Single-pass restore after a media failure

Caetano Sauer
TU Kaiserslautern
Germany
csauer@cs.uni-kl.de

Goetz Graefe
Hewlett-Packard Laboratories
Palo Alto, CA, USA
goetz.graefe@hp.com

Theo Härder
TU Kaiserslautern
Germany
haerder@cs.uni-kl.de

Abstract: When persistent storage fails, traditional media recovery first restores an old backup image followed by replaying the recovery log since the last backup operation. Restoring a backup can take hours, but log replay often takes much longer due to its random access pattern. We introduce single-pass restore, a technique in which restoration of all backups and log replay are performed in a single operation. This allows hiding log replay within the initial restore of the backup, thus substantially reducing the time and cost of media recovery and, incidentally, rendering incremental backup techniques unnecessary.

Single-pass restore is enabled by a new organization of the log archive, created by a continuous process that is easily incorporated into the traditional log archiving process. Our empirical analysis shows that the imposed overhead is negligible in comparison with the substantial benefits provided.

1 Introduction

With "big data" ever increasing in size as well as individual storage devices ever increasing in capacity, failure and recovery of storage becomes an increasing concern, at least among people with operational responsibility for large databases. Restoring a transactional database on a replacement device requires not only copying a full database backup but also replaying log records captured during the backup operation and the hours or days hence. Traditional log replay incurs many random I/O operations on the replacement device and thus often takes much longer than restoring a full backup. Thus, database software often supports and database administrators often employ incremental backups, e.g., daily backups of database pages modified since the last backup.

The incremental backup approach does not eliminate the core problem, which is the random I/O incurred on database pages during log replay. Rather, it tries to alleviate the issue by restricting the length of log replay. The problem is that incremental backups are cumbersome to maintain—both in terms of implementation effort and processing overhead. Furthermore, as transaction throughput increases due to modern hardware, log volumes are expected to grow much faster. This requires incremental backups to be taken more frequently, which not only disturbs transaction processing activity but also quickly becomes ineffective, as frequently updated pages are backed up over and over. Therefore, in case of data losses, system administrators are left with no choice but to pay the extremely high penalty of traditional media recovery.

1.1 Media recovery costs

As an example scenario for predicting the time required for media recovery, consider a database storage device of 1 TB. At 150 MB/s, a full backup or a restore operation takes about 2 hours. If 5% of all database pages change over a day, the size of a daily incremental backup is 50 GB, or 6.25 million pages of 8 KB. The restore operation for each incremental backup requires (at 1 ms average access time per page) 6,250 seconds or 1 hour and 45 minutes. Assuming that there are two log records per modified page, that the buffer hit ratio during log replay is 75%, and that the miss penalty is a random I/O operation of 4 ms, then each day of log replay takes ∼12,500 seconds or 3.5 hours. Therefore, a media failure occurring 7 days after the last full backup may take almost 16 hours to recover (2h full backup + 6 × 1.75h incremental backups + 3.5h log replay). For around-the-clock online businesses, these prospects are frightening. For some practical scenarios, this example may even be considered conservative—if data volumes and transaction throughput are much higher, recovery may easily reach the scale of multiple days.

The inefficiency of traditional log replay is emphasized in Figure 1. We measure the time required to replay a log produced by ∼7.5 million transactions, i.e., one day worth of log under a load of ∼5k transactions per minute. In a realistic scenario where the device capacity is orders of magnitude larger than the amount of main memory, single-pass restore replays all updates in 7 minutes, as opposed to almost 4 hours in the traditional setting. This experiment is described in more detail in Section 4.

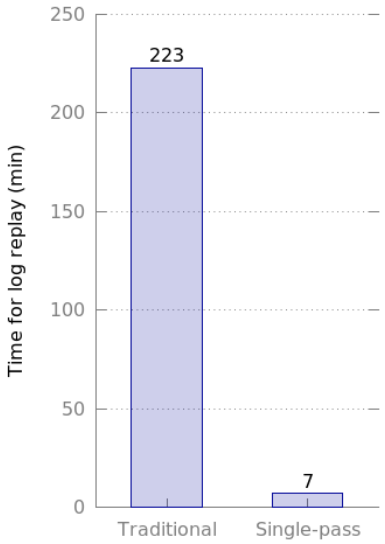


Figure 1: Time to perform log replay from a traditional vs. a sorted log

As a remedy for this situation, enterprises may choose the unconventional approach of backing up data on flash devices (or even on non-volatile RAM) to speed up recovery time, essentially decreasing the time to repair and thus improving availability. However, as observed in our previous work [SGH14], the fundamental problem is the random I/O pattern which is inherent to traditional "redo" recovery (from both system and media failures). Therefore, low-latency devices are (like incremental backups) not a solution to the problem but merely a temporary remedy—as transaction throughput increases due to new hardware and software techniques, such "fast" storage devices are expected to become the bottleneck once again. What is needed, therefore, is a software solution, preferably a simple one that requires only small incremental changes to existing techniques.

1.2 Our contribution

Single-pass restore is a novel technique that provides much faster log replay and hence media recovery. This is achieved by simply reorganizing the log into a different sort order while performing log archiving, i.e., copying the recovery log from a latency-optimized to a capacity- and cost-optimized device. With log replay practically free, incremental and differential backups lose their justification. In the example scenario described above, recovering a failed device to the most up-to-date state can take as long as simply restoring an outdated full backup, i.e., $\sim$2 instead of 16 hours.

The complexity of the new logic for log archiving and for restore operations is comparable to that of external merge sort, i.e., quite moderate. We demonstrate empirically that the overhead during normal transaction processing is negligible in a normal setting, and even in extremely fast OLTP scenarios it is as low as 1.5%.

In the remainder of this paper, Section 2 reviews related prior work, both competing approaches and technology adapted in the proposed data structures and algorithms. It also lays out some of the assumptions of our current prototype design. Section 3 introduces the new data structure for the log archive as well as algorithms for log archiving and for single-pass restore operations. Section 4 reports on our prototype implementation and its performance. Section 5 extends the basic techniques in order to overcome the simplifying assumptions. Section 6 offers a summary and some conclusions.

2 Related prior work

We divide related prior work into competing approaches, for which the new techniques may serve as alternatives, and adopted technology, upon which the proposed algorithms and data structures rely.

2.1 Competing approaches

Multiple techniques reduce the probability of a media failure or the duration of a media recovery. Among the former, hardware techniques such as disk mirroring [BG88] and RAID [PGK88] hide some failures from the operating system and all applications. Nonetheless, both mirroring and RAID can experience data loss. For those cases, recovery in the data management software remains required. Moreover, the new techniques do not require redundancy in the data store and thus save hardware costs and related costs, e.g., energy. Hot spares and replication have similar costs and dangers of data loss as mirroring and RAID. Nonetheless, mirroring is a suitable technology for the recovery log and RAID (e.g., RAID-6) is a suitable technology for the log archive.

Among the latter, incremental backups require a special in-database data structure for tracking database pages changed between backups. This data structure is expensive to

maintain during transaction processing [MN93] and very difficult to switch over in online incremental backup operations. In contrast, the new techniques hide days or weeks of log replay within the restore operation of the full backup, i.e., all effort related to incremental backups render the traditional restore techniques slower than the proposed ones. Since we consider incremental backups to be the main competing approach to single-pass restore, we provide a qualitative comparison between both approaches in Section 6.

In cases of single-page failures, traditional recovery requires media recovery. Some systems support selective media recovery, i.e., a full log scan with log replay limited to specific pages, e.g., Microsoft SQL Server [Mic14]. Some systems may also support dedicated and efficient single-page recovery [GK12] in the future. In that sense, single-page recovery is a competitor to media recovery if the recovery log embeds backward pointers not only per transaction but also per database page. However, both solutions still suffer from the random I/O pattern of traditional "redo" recovery.

Recent main-memory database systems make use of logical logging to eliminate the overhead of generating log records for every update [MWMS14]. The techniques proposed here apply to physiological or physical logging, as employed in ARIES [MHL+92], and therefore in the vast majority of database systems deployed in production businesses today. Essentially, we rely on the concept of pages as a fundamental characteristic of logging and recovery: a page is the smallest unit of fault containment and repair. Logical recovery schemes, such as those of in-memory databases, have no such concept. A comparison between logical and physiological logging techniques is beyond the scope of this paper.

2.2 Adopted technologies

The proposed techniques rely on various existing technologies. The most important among them is write-ahead logging [GR93] with physiological log records [MHL+92]. The new techniques rely on reliable and efficient access to log records, including (where single-page recovery [GK12] is used) efficient access to the history of a single database page using appropriate backward pointers among log records in the recovery log.

Log archiving frees up space on latency-optimized stable storage by copying to cost- and capacity-optimized long-term stable storage. The new techniques merely add some additional logic to log archiving, which remains a single-pass process, i.e., it reads and writes each log record only once.

Log records in a recovery log are sorted on time, in the order they are generated by executing transactions. Sorting log records differently is an old idea. For example, Gray wrote "For disk objects, log records can be sorted by cylinder, then track then sector then time" and "Since it is sorted by physical address, media recovery becomes a merge of the image dump of the object and its change accumulation tape" [Gra78]. The new techniques sort log records but not at the expense implied by Gray's design as sketched by the quoted sentences. Instead of applying sorting as a pre-processing step prior to actual data recovery, we propose maintaining a partially sorted log archive during normal operation, i.e., as part of the log archiving process.

Sorting the log is essentially an external merge sort operation, which is executed in multiple phases: run generation fed by a scan or an input process, some (zero or more) intermediate merge steps, and a single final merge step feeding directly the user of the sorted data, e.g., a merge-join algorithm. The new techniques run the traditional phases of external merge sort (input and run generation, intermediate merge steps, final merge step and production of output), but not one phase immediately after another.

2.3 Assumptions

The initial description of the proposed techniques relies on several simplifying assumptions; Section 5 alleviates most of these. Our initial assumptions include that (i) there is a single failure only; (ii) the database system and its transaction manager remain active throughout failure and recovery, i.e., throughout media loss and restore; yet (iii) the system stops transaction processing during a restore operation; and (iv) the latency-optimized stable storage for the recovery log has limited capacity such that long-term online transaction processing requires log archiving with or without the new techniques. Furthermore, we assume that (v) media recovery only requires "redo" recovery. This means that either backup operations are offline, i.e., each backup is transaction-consistent; or that uncommitted updates can be rolled back by a following crash recovery process, which feeds exclusively from the normal recovery log instead of the log archive. In practice, the latter case would be more common.

3 Partially sorted log archives

The new data structure which enables single-pass restore is a partially sorted log archive. The log archive is partially sorted because it is a collection of sorted runs instead of a single sorted file. Within each run, log records are sorted by page identifier (within the database) and by LSN. Figure 2 shows a sample recovery log for which four runs were generated in the archive. To simplify the example, we consider only three database pages, illustrated in three different shades. Due to the nature of physiological logging, only log records with page identifiers, i.e., those representing modifications, are selected for archiving. This is aligned with the assumption that the log archive supports only "redo", whereas "undo" relies on the active recovery log.

The process of generating the partially sorted log archive, including the sorting algorithm to use, is discussed in Section 3.1 below. To perform single-pass restore, all runs must be merged into a single sorted input stream. This process is discussed in Section 3.2. Finally, since failures may happen during any phase of both archiving and restore, we describe a restart mechanism that guarantees atomicity of archive and restore operations in Section 3.4.

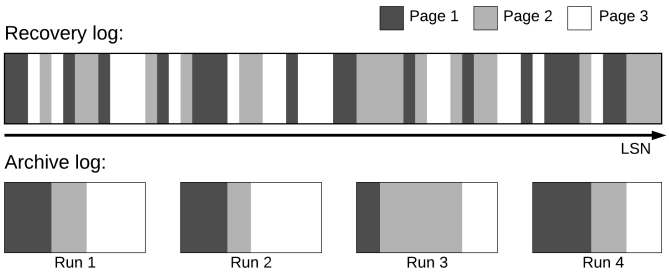


Figure 2: Visualization of partially sorted log archiving from the normal recovery log.

3.1 Log archiving logic

Traditional log archiving consists of copying log records from a latency-optimized device, where the recovery log is kept, into a capacity- and cost-optimized device. A latency-optimized device, such as a flash device or even non-volatile memory, allows more efficient transaction processing in which the log can be flushed at a higher ratio (in terms of I/O operations per second) and thus more transactions are committed per second. The log archive device, on the other hand, provides higher capacity and lower cost in terms of dollar per gigabyte. Thus, it is usually a hard disk, potentially replicated with RAID. The use of tape is discouraged in our approach, because external merge sort would be prohibitively slow.

Instead of copying "raw" log data, e.g., by copying files between file systems or physical blocks between devices, log archiving processes a log record at a time. This is done in order to compress the log archive by removing unnecessary log records such as transaction begin and commit/abort, checkpoint information, page flushing activity, etc. In essence, only log records that modify physical data, i.e., pertaining to a page, are kept. Furthermore, additional compression techniques can be applied such as removing "undo" information, combining updates into a "net change" record, or skipping aborted updates, as done in ARIES with the technique of restricted repeating of history [MP91]. Such inspection and manipulation of log records entails a certain CPU overhead to the process of log archiving. A partially sorted log archive increases the CPU effort by additionally sorting log records.

For example, in order to generate a run of 1 M log records, each log record participates in $\sim$20 comparisons (i.e., $log(n)$). In a fully tuned implementation, this amounts to 1,000 instruction cycles. Assuming the average log record size is 100 bytes, the process requires 10 CPU cycles per byte sorted. Thus, in order to maintain a sorting throughput of 200 MB/s, which is the typical sequential write speed of enterprise hard disks, 2 billion cycles per second, or 1 whole CPU core, must be reserved to the sorting procedure. This may seem like a large overhead at first, but typical OLTP workloads rarely fully utilize the CPU [TPK$^+$13], meaning that sorting can exploit idle CPU cycles, thereby causing little to nonexistent impact on transaction processing. In Section 4, we present experiments that quantify this overhead on a real system.

The sorting algorithm used to generate runs also plays an important role. While quicksort performs quite well in terms of CPU utilization, it exhibits a periodic load-sort-store behavior, in which a chunk of data is first loaded into the sort workspace, then fully sorted, and finally written to external memory in a single pass. The disadvantage of this behavior is that it does not allow read and write operations to occur in parallel if the output device is different from the input device, which is the case in log archiving. Furthermore, I/O operations cannot overlap with the CPU effort of sorting. For these reasons, we propose the replacement-selection algorithm [Gra06], which exhibits a continuous behavior where read and write activity can occur in parallel.

One advantage of replacement selection is that it can generate runs larger than the in-memory sort workspace. This is done by holding log records in the selection tree in order to optimize the sorting distance within each run [Knu73]. However, we do not make use of this feature in our prototype implementation, because the resulting runs cannot be mapped to contiguous regions of the input, which is the recovery log. If such mapping is not provided, it becomes cumbersome to keep track of regions already archived, which is essential for resuming the log archiving process after a system crash, as we discuss in Section 3.4 below. In order to maintain this simple mapping with replacement selection, we simply eliminate the logic that places incoming log records into the current or the next run depending on their key value. Instead, log records are always assigned to a run number greater than the current root of the selection tree, regardless of their key values.

Despite influencing the behavior of the log archiving process, the choice of sorting algorithm for run generation is orthogonal to the algorithms of single-pass restore. Therefore, an implementation based on quicksort, or any other internal sorting algorithm, is equally conceivable.

3.2 Media restore logic

Restoring a database device after a media failure can occur in two different ways, depending on how the latest full backup is made available. If the backup is maintained in a disk on hot stand-by, then only log replay on this device is required. If the backup must first be fetched from a separate (potentially slower but cheaper) device, then log replay is performed on individual pages as they are restored from the backup. Regardless of which scenario is considered, the logic for log replay is the same. Therefore, we focus on the first use case: replaying the log on an existing backup copy. Later on, we discuss how log replay can be incorporated on the initial backup restore.

Log replay for single-pass restore must be executed on one page at a time in sequential order. Therefore, it requires the existing log archive runs to be merged into a single sorted input stream. Since runs are partitioned by LSN and, thus, by time, it is possible to merge only runs newer than the oldest LSN found in the backup. We omit such details in this discussion. A traditional multi-way external merge can be used to that end, and, at this point, we assume that enough memory is available to merge all runs in a single pass. Later on, we discuss techniques to alleviate this assumption.

Since log records are sorted by page identifier, the database can be restored one page at a time, by replaying all relevant updates of a page in a single read/write cycle. The end effect is that the database is restored from the backup in a single pass, thus exploiting the device sequential bandwidth for restore. A traditional log replay, on the other hand, applies log records to pages in an arbitrary order of page identifiers, which requires a substantial amount of random I/O operations. Furthermore, the efficiency of single-pass restore does not depend on the size of the buffer pool, whereas the performance of traditional log replay degrades for smaller buffer pools. These hypotheses are verified empirically in Section 4.

If the backup must first be copied from a separate capacity-optimized device such as tape or network-attached storage (which is the most common case in practice), then log replay can be executed on each page as it is copied. This corresponds essentially to a merge-join operation between the backup and the log archive, as noted in the early work of Gray [Gra78]. If the total log volume is less than the size of the database, then an up-to-date, fully operational database can be restored in the same time it takes traditional restore to simply copy an old database backup. Therefore, it is again a single-pass process, on both the log and the backup.

In some cases, the log volume to be replayed can be greater than the size of the device. This can happen on small OLTP databases with very high update frequency. In this case, the cost of single-pass restore is bounded by the size of the log, and not by the backup size. The same situation occurs in a merge-join algorithm: the cost is determined by the largest input. The point which must be emphasized is that in both situations (smaller or larger log), media recovery is performed as a single-pass, sequential operation. Note that traditional media recovery performs much worse if the log is larger. We analyze these situations empirically in Section 4.

3.3 Asynchronous merging

In order to maintain a manageable number of runs in the log archive, thereby enabling a single merge pass for log replay, an asynchronous merge daemon can be used to merge runs in the background during normal operation. The daemon can run at any specified pace and using as little memory as made available by the system. It can be scheduled to run during relatively idle periods, or to run continuously, but at a slow, unobtrusive pace. Our prototype implementation includes a simple merge daemon, which we evaluate empirically in Section 4.

If, during restore, there is not enough memory to merge the log archive in a single pass, despite proper asynchronous merging policies, a slower form of restore (but probably still much faster than traditional restore) is still possible. One obvious option is to perform offline merge steps until the number of runs decreases to the amount required by a single pass. In this procedure, the smallest runs should be merged first. Because these are probably the most recently generated runs, they will correspond to a very small percentage of the overall log archive volume, and, therefore, the added delay should be relatively small. For example, assume a scenario with 120 runs amounting to a total log archive volume of

1 TB. Despite the large overall volume, most of the runs, say, 80 of them, will probably be initial runs, i.e., runs generated directly from the sort workspace of log archiving. If the maximum merge fan-in is 100, simply merging 21 of these small runs is enough in order to perform single-pass restore. If we assume that the small runs are 1 GB in size, the offline merge incurs only a 4% delay (2% for each read and write) on the total cost of log replay. Note, however, that such scenario should be very unlikely in practice due to asynchronous merging.

3.4 Restart logic for log archiving and merging

Tolerating failures during the archiving process, for both the log and database backups, is an essential requirement. A proper implementation must not only restart the archiving process correctly, but it should also minimize the amount of work that must be repeated (e.g., copying log records multiple times). In our approach, such requirements are of utmost importance, since archiving is always active as an inherent part of normal transaction processing. This is in contrast to incremental backups, which run periodically in a fashion similar to fuzzy checkpoints [MN93].

In order to restart the log archiving and asynchronous merging daemons after a system failure, the filesystem must provide certain guarantees in order to generate runs (both initial and merged) in atomic steps. Since initial runs are generated by a continuous-output sorting algorithm—in this case replacement selection—, log records are first written into a temporary file. Once the run is finished, the file is renamed into its permanent format, which contains the LSN boundaries of the log records contained in it. This means that runs are mapped to contiguous regions of the recovery log (as discussed in Section 3.1) by means of file naming conventions. The end boundary of each file is an exclusive one, meaning that the log archive can be checked for consistency (i.e., absence of "holes") by simply comparing the end of one run with the begin of the next.

The contiguous mapping of runs to LSN ranges provides an easy and safe way of determining the "low water mark" of log archiving [GR93]. If traditional replacement selection is used instead of our adapted version, determining this mark can be quite cumbersome. Not only that, the amount of effort lost would be higher, since retaining log records in the selection tree essentially blocks the advance of the low water mark.

In order to generate initial runs atomically, file renaming must be provided as an atomic operation. Fortunately, this is the case for most common UNIX filesystems [GLI14]. Upon restart, the temporary file corresponding to the run that was being generated at the time of failure can simply be deleted, and log archiving restarts from the end boundary of the last generated run. Unfortunately, there is no simple way to avoid losing the whole temporary run, because that would require information about the in-memory selection tree, which is lost in a crash[1]. Note, however, that log archiving may be resumed while the system recovers, perhaps even in coordination with the log analysis or "redo" phases. Therefore, it is likely that the lost run will be restored by the time the system comes back up.

[1] Other sorting methods such as quicksort unavoidably lose the current run, since it is kept entirely in memory.

Asynchronous merging works in a similar way: runs are produced in a temporary file and renamed when completed. One crucial requirement is that only adjacent runs may be merged, in order to keep the contiguous mapping to LSN ranges. One additional concern in this case is that, after the file is renamed, the input runs must be deleted, which cannot be done atomically. Therefore, upon restart, the system must check all file names for overlapping LSN ranges, deleting the smaller runs it finds. This is essentially a "redo" operation that guarantees atomicity of run generation, in conjunction with the "undo" step of deleting temporary runs.

4 Experiments

4.1 Environment and prototype implementation

We implemented the basic algorithms for partially sorted log archiving and single-pass restore in Shore-MT [JPH+09], an open-source transactional storage manager which scales well on multi-core CPUs. In order to generate workload, we executed the TPC-C benchmark using the Shore-Kits package[2], which implements several standard benchmarks on top of Shore-MT. Our code is made available in an online open-source repository[3].

Being a research prototype, our system lacks some important features that would be expected on a product-level implementation. For instance, it does not support multiple devices for database pages or additional replication measures to support multiple failures. It also does not support the detection and automatic repair of media failures—single-pass restore must be invoked explicitly and while the system is offline. Furthermore, it does not implement an online backup utility—all experiments rely on backups generated explicitly also while the system is offline. However, it is important to note that these limitations are simply due to implementation effort and not conceptually inherent to the design.

The experiments described here were carried out on an Intel Xeon X5670 server with 96 GB of 1333 MHz DDR3 memory. The system provides dual 6-core CPUs with hyper-threading capabilities, which gives a total of 24 hardware thread contexts. The operating system is a 64-bit Ubuntu Linux 12.04 with Kernel version 3.11.0. Unless noted otherwise, the log archive and database backups are kept on a high-capacity hard disk.

4.2 Hypotheses under test

Our experiments are designed to test the following hypotheses:

1. In single-pass restore, the cost of log replay can be hidden in the initial phase of copying a full database backup, leading to substantially faster recovery.

[2] `http://bitbucket.org/shoremt/shore-kits`
[3] `http://bitbucket.org/caetanosauer`

2. Contrary to traditional media recovery, the time required to perform single-pass restore is independent of the amount of memory available.

3. Log archiving with run generation can be executed concurrently with high-throughput transaction processing with negligible impact on performance.

When analyzing the results of the experiments below, we refer back to these hypotheses and show that our method successfully accomplishes the goals laid out therein.

4.3 Media recovery performance

The first-glance results provided in Figure 1 demonstrate the potential gains of maintaining a partially sorted log archive. This section analyzes the performance of single-pass restore in more detail. First, we analyze the total time to perform media restore, with the goal of testing Hypothesis 1. Then, we provide a detailed analysis of log replay costs in isolation, as performed for Figure 1, but this time varying the amount of main memory available.

The restore procedure based on a sorted log archive is called "single pass" because it allows the failed device to be restored one page at a time in sequential order. The algorithm for single-pass restore is essentially a merge join between a full database backup and the log archive (i.e., the stream of log records produced by a merge of all existing runs). Since a merge join requires a single pass over each input, the device can be restored in $O(n)$ time, where n is the number of pages to be restored. In traditional restore, copying an outdated database backup alone already requires $O(n)$ I/O operations. Thus, we compare single-pass restore with copying a full backup in terms of total execution time. If Hypothesis 1 is correct, single-pass restore should be slower due to the additional merge logic, but only by a negligible margin.

In order to test this hypothesis, we performed an experiment where database files of different sizes are restored. We generate TPC-C databases of exponentially increasing scale factors, starting from 32 and up to 512, which yield sizes of 4 to 65 GB, respectively. For single-pass restore, we consider a log with about 50%–60% the size of the full backup. Note that the cost of a merge join depends only on the larger input, and so the log volume is only relevant if it is greater than the database size. As discussed in Section 3.2, even if the log is much larger than the device capacity, restore is still a single-pass operation, but, in this case, the cost is bound by the log size. Figure 3 shows the results of this experiment. Note that both axes are in logarithmic scale.

As the results confirm, there is only a marginal difference on every scale factor considered. This shows that the cost of log replay is indeed completely hidden in the process of restoring a full backup, as predicted by Hypothesis 1.

The experiment does not consider new pages which are allocated during replay of the log. Such pages can be either restored directly in the buffer pool or written to the replacement device in the same single-pass process. In the latter case, a slightly larger margin—dependent on workload characteristics—would be observed for the results above. Note that this is still much more efficient than restoring new pages during log replay, as done in

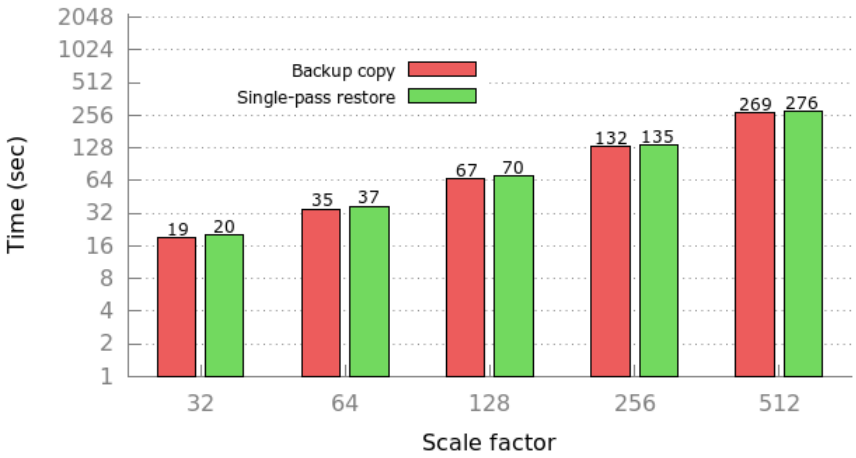


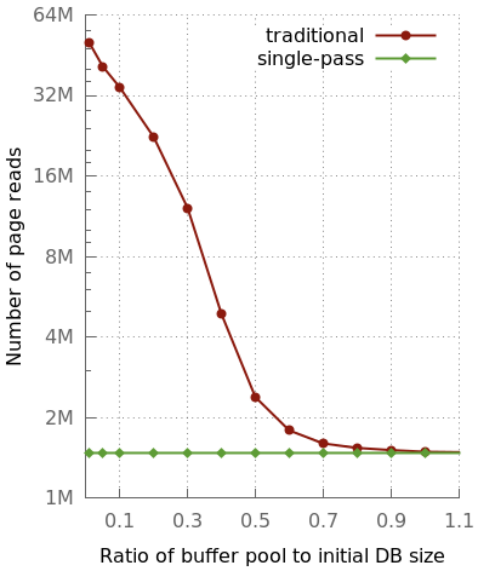
Figure 3: Time to perform single-pass restore vs. copy outdated full backup

traditional media recovery. Nevertheless, we emphasize the fact that the times measured for single-pass restore are for up-to-date recovery of device contents, whereas the baseline is just restoring an outdated backup, i.e., the long-running phase of log replay is still required afterwards. As we demonstrate on the following experiments, traditional log replay can be orders of magnitude more expensive than replay of a partially sorted log archive.

The cost for log replay in traditional restore is more complex to predict, since it depends on the amount of memory available and buffer replacement policy. If the complexity of buffer replacement is abstracted, the cost is $O(nm)$, where m is the average number of times each page gets replaced in the buffer pool. Since m grows very fast as the ratio between buffer size and device capacity decreases, so does the total cost of log replay. A sorted log archive, on the other hand, can be replayed using roughly the same number of I/O operations regardless of buffer pool size. To demonstrate these facts empirically—in support of Hypothesis 2—we compare the number of page reads observed with varying buffer pool sizes. For this experiment, we consider a log volume of 48 GB, which is generated by $\sim$7.5 million transactions, and a database with scale factor 64 (8 GB). Such discrepancy between log volume and database size is chosen deliberately in order to emphasize the costs of log replay.

Figure 4 shows the results for this experiment. As predicted by Hypothesis 2, the number of page reads required for log replay in single-pass restore remains constant. As the table on the right shows, there is actually a very small variation of up to 3 page reads in single-pass restore. This is a limitation of the Shore-MT recovery algorithms, and it can occur if the same slot on the disk is reused by pages being released and allocated multiple times. If the replay algorithm can properly detect when a page is being reformatted, such rare superfluous reads can be avoided.

Traditional log replay, on the other hand, varies drastically as the buffer size changes. For a buffer of 1% of the device size, 50 million random page reads are required, as opposed



Ratio	Traditional	Single pass
0.01	50,335,811	1,472,945
0.05	41,020,158	1,472,945
0.1	34,283,852	1,472,945
0.2	22,373,075	1,472,945
0.3	12,155,559	1,472,945
0.4	4,879,686	1,472,944
0.5	2,387,143	1,472,942
0.6	1,791,567	1,472,942
0.7	1,601,986	1,472,943
0.8	1,542,317	1,472,943
0.9	1,513,885	1,472,942
1.0	1,491,613	1,472,942

Figure 4: Number of page reads performed during log replay for varying buffer pool sizes

to 1.4 million sequential reads in the single-pass scenario. If the backup is stored in a hard disk with average read latency of 4 ms, traditional log replay would require approximately 56 hours. Single-pass restore, in this case, would be bound by the log size, which is much larger than the database size in this experiment. Even then, assuming that a read bandwidth of 150 MB/s can be sustained (i.e., typical of modern desktop hard drives), the log of 48 GB could be replayed in 3½ minutes.

4.4 Impact on transaction processing

The second set of experiments analyzes the impact of log archiving on normal transaction processing, in order to test Hypothesis 3. We compare four different scenarios with different archiving configurations. The baseline system has all log archiving features turned off, meaning that transactions can be executed at full speed without any interference. It is an impractical scenario because it does not support media recovery, but it gives a general baseline to any kind of archiving method. The second scenario represents traditional log archiving, i.e., without sorting. It requires some CPU overhead to process individual log records for suppressing log records which are irrelevant for media recovery, e.g., "undo" log records, checkpoints, transaction begin and commit/abort, etc. In the last two scenarios, partially sorted log archiving is performed, first with asynchronous merging turned off and lastly with this feature turned on.

To measure transaction processing performance in the four scenarios, we performed 30 TPC-C runs of five minutes each on a warm buffer. The environment was set up to provide

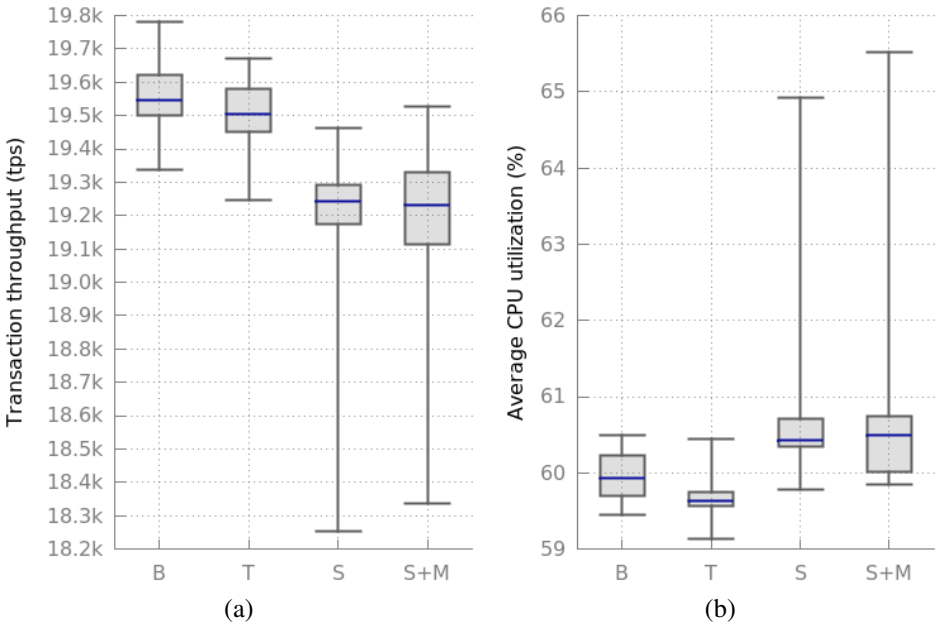


Figure 5: Transaction throughput (a) and CPU utilization (b) on 30 repetitions of each scenario

maximum transaction throughput: all CPU contexts are utilized, the complete dataset fits in the buffer pool, and the recovery log is kept in RAM. Logging to volatile memory is obviously incorrect from a transactional perspective, but it permits a worst-case analysis of the interference of log archiving, because I/O bottlenecks are eliminated and CPUs are better utilized. It also provides expectations for future non-volatile memory devices, for which write-ahead logging is a very suitable application. To provide measurements representative of current technology, we also provide results with the log on a flash device (Samsung SSD 840 Pro).

Figure 5 shows the transaction throughput (a) and the CPU utilization (b) observed in the 30 benchmark runs. Values on the y-axis represent the average transaction throughput achieved on a single 5-minute run, while the x-axis shows the four different scenarios: baseline (B), traditional archiving (T), and partially sorted archiving, both without (S) and with asynchronous merging (S+M). Note that the y-axis does not start on zero. The results are presented in a box plot format. The line in the middle of each box represents the median observation. Each box ranges from the 25-percentile on the lower boundary to the 75-percentile on the upper one. The lines extending below and above the boxes represent the minimum and maximum values observed, respectively. The box plot representation provides a summary of the statistical distribution of the observed throughput. It is therefore more useful than a simple average value.

The results show that partially sorted log archiving with asynchronous merging (S+M) has a median throughput of a little over 19.2 ktps, which is only about 1.5% less than traditional log archiving (T) at 19.5 ktps. However, it seems like partially sorted log archiving

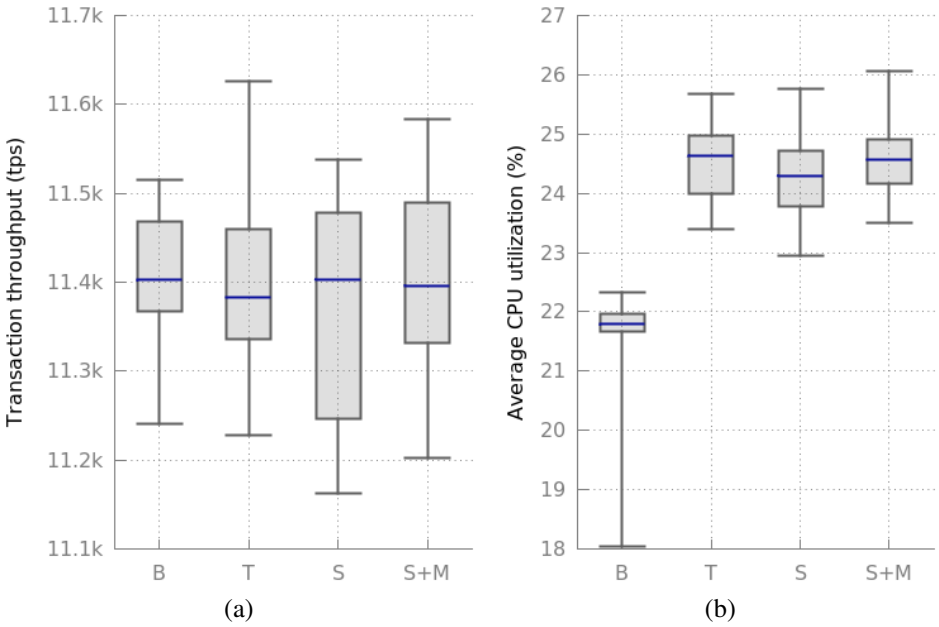


Figure 6: Transaction throughput (a) and CPU utilization (b) with log on an SSD device

produces more outliers on the lower side, as can be seen on the long bars extending below the quartile boxes. Such variation possibly originates from the randomness of page ID orders in the recovery log—less order in the input requires more swap operations in the sorting algorithm. Similar behavior can be observed on the CPU utilization results. The additional overhead of sorting translates directly into more CPU usage, again with a difference of approximately 1.5%. Note that the baseline actually has higher CPU utilization than traditional log archiving. This can be attributed to the fact that the baseline scenario has higher transaction throughput, probably because context switches are more costly than the actual computational effort in traditional log archiving.

Figure 6 presents the same analysis with the log on an SSD device. Contrary to the previous experiment, there is no observable difference between the four scenarios. This is expected because log I/O becomes the bottleneck, therefore leaving much more idle CPU time, which log archiving can exploit. This fact can be confirmed by comparing the CPU utilization of the baseline scenario with the rest, which is the only noticeable difference in the results. Note that CPU utilization fluctuates around 24%, as opposed to 60% in the previous scenario. The transaction throughput is also substantially lower.

Based on these results, we conclude that partially sorted log archiving only produces an observable impact on transaction processing performance if the I/O bottleneck is completely eliminated, which traditionally is not the case in database systems. Even if the system delivers main-memory performance for I/O, we showed that the overhead is usually as little as 1.5%, which is acceptable given the dramatical decrease on recovery time. These observations confirm Hypothesis 3 presented earlier.

Two final observations are important to conclude our analysis on transaction processing impact. First, as already hinted to above, the overhead depends on the amount of idle CPU cycles observed during the benchmark. If we consider a workload with much less logical contention, e.g., an insert-heavy benchmark such as TATP [TAT12], transactions running at full speed are able to better exploit the CPU and thus leave less room for log archiving. However, such workloads are more of a special case and not the general norm. Furthermore, an OLTP system running at full speed around the clock is an extremely uncommon scenario in real deployments. This leads to our second observation: log archiving can be scheduled to run at periods of lower transaction activity, and at an adjusted pace. Such policies are highly recommended in a product-level implementation. Nevertheless, our experiment shows that even at periods of peak activity, the overhead of log archiving can be considered negligible.

5 Extensions

The system for log archiving and media recovery as described so far is limited by the assumptions introduced above. The present section lifts some of these assumptions and introduces further promising extensions. The text here merely sketches opportunities and techniques. Future work will develop more detail on assumptions, conditions, algorithms, expected performance and scalability, tuning, etc.

5.1 Online backups

Online backups have been a standard industrial capability for a long time [Gra78]. While a backup operation is active, concurrent queries and update transactions may not only remain incomplete but may actively modify the database. Individual page images in the backup may or may not include updates by transactions concurrent to the backup operation.

After an offline backup, i.e., without concurrent transaction processing, all page images in the database backup have PageLSN values older than the start of the backup operation. After an online backup, some database pages in the back may have newer PageLSN values, but always older than the end of the backup operation. Restore from a backup taken with concurrent transaction processing must consider the PageLSN value of each database page and apply only new log records.

Thus, the log archiving logic must suspend log compression for the duration of the backup operation, i.e., combining log records. The restore logic must suppress log records preceding a page image obtained from a backup. Gray suggested to post-process each backup to obtain a transaction-consistent backup; in contrast, we propose that the required logic be in the restore operation.

5.2 Online restore

Ideally, a transactional data service remains available while one of its persistent devices or storage volumes is in media recovery. When a device first fails, if the buffer pool holds some pages of the failed device at the time of the failure, all those pages may be marked dirty immediately and do not require any further restore logic. While the database system recovers a failed device, transaction processing continues on all other devices. Incomplete transactions with updates on the failed device may, after the restore operation is complete, resume or roll back. Several of the following extensions relax this restriction.

5.3 Incremental media recovery

If the media recovery merges backup and (partitions of the) log archive in the order of database page identifiers, the replacement media can become available to transaction processing incrementally. New log records ought to capture the progress, i.e., indicate the restored and available page ranges on the replacement media. Transaction processing—including queries, updates, transaction rollback, and even restart after a crash—manages restored page ranges in the same way as media without failure or ongoing restore.

5.4 Multiple failures

During recovery from a media failure, other failures may occur. In an online restore operation, this includes transaction failures—rollback should be possible as much as forward processing. Single-page failures (other than ahead of the restore process) can proceed precisely in the same way as in the absence of concurrent media failure and restore operations.

A media failure of another device simply invokes another instance of the (single-pass) restore logic. A media failure of a replacement device simply requires another replacement device and a new invocation of the (single-pass) restore logic. A system failure (software crash) with an incomplete media recovery is the most complex case. It needs to resume the media recovery at a database page identifier definitely reached prior to the system failure. Thus, media recovery ought to log its progress occasionally as also suggested above for incremental media recovery and incremental availability.

5.5 Virtual backups and remote virtual backup

In order to obtain an up-to-date full database backup, it is not required to increase the load on a database server. Instead, merging an existing database backup and the runs in the log archive produces the same result. Such a virtual backup “creates a backup without taking a backup.”

If database transaction processing and recovery log are on one node in a network and backups, log archiving, and log archive are on a second node, then this second node can create a new, up-to-date backup without any load on the first node or the network between the nodes. A remote virtual backup has great advantages for provisioning the network, i.e., network bandwidth is required only for transaction processing but not for bursts due to across-the-network backup operations.

6 Comparison with incremental backups

In traditional systems, incremental backups are the standard technique employed to increase availability by decreasing time to repair on media failures. An incremental backup is simply the set of pages that changed since the last full or incremental backup was taken. As argued in this paper, partially sorted log archives render incremental backups completely unnecessary and thus obsolete. In this section, we provide a more in-depth discussion on the drawbacks of incremental backups as described by Mohan in the ARIES family of algorithms [MN93]. The discussion also applies to differential backups, which are taken always with respect to the last full backup.

Incremental backups require a special data structure to keep track of database pages that changed since the last backup. In order to achieve acceptable performance, access to this data structure must be done diligently, such that multiple updates to the same page in between backups only require one update on the data structure. In Mohan's method, a special transaction first collects a list of pages that must be included in the backup, resetting their state in the tracking data structure. Based on this list, the actual backup copy is performed. The LSNs generated by this transaction are used as heuristic to avoid updating the tracking state on every page update. The idea is that if the last update on a certain page (given by its PageLSN) happened after the last backup was taken, then the tracking state must have been already set by that last update. Thus, no extra access is required.

We identify two main problems with Mohan's approach. First, it adds complexity to the buffer pool logic, since the responsibility of maintaining the tracking data structure falls on normal page update operations. Our approach, in contrast, is completely independent from buffer pool operations, since log archiving is a separate process that feeds only from log data. This logical separation also implies less code complexity, less testing and maintenance effort, less concurrency control, and simpler recovery algorithms. Second, and most importantly, incremental backup techniques become ineffective if higher availability is desired. This is because frequent backups render the optimizations described by Mohan [MN93] mostly ineffective, since the ratio between updates to the tracking data structure and actual page updates increases. In contrast, partially sorted log archiving can be executed at a very aggressive pace without any logical interference on transaction processing activity. Only physical interference in the form of CPU and memory usage is expected, which may not only be an acceptable trade-off for availability, but can also be mitigated if log archiving is performed on a separate node in a network. This concern of logical separation was also raised in the original paper [MN93], which proposes a technique to

perform full backups directly from data pages on disk, i.e., without involving the buffer pool and thus not interfering with transaction processing. However, this technique is not available for incremental backups.

One argument in favor of incremental backups may be that a single copy of a frequently updated page may capture an arbitrarily large amount of updates since the last backup. Relying on the archive log for redundancy, on the other hand, means that additional space is consumed by log records of such hot-spot pages. However, as mentioned earlier, aggregating updates on the same database page into "net changes" may be easily incorporated in the processes of sorting the log (both during run generation and merging), thus reducing the log volume of hot-spot database pages.

7 Summary and conclusions

In summary, the partially sorted log archive slightly increases the cost of transaction processing, but reduces the duration of restore operations substantially. During transaction processing, log archiving partially sorts the log records, quite similar to run generation within an external merge sort. Restore operations merge a full database backup with runs in the log archive. For a small and controllable increase in system load during transaction processing, mean time to repair improves by a small factor, in some cases even an order of magnitude. Improving the mean time to repair by an order of magnitude adds another "9" to system availability, e.g., improving availability from 99.99% to 99.999%.

In conclusion, database backup and restore operations are ripe for innovation. The techniques introduced and measured above reduce the time for media recovery by a substantial factor. At the same time, with log replay practically free, there no longer is any advantage in taking differential and incremental backups. Thus, the data structures required to guide those partial backups are no longer required, simplifying database system implementation, quality assurance, and operations. Full backups no longer require access to the active database. In fact, clusters with multiple nodes may create new, up-to-date backups without access to the database server. In sum, single-pass restore based on the novel organization of the log archive may do away with all database backup operations as we know them today, in addition to speeding up restore operations and improving system availability.

Acknowledgments

We thank Pinar Tözün and Ryan Johnson for kindly and generously answering our questions about Shore-MT and Shore-Kits. Our gratitude is also extended to the Shore and Shore-MT development teams for making their code available as open source, without which this work would have been much harder to realize.

References

[BG88] Dina Bitton and Jim Gray. Disk Shadowing. In *Proc. VLDB*, pages 331–338, 1988.

[GK12] Goetz Graefe and Harumi A. Kuno. Definition, Detection, and Recovery of Single-Page Failures, a Fourth Class of Database Failures. *PVLDB*, 5(7):646–655, 2012.

[GLI14] GLIBC. The GNU C Library Reference Manual. Available at: `http://www.gnu.org/software/libc/manual/html_node/Renaming-Files.html`, 2014. Accessed: 2014-10-06.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra78] Jim Gray. Notes on Data Base Operating Systems. *Lecture Notes in Computer Science*, 60:393–481, 1978.

[Gra06] Goetz Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3):10, 2006.

[JPH+09] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. EDBT*, pages 24–35, 2009.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[MHL+92] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992.

[Mic14] Microsoft. Restore and Recovery Overview (SQL Server). Available at: `http://msdn.microsoft.com/en-us/library/ms191253.aspx`, 2014. Accessed: 2014-10-09.

[MN93] C. Mohan and Inderpal Narang. An Efficient and Flexible Method for Archiving a Data Base. *SIGMOD Rec.*, 22(2):139–146, June 1993.

[MP91] C Mohan and Hamid Pirahesh. Aries-RRH: Restricted repeating of history in the ARIES transaction recovery method. In *Proc. ICDE*, pages 718–727, 1991.

[MWMS14] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. In *Proc. ICDE*, pages 604–615, 2014.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *SIGMOD Rec.*, 17(3):109–116, June 1988.

[SGH14] Caetano Sauer, Goetz Graefe, and Theo Härder. An empirical analysis of database recovery costs. In *RDSS (SIGMOD Workshops), Snowbird, UT, USA*, 2014.

[TAT12] TATP. Telecom Application Transaction Processing Benchmark. Available at `http://tatpbenchmark.sourceforge.net/`, 2012. Accessed: Oct 2014.

[TPK+13] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In *Proc. EDBT*, pages 17–28, 2013.