Instant restore after a media failure (extended version)

Caetano Sauer[b,1], Theo Härder[a], Goetz Graefe[c]

[a] *TU Kaiserslautern, Germany*
[b] *Tableau Software*
[c] *Google, Madison, WI, USA*

Abstract

Media failures usually leave database systems unavailable for several hours until recovery is complete, especially in applications with large devices and high transaction volume. Previous work introduced a technique called single-pass restore, which increases restore bandwidth and thus substantially decreases time to repair. Instant restore goes further as it permits read/write access to any data on a device undergoing restore—even data not yet restored—by restoring individual data segments on demand. Thus, the restore process is guided primarily by the needs of applications, and the observed mean time to repair is effectively reduced from several hours to a few seconds.

This paper presents an implementation and evaluation of instant restore. The technique is incrementally implemented on a system starting with the traditional ARIES design for logging and recovery. Experiments show that the transaction latency perceived after a media failure can be cut down to less than a second. The net effect is that a few "nines" of availability are added to the system using simple and low-overhead software techniques.

1. Introduction

Advancements in hardware technology have significantly improved the performance of database systems over the last decade, allowing for throughput in the order of thousands of transactions per second and data volumes in the order of petabytes. Availability, on the other hand, has not seen drastic improvements, and the research goal postulated by Jim Gray in his ACM Turing Award Lecture of a system "unavailable for less than one second per hundred years" [1] remains an open challenge. Improvements in reliable hardware and data center technology have contributed significantly to the availability goal, but proper software techniques are required to not only avoid failures but also repair failed systems as quickly as possible. This is especially relevant given that a significant share of failures is caused by human errors and unpredictable defects in software and firmware, which are immune to hardware improvements [2]. In the context of database logging and recovery, the state of the art has unfortunately not changed much since the early 90's, and no significant advancements were achieved in the software front towards the availability goal.

Instant restore is a technique for media recovery that drastically reduces mean time to repair by means of simple software techniques. It works by extending the write-ahead logging mechanism of ARIES [3] and, as such, can be incrementally implemented on the vast majority of existing database systems. The key idea is to introduce a different organization of the log archive to enable efficient on-demand, incremental recovery of individual data pages. This allows transactions to access recovered data from a failed device orders of magnitude faster than state-of-the-art techniques, all of which require complete restoration of the entire device before access to the application's working set is allowed.

The problem of inefficient media recovery in state-of-the-art techniques, including ARIES and its optimizations, can be attributed to two major deficiencies. First, the media recovery process has a very inefficient random access pattern, which in practice encourages excessive redundancy and frequent incremental backups—solutions that only alleviate the problem instead of eliminating it. The second deficiency is that the recovery process is not incremental and requires full recovery before any data can be accessed—on-demand schedules are not possible and there is no prioritization scheme to make most needed data available earlier. Previous work addressed the first problem with a technique called single-pass restore [4], while this paper focuses on the second one.

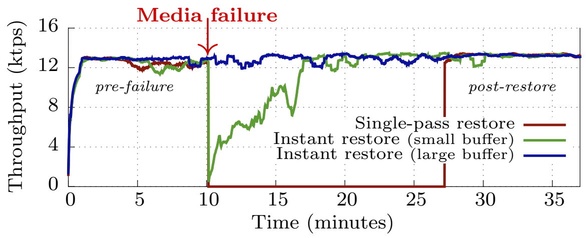


Figure 1: Effect of instant restore

The effect of instant restore is illustrated in Fig. 1, where transaction throughput is plotted over time and a media failure occurs after 10 minutes. In single-pass restore, as in ARIES, transaction processing halts until the device is fully restored (the red line in the chart), while instant restore continues processing transactions, using them to guide the restore process (blue and green lines). In a scenario where the application working set fits in the buffer pool (blue line), there is actually no visible effect on transaction throughput.

In the remainder of this paper, Section 2 describes related work, both previous work leading to the current design as well as competing approaches. Then, Section 3 describes the instant restore technique. Finally, Section 4 presents an empirical evaluation, while Section 5 concludes this paper.

A high-level description of instant restore was previously published in a book chapter [5] among other instant recovery techniques. This paper, which is an extension of an earlier conference publication [6], contributes with a more detailed discussion of the design and implementation aspects as well as an empirical evaluation of the technique with an open-source prototype.

Table 1: Failure classes, their causes, and effects

Failure class	**Loss**	**Typical cause**	**Response**
Transaction	Single-transaction progress	Deadlock	Rollback
System	Server process (in-memory state)	Software fault, power loss	Restart
Media	Stored data	Hardware fault	Restore
Single page	Local integrity	Partial writes, wear-out	Repair

2. Related work

2.1. Failure classes and assumptions

Database literature traditionally considers three classes of database failures [7], which are summarized in Table 1 (along with single-page failures, a fourth class to be discussed in Section 2.6). In the scope of this paper, it is important to distinguish between system and media failures, which are conceptually quite different in their causes, effects, and recovery measures. System failures are usually caused by a software fault or power loss, and what is lost—hence what must be recovered—is the state of the server process in main memory; this typically entails recovering page images in the buffer pool (i.e., "repeating history" [3]) as well as lists of active transactions and their acquired locks, so that they can be properly aborted. The process of recovery from system failures is called *restart*.

Instant restart [5, 8] is an orthogonal technique that provides on-demand, incremental data access following a system failure. While the goals are similar, the design and implementation of instant restore require quite different techniques.

In a media failure, which is the focus here, a persistent storage device fails but the system might continue running, serving transactions that only touch data in the buffer pool or on other healthy devices. If system and media failures happen simultaneously, or perhaps one as a cause of the other, their recovery processes are executed independently, and, by recovering pages in the buffer pool, the processes coordinate transparently.

The present work makes the same assumptions as most prior research on database recovery. The log and its archive copy reside on "stable storage", i.e., they are assumed to never fail. We consider failures on the database device only, i.e., the permanent storage location of data pages. Recovery from such failures requires a backup copy (possibly days or weeks old) of the lost device and all log records since the backup was taken; these may reside either in the active transaction log or in the log archive. The process of recovery from media failures is called *restore*. The following sections introduce some background techniques and briefly describe previous restore methods.

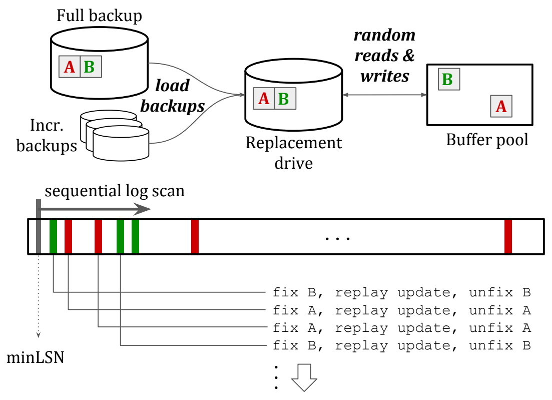
2.2. Logging

This paper is based on write-ahead logging as implemented in ARIES [3], most importantly the concept of *physiological logging*. With physiological logging, every logged update is associated with a database page, which is a physical unit of data storage and fault containment. Within a database page, the changes described in a log record may be logical; the insertion of a record in a B-tree page, for example, only needs to log the key and the record's contents rather than a delta of physical contents of the page. Physiological logging guarantees the important property of "recovery independence amongst objects" [3], which, in the case of instant recovery techniques, enables on demand recovery of individual pages or contiguous sets thereof.

The concept of *logical undo with compensation* is also a key component or ARIES on which the techniques in this paper rely. It dictates that undo actions must be the logical compensation of the original action; "logical" here implies granularity greater than a page, i.e., a table, index, or the whole database. For example, an insertion of a record on a table is undone as the deletion of that same record from that table, regardless of the effects on physical data structures—during undo, the record might have been moved to a different page due to a page split operation, which is logged as a *system transaction* [9] (also known as *top-level action* in ARIES [3]). These undo actions generate special redo-only log records known as *compensation log records* (CLR), which are processed like any other log record during redo but guide the undo logic in a way that guarantees *idempotence*, i.e., recovery actions are applied exactly once, even in the presence of system failures. The key advantage of this undo scheme is that it enables partial rollbacks, and, most importantly, *record-level locking*, i.e., concurrency control with granularity finer than a page.

2.3. ARIES restore

Techniques to recover databases from media failures were initially presented in the seminal work of Gray [10] and later incorporated into the ARIES family of recovery algorithms [3]. In ARIES, restore after a media failure first loads a backup image and then applies a redo log scan, similar to the redo scan of restart after a system failure. Fig. 2 illustrates the process, which we now briefly describe. After loading full and incremental backups into the replacement device, a sequential scan is performed on the log archive and each update is replayed on its corresponding page in the buffer pool. A global *minLSN* value (called "media recovery redo point" by Mohan et al. [3]) is maintained on backup devices to determine the begin point of the log scan.



```
fix B, replay update, unfix B
fix A, replay update, unfix A
fix A, replay update, unfix A
fix B, replay update, unfix B
```

Figure 2: Random access pattern of ARIES restore

Because log records are ordered strictly by LSN, pages are read into the buffer pool in random order, as illustrated in the restoration of pages A and

B in Fig. 2. Furthermore, as the buffer pool fills up, they are also written in random order into the replacement device, except perhaps for some minor degree of clustering. As the log scan progresses, evicted pages might be read again multiple times, also randomly. This mechanism is quite inefficient, especially for magnetic drives with high access latencies. Thus, it is no surprise that multiple hours of downtime are required in systems with high-capacity drives and high transaction rates [4].

Another fundamental limitation of the ARIES restore algorithm is that it is not incremental, i.e., pages cannot be restored to their most up-to-date version one-by-one and made available to running transactions incrementally. As shown in the example of Fig. 2, the last update to page A may be at the very end of the log; thus, page A remains out-of-date until almost the end of the log scan. Some optimizations may alleviate this situation (e.g., reusing checkpoint information), but there is no general mechanism for incremental restoration. Furthermore, even if pages could somehow be released incrementally when their last update is replayed, the hottest pages of the application working set are most likely to be released only at the very end of the log scan, and probably not even then, because they might contain updates of uncommitted transactions and thus require subsequent undo. This leads to yet another limitation of this approach: even if pages could be restored incrementally, there is no effective way to provide on-demand restoration, i.e., to restore most important pages first.

Despite a variety of optimizations proposed to the basic ARIES algorithm [3, 11, 12], none of them solves these problems in a general and effective manner. In summary, all proposed techniques that enable earlier access to recovered data items suffer from the same problem: early access is only provided for data for which early access is not really needed—hot data in the application working set is not prioritized and most accesses must wait for complete recovery.

Finally, commercial database systems that implement ARIES recovery suffer from the same problems. IBM's DB2 speeds up log replay by sorting log records after restoring the backup and before applying the log records to the replacement database [13]. While a sorted log enables a more efficient access pattern, incremental and on-demand restoration is not provided. Furthermore, the delay imposed by the offline sort may be as high as the total downtime incurred by the traditional method. As another example, Oracle attempts to eliminate the overhead of reading incremental backups by incrementally maintaining a full backup image [14]. While this makes recovery slightly more efficient, it does not address the deficiencies discussed earlier.

2.4. Replication

Given the extremely high cost of media recovery in existing systems, replication solutions such as disk mirroring or RAID [15, 16] are usually employed in practice to increase mean time to failure. However, it is important to emphasize that, from the database system's perspective, a failed disk in a redundant array does not constitute a media failure as long as it can be repaired automatically. Restore techniques aim to improve mean time to repair whenever a failure that cannot be masked by lower levels of the system occurs. Therefore, replication techniques can be seen largely as orthogonal to media restore techniques as implemented in database recovery mechanisms.

Nevertheless, a substantial reduction in mean time to repair, especially if done solely with simple software techniques, opens many opportunities to manage the trade-off between operational costs and availability. One option can be to maintain a highly-available infrastructure (with whatever costs it already requires) while availability is increased by deploying software with more efficient recovery. Alternatively, replication costs can be reduced (e.g., downgrading RAID-10 into RAID-5) while maintaining the same availability. Such level of flexibility, with solutions tackling both mean time to failure and mean time to repair, are essential in the pursuit of Gray's availability goal [1].

Many open-source and commercial database systems provide an additional level of replication in the form of a *hot stand-by* server [17, 18, 19, 20]. In principle, such solutions are orthogonal to media recovery approaches such as instant restore, in which only the database storage device is assumed to fail and not the entire server on which the database is running. However, a hot stand-by server can (and often is) used in practice to recover from a media failure on the primary server with relatively low mean time to repair, and thus we provide a brief overview below.

The main idea behind hot stand-by designs is to ship log records continuously from the primary database server to a secondary server, which maintains a fresh copy of the database by continuously replaying the received log records. These approaches are very costly, not only in terms of the computational power required to maintain a fresh database copy on a secondary server, but also in terms of the added network latency to the commit path of transactions (assuming durability is a requirement, i.e., synchronous replication). Instant restore makes hot stand-by servers much less appealing as a means to recover from media failures, because very similar levels of availability can be achieved with a single (primary) server and thus a significantly lower hardware investment.

Instant restore can also be combined with the related *instant restart* technique to provide *instant failover* [5], a recovery technique for server failures in networked environments that has substantially lower costs than traditional hot stand-by techniques.

2.5. In-memory databases

Early work on in-memory databases focused mainly on restart after a system failure, employing traditional backup and log-replay techniques for media recovery [21, 22]. The work of Levi and Silberschatz [23] was among the first to consider the challenge of incremental restart after a system failure. While an extension of their work for media recovery is conceivable, it would not address the efficiency problem discussed in Section 1. Thus, it would, in the best case and with a more complex algorithm, perform no better than the algorithm discussed later in Section 2.6.

Recent proposals for recovery on both volatile and non-volatile in-memory systems usually ignore the problem of media failures, employing the unspecific term “recovery” to describe system restart only [24, 25, 26]. Therefore, recovery from media failures in modern systems either relies on the traditional techniques or is simply not supported, employing replication as the only means to sustain service during storage hardware faults. As discussed above, while relying on replication is a valid solution to increase mean time to failure, a highly available system must also provide efficient repair facilities. In this aspect, traditional database system designs—using ARIES physiological logging and buffer management—provide more reliable behavior. Therefore, we believe that improving traditional techniques for more efficient recovery with low overhead on memory-optimized workloads is an important open research challenge.

2.6. Single-page repair

Single-page failures are considered a fourth class of database failures [27], along with the other classes summarized in Table 1. It covers failures restricted to a small set of individual pages of a storage device and applies online localized recovery to each individual page instead of invoking media recovery on the whole device. The single-page repair algorithm, illustrated in Fig. 3 (with backup and replacement devices omitted for simplification), has two basic requirements: first, the LSN of the most recent update of each page (i.e., the current PageLSN value) must be known without having to access the page; second, starting from the most recent log record, the system must be able to retrieve the complete history of updates to a page. The former requirement can be provided with a page recovery index—a data structure mapping page identifiers to their most recent PageLSN value. Alternatively, the current PageLSN can be stored together with the parent-to-child node pointer in a B-tree data structure [28]. The latter requirement is provided by per-page log record chains, which are straight-forward to maintain using the PageLSN fields in the buffer pool.

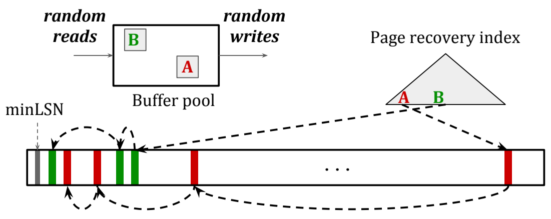


Figure 3: Single-page repair

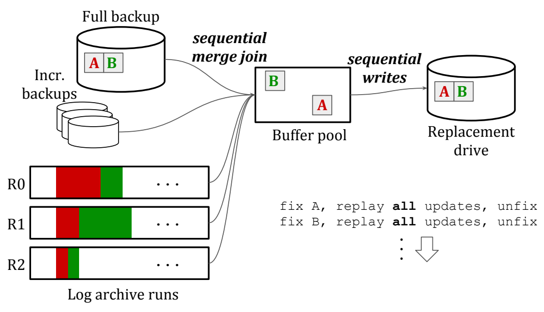
In principle, single-page repair could be used to recover from a media failure, by simply repairing each page of the failed device individually. One advantage of this technique is that it yields incremental and on-demand restore, addressing the second deficiency of traditional media recovery algorithms mentioned in Section 1. To illustrate how this would work in practice, consider the example of Fig. 3. If the first page to be accessed after the failure is A, it would be the first to be restored. Using information from the page recovery index (which can be maintained in main memory or fetched directly from backups), the last red log record on the right side of the diagram would be fetched first. Then, following the per-page chain, all red log records until *minLSN* would be retrieved and replayed in the backup image of page A, thus yielding its most recent version to running transactions.

While the benefit of on-demand and incremental restore is a major advantage over traditional ARIES recovery, this algorithm still suffers from the first deficiency discussed in Section 1—namely the inefficient access pattern. The authors of the original publication even foresee the application to media failures [27], arguing that while a page is the unit of recovery, multiple pages can be repaired in bulk in a coordinated fashion. However, the access pattern with larger restoration granules would approach that of traditional ARIES restore—i.e., random access during log replay. Thus, while the technique introduces a useful degree of flexibility, it does not provide a unified solution for the two deficiencies discussed.

2.7. Single-pass restore

Our previous work introduced a technique called single-pass restore, which aims to perform media recovery in a single sequential pass over both backup and log archive devices [4]. Eliminating random access effectively addresses the first deficiency discussed in Section 1. This is achieved by partially sorting the log on page identifiers, using a stable sort to maintain LSN order within log records of the same page. The access pattern is essentially the same as that of a sort-merge join: external sort with run generation and merge followed by another merge between the two inputs—log and backup in the media recovery case.

The idea itself is as old as the first recovery algorithms (see Section 5.8.5.1 of Gray's paper [10]) and is even employed in DB2's "fast log apply" [13]. However, the key advantage of single-pass restore is that the two phases of the sorting process—run generation and merge—are performed independently: runs are generated during the log archiving process (i.e., moving log records from the latency-optimized transaction log device into high-capacity, bandwidth-optimized secondary storage) with negligible overhead (at most 1% [4]); the merge phase, on the other hand, happens both asynchronously as a maintenance service and also during media recovery, in order to obtain a single sorted log stream for recovery. Importantly, merging runs of the log archive and applying the log records to backed-up pages can be done in a sequential pass, similar to a merge join. The process is illustrated in Fig. 4. We refer to the original publication for further details [4] as well as a related dissertation [8] for details on the run generation process with replacement selection.



```
fix A, replay all updates, unfix
fix B, replay all updates, unfix
```

Figure 4: Single-pass restore

Having addressed the access pattern deficiency of media recovery algorithms, single-pass restore still leaves open the problem of incremental and on-demand restoration. Nevertheless, given its superiority over traditional ARIES restore (see related publications [4, 5, 8] for an in-depth discussion), it is a promising approach to use as starting point in addressing the two deficiencies in a unified way. Therefore, as mentioned in Section 1, single-pass restore is taken as the baseline for the present work.

3. Instant restore

The main goal of instant restore is to preserve the efficiency of single-pass restore while allowing more fine-granular restoration units (i.e., smaller than the whole device) that can be recovered incrementally and on demand. We propose a generalized approach based on *segments*, which consist of contiguous sets of data pages. If a segment is chosen to be as large as a whole device, our algorithm behaves exactly like single-pass restore; on the other extreme, if a segment is chosen to be a single page, the algorithm behaves like single-page repair. As discussed in this section and evaluated empirically in Section 4, the optimal restore behavior lies somewhere between these two extremes, and simple adaptive techniques are proposed to robustly deliver good restore performance without turning knobs manually.

This section starts by introducing the log data structure employed to provide efficient access to log records belonging to a given segment or page; then, we present the restore algorithm based on this data structure. After that, performance expectations are addressed in a discussion of latency vs. bandwidth trade-off, which is followed by a discussion of how simultaneous recovery from multiple failures are coordinated, and, finally, a brief exposure of some key implementation issues.

3.1. Indexed log archive

In order to restore a given segment incrementally, instant restore requires efficient access to log records pertaining to pages in that segment. In single-page repair, such access is provided for individual pages, using the per-page chain among log records [27]. As already discussed, this is not efficient for restoration units much larger than a single page. Therefore, we build upon the partially sorted log archive organization introduced in single-pass restore [4].

In instant restore, the partially sorted log archive is extended with an index. The log archiving process sorts log records in an in-memory workspace and saves them into runs on persistent storage. These runs must then be indexed, so that log records of a given page or segment identifier can be fetched directly. Sorting and indexing of log records is done online and without any interference on transaction processing, in addition to standard archiving tasks such as compression.

In an index lookup for instant restore, the set of runs to consider would be restricted by the given *minLSN* (see Section 2.3) of the backup image, since runs older than that LSN are not needed. Furthermore, Bloom filters can be appended to each run to restrict this set even further. The result of the lookup in each indexed run is then fed into a merge process that delivers a single stream of log records sorted primarily by page identifier and secondarily by LSN. This stream is then used by the restore algorithm to replay updates on backup segments.

Multiple choices exist for the physical data structure of the indexed log archive. Ideally, the B-tree component of the indexing subsystem can be reused, but there is an important caveat in terms of providing atomicity and durability to this structure. A typical index relies on write-ahead logging, but that is not an option for the indexed log archive because it would introduce a kind of self-reference loop—updates to the log data structure itself would have to be logged and used later on for recovery. This self-reference loop could be dealt with by introducing special logging and recovery modes (e.g., a separate "meta"-log for the indexed log archive), but the resulting algorithm would be too cumbersome. In our prototype, we chose a simpler solution: each run of the log archive is maintained in its own read-only file; temporary shadow files are then used for merges and appends. In this scheme, atomicity is provided by the file rename operation, which is atomic in standard filesystems [29].

A thorough explanation of the indexed log archive and its implementation, as well as its applications in techniques beyond instant restore is provided in a related thesis [8].

3.2. Restore algorithm

When a media failure is detected, a *restore manager* component is initialized and all page read and write requests from the buffer pool are intercepted by this component. The diagram in Fig. 5 illustrates the interaction of the restore manager with the buffer pool and all persistent devices involved in the restore process: failed and replacement devices, log archive, and backup. For reasons discussed in previous work [4], incremental backups are made obsolete by the partially sorted log archive; thus, the algorithm performs just as well with full backups only. Nevertheless, incremental backups can be easily incorporated, and the description below considers a single full backup without loss of generality.

In the following discussion, the numbers in parentheses refer to the numbered steps in Fig. 5. The restore manager keeps track of which segments were already restored using a segment recovery bitmap, which is initialized with zeros. When a page access occurs, the restore manager first looks up its segment in the bitmap (1). If set to one, it indicates that

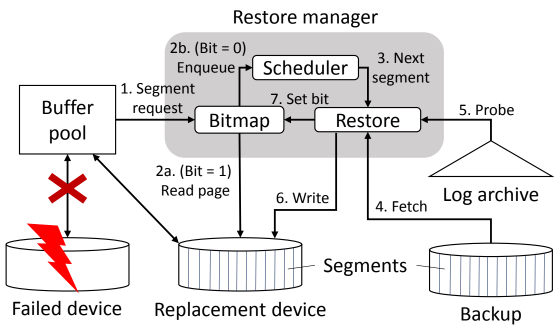


Figure 5: Instant restore flow chart

the segment was already restored and can be accessed directly on the replacement device (2a). If set to zero, a segment restore request is enqueued into a restore scheduler (2b), which coordinates the restoration of individual segments (3).

To restore a given segment, an older version is first fetched from the backup directly (4). This is in contrast to ARIES restore, which first loads entire backups into the replacement device and then reads pages from there [3]. This has the implication that backups must reside on random-access devices (i.e., not on tape) and allow direct access to individual segments, which might require an index if backup images are compressed. These requirements, which are also present in single-page repair [27], seem quite reasonable given the very low cost per byte of current high-capacity hard disks. For moderately-sized databases, it is even advisable to maintain log archive and backups on flash storage.

While the backed-up image of a segment is loaded, the indexed log archive data structure is probed for the log records pertaining to that segment (5); the results of each probe are merged to form a single sorted log stream. Then, log replay is performed to bring the segment to its most recent state, after which it can be written back into a replacement device (6).

Finally, once a segment is restored, the bitmap is updated (7) and all pending read and write requests can proceed. Typically, a requested page will remain in the buffer pool after its containing segment is restored, so that no additional read is required on the replacement device when a transaction accesses a restored page.

All read and write operations described above—log archive index probe, segment fetch, and segment write after restoration—happen asynchronously with minimal coordination. The read operations are essentially merged index scans—a very common pattern in query processing [30]. Writing a restored segment to the replacement device can also be done asynchronously, preferably by simply reusing the page write process of the buffer pool.

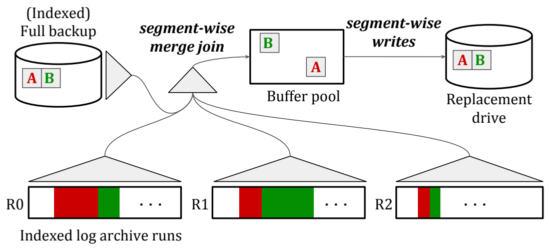


Figure 6: Access pattern of instant restore

The output of log archive merges performed during instant restore can be saved as new, higher-level runs in the log archive, thereby reusing the merge effort for future log archive accesses. Unlike a background merge operation, which merges whole runs at a time, the instant restore logic merges fragments of each input run pertaining to the segment(s) being restored. As such, an additional data structure is required to keep track of these fragments, the details of which we omit here. Note that with a partitioned B-tree [**?**], these fragments are simply an instance of a key-range merge, which is supported natively by the data structure without additional bookkeeping.

To illustrate the access pattern of instant restore, similarly to the diagrams in Section 2, Fig. 6 shows an example scenario with three log archive runs and two pages, A and B, belonging to the same segment. The main difference to the previous diagrams is the segment-wise, incremental access pattern, which delivers the efficiency of pure sequential access with the responsiveness of on-demand random reads.

Using this mechanism, user transactions accessing data either in the buffer pool or on segments already restored can execute without any additional delay, whereby the media failure goes completely unnoticed. Access to segments not yet restored are used to guide the restore process, triggering the restoration of individual segments on demand. As such, the time to repair observed by transactions accessing data not yet restored is multiple orders of magnitude lower than the time to repair the whole device. Furthermore, time to repair observed by an individual transaction is independent of the total capacity of the failed device. This is in contrast to previous methods, which require longer downtime for larger devices.

3.3. Latency vs. bandwidth trade-off

One major contribution of instant restore is that it generalizes single-page repair and single-pass restore, providing a range of choices of restore granularity between the two. In order to optimize restore behavior, the restore manager must adaptively and robustly choose the best option within this range. In practice, this boils down to choosing the correct granularity of access to both backup and log archive, in order to balance restore latency and bandwidth.

Restore latency is defined as the additional delay imposed on the page reads and writes of an individual transaction due to restore operations. Hence, it follows that if a single page can be read and restored in the same time it takes to just read it, the restore latency is zero—this is the “gold standard” of restore performance and availability. For a single transaction, restore latency can be reduced by setting a small segment size—e.g., a single page. However, this is not the optimal behavior when considering average restore latency across all transactions. Therefore, restore bandwidth, i.e., the number of bytes restored per second, must also be optimized. The optimized restore behavior is such that in the beginning of the restore process, pages which are needed more urgently should be restored first, so that restore latency is decreased; towards the end, less and less transactions must wait for restore, so the system can effectively increase restore bandwidth while a low restore latency is maintained. To adaptively choose a restore unit that optimizes for either latency or bandwidth when applicable, a small segment size (e.g., 1 MB) is chosen, so that individual segments can be restored quickly, but multiple adjacent segments are restored together when there is no pending restoration request in the restore manager. This way, low restore latency is prioritized in the initial phase of restore, where transactions are more likely to trigger segment restorations; during the end phase, on the other hand, the portions of the device that are not accessed frequently by running transactions are restored in bulk, with large sequences of adjacent segments.

It is also worth noting that devices with low latency and inherent support for parallelism, e.g., solid-state drives, make these trade-offs less pronounced. This does not mean, however, that instant restore is any less significant for such devices—a point which is emphasized in the next two paragraphs.

As discussed earlier, previous restore techniques suffered from two deficiencies: inefficient access pattern and lack of incremental and on-demand recovery. Solid-state devices shorten the efficiency gap between restore algorithms with sequential and random access, but this gap will never be entirely closed, especially considering the locality and predictability of sequential access, and thus its proneness to prefetching.

As for the second deficiency, low-latency devices directly contribute to the reduction of restore latency, because the time to recover a single segment is reduced with faster access to backup and log archive runs. Therefore, with instant restore, any improvement on I/O latency directly translates into lower time to repair—as perceived by a single transaction—and thus higher availability. Non-incremental techniques, where the restore latency is basically the time for complete recovery, do not benefit as much from low-latency storage hardware when it comes to improving restore latency.

In terms of latency and bandwidth trade-off in the instant restore algorithm, the first choice to be made is the segment size. In order to simplify the tracking of restore progress with a simple bitmap data structure, a fixed segment size must be chosen when initializing the restore manager. A good choice seems to be a size such that acceptable bandwidth is delivered even for purely random access, but not too many segments exist such that the bitmap would be too large—e.g., 1 MB for both SSD and HDD.

In order to exploit opportunities for increasing bandwidth, multiple contiguous segments should be restored in a single step when applicable. One technique to achieve that dynamically and adaptively is to simply run single-pass restore concurrently with instant restore. Since the two processes rely on the same algorithm, no additional code complexity is required. Furthermore, the coordination between them is essentially the same as that between concurrent instant restore processes—they both rely on the buffer pool and the segment recovery bitmap.

In terms of log archive access, the size of initial (i.e., not-yet merged) runs poses an important trade-off between minimizing merge effort and minimizing the lag between generating a log record and persisting it into the log archive. In order to generate larger runs, log records must be kept longer in the in-memory sort workspace. On the other hand, correct recovery requires that all log records up to the time of device failure be properly archived before restore can begin; thus, smaller initial runs imply lower restore latency for the first post-failure transactions. One simple technique that can potentially mitigate this concern is to adapt instant restore to use the recovery log with per-page log chains, as in single-page repair. Instant restore could be used to bring all pages in a segment to the state up to the end of the log archive. After that, single-page repair using the per-page log chain could be applied to apply the remaining log records to each page of a segment. This should only be necessary for the first few segments, as the log archiving process should quickly reach the *minLSN* restore point; from that point on, restore operations can rely solely on the log archive.

Besides these concerns specific to instant restore, established techniques to choose initial run size and merge fan-in based on device characteristics directly apply [30]. This is mainly because the access pattern of instant restore basically resembles that of an external sort followed by a merge join.

3.4. Coordination of multiple failures

As mentioned briefly above, the segment recovery bitmap enables the coordination of concurrent restore processes, allowing configurable scheduling policies. Another important aspect to be considered is the coordination among restore and the other recovery modes summarized in Table 1. This section discusses how to coordinate all such recovery actions without violating transactional consistency.

The first failure class—transaction failure—is the easiest to handle because its recovery is made transparent to the other classes thanks to rollback by logical compensation actions, as introduced in ARIES [3] and refined in the multi-level transaction model [31]. In this multi-level model, all undo actions—either from transaction failures or undo recovery after restart—are logical operations that operate on the lower-level, page-based storage interface in which redo recovery with physiological log records occurs. The implication is that recovery for the other failure classes must distinguish only between uncommitted and committed transactions. Transactions that abort are simply considered committed—it just happens that they revert all changes they made, i.e., they "commit nothing". Therefore, for the purposes of instant restore, transactions that issue an abort behave exactly like any other in-flight transaction, including those that started after the failure: they hold locks to protect their reads and writes and access data through the buffer pool, which possibly triggers segment restoration as described earlier.

Instant restart and single-page repair can be executed concurrently because they both perform log replay on a single page at a time, and thus coordination relies on the latching protocol of the buffer pool. If a single-page failure occurs during instant restart, it is detected in the fix operation, which invokes single-page repair to bring that page to its most recent state. Therefore, no further action will be required for the redo of that page in instant restart, because it will detect that the page LSN matches the expected value registered in the dirty page table. If a single-page failure happens after restart redo has recovered it, then single-page repair is simply invoked as it would normally be—regardless of ongoing restart recovery on other pages.

If a system failure happens during single-page repair, the failure will be detected again during restart, because the fix call on that page will result in the same failure. Therefore, single-page repair is simply re-invoked. Alternatively, a system transaction can be used to register the fact that a single-page failure was detected, along with the new storage location of the recovered page—this is similar to page migration in write-optimized B-trees [32]. Upon restart, the single-page repair process is simply resumed, and coordination works exactly as described above.

While instant restart and single-page repair both work at a page granularity and can thus be coordinated using latches in the buffer pool, that is not enough for instant restore. Here, a segment, whose size is fixed when a failure is detected, is the unit of recovery, and coordination relies on the segment recovery bitmap. Using two states—restored and not restored—avoids restoring a segment more than once in sequence, but additional measures are required to prevent that from happening concurrently. One option is to simply employ a map with three states, the additional one being simply "undergoing restore". A thread encountering the "not restored" state attempts to atomically change it to "undergoing restore": if it succeeds, it initiates the restore request for the segment in question; otherwise, it simply waits until the state changes to "restored".

Alternatively, coordination of segment restore requests can reuse the lock manager. A shared lock is acquired before verifying the bitmap state, and, in order to restore a segment, the shared lock must be upgraded to exclusive with an unconditional request. The thread that is granted the upgrade is then in charge of restoration, while the others will automatically wait and be awoken by the lock protocol, after which they see the "restored" state.

While the segment recovery bitmap provides coordination of concurrent restore processes, the buffer fix protocol is again used to coordinate restore with the other recovery modes. Concomitant restart and restore processes may occur in practice because some failures tend to cause related failures. A hardware fault, for instance, may not only corrupt persistent data, but also cause an operating system crash.

In order to not lose the progress of instant restore in case of a system failure, three restore actions must be logged: *begin*, *segment restore*, and *end*. During log analysis after a system failure, a begin log record causes the system to initialize the data structures of instant restore and redirect all page reads and writes to the restore manager, as described earlier. As segment-restore log records are found, the restore bitmap is updated accordingly, letting the system know which segments have already been restored prior to the system failure. In this case, a segment can only be considered fully restored once it has been fully written into the replacement device; thus, the segment-restore log record is only generated after that happens. Finally, if an end log record is found, the system knows that full restoration was completed, and it can return to normal mode.

After log analysis is completed, the restart and restore processes will be automatically coordinated with the methods described above. Restart recovery will fix pages in the buffer pool prior to performing any redo or undo action. The fix call, in turn, will issue a read request on the device. If the device has failed, the restore manager will intercept this request and follow the restore protocol described above. Only after the containing segment is restored, the fix call returns. After that, the page may still require log replay in the redo phase of restart, which is fine—the two recovery modes will simply replay different ranges of the page's history.

3.5. Implementation issues

The conceptual diagram of Fig. 5 does not illustrate how restore interacts with other parts of the database system architecture, most importantly the buffer pool. Reusing the buffer pool component is highly advisable, as it provides several advantages. The following paragraphs discuss the interaction between the restore manager and the buffer pool in detail, emphasizing the advantages of reusing the buffer pool as well as highlighting some important concerns for a practical implementation.

Relying on the normal *fix* protocol to access pages and replay log records on them allows high concurrency between restore actions and transactions accessing pages already in the buffer pool, i.e., pages that do not require immediate on-demand restoration. Transactions accessing these pages do not observe any delay from restore actions. Furthermore, once the segments containing these pages are picked by the restore scheduler, log replay can be skipped on them, since they are already up to date.

Frames of the buffer pool should also be used as the restore workspace, serving as buffer into which pages are loaded from the backup and log records are replayed into. In order to exploit large reads and fetch whole segments at once, the buffer pool should provide a prefetch function that allocates multiple frames and reads pages from the backup with a single scatter-gather I/O call (e.g., `readv()` in Linux). Once the read is performed, each allocated frame should only be added to the buffer pool if that page is not yet cached (i.e., not found in the page-ID lookup table). This prefetch function is not exclusive to instant restore—regular prefetch functionality has the same logic and should therefore be reused. Lastly, because such prefetched pages are in an older state, a control-block flag is used to indicate that they are still in need of recovery; user transactions that happen to fix them must therefore coordinate with the restore manager.

As segments are restored, the regular page cleaning protocol of the buffer pool takes care of writing them into the replacement device. Using proper I/O scheduling that can exploit large page writes [33], whole segments—or even multiple adjacent segments—are written at once without any explicit involvement of the restore manager. As in the prefetch case, buffer pool functionality is reused with minimal restore-specific code. One crucial requirement in this case is that segment-restore log records, which are required to support recovery from system failures during restore, should only be produced after all pages of a segment have been successfully written out. Alternatively, log records can be generated immediately after log replay; in case of a concomitant system failure, log analysis must then additionally mark a segment as "unrestored" if any of its pages are in the dirty page table.

One important concern involving the synchronization of restore actions and user transactions is to avoid lost updates that can occur when restoring segments in the buffer pool. This can happen with the following sequence of events: (1) a page of the failed device is already in the buffer pool when the failure is detected; (2) when the restore manager is initialized, it waits for the `restore_begin` log record to be archived (as discussed earlier, this is required to make sure that all updates up to the failure point are replayed during restore); (3) after that, the page is updated by a user transaction in LSN x, written into the replacement device, and subsequently evicted from the buffer pool; (4) now, the segment containing that page is picked for restore, which fetches the old version from the backup and replays all log records until the `restore_begin` LSN; (5) finally, that page is written by the cleaner and evicted. When this last write is performed, the update on LSN x is lost, because the restored version of the page was older than the one on the replacement device.

Unfortunately, there is no straight-forward solution for this lost-update problem. The key issue here is that neither the restore manager nor the buffer pool have knowledge of individual pages written by the page cleaner. Thus, the situation above can only be detected if the system keeps track of individual pages restored rather than whole segments. In step 4 above, for example, an auxiliary data structure could be used to inform the restore manager that that particular page does not need recovery, or that it should be fetched from the replacement device rather than the backup. Such data structure would complement the segment recovery bitmap by keeping track of individual pages already restored in the segments that are not yet marked as restored in the segment recovery bitmap.

An alternative solution reuses the single-page repair infrastructure. If a page recovery index is maintained, either as a separate data structure or embedded in parent-to-child page pointers [27], then the overwrite of step 5 below would be detected when the page is re-fetched, applying the necessary single-page repair actions as needed. This mechanism also forgoes the need to wait for the log archiver to reach the `restore_begin` log record, since every page would be guaranteed to be fully recovered with a combination of log-archive replay and single-page repair. Furthermore, it could be easily combined with the write elision technique [5] to alleviate write pressure on the replacement device. Lastly, a middle-ground solution would be to wait for the log archiver as described earlier, but only create and maintain the page recovery index for the duration of instant restore; this achieves the same guarantees without having to maintain the page recovery index during normal processing. This last solution was implemented in the prototype of this paper, because it is relatively simple to implement if the system already provides a page recovery index and single-page repair, which is the case in our prototype. A more in-depth discussion of implementation issues for instant recovery techniques, relying on the same prototype used here, is provided in a related dissertation [8].

3.6. Summary of instant restore

Instant restore is enabled by an indexed log archive data structure that can be generated online with very low overhead. By partitioning data pages into segments, the recovery algorithm provides incremental and on-demand access to restored data. The algorithm requires a simple bitmap data structure to keep track of progress and coordinate restoration of individual segments under configurable scheduling policies.

The generalized nature of instant restore enables a wide range of choices for trading restore latency and bandwidth. These choices can be made adaptively and robustly by the system using simple techniques. Moreover, while instant restore mitigates many of the issues with high-capacity hard disks, making them a more attractive option, it still benefits greatly from modern storage devices such as solid-state drives. Therefore, the technique is equally relevant for improving availability with any kind of storage hardware.

Lastly, the restore processes can be easily coordinated with processes from other recovery modes—the independence of these modes and the integrated coordination using the buffer pool ensure transaction consistency in the presence of an arbitrary mix of failure classes.

4. Experiments

This section presents an empirical evaluation of instant restore, focusing on the impact of restore on concurrent transactions as well as on the efficiency of restore itself, i.e., how quickly the failed media is fully restored. The main goal is to investigate how much the media failure disturbs transaction processing, looking at throughput and latency for restore actions as well as individual transactions. The experiments presented here serve as a proof of concept for how instant restore can improve availability in practice, focusing on detailed metrics of system behavior during individual media restore executions.

4.1. Environment and workload

We implemented instant restore in a fork of the Shore-MT storage manager [34] called *Zero*. The code is available as open source [1]. The workload consists of the TPC-C benchmark as implemented in Shore-MT, but adapted to use the Foster B-tree [35] data structure for both table and index data.

All experiments were performed on dual six-core CPUs with HyperThreading. The system has 100 GB of RAM and several Samsung 840 Pro 250 GB SSDs. The operating system is Ubuntu Linux 16.04 with Kernel 4.4.0 and all code is compiled with `gcc` 5.4 and `-O3` optimization.

To generate the dataset for these experiments, a database of 10 GB is loaded (TPC-C scale factor 75) and copied to a separate file to serve as full backup. Then, the benchmark is executed until another 10 GB of log data is produced; the log is then archived into one level-3 run (i.e., merged twice) of 5 GB, six level-2 runs of 750 MB, and four level-1 runs of 110 MB. Finally, the database is cleaned (i.e., all dirty pages are flushed) and the recovery log is emptied. During an experiment, the benchmark is executed on this dataset with 8 worker threads for five minutes to warm-up the buffer pool, after which a media failure is injected. The experiment then continues until the complete device is restored and then for five more minutes after that. All experiments described below use the same dataset and follow this same pattern.

[1] `http://github.com/caetanosauer/zero`

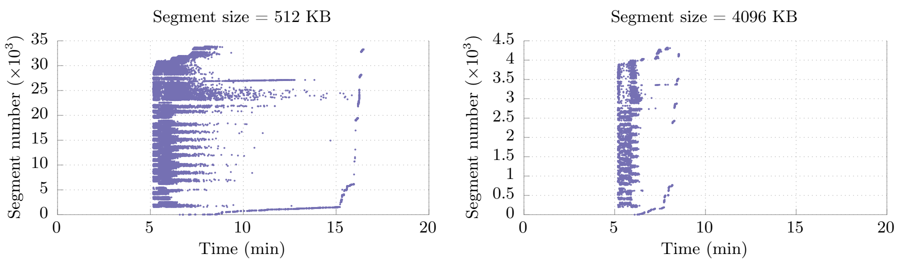


Figure 7: Pattern of segment restoration during instant restore

4.2. Segment size, latency, and bandwidth

The first experiment analyzes restore efficiency with different segment sizes. A buffer pool of 10 GB was chosen for the benchmark; this corresponds to the size of the initial database and it is not too small so that the process would be I/O-bound, but also not too large so that on-demand restore requests are very rare. The sizes analyzed are between 512 KB (64 pages) and 8 MB (4,000 pages), varying in exponential steps.

The first result, shown in Fig. 7, shows the pattern of segment restoration over time for segment sizes of 512 KB and 4 MB. The x-axis shows the time since the beginning of the experiment, while the y-axis shows segment numbers (in increments of 1,000). Each dot in the chart represents the successful restoration of one segment. As the pattern shows, many segments are restored in a very scattered way in the beginning of the restore process; this is expected because, in this phase, many on-demand requests from transactions arrive in the restore scheduler. As time goes on, less requests arrive, making the pattern more sparse.

One visible feature in this experiment is the effect of a background restore thread, which performs single-pass restore in parallel but with low priority, i.e., it only picks up segments to restore if no requests exist in the restore scheduler. The effect of background single-pass restore is shown as the sequence of dots in the lower part of the charts, which at the very end rises to the top rapidly. The lower range of segment numbers contains important pages, such as catalogs and B-tree inner nodes—these are kept cached in the buffer pool, so that no restore request ever arrives. Thus, it is up to the background process to restore them. Towards the end of the experiment, the background restore process picks up the remaining segments, filling up the "holes" in the segment recovery bitmap. Once is passes the last segment, the replacement device has been fully restored and it may thus terminate the restore procedure. This restore pattern described above is observed in both segment sizes shown in Fig. 7, but in different granularities. These charts also show that the smallest segment size (512 KB) requires about 12 minutes for complete restoration, while the largest one (8 MB) requires less than 4 minutes.

The next result, based on the same experiment, is shown in Fig. 8, which plots average restore bandwidth over time for four segment sizes. This chart shows an interesting result: in the first minutes of the restore process, band-

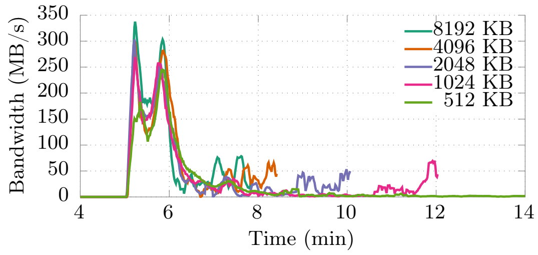


Figure 8: Restore bandwidth over time with different segment sizes.

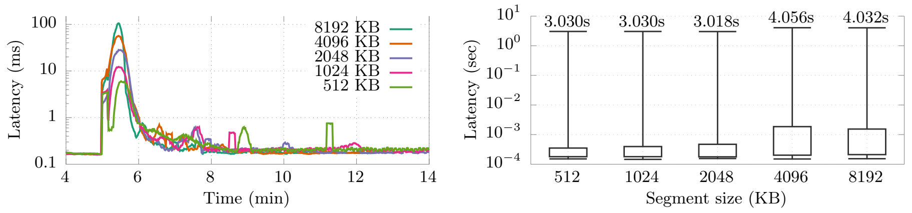


Figure 9: Average restore latency during instant restore: time series (left) and distribution of individual values (right).

width, i.e., the rate at which segments are restored, is very similar for all segment sizes. This is because, in this stage, all worker threads are performing restoration of the segments they request in parallel, so that the aggregate bandwidth does not depend so much on the segment size. This is largely due to the fact that the experiments use SSDs, which support a high degree of I/O parallelism. Towards the end phase of the restore process, most segments are restored by the background single-pass procedure, and thus the observed bandwidth becomes much more dependent on the segment size. The bandwidth observed in the end phase is therefore the main factor determining the total restore time.

Fig. 9 plots the average transaction latency during this same experiment in two different representations: on the left, a time series of average latency values and, on the right, the distribution of these values in a box plot. On the left chart, the expected behavior described earlier in Section 3.3 is observed: the smaller the segment, the less restore latency is incurred on transactions waiting to access data on the failed device. Note that the y-axis is in logarithmic scale; thus, during the first minute of media failure, average latency with 512-KB segments is one order of magnitude lower than with 8-MB segments. On the box plot on the right, 50% of the observed latency values fall within the boxes, while the lines at the bottom and at the top extend to the lowest and highest latency observed, respectively. The maximum value is due most importantly to the wait for the log archiving process to reach the `restore_begin` log record; since the experiment leaves the archiving process running eagerly in the background, this wait is about 3 seconds only.

The results observed for this experiment show a clear trade-off between segment size and restore efficiency. Larger segments allow for higher bandwidth

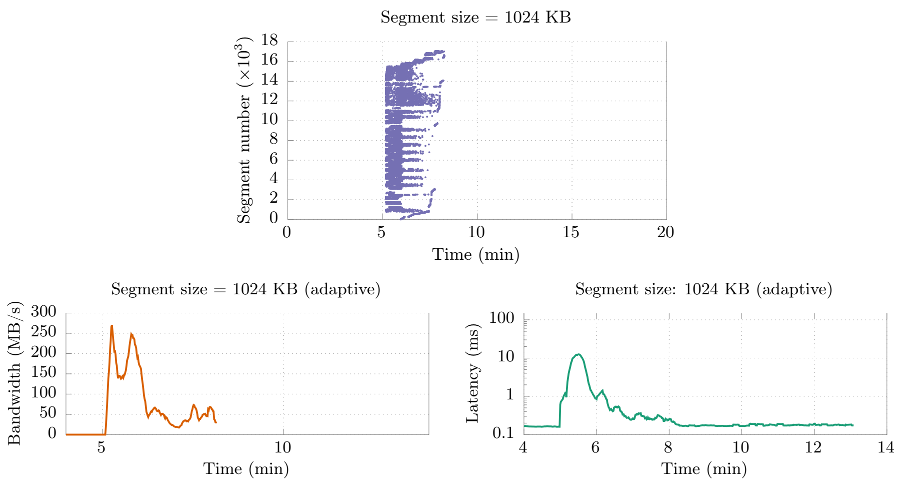


Figure 10: Restore pattern, bandwidth, and latency with adaptive technique

in the background single-pass restore process, which provides shorter total restore time. Smaller segments, on the other hand, reduce the restore latency as perceived by individual transactions in the beginning phase. To eliminate this trade-off and provide the best of both worlds, an adaptive technique can be used, in which the single-pass background service attempts to restore multiple adjacent segments whenever it encounters them. With this technique, small segments can be used, so that low latency is observed in the beginning phase. Then, during the end phase, background restoration performs large reads and writes of multiple segments, as if a much larger segment size would be used.

Results for this adaptive technique are shown in Fig. 10. It uses segments of 1 MB, which can be coalesced into large units of up to 8 MB by the background restore process. The top chart shows the restore pattern, which clearly shows that overall restore lasts about as long (3 minutes) as for the largest segment in the previous experiment (8 MB) while the beginning phase has a dense pattern like the one observed earlier for the 512-KB segment size. The plots for restore bandwidth (bottom left) and latency (bottom right) also show that the advantages of small and large segments are combined.

4.3. Transaction throughput

The next experiment fixes the segment size at 1 MB with the adaptive technique and varies the buffer pool size. The goal is to evaluate the impact of the media failure and concurrent restore actions on running transactions. With a sufficiently large buffer pool, a media failure should go completely unnoticed, since all page accesses are hits, which do not trigger segment restore. In that case, the background restore process will perform single-pass restore, using the largest possible unit thanks to the adaptive technique described above. Therefore, this configuration should yield the shortest total recovery time. For smaller buffer pools, instant restore should cause a significant dip in transaction

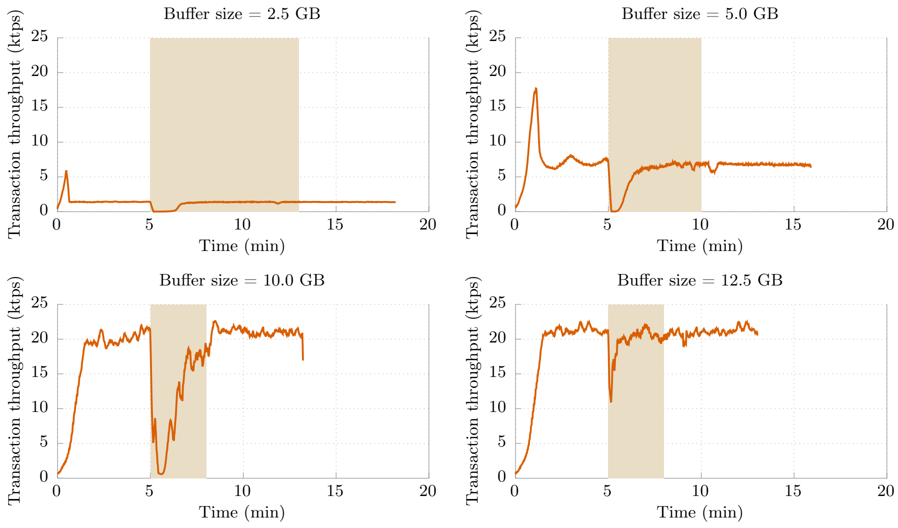


Figure 11: Transaction throughput observed during instant restore with different buffer pool sizes

throughput, but it should gradually raise back up as segments of the working set are restored.

The buffer pool sizes considered here vary from 2.5 GB to 12.5 GB. Fig. 11 shows the results. In each chart, the x-axis shows elapsed time, as in the previous charts, whereas the y-axis shows transaction throughput. The shaded area starting at the five-minute mark shows the time during which instant restore was active. As the top-left chart shows, the smaller buffer pool incurs the longer restore time, since the high miss ratio causes almost all segments to be restored on demand. However, after about two minutes only, the pre-failure throughput is reestablished, thus demonstrating the main benefit of instant restore. With the remaining buffer sizes, the pattern is similar, but with higher transaction throughput. Note, for example, that with the 10-GB buffer pool, restore finishes before the pre-failure throughput is reestablished; this is expected because restore uses frames of the buffer pool, and thus a higher miss rate is observed until all restored pages that are not in the working set are evicted.

With the buffer pool size of 12.5 GB, which is larger than the working set, the dip in throughput is very small, incurred mostly by the wait for the log archiver. This wait is only necessary because the buffer pool was probably not warm enough after the five-minute execution, and thus a few misses still occur. Unfortunately, the prototype system used here does not support a way to manually warm-up the buffer pool or to detect when it is fully warmed-up. Nevertheless, the results clearly demonstrate the effect of larger buffer pool sizes on instant restore—as it gets larger than the working set, the media failure goes practically unnoticed.

5. Conclusions

Instant restore improves perceived mean time to repair and thus database availability in the presence of media failures. We identified two main deficiencies with traditional recovery techniques, such as the ARIES design [3]: (i) media recovery is very inefficient due to its random access pattern on database pages, which means that time to repair is unacceptably long; and (ii) data on a failed device cannot be accessed before recovery is completed. The first deficiency was addressed with single-pass restore [4], which introduces a partial sort order on the log archive, eliminating the random access pattern of log replay.

The second deficiency is addressed with the instant restore technique, which was first described in earlier work [5] and discussed in more detail, implemented, and evaluated in this paper. By generalizing single-pass restore and other recovery methods such as single-page repair, instant restore is the first media recovery method to effectively eliminate the two deficiencies discussed. In comparison with traditional ARIES media restore, instant restore delivers not only the benefits of single-pass restore (i.e., substantially higher bandwidth and therefore shorter recovery time), but also much quicker access (e.g., seconds instead of hours) to the application working set after a failure.

Our empirical analysis shows that instant restore is able to effectively deliver the efficiency of single-pass restore while cutting down restore latency by multiple orders of magnitude. Thanks to an adaptive technique, small segments can be used to prioritize the application working set after a failure and still achieve high average bandwidth by restoring multiple adjacent segments when appropriate. The experiments also analyze the impact of a failure on transaction throughput, which largely depends on the size of the working set in relation to the buffer pool size. The results confirm our expectation that the pre-failure transaction throughput is re-established earlier as memory size increases—up to a point where a media failure goes completely unnoticed. The net effect is that availability is greatly improved and the number of missed transactions due to media failures is significantly reduced.

References

[1] J. Gray, What next?: A dozen information-technology research goals, J. ACM 50 (1) (2003) 41–57.

[2] J. Gray, Why do computers stop and what can be done about it?, in: Symp. on reliability in distributed software and database systems, 1986, pp. 3–12.

[3] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz, ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, ACM Trans. Database Syst. 17 (1) (1992) 94–162.

[4] C. Sauer, G. Graefe, T. Härder, Single-pass restore after a media failure, in: Proc. BTW, LNI 241, 2015, pp. 217–236.

[5] G. Graefe, W. Guy, C. Sauer, Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2016.

[6] C. Sauer, G. Graefe, T. Härder, Instant restore after a media failure, in: Proc. ADBIS, 2017.

[7] T. Härder, A. Reuter, Principles of transaction-oriented database recovery, ACM Comput. Surv. 15 (4) (1983) 287–317.

[8] C. Sauer, Modern techniques for transaction-oriented database recovery, Ph.D. thesis, TU Kaiserslautern, Germany, Dr.Hut-Verlag München (2017).

[9] G. Graefe, A survey of B-tree logging and recovery techniques, ACM Trans. Database Syst. 37 (1) (2012) 1. doi:10.1145/2109196.2109197. URL http://doi.acm.org/10.1145/2109196.2109197

[10] J. Gray, Notes on data base operating systems, in: Operating Systems, An Advanced Course, 1978, pp. 393–481.

[11] C. Mohan, I. Narang, An Efficient and Flexible Method for Archiving a Data Base, SIGMOD Rec. 22 (2) (1993) 139–146.

[12] C. Mohan, K. Treiber, R. Obermarck, Algorithms for the management of remote backup data bases for disaster recovery, in: Proc. ICDE, 1993, pp. 511–518.

[13] D. J. Haderle, T. Majithia, Fast log apply, US Patent 6,289,355 (Sep. 11 2001).

[14] Oracle Corporation, RMAN Incremental Backups, Oracle Database Documentation 10g, Sect. 4.4, 2015.

[15] D. Bitton, J. Gray, Disk Shadowing, in: Proc. VLDB, 1988, pp. 331–338.

[16] P. M. Chen, et al., RAID: high-performance, reliable secondary storage, ACM Comput. Surv. 26 (2) (1994) 145–185.

[17] IBM, High availability through log shipping, DB2 10.5 for Linux, UNIX, and Windows Documentation.

[18] MySQL, Replication, MySQL 5.7 Reference Manual, Chapter 16.

[19] Microsoft Corporation, About Log Shipping (SQL Server), SQL Server 2017 Documentation.

[20] The PostgreSQL Global Development Group, High Availability, Load Balancing, and Replication, PostgreSQL 10.5 Documentation.

[21] M. H. Eich, A classification and comparison of main memory database recovery techniques, in: Proc. ICDE, 1987, pp. 332–339.

[22] T. J. Lehman, M. J. Carey, A recovery algorithm for A high-performance memory-resident database system, in: Proc. SIGMOD, 1987, pp. 104–117.

[23] E. Levy, A. Silberschatz, Incremental recovery in main memory database systems, IEEE Trans. Knowl. Data Eng. 4 (6) (1992) 529–540.

[24] N. Malviya, A. Weisberg, S. Madden, M. Stonebraker, Rethinking main memory OLTP recovery, in: Proc. ICDE, 2014, pp. 604–615.

[25] J. Arulraj, A. Pavlo, S. Dulloor, Let's talk about storage & recovery methods for non-volatile memory database systems, in: Proc. SIGMOD, 2015, pp. 707–722.

[26] I. Oukid, et al., SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery, in: Proc. DaMoN, 2014, pp. 8:1–8:7.

[27] G. Graefe, H. A. Kuno, Definition, Detection, and Recovery of Single-Page Failures, a Fourth Class of Database Failures, PVLDB 5 (7) (2012) 646–655.

[28] G. Graefe, H. A. Kuno, B. Seeger, Self-diagnosing and self-healing indexes, in: Proc. DBTest, 2012, p. 8.

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

[30] G. Graefe, Query Evaluation Techniques for Large Databases, ACM Comput. Surv. 25 (2) (1993) 73–170.

[31] G. Weikum, G. Vossen, Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann, 2002.

[32] G. Graefe, Write-optimized b-trees, in: Proc. VLDB, 2004, pp. 672–683.

[33] C. Sauer, L. Lersch, T. Härder, G. Graefe, Update propagation strategies for high-performance OLTP, in: Proc. ADBIS, 2016.

[34] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, B. Falsafi, Shore-MT: a scalable storage manager for the multicore era, in: Proc. EDBT, 2009, pp. 24–35.

[35] G. Graefe, H. Kimura, H. A. Kuno, Foster B-trees, ACM Trans. Database Syst. 37 (3) (2012) 17.