified implementation and benchmark studies for future studies?

- **Contributions:**
  1. We find that the simple fork() function indeed outperforms the state-of-the-arts in update-intensive workload scenarios.
  2. We propose two simple yet effective modifications of the state-of-the-arts, which have better tradeoff among latency, throughput, complexity and scalability.
  3. We opensource our implementations, algorithmic improvements, and benchmark studies as guidance for future researchers.

## Problem Statement

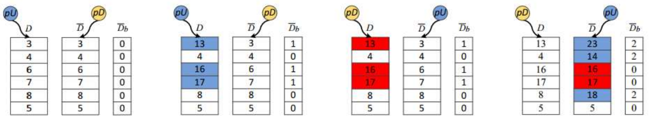
- **Problem:** Given a database D, and assume D is updated intensively by clients. We aim to take an in-memory consistent time-in-point snapshot, at the same time, the clients should satisfy the following constraints:
  - **Read constraint:** Clients should be able to read the latest data items.
  - **Update constraint:** Any data item in the snapshot should not be overwritten. In other words, the snapshot must be read-only.

## Proposed Snapshot Algorithm



- **Hourglass:** As shown in Fig. 6(a), assume that at time $t_0$, D= $\overline{D}$ ={3, 4, 6, 7, 8, 5}. $\overline{D}_{b1}$ and $\overline{D}_{b2}$ are initialized with zeros and ones, respectively. $\overline{D}_{br}$ is initialized with ones. During $P_1$, when an update occurs on page i, $\overline{D}_{b1}[i]$ is set to 1 and $\overline{D}_{br}[i]$ is set to 0. $\overline{D}$ 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 $\overline{D}$ has been accessed, 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. 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 $\overline{D}$ 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 will be 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 to execute transactions. The state at the end of $P_2$ is shown in Fig. 6(d).

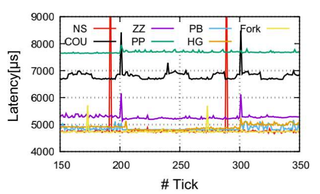
- **Piggyback:** Initially, pU and pD are pointed to D and $\overline{D}$ , respectively. The bit array $\overline{D}_b$ is set to zeros as shown in Fig. 7(a). Fig. 7(b) shows the situation at time $t_1$. The client thread updates pages D[0], D[2], and D[3] (blue shadow) during the first period. The corresponding two-bit elements in $\overline{D}_b$ 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 in 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. 7(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. 7(d).

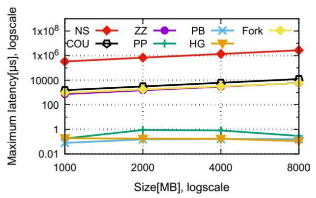


## Comparison of algorithms in different metrics

| Algorithms | Average Latency | Latency Spike | Snapshot Time Complexity | Max Throughput | Is Full Snapshot | Max Memory Footprint |
|---|---|---|---|---|---|---|
| Naive Snapshot [14], [15] | low | (*) high | (*) O(n) | low | yes | 2× |
| Copy-on-Update [2], [16], [17] | (*) high | (*) middle | (*) O(n) | middle | yes | 2× |
| Fork [19] | low | (*) middle | (*) O(n) | high | yes | 2× |
| Zigzag [18] | middle | (*) middle | (*) O(n) | middle | yes | 2× |
| Ping-Pong [18] | (*) 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× |

## Performance Evaluation




- **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. 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.

## Implementation in REDIS system