9 Recovery

Learning Goals:
- Understand and be able to explain how the fundamental architecture of database systems plus the need for performance leads to problematic situations in case of errors for which consistency strategies have been developed
- understand and be able to execute in sample scenarioes various recovery strategies
- recognize and be able to describe and solve problems in given recovery scenarioes
- understand the differences between the consequences of software and hardware failure and be able to select the appropriate algorithms

Goal of Recovery:
Preserve correctness and consistency of data over time, allowing for parallel access (transactions) of multiple users and occuring errors.

9.1 Problem Sources

Potential reasons for transaction failure:

Software error
- in an application program
- in the operating system
- in the database system

Cancellation by Concurrency Control

- Some methods for avoiding errors in concurrent execution of transactions cancel transactions when a problem can be solved this way (for instance deadlock or conflict with other transactions).

Hardware error
- disk error or damage (e.g. a head crash)
- CPU error
- bus error
- power failure or fluctuation

Operator error (e.g. mounted wrong tape, wrong file system)

Sabotage

Recovery is supposed to restore a state of the database that is known to be correct, after an occurred or presumed error.

General principle used for recovery: redundancy.

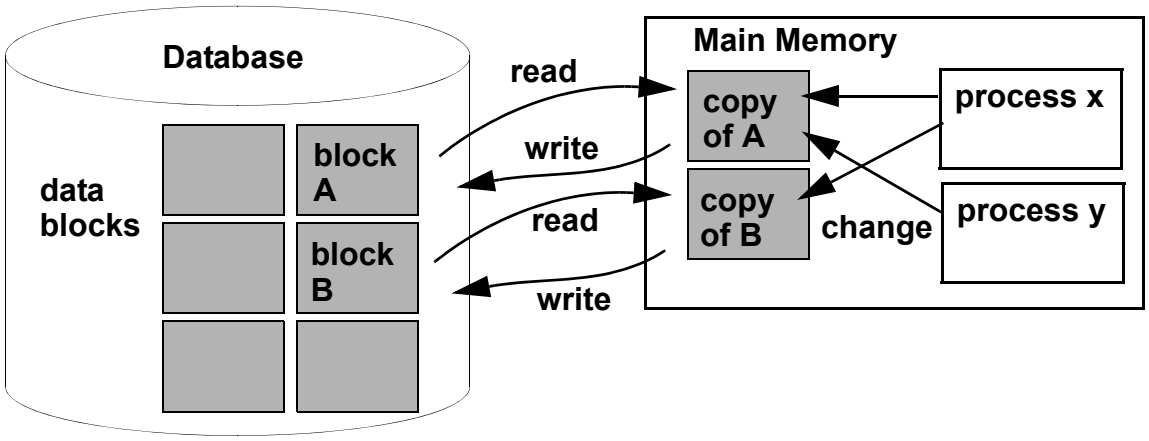
9.2 Logging and Recovery from Software Failures

9.2.1 Reading and Writing Data

Reading and writing data items from and to a database are not immediate, atomic operations, but actions consisting of several steps:

Reading a data item from the database requires the following steps:
1. Find the address of the disk block that contains the data item.
2. Copy this disk block into a buffer in memory (if it is not there already).
3. Copy the value of the data item into a variable in the application program.

Writing a data item to the database requires the following steps:
1. Find the address of the disk block that contains the old version of the data item or the address of the disk block where the data item is to be newly entered or deleted from.
2. Copy this disk block into a buffer in memory (if it is not there already).
3. Copy the value of the data item from an application program variable into the correct location in the block in the buffer.
4. Copy the updated block from the buffer back to disk. This may be done immediately or later (when the buffer is full or when the transaction commits or when some explicit request is being made). Writing the buffer contents to disk is also called "flushing". A DBMS will usually keep several such buffers in memory.



One can assume that the operating system ensures that writing a block from a memory buffer to disk is atomic. If a case happens where that is not achieved, it is counted as a media failure (of the disk or the disk controller).

From the above follows that it may happen that a change that has been made by a transaction is not permanently recorded yet, when an error occurs.
This may even be the case after the transaction has committed!

Undo and Redo Operations

Undo:
Reverse a write operation made to the database.

Redo:
Repeat a write operation made to the database

A transaction should be atomic, i.e. all or nothing of its operations must be performed. If a transaction is interrupted before the commit, the system must go back to the status of data before the transaction was started. One way of achieving this is by undoing its write operations. This is also called a rollback.

If a transaction has committed and a problem occurs immediately afterwards so that it is not certain whether all its write operations have been successfully transferred to disk, the system must restore the status of data after the commit of the transaction. Usually this is done by redoing, i.e. repeating, these write operations.

Idempotence of Redo and Undo

Undo and Redo can be interrupted by errors and thus may have to be executed several times.
In order to not produce new errors in this situation, the redo and undo functions need to be idempotent:
Undo (Undo (Undo ... (Undo (x)) ...)) = Undo (x) for all operations x
Redo dito.

Definitions

- After image:
  the value of a data item after a transaction has changed it.
- Before image:
  the value of a data item before it was changes by a transaction.
- Log file:
  Many DBMS keep a file in which they store information about the actions of transactions. If a log file exists, in particular, the before images and after images of write transactions are stored in it.

9.2.2 Timing of Writing Updates to the Database

There are different approaches as to when the updates that a transaction has performed on a block in memory are copied to disk. Three principle approaches are described here:

1. Deferred Updates:
In some systems, updates are only made to disk after a transaction has committed (known as *deferred updates*). In this case, a transaction abort does not need to do anything (in particular, no undoes are necessary), since no changes have been made to the database. Recovery only needs to redo the changes of committed transactions. The log file then only needs to store after images, not before images.

Advantage:
Recovery is simpler than in the case where changes may be written to disk before the commit point.

Disadvantage:
Only good in practice when transactions are short and perform few updates. Otherwise too much time is lost at the end, when all changes are finally written. Also: If too much data is changed, the buffer space may not be sufficient.

2. Immediate Updates:

In contrast to that, *immediate updates* means that the updates are written to disk as they happen without waiting for the commit point of the transaction. In this case, recovery requires undoes in the case of unfinished transactions or aborted transactions. When transaction updates are immediately forced to disk, only undoes are necessary. Nothing has to be redone of a committed transaction, because at the commit point, all changes are already safely on disk.

Advantage:
Recovery requires no redoes.

Disadvantage:
Recovery of unfinished transactions requires more work than the deferred updates approach.
Immediately writing changed blocks to disk means that if another transaction wants to change something in the same block, this block will have to be written twice. Writing it just once would be more efficient.

3. Steal/No-Force:

Most systems nowadays use what is called a "steal/no-force" approach, meaning that:
- ***Steal***
  updated blocks may be written to disk before the transaction commits (called "steal", because the block may be "stolen" out of the buffer pool from a running transaction, so that the freed space may be used for a different block, possibly of a different transaction), (the writing is however not necessarily done immediately after the update, so not a force-write)
- ***No-Force***
  blocks with transaction updates are not force-written to disk when the transaction

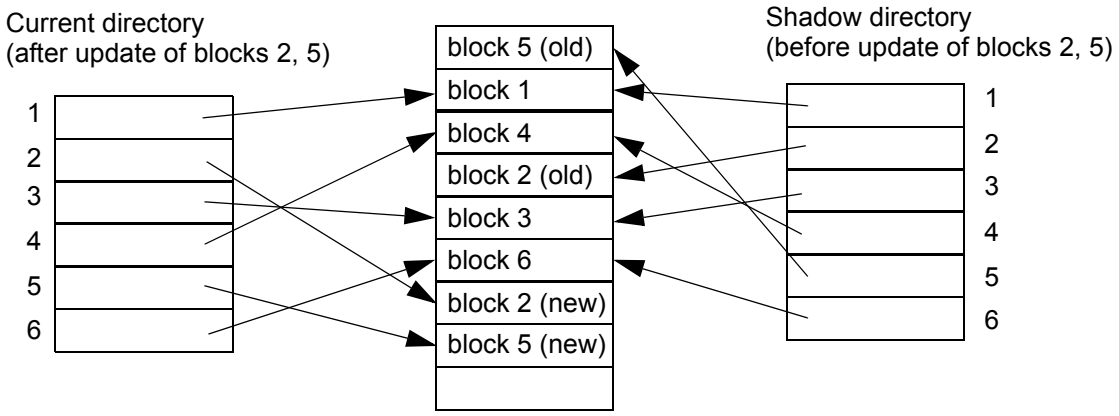
commits[1]. Therefore, after the commit of a transaction not all of its updates may have been written to disk yet.

The advantage of "steal" is that not so many pages need to be kept in memory to account for pending changes (in comparison to keeping all changes in memory until a transaction commits).
The advantage of "no-force" is that in systems with many transactions operating on the same data, the updated block of a committed transaction may still be in memory when a new transaction also wants to make changes to it. Then the I/O time to write it to disk can be saved.

9.2.3 Shadow Paging

This is a way to remember the changes without using a log file and without needing either undoes or redoes for recovery.
The "current directory" is a list of pointers in main memory to the database blocks. At the beginning of a transaction, a copy of this list of pointers to the pages - the "shadow directory" is copied to disk. When a value is changed, the update is performed in the database, but the block containing the new information is stored in a different place from where it was before and the current directory is adjusted. The shadow directory still points to the old place. When the transaction commits, the shadow directory and the referred old pages are freed. If the transaction must be rolled back, the shadow directory is copied to the current directory.

Example for Shadow Paging:



Note:
Shadow paging is not a typically used strategy because of its disadvantages:
- An updated block changes its physical location on disk. This makes it difficult to keep related blocks close together. It also makes the management of file structures more complex.
- Garbage collection must be implemented.
- The change between shadow directory and current directory must be implemented in an atomic way, i.e. mechanisms for recovery in case of interruption must be pro-

1. Log buffers containing the info about transaction updates, however, must be forced to disk.

vided.
- When the disk is corrupted, the before and after images are lost as well. Copying all the concerned blocks to another disk would involve too much I/O. A log file, however, is typically stored on a different disk.

9.2.4 Logging

Changes to the database are written to a log file to record what has been done by transactions. This will be used during recovery.
The log file contains transaction records consisting of:
- transaction identifiers
- type of the log record (begin_transaction, insert, update, delete, commit, abort)
- identifier of the data item affected by the database access
- before-image of the data item (value before the change, in the case of delete or update; in the case of an insert, there is no before-image)
- after-image of the data item (value after the change, in the case of insert or update; in the case of a delete, there is no after-image)
- log management information, such as a pointer to previous and next log records for that transaction.

Note:
Logfiles in MySQL are stored in binary format and thus not directly readable. However, the MySQL utility mysqlbinlog (that you can enter to the MySQL command line prompt) can transform the binary format to a readable file.
Example:
You can type in a Unix shell:
```
shell> mysqlbinlog binlog.000001 > filename.txt
```
where binlog.000001 is the name of a logfile you want to read.
This will store the (probably very large) output of the command to a file named filename.txt, so you can more easily inspect it.
The binary log files in MySQL are typically stored in a directory named /var/lib/mysql.

Write Ahead Logging

Writing information to the log is also a writing process.

Problem:
If the log is written after the database update, the following scenario may happen:
A value is changed in the database. Then an error occurs that leads to a crash.
=> The update has not been logged and thus cannot be undone.

Solution:
The information for the log is written before the update to the database ("write ahead log protocol").
Possible: the log records an update that was never executed.
Question: Why is that not a problem?

„Force at Commit“ Rule

The log information is also written into a buffer in memory first, and not every change may be written to disk immediately. But at the latest when a transaction has reached its commit point, the log buffer is force-written to the disk to ensure that the changes are recorded in the log file.

9.2.5 Recovery Approaches with Logging

Typical approach for recovery in a steal/no force situation, using a log file

1. Periodically, the entire database is copied (backup), typically to tape or (magneto-) optical disks. Backups also can be made incrementally, i.e. only the changes since the last backup are stored.
2. A log file is used to record the changes in the database at each update. Typically, the old and the new value of the updated attribute(s) are stored.
3. When an error occurs:

   a) If the database itself is damaged (media failure, e.g. by a head crash)
   =>
   - The last backup is loaded
   - All committed transactions that were executed since the time of the last backup are repeated as shown by the log. ("redo"). Those transactions that had not committed yet, need to be restarted.

   b) If the database itself is not damaged, but the correctness of its contents cannot be guaranteed (e.g. because a program with updates crashed)
   =>
   - The log is used for repeating committed changes (redo) and reversing non-committed changes (undo). Then, the non-committed transactions can be restarted.

Other approaches, using a log file:

- deferred update (no undo technique)
- no redo technique (all updates are immediately force-written, so no after-images need to be stored)

Checkpoints

The log file can grow very large, in large systems easily 10 GB and more a day.
Therefore:

- Do not keep everything online.
- Set "checkpoints" so that not the entire log file needs to be read for quick recovery of minor failures.

Checkpoints are frequently done, for instance every 5 minutes or every time the log grows by 1 MB.

Procedure for setting a sharp (or quiescent) checkpoint:

1. Temporarily all executing transactions are suspended.

2. Write the contents of the log buffer to disk (force-write).
3. Write all updated database data in buffers to disk (force-write).
4. Write a checkpoint record (with information about the currently active transactions) into the log file on disk (force-write).
5. Write the address of the new checkpoint record into a restart file.
6. Resume the suspended transactions.

Purpose: guarantee that all updates that were still in memory are stored on disk.
=> Everything is saved up to the checkpoint.

Trade-off: Frequent checkpoints guarantee quick restart after errors, but they cost performance.

Note:
The above is also called a *"sharp checkpoint"* or *"quiescent checkpoint"*, since it records everything at the same point in time and the rest of the system is inactive while it happens.

- Advantage: Nothing before the checkpoint needs to be redone at recovery.
- Disadvantage: The suspension of transactions which may cost too much performance.

An alternative is doing a *"fuzzy" (or nonquiescent) checkpoint*. Here, the transactions are allowed to proceed while the checkpoint is being performed. => The time of the checkpoint is no longer "sharp".

Procedure for setting a fuzzy checkpoint:
1. Temporarily suspend all executing transactions
2. Make a list of all dirty[1] log pages and all dirty database pages in memory
3. Make a list of all active transactions
4. Write a checkpoint record (with information about the currently active transactions) into the log file on disk (force-write).
5. Resume execution of transactions
6. Write the contents of the dirty log pages to disk while new transaction records are accepted (written to memory and also transferred to disk).
7. Write all dirty database pages to disk while also new transaction records are accepted (written to memory and also transferred to disk).
8. Write an end of checkpoint record into the log file.
9. Write the address of the new checkpoint record into a restart file.

Steps 6 and 7 here work with low priority; the cache manager performs them only when it has spare time. => It may take some time until all pages are flushed.

Note:
The next checkpoint may only be started after the completion of step 7, i.e. after all dirty pages have been written to disk. Only then can we be sure that all updates up to the start of this checkpoint are safely in the log and on disk. This means that when the

1. Dirty means: updated and thus needs to be written to disk. Pages = blocks.

next (fuzzy) checkpoint record is written, we can only be sure about the updates that happened before the previous checkpoint!

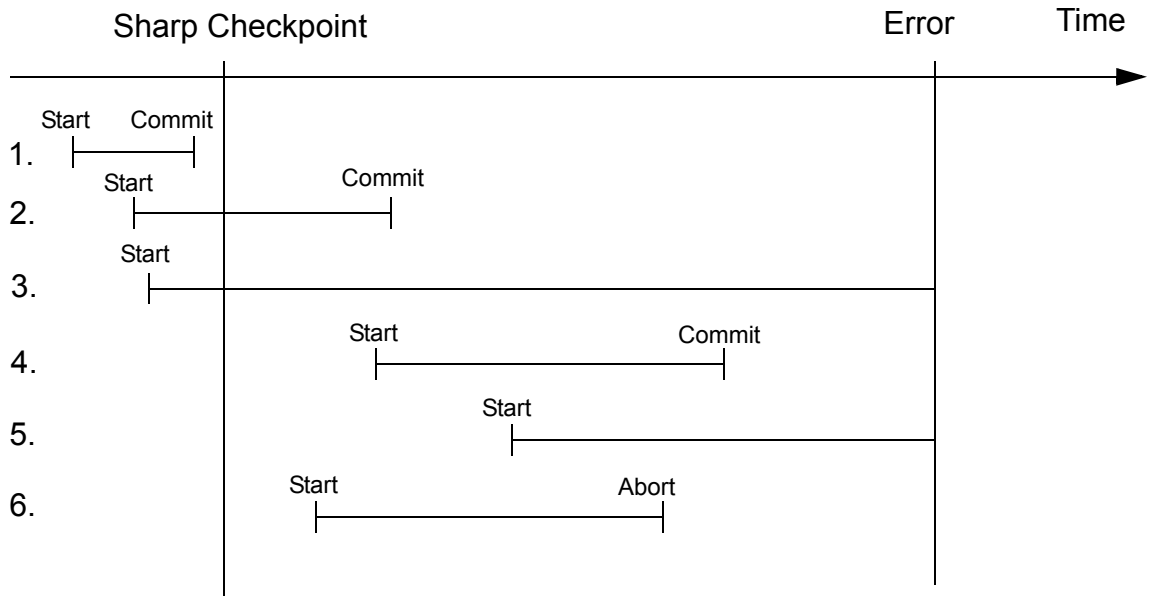
- Advantage: better system performance (which is why this is usually done).
- Disadvantage: During the execution of the checkpoint, new transaction log records are created. Redo must start with the transaction log record that was being done when the checkpoint started and undo must go back as far as it. Or, if the failure occurs while the checkpoint was still in process, one must go back to the checkpoint before it.

Since it is advantageous for quick recovery to make frequent checkpoints, their execution algorithms are heavily optimized in most systems. (See literature, for instance Bernstein, Newcomer: "Principles of Transaction Processing for the System Professional", Morgan Kaufmann; or Gray, Reuter: "Transaction Processing - Concepts and Techniques", Morgan Kaufmann).

Recovery with Log and Checkpoints

Recovery with sharp Checkpoints:

We assume a steal / no force approach.
In case of a software error (database on disk not damaged), the following scenarios can occur:



1. No recovery required (during the checkpoint, everything was written to disk.

2. Redo required, at least for all changes after the checkpoint (since there is no guarantee that they have been written to disk).

3. Undo required, since the transaction did not commit.

4. Complete redo required.

5. Undo, since the transaction did not commit.

6. Undo required, since the transaction aborted. The abort procedure was supposed to undo all changes, but there is no guarantee that all changes have been restored on disk yet.

Procedure of the recovery manager (a software module):

I. Preparation

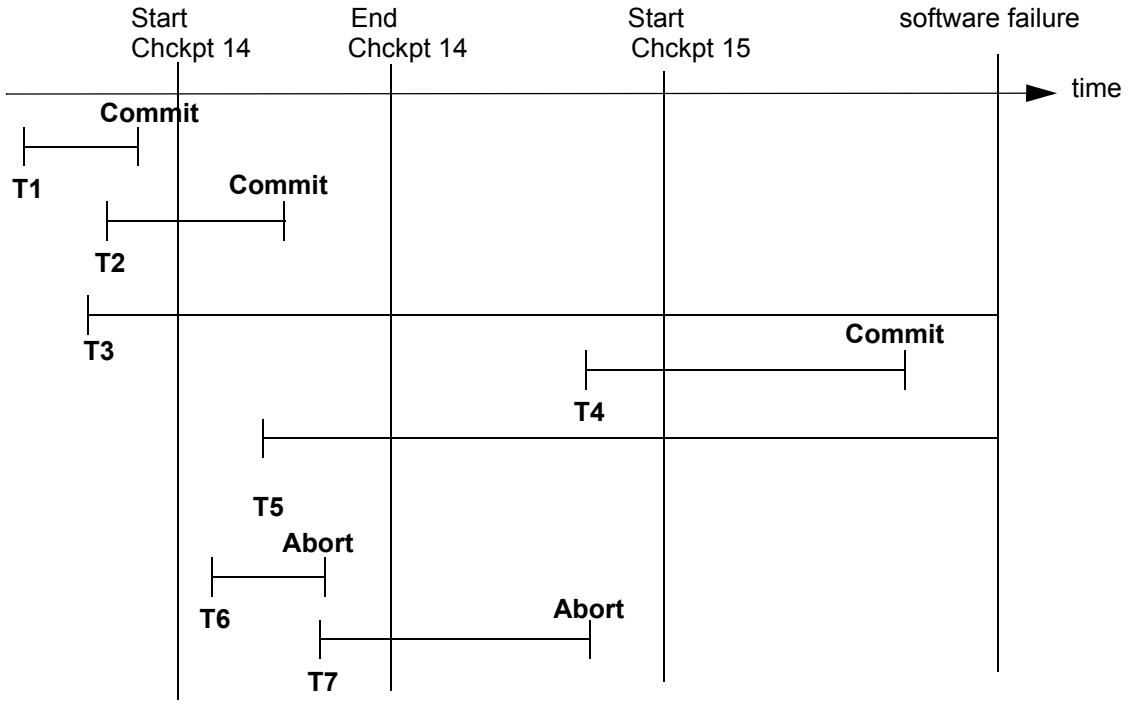
1. Create an UNDO list and a REDO list.
2. Append all active transactions of the last checkpoint to the UNDO list.
3. Search the log file, starting with the last checkpoint:
   - If a Begin_Transaction is found, append the respective transaction to the UNDO list.
   - If a COMMIT is found, transfer the respective transaction from the UNDO list to the REDO list.

II. Recovery

1. Proceed backwards through the log file: undo all transactions in the UNDO list, using the logged information.
2. Proceed forwards through the log file (starting with the last checkpoint): redo all transactions in the REDO list.

Recovery with fuzzy checkpoints:

We assume a steal / no force approach.
In case of a software error (database on disk not damaged), the following scenarios can occur:



1. T1: No recovery needed, because all dirty blocks of T1 were collected at the start of Checkpoint 1 and written to disk until the end of Checkpoint 14.

2. T2: Redo all changes of T2, at least those that happened after the start of Checkpoint 14.

3. T3: Undo all changes (those made before the start of Checkpoint 14 and those made after), because the transaction was not successful.

4. T4: Redo all changes of T4 because it is not certain which changes have already been transferred to disk.

5. Analogous to case 3.

6. T6: Undo all changes of T6, because it is not certain whether the entire rollback has been written back to disk yet.

7. T7: Like case 6.

A note on the order of redo and undo operations:

The typical recovery algorithms perform first undo backwards and then redo forwards. This is the simplest and safest way of implementing it. Given some information about the concurrency control mechanism, optimizations and variations are possible.

Checkpoints in MySQL

Excerpt from the online MySQL documentaton:

"InnoDB implements a checkpoint mechanism called a "fuzzy checkpoint." InnoDB will flush modified database pages from the buffer pool in small batches. There is no need to flush the buffer pool in one single batch, which would in practice stop processing of user SQL statements for a while.

In crash recovery, InnoDB looks for a checkpoint label written to the log files. It knows that all modifications to the database before the label are already present in the disk image of the database. Then InnoDB scans the log files forward from the place of the checkpoint, applying the logged modifications to the database.

InnoDB writes to the log files in a circular fashion. All committed modifications that make the database pages in the buffer pool different from the images on disk must be available in the log files in case InnoDB has to do a recovery. This means that when InnoDB starts to reuse a log file in the circular fashion, it has to make sure that the database page images on disk already contain the modifications logged in the log file InnoDB is going to reuse. In other words, InnoDB has to make a checkpoint and often this involves flushing of modified database pages to disk.

The preceding description explains why making your log files very big may save disk I/O in checkpointing. It can make sense to set the total size of the log files as big as the buffer pool or even bigger. The drawback of big log files is that crash recovery can take longer because there will be more logged information to apply to the database."

Another Example: ARIES

A practical example of a family of recovery algorithms that implement such strategies is ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) developed by C. Mohan at IBM Almaden Research Center. The ARIES algorithms are used for instance by DB2, Sybase, and Microsoft SQL Server as well as quite a few research DBMSs. For a description see for instance Elmasri/Navathe or http://www.almaden.ibm.com/u/mohan/ARIES_Impact.html

9.3 Recovery from Media Failures

The above methods describe recovery when the disk where the database resides is not harmed. In case of media failure like a disk crash, recovery must use an archived backup plus as much of the log file as was archived to a safe place.

Archiving strategies:

- Full dump:
  The simplest way of making a backup is suspending transaction execution and making a full copy (a full dump) of the database to some other disk or tape archive.

=> This may take a long time with a large database and may not be feasible with a heavily used system.

- Incremental dump:
  From time to time, copy only those parts that were changed to the archive
  => This is faster than a full dump, but if transactions are suspended it may still cost too much performance.
- Nonquiescent archiving:
  Allow transactions to execute while running a full or incremental dump.
  => The dump in the archive may not reflect any snapshot of the real database.
  Solution: Store all changes made by the transactions that are running during the execution of the dump in a log file and archive this log file also. Restoring the database is done by using the incremental backups (first to newest) and then using the log to bring it up to a consistent state.
  Procedure:
  1. Write a log record START DUMP.
  2. Perform a checkpoint (quiescent or non-quiescent)
  3. Perform a full or incremental backup of the data on disk.
  4. Copy the log file to a secure archive as well.
  5. Write a log record END DUMP and copy it to the archive as well.

<u>Recovering after a media failure of the database:</u>

1. Restore the database from the archive:
   - find the most recent full dump and copy it into the database
   - use the following incremental dumps to modify the database accordingly (first to last)
2. Modify the database using first the log file in the archive and then the current log file.

Example (see Garcia-Molina, Ullman, Widom):
Logrecords made during the dump:

```
START DUMP
START CHECKPOINT (T1, T2)                    //T1 and T2 are active
T1, A, 1, 5                          // T1 updates A, old value: 1, new value: 5
T2, C, 3, 6
COMMIT T2
T1, B, 2, 7
END CHECKPOINT
START T3
T3, D, 1, 10
T1, E, 2, 5
                                     // Dump completes
END DUMP
```

The transaction updates after the start of the dump may not have been stored in the archive.
=> In case of a media failure after this dump, the backup is loaded and these log records are used to redo T2 (because it committed) and undo T1 and T3 (because they did not), unless commit records for them can be found in the following logfile.

Shadowed Disks and RAID

Recovering from a media failure can take a significant amount of time. To reduce it, many systems use the concept of shadowed disks: Each disk required to store the database is duplicated on another disk. All write operations are performed on both disks in parallel so that the contents are always identical. If one fails, the other is still available.

Another option is RAID:

RAID (Redundant Array of Independent[1] Disks) was first patented by IBM in 1978. A RAID appears to the operating system to be a single logical hard disk. The data on these disks can be duplicated as well as distributed in a way that accessing it is much faster than from a single disk (e.g. by distributing one large record over several disks, so that its parts can be read in parallel which reduces I/O time).

Data striping:
A file is distributed over several disks and can be accessed in parallel.
With striping over n disks, I/O time will be about n times faster than with a single disk.
Striping can be done for instance at block-level or at bit-level.

A RAID can also be used to improve reliability by keeping copies of data on several disks that are treated as one logical disk. This is called mirroring or shadowing. When one disk fails, another disk can be used until the first one is repaired. Usually, two disks are used.
Problem:
The more disks are used, the lower the reliability of the system.
Reason: Reliability of a disk can be defined as Mean Time to Failure (MTTF). A RAID then has the reliability (Average MTTF of the disks) / number of disks. (Intuitively: The probability that one of a set of devices (disks, lightbulbs, whatever) fails in a given period of time is higher than the probability of failure for a just a single one. In a house with many lightbulbs, all the time a bulb needs to be replaced. If you have just one lamp, you have to change lightbulbs only rarely.
Two disks are therefore approximately only half as reliable as a single one.

Example: If mean time to failure (MTTF) for a single disk is 200,000 hours (~ 22.8 years), the MTTF for a set of 100 disks (for instance at a bank) is only 2000 (= 200,000 / 100) hours (83.3 days)! => High risk that data is lost without backups.

<u>Advantages of RAID:</u>
Using RAID can increase
- throughput (Reading from several disks in parallel is faster than reading from one disk only)
- fault-tolerance (If one disk fails, the others may still be available)
- data integrity (Redundancy can help eliminate errors)

1. This used to be called Redundant Array of *Inexpensive* Disks

There are different types of RAID, classified into "levels". The level naming underwent semantic changes over time and thus the terms are sometimes used somewhat inconsistently.

Disaster

A disaster is an unrecoverable error. Examples how this can happen are serious media failures, especially if parts of the log file are lost because it may contain update information (before images and after images) that may not be available elsewhere.

It is a good strategy to store the log always on a different disk than the database itself to ensure that in case the disk with the database crashes, the log is not lost as well. Optimally, the log is stored additionally on a shadow disk.