Logical recovery from single-page failures
Goetz Graefe; Bernhard Seeger
Hewlett-Packard Laboratories; Philipps-Universität Marburg
goetz.graefe@hp.com; seeger@informatik.uni-marburg.de

Abstract: Modern hardware technologies and ever-increasing data sizes increase probability and frequency of local storage failures, e.g., unrecoverable read errors on individual disk sectors or pages on flash storage. Our prior work has formalized single-page failures and outlined efficient methods for their detection and recovery. These prior techniques rely on old backup copies of individual pages, e.g., as part of a database backup or as old versions retained after a page migration. Those might not be available, however, e.g., after recent index creation in “non-logged” or “allocation-only logging” mode, which industrial database products commonly use.
The present paper introduces techniques for single-page recovery without backup copies, e.g., pages of new indexes created in allocation-only logging mode. By rederiving lost contents of individual pages, these techniques enable efficient recovery of data lost due to damaged storage structures or storage devices. Recovery performance depends on the size of the failure and of the required data sources; it is independent of the sizes of device, index structure, etc.

1 Introduction
Efficient recovery from transaction, media, and system failures has long been a hallmark of database technology and has been employed in other domains as well, e.g., metadata in modern file systems. All these recovery techniques relied on copies of old and new values of pages, records, and individual fields. For example, when a transaction fails and needs to roll back its updates, “undo” log records or “undo” components of “redo-undo” log records permits copying old values into the appropriate pages and records. Logical and “physiological” logging rely on the same principle; their logical aspect is the location of the record, which might have moved from one page to another between the original “do” action and the “undo” action, e.g., due to a node split in a B-tree index.
Single-page failures, a fourth class of database failures that matches failure scenarios of high-density disk drives and of present and future semi-conductor storage, has recently been described [GH12]. The suggested recovery techniques for an unreadable or inconsistent page rely on recovery logs as commonly used in database systems and on an earlier copy of the page. These copies may come from a database backup, delayed deallocation after a page migration, or a log record describing initial formatting of a page newly allocated from free space. Both the earlier page image and the recovery log rely on copying and on copies, either of an entire page or of individual records and field values.
For some operations commonly used in database systems, however, these techniques do not work. For example, creation of a secondary index for a table usually is implemented without logging the individual index entries or the new index pages. Instead, merely the page allocation actions are logged. For example, if 500 index entries of 16 bytes fit on each database page of 8 KB, logging individual index entries of 16 bytes may take log records of 40 bytes (equivalent to 20 KB of log volume for each database page of 8 KB); logging entire new pages takes 8 KB per page plus a little overhead; and logging only the page allocation may take 80 bytes of log per database page of 8 KB. Thus, allocation-only logging produces about 100× less log volume than full-detail logging.

Before a transaction with non-logged index creation can commit, it must flush all new index entries from the buffer pool to persistent database storage. This has been termed a “no steal – force” policy [HR 83]. If the system suffers a system or media failure soon thereafter, recovery of the newly created secondary index relies on repeating the entire index operation, i.e., it reaches back to the base table. Recovery of the index contents from log records is not possible after allocation-only logging.

Figure 1 schematically illustrates this recovery process. Runs in the external merge sort are shown as partitions of a B-tree; the final merge step creates the desired B-tree index; and recovery after failure reaches back to the base table, i.e., repeats the entire effort of index creation.

After a successful index creation, the entire new index must be backed up, too, during the next backup of the recovery log; otherwise, the log backup may contain log records describing updates to index pages that cannot be repeated in a possible later failure. In other words, while “allocation-only logging” optimizes the size of the recovery log, it does not reduce the volume of the log backup. Thus, existing techniques for “non-logged” or “allocation-only logged” index creation solve only part of the problem.

These techniques also fail to enable recovery of single-page failures in the newly created index, at least until the index has been backed up as part of a database backup or a log backup. If an individual index page (or a small set of index pages) becomes unreadable, it is also unrecoverable except by dropping and recreating the entire index. Thus, a new technique is needed for single-page recovery for database pages filled by non-logged operations.
This lack of support includes, unfortunately, many database utilities that modify the logical and physical database design. In addition to index creation, this includes initial materialization of views as well as splitting a table in two vertical slices in order to permit new one-to-many or many-to-one relationships. For example, if the original database design is limited to one address per customer, yet an application change requires multiple shipping addresses (e.g., to enable gift orders), the original table for customers must be split into two.

It is desirable to enable recovery from single-page failures even after changes in the logical and physical database design. Prior designs and techniques failed at this goal; therefore, the present paper proposes a design to fill this void. The foundation is focused repetition of an operation such as index creation. In order to ensure efficiency, the design relies on specific data structures and on incremental creation and optimization of new data structures, e.g., new secondary indexes.

Future work will cover more complex data structures, e.g., materialized views and their initial materialization. Once a view has been materialized, the techniques here suffice to add secondary indexes. For the initial materialization step, the scan techniques (including snapshot isolation by single-page “undo” recovery) need to be extended to multiple data sources.

The remainder of this paper is organized as follows. The next section reviews a variety of related work that influenced and informs the proposed design. Section 3 lays out the principles and foundation for the design, whereupon Section 4 applies them to a specific case of single-page recovery in newly created secondary indexes. Section 5 demonstrates the performance potential with a few preliminary experiments and Section 6 offers our conclusions from this research effort.

2 Related prior work

The following reviews relevant system designs, failure classes and their recovery techniques, and data structures. The subsequent sections combine those prior techniques into logical recovery from small failures by re-deriving lost contents.

2.1 Bubba

A proposal for data storage, reliability, and recovery in the Bubba database machine [CAB 88] suggested that a lost index may be recovered from one or more other indexes, with those indexes containing a super-set of the columns in the lost index. Novel at the time, it seems that this design has never been implemented.

2.2 Vertica

In the Vertica columnar database [LFV 12], tables are stored as one or more “projections.” Each projection has a sort order. A “super-projection” is similar to a traditional primary index as it contains all rows and all columns of the table. It is different from a traditional primary index as it may include functionally dependent columns from other tables and it uses
a separate file for each column. Other projections store a subset of columns for all rows, similar to a traditional secondary index.

For reliability, each projection may be stored multiple times for “k-safety” against data loss even in the case of losing k copies (e.g., 2 copies for 1-safety). These “buddy projections” must have the same column set (as well as all rows of the table) but may have a different sort order. In case of a failure, recovery of one projection may use its buddy projection, i.e., may require a sort operation.

In contrast, our work focuses on efficient recovery of parts of an index (e.g., a single page or a few pages) that does not require a full sort operation. Recovery after loss within a single column only is left to future work.

2.3 Traditional index operations

Traditional index operations include creation, defragmentation, removal, consistency check and repair, etc. of primary (clustered) and secondary (non-clustered) indexes on tables and (materialized) views. These operations may run offline (with a shared lock on the table) or online (permitting updates by concurrent transactions), logged (each index entry or each index page) or non-logged (also known as allocation-only logging), using temporary space (for runs of the preparatory external merge sort) or in target space (recycling target space during the final merge), serial or parallel (in scan, sort, or write), etc. The indexes may be hashed or sorted, single- or multi-dimensional, unique or non-unique (permitting duplicate key values), uncompressed or compressed (using run-length encoding, prefix- and suffix truncation [BU 77], etc.), using pointers (e.g., record identifiers or key values in the primary index) or bitmaps, partitioned or not, versioned (for snapshot isolation and multi-version concurrency control) or not, etc.

This variety of indexes and of index operations may seem bewildering except for developers working on or with full-featured commercial databases. Therefore, we focus here on creation of secondary B-tree indexes. Applying the principles introduced here to all indexes and index operations certainly is a significant development effort but not a research problem. The exception to this statement are online index operations, which are considered later, e.g., in Section 4.4.

With respect to keeping runs in the target space of the database, we assume sufficient space to keep 2-3 copies of each index entry, i.e., runs are not deleted immediately after a merge operation but somewhat later. Moreover, we assume that logging all new index entries, with a log record per index recovery or per index page, is prohibitive due to the required space in the recovery log, as is the case in most real-world index operations.

2.4 Traditional failure classes

The traditional failure classes are transaction failures, media failures, and system failures [G 78]. Early designs for recovery from those failures relied on “idempotent” recovery actions, which all relied on byte-for-byte copying but also limited the finest granularity of locking to page locks. Subsequent concurrency control and recovery techniques enable row-
level locking (including key range locking in B-tree indexes) but still copy records and fields to and from the recovery log [G12, HR 83, MHL 92, W 91]. In other words, detailed logging remains required and both logging and recovery still copy field values, records, or pages.

2.5 Single-page failures

This recently proposed fourth failure class and its proposed recovery technique [GH12] is another prototypical example for “recovery by copy.” Starting with a backup page copy, replaying “redo” log records obtains an up-to-date instance of a data page. Recovery might be achieved after reading tens or at most hundreds of log records, i.e., within about a second and thus much faster than media recovery.

The default recovery technique for single-page failure relies on traditional backups and a traditional recovery log, i.e., copying records or field values between database page and recovery log. The locations of both the backup page and the most recent log record are kept in the “page recovery index,” one per database. Recovery of a single-page failure from a formatting log record (i.e., the operation immediately following allocation from free space) has aspects of both recovery from a copy and recovery by re-deriving contents. On the other hand, the log record with formatting parameters is more a compressed copy than a recipe for deriving database contents, which is the focus of the recovery techniques in this paper.

Logical recovery, as proposed in a subsequent section, is similar to single-page recovery in the sense that it recovers individual pages rather than an entire device. Logical recovery recovers an entire key range, however, not just a single page at a time. More significantly, it recovers the key range by re-deriving the lost contents by repeating the original logic rather than copying the lost contents from a backup and from log records.

2.6 Self-repairing indexes

Self-repairing indexes [GKS12] combine two facilities, self-diagnosing faults and self-healing. Faults may be unreadable storage pages or implausible page contents, e.g., an inconsistency between a parent node and a child node or between two neighboring child nodes. Self-healing requires efficient automatic recovery of the correct, up-to-date page contents.

Symmetric fence keys in each B-tree page enable continuous, incremental, and comprehensive verification of all cross-node invariants of a B-tree structure [GS09], i.e., self-diagnosing indexes. Self-healing can be achieved by moving information for single-page recovery from the database-wide page recovery index into the index itself. In a B-tree variant with only one (incoming) pointer per node, e.g., a Foster B-tree [GKK12], this information can be associated with each child pointer.

The resulting self-repairing B-tree is just one example for localized detection and recovery of errors in a complex data structure. Frequent local (and thus inexpensive) checks enable efficient root cause analysis during quality assurance as well as reliable data structures after
deployment. Embedding and maintaining consistency information within the B-tree data structure requires a single pointer to each node, e.g., Foster B-trees.

Compared to the initial design for recovery from single-page failures, self-repairing B-trees differ in the bookkeeping, i.e., keeping track of the latest log record for each page without a page recovery index. Logging and recovery are unchanged from the original design for single-page failures, i.e., continue to rely on copying records and field values between database page and recovery log.

2.7 Partitioned B-trees

Partitioned B-trees [G03] are standard B-trees with a partition identifier added as prefix to the user-defined index key. Prefix truncation (compression) reduces the additional storage requirement to one integer per page in most pages. The resulting indexes preserve sort order (within each partition), enable ordered scans (by merging), and support (reasonably) efficient query execution for key range predicates. In addition to the standard advantages of B-trees, partitioned B-trees enable incremental creation and optimization. Each step produces a valid index, even the initial extraction of index entries from the base table and even if a merge step covers only a partial key range within the index.

Partitioned B-trees can be used for efficient loading (adding new data as one or more memory-sized partitions – the advantage is that loading proceeds at full sequential write bandwidth but the index is immediately complete and searchable), for index creation by external merge sort (with each run stored as a B-tree partition, not as a traditional run file – the advantage is that the index is complete and can be searched immediately after run generation), and for sorting (external merge sort with deep read-ahead, parallel merge using range partitioning, ‘pause and resume’ without delay and without duplicated effort).

2.8 Adaptive indexing

If the optimal set of indexes for a database or for a table cannot be predicted, adaptive indexing creates useful indexes as side effect of query execution. If the set of desired indexes can be predicted, they can be defined in the catalogs but data movement and data structure optimization can be accomplished as side effect of query execution. Alternatively, the catalogs might also indicate indexes that are permissible or prohibited. An index tuning tool may set such properties, and query processing can take the information as guidance during query optimization and query execution. For example, query optimization may assume that a desirable index will exist at run-time even if it does not exist at compile-time. Repeated query execution will, as side effects and in multiple steps, create and optimize such an index.

There are two forms of adaptive indexing, plus hybrids [IMK11]: database cracking [IKM07] optimizes in-memory column stores, whereas adaptive merging [GK10] optimizes partitioned B-trees. Each form of adaptive indexing has two types of steps: initial index creation and incremental index optimization. In database cracking, the initial index is a single unsorted partition and each optimization step divides an existing partition using a Pivot key equal to the key value of an exact-match query or to an end point of a range query. In
adaptive merging, run generation produces the initial index and each optimization step applies one merge step to key values within or around the range query or the exact-match query.

For side effects with acceptable efficiency, concurrency control and recovery must be practically free. Full detailed logging is not acceptable. Concurrency control is a solvable problem, because index optimization is merely a change in physical database representation, not in logical database contents; thus, merely latches are required but not locks [GHI 12]. Logging and recovery, however, could introduce excessive overhead to query execution.

A simple and effective technique is to let only one thread (one query) perform a merge step for a given key range and let all other queries access that key range in the existing data structure in read-only mode. When the merge step is complete, all new query executions may use the merge result; all existing executions may continue using the older partitions in read-only mode. Only when all existing executions are complete, the older partitions may be removed. In other words, it is not the merge step that removes its input partitions and reclaims their space but an asynchronous process invoked based on usage. A pragmatic implementation might use fixed key ranges and reference counting for each key range, which permits space reclamation as soon as possible but also multiple threads (queries) merging disjoint key ranges at the same time.

3 Recovery by re-deriving contents – principles

In order to re-derive lost database contents, the source of the last derivation step must remain available. For example, if the result of a merge step is lost, it can be recovered if the merge inputs still exist. If only a small section of the merge result is lost, and if small sections of source and destination, e.g., specific key ranges, can be accessed directly, then the lost section can be recovered very efficiently.

Thus, we propose to retain these merge runs even after index creation is complete. Delayed removal of such intermediate files adds little cost; storage space is nowadays plentiful and inexpensive in most environments. There is no need to include these merge runs in the next database backup, and in fact the next database backup enables more efficient recovery techniques than logical single-page recovery.

This is somewhat similar to data processing of years past, with a master tape and files with recent changes, with new master tapes created by merging old master tape and all changes, and with a lost master recovered by repeating the appropriate merge step. The difference of the proposed techniques and those “ancient” techniques is that only the required key ranges are recovered, with no need to repeat completed steps in their entirety.

The proposed recovery techniques are quite different from recovery using copies or replicas. If each data page exists twice (or thrice, or even more times), then a single lost copy can be readily re-created simply by copying. The proposed technique holds multiple (typically two) copies of the logical contents but only one copy of each physical page. If some data page is lost, there is no way to recover the loss by copying. Instead, a processing step must be
performed. In the simplest case, which is the focus here, a processing step must be repeated. Ideally, it is repeated only partially, optimized to re-produce the lost data and no more.

Logical recovery by re-deriving database contents and their data structures requires that data processing steps are non-destructive. In other words, rather than modifying an existing structure, the original derivation step (as well as the re-derivation steps during logical recovery) must merely read the existing data structure and create new ones. For example, database cracking (i.e., adaptive index improvements in an in-memory column store by partitioning steps similar to those of quicksort) does not qualify for logical recovery, because the partitioning steps occur in place in order to minimize the number of data items that need to move. On the other hand, in the other prototypical adaptive indexing technique, adaptive merging, each step merges multiple runs and put the result into a different run. Thus, even if adaptive merging keeps all runs in a single B-tree, the merge input remains unchanged in each merge step. Thus, adaptive merging and its data structures can serve as prototypical use case for logical recovery by re-deriving lost data pages, but the technique also applies to other index formats and indexing techniques as well as to materialized and indexed views.

The following discussion focuses on failure and recovery of leaf pages in B-tree indexes. Non-leaf pages, typically only 1% to 1‰ of all pages in a B-tree, should be fully logged such that existing recovery techniques suffice, e.g., log-based single-page recovery [GH 12].

4 Recovery of index pages and key ranges

Partitioned B-trees and adaptive merging lend themselves to logical single-page recovery, i.e., re-deriving lost contents from retained prior data. This is due to index operations proceeding in distinct simple steps with valid and useful states in between, even if each merge step merges only a small key range. The following sub-sections cover index creation, index optimization, index maintenance, and recovery after updates.

4.1 Index creation

Creation of new secondary B-tree indexes usually employs an external merge sort. The first step, run generation, scans the table’s primary data structure, extracts all required information for future index entries, and produces initial runs for the external merge sort, perhaps as partitions in a partitioned B-tree. For the discussion here, index creation is complete when all index entries are in the future index structure. Index optimization merges partitions in order to organize all index entries into a single sorted sequence with query and update efficiency of a traditional B-tree.

Should one of the initial runs become unreadable, it can be recovered if the appropriate part of the primary data structure can be identified, retrieved, and re-sorted. If only a key range within an initial run becomes unreadable, this key range translates to a predicate when re-scanning the primary data structure, which reduces the sort effort but not the scan effort.

Figure 2 illustrates the technique, where recovery of a single run (center, red) reaches back to the original table (left, blue) but scans only a part of it (left, red). The final index (right) does
not participate in this scenario. It might not even exist yet and is thus drawn with dashed lines. A comparison with Figure 1 identifies the difference: whereas traditional logical recovery can re-derive only the final index and only from the original table using a complete table scan, single-step recovery can re-derive individual runs by scanning only parts of the original table.

![Figure 2. Single-step recovery: run generation.](image)

Such recovery works very efficiently if each partition in the new index maps to a specific segment of the source data structure. Ideally, a table’s primary data structure is a B-tree index (a clustered index also known as index-organized table), the scan providing input to run generation uses the index order (as opposed to an allocation-order scan), and run generation proceeds in read-sort-write cycles (e.g., using quicksort, not using a continuous process such as replacement selection). In this case, the read-sort-write cycles and the index-order scan provide a simple mapping from a run in the new index to a key range in the data source, and the primary index provides efficient access to just that key range. In contrast, run generation by replacement selection permits only less precise mappings, and an allocation-order scan or a primary data structure other than an index requires an unusual predicate on a page range rather than a standard predicate on a key range.

If only a single page within a run is unreadable, it can be re-derived efficiently using a partial scan of the original table. Differently from the partial table scan in Figure 2, this partial scan applies a predicate matching the key range of the unreadable page. If a B-tree represents each run or if a single partitioned B-tree represents all runs, the parent page in the B-tree structure can provide the required key range.

![Figure 3. Single-page recovery by run generation.](image)
Figure 3 illustrates recovery of a single page in a partition. Scanning the appropriate fraction of the data source quickly produces the index entries that belong into the unreadable page of the index partitions.

### 4.2 Index optimization

Run generation is only the first step of index creation. The second step, required only for very large tables and indexes, merges initial runs to form intermediate runs. The third step merges a small set of final runs to form the desired index.

Should an intermediate run or a key range within such a run become unreadable, logical recovery repeats the merge logic for that key range. The same is true for the final merge step producing the final, fully optimized index: Should a part of the final index become unreadable, it can be recovered by re-merging data from the final runs.

Repeating a merge step for the entire domain of user-defined keys requires that the input runs have been retained, i.e., not deleted immediately during or after the merge step. Repeating a merge step only for a limited key range requires that the data in the required key range can be retrieved efficiently, i.e., these runs are organized in a partitioned B-tree.

![Figure 4. Single-step recovery by merging.](image)

Figure 4 illustrates single-step recovery from intermediate runs, i.e., it complements the single-step recovery illustrated in Figure 2. If intermediate runs still exist, recovery of the final index can omit table scan and run generation, instead repeating only the final merge step.

![Figure 5. Single-page recovery by merging.](image)
Figure 5 illustrates single-page recovery by partially repeating a merge step. If a single page (or a small set of pages) is unreadable in the final index, intermediate runs stored in a B-tree permit direct access to the required key range. A short merge operation can reproduce precisely the unreadable key range without wasting any effort on other key ranges.

During merge steps in a partitioned B-tree, e.g., during adaptive merging, a limited merge fan-in reduces the memory allocation required for the side effect of query execution. Thus, there is a tradeoff between efficiency of a merge step (favoring a high merge fan-in) and the overhead of memory allocations (favoring a small merge fan-in). Logical recovery adds another consideration: a small merge fan-in permits logical recovery from less source data. In other words, the reliability of the storage technology and the probability of data loss requiring logical recovery influence the heuristics setting the merge fan-in. Of course, the same arguments apply to index creation and the memory allocation for run generation.

### 4.3 Maintenance of existing indexes

An entire index is fully optimized when only a single partition remains. In that state, searches and updates in a partitioned B-tree are just as efficient as in a traditional B-tree without partitions. This single final partition is permanent in the sense that it will not serve as merge input in future merge steps. A key range is fully optimized when all index entries within the key range are in the partition intended to be the only permanent partition. Note that an index or a key range can lose this status, e.g., when a large load operation adds new partitions to the B-tree.

When a key range is fully optimized, updates modify the permanent partition. Otherwise, it might be most pragmatic to leave each partition read-only once a merge step has created it, while updates go to one or two dedicated partitions that absorb all updates. These partitions remain in memory and sort the updates similarly to run generation by replacement selection. Read-only and read-write partitions require different techniques for logical recovery.

Read-only partitions can be recovered, if necessary, simply by repeating the run generation logic or the merge step that created them (as discussed in Section 4.2). Read-write partitions require recovery in two stages. First, the original partition contents are recovered by repeating the logic that created the partition. Second, single-page “redo” recovery must carry pages forward by finding and replaying the appropriate log records.

![Figure 6. Traditional log-based recovery.](image)
Figure 6 sketches traditional recovery based on write-ahead logging. Update operations produce log records with before- and after-images of database records, index entries, and individual column values, whereas “redo” and “undo” recovery copy from the log into the database. This technique also applies to read-write partitions during the second stage of recovery.

Read-write partitions absorb not only insertions but also deletions. A deletion in a read-only partition inserts a “tombstone” or “anti-matter” record into the appropriate read-write partition. When merged with a valid record, neither anti-matter nor valid record with the same key value appear a merge output or a query result. An update is a deletion and an insertion.

4.4 Updates between index operations and their recovery

If an index contains the same data item twice, typically once in a recent merge output and once in a retained merge input, then only the latter is used to answer queries. If a recent merge output is a read-write partition, then updates modify only that partition. In other words, each update is applied only once. A partition that has been merged into another one is frozen (no further queries, no further updates) and therefore can serve as backup (or input) during recovery. Once a former merge output has served as merge input, the oldest merge input is no longer useful as backup and is dropped. In other words, each data item exists only in two places, not in three or more places.

Online index operations permit concurrent updates, contrary to offline index operations that retain read locks on the entire table for the operation’s entire duration. Depending on the precise timing of updates and in particular properties of index operations, it may be impossible to repeat a merge operation if the merge output subsequently becomes unreadable. Moreover, a merge output may become unreadable only after some updates subsequent to the merge operation. In both those cases, log-based recovery (based on write-ahead logging during updates) complements logical recovery.

In order to repeat a prior merge step in spite of subsequent updates to the merge input, rolling back pages of the merge input ensures precise repetition. Thus, if logical recovery encounters a page with a PageLSN (timestamp, pointer to a log record) newer than the original merge operation, single-page “undo” recovery based on per-page chains of log records rolls back a temporary page copy in the buffer pool. This design for single-page rollback is very similar to the implementation of snapshot isolation in Oracle databases. After the repeated merge operation is complete, single-page “redo” recovery can repeat updates applied after the original merge operation.
Figure 7. Single-step recovery of run generation and updates.

Figure 7 adds individual updates and their log-based recovery to the scenario of Figure 2. While rollback of merge input pages occurs during the scan feeding the repeated merge operation, roll forward of merge output may occur either in a subsequent step or during the merge operation, i.e., immediately after the merge logic has filled an entire output page.

Figure 8. Single-page recovery of run generation and updates.

Figure 8 illustrates logical single-page recovery by repeating run generation. The scan repeats only that part of the source scan that corresponds to the run with lost pages; the run generation logic consumes all scanned data but overwrites only the lost pages. The scan integrates rollback of pages in the original table, if required. Note that single-page rollback must precede predicate evaluation. Writing the run also rolls forward the page (or pages) found unreadable in the run. Alternatively, single-page “redo” recovery may be a separate step.
Figure 9. Single-step recovery of merging and updates.

Figure 9 illustrates recovery of an entire merge operation with updates to both merge input and merge output. The scans feeding the merge logic roll back input pages in order to enable precise repetition of the original merge operation; the merge output pages are rolled forward either immediately or in a subsequent step.

Figure 10. Single-page recovery of merging and updates.

Finally, Figure 10 illustrates single-page recovery in the final index by extracting and merging appropriate pages in intermediate partitions. Log-based single-page “undo” applied to all merge input pages ensures precise repetition of the merge logic for the required key range; log-based single-page “redo” of the merge output ensure correct final index contents and structure.

4.5 Summary of logical recovery for indexes

In summary, logical recovery for index operations merely requires retaining data structures, i.e., delaying their removal from temporary storage space. Doing so enables efficient recovery of both large and small failures, e.g., single-page failures in intermediate data
structures (e.g., runs during index creation) and in final index structures. While the prior design for single-page recovery requires extensive logging, the new design relies on data structures created in the standard sequence of steps.

In addition to offline index operations (with the underlying table locked in read-only mode), logical recovery also support online index operations (i.e., updates by concurrent user transactions). Concurrent updates must be logged using standard write-ahead logging with appropriate “undo” and “redo” information. Using techniques known from Oracle’s implementation of snapshot isolation, single-page “undo” takes the data source back to the time of the original index creation step; using techniques from the original design for single-page recovery, single-page “redo” recovery rolls the merge output forward to reflect updates subsequent to the original merge step.

5 Performance and scalability

This section reports the results of a preliminary performance comparison of logical recovery with the traditional approaches to recovery, with the focus on secondary indexes and their creation in a row-store database. The first set of experiments is designed to assess the performance improvements due to logical recovery of an entire index in comparison to reloading the entire index from scratch. A second set of experiments focuses on cases where only a few pages of the index need recovery. All these experiments assume a static scenario with no updates after index creation. In other words, the experiments reflect the technique illustrated in Figure 4 and Figure 5.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main memory</td>
<td>20, 80 MB</td>
</tr>
<tr>
<td>Page size</td>
<td>8 KB</td>
</tr>
<tr>
<td>Tuple size</td>
<td>40, 200, 500 bytes</td>
</tr>
<tr>
<td>Tuple count</td>
<td>100 M</td>
</tr>
<tr>
<td>Key size</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Pointer size</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

Figure 11. Experimental parameters.

5.1 Experimental environment

The experiments use a base relation with 100’000’000 tuples of 200 bytes, for a table size of 20 GB. A secondary B-tree index is created on an attribute of size 8 bytes. An index entry of this B-tree occupies 16 bytes; 8 bytes for the search key (attribute) and 8 bytes for a pointer.
Search keys are to be uniformly distributed. All experiments use a page size of 8 KB, a setting that is in agreement with the recommendations of database vendors.

All software used in these experiments are implemented in Java using the open-source XXL library [BBD 01]. In particular, the following components of XXL are employed: external sorting algorithm with replacement-selection, B-tree implementation, and index loading algorithm. All experiments are conducted on a machine with an Intel Core i7 2600 / 3.4 GHz with 8 GB of main memory and a WD Caviar Black WD1002FAEX disk with 64 MB cache. Rather than using the ordinary I/O interface, this implementation uses the raw interface to avoid the interference of buffering in the operating systems. Note however, that caching and prefetching are common features within modern disks today. This standard feature remained enabled during the experiments. However, each recovery experiment started with clearing the disk cache (simply by reading useless data). We conducted all experiments for various main memory sizes. Because the observations did not differ qualitatively, the following presents only the results for an available memory of 20 MB or 2'500 pages. Figure 11 summarizes the experimental parameters with default values underlined.

5.2 Logical recovery of an entire index

The first set of experiments addresses the problem of recovering the entire B-tree. The standard recovery method is to create the B-tree from scratch starting from the base relation. For 20 MB main memory, the sorting algorithm generates 40 initial runs in the run-generation phase (each of them occupies about 40 MB of main memory on average due to replacement selection). The run generation phase requires 871 s in total (21.8 s per initial run), while the final merge phase (including creation of the B-tree) only needs 225 s, for a sum of 1'096 s. The standard recovery method thus requires 1'096 seconds, whereas logical recovery invokes only the final merge and thus requires only 225 seconds.

![Figure 12. Recovery times for an entire secondary index.](image)
Figure 12 shows the performance of traditional recovery and logical recovery for memory sizes of 20 MB and 80 MB. Note that results are quite similar. In both memory settings, logical recovery achieves performance benefits of about a factor of five if the initial runs are still available. Even in case that some of the initial runs are not available anymore, it still would be better to reconstruct missing runs rather than using standard recovery (see Figure 2).

The reason for the performance improvements are that there is no need to read the tuples from the base relation again. In order to illustrate the influence of the tuple size, Figure 13 shows the performance of traditional recovery and logical recovery for different tuple sizes. As expected, the performance improvements increases in the tuple sizes, but performance improvements of a factor of 2 can still be achieved even for small tuples of size 40 bytes.

5.3 Logical single-page recovery

A second set of experiments considers the case that not the entire B-tree is lost but only a few of its leaf pages. Recall that traditional recovery does not take advantage of partial failures and must recreate the entire index from the base relation, which takes 1’096 seconds.

We assume that unavailable leaf pages correspond to an adjacent sequence of leaves within the B-tree. This assumption is justified for the following reason: adjacent leaf pages are often physically clustered on a single track of a disk. Thus, a track failure would cause the loss of adjacent leaf pages. We introduce parameter $k$ to express the number of those leaves. If multiple key ranges (page sequences) are lost, logical recovery is invoked for each one in turn.

Recovery of a sequence of adjacent pages performs a range query on each of the final runs. It is therefore beneficial to have a B-tree maintained on the sorted runs. We examine two possibilities for indexing the sorted runs. The first is to create a separate B-tree on each of the runs, while the other is to create a partitioned B-tree over all the runs. The extra time to
create these indexes is similar to the time required to build the final B-tree from the runs (221 s in our experiments).

Figure 14 shows the performance of logical recovery for $k = 1, 10, 100, 500$ and $1'000$ leaf pages. The figure shows a group of three bars (each the average of 10 experiments) for each setting of $k$. The first two bars within a group refer to the case where an index is built on each of the runs, while the third illustrates the performance in case of using a single partitioned B-tree for all runs. The first bar displays only the query time of the range queries performed on the different B-trees. As this does not include the time for opening files and initializing the B-tree, the second bar includes all these costs. It reveals that the cost of these preparatory actions is high and can become the dominant factor for small queries ($k = 1$). The third bar represents the recovery costs in case of using a partitioned B-tree including the costs for opening the corresponding file. Note that the cost of the preparatory actions is almost negligible for the partitioned B-tree. Another positive effect can be observed in case of partitioned B-trees for large values of $k$. The costs for processing queries are substantially smaller for a partitioned B-tree in comparison to using an index on each of the runs. The reason is that the clustering of pages on the disk is substantially better within a single index in comparison to the data being distributed among multiple indexes. Thus, the average cost for reading a page is lower for the partitioned B-tree.

Figure 15 compares these performance results with the ones obtained for traditional recovery. As discussed above, the standard recovery create an index from the base relation again; this takes $1'096$ s in our experiments. If the runs are kept in a partitioned B-tree, logical recovery of a single page failure ($k = 1$) takes less than half a second. Note that the corresponding red bars are not visible anymore, thus the total time is indicated above the bars. In summary, the savings due to logical recovery exceed three orders of magnitude when compared to traditional recovery.

Figure 14. Logical recovery for page failures.
6 Summary and conclusions

In summary, a number of commercial database systems optimize logging during index operations such as index creation. These techniques are known non-logged index creation, minimal logging, or allocation-only logging. These techniques require a “force” policy upon completion as well as log backups including the entire new index. The new technique requires neither flushing the new index to storage nor including the new index in the next log backup.

Moreover, the new techniques permit efficient recovery after small and large failures, e.g., due to locally worn-out flash storage. If an entire intermediate run is lost, its recovery merely repeats the step that created it. If only a few pages are lost, their recovery repeats only the minimal necessary index creation logic. For example, it repeats the logic to merge multiple intermediate runs into the final B-tree tightly limited to the key range of the lost pages.

An experimental performance evaluation demonstrates the efficiency of the new logical recovery techniques. Recovery of an individual page takes a fraction of a second; recovery of multiple contiguous pages proceeds with I/O bandwidth.
References


