Architecture and Implementation of Database Systems (Summer 2018)

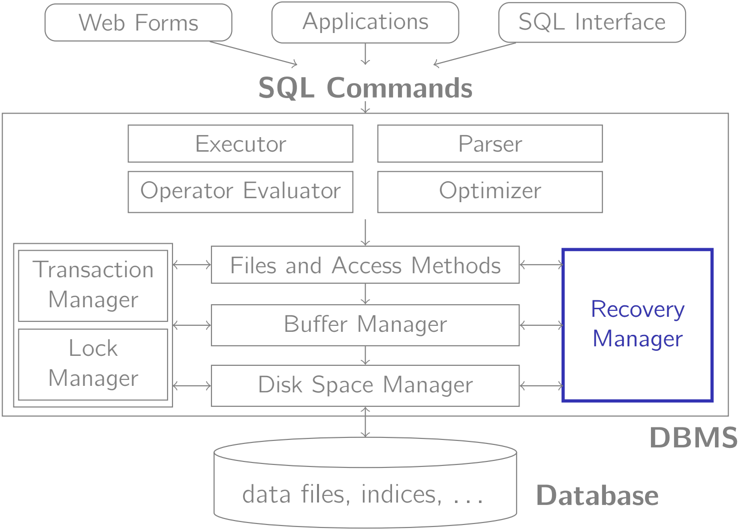
Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Summer 2018

Part IX

Recovery

Recovery


Web Forms
Applications
SQL Interface
SQL Commands
Executor
Parser
Operator Evaluator
Optimizer
Transaction Manager
Lock Manager
Files and Access Methods
Buffer Manager
Disk Space Manager
Recovery Manager
DBMS
data files, indices, . . .
Database

Failure Recovery

We want to deal with **three types of failures**:

transaction failure (also: ‘process failure’)

A transaction voluntarily or involuntarily **aborts**. All of its updates need to be **undone**.

system failure

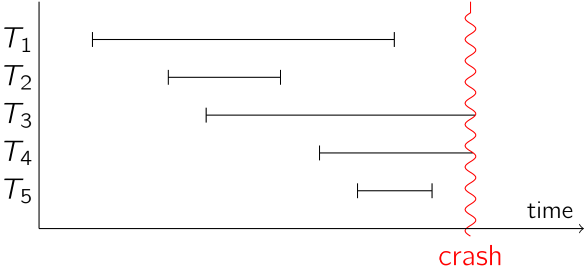
Database or operating **system crash**, power outage, etc. All information in main memory is lost. Must make sure that **no committed transaction is lost** (or **redo** their effects) and that all other transactions are **undone**.

media failure (also: ‘device failure’)

Hard disk crash, catastrophic error (fire, water, . . .). Must **recover database** from stable storage.

In spite of these failures, we want to guarantee **atomicity** and **durability**.

Example: System Crash (or Media Failure)



- Transactions T_1, T_2, and T_5 were committed before the crash.
  - → **Durability:** Ensure that updates are **preserved** (or **redone**).
- Transactions T_3 and T_4 were not (yet) committed.
  - → **Atomicity:** All of their effects need to be **undone**.

Types of Storage

We assume three different types of storage:

volatile storage

This is essentially the **buffer manager**. We are going to use volatile storage to **cache** the **write-ahead log** in a moment.

non-volatile storage

Typical candidate is a **hard disk**.

stable storage

Non-volatile storage that survives all types of failures. Stability can be improved using, *e.g.*, (network) **replication** of disk data. Backup tapes are another example.

Observe how these storage types correspond to the three types of failures.

Shadow Pages

- Since a failure could occur **at any time**, it must be made sure that the system can **always** get back to a consistent state.
- Need to keep information **redundant**.
- System R: **shadow pages**. Two versions of every data page:
  - The **current version** is the system's "working copy" of the data and may be inconsistent.
  - The **shadow version** is a consistent version on stable storage.
- Use operation `SAVE` to save the current version as the shadow version.
  - `SAVE` ↔ **commit**
- Use operation `RESTORE` to recover to shadow version.
  - `RESTORE` ↔ **abort**

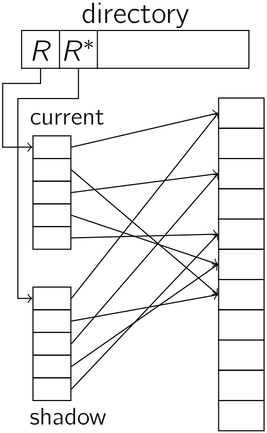
Shadow Pages

1. Initially: shadow $\equiv$ current.
2. A transaction T now changes the **current** version.
   - Updates are **not** done in-place.
   - Create new pages and alter current page table.

3a. If T **aborts**, overwrite current version with shadow version.

3b. If T **commits**, change information in **directory** to make current version persistent.

4. Reclaim disk pages using **garbage collection**.



Shadow Pages: Discussion

- Recovery is instant and fast for **entire files**.
- To guarantee **durability**, all modified pages must be **forced** to disk when a transaction **commits**.
- As we discussed on slide 40, this has some undesirable effects:
  - high I/O cost, since writes cannot be cached,
  - high response times.
- We'd much more like to use a **no-force** policy, where write operations can be deferred to a later time.
- To allow for a no-force policy, we'd have to have a way to **redo** transactions that are committed, but haven't been written back to disk, yet.

↗ Gray *et al.*. The Recovery Manager of the System R Database Manager. *ACM Comp. Surv.*, vol. 13(2), June 1981.

Shadow Pages: Discussion

- Shadow pages do allow **frame stealing**: buffer frames **may** be written back to disk (to the "current version") **before** the transaction T commits.
- Such a situation occurs, *e.g.*, if another transaction T' wants to use the space to bring in its data.
  - T' **"steals"** a frame from T.
  - Obviously, a frame may only be stolen if it is **not pinned**.
- Frame stealing means that **dirty** pages are written back to disk. Such writes have to be **undone** during recovery.
  - Fortunately, this is easy with shadow pages.

Effects on Recovery

- The decisions force/no force and steal/no steal have implications on what we have to do during recovery:

	force	**no force**
no steal	no redo no undo	must redo no undo
steal	no redo must undo	must redo must undo

- If we want to use **steal** and **no force** (to increase concurrency and performance), we have to implement **redo** and **undo** routines.

Write-Ahead Log

- The **ARIES**[21] recovery method uses a **write-ahead log** to implement the necessary redundancy. Data pages are updated **in place**.

  ↗ Mohan *et al.* ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, vol. 17(1), March 1992.

- To prepare for **undo**, undo information must be written to stable storage **before** a page update is written back to disk.
- To ensure **durability**, **redo** information must be written to stable storage **at commit time** (no-force policy: the on-disk data page may still contain old information).

[21] Algorithm for Recovery and Isolation Exploiting Semantics

Content of the Write-Ahead Log

LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

LSN (Log Sequence Number)

Monotonically increasing number to identify each log record.
Trick: Use byte position of log record ✎ **Why?**

Type (Log Record Type)

Indicates whether this is an **update record** (`UPD`), **end of transaction record** (`EOT`), **compensation log record** (`CLR`), . . .

TX (Transaction ID)

Transaction identifier (if applicable).

Content of the Write-Ahead Log (cont.)

Prev (Previous Log Sequence Number)

LSN of the preceding log record written by the same transaction (if applicable). Holds '–' for the **first** record of every transaction.

Page (Page Identifier)

Page to which updates were applied (only for `UPD` and `CLR`).

UNxt (LSN Next to be Undone)

Only for `CLR`. Next log record of this transaction that has to be processed during **rollback**.

Redo

Information to **redo** the operation described by this record.

Undo

Information to **undo** the operation described by this record. Empty for `CLR`.

Example

Transact. 1	Transact. 2	LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
$a \leftarrow \texttt{rd}(A)$;									
	$c \leftarrow \texttt{rd}(C)$;								
$a \leftarrow a - 50$;									
	$c \leftarrow c + 10$;								
$\texttt{wr}(a, A)$;		1	UPD	T_1	–	$\cdots$		$A := A - 50$	$A := A + 50$
	$\texttt{wr}(c, C)$;	2	UPD	T_2	–	$\cdots$		$C := C + 10$	$C := C - 10$
$b \leftarrow \texttt{rd}(B)$;									
$b \leftarrow b + 50$;									
$\texttt{wr}(b, B)$;		3	UPD	T_1	1	$\cdots$		$B := B + 50$	$B := B - 50$
`commit` ;		4	EOT	T_1	3	$\cdots$			
	$a \leftarrow \texttt{rd}(A)$;								
	$a \leftarrow a - 10$;								
	$\texttt{wr}(a, A)$;	5	UPD	T_2	2	$\cdots$		$A := A - 10$	$A := A + 10$
	`commit` ;	6	EOT	T_2	5	$\cdots$			

- `rd` is for "read"; `wr` is for "write"

What Redo/Undo Information to Log

Redo/undo information can be **encoded in different ways**.

In **physical logging**, the exact byte representation of every page is faithfully logged and preserved.

- *E.g.*, **before** and **after image** of the entire page.
- Typically try to be smarter: log only modified parts of the page and/or compress log entries.

Advantages:

- Recovery mechanism is **object-independent** (whether the page is an index/data/... page doesn't matter).
- Recovery is **page-oriented** (and pages are the granularity for atomic data changes on disk).

Disadvantages:

- **Log volume** can become very large. Logs are a key limitation of today's transaction processing systems.

Physical Logging

Disadvantages (cont.):

- A **transaction abort** might force other transactions to abort when they altered the same pages.
  - → This can happen even if the transactions do **not conflict** on the logical level.

Observe that

- Physical logging not only preserves the logical database content but also its **physical representation**.
- To this end, **every** page modification must be logged.
  - → Even cleanup operations or internal page re-organizations.

As such, physical logging does **more than needed**.

- The physical representation of data is not visible to the user.
- ACID only refers to the logical representation.

Logical Logging

Logical logging is an alternative.

- Log **high-level operations**
  - → *E.g.*, "insert tuple $\langle \dots \rangle$ into table R"
- A single such log record often implies a **series of changes**.
  - → Insert tuple in data pages, indexes, etc.
  - → May have to split index pages, allocate new heap space, etc.
- Logical logging will **not** preserve the physical representation.
  - → *E.g.*, don't undo an index page split.
  - → During redo, tuples or index entries might end up on completely different pages.

Logical Logging

Advantages:

- Log volume **very** small.
  - → Individual log entries very small.
  - → Maintenance operations need not be logged at all.
- Potential to improve undo/recovery performance
  - → *E.g.*, don't undo a page split

Disadvantages:

- Very hard to get right.
  - → Logged operations are not atomic with respect to disk operations.
  - → Idempotency of redo/undo operations?

Problems of Logical Logging

Example:

- The insertion of a tuple into table T implies a new entry in indexes A and B.

Problem 1: Partial Actions

- A **transaction failure** could occur at any of the three steps.
- Need to know which prefix of changes to T, A, B has to be undone.

Problem 2: Action Consistency

- In the case of **crash recovery**, any subset of the affected pages could have reached the disk before the crash.

It is even worse:

- A B-tree insertion itself may affect multiple pages (splits, etc.).

Compromise: Physiological Logging

Idea:

- *"physical to a page, logical within a page"*

Physical part:

- Every log record refers to a particular **physical page**.

Logical part:

- Use logical logging to describe changes **within one page**.

Example log entry:

$$[\ldots, \text{insert}, \ldots, \text{page 4711}, \ldots, \text{record value } r]$$
$$[\ldots, \text{ix insert}, \ldots, \text{ix page 0815}, \ldots, \text{ix key: } k_1, \text{rid: } v]$$
$$[\ldots, \text{ix insert}, \ldots, \text{ix page 4242}, \ldots, \text{ix key: } k_2, \text{rid: } v]$$

Physiological Logging

Important: page action consistency

- Complex actions composed of **single-page actions**, each such action is **logged**.
- Pages may be inconsistent during single-page action, but this situation is **protected by latches**.

In practice:

- Write undo/redo log record before releasing the page latch.
- Page modifications are **atomic** from the perspective of the logging/recovery mechanism.

Idempotency

The **idempotency** challenge of logical logging remains.

E.g., insert log record for new tuple t

→ t should be inserted exactly **once**.

→ Must not re-insert t (again and again), *e.g.*, in case of **crash recovery**.

Thus:

- Assign a **unique, monotone log sequence number (LSN)** to each log entry.
- Record the LSN of the **latest page update** in each page header.
- During redo, apply operation **only if** $\mathit{pageLSN} < \mathit{logLSN}$.

Redo instructions themselves need **not** be idempotent!

Writing Log Records

- For performance reasons, all log records are first written to **volatile storage**.
- At certain times, the log is **forced to stable storage** up to a certain LSN:
  - All records until T's `EOT` record are forced to disk when T **commits** (to prepare for a **redo** of T's effects).
  - When a **data page** p **is written** back to disk, log records up to the last modification of p are forced to disk (such that uncommitted updates on p can be **undone**).
- The log is an ever-growing file (but see later).

Normal Processing

During normal transaction processing, keep two pieces of information in each Transaction Control Block (slide 328):

LastLSN (Last Log Sequence Number)
: LSN of the last log record written for this transaction.

UNxt (LSN Next to be Undone)
: LSN of the next log record to be processed during **rollback**.

Whenever an update to a page *p* is performed,

- a **log record** *r* is written to the WAL and
- The **LSN** of *r* is recorded in the **page header** of *p*.

Transaction Rollback

To **roll back** a transaction T after a **transaction failure**:

- Process the log in a **backward** fashion.
- Start the **undo** operation at the log entry pointed to by the UNxt field in the transaction control block of T.
- Find the remaining log entries for T by following the Prev and UNxt fields in the log.

Undo operations modify pages, too!

→ **Log** all undo operations to the WAL.

→ Use **compensation log records (CLRs)** for this purpose.

Transaction Rollback

```
Function: rollback(SaveLSN, T)
UndoNxt ← T.UNxt ;
while SaveLSN < UndoNxt do
    LogRec ← read log entry with LSN UndoNxt ;
    switch LogRec.Type do
        case UPD do
            LSN ← write log entry
                    ⟨CLR, T, T.LastLSN, LogRec.Page, LogRec.Prev, ··· , ∅⟩ ;
            perform undo operation LogRec.Undo on page LogRec.Page ;
            in page header of LogRecPage: set LSN = LSN ;
            T.LastLSN ← LSN ;
            UndoNxt ← LogRec.Prev ;
        case CLR do
            UndoNxt ← LogRec.UNxt ;
    T.UNxt ← UndoNxt ;
```

Undo Processing

Write **compensation log records (CLRs)** during undo.

- The **redo** information in the CLR describes the performed **undo** operation.
- The undo operation **increases** the page's LSN.

✎ **Why?**

Undo Processing

Undo need **not** precisely re-establish the page to the representation before the corresponding ‘do’ operation.

E.g.,

- The ‘undo’ for an ‘insert’ might be a ‘delete’.
  → The deleted record might remain as a ghost.
- A B-tree node split might not be un-done at all.
- An insert might have required page compaction.

Undo only needs to re-establish the **logical** contents of a page, but **not** its physical representation.

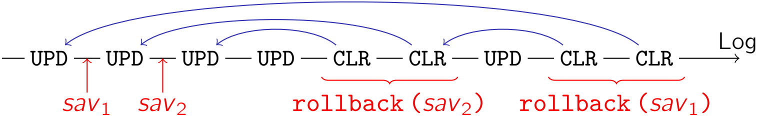
→ This is the “logical” aspect of “physiological”.

Transaction Rollback

- Transaction can be rolled back **partially** (back to *SaveLSN*).

✎ **Why is this useful?**

- The UNxt field in a CLR points to the log entry before the one that has been undone.



Crash Recovery

Restart after a **system failure** is performed in **three phases**:

1. **Analysis Phase:**
   - Read log in **forward** direction.
   - Determine all transactions that were **active** when the failure happened. Such transactions are called **losers**.
2. **Redo Phase:**
   - **Replay** the log (in **forward** direction) to bring the system into the state as of the time of system failure.
3. **Undo Phase:**
   - **Roll back** all loser transactions, reading the log in a **backward** fashion (similar to “normal” rollback).

Analysis Phase

```
Function: analyze ()
foreach log entry record LogRec do
    switch LocRec.Type do
        create transaction control block for LogRec.TX if necessary ;
        case UPD or CLR do
            LogRec.TX.LastLSN ← LogRec.LSN ;
            if LocRec.Type = UPD then
                LogRec.TX.UNxt ← LogRec.LSN ;
            else
                LogRec.TX.UNxt ← LogRec.UNxt ;
        case EOT do
            delete transaction control block for LogRec.TX ;
```

- In practice, systems also use the analyze phase to collect further information, *e.g.*, to **prefetch** pages for redo.

Redo Phase

```
1 Function: redo ()
2 foreach log entry record LogRec do
3     switch LocRec.Type do
4         case UPD or CLR do
5             $v \leftarrow$ pin (LogRec.Page) ;
6             if $v$.LSN < LogRec.LSN then
7                 perform redo operation LogRec.Redo on $v$ ;
8                 $v$.LSN $\leftarrow$ LogRec.LSN ;
9             unpin ($v$, $\cdots$) ;
```

System crashes can occur **during** recovery!

- Undo and redo of a transaction T must be **idempotent**:

$$\begin{aligned} \text{undo}(\text{undo}(T)) &= \text{undo}(T) \\ \text{redo}(\text{redo}(T)) &= \text{redo}(T) \end{aligned}$$

- Check LSN before performing the redo operation (line 6).

Redo Phase

- Note that we redo **all** operations (even those of losers) and in **chronological order**.
- After the redo phase, the system is in the **same state as it was at the time of the system failure**.

  Some **log entries** may not have found their way to the disk before the failure. Committed operations would have been written to disk, though (slide 402). All others would have to be undone anyway.
- We'll have to **undo** all effects of **loser transactions** afterwards.

- As an optimization, the analyze pass could instruct the buffer manager to **prefetch** dirty pages.

Undo Phase

- The **undo phase** is similar to the rollback during "normal processing".
- This time we roll back **several transactions** (all losers) at once.
- All loser transactions are rolled back completely (not just up to some savepoint).

```
Function: undo ()
while transactions (i.e., TCBs) left to roll back do
    $T \leftarrow$ TCB of loser transaction with greatest UNxt ;
    $LogRec \leftarrow$ read log entry with LSN $T$.UNxt ;
    switch $LogRec$.Type do
        case UPD do
            $LSN \leftarrow$ write log entry
                    $\langle$CLR, $T$, $T$.LastLSN, $LogRec$.Page, $LogRec$.Prev, $\cdots$ , $\emptyset\rangle$ ;
            perform undo operation $LogRec$.Undo on page $LogRec$.Page ;
            in page header of $LogRec$.Page: set LSN $= LSN$ ;
            $T$.LastLSN $\leftarrow LSN$ ;
            $UndoNxt \leftarrow LogRec$.Prev ;
        case CLR do
            $UndoNxt \leftarrow LogRec$.UNxt ;
    $T$.UNxt $\leftarrow UndoNxt$ ;
    if $T$.UNxt = '–' then
        write EOT log entry for $T$ ;
        delete TCB for $T$ ;
```

Forcing Log Records to Stable Storage

The effects of all **committed** transactions must be durable.

→ When committing a transaction T, **force** the log to stable storage (at least) up until the **commit record** of T.

Conversely, in case of a **crash recovery**, any effects caused by transactions that did **not yet commit** must be undone.

→ When **evicting** a page p from the buffer pool, first **force** the log to stable storage until the LSN recorded in p.

✎ **What about aborted transactions? Force abort records, too?**

Content vs. Representation

Logical (and mostly also physiological) logging protects only the **logical content** of the database.

Physical logging, by contrast, also protects the **physical representation**.

We saw something similar before:

- **User transactions** perform changes to the **logical content** of the database.
- **System transactions** only change the **physical representation**.

Indeed, the distinction allows us to reduce the overhead of logging.

System Transactions

System transactions need to log their operations just like user transactions.

Example: B-tree node split

- Migration of tuples to new node changes ("logical") content of those pages (even if not of the B-tree overall), which thus has to be logged.
- Space allocation must be logged.

But:

- Whether or not the system transaction T_x **commits** is immaterial until a user transaction depends on changes done by T_x.
- System transactions do **not** have to **force** the log to stable storage at commit time.
- A dependent user transaction (upon commit) will implicitly force the effects of T_x to stable storage.

System Transactions

In case the invoking user transaction **aborts**, system transactions do not have to be rolled back.[22]

- → Reduced rollback overhead compared to alternative without system transactions.
- → **Other transactions** that already saw the effects of invoked system transactions need **not** be rolled back.
  - *E.g.*, B-tree node split: other transactions might already have put new entries on new node.

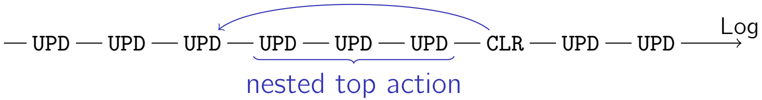
[22] In fact, they cannot be rolled back if they already committed.

Nested Top Actions (Without System Transactions)

Similar effects **can** be achieved also without system transactions.

Trick:

- When finishing a “nested top action,” write a **dummy CLR** to the log:



UPD — UPD — UPD — UPD — UPD — UPD — CLR — UPD — UPD → Log

nested top action

- In case of an **undo**, processing will **skip** over all actions of the nested top action, thus preserve its effects.

Logging Overhead for System Transactions

System transactions lead to **additional** log records.

→ At a minimum, there is the added commit record.

However, system transactions may also **reduce** log volume.

Example: Tuple deletion

- **Without sytem transactions:**
  - Need to log deletion (for redo) **and** deleted tuple value (for undo).
- **With system transactions:**
  - Turn tuple in to **ghost** only (log only bit flip).
  - Once the user transaction has committed, the tuple is logically NULL. A clean-up system transaction thus need **not** log deleted tuple value (only the key).

Log Fusion

Such delete operations can be optimized even further.

Step 1:

- **Merge** `UPD` for ghost deletion and `EOT` (commit record) into single log record.

  (The operation occurs frequent enough to warrant a special entry type.)
- There is now only a **single log record** when a system transaction deletes a ghost.

Step 2:

- Since there is only a single log record, the system transaction cannot fail in-between deletion and commit.
- Thus: **omit** logging undo operations altogether.

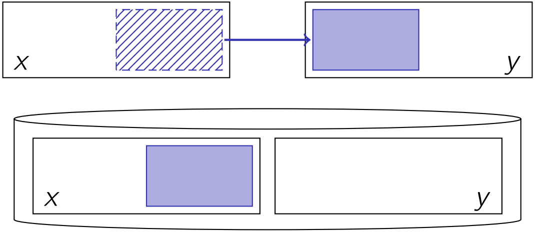
Write Ordering

Log volume can further be reduced by careful **write ordering**.

Example: B-tree node split

→ A log record like *"Move entries $k_i, \ldots, k_j$ from page x to page y."* covers all information needed for the two split nodes x and y.

The operation is performed **in memory** first:



If x is **written back** to disk **before** y, data behind $k_i, \ldots, k_j$ is no longer persistent on disk!

Write Ordering

There is **no problem** if page y is written back **first**.

Thus: Add **write order** information to buffer pool meta data.

1. Let destination page y **depend** on x:
   - Add dependency pointer to y.
   - Increment a **reference counter** in x for each dependence; decrement it when y is flushed to disk.
   - Flush pages only when their reference counter is 0.
2. Alternatively: For each page maintain a **list of pages** that have to be flushed first.
   - Add pointer $x \rightarrow y$ to x.
   - When x is chosen for replacement, flush all referenced pages first.

For **append-heavy indexes**, write ordering can lead to "write convoys".

Checkpointing

- We've considered the WAL as an ever-growing log file that we read **from the beginning** during crash recovery.
- In practice, we do not want to replay a log that has grown over days, months, or years.
- Every now and then, write a **checkpoint** to the log.
  - (a) **heavyweight checkpoints**
    Force all dirty buffer pages to disk, then write checkpoint.
    Redo pass may then start at the checkpoint.
  - (b) **lightweight checkpoints** (or "fuzzy checkpoints")
    Do not force anything to disk, but write information about dirty pages to the log. Allows redo pass to start from a log entry shortly **before** the checkpoint.

Fuzzy Checkpointing

Periodically write checkpoint in three steps:

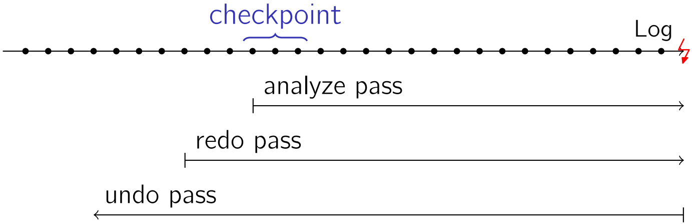
1. Write **begin checkpoint** log entry `BCK`.
2. Collect information about
   - all **dirty pages** in the buffer manager and the LSN of the **oldest** update operation that modified them and
   - all **active transactions** (and their LastLSN and UNxt TCB entries).

   Write this information into the **end checkpoint** log entry `ECK`.
3. Set **master record** at a known place on disk to point to the LSN of the `BCK` log entry.

Recovery with Fuzzy Checkpointing

During crash recovery

- start **analyze pass** at the `BCK` entry recorded in the master record (instead of from the beginning of the log).
- When reading the `ECK` log entry,
  - Determine **smallest LSN** for **redo** processing and
  - Create TCBs for all transactions in the checkpoint.



Media Recovery

- To allow for recovery from **media failure**, periodically **back up** data to stable storage.
- Can be done **during normal processing**, if WAL is archived, too.
- If the backup process uses the **buffer manager**, it is sufficient to archive the log starting from the moment when the backup started.
  - Buffer manager already contains freshest versions.
  - Otherwise, log must be archived starting from the oldest write to any page that is dirty in the buffer.

- Other approach: Use log to **mirror** database on a remote host (send log to network **and** to stable storage).

Locking and Recovery: Transaction Rollback

✎ **What locks have to be acquired during a transaction rollback?**

- In strict two-phase locking, all locks are kept until the transaction commits.
  - → Locks are still held, **no new locks** have to be acquired **during** rollback.
- This also means that a transaction **cannot** run into a **deadlock** situation during rollback.

Locking and Recovery: Crash Recovery

✎ **And what about locking after a crash?**

- Concurrency issues have **already been resolved** when the transactions were normally running.
  - → **No need** to isolate them again during recovery.

- **New transactions**, issued after restart, **might** conflict with those of the recovery process.
  - → If new transactions are allowed to enter the system **during** the recovery process, locks must be acquired (for old and new transactions).
  - → Since "old" transactions don't conflict with each other, they can all run under the **same** recovery transaction.

Locking and Recovery: Crash Recovery

The **log analysis pass** helps fast recovery/early restart of new transactions.

- With the analysis pass, determine which locks have to be acquired for recovery.
  - → Note that log analysis runs relatively fast, because it does only a **sequential read** of the log.
  - → Analysis runs faster if **checkpoints** are done more often.
  - → To acquire locks, list locks held by **indoubt transactions** in **checkpoint information**.
  - → Locks **cannot conflict** at this stage (→ speed-up analysis)

Locking and Recovery: Fast Restart

- Once all locks are acquired, **new transactions** can be allowed into the system.
  - → Locks for recovery need not (necessarily) be acquired at the **same granularity** as the original transactions did.
    (Again, this might help speed-up the analysis pass.)

The actual redo/undo takes much longer than log analysis.

→ Many **data pages** have to be fetched in **random order**.

Effectively, many pages will be read **unnecessarily**.

→ Often, the disk will already contain the **latest version** of the data. But we cannot tell that in advance just from analyzing the log.

→ Possible improvement: Log **write-back** of pages, so analysis pass can detect the situation and avoid unneccessary page reads.

Wrap-Up

ACID and Serializability

To prevent from different types of **anomalies**, DBMSs guarantee **ACID properties**. **Serializability** is a sufficient criterion to guarantee **isolation**.

Two-Phase Locking

Two-phase locking is a practicable technique to guarantee serializability. Most systems implement **strict 2PL**. SQL 92 allows explicit **relaxation** of the ACID isolation constraints in the interest of performance.

Concurrency in B-trees

Specialized protocols exist for concurrency control in B-trees (the root would be a locking bottleneck otherwise).

Recovery (ARIES)

The ARIES technique aids to implement **durability** and **atomicity** by use of a **write-ahead log**.