Instant recovery for data center savings

Goetz Graefe
Hewlett Packard Laboratories
Palo Alto CA 94304
Goetz.Graefe@HP.com

ABSTRACT

Today's data centers routinely employ triple redundancy, i.e., each disk page of a database or of a key value store is stored three times (or even more, e.g., in database and file system backups). In contrast, write-ahead logging can reduce the cost of database operations and of data centers, assuming suitable techniques for logging, log archiving, backing up, and recovery. The present paper summarizes our work to-date on single-page repair, instant restart, and instant restore; it describes our techniques for self-repairing indexes, single-pass restore, and virtual backups; and outlines the opportunities for single-copy databases, for avoidance of ever taking a backup (full or incremental), yet for availability and reliability matching today's double- or triple-redundant data centers.

1. INTRODUCTION

Data centers are very expensive, from real estate to construction, compute and storage equipment, power and cooling, etc. Reducing data center costs for databases, key value stores, and file systems is our goal, principally by replacing double- or triple-redundancy storage through single copies plus write-ahead logging.

The purpose of double- and triple-redundancy is to enable and ensure continuous service, not only database service but also applications and web service. Any alternative design must provide comparable or better availability and reliability. This paper outlines a vision for single-copy databases with high availability and reliability yet lower costs.

The principal technique is efficient recovery based on traditional write-ahead logging plus a few techniques of moderate complexity. These include per-page chains of log records, partial sorting during log archiving, and indexes for database backups and for log archives. Note that the recovery log formats of some major products such as Oracle and SQL Server already include per-page chains of log records and that some products already compress and aggregate log records during archiving, and other products sort log records during restart and restore operations. The required indexes for backups and log archives are b-trees loaded from sorted streams.

In addition to requiring fewer nodes in a data center, the proposed techniques enable self-repairing indexes and virtual backups. Self-repairing indexes enable continuous comprehensive consistency checks and automatic recovery after failures and defects. Virtual backups produce media or files equal to traditional backups but do so without any involvement of the database server. Efficient virtual backups render traditional backups obsolete, i.e., future database require neither processing nor networking bandwidth for backup operations in addition to query and transaction processing. Efficient restore algorithms provide up-to-date replacement databases without the expense of incremental backups or of log replay.

2. FOUNDATIONS

Following [2], the discussions below assume a simple database system or key-value store implemented on a conventional computer system. The principal hardware assumptions include page-access persistent storage and a buffer pool in volatile memory. Moreover, there is no hardware or software replication or mirroring, except that storage of log records is very reliable. The required "stable storage" often relies on mirrored storage for the recovery log and, if one exists, for the log archive. The techniques in this paper do not address failures in the recovery log or in the log archive, instead assuming the fiction of "stable storage" like prior research and commercial work in database recovery. Similarly, the techniques in this paper do not address memory failures, i.e., correctness of volatile memory such as traditional DRAM.

The only form of redundant storage is write-ahead logging supporting transactions, commit log records, in-place updates of pages and records, "exactly once" log records including rollback (compensation) log records, and checkpoints. Transactions support all ACID properties: atomicity ("all or nothing"), consistency, isolation (concurrency control), and durability (persistence). Atomicity is guaranteed even in cases of transaction failures (rollback), system failures (e.g., operating system crash), media failures (e.g., head scratch), and single-page failures (e.g., local wear). Concurrency control may employ page-level locking as well as record-level locking, key-value locking, or key-range locking.

Figure 1 illustrates the main data structures participating in update processing, system restart after system failures, and restore operations after media failures. Transaction logic modifies images of database pages in the buffer pool and writes appropriate log records to the recovery log. Database backups provide long-term storage for database contents and the log archive provides long-term storage for log records. In case of a system failure, restart ensures up-to-date buffer pool contents from the available database and the recovery log, along with other server state in the transaction manager and the lock manager. In case of a media failure, restore operations combine database backups and log archive into an up-to-date replacement database.

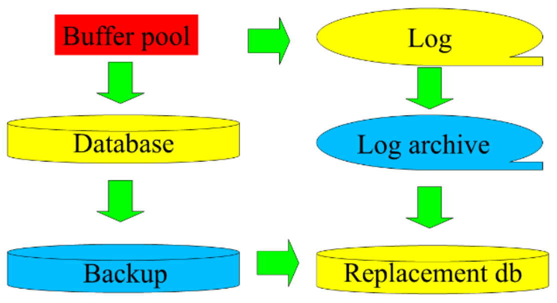

Figure 1. System model and update progagation.

2.1 ARIES

ARIES [6] recovery relies on write-ahead logging [1]. ARIES enables fine granularities of locking, e.g., row-level locking (ARIES/IM [7]) and key-value locking (ARIES/KVL [8]). Transaction rollback is logical, i.e., it performs the reversing update and writes a compensation log record.

Each log record describing a transaction update points to the most recent prior log record of the same transaction. Each rollback log record includes a field called "next undo lsn" (a log sequence number is the address of a log record within the recovery log), which guides rollback after an interruption, e.g., a system failure during transaction rollback. Each database page contains a PageLSN field that indicates the most recent log record reflected in the page image, enabling "exactly once" application of log records to database pages.

2.2 Log shipping

Traditional techniques for high availability require one or more secondary servers running on separate nodes in a cluster or in separate failure domains. Each secondary server holds a complete copy of the database and keeps it up-to-date at all times. Maintenance of such secondary database copies may rely on writing each dirty database page not only to the primary database copy but also to each secondary copy. Alternatively, the primary database server may send its log records to all secondary servers, which apply those log records to their local database copy in continuous log replay and "redo" recovery. Common names for these alternatives are database mirroring and log shipping.

When the primary database server or its network connection fails, a secondary server takes over. Typically, the new primary server rolls back all incomplete transactions.

3. ON-DEMAND PAGE RECOVERY

Traditional write-ahead logging and log-based recovery enable transaction-by-transaction "undo" – the first innovation towards instant recovery is page-by-page "redo," originally motivated by flash storage.

3.1 Single-page failures and repair

None of the traditional failure classes considered in transaction processing and database systems properly describe failures of individual pages on storage such as flash. Single-page failures are different from transaction failures, from media failures, and from system failures [3]. Among the three traditional failure classes, single-page failures are most similar to media failures. They differ from media failures, however, since only individual pages fail, not an entire device. Treating a few failed pages as a failure of the entire device seems very wasteful.

Single-page recovery uses a page image from a backup and the history of the page from the recovery log, specifically the "redo" portions of log records for the specific page. Efficient access to all relevant log records requires a pointer to the most recent log record and, within each log record, a pointer to the prior one.

The original proposal for single-page failures suggests a "page recovery index" for each database or each table space. With an index entry for each page in the database, the page recovery index points to the most recent log record for each database page not present in the buffer pool. Each time the buffer pool writes a dirty database page to storage, an entry in the page recovery index requires an update with a new LSN value.

For the per-page list of log records, each log record merely embeds the PageLSN value found before the log record's update. In other words, this value is easy to obtain during transaction processing and it can serve other purposes than single-page recovery. These include consistency checking during log-based replication and during log replay while restoring a failed storage device. The recovery logs of Oracle and Microsoft databases contain per-page chains of log records, but not for the purpose of single-page recovery.

Lookup in the page recovery index, in a backup file, and in the log record might require multiple I/O operations. While seemingly expensive, it is cheap and efficient when compared to data loss or device mirroring with doubled costs from initial hardware acquisition to each write operation.

3.2 Self-repairing indexes

Self-repairing indexes [5] combine efficient (yet comprehensive) detection of single-page failures with immediate single-page recovery. Comprehensive fault detection requires in-page checks as well as cross-page checks. In a self-repairing b-tree index, each node includes low and high fence keys that define the node's permissible key range. Along the left and right edges of the b-tree, these fence keys have values $-\infty$ and $+\infty$, including in the root node. Otherwise, a node's fence keys equal two keys in the node's parent, i.e., typically branch keys. A node and its left-most child share the same low fence key value; a node and its right-most child share the high fence key value.

For both fault detection and repair, each parent-to-child pointer in a self-repairing b-tree carries an expected PageLSN value for the child page. For simplicity of maintenance, this requires that there be at all times only a single pointer to each page as in Foster b-trees [4].

3.3 Instant restart

Database system failures and the subsequent recovery disrupt many transactions and entire applications, usually for an extended duration. For those failures, new on-demand "instant" recovery techniques reduce application downtime from minutes or hours to seconds. These new recovery techniques work for all data stores that employ write-ahead logging, including databases, file systems, and key-value stores.

In traditional recovery from a system failure, e.g., a crash of the database server process, applications may resume and start new transactions after recovery has performed the "redo" actions of all pre-crash log records and then all "undo" (compensation) actions for failed transactions, i.e., those left incomplete at the time of the crash. Both "redo" and "undo" phases may require many random database reads and thus a relatively long time. The design and some implementations of ARIES support new transactions concurrent to the "undo" phase after lock acquisition during the "redo" phase.

For even better availability, instant restart permits new database transactions immediately after log analysis and thus before "redo" and "undo" work, imposes little load concurrent to new transactions, and prioritizes recovery of database contents to unblock new transactions. Lock re-acquisition for loser transactions must precede new transactions. Techniques new for log analysis include a backward log scan, transaction-by-transaction lock re-acquisition after log analysis, and database checkpoints immediately after log analysis and lock re-acquisition. After log analysis, new transactions guide on-demand recovery. An access to an "in doubt" database page triggers single-page repair and a lock conflict with a loser transaction triggers single-transaction rollback. Traditional "redo" and "undo" may run as background process. Techniques new for restart include page-by-page "redo" and transaction-by-transaction "undo," both single-threaded or in many parallel threads.

4. BACKUP AND RESTORE

An informal poll of database administrators identified backup operations as one of the most onerous aspects of owning a database. Thus, we set out to reduce or remove the need for taking database backups. The new techniques not only eliminate the need to take backups but also speed up restore operations, even permitting query and transaction processing against a replacement device practically instantly.

4.1 Single-pass restore

In this design [9], the log archive primarily serves log replay after media failures. Single-page recovery may use both the recovery log (for recent log records) and the log archive (for older log records). The discussion within this section focuses on the log archive and recovery from media failures.

The essence of restoring a database volume to an up-to-date state in a single pass is the order of database pages and of log records. For backup and replacement media, the ideal order proceeds by page identifier, from the lowest to the highest, i.e., a single sequential pass over backup and replacement media. The problem is that transactions write the recovery log in the order of time or of LSN values, not in the order of database pages.

A new technique enables single-pass continuous log archiving as well as immediate single-pass up-to-date restore operations: to sort log records during archiving, yet to divide the logic of external merge sort between log archiving and restore operation. The obvious way to divide this logic is to separate run generation and merging. In other words, log archiving not only suppresses some log records and compresses the remaining ones but also sorts log records into runs. Each run in the log archive captures a time interval in the original recovery log, i.e., a continuous range of LSN values. Within each run, log records are sorted by the page identifier within the database.

When compared to a fully sorted log archive, the crucial advantage of a partially sorted log archive is the efficiency in its creation. Creation of a partially sorted log archive is akin to run generation in an external merge sort. Thus, log archiving writes each log record to storage only once, the final log archive. Of course, due to in-memory sorting and run generation, archiving writes log records not immediately but after a short delay for in-memory processing.

When compared to a traditional, unsorted log archive, the crucial advantage of a partially sorted log archive is the efficiency in its use, i.e., during a restore operation. Replaying the log records in an unsorted log archive requires many random accesses in the replacement database. In contrast, a single merge step can merge many runs from a partially sorted log archive. The merge logic may pipeline the log records into the restore logic without intermediate files. Thus, the merge logic applies the sorted log records directly to the stream of database pages flowing from the backup to the replacement device, not to the replacement device in a second step after an initial restore operation.

4.2 Virtual backups

A virtual backup operation produces an up-to-date backup without even accessing the database. In other words, it is not a backup at all in the sense that it does not read any information from the active database.

A virtual backup does not require any new techniques: take an old backup and replay the recovery log in order to produce not an up-to-date database image but an up-to-date backup image. In the past, however, this process would require completely sorting the recovery log covering days or weeks or it would require decompressing the old backup onto standard database storage followed by log replay with hours or days of random I/O operations.

With an older backup and a partially sorted log archive, a virtual backup can be simple and fast. The operation merely merges the older backup and the partitions of the log archive, very similarly to a single-pass restore operation. The difference, if any, is the precise format of the produced backup. For example, while a restore operation writes page images on page boundaries in the target space, a backup operation may compress page contents, suppress empty space within pages (often >25% in b-tree pages), etc.

4.3 Instant restore

Obviously, it is impossible to finish a restore operation much faster than single-pass restore does. Therefore, "instant restore" merely gives the illusion of a truly instant restore operation: it permits queries and updates practically immediately after a replacement device is available, i.e., formatted but empty. Nonetheless, in spite of concurrent transactions, the restore operation may complete in about the same time as an offline single-pass restore operation, i.e., much faster than a traditional restore operation with log replay using log records in their original order.

The principal technique enabling the appearance of instant restore is on-demand restore operations for the immediately needed pieces of the failed media. In other words, the failed media and its replacement are conceptually divided into segments (a page or a set of contiguous pages), and each time a transaction attempts to access one of those segments for the first time after the failure, that particular segment is recovered from the most recent backup and from the log archive.

In order to enable efficient on-demand single-segment media recovery, both the backups and the log archive require appropriate indexes. A full backup does not require an index if a database page identifier implies a byte offset in the backup file, i.e., if the backup logic fails to skip over unallocated pages, to eliminate free space within pages, and to forgo other opportunities for compression. In the indexes for the backups, each index entry maps a segment, e.g., a database page identifier, to a location in the backup file. A large device might be broken into 1 M segments and therefore the index requires up to 1 M entries per partition or run.

With those indexes, on-demand recovery restores one segment at a time, i.e., a pre-defined set of contiguous database pages, with the logic of single-pass restore within each segment. The indexes enable efficient access to the relevant database pages in the backup and log records in the partitions of the log archive.

Moreover, since data structures and algorithms are so similar, segment-at-a-time restore operations can run both lazily, i.e., on demand and guided by active transactions, and eagerly, i.e., sweeping through all segments in the manner of single-pass restore. Thus, a restore operation should complete in about the same time as with offline single-pass restore, i.e., much faster than traditional restore operations with log replay and many random I/O operations, and on-demand restore guided by active transactions should not extend media recovery time.

5. CLUSTERS AND FAILOVER

Instant failover assumes a cluster with multiple nodes and log shipping from a primary node to one or more secondary nodes. Using a buffer pool, the database, and a recovery log, a primary node executes queries and updates while each secondary node holds a backup and the log archive. Their principal communication is continuous log shipping from the primary to each secondary node. The principal design question is how quickly a secondary node can take over query and transaction processing after the primary node fails.

In the traditional approach to high availability and fast failover, the secondary node holds a complete copy of the database and always keeps it up-to-date by immediate "redo" of all log records received via log shipping. In the alternative design, instant failover, a secondary node does not require an up-to-date copy of the database. It merely requires empty space for the database, a full database backup (days, weeks, or months old), and a log archive covering the time since the full database backup. Both database backup and log archive must support fast access by page identifier, typically using indexes similar to those required for instant restore.

5.1 Instant failover

Failover requires recovery of server state and database contents. The server state includes transaction manager, lock manager, and buffer pool. Recovery of server state resembles system restart. Recovery of database contents resembles media restore operations. Thus, instant failover must combine log information and recovery actions from instant restart and instant restore.

The relevant server state for restart and failover includes the set of active transactions (except read-only transactions) and the set of locks held by active transactions (except read-only locks). Instant failover does not need in-doubt pages modified by recent and active transactions because all database pages require recovery. Instant failover recovers server state using techniques adapted from instant restart.

In instant restart, log analysis recovers the required server state such that transaction processing can resume immediately with concurrent "redo" recovery actions (in the form of single-page repair) and "undo" actions (in the form of single-transaction rollback). For instant failover, the new primary (formerly secondary) site may perform log analysis either continuously while receiving log records, i.e., before the failover, or as part of taking over primary responsibility for the database, i.e., during the failover. Mixed models are possible. For example, a secondary site may track the set of active transactions continuously during log shipping yet acquire locks only after a failover. With such a design, a secondary site avoids expensive lock management yet can acquire locks for active transactions quickly based on log records of active transactions, which are readily available using the per-transaction log chains.

Transaction processing after a failover assumes that the database on the new primary site is up-to-date and transaction-consistent. Instant failover starts with a full database backup and a log archive, with some of the log records still being sorted in order to form a partially sorted log archive indexed by a partitioned b-tree. Those log records may still linger in memory, i.e., the failover site has received but not added them yet to the persistent log archive yet. Instant failover recovers database contents using techniques adapted from instant restore.

The first task towards database recovery during instant failover sorts and indexes those log records. They may remain in memory and not immediately be added to the persistent log archive. The important aspect is that the restore logic can readily access all log records received from the failed former primary site. If additional sites require further log shipping, the new primary site sets up appropriate connections and log-shipping streams.

The second task provisions media for future database storage, recovery log, and log archiving.

The third task initiates single-pass restore in the background. It will run until all restore operations are complete. It will coordinate with on-demand incremental restore operations by avoiding concurrent recovery of the same database segment, by marking database segments it recovered, and by skipping over database segments already recovered by on-demand incremental restore operations.

The fourth task of instant failover resumes transaction processing. When a transaction invokes the buffer pool for a database page not yet recovered, the logic of instant restore recovers the appropriate segment on demand, exploiting appropriate indexes on the backup and on the partitions of the log archive. When a transaction requests a lock conflicting with one of the transactions active prior to the failover, that transaction must roll back. Transaction rollback may invoke restore operations for one or multiple segments by invoking the buffer pool for database pages not yet recovered.

The final task of instant failover rolls back pre-failover transactions not yet aborted on demand upon lock conflicts with new transactions. If desired, this task may start earlier, in which case it may well trigger restore operations for individual database segments. On the other hand, earlier execution of this task focuses early recovery efforts on the application's working set within the database. Of course, pre-failover transactions write rollback log records as during rollback without failover.

In summary, instant failover requires log shipping before the failure and techniques from instant restart and from instant restore after the failover. As in traditional log shipping, a transaction is durable even in the event of a complete node failure only after the secondary site has received all its log records including the commit log record. As in instant restart, log analysis recovers server state, in particular the transaction manager's set of active transactions and these transactions' non-read-only locks. As in instant restore, a partially sorted and indexed log archive permits log replay by merging backup and partitions of the log archive.

Log shipping for instant failover using these techniques requires much fewer resources on the secondary site than traditional log shipping. Instant failover merely requires a database backup, whereas traditional log shipping and failover requires an active database; and instant failover merely merges backup and log archive partitions in quasi-sequential I/O operations, whereas log replay in traditional log shipping and failover requires either many random I/O operations or an extremely large, dedicated buffer pool. Nonetheless, failover latency is quite similar in both techniques, gated by the delay in log shipping, and transaction processing performance after the failover suffers only slightly immediately after instant restart.

5.2 Failover pools

For the fastest possible failover, a secondary site may pre-start a database server process with no information yet in transaction manager, lock manager, and buffer pool. Such a server can take over for any database. A pre-started database server might serve other databases before and after it takes over for a failed database. Thus, instant failover may add information into the server state of an active server, which might require evicting some information from within this server, in particular database pages from the buffer pool.

In general, there might be multiple secondary sites. In case of a failure of the primary site, one of the secondary sites must take over. We ignore here the question on how to choose among multiple secondary sites and focus on techniques for instant failover on the chosen secondary site.

Conversely, a single site may partition its data such that failover spreads responsibility widely to multiple secondary sites. In this case, log shipping must split the log records to the correct secondary sites. The remainder discusses failing over an entire database but it might be only a partition within a database.

Finally, a site may serve as a secondary site for multiple databases or even multiple primary sites. This assumes, of course, that this site can access backups of all pertinent databases and receives a log-shipping stream from all relevant databases and sites. If site failures are rare and double failures are exceedingly rare, such a design promises to be viable.

Two possible deployment options suggest themselves. First, a large set of sites may serve as secondary sites for one another, in such a way that each site has multiple partitions and any failure spreads the load across multiple sites. Thus, even with a small number of failures, service continues with reasonable performance. Each partition requires multiple primary sites such that load balancing after is possible in cases of multiple site failures. This design also permits load balancing in the absence of failures, e.g., while one database or partition experiences high temporary workloads.

Second, a large set of primary sites serve the database workload with only a few secondary sites. All secondary sites have access to all backups and all log archives, e.g., in shared network-attached storage. If a primary site fails, any secondary site can take over and assume the role of the specific failed site. With this deployment design, a few extra servers may achieve high availability rather than 3× the number of servers as required in traditional designs for high availability.

6. CONCLUSIONS

In summary, two simple techniques – per-page chains of log records in the recovery log and partially sorting log records in the log archive – enable page-by-page "redo" and thus a wealth of new features and functions in database backup and recovery. Combining recovery of server state from instant restart with recovery of database contents from instant restore enables instant failover to a database node that holds a database backup and a log archive but no up-to-date database image.

A third simple technique – error detection by parity or cyclic redundancy check calculations in data pages and check values in inodes and indirection pages – transfers the capabilities of self-repairing indexes to file systems and their data pages, despite their lack of page headers as found in database pages. Thus, if a file system can be extended from "double-write" journaling to write-ahead logging with a durable log archive, all forms of instant recovery including instant failover and failover pools can apply to practically all storage. This can reduce redundancy in data centers from typically 3N to N+3, i.e., by almost a factor of 3.

REFERENCES

[1] Jim Gray: Notes on data base operating systems. Advanced course on operating systems. Springer LNCS #60, 1978: 393-481.

[2] Goetz Graefe, Wey Guy, Caetano Sauer: Instant recovery with write-ahead logging: page repair, system restart, media restore. Morgan & Claypool Synthesis Lectures on Data Management, 2014.

[3] Goetz Graefe, Harumi A. Kuno: Definition, detection, and recovery of single-page failures, a fourth class of database failures. PVLDB 5(7): 646-655 (2012).

[4] Goetz Graefe, Hideaki Kimura, Harumi A. Kuno: Foster b-trees. ACM TODS 37(3): 17 (2012).

[5] Goetz Graefe, Harumi A. Kuno, Bernhard Seeger: Self-diagnosing and self-healing indexes. DBTest 2012: 8.

[6] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS 17(1): 94-162 (1992).

[7] C. Mohan, Frank E. Levine: ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. ACM SIGMOD 1992: 371-380.

[8] C. Mohan: ARIES/KVL: a key-value locking method for concurrency control of multi-action transactions operating on b-tree indexes. VLDB 1990: 392-405.

[9] Caetano Sauer, Goetz Graefe, Theo Härder: Single-pass restore after a media failure. BTW 2015: 217-236.