

Databases II B: DBMS-Implementation

Chapter 6: Logging and Crash Recovery

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Winter 2025/26

<http://www.informatik.uni-halle.de/~brass/dbi25/>

Objectives

After completing this chapter, you should be able to:

- enumerate and explain possible scenarios which require recovery,
- explain how the redo log and the undo tablespace work in Oracle,
- explain the temporal relation between the writing of database blocks and the redo log information,

This include the delayed writing of database blocks, the WAL (“write ahead log”) principle, and checkpoints.

Contents

- 1 Introduction
- 2 Buffer Management
- 3 The Log File
- 4 Crash Recovery
- 5 ARIES

Types of Failures (1)

- A transaction fails (before it is committed).

E.g. the program detects some error condition and calls “ROLLBACK”. Or the process is the victim selected in deadlock resolution. Or the user switches the client computer off without properly finishing the application program.

- The main memory is lost.

E.g., because of a power loss or a failure of the operating system.

- The contents of a disk is lost (media failure).

E.g., because of a head crash in the disk (the head, which normally flies on an air cushion over the surface, touches the surface and cuts some of the magnetic coating from it). If only the power supply (or electronics) is faulty, this can be repaired, but it will take some time.

Types of Failures (2)

- For the above failures, the transaction guarantees (atomicity and durability) require that the DBMS can handle them.
- In this chapter, we only consider
 - Rollback of a single transaction.
 - Crash recovery, i.e. restart of the system after the contents of main memory was lost (e.g., in case of a power failure).

In this case, all unfinished transactions must be rolled back.

The changes of all transactions that were committed must be in the database files (after recovery).

- A later chapter will treat media failures.

Types of Failures (3)

- There are also user failures:
 - A user executes some wrong SQL statement.
And has Autocommit on.
 - An administrator drops the wrong table.
 - An application program contains a bug and inserts wrong data (and COMMITs the changes).
- For user errors, Oracle has the recycle bin to undo an accidental “DROP TABLE” and flashback queries to query an older database state (before unwanted updates).

Depending on the system configuration, this might not lead very far into the past. The default is 900 s, i.e. 15 min.

Crash Recovery (1)

- Crash recovery is done when the system starts and detects that the system was not shut down normally.

E.g. the DBMS writes a status byte indicating “open” to the database files when it starts. In a clean shutdown it writes “closed” to the file. If the files still contain the “open” mark when the system starts, this indicates that there was no clean shutdown.

- I.e. it must assume that
 - a power failure occurred or
 - the DBA did an “immediate” shutdown, i.e. simply killed the server processes of the DBMS, or
 - the operating system did not inform the DBMS that it will shut down or did not give sufficient time so that the DBMS can save all data.

Crash Recovery (2)

- Data in main memory (lost when the system crashed):
 - The database blocks in the database buffer cache, in particular also “dirty blocks”, i.e. versions of blocks that were newer than the version on disk.
 - Table of currently running transactions.
- Note also that the disk might contain versions of blocks that contain changes of uncommitted transactions (i.e. changes that must be undone).
- The blocks in the data files are from different times (whenever they were saved to disk). I.e. there is not a consistent view of the data.
 - The data files are useless without crash recovery.

Undo and Redo Information

- For every change to the database (**INSERT**, **UPDATE**, **DELETE**), the system can manage UNDO and REDO information.
- UNDO information (i.e. the inverse operation) is needed when a transaction must be rolled back.

This might also be needed as part of the crash recovery if uncommitted changes were already stored on disk.

- REDO information is needed when the change was done in the copy of a database block in the main memory buffer, but did not make it back to persistent storage (disk) before the crash.

Contents

- 1 Introduction
- 2 Buffer Management**
- 3 The Log File
- 4 Crash Recovery
- 5 ARIES

Writing Changed Blocks (1)

- Changes are first done in the buffered copies of database blocks.
- There are several possibilities for restricting the time when changed blocks are written back to disk.
 - “steal” policy: Blocks may be written to disk although they contain uncommitted changes.

I.e. the buffer manager can “steal” the buffer from a transaction whenever the buffer is unpinned, although the transaction is still running (and might later need to be rolled back). “Dirty data”.
 - “no-steal”: Only committed changes may be written to disk (transactions keep all modified buffers until the COMMIT).

The database files on disk contain no traces of unfinished transactions.

Writing Changed Blocks (2)

- With the “steal” policy, the atomicity of transactions (“all or nothing”) requires that
 - undo information is needed (in a log file),
 - and must be written to disk before the changed block (WAL principle: “write ahead log”).

In case of a power failure, the changes by transactions which did not run to the end must be undone.

- With “no-steal”, the available memory for buffering blocks might be sub-optimally used.

If a long-running transaction changes many blocks all these blocks must be kept in memory and the buffers cannot be used otherwise.

Writing Changed Blocks (3)

- If one wants to avoid undo information with a “no-steal” policy, this also has the consequence that transactions must lock entire blocks, not single rows.
 - If a transaction must be rolled back, the blocks that it changed are simply deleted from the buffer.
 - However, then blocks with changes of committed transactions must be saved to disk before a new transaction can be allowed to change the block.

In effect, the block is locked even after the transaction finished, or we must do synchronous writing of disk blocks at commit time.

- At least, it is not necessary to store undo information on disk with a “no-steal” policy.

For crash recovery, it is not needed. There are no uncommitted changes on disk.

Writing Changed Blocks (4)

- Note that a transaction can change more than one block.
 - Thus, writing the changed blocks takes some time (it is not an atomic operation).
 - However, the transaction is only finished when all its changes are durable.
 - One cannot directly overwrite changed blocks at the end or needs to make REDO information persistent (on disk) before one starts to overwrite changed blocks.

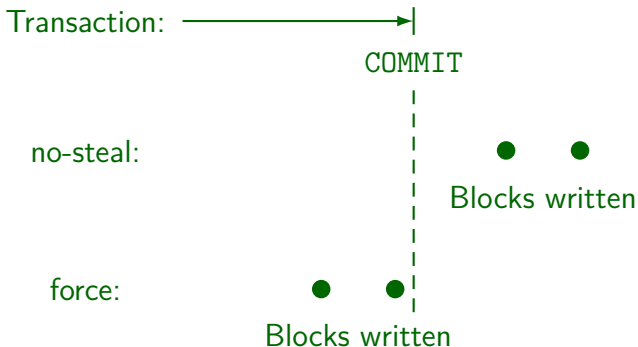
Otherwise, if the power failure happens during the commit, there would be changes of an unfinished transaction in the database files, contradicting the no-steal policy.
 - One needs a mechanism to make the change of several blocks an atomic operation.

Writing Changed Blocks (5)

- Another restriction on writing changed data blocks is:
 - “**force**” policy: Before a transaction can commit, all blocks it has changed must be written to disk.
 - “**no-force**”: Blocks with committed changes do not have to be saved immediately.
- With the “force” policy, one can avoid REDO information.
- With the “no-force” policy, the durability of transactions requires that REDO information
 - redo information is needed (in a log file), and
 - it must be on disk before the COMMIT.

Writing Changed Blocks (6)

- Restrictions on the time when changed blocks can be written to disk:



Writing Changed Blocks (7)

- Another basic design decision for the buffer manager related to recovery is whether it uses
 - “**update in place**”: Changed blocks are written back to the same disk position (thereby overwriting the old version).
 - “**no update in place**”: Old blocks are not immediately overwritten (with various methods, e.g. “shadow page concept”, “twin block concept”, “differential file”).

In this case one must distinguish between the logical page and the physical block on disk. The mapping from pages to blocks is now indirect and needs some additional data structures.

- At least the combination “no-steal, force, update in place” is impossible: A transaction might change several blocks, but atomicity requires that the writing becomes effective at a single point in time.

Writing Changed Blocks (8)

- Advantages of not using “update in place”:
 - Even for the “steal” policy, no explicit UNDO information might be needed.

Actually, there is UNDO information, namely the old block versions which were not overwritten.

- The recovery is usually very fast.
- There are no problems with partially written blocks.

If one has neither the old version of the block completely, nor the new version, the UNDO information might be not sufficient to get the old version back.

Writing Changed Blocks (9)

- Disadvantages:

- Fast sequential reading of blocks on disk becomes less valuable.

However, with SSDs, this is no longer an issue.

- More disk space is needed (two versions of blocks).
- Memory is needed for buffering mapping information from pages to blocks (or more time for reading blocks).

- Most modern systems (e.g. Oracle) use

- steal
- no-force
- update-in-place

The writing of changed disk blocks is decoupled from transaction ends.

Contents

- 1 Introduction
- 2 Buffer Management
- 3 The Log File**
- 4 Crash Recovery
- 5 ARIES

The Log File (1)

- An essential tool for recovery is the log file.
- It contains a record of all changes to the database.

Including not yet committed changes.

- A log entry normally contains
 - REDO information: This is needed in case the modified block in the database buffer cache was not written to disk before the main memory loss (then the system must redo the change when it later starts again).

REDO information permits to “replay” the transaction.

- UNDO information: This is needed to undo a change in case the transaction must be rolled back.

The Log File (2)

- In principle, the REDO and UNDO information can be
 - logical (**INSERT**, **UPDATE**, **DELETE** with the corresponding inverse operation) or
 - physical (changed bytes).
- In the recovery method explained here, each log entry refers to a single database block.

E.g., an **INSERT** might change several blocks (including index blocks), but then there will be several log entries for this operation.
- REDO is simple, because it will later be applied to exactly the same block version.

E.g., one can log calls of methods to change the block (including the parameter values).

The Log File (3)

- UNDO is more tricky, because it might be applied to a later block version.
 - I.e. the block might contain changes that were not there when the original update was done that must be undone.

Transactions that changed the block later might have finished earlier.
 - This is related to the locking: Only updates of unfinished transactions are undone, i.e. they still were holding the locks on the objects they changed. Thus no later transaction can have changed these objects.

If entire blocks are locked, the UNDO can rely on the fact that there are no later changes from other transactions in the block. But locking entire blocks is not state of the art. Currently, it is common to lock single table rows. Then the UNDO is applied to the right version of the row (after the update that has to be undone), but other rows in the same block might have changed.

The Log File (4)

- Each log entry has a unique sequential number, the log sequence number (LSN). It is a version number for the current state of the database (like a timestamp).

Of course, the log sequence number is always monotonically increasing.
In Oracle, the LSN is called SCN (“system change number”).

- Each database block has a field for the LSN of the last change that was applied to it.

With this, one can later check whether the version on disk contains a certain change (identified by an LSN) or not. In this way, REDO operations become idempotent: If the LSN of the block is larger than the LSN of the change, the REDO is simply ignored.

- Each entry contains also the number of the transaction to which this change belongs.

The Log File (5)

- Rollback can become more efficient if each log entry also contains the LSN of the previous change of the same transaction (“prevLSN”).

It is Null for the first change of a transaction.

- The log file contains also records for the COMMIT operation.

If this entry is found in the logfile when the systems starts after a power failure, the transaction was committed (changes of this transaction will be redone if necessary). If the entry is not found in the logfile, the transaction was incomplete and will be rolled back (all changes undone).

- Furthermore, the end of the undo phase after a ROLLBACK is marked in the log.

This is the point when one can forget about the transaction: All its changes have been undone (at least in the version of the main memory buffers).

Details of Writing Changed Blocks (1)

- One must always consider also transactions running in parallel.
- While a block is written back to disk, it must not be changed:
 - It is either the version before the change described by a log entry (so the REDO can be applied later),
 - or the version after the change (and the UNDO can be applied if necessary).
- While the logging helps to guarantee atomicity on a higher level, the algorithms need atomicity on a lower level.

Details of Writing Changed Blocks (2)

- While a block is written, it must be locked against changes by parallel transactions (with a shared lock).
- Transactions that wish to change a block must lock it with an exclusive lock.
- The block can be unlocked immediately after the operation.
- In order to distinguish these short lived locks from the locks on changed tuples that are kept until the end of the transaction, these short lived locks on blocks are called “latches”.

The Log Buffer (1)

- Log entries are first written to a log buffer in main memory.
- Only when a COMMIT is processed, all entries up to and including the COMMIT Record must be written to disk (“flushed”).

This must be done before the COMMIT is acknowledged. I.e. when the user gets the information that his/her COMMIT was successful, all log entries of that transaction are already safely stored to disk (made persistent). Since log entries are always written sequentially, all other log entries with a smaller LSN (even from other transactions) are also stored on disk.

- The COMMIT requires synchronuous writing to the log file.

The changed database blocks can be written later: Since the log contains the REDO information for all changes of that transaction, no update of that transaction is lost, even if the power failure happens a moment after the COMMIT.

The Log Buffer (2)

- The contents of the log buffer can be written to the log file at any time.
- Normally, one would wait until a block is full.

One can write only complete blocks to the log file. However, the block size could be smaller than the block size of the database files (but it must be at least the sector size). It is very important that by writing a bunch of log entries, previously written log entries cannot be damaged.

- In this way, one can ensure that the log buffer never fills up completely.

However, the UNDO information is needed for ROLLBACK operations even without system crash. It would be good if this information were still in main memory when the ROLLBACK happens. In Oracle, this is stored in the undo tablespace (separately from the log).

The Log Buffer (3)

- DB2 has the option to delay the **COMMIT** a little bit (milliseconds) in the hope that other transactions commit during this waiting time, and only one block has to be written to the log containing **COMMIT** records for all of them.
- In case of a system crash a suffix of the created log records is lost (records in the log buffer that were not yet written to the log file).
 - This suffix does not contain **COMMIT** records.
 - This suffix also does not contain UNDO information for database blocks that were already written to disk (WAL principle: “Write Ahead Log”).

We must be able to UNDO changes of unfinished transactions.

Therefore, we cannot write changed blocks back to disk before the log entries of these changes are safely stored in the log file.

The Log Buffer (4)

- It is much more efficient to write the log buffer at COMMIT time than to write all changed database blocks:

- The log is a concise representation of changes distributed over many database blocks.

A single short **INSERT** changes at least one block in the table data, but also at least one block in each index defined on that table. One block of log information can contain log records describing many changes of many transactions.

- The log is written sequentially, i.e. the disk head is always at the right position if the disk contains no other files.

In contrast, database blocks are distributed over the entire disk.

- A “hot” database block might be changed many times by many transactions before it is finally written back to disk.

In this case, being lazy (delaying work) actually saves work.

Transaction Table

- The system has a table of the currently active transactions.

A transaction is active until the COMMIT or until the UNDO of all changes after a ROLLBACK is finished. In the first case, the log contains the COMMIT record, and the transaction is deleted from the table immediately after the COMMIT-record is safely stored in the log file on disk. In the second case, the log file contains an “end of transaction” record (which also permits to delete the transaction from the transaction table).

- This table contains
 - a transaction number (unique, monotonically increasing),
 - the LSN of the first change of the transaction (“firstLSN”),
 - the LSN of the last change of the transaction (“lastLSN”).
- The table is in main memory, i.e. it is lost in case of a crash (but can be reconstructed, see below).

Compensation Log Records (1)

- During normal processing (even without crash), a transaction might end with a **ROLLBACK** command.
- The transaction table contains the LSN of the last log record of that transaction (“lastLSN”).
 - This change is undone first,
 - then the log record links to the previous log record of that transaction (“prevLSN”), which is undone next, and so on.

As in a stack, the last change of the transaction is undone first, because one starts in the last state and works back to the beginning.

- Each such UNDO operation needs a log sequence number: The LSN in the data blocks should increase for each change.

Even if that change is an UNDO of a previous change. This permits to easily check whether the UNOD was done in a block version.

Compensation Log Records (2)

- The log record written for an UNDO operation is called a “compensation log record”.
 - It differs from normal log records that it contains only a REDO part (which was the UNDO part of the original change).

When a ROLLBACK is started, the decision is final: One cannot undo a ROLLBACK.

- The “LastLSN” field in the transaction table is made pointing to the compensation record, and the “prevLSN” field in the compensation record is set of the value of “prevLSN” in the log record that was undone.

Thus, if a crash should happen, one can find the log records that still have to be undone.

Checkpoints (1)

- Normally, the log file is written sequentially.
- After some time, the disk will be full.
- Therefore, the system does “checkpoints” from time to time:
 - When the checkpoint starts, the current LSN n is recorded.
 - Then all changed database blocks in the database buffer cache are written to disk.

At least all blocks that contain unsaved changes up to LSN n .
 - This takes some time, but when it is finished, it is guaranteed that all changes before LSN n are on disk.

As long as there is still enough disk space for the log, the checkpoint process can have lower priority and work only when the system is otherwise idle.

Checkpoints (2)

- When a checkpoint started at LSN n is complete, the REDO information of all log entries before LSN n is no longer needed (for crash recovery).

Media recovery does need it until we also have a backup of the database files that includes all changes up to LSN n .

- The UNDO information of a transaction is no longer needed after the transaction is committed (or ROLLBACK processing is complete).

However, when we use multiversion concurrency control, we must be able to restore old versions of database objects that were current when a query started. In that case, UNDO information of transaction t becomes obsolete only after t is committed and all queries that started before the COMMIT of t are finished. The alternative is to terminate very long running queries with an error message (“snapshot too old”) when the necessary UNDO information is no longer available.

Checkpoints (3)

- Note that UNDO information of transactions active at the time when the checkpoint started (LSN n) is still needed:
 - The checkpoint guarantees that all changes before LSN n are in the database files on disk.
 - But a transaction active at the time of LSN n might rollback later.
 - This needs the UNDO part of log records before LSN n .
- When a transaction is committed, we only need the REDO part of the log records after LSN n .
- When the **ROLLBACK** processing of a transaction is complete, the compensation records contain all information to redo the UNDO.

Starting from the database files at the version of LSN n or later.

Checkpoints (4)

- One solution is to use not LSN n as the cutoff point for old log records that can be deleted/overwritten, but to compute the minimum LSN of all changes of the transactions active at this point, and use that as cutoff point.

A transaction that did a “**ROLLBACK**” is considered as “still running” until the UNDO of all its changes is complete (and the “end record” was made persistent in the log file). In the transaction table, one can have a field for the LSN of the first change of that transaction. One computes the minimum of that field. Note that even computing the minimum takes a little bit of time. During this time, transaction processing continues and rollbacks are done and transactions are removed from the transaction table (and thus are not counted for the minimum). Suppose that LSN $n' \geq n$ is current when the computation of the minimum is finished. Only at this point, one really starts to save dirty pages, so that at the end of the checkpoint, all changes up to LSN n' are guaranteed to be in the database files.

Checkpoints (5)

- An alternative is as follows:
 - At the start of the checkpoint (with the cut-off LSN n), one first waits until all transactions are finished that were active at LSN n .
 - Only after that, one starts saving all dirty pages.
- In both cases, one can guarantee that during crash recovery, one never needs a log record before the computed cut-off point.
- When this change information is no longer needed, it can be deleted or overwritten (to reuse the disk space).

Checkpoints (6)

- For media recovery (after a disk failure), the log information is still needed to get from an old backup to the current database state.

If the backup was done while the database was running (“hot backup”), also the undo information in the log is needed. In this case, the backup contains an inconsistent database state (the different blocks in the backup are from different times and contain changes until different LSNs). Without log information, such a backup is unusable.

- However, the old log information (no longer needed for crash recovery) can be copied to tape or a slower disk.

A media failure is seldom and recovery from it will anyway take some time.

Dirty Page Table

- For setting a checkpoint, one must find all “dirty” blocks in the buffer cache (i.e. blocks that must be saved to disk during the checkpoint).

Also during normal operation, there should be sufficiently many buffers with non-dirty blocks (so that they can easily be reused). Thus, writing dirty blocks back to disk is not only done during a checkpoint. But during normal processing, blocks are saved that are near the front of the LRU queue, so we can anticipate that their buffer will be needed soon. During a checkpoint, all blocks containing (older) changes are saved.

- One can use a special table of dirty blocks in the buffer.
- It is useful if it contains the first LSN of a change that made the block dirty.

If this change is after the LSN n at the start of a checkpoint, the block does not have to be saved during the checkpoint.

Separation of Redo and Undo Data (1)

- Because the time that REDO and UNDO information is needed is different, some systems store them separately.
- Oracle has
 - Redo log files
 - Undo tablespaces (or Rollback segments)
- However, changes to undo tablespaces are also logged in the redo log files!

[<https://asktom.oracle.com/pls/apex/f?p=100:11:::::>

P11_QUESTION_ID:9545686800346340839]

When the system crashes, the undo information is essential to undo unfinished transactions. If one wants to separate them completely, the undo information must be just as safe as the redo information (but usually for a shorter time).

Separation of Redo and Undo Data (2)

- However, Undo information is needed during normal processing (for **ROLLBACK**), whereas Redo information is needed only in case of a system crash.
- The writing of the log file must be very fast, because a transaction cannot commit before its log records are really stored on disk.

The writing of changed data blocks can be done asynchronously (“lazy”, “delayed”). However, the log must be written synchronously: The user must wait for the acknowledgement of the COMMIT until the writing of the log entries is finished.

- Therefore, it is good that the log is normally “write only”.

Oracle does not have to read log information during normal processing. Therefore, the disk head can remain on the track where the end of the log file is (if the log is the only file on that disk).

Contents

- 1 Introduction
- 2 Buffer Management
- 3 The Log File
- 4 Crash Recovery**
- 5 ARIES

Crash Recovery (1)

- During crash recovery, the system reads the log file from the cut-off point of the last checkpoint (or from the beginning if there was no checkpoint yet).

The information where to start in the log file can be written to some other short file (whenever a checkpoint is finished). In Oracle, it is stored in the control file(s).

- The recovery works in two phases:
 - REDO Phase
 - UNDO Phase
- During the REDO phase, all changes in the log file are redone (this includes changes of transactions that later do a **ROLLBACK** or were still active at the time of the crash).

Crash Recovery (2)

- Since each block contains the LSN of the last change that was applied to it, one can easily check whether the redo of a log entry really has to be done.

This ensures that no change is done twice.

- During the REDO phase, the transaction table is reconstructed in main memory.

Whenever a new transaction number is found in a log record, the transaction is added to the transaction table. Whenever a COMMIT oder "end of transaction" record is found, the transaction is removed from the transaction table.

- Therefore, at the end of the REDO phase, the transaction table contains those transactions that were still active at the crash and must be undone.

Crash Recovery (3)

- In the same way, the dirty page table is reconstructed in main memory during the REDO phase.

When the REDO part of a log entry is reapplied to a block, the block is of course in the main memory buffer. It does not have to be written back to disk immediately. Maybe it will be changed again by further log entries. Even during crash recovery, the buffering works as usual.

- Once the REDO Phase is finished, all database blocks have the same contents as at the time when the last log entry was written to the log file.

This is shortly before the crash. Furthermore, all changes that still happen after this last log entry are from unfinished transaction and did not make it to the database files. Therefore, they are not needed.

Crash Recovery (4)

- In the UNDO phase, all transactions still in the transaction table are undone as if they had called **ROLLBACK** (see above).

They were unfinished at the time of the crash.

- As explained, compensation records are written for each change that is undone.

The original change was from A to B, and now we have a new change back from B to A.

- Even a crash happening during crash recovery is no problem.

E.g., when the first crash happened because of a power failure, the power supply might still be unstable when the system automatically restarts.

Contents

- 1 Introduction
- 2 Buffer Management
- 3 The Log File
- 4 Crash Recovery
- 5 ARIES

ARIES (1)

- ARIES is a well-known recovery method.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.:

ARIES: A Transaction Recovery Method Supporting Fine-Granularity

Locking and Partial Rollbacks Using Write-Ahead Logging.

ACM Transactions on Database Systems, Vol. 17, No. 1, March 1992, 94–162.

- It is quite similar to what was explained above, however, setting a checkpoint is different:
 - Only the current transaction table and the dirty page table are stored to the log.
 - The writing of dirty pages (blocks) is an independent process.
- Setting a checkpoint is much quicker, but the cut-off point for the log can be far earlier from the checkpoint.

ARIES (2)

- Recovery is done in three phases:
 - Analysis
 - REDO
 - UNDO
- The task of the Analysis Phase is to reconstruct the transaction table and the dirty page table.
- Whereas in the above algorithm, the dirty page table was only an option for finding dirty pages quicker in the buffer pool, it is essential for the ARIES algorithm.

Because it defines where in the log the REDO phase must start.

ARIES (3)

- The dirty page table contains for each modified block the first LSN that made it dirty (“Recovery LSN”/“recLSN”). The REDO phase must start at the earliest “recLSN” in the dirty page table (earlier updates are in the DB files).
- During the Analysis Phase, the dirty page table is rebuilt as follows:
 - The algorithm starts with the dirty page table that was saved as part of the last checkpoint.
 - From there, it follows the log: Each log entry contains the number of the changed block.
 - ARIES needs log entries also for blocks that were saved back to disk and therefore have to be removed from the dirty page table.

References

- Raghu Ramakrishnan, Johannes Gehrke: Database Management Systems, 2nd Ed. McGraw-Hill, 2000, ISBN 0-07-232206-3, 906 pages.
- H. Garcia-Molina, J. D. Ullman, J. Widom: Database System Implementation. Prentice Hall, ISBN 0130402648, 672 pages, ca. \$60.00
- Theo Härder, Erhard Rahm: Datenbanksysteme, Konzepte und Techniken der Implementierung (in German). Springer, 1999, ISBN 3-540-65040-7, 582 pages.
- Kemper/Eickler: Datenbanksysteme (in German), 7rd Edition, Oldenbourg, 2009. Kap. 10: Fehlerbehandlung ("Treatment of Faults").
- Oracle Database Online Documentation 11g Release 1 (11.1) Database Administration Oracle Corporation, 2016
[https://docs.oracle.com/cd/B28359_01/nav/portal_4.htm]
- Immanuel Trummer (Cornell University): 21: Database Recovery via ARIES.
[<https://www.youtube.com/watch?v=csWHS0SLTtM>]
- Immanuel Trummer (Cornell University): 22: Recovery Phase of ARIES
[<https://www.youtube.com/watch?v=iVBlnYEmHII>]