# Filtering and Indexing in a Partially-sorted Log Archive

Lucas Lersch

University of Kaiserslautern
Germany

**Abstract.** In order to provide the illusion of an instant restore of a database storage device in the case of a media failure, the ability to restore single database pages on demand is required. Therefore, it is essential to have an efficient access to log records that refer to a certain database page. To achieve such an efficiency, the first step is to sort log records by the page identifier they refer to, prior to writing them from a "temporary log" device to a "log archive" device. This enables the archived log records which refer to the same page to be read sequentially when performing log replay as part of a media restore. Many runs of sorted log records are stored in the log archive, since an external merge sort is required when sorting the log records from the temporary log. In other words, log records which refer to the same page are spread across multiple runs and the whole log archive is said to be partially-sorted. Thus, an index-like data structure is also required for accessing directly the log records that refer to a certain page. In addition, since large runs of sorted log records can be generated during the log archiving process, an efficient load performance is also desirable. Atomicity of operations in this data structure must also be considered in order to enable tolerance to system failures. In this report, we discuss some of the suitable data structures, try to identify their advantages and disadvantages, and propose an overview of the use of one of these data structures in the context of the log archive and instant restore.

## 1 Introduction

Most database systems, old and modern, rely on logging to enable recovery from system failures and media failures as well as rollback in the case of transaction aborts. In other words, logging ensures that, in the case of a failure, the most recent and consistent database state can be reached from a chaotic state.

The ARIES [8] algorithm employs write-ahead logging (WAL), which essentially consists of forcing log records of each operation to stable storage prior to writing back modified pages. In a no-force system [5], writing modified pages back to stable storage is not required as part of transaction commit. However, writing log records up to that transaction is required for committing. This means that even log records generated before by other transactions must also be written in a way to preserve the log sequence number correlation. For this reason, a storage device separated from the main database storage device is usually dedicated to logging purposes. It is highly desirable for such a device to be latency-optimized (in terms of cost), as the write operations on the log directly influence the throughput of the whole database system.

However, until the present time, latency-optimized devices (such as SSD) do not provide good storage costs (in terms of MB/$) when compared to other storage devices such as hard drives or tape devices. Since the system generates logs continuously over time, log entries tend to become obsolete with time and consequently expensive to maintain in a latency-optimized storage device. For this reason, it is highly desirable to move old log entries (from already completed transactions) from this device to a device with less expensive storage costs. Such devices are also bandwidth-optimized, in the sense that they offer a better relation between sequential speed and cost.

The operation of moving old log records to a less expensive device is known as "log archiving". In other words, the log archive keeps obsolete log records that are required only in the case of recovering from a media failure, where the replay of these log records might be necessary. For clarity in the remainder of this work, the term "temporary log" is used to differ from the "log archive".

In addition to logging, full backups that reflect the database image are periodically taken. It is important to note that the backup is generated one page at a time in a fuzzy way, meaning that the backup does not reflect a snapshot of the database at some point in time. Instead, it reflects a snapshot of each page – each at a different point in time. Therefore, in the case of a media failure, the steps for recovering the database to the most recent operational state consist of copying the last full backup to the replacement storage media and replaying all operations that happened after this full backup was made. It is important to note that the log replay operation induce random access to database pages in the replacement media. As an alternative to alleviate the costs of replaying the temporary log and the log archive, lightweight incremental backups can be taken more frequently, since they store only differential information. In this case, incremental backups are restored after the full backup and before replaying the contents of the temporary log and log archive. The relation between devices is shown in Figure 1.

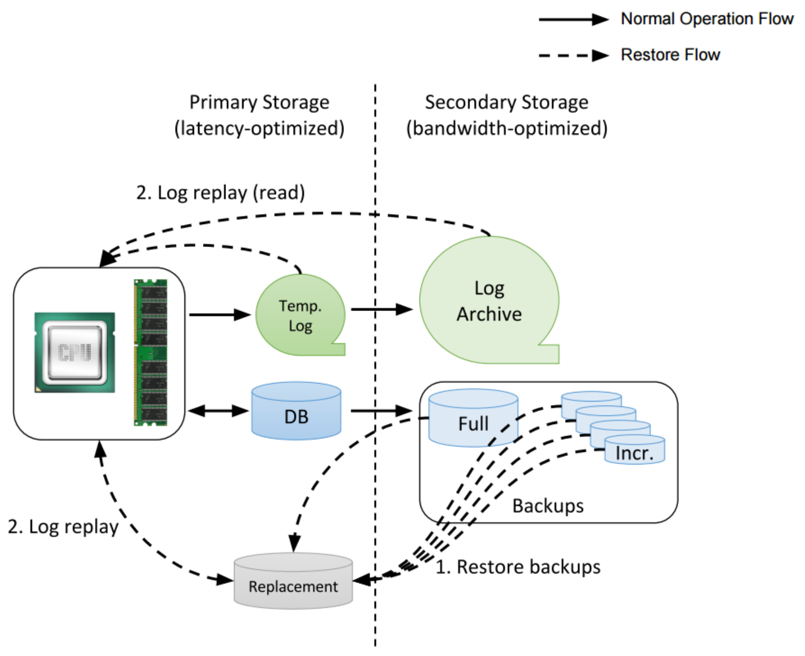

**Fig. 1.** Devices involved in a database system and their relations during restore.

The single-pass restore technique [9] aims at minimizing the costs of restoring a failed media device. It does so by reducing the costs of log archive replay to a point where even differential backups do not offer a great advantage in the whole restore process. The main idea is to store the log records in the log archive partially sorted by the page identifier, so both the backup device and the log archive device can be sequentially read, and a merge operation between these two can be performed. It is important to mention that the temporary log does not participate in the merge join and it is replayed only after the log archive replay has completed. In addition, assuming the backup and the log archive are in different devices, the sequential reads can be done in parallel. The devices involved in single-pass restore can be seen in Figure 2.

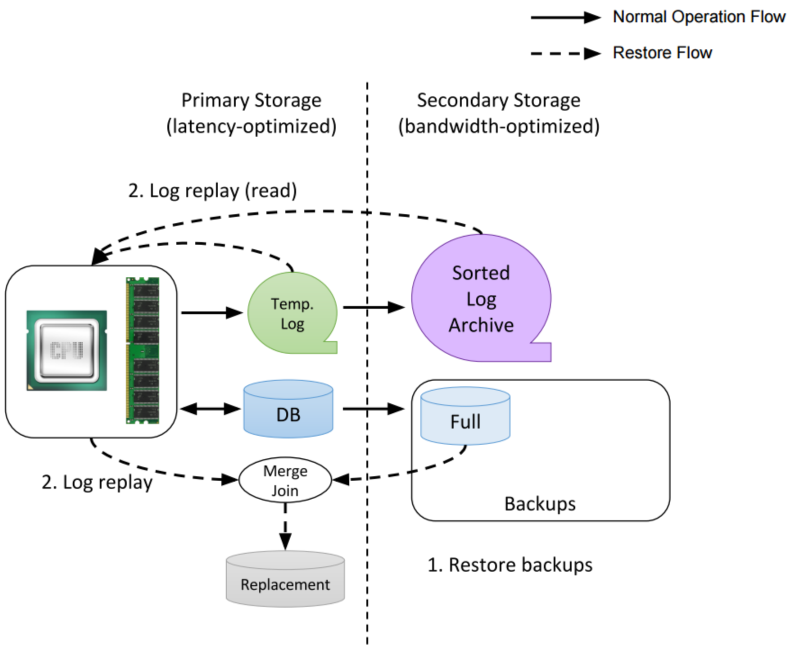

**Fig. 2.** Devices involved in a database system and their relations during single-pass restore.

In this context, the “instant restore”[3] technique is introduced, which aims at taking the basic idea behind single-pass restore even further by enabling transactions to run practically immediately after the replacement device is available, not requiring the whole single-pass media restore to take place prior to that. This is done by allowing transactions to execute at an early state and recovering database pages on demand, giving the illusion of an instant restore operation. In addition, in order to eventually achieve a complete restored state, it is desirable to have a process running in the background responsible for restoring pages not yet restored by the on-demand scheduler.

To realize the idea of instant restore, efficient access to the log entries in the partially-sorted log archive is required. This efficient access can only be achieved by creating an indexing structure over the log archive. The main focus of this work is then to define important requirements that would be taken into consideration for implementing an ideal solution. The goal in providing a well-defined design is to enable a future work on the implementation of the instant restore on the Zero storage manager[1] (based on Shore-MT [6]). Therefore, the remainder of this report is going to analyze the log archiver structure as it was initially implemented in the Zero system. In Section 2, the properties of B-trees, a data structure commonly used for indexing purposes, are discussed, its advantages and disadvantages as well the reasons why it is not considered suitable for the problem. Section 3 introduces zone

[1] https://github.com/caetanosauer/zero

filters and zone indexes as an alternative data structure that aims to be simpler and more adequate for dealing with necessary requirements, but still not applicable to the exact scenario of log archiving. Finally, Section 4 considers a tailored data structure which is a mix of partitioned B-trees and zone indexes. Based on our initial qualitative analysis, this data structure provides the best trade-off between simplicity, efficiency, and effectiveness.

## 2 Partitioned B-trees

B-trees [1] are a ubiquitous data structure in the context of database systems. Largely employed to implement indexes, it would be natural to think about employing these widely used and well accepted data structures or one of its variations to achieve such a goal of indexing the log archive.

The partitioned B-tree variant [2] was introduced to alleviate the trade-off between good lookup response time at the cost of load bandwidth, which is inherent from traditional B-trees. The main idea behind a partitioned B-tree is to have an additional artificial leading key value which represents the partition number. This simple variation introduces multiple benefits.

In the case of a partially-sorted log archive, a partitioned B-tree allows to put all the sorted runs into a single data structure, having the run number as artificial leading key column. The direct benefit from this is that inserting new runs in the index structure is less expensive than in a traditional B-tree. However, in order to enable more efficient loads, partitioned B-trees introduce some overhead that should be considered. These are discussed below.

First, the additional leading key column incurs additional storage costs. Fortunately, prefix truncation techniques can be employed to reduce disk space and bandwidth costs to a level where they can be safely ignored.

Second, the search operation becomes more expensive, as it has to be performed in every partition. In other words, the number of partitions has a direct impact on the performance of search operations. Therefore, if there is an asynchronous process running in the background merging runs, the number of partitions can be kept to a desired level that does not deteriorate the performance.

Third, there is the need for atomicity in order to enable tolerance to system failures. In other words, if a system failure happens while a B-tree node is being written, the whole B-tree structure is compromised. Even in the case where only internal nodes are affected and it is possible to rebuild the whole index from the leaf nodes, it is an extremely costly operation, since the log archive tends to grow indefinitely as new log records are generated. The required atomicity could be easily achieved by reusing the logging infrastructure already present in regular B-tree indexes. However, generating new log records for indexing existing log records creates a cyclic behavior in the whole log archiving process. As a result, we have so-called "meta" log records, i.e., log records referring to operations related to the indexing of other log records. In the case of a system failure, this "meta" log would have to be recovered prior to starting or resuming the instant restore technique. Therefore, this approach introduces a whole new level of complexity for treating these "meta" log records, such as new logging and recovery modes.

An alternative to using the logging infrastructure is to rely on a copy-on-write strategy to achieve the desired atomicity in the B-tree index, while avoiding the creation of "meta" log records. By doing this, every time an update is made in the index, all the B-tree nodes from the affected node up to the root are copied, the changes are then made in these copies and later the copies replace the original B-tree nodes in a single atomic operation. The obvious disadvantage is that this approach would require a disk space management layer to keep track of old B-tree pages that were replaced by updated ones and free disk space that can be used. However, the complexity of implementation introduced by this additional layer is not desirable.

To summarize, using a B-tree index, which at first seems to be the natural fit for the problem, requires additional complexity that causes the implementation of this approach to become cumbersome. In addition, log archiving generates large runs, which makes the performance of the loading process in an index structure extremely important. Even though partitioned B-trees offer a much better load bandwidth than regular B-trees, we can resort to a much simpler data structure that offers an even better load performance from which we can directly benefit.

## 3 Zone Filters & Zone Indexes

The use of index structures in database systems usually imposes a trade-off between better query performance and load bandwidth in bulk insertions. In other words, the maintenance requirements of the index structure in relational data warehousing result in additional load bandwidth costs. In this sense, the behavior of the log archiver is similar to load operations in a data warehouse, since large runs of sorted log records are created. These costs, in the case of loading large amounts of data, refer to the random insertions or sorting the incremental changes followed by merging these changes into an existing index. As mentioned in the previous section, partitioned B-trees create the ability to perform the work later and incrementally, but do not reduce such effort.

Netezza's zone maps, which are closely related to "small materialized aggregates" [7], were introduced to alleviated this trade-off by offering a higher load bandwidth without the costs related to index maintenance and index tuning at the same time as it offers very fast scans for typical queries. The main idea behind zone maps is to group data records in so-called "zones" (which can be equivalent to a database segment) and to gather the minimum and maximum values of columns for each of these zones. The minimal and maximal information can be easily gathered during the initial load, as well as updated in future insertions (maintenance of these values during deletion is omitted). By having the interval range of values within a zone, it is possible to simply skip over zones in which a queried value is guaranteed to be absent (the presence of a value within a zone cannot be guaranteed by minimal and maximal values). However, some characteristics should be considered as possible weaknesses, as they might impose important limitations to the use of zone maps. We discuss these below.

First, if the range of values of a zone map contains outliers, storing the minimal and maximal values of this zone does not offer a great benefit.

Second, there is the requirement for correlation between load sequence and the values of the attributes being captured. As an example, assume an e-commerce database with a table that keeps track of information about client orders and a column that contains timestamp values referring to the order creation time. In this case, the values of this column have a direct correlation to the sequence in which each row was inserted. Attributes without such a correlation do not benefit from zone maps, since it is unlikely for the range of values within a zone to be substantially smaller than the entire domain of values. In other words, if there is no correlation, then a large portion of the entire domain of values is present in every zone and no zone can be skipped, since almost every value is comprised in the range between the minimum value and maximum value of a zone.

In the partially-sorted log archive, if we consider the domain of a single run file, there is a direct correlation between the page identifier and the "load sequence", since the log records within a single run file are primarily sorted by the page identifier of the page they refer to. It is important to note that the mentioned "load sequence" refers to the sequence given by the partial sorting and not to the LSN sequence. However, if we treat each run file as a "zone", such correlation does not necessarily exist, since log records referring to a certain page can be present in multiple run files.

Zone filters [4] were introduced to address the limitations of zone maps without reducing the advantages. They generalize the idea of zone maps by introducing multiple boundary values, i.e., multiple lowest and highest values for a certain range. This can alleviate the problem with outliers in the sense that more lowest (or highest) values can be used to surpass the number of expected outliers. In addition to this, a bit vector can also used for each zone to provide more precise information of the values present in that zone by mapping these values to positions in the bit vector. In this sense, the bit vector helps to filter columns without correlation to the load sequence.

Zone indexes extend the concept of zone filters by allowing direct access to values within a zone. In combination, these techniques provide near-direct access to certain key values in an index structure that can be loaded sequentially. By using zone filters, regions of the input data that do not contain the key values of interest can be skipped during a scan. By using zone indexes, key lookup is supported within a zone, allowing direct access even in the case of large zones and low selectivity. In addition to that, zone indexes can be built very efficiently during the loading process, at least more efficiently than B-trees, because it does not require sorting. However, this is not relevant in our scenario, since the input is (partially) sorted regardless of the index structure.

In summary, even if partitioned B-trees create the illusion of alleviating the load costs of traditional B-trees, they do not really reduce the costs (only postpone it). On the other hand, zone filters and zone indexes can be employed in a non-indexed structure for improving search performance without impact on load performance. While this difference between partitioned B-trees and zone indexes is present in common scenarios, there is no such distinction in the log archive scenario, since the log records are already pre-sorted. Nevertheless, these data structures can be constructed as part of the load process, with little processing effort and memory. All the costs related to sorting the future index records and to external merge sort, in the case of a fully-indexed structure, are relinquished.

In the next section, we evaluate the application of zone filters and indexes to the log archive. Furthermore, we provide a blueprint for implementing this approach in an existing system with partially-sorted log archives.

## 4 Indexed Log Archive

In order to realize the idea of instant restore, log records pertaining to a certain page must be retrieved efficiently. In this section we propose a design inspired by zone filters and zone indexes to achieve this desired efficiency. A partially-sorted log archive, as used in single-pass restore, is divided into sorted runs (files) as illustrated in Figure 3.

Each run file contains a disjoint range of the LSNs in the temporary log. The log records within a single run file are sorted by page_id and stored in blocks of fixed length. The information about the LSN range of each run is present in the file name. When performing on-demand restore, the log archive is searched for log records of a certain page_id beginning at a certain start_lsn and optionally finishing at a certain end_lsn. Thus, initially we can define a query in the log archive index by a tuple:

```
<page_id, start_lsn, (end_lsn)>
```

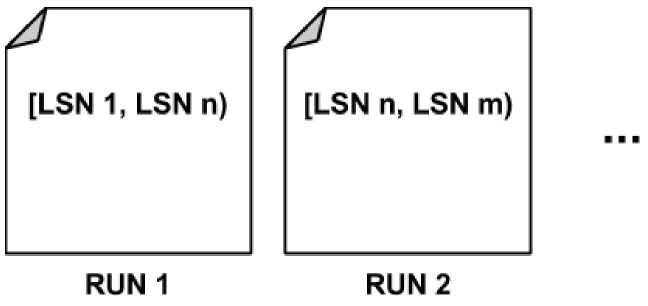

**Fig. 3.** Run files in the log archive.

Moreover, the search process can be divided into two steps:

1. Search for run.
2. Search for block within a run.

First the run to be searched must be determined. Second, since runs tend to be large files, given a run, the access to the block containing the first log record in the search query must be as efficient as possible.

### 4.1 Filtering runs

From a query, since the start_lsn is known, it is possible, based on the LSN range of each run, to determine the first run file to search for the log record with start_lsn. Since the log records in a single run are sorted by page_id, once the first log record is found, all the following log records from the same page_id within that run are read. However, log records for a certain page span across multiple runs and not all pages have log records in every run. Thus, after the last log record for a page in a run is read, there is no information about the LSN of the next log record pertaining to the same page and, as a result, it is not possible to precisely determine which run does it contain.

Even if the runs are sorted and an efficient search algorithm can be employed to scan the run file, searching for a log record of a certain page without the aid of an index can be expensive, even more if the run file is scanned just to find out that log records about that page are not even present. Therefore, it is highly desirable to have additional information about runs to enable us to skip the ones that do not contain log records of the page_id being searched. Each run file is treated as a "zone" in such a way that it is possible to filter only the runs that contain desired information. To this end, a data structure is needed to store such information that will enable runs to be filtered. It is desirable for this data structure to be simple and lightweight to minimize the storage overhead, since it has to be stored alongside each run file. For a naive approach, let us consider that information about the minimum and maximum page_ids for each run can be used to discard runs that do not have the queried page_id in its interval.

The trade-off from filtering runs has to be considered. Information required for filtering runs adds additional storage costs in the log archive. If we have a run file that does not contain the queried page_id and there is no filtering, a portion of this run file, the one where the page_id was supposed to be, is loaded into memory. This portion of the file is then searched only to find out that the page_id is not present. If an effective filter is employed, in such a way that it is possible to determine the absence of the page_id in the run, there is no need to load any portion of the run. Therefore, the main gain from having a filter is to avoid one useless I/O operation. In other words, the additional costs only pay off if the filter enables a run to be skipped. Then, the main challenge is to determine if applying a filter to a certain run offers a sufficient gain.

Runs have sorted log records referring to page_ids within a certain interval. This interval is not necessarily, and unlikely, continuous, meaning that not all page_ids defined by the interval are present in the run. The more absent page_ids defined by the interval of a run, the higher is the probability of false positives, i.e., the filter indicates that an absent page_id is present. As an example, storing the minimum and maximum page_ids for a run that has only odd page_ids has a false positive ratio of about 50%, assuming that pages are queried following a uniform distribution. Therefore, in addition to the minimum and maximum page_id of each run file, it is desirable to store a more detailed "summary" about the distribution of page_ids within the run. Considering that the page_ids in a run are known prior to writing the run file to disk, it is possible to use such information to determine if applying a filter is worth the additional costs (in the case where there is only one filter format) as well as to tailor a filter that is more suitable for each run file (in the case where runs are allowed to have different filter formats).

Finally, even if it is unlikely that a page does not have log records in every run (assuming on-demand restore) and, consequently, not many runs will be filtered, the gain from skipping a single run and, consequently, saving an I/O operation should justify the additional costs.

## 4.2 Finding the block

Once a run file that may contain the desired log record is determined, the problem then is how to directly access the portion of the file where the record might be, instead of searching the whole file. Therefore, additional information for zones of the file must also be maintained, similar to the filter information for each run mentioned in the previous section.

By dividing the run file in "blocks" of fixed size, it is possible to have information for each block about the log records it contains. Such information may comprise the page_id of the first log record in the block, and, therefore, it acts like a "separator key" in a B-tree. The index for a single run consists of a list of separator keys that represent the page_id of the first log record of each block in the run, as well as the offset of each block within the run file. From that point of view, it is as if the list of separator keys is a variable-length root node (large enough to contain all separator keys of a run) of a B-tree of height 1, and the fixed-length blocks are the leaf nodes of this said B-tree. The log archive index is then defined as a list that contains the indexes for every run in the log archive.

While fixed-length blocks may consume additional storage space in the cases where the next log record does not fit in the remaining space of that zone, the storage space "lost" in these cases should be small enough to a point where it justifies the gain of reduced complexity. The index of a run is then stored alongside the run file itself. The example of Figure 4 shows a run file divided by blocks of fixed size (say, 1KB). For each "data block", i.e., blocks that contain log records, there is a separator key. If we have 50 "data blocks" and each separator key has 32 bytes, the total amount of additional space required for storing these entries is 1600 Bytes (50 * 32), i.e., two additional "index blocks" are required at the end of the run file.

When the desired run file is known, all that has to be done is read this list of separator keys to determine, based on the matching key value, the offset of the block where the log records for the desired page starts.

Figure 5 illustrates the design of a run index. Each run has a list of all separator keys within that run, each one pointing to the block where the first log record refers to the corresponding page_id. Since the log records within a run are sorted by the

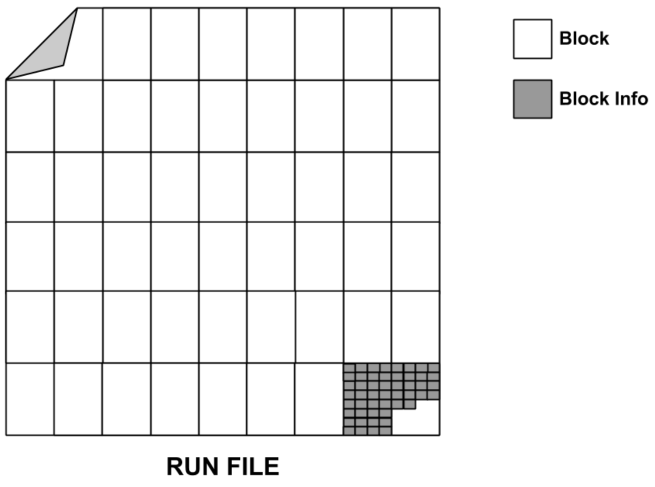

**Fig. 4.** Internal organization of a run file.

page_ids, to determine in which block the log records that refer to a certain page_id start, a binary search based on the range of page_ids between two separator keys can be employed. It is important to mention that, if the log records that refer to the desired page_id are in the beginning of a block, the previous block must also be read, since it is likely that log records referring to the same page_id are also present in the end of the previous block. In addition, it is undesirable to have multiple blocks with log records referring to a single page_id, as this would require all of these blocks to be read.

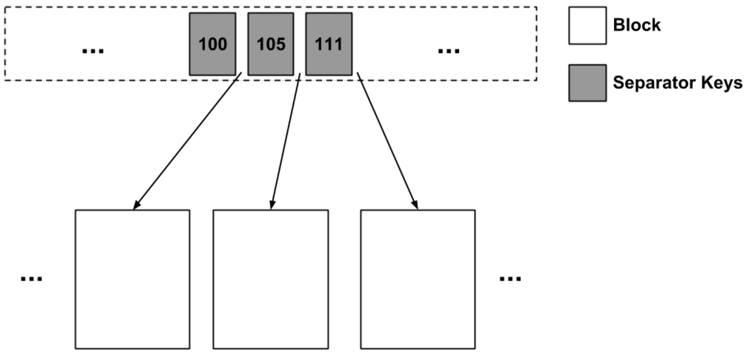

**Fig. 5.** Relation between blocks and separator keys in a run index.

Finally, the performance gains of this strategy are combined with the ones from filtering run files, providing access to any log record efficient enough to realize the idea of on-demand restore.

### 4.3 Concurrency

By analyzing the concurrency control requirements of the proposed archive index structure, there are three types of processes involved: the archiver process, the merger processes and the restorer process. During normal database operation, the archiver process is running, reading log records from the temporary log and writing them into sorted run files in the log archive. The merger processes runs asynchronously in the background merging run files. After a media failure, the restorer process is triggered and it accesses the archived log records by querying the archive index in order to replay the operations and restore the most recent state of each database page. Even if multiple transactions trigger the on-demand restore of pages concurrently, the access to the disk where the log archive is stored is still sequential. Therefore, each transaction emits requests for restoring a certain page. These requests are received by the restorer process, which is responsible for scheduling and processing them. While the media restore is taking place, there are two alternatives on how these three types of processes interact with each other.

First, we assume that the merge of runs does not take place while there are still pages to be restored. Stopping the archiver process, in addition to the merger process, is not desirable, as it would cause log records to accumulate in the temporary log, since new transactions are being executed. If this was the case, when the restore of all pages completes and the archiver process restarts, the volume of log records to be archived would be bigger. Furthermore, if the temporary log contains too many log records, and since it is not indexed, it might be expensive to replay all its contents in case of a restore from a media failure. Thus, stopping only the merger process is the simplest approach, as it does not add much limitations and requires a rather simply concurrency control to synchronize the access to the archive index structure by the archiver process and restorer process.

The second alternative is to allow the merger process to merge existing runs while there are still database pages to be restored. This behavior is desired in a scenario where pages are constantly being restored during normal database processing. A new level of complexity is introduced, as there is the need of a much more complex synchronization to access the index structure in order to prevent incorrect behaviors. A more detailed view from the inner operation of each of the types of processes is required.

From the perspective of the writer process, a new run is written to a temporary file and after the runs if finished, the file is renamed to its permanent name. Since renaming a file is an atomic operation, the index structure, i.e., the list of "block information" entries for that run, is made available only after successfully creating the run file. In case there is a failure in-between, the index structure is normally loaded from the run file at startup.

The merger process merges two adjacent files (in order to keep the contiguous mapping to LSN ranges) in a temporary file. Similarly to the write operation, after the merge is complete the temporary file is renamed to its permanent name. The deletion of the two original files does not need to be done atomically. In case of a system failure after the temporary file was renamed and before the deletion of the two merge files, the system checks for run files with overlapping LSN ranges at startup and deletes the smaller runs. Therefore, before the two original files are deleted, their in-memory index must be replaced by the new merged index. In order to achieve correctness, before deleting the two older indexes, it must be guaranteed that no other process is still accessing them.

Since the archive index consists of multiple indexes for each individual run, the access synchronization could be kept at the level of a single run index. In this case, the possible outcome of interleaved operations of the restorer process and the archiver process must be carefully considered in order to guarantee the access integrity of the data structure used to store the run indexes. However, analyzing the interactions between these two processes might turn out to be a quite cumbersome task that will not pay off. Therefore, in order to keep the design less complex, the restorer process and the archiver process may request an exclusive access to the whole archive index structure (indexes of all runs).

Finally, regarding synchronization between the restorer process and the merge process, the only requirement, as mentioned before, is that the deletion of an index of a run cannot occur while the restorer process is accessing that same index. Furthermore, the restorer process must handle the situation where the portion of the indexed it was going to access was deleted as a result of a merge. In these cases, the restorer process must retry the query from the beginning, being able to access the new index structure created for the new merged file. Again, if both restorer process and merger process require an exclusive access to the whole archive index structure, these guarantees are achieved without introducing further complexity to the design.

### 4.4 Caching

In a single-pass restore operation, the whole log archive is read only a single time in order to replay the log records. The on-demand access, as a requirement for instant restore, deviates from this single-pass behavior. If a block is read from the archive to memory only to replay a portion of the log records it contains and it is thrown away afterwards, the same block will need to be read again in the future for replaying the remaining log records. For this reason, it is desirable to keep these unused log records in memory, every time a block is read. This caching of log records not only avoids the need of re-reading the same block over and over, but also enables a faster response time for queries for log records present in the cache.

However, the caching does not guarantee a single pass through the whole log archive. Consider the scenario where a block is read into memory and the unused portion of log records is kept in the cache. If the run to which that block pertains was merged afterwards, the log records currently in cache are now part of a different block. This different block might be read into memory as a consequence of a query for another page_id and thus the same log records are read into memory again together with the block. In order to avoid this, the cache may proactively try to merge cached portions of the log, making it more efficient. However, this would further increase the complexity. A better alternative to workaround this behavior is to prevent runs to be merged while restoring is taking place, as mentioned in the previous section. Still, even if the caching of log records does not guarantee a single-pass property, it is effective to keep the number of passes to a minimum desired level.

Even with all the advantages mentioned above, there are important aspects that must be considered. Caching introduces additional memory consumption costs as well as the requirement for efficient memory management, since the data structure used would have to manage small variable-length portions of blocks, instead of blocks themselves. Furthermore, caching may not even pay off if the log volume is typically much lower than the data volume (i.e., backup), which is quite common. On the other hand, caching becomes more interesting if we consider using the partially-sorted log archive for restart as well (not just restore). Thus, a detailed analysis of the gains and the implied trade-offs must be made in order to determine how desirable a caching mechanism is.

## 5 Conclusion

The core concept to realize the instant restore strategy is to restore single database pages on demand. To achieve this, efficient access to log records pertaining to a certain page is required. Since the log records are already stored partially-sorted in a log archive, there is the need for a structure to provide such efficient access at the same time it enables efficient loads of large runs of sorted log records.

B-trees arise as the most common data structure for indexing. Partitioned B-trees particularly offer a better trade-off between search and load performance in the context of log archiving. Costs related to large load operations and index maintenance are not reduced, but rather than that, they are postponed. However, in order to maintain the data structure resilient to system failures, new levels of complexity must be introduced. Simply reusing the existing logging infrastructure of regular B-trees seems to be not a good approach, since it generates new log records while already indexing old log records.

As an alternative, zone filters and zone indexes indeed offer a load performance improvement, at the same time it enables efficient search operations. Since the data structure required for zone filters and zone indexes tend to be small, storing and maintaining them can be realized in a simple way to guarantee atomicity in the case of system failures.

By employing zone filters for each run file in a partially-sorted log archive, it is possible to skip run files that do not contain the desired log record. Since runs can be large files, even if not many runs are discarded, skipping a single run offers a gain of one I/O operation that may already justify the additional costs. In addition to that, information about fixed-length blocks within a run is stored together within the run file in the form of a zone index. This information enables to access the block (offset in the run file) in which the log records of a certain page start within that run.

In order to keep a good performance and response time for operations accessing the index structure, concurrency and caching aspects must also be considered. Even if the performance requirements of the proposed approach are yet unclear, its design is flexible enough to enable the composition of new techniques to fulfill future requirements. This approach is expected to offer a good solution for the requirements needed for realizing the on-demand restore of pages.

## References

1. D. Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.
2. G. Graefe. Sorting And Indexing With Partitioned B-Trees. In *Proc. CIDR*, 2003.
3. G. Graefe, W. Guy, and C. Sauer. Instant recovery with write-ahead logging: Page repair, system restart, and media restore. *Synthesis Lectures on Data Management*, 6(5):1–85, 2014.
4. G. Graefe and H. A. Kuno. Fast Loads and Queries. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 2:31–72, 2010.
5. T. Haerder and A. Reuter. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.
6. R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proc. EDBT*, pages 24–35, 2009.
7. G. Moerkotte. Small Materialized Aggregates: A Light-Weight Index Structure for Data Warehousing. In *Proc. VLDB*, pages 476–487, 1998.
8. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.
9. C. Sauer, G. Graefe, and T. Härder. Single-pass restore after a media failure. In *Proc. BTW, LNI 241*, pages 217–236, 2015.