Received July 6, 2021, accepted July 28, 2021, date of publication August 9, 2021, date of current version August 16, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3103862

Validity Tracking Based Log Management for In-Memory Databases

KWANGJIN LEE[1,2], HWAJUNG KIM[1], AND HEON Y. YEOM[1], (Member, IEEE)
[1]School of Computer Science and Engineering, Seoul National University, Seoul 08826, Republic of Korea
[2]Memory Division, Samsung Electronics, Hwaseong 18448, Republic of Korea

Corresponding author: Heon Y. Yeom (yeom@snu.ac.kr)

This work was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) by the Ministry of Science and ICT under Grant NRF-2021R1A2C2003618.

ABSTRACT With in-memory databases (IMDBs), where all data sets reside in main memory for fast processing speed, logging and checkpointing are essential for achieving persistence in data. Logging of IMDBs has evolved to reduce run-time overhead to suit the systems, but this causes an increase in recovery time. Checkpointing technique compensates for these problems with logging, but existing schemes often incur high costs due to reduced system throughput, increased latency, and increased memory usage. In this paper, we propose a checkpointing scheme using validity tracking-based compaction (VTC), the technique that tracks the validity of logs in a file and removes unnecessary logs. The proposed scheme shows extremely low memory usage compared to existing checkpointing schemes, which use consistent snapshots. Our experiments demonstrate that checkpoints using consistent snapshot increase memory footprint by up to two times in update-intensive workloads. In contrast, our proposed VTC only requires 2% additional memory for checkpointing. That means the system can use most of its memory to store data and process transactions.

INDEX TERMS Checkpointing, in-memory database, logging, persistence, snapshot.

I. INTRODUCTION

In-memory databases (IMDBs) are designed to achieve fast response time by processing data using the main memory, without accessing the disk. For this reason, IMDBs are widely adopted for various applications [1], such as e-commerce online transaction processing (OLTP) services, online games [2], finance [3], and more. The entire data residing in memory guarantees fast processing, but there is a risk of data loss due to system crashes, hardware failures, and power outages. To improve fault tolerance in long-running applications, IMDBs provide persistence through a variety of strategies. Checkpointing and logging are widely used techniques for the durability of IMDBs. Disk-based databases prefer ARIES-style [4] logging and checkpointing protocols, while most IMDBs record only redo logs excluding undo logs to reduce logging overhead and help performance. In addition, IMDBs need to checkpoint much more data than disk-based databases, so it is common to use an algorithm suitable for this, such as consistent snapshots [5], [6].

The associate editor coordinating the review of this manuscript and approving it for publication was Genoveffa Tortora.

Many systems provide persistence by combining logging and checkpointing. Systems that guarantee data durability only with periodic checkpointing can reduce run-time overhead, but the trade-off is that the system can lose a large amount of data due to system failure. Systems that use logging can lower the risk of data loss. Although logging increases the recovery interval and requires more storage space due to logs that accumulate over time, these problems can be alleviated by using checkpointing together. The combination of logging and checkpointing reduces recovery time by loading the latest checkpoint file and rerunning only subsequent logs. Furthermore, it allows for the space used by the logs to be reused.

Checkpointing plays an important role in effectively providing persistence for IMDBs, but it incurs significant costs for system throughput, latency, and peak memory usage. Figure 1 shows the memory footprint and throughput of Redis, one of the most popular commercial IMDBs, during checkpointing. When using Redis, the memory footprint continues to grow over time until checkpointing is complete. As a result, memory usage at the end of checkpointing has increased up to 67%. Moreover, throughput decreases by 38% at the beginning of checkpointing due to frequent

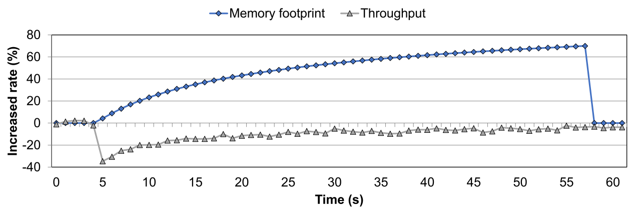


FIGURE 1. Redis memory footprint and throughput during checkpointing for the Yahoo! Cloud Serving Benchmark (YCSB) (50% update).

copy-on-write (CoW) operations. Redis performs checkpointing by periodically taking consistent snapshots and storing them to stable storage. Data in a consistent snapshot should not be overwritten during checkpointing, so changes made to a database during client update requests are handled by CoW semantics. Since physical pages copied by CoW are not reclaimed until the checkpoint is completed, the memory footprint may increase up to two times during an update-intensive workload.

In a broad sense, a checkpoint is a technique that aims to keep the persistent state of a database up to date with the goal of reducing recovery time and reusing log space. Redis [7] and Hyper [8] simply take a consistent snapshot and store the contents of the snapshot through the `fork()` system call [9] and CoW semantics supported by the OS. However, this method has problems such as latency spikes [10] and increased memory usage. DMC [11] allows pages to be returned to the OS sooner before checkpointing is complete. This lowers the peak memory usage during checkpointing, but does not completely solve the increase in memory according to the update rate. Hekaton [12] and CALC [13] attempt to reduce the checkpointing overhead by using a partial checkpoint algorithm. This algorithm reduces cost by taking partial checkpoints that contain only some of the latest records. However, the process of merging partial checkpoints to create a complete checkpoint incurs another overhead.

In this paper, we propose a validity tracking-based log management scheme to provide improved durability and efficient checkpointing by minimizing the use of additional memory. By distributing and storing logs across multiple storage devices, we can hide the latency caused by the log buffer flush so that logging does not affect the throughput of the system. It also improves durability by reducing the flush cycle of the log buffer. Instead of generating checkpoints from the data on the main memory, our validity tracking-based compaction (VTC) scheme creates checkpoints by identifying valid logs in log files. VTC uses only a small amount of extra memory because it does not require physical page duplication or multiple versioning in main memory.

We provide the proposed scheme as a simple user-level API. To test this system, we applied our scheme to Redis 5.0.6 and evaluate it with the Yahoo! Cloud Serving Benchmark (YCSB) [14] to compare its performance against the persistence schemes of Redis. The experimental results show that VTC consumes little memory to perform checkpoints and does not adversely affect system throughput and checkpoint time. With update-intensive workloads, VTC uses less than 2% of the size of the data set, while the memory footprint of the checkpointing scheme using consistent snapshots increases memory usage by up to 200%. This means that VTC permits most of an IMDB's primary resource, system memory, to be used for data storage.

Our contributions can be summarized as follows.

- We analyze the persistence scheme for an existing in-memory database.
- We propose a persistence scheme that distributes and stores logs on multiple storage devices and removes unnecessary logs in the file by tracking log validity.
- We provide high-level APIs that can be easily applied to the existing IMDB with little modification.
- The experimental results show that our scheme offers slightly better throughput than the existing Redis logging scheme and maintains a more stable memory footprint than the existing Redis checkpointing scheme.

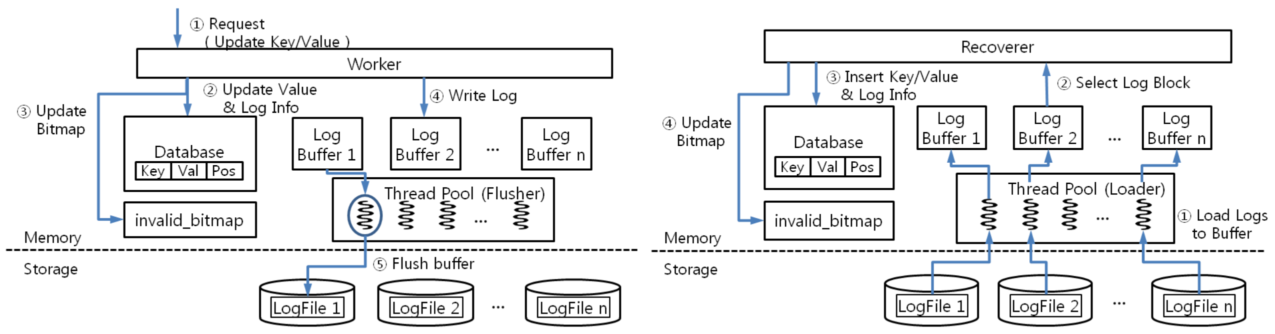
The rest of the paper is organized as follows. Section II describes the background and motivation. Section III introduces the design and implementation of our proposed scheme. Section IV evaluates VTC and other persistence schemes. Section V presents a discussion of the design and experimental results. Section VI discusses the related works. Finally, Section VII concludes this paper.

II. BACKGROUND AND MOTIVATION

A. PERSISTENCE IN IN-MEMORY DATABASES

Because all of the data for IMDBs is in DRAM, which is a volatile memory, it is crucial to guarantee data durability with a fault-tolerance mechanism that will help prevent data loss in case of system failure [15]. IMDBs prefer data replication for fast failover and generally maintain replicas across multiple nodes to achieve high availability [12], [16], [17], [18]. However, catastrophic failures such as cluster-wide power outages can cause data loss if the data are not in stable storage. To avoid this issue, data must be kept in stable storage to ensure durability. The traditional techniques used for database durability are logging and checkpointing.

Most disk-based databases guarantee transaction durability with ARIES-style [4] logging. The ARIES protocol uses a write-ahead logging (WAL) scheme that sequentially records changes before modified pages are written to disk, and log records include redo and undo. Early IMDBs used similar techniques [19], [20]. However, logging to IMDBs is gradually optimized for high throughput and low latency, and the traditional logging scheme, which is relatively expensive compared to light transaction processing without disk access, is simplified for in-memory systems. In general, IMDBs reduce log volume by recording only the redo log and minimizing the log record's information to mitigate the effects from log creation and log I/O overhead. However, replaying the log for recovery increases the recovery time. Additionally, WAL can recycle log space after applying all logs to the data file, whereas logging in IMDBs consumes more space



(a) Logging with multiple SSDs (b) Recovery with multiple buffers

FIGURE 2. **Overall procedure.**

over time. Therefore, periodic checkpointing is required to reduce the recovery time and recycle log space.

Checkpointing is also different in IMDBs. Because the entire data sets for IMDBs are kept in the main memory, it is common for their checkpoints to be larger than those of disk-based databases. Many IMDBs use a checkpointing algorithm that takes a consistent snapshot and stores it in stable storage. The wide application of consistent snapshots has led to extensive research in academia, and various algorithms [2], [21], [22] have been proposed. In fact, the commercial systems Redis and Hyper use the `fork()` system call as a consistent snapshot algorithm.

Redis, the most popular key-value IMDB [23], provides persistence via Redis data backup (RDB) and an append-only file (AOF). RDB is a feature that backs up the entire database in memory. It obtains a consistent snapshot using `fork()` and stores the contents of the snapshot in stable storage in the background through the child process created by `fork()`. AOF, a logging feature supported by Redis, appends a log of events that have changed the database to the log file. When a log file exceeds a specific size, the system acquires a snapshot, converts its contents to log format, and saves them as a file. As a result, it creates a new log file consisting of only the logs needed for recovery. This process mitigates the increase in log space and recovery time.

B. FORK-BASED CHECKPOINTING

Fork-based checkpointing is a simple but efficient scheme that creates point-in-time consistent snapshots and store them in stable storage with OS supports. It has been demonstrated that the fork-based consistent snapshot algorithm outperforms other algorithms [2], [21], [22] for update intensive workloads [6]. In fact, many industrial IMDBs like Hyper [8] and Redis [7] employ the algorithm for checkpointing.

The `fork()` system call is used to create a child process by duplicating a process. Physical pages are not actually copied by `fork()`, and both processes refer to the same physical pages through virtual memory pages. If a page shared by both processes needs to be updated, the CoW technique copies the physical page to a new memory space and modifies it. Thanks to this CoW technique, point-in-time data on the child process (checkpointer) is not affected even as the parent process (worker) handles the client's update request. After `fork()`, the checkpointer traverses the snapshot and saves point-to-time data to a file.

However, there are well-known problems with fork-based checkpointing. The first problem is the latency spike due to the blocking operation, `fork()`, which occurs because IMDBs cannot process or respond to client requests while creating a process by `fork()`. Moreover, latency due to `fork()` increases as the size of the data set increases. The second problem is increased latency due to CoW. During the checkpointing period, the overhead due to CoW for processing update requests increases latency and affects throughput. If the update rate of the workload is high, CoW will frequently have a greater impact on the throughput. Finally, the increase in memory footprint by CoW is the most crucial problem. The parent process handles client requests while the child process created by `fork()` writes the snapshot data to the file. If the request is an update, the physical page is copied by the CoW, which causes an increase in the memory footprint. Moreover, the memory footprint increases proportionally to the update rate of the workload. In the worst case of all pages being updated, the required memory size is twice that of the data set. Furthermore, the increased memory cannot be reclaimed until the child process is terminated. If there is no more available memory, either the transaction processing and checkpoint speed will be significantly slowed during the swap, or the out-of-memory killer will kill the processes. For this reason, many IMDB vendors [24] recommend that users take into consideration the memory increase due to `fork()` and set the swap to prevent out-of-memory problems.

We focused on a persistence scheme that combines logging and checkpointing, along with a checkpointing algorithm to minimize the memory use increase and provide stable throughput.

III. DESIGN AND IMPLEMENTATION

A. DESIGN OVERVIEW

Our key idea is to distribute the logs into multiple files and reorganize them by identifying valid logs in the files. Figure 2 shows the overall procedure of the proposed scheme.

The worker creates logs of events that change the database and writes them to the log buffers. The flushers then flush the log buffer to the stable storage in the background. We hid the latency caused by flush by dividing the log into several SSDs and improved durability by reducing the flush interval.

The VTC scheme performs checkpointing based on logs stored in storage rather than on data in the main memory. VTC leaves only the logs needed for recovery through file-to-file copying, so to achieve this, we maintained an up-to-date log location for each database entry. An `invalid_bitmap` is allocated per log file, and each bit indicates the validity of each log in the file. The `invalid_bitmap` makes it simple to examine the validity of the log during checkpointing. When the number of invalid logs in each log file reaches the threshold, chekcpointing is triggered. Only one file can be checkpointed at a time, and logs are not stored in the file until checkpointing is complete. The separation of I/O between logging and checkpointing reduces checkpointing time and avoids log flush delays due to latency spikes caused by checkpointing.

Recovery works by sequentially replaying logs read from log files. To maximize the I/O bandwidth during recovery, several loaders simultaneously read logs from files and fill the buffers. Since the logs in each log block are guaranteed to be serialized, the recoverer compares the timestamps of the log blocks and processes them in order, starting with the log block having the smallest value.

We will explain each scheme in more detail in the following sections.

B. DISTRIBUTED LOGGING AND LOG DATA FORMAT

For strong durability, logs should be immediately stored in stable storage. Unfortunately, synchronous durability leads to performance degradation. Therefore, many systems adopt asynchronous durability that buffers logs and flushes them to stable storage periodically. If the interval is too long, a large amount of data may be lost in case of system failure. If the interval is too short, the system throughput may be degraded due to write stalls. Write stalls occur when the storage device operation for the previous buffer flush is not completed when attempting to flush the buffer.

We use multiple storage devices to store logs in order to overcome the limitation on durability due to the storage device's performance. In addition, the use of multiple storage devices makes it possible to separate storage I/O for logging and checkpointing. This strategy ensures a stable log buffer flush cycle and system throughput by avoiding the effect of latency spikes on logging that can occur due to large data storage during checkpointing.

Figure 2(a) shows the processing and logging for client requests. When a request such as insert or update is received from a client (①), the worker reflects the processing result to the corresponding entry in the database and also stores the location information where the log will be stored in a variable called `log_pos` (②). The variable exists for each entry in the database and is used for checkpointing. If the entry already exists, the worker updates the `invalid_bitmap` to invalidate its old log (③). After that, the worker creates a log and writes it to the active log buffer (④). When the log buffer is filled to more than the minimal number of logs, the worker requests that the log buffer be flushed and then activates the next log buffer. Finally, durability is guaranteed when the buffer is flushed to stable storage by the flusher (⑤). For recovery, it is necessary to identify the order of logs distributed across multiple SSDs, but because the logs stored in each buffer are serialized, we only need to clarify the order between log blocks, which is a unit flushed from buffer to storage. To do this, we add a 4-byte timestamp to the header of the log block.

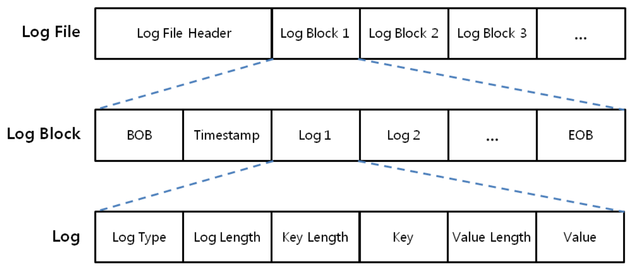


FIGURE 3. Proposed log format.

Figure 3 shows our proposed log format. An existing log file consists of a log file header and logs. We add a compaction timestamp to the header to demarcate the checkpointed and unchecked parts of the file. In addition, for efficient recovery, we store logs that are flushed to log files at once as log blocks. The log block have a 4-byte timestamp to determine the recovery order. 1-byte BOB (Beginning of Block) and 1-byte EOB (End of Block) indicate the start and end of a log block, respectively. The existing log contains information such as log type, key, and value necessary for recovery. Our log contains the log length along with the existing information. The value helps ascertain the size of each log without complex parsing when copying logs during checkpointing.

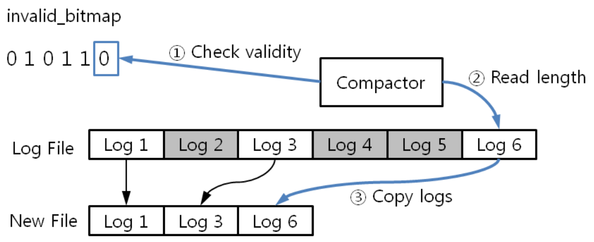


FIGURE 4. Log file compaction.

C. LOG FILE COMPACTION

We propose the VTC, a checkpointing scheme to reduce recovery time and recycle log space. Figure 4 presents the key idea of VTC. The VTC identifies the log needed for recovery by referring to `invalid_bitamp` (①). It then checks the length of the log (②) and copies it to a new file if it is valid (③).

To manage the validity of logs in files, it is necessary to know the location of the latest log for each entry in the database. For this reason, the variable `log_pos` was defined to indicate the location of the log within the structure of the existing database entries. Logs in the file are accessed sequentially from the first log when restoring data or checkpointing. Because there is no need to search for a specific log, `log_pos` represents the order of logs in the file rather than a byte address. The upper few bits of `log_pos` are used to indicate which file the log is stored in. The `invalid_bitmap` allows the VTC to immediately recognize whether the logs are valid. One `invalid_bitmap` exists for each log file, and its bits indicate the validity of each log in the file. If the bit is 1, it means that the log is no longer needed for recovery.

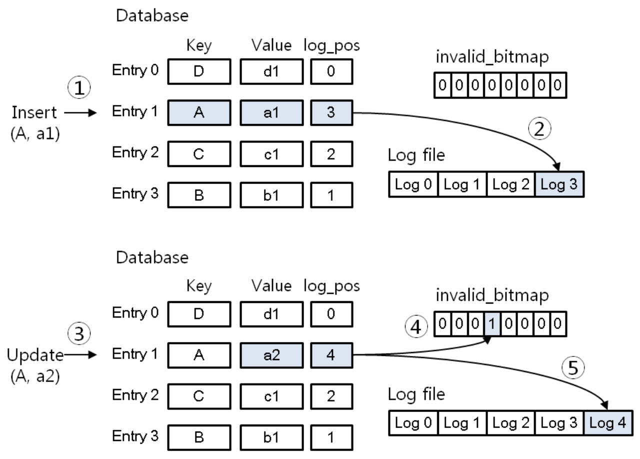


FIGURE 5. Example of logging for insert and update.

Figure 5 shows how the worker processes inserts and updates from clients. First, when a client requests an insert for a new key, the worker creates an entry with the key and value and adds it to the database. In addition, the location where the log will be stored is recorded in the variable `log_pos` in the entry (①). Then, the worker creates a log for insert and appends it the active log file (②). Later, when an update request for the same key is received, the worker updates the value and `log_pos` in entry (③). Because the previous insert log for the key is no longer needed for recovery, the worker sets the bit in the `invalid_bitmap` corresponding to the old value of `log_pos` (④). Finally, a log for the update request is created and stored in the active log file (⑤). The insert log and update log for the same entry may be saved in different files. The procedure for delete requests is similar. The worker deletes the entry and sets the `invalid_bitmap` in the same way as for update. Subsequently, the worker appends the delete log to the active log file for recovery.

The gradually accumulated logs not only require more disk space but also take more time to recover. VTC prevents

Algorithm 1 Overview of the VTC Procedure

```
1:  cp_time = get_curr_time();                              ▷ step 1
2:  min_cp_time = get_min_cp_time();
3:  dst_fp = create_file(temp.log)
4:  write_cp_header(dst_fp, cp_time);
5:  delta = allocate(log_count)
6:  for each log blocks stored in file do                   ▷ step 2
7:      blk_ts = get_timestamp(block);
8:      for each logs stored in logblock do
9:          type = get_type(log)
10:         len = get_len(log)
11:         if type = delete then
12:             if blk_ts > min_cp_time then
13:                 copy_log(dst_fp, log, len)
14:             else
15:                 removal = removal + 1
16:         else
17:             if is_valid(bitmap, i) then
18:                 copy_log(dst_fp, log, len)
19:             else
20:                 removal = removal + 1
21:         delta[log_num] = removal
22:         i = i + 1
23: flush(dst_fp)
24: clear_invalid_bitmap(bitmap)
25: for each entries stored in DB do                        ▷ step 3
26:     old_pos = entry→log_pos
27:     if is_on_compaction(old_pos) then
28:         entry→log_pos = old_pos − delta[old_pos]
29:     if entry→lazy = 1 then
30:         set_invalid_bitmap(bitmap, entry→log_pos)
31:         entry→lazy = 2
32: release(delta)                                          ▷ step 4
33: rename_file(dst_fp)
34: delete_file(src_fp)
```

this problem by removing unnecessary logs from log files. Algorithm 1 describes the VTC procedure, which requires four steps: preparing, copying, remapping, and completing. In the preparing step, VTC creates a new file, `temp.log`, to copy the valid log and writes the current timestamp as the compaction timestamp. In addition, VTC allocates memory for `delta`, a temporary array for calculating the locations of logs moved by copying (step 1, lines 1—5). The size of the array is determined by the number of logs in the file. Then, in the second step, copying, VTC sequentially reads logs from the target log file and copies only valid logs to `temp.log` file. VTC identifies the validity of each log by referring to the `invalid_bitmap` and copies the logs by referring to the log length in the header, without complicated parsing (step 2, lines 6—20). This step also fills the `delta` array to be referred to in the next step, remapping (step 2, line 21). Each element of the `delta` array corresponds to the logs in the file at the start of compaction. The values of the elements indicating the number of previously removed logs are referred to update the location of logs moved by compaction. After copying all valid logs, the VTC completes the second step by clearing all bits of the `invalid_bitmap` for reuse (step 2, line 24). The next step is remapping to update the log location of each entry in the database (step 3, lines 25—28). VTC traverses all the entries in the database and adjusts the value of `log_pos`, which has the location of the latest log. VTC determines how much to change the `log_pos` value of each entry by referring to the `delta` array filled in step 2. For example, if `log_pos` is n, VTC gets the value of delta[n] and then decreases `log_pos` by the value of delta[n]. After updating the log location of entry, if the entry is lazily invalidated, the VTC reflects it in the `invalid_bitmap` (step 3, lines 29—31). Lazy invalidation will be discussed in more detail in the next section. When the above steps are completed, the VTC completes compaction by releasing the temporary array, deleting old log file, and renaming the new log file (step 4, lines 32—34).

The VTC's handling of delete logs is different from write logs. Because the delete log is not invalidated by other logs, VTC does not refer to the `invalid_bitmap` when removing the delete log. Instead, the delete log can be removed when all other logs for the same entry are removed through checkpointing. The VTC determines whether to remove the delete log by comparing the delete log's timestamp with the compaction timestamp of each file. If the timestamp value of the delete log is smaller than the compaction timestamps, the VTC removes it. Otherwise, the VTC should keep the delete log in the log file to ensure correct recovery (step 2, lines 11–15). We will explain the removal of the delete log in more detail in section III-F.

D. LAZY INVALIDATION

Even if the worker does not append logs to the log file where compaction is in progress, the logs in the target file may be invalidated by an update or deletion. This may cause problems in log management. For example, if the worker updates an `invalid_bitmap` to invalidate a log that has already been copied during the copying step, the information is lost due to the `invalid_bitmap` initialization at the end of the step. As a result, the log is not removed even by subsequent compaction. This inconsistency continues until the `invalid_bitmap` is rebuilt by replaying the logs on recovery.

We solve this problem by applying a lazy invalidation strategy. If the log in the file where compaction is in progress needs to be invalidated by an update or delete request, we delay it until compaction is complete. Before updating the entry, the worker checks to see if the old log of the entry belongs to a file that is undergoing compaction. If it is true, the worker sets the entry's lazy variable to 1 and does not change the `invalid_bitmap`. Instead, the VTC creates a new entry and adds it to the database. Lazy invalidation immediately releases the memory for the entry's key and value, thereby mitigating the increase in memory usage caused by entries that are delayed for deletion.

Lazily invalidated entries are dealt with in the VTC's remapping step. In the remapping step, when after the `log_pos` of the lazily invalidated entry is adjusted, the compactor notes the new location of the log with the `invalid_bitmap`, and the log is removed in the next compaction. The compactor completes the processing of lazy invalidation entries by setting the lazy variable of those entries to 2, indicating that they should be deleted by the worker later. If the compactor were to delete those entries, there could be a conflict with the worker's add entry or delete entry, which is why we give the worker the role of deleting those entries.

E. RECOVERY

As shown in Figure 2(b), the recoverer restores data by sequentially executing logs. In the VTC, we allocate loader and buffer per storage to maximize I/O bandwidth.

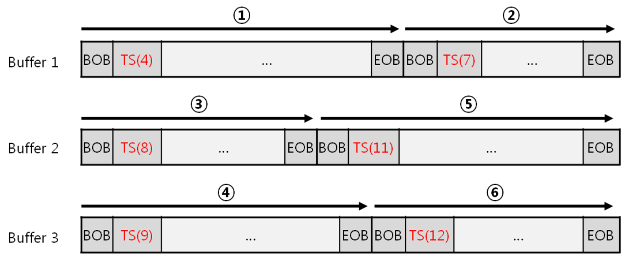


FIGURE 6. The recovery order (TS: timestamp).

The loaders read log blocks from their respective files and fill the designated buffers (①). The recoverer selects log blocks in timestamp order and recovers data by replaying the logs in the log blocks (②). As the recoverer inserts the key and value into the database, it also stores the location of the log read for compaction (③). If overwrite or delete occurs during recovery, the recoverer records the event in the `invalid_bitmap` (④). For example, as shown in Figure 6, the recoverer compares the timestamp of each buffer's first log block and selects the log block with TS(4) as the first recovery block. After completing the recovery of log block with TS(4), the recoverer determines the next log block to recover by comparing the timestamp of log block with TS(7) with others. In general, the use of timestamps causes an increase in recovery time. However, our recovery scheme uses a timestamp to determine the recovery order of log blocks. Since timestamp comparison is unnecessary while logs in a log block are sequentially executed, overhead due to timestamp use is insignificant.

When the recoverer encounters the write log, it checks whether an entry containing the same key exists in the database, and replays the log by selecting between insert and update. Therefore, a lookup is required for every log execution. To reduce the overhead caused by lookup, we divide the log into two groups based on whether a lookup is needed.

The smallest value among compaction timestamps in each log file is the criterion for separating the two groups. The front group logs are guaranteed to have no duplicate keys, so keys and values are inserted immediately, without lookup.

Furthermore, we apply optimizations for sorted sets [26] on recovery. Since sorted sets, a data type provided by Redis, must be stored in ascending order, they take a lot of time to restore. To optimize for this, we allocate a dedicated log buffer for the logs of sorted sets during recovery. The recoverer collects the logs that require sorting in a dedicated log buffer and batches them. This method helps reduce the recovery time by increasing the cache hit rate during sorting.

F. CORRECTNESS

Having explained how the VTC performs checkpointing by reconstructing log files using only the latest log for each entry, we will now provide proof of how the VTC guarantees correctness in all scenarios. The VTC maintains logs by entry, which is the smallest unit that the system can modify. This ensures that the system can restore data by replaying only the latest log of all entries. For instance, consider an entry A that is initially inserted with the value of a1, then updated to a2, and finally updated with the value of a3. These three write logs are stored across multiple log files. In this state, if a recovery proceeds due to system failure, the system executes three consecutive writes by referring to the timestamp of the log block. However, as a result, the final state of entry A is determined by the a3 update log, and the other two logs do not affect it. The VTC performs checkpointing individually for each file, and thus one or both of the a1 and a2 logs can be removed. Nevertheless, entry A can be restored correctly using the a3 update log remaining in the log file.

Next, we look at the process of the VTC removing the delete log. As we discussed, to properly remove the delete log, the VTC needs to confirm that all other logs of the deleted entry have been removed by checkpointing. If the VTC deletes the delete log without going through this process, at the time of recovery, entry A may be erroneously revived by a write log that may have remained in another log file. For example, consider the process in which VTC removes its logs after entry A is deleted. To remove the delete log of entry A, the VTC must first confirm that logs for three writes do not exist in other log files, which necessitates tracing all of the logs for each entry, incurring significant overhead. To avoid this, we use a compaction timestamp that represents the last checkpointing time for each file.

The VTC compares the delete log's timestamp with all compaction timestamps to determine removing the delete log of entry A. The timestamp of the delete log can be found by referring to the timestamp of the log block to which it belongs. If the delete log timestamp is less than the compaction timestamp from all of the log files, the VTC can safely remove the delete log because it is guaranteed that all three writes of entry A have been removed. Thus, in this case, no log for entry A remains, so no processing for entry A will occur during recovery. Conversely, if the compaction timestamp of any file has a value smaller than the delete log's timestamp, correct recovery can be guaranteed by maintaining the delete log for the corresponding entry. To do this, the VTC retains the delete log by copying it to a new file. In this case, the write log for entry A may be executed during recovery, but the delete log also remains, so entry A can be deleted and restored to the correct state.

G. IMPLEMENTATION

We implemented the proposed scheme on Redis 5.6.0. In Redis, write operations either create entries for new key-value pairs or update existing ones. Then write operations generate logs and write them to the log buffer. For these operations, we used the code path of Redis. To handle overwriting when the old log of an entry is stored in a file in which compaction is in progress, we insert codes for lazy invalidation, which sets a lazy variable and inserts a new entry instead of updating the entry. In addition, the entries contain the `log_pos` variable, which is 8 bytes in size, to keep track of their latest log. The upper 2 bits of the variable are reserved for the lazy invalidation of the entry. For the read operation we follow the Redis code path and add only the code to handle lazily invalidated entries. We also add functions for new algorithms and change the call path in order to replace the existing Redis algorithms such as logging, checkpointing, and recovery.

We allocate as many threads as the number of files (storage devices) to flush the log buffer. These threads are responsible for reading log blocks from a file during recovery; we also add one thread for checkpointing. We only allow workers to add or remove entries in the database. This restriction prevents performance degradation due to contention between the worker and the compactor. Additionally, we use atomic operations to ensure atomicity for some variables that are shared between threads. We count the number of logs and the number of invalidation processes for each file for checkpointing with an appropriate frequency, and checkpointing is triggered when the number of invalidated logs and their ratio to total logs reach the thresholds. The user can determine thresholds in consideration of checkpointing frequency and execution time.

IV. EVALUATION

A. EXPERIMENTAL SETUP

In this section, we describe the experiments conducted to measure the performance of the proposed scheme under various conditions. We conducted all experiments using two machines as a client and server, each of which is equipped with an Intel Xeon W-2245 CPU running at 3.9 GHz; the CPU had 8 physical cores and 16 logical cores with hyper-threading and 32 GB of DRAM memory. The machines were connected through a 10 Gbps network. We used Samsung 860 PRO [25] SATA SSDs to store logs and checkpoints. Our scheme distributes logs to three SSDs, and Redis-AOF, which uses the existing logging scheme, stores logs in a single

SSD or a RAID-0 array with three SSDs. One additional SSD was used as a swap device to prevent the process from being killed by an out-of-memory killer. The machines ran Ubuntu 18.04.4 LTS distribution with the Linux kernel 4.15.0.

To demonstrate the efficiency of our scheme, we applied the VTC to Redis-5.0.6 and compared the VTC performance to the following Redis protocols:

Redis-RDB: No logging, and all of the data were periodically backed up to a file through checkpointing.

Redis-AOF: This records the log of all events that change the database and manages the size of the log file through periodic checkpointing.

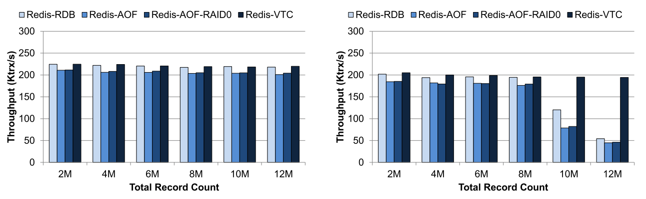
Redis-AOF with RAID-0 Setup: This has the same configuration as Redis-AOF except that it uses a RAID-0 array across the three SSDs handled through a software RAID driver in Linux.

TABLE 1. Parameters of YCSB workloads.

Parameters	Setting
Record Count	2M, 4M, 6M, 8M, 10M, 12M
Update proportion	10%, 30%, 50%, 70%, 90%
Record Size	default
Distribution	zipfian
Number of threads	128

To evaluate the performance of each scheme, we used Yahoo! Cloud Serving Benchmark (YCSB) [14] as the target workload. Table 1 summarizes the parameters of YCSB used for throughput evaluation. In order to evaluate the performance of each scheme with various configurations, we changed the number of records and the update proportions of workload A (mix of 50% update and 50% read), as shown in the table. After loading YCSB data into Redis, we measured the performance while the YCSB workload was running. For fair comparison we forced checkpointing at the same time. If the checkpointing time increased rapidly due to swap, we measured the performance for up to 600 seconds.

We disable the RDB compression option for a fair comparison because our prototype does not currently support data compression.

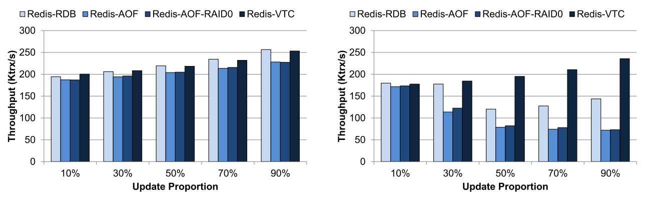


(a) Normal Throughput (b) Throughput during Checkpointing

FIGURE 7. Throughput for varying record count (50% update proportion).

B. THROUGHPUT

Figure 7 shows the throughput of each system for the YCSB workload with various record counts. As shown in Figure 7(a), Redis-VTC offers an average of 8% higher throughput than Redis-AOF with logging and similar throughput as Redis-RDB using only checkpoints. This means that Redis-VTC has little overhead for logging processing. Also, as the throughput of AOF-RAID0 is similar to that of Redis-AOF, we can see that the performance of the storage device does not affect the system throughput. Overall, for all of the schemes, the throughput was not appreciably affected by the size of the data set. However, throughput during checkpointing tends to be different for each system depending on the size of the data set. Figure 7(b) shows the throughput of the YCSB workload for a 50% update proportion while checkpointing was performed in the background. When the system memory was sufficient (a record count of 8M or less), each scheme exhibited a similar trend to normal throughput. Conversely, when the size of the data set increased, there was a difference in results that corresponded to the checkpointing scheme. In contrast to Redis-VTC, which showed stable throughput regardless of data set size, other schemes had severely degraded throughput when the size of the data set was larger than about 60% of system memory. Specifically, systems that use fork-based snapshots for checkpointing increase memory usage by CoW when processing an update request from a client. If the record count is 10M and the update proportion is 50%, more than 10 GB of memory is used by CoW during checkpointing. Eventually, swap due to insufficient system memory causes throughput degradation. In contrast, Redis-VTC performs checkpointing based on the validity of the logs and thus requires only a small amount of additional memory.



(a) Normal Throughput (b) Throughput during Checkpointing

FIGURE 8. Throughput for varying update proportion (10M records).

Figure 8(b) shows the throughput for varying update proportions with a record count of 10M. As shown in Figure 8(a), a high update proportion usually has a positive effect on performance. However, when the system memory is marginal, a high update proportion causes frequent CoW during checkpointing, which can cause swap. This means that using fork-based snapshots requires more extra system memory for update-intensive workloads.

C. MEMORY FOOTPRINT

Figure 9(a) shows the memory footprint of the data set with varying record counts. Redis-VTC had a 3.5% larger memory footprint than other schemes because variables for managing logs were added for each entry. However, during checkpointing, Redis-VTC required less additional memory than other protocols. As shown in Figure 9(b), during

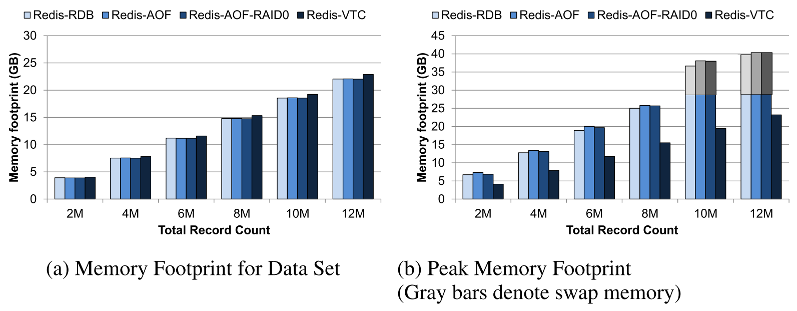


FIGURE 9. Memory footprint comparison with existing schemes (50% update proportion).

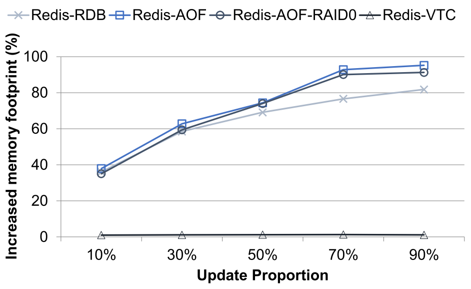


FIGURE 10. Increased memory footprint (8M records).

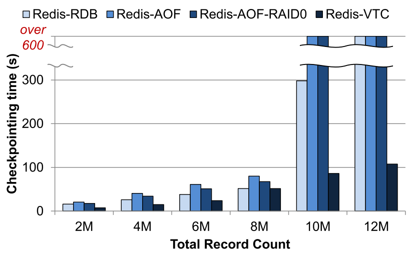


FIGURE 11. Checkpointing time.

checkpointing, the memory footprint of Redis-RDB and Redis-AOF increased by 69.6% and 79.7%, respectively, while the memory increase of Redis-VTC is less than 2%. Figure 10 shows that this gap can be wider as the update proportion is increased. This is because the higher the update proportion, the more CoW that occurs during checkpointing, which consumes more memory.

Redis-VTC creates a temporary array to update the location of the logs during checkpointing, but this is insignificant compared to the size of the entire data set because it only requires 8 bytes per entry. However, with Redis-RDB and Redis-AOF, memory usage continuously increases from CoW during checkpointing. Moreover, the increased memory cannot be reclaimed until the process in charge of storing the snapshot is terminated.

D. CHECKPOINTING TIME

The VTC performs checkpointing independently for each log file, but in order to have a fair comparison, we measured time by sequentially performing checkpointing for all log files. Redis-RDB was also configured not to use incremental-fsync option as Redis-VTC uses `fsync()` only once, after the last write when saving checkpoints to a file for optimization. However, this option was enabled in Redis-AOF because logging can be affected by latency spikes, and enabling this option increases the checkpoint time by about 20% because `fsync()` is called for every 32 MB write.

Figure 11 shows the time taken for checkpointing with varying record counts. In all schemes, under conditions of sufficient memory (record counts of less than 6M), the processing time increased as the size of the data set rose. We found that Redis-VTC required less than half the time of the other schemes under this condition. Because Redis-VTC performs checkpointing through file-to-file copy, it requires more I/O than other schemes to read the logs from storage device. However, if the system memory is sufficient, Redis-VTC can read logs to be copied from the buffer cache. Furthermore, a simple way to determine which log to copy by bitmap reference reduces checkpointing time.

We can see that the processing time for all three schemes increases as memory becomes insufficient. In particular, for schemes other than Redis-VTC, checkpointing time increases rapidly by swap. In this case, we only plotted up to 600 seconds, but checkpointing would normally take ten or more minutes. Redis-VTC also takes more time for checkpointing if the record count is 8M or more, because some logs are read from disk due to insufficient memory used as the buffer cache. However, Redis-VTC alleviates the increase in processing time because it reads the contents of a file sequentially, without random access.

E. FILE SIZE

Redis allows insert or update requests with set data [26] that include several sub-key and value pairs in the key. Because Redis-VTC has to manage the validity of logs by entry, it records the set data as separate logs for each sub-key/value pair. In this case, the file size increased because each separate log must include the parent key's information. For this reason, the YCSB workload, which uses bundled requests for data set loading before performance measurement, is not good for Redis-VTC in terms of file size.

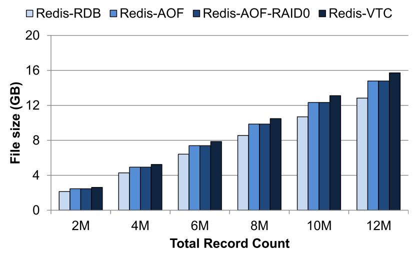


FIGURE 12. **File size.**

Figure 12 is the result of measuring the file size with a varying record count for the three schemes. We measured the file size when the YCSB workload ran the workload at a 50% update rate for 60 seconds after loading the data. The file sizes of Redis-VTC and Redis-AOF were measured before checkpointing. The file size of all schemes increases in proportion to the record count. For the reasons described above, the file size of Redis-VTC was, on average, 22% larger than Redis-RDB and 6% larger than Redis-AOF. For all schemes other than Redis-RDB, which does not use logging, the file size may be larger depending on the checkpointing interval and running time.

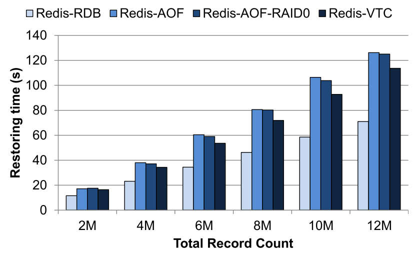


FIGURE 13. **Restoring time.**

F. RESTORING TIME

Figure 13 shows the restoring time by record count. We measured the data recovery time under the same conditions as measuring the file size. Redis-VTC took longer by an average of 53% to restore as compared to Redis-RDB. As described in the result of the file size, because the set data are logged separately, Redis-VTC executes each log independently for recovery. Conversely, Redis-RDB reduces the recovery time by handling the data set all at once. Moreover, as a property of checkpointing using snapshot, all data with the same parent key are stored together. This enables fast sorting with a high cache hit rate when recovering the sorted data set, which is one of the data types used by the YCSB workload. According to the results, Redis-VTC had 10% faster recovery time than Redis-AOF. However, if recovery was attempted after checkpointing, Redis-AOF recovered as quickly as Redis-RDB. Although our scheme takes more time to restore than other schemes with consistent snapshots, we believe that reducing memory usage and maintaining stable throughput are preferable for IMDBs. Moreover, data restoration may not occur frequently.

V. DISCUSSION

As we discussed, the VTC reads logs from the log file for checkpointing. Since the storage device has a relatively long read latency than the main memory, VTC may increase the checkpointing time compared to the conventional method. Therefore, we considered a method of copying valid logs from one SSD to another using two SSDs for fast checkpointing. However, this method requires an extra SSD, and one SSD is always idle except for the checkpointing period. In general, considering that the checkpointing time is short compared to the overall system operating time, this causes a waste of storage resources. Finally, we chose to handle checkpointing within a single SSD and instead use only sequential reads to suppress the increase in checkpointing time. Also, fortunately, when the system memory is sufficient, the read operations to the storage device do not occur thanks to the buffer cache supported by the OS.

We conducted experiments in a single-node environment, but we believe VTC can be applied to other environments like multi-nodes or flash arrays. When a distributed in-memory database is used, each node performs logging operations in an orthogonal manner. Therefore, VTC can be applied to each node without major modifications. Flash arrays can hide the internal flush delay of flash storage. Therefore, we expect that even if log flush and checkpointing are processed simultaneously in the flash array, performance similar to that of separating them using multiple SSDs is expected. We plan to study the effectiveness and scalability of VTC for various classes of systems in future work.

VI. RELATED WORK

In this section, we epitomize the techniques required to provide persistence in IMDBs. There have been several studies to provide logging suitable for IMDBs. The fast processing speed of IMDBs makes the run-time overhead of traditional logging relatively large. To avoid the throughput degradation caused by logging, many IMDBs [8], [27] attempt lightweight logging based on logical logging. Command logging [28] used by H-Store is a logical logging variant that records a single log record including only a procedure ID and input parameters. While redo-only logging such as command logging reduces run-time overhead, it takes more time to recover because logs are replayed for recovery.

Adaptive logging [29] is a logging method that focuses on the balance between run-time and recovery performance. It extends command logging to distributed systems and allows all nodes to perform recovery in parallel. Moreover, It reduces the recovery time by re-executing only transactions related to the failed node through dependency analysis between transactions. Taurus [30] and SiloR [31] also use distributed logging to solve the performance bottleneck problem caused by logging. Taurus performs logging in parallel using a log sequence numbers vector to manage transaction dependency. Taurus performs logging in parallel and manages the dependency of logs scattered in several files using a log sequence numbers vector. SiloR allocates one thread for each storage to write and flush logs in parallel. The system performs group commit in the epoch using optimistic concurrency control.

Our logging system is in line with these approaches [29]–[31], in terms of storing logs across multiple storage devices. However, in general, single-stream logging records are logged in one storage device, but our scheme improves durability by distributing logs across multiple storage devices. Furthermore, our scheme avoids logging and checkpointing I/O from affecting each other by separating them into different storage devices.

An efficient checkpointing method has been continuously proposed by several studies. Systems for some applications are sufficient to guarantee durability with only periodic checkpoints. However, most systems use a combination of logging and checkpointing to minimize data loss due to system failure. The checkpointing method employed by many IMDBs is to use consistent snapshots. It takes a consistent snapshot of the in-memory and stores its contents in stable storage. Representative consistent snapshot algorithms include naive snapshot [32], copy-on-update(COU) [2], [21], Zigzag [22], and PingPong [22]. Naive snapshot stops the system for a consistent snapshot. It is the simplest algorithm, but it is not suitable for in-memory database systems that need to process transactions even during checkpointing. COU is the most widely used algorithm for non-blocking checkpointing, and a number of variants have been studied. In contrast to the general COU algorithm using physical page shadowing, SIREN [21] proposes a COU algorithm based on tuple units smaller than pages. Small duplication granularity has the effect of reducing memory usage, but the average latency may increase due to tuple-level locking. Algorithms using `fork()` are applied to many IMDBs [7], [8] because it can easily implement COU with OS support, but there are problems with latency spike and memory increase. DMC [11] uses a memory dump to overcome the increase in memory usage, returning the page to the OS before the checkpoint is complete. However, at high update proportions, this scheme also requires a significant memory footprint.

Incremental or partial checkpointing reduces cost by limiting the amount of data processed at one time. Hekaton [12] creates a checkpoint file from the transaction logs not covered by a previous checkpointing and manages updates or deletions by recording them in delta files. The incremental checkpoints used by Hekaton can lower the cost by creating a checkpoint file only for new transactions. In contrast, a large number of files and a high ratio of deleted contents in the checkpoint file degrades recovery performance. To alleviate this, an additional process such as merging between checkpoints files is required. CALC [13] supports partial checkpointing. It performs checkpointing, including only records that have changed since the most recent checkpoint. It is effective in workloads where updates are not frequent, whereas in the opposite case, it may be inefficient in comparison to complete checkpointing due to overhead for merging files.

These systems that checkpoint data in main memory require page copying or version control, which causes increased memory usage. However, our scheme consumes less memory than the existing checkpointing scheme because it performs checkpointing using the log in the file. This enables efficient use of memory, the most important resource for IMDBs.

VII. CONCLUSION

This paper proposed an effective transaction log-based persistence scheme for IMDBs. Our key idea is to distribute logs across multiple storage devices and use log file compaction for checkpointing. The use of multiple storage devices improves the durability of the log without sacrificing system throughput. The log file compaction by managing log validity can keep the peak memory usage lower than the scheme using consistent snapshots. We implemented and evaluated our scheme in a famous IMDB, Redis, and the experimental results show that our proposed scheme can more stably manage memory during checkpointing compared to the scheme currently applied to commercial IMDBs. This is very desirable for IMDBs, because it allows the memory reserved for peak usage to be used for data storage.

REFERENCES

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

[2] M. Vaz Salles, T. Cao, B. Sowell, A. Demers, J. Gehrke, C. Koch, and W. White, ''An evaluation of checkpoint recovery for massively multiplayer online games,'' *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 1258–1269, Aug. 2009.

[3] B. K. Park, W. W. Jung, and J. Jang, ''Integrated financial trading system based on distributed in-memory database,'' in *Proc. Conf. Res. Adapt. Convergent Syst. (RACS)*, Oct. 2014, pp. 86–87.

[4] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, ''ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging,'' *ACM Trans. Database Syst.*, vol. 17, no. 1, Mar. 1992, pp. 94–162.

[5] L. Li, G. Wang, G. Wu, and Y. Yuan, ''Consistent snapshot algorithms for in-memory database systems: Experiments and analysis,'' in *Proc. 34th IEEE Int. Conf. Data Eng.*, Apr. 2018, pp. 1284–1287.

[6] L. Li, G. Wang, G. Wu, Y. Yuan, L. Chen, and X. Lian, ''A comparative study of consistent snapshot algorithms for main-memory database systems,'' *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 2, pp. 316–330, Feb. 2019.

[7] (2021). *Redis*. [Online]. Available: http://Redis.io/

[8] A. Kemper and T. Neumann, ''HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots,'' in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 195–206.

[9] Fork. (2021). *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Fork_(system call)

[10] (2021). *Redis Latency Problems Troubleshooting*. [Online]. Available: http://redis.io/topics/latency

[11] J. Park, Y. Lee, H. Y. Yeom, and Y. Son, ''Memory efficient fork-based checkpointing mechanism for in-memory database systems,'' in *Proc. 35th ACM Symp. Appl. Comput.*, Mar. 2020, pp. 420–427.

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

[13] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson, ''Low-overhead asynchronous checkpointing in main-memory database systems,'' in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2016, pp. 1539–1551.

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

[15] F. Faerber, A. Kemper, P. Larson, J. Levandoski, T. Neumann, and A. Pavlo, "Main memory database systems," in *Proc. Conf. Found. Trends Databases*, Aug. 2017, pp. 1–130.

[16] T. Wang, R. Johnson, and I. Pandis, "Query fresh: Log shipping on steroids," *Proc. VLDB Endowment*, vol. 11, no. 4, pp. 406–419, Dec. 2017.

[17] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann, "ScyPer: Elastic OLAP throughput on transactional data," in *Proc. 2nd Workshop Data Anal. Cloud*, Jun. 2013, pp. 11–15.

[18] M. Stonebraker and A. Weisberg, "The VoltDB main memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, Jun. 2013.

[19] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan, "Dali: A high performance main memory storage manager," in *Proc. Int. Conf. VLDB*, vol. 94, Sep. 1994, pp. 48–59.

[20] H. V. Jagadish, A. Silberschatz, and S. Sudarshan, "Recovering from main-memory Lapses," in *Proc. Int. Conf. VLDB.*, vol. 93, Aug. 1993, pp. 391–404.

[21] A.-P. Liedes and A. Wolski, "SIREN: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases," in *Proc. 22nd Int. Conf. Data Eng. (ICDE)*, Apr. 2006, p. 99.

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

[23] (2021). *DB-Engines Ranking*. [Online]. Available: http://db-engines.com/en/ranking

[24] (2021). *Redis Administration*. [Online]. Available: https://Redis.io/topics/admin

[25] (2021). *Samsung SSD 860 PRO*. [Online]. Available: https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/860pro

[26] (2021). *Redis Data Types*. [Online]. Available: https://Redis.io/topics/data-types

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

[28] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker, "Rethinking main memory OLTP recovery," in *Proc. IEEE 30th Int. Conf. Data Eng.*, Mar. 2014, pp. 604–615.

[29] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu, "Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 1119–1134.

[30] Y. Xia, X. Yu, A. Pavlo, and S. Devadas, "Taurus: Lightweight parallel logging for in-memory database management systems," *Proc. VLDB Endowment*, vol. 14, no. 2, Oct. 2020, pp. 189–201.

[31] W. Zheng, S. Tu, E. Kohler, and B. Liskov, "Fast databases with fast durability and recovery through multicore parallelism," in *Proc. 11th USENIX Symp. Oper. Syst. Design. Implement. (OSDI)*, Oct. 2014, pp. 465–477.

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

KWANGJIN LEE received the B.S. degree in mechanical and industrial system engineering from Kyung Hee University, South Korea, in 2003. He is currently pursuing the M.S. degree in computer science and engineering with Seoul National University. From 2013 to 2018, he was a Senior Engineer with the Memory Division, Samsung Electronics, South Korea, where he is a Principal Engineer. His research interests include storage systems, distributed systems, and database systems.

HWAJUNG KIM received the B.S. degree in computer science and engineering from Pohang University of Science and Technology, South Korea, in 2006, and the M.S. degree from the Department of Computer Science and Engineering, Seoul National University, South Korea, in 2018, where she is currently pursuing the Ph.D. degree in computer science and engineering. From 2006 to 2018, she was a Research Engineer with Samsung Electronics. Her research interests include operating systems, distributed systems, and database systems.

HEON Y. YEOM (Member, IEEE) received the B.S. degree in computer science from Seoul National University, in 1984, and the M.S. and Ph.D. degrees in computer science from Texas A&M University, in 1986 and 1992, respectively. He worked with Texas A&M Transportation Institute, as a Systems Analyst, from 1986 to 1990, and Samsung Data Systems, as a Research Scientist, from 1992 to 1993. He joined the Department of Computer Science, Seoul National University, in 1993, where he currently teaches and researches on distributed systems and transaction processing, and a Professor with the School of Computer Science and Engineering.