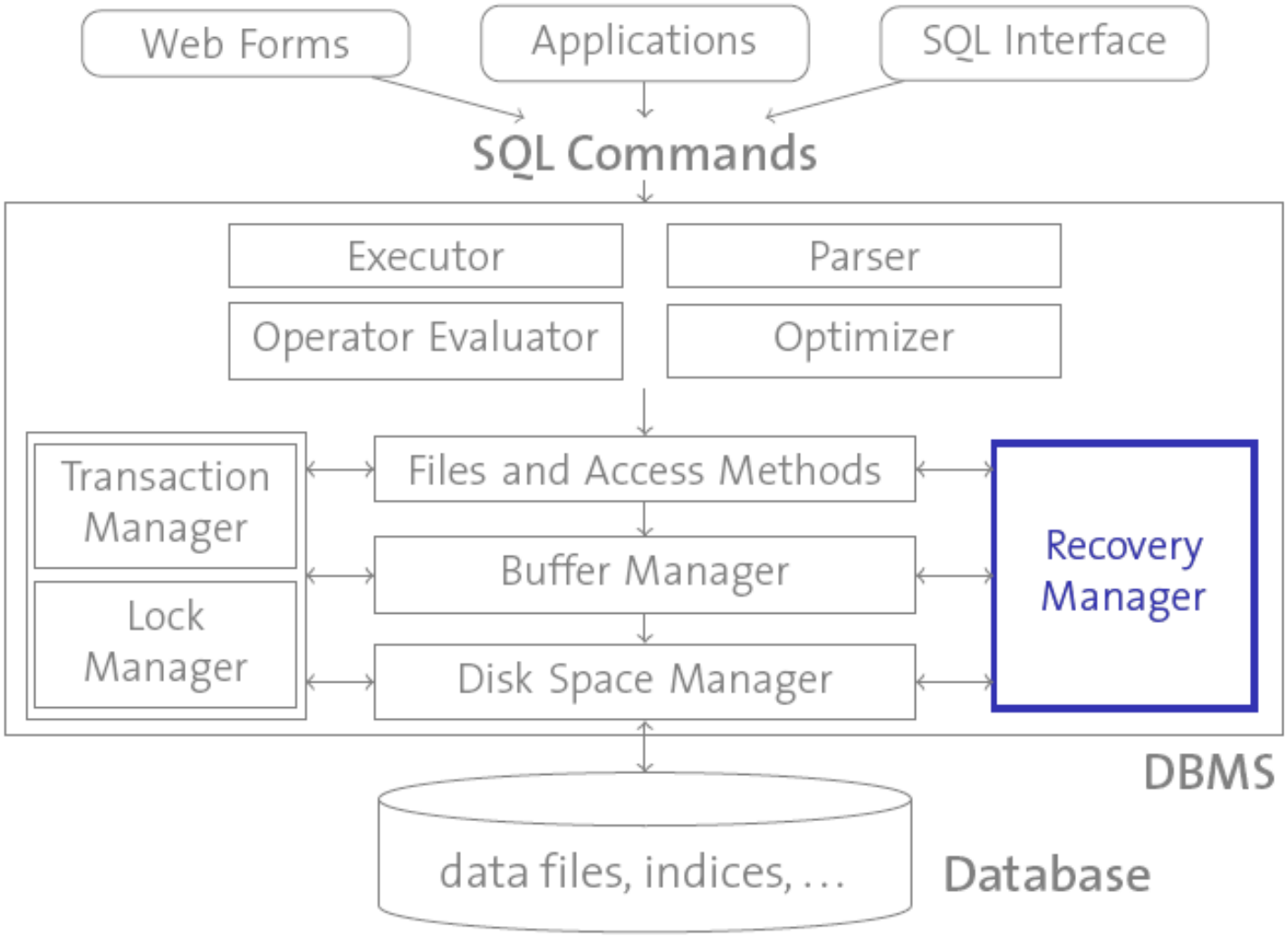
Information Systems

Computer Science Department
ETH Zurich
Spring 2012

Lecture VI: Transaction Management (Recovery Manager)

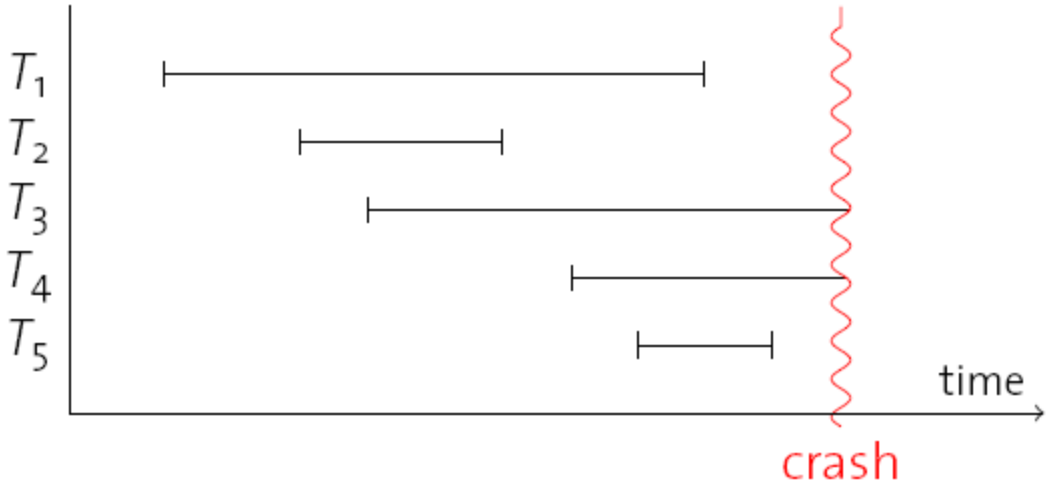
Recovery Manager



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



- 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

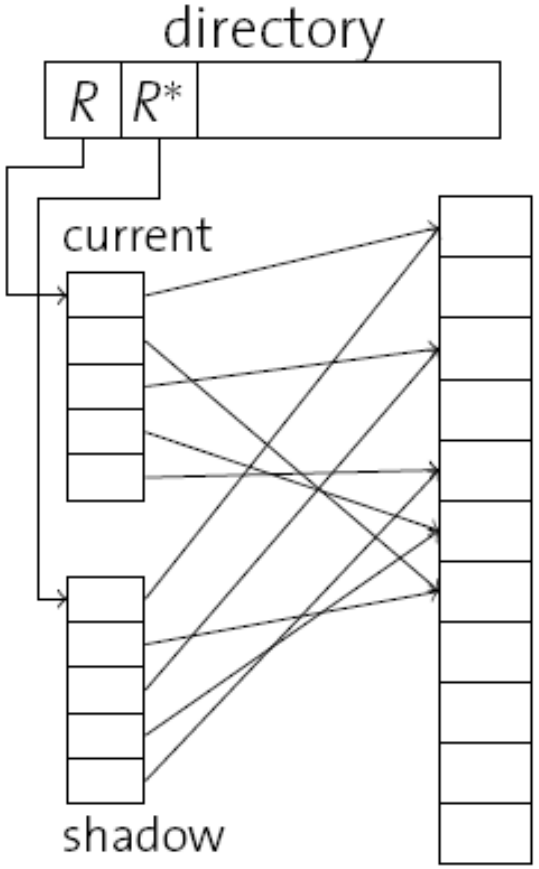
- We assume three different types of storage:
  - **volatile storage**
    - This is essentially the **buffer manager**. We are going to use volatile storage to **cache** the "**write-ahead log"** in a moment.
  - **non-volatile storage**
    - Typical candidate is a **hard disk**.
  - **stable storage**
    - 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 Paging

- Since a failure could occur at any time, it must be ensured that the system can always get back to a consistent state.
  - Need to keep redundant information.
- 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 (so that the recent changes become permanent).
  - SAVE $\leftrightarrow$ **commit**
- Use operation RESTORE to recover to shadow version (so that the recent changes can be undone).
  - RESTORE $\leftrightarrow$ **abort**

Shadow Paging

1. Initially: shadow ≡ current.
2. A transaction *T* now would like to change the current version.
   - Updates are not done in-place.
   - Create new pages and alter the current page table (page ids → disk addresses).
3. If *T* aborts, overwrite current version with shadow version (RESTORE).
4. If *T* commits, change information in directory to make current version persistent (SAVE).
5. Reclaim disk pages using garbage collection (i.e., free new ones in case of RESTORE and old ones in case of SAVE).



directory : file id -> page table
page table : page id -> disk address

Shadow Paging: 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 earlier, 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 Computing Surveys, 13(2), 1981.

Shadow Paging: 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.

Steal and No-force (Recap)

- Steal:
  - If a page frame is dirty and chosen for replacement, the page it contains is written to disk even if the modifying transaction is still active.
  - Improved throughput, but atomicity more difficult

- No-force:
  - Pages in the buffer pool that are modified by a transaction are not forced to disk when the transaction commits.
  - Improved response time, but durability more difficult

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.

Main Drawbacks of Shadow Paging

- Data becomes fragmented due to the replacement of pages by current versions which may be located far away from the shadow version ($\rightarrow$ the need for systematic garbage collection).
- Degree of concurrency is low ($\rightarrow$ can be improved by steal and no-force).
- Keeping two versions of pages leads to storage overhead.

ARIES Algorithm

- **A**lgorithm for **R**ecovery and **I**solation **E**xploiting **S**emantics
- A better alternative to shadow paging
- Works with steal and no-force
- Data pages are updated in place
- Uses “logging”
  - Log: An ordered list of REDO/UNDO actions.
  - Record REDO and UNDO information for every update.
    - Sequential writes to log (usually kept on separate disk(s)).
    - Minimal info written to log $\rightarrow$ multiple updates fit in a single log page.

Three Main Principles of ARIES

- Write-Ahead Logging
  - Record database changes in the log at stable storage before the actual change.
- Repeating History During Redo
  - After a crash, bring the system back to the exact state at crash time; undo the transactions that were still active at crash time.
- Logging Changes During Undo
  - Log the database changes during a transaction undo so that they are not repeated in case of repeated failures and restarts (i.e., never undo an undo action).

Write-Ahead Log (WAL)

- The **ARIES** recovery method uses a "write-ahead log" to implement the necessary redundancy.
  - Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", ACM TODS, 17(1), 1992.
- WAL: Any change to a database object is **first** recorded in the log, which must be written to stable storage **before** the change itself is written to disk.
  - 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).

Content of the Write-Ahead Log

- **LSN (Log Sequence Number)**
  - Monotonically increasing number to identify each log record.
  - Can be the address of the first byte of the log record.
- **Type (Log Record Type)**
  - Indicates whether this is an update record (UPD), end of transaction record (EOT), compensation log record (CLR), commit record, or abort record.
  - CLR is used when undoing the change in an update log record.
- **TX (Transaction ID)**
  - Transaction identifier.

Content of the Write-Ahead Log (cont'd)

- **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.

Example

Transaction 1	Transaction 2
$a \leftarrow \texttt{read}(A)$;	
	$c \leftarrow \texttt{read}(C)$;
$a \leftarrow a - 50$;	
	$c \leftarrow c + 10$;
write(a,A);	
	write(c,C);
$b \leftarrow \texttt{read}(B)$;	
$b \leftarrow b + 50$;	
write(b,B);	
commit;	
	$a \leftarrow \texttt{read}(A)$;
	$a \leftarrow a - 10$;
	write(a,A);
	commit;

LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
1	UPD	T_1	–	···		$A := A - 50$	$A := A + 50$
2	UPD	T_2	–	···		$C := C + 10$	$C := C - 10$
3	UPD	T_1	1	···		$B := B + 50$	$B := B - 50$
4	EOT	T_1	3	···			
5	UPD	T_2	2	···		$A := A - 10$	$A := A + 10$
6	EOT	T_2	5	···			

Redo Information

- ARIES assumes **page-oriented redo**:
  - No other pages need to be examined to redo a log entry.
  - E.g., **physical** logging:
    - store byte images for (parts of) a page
    - before image: byte image before update was performed
    - after image: byte image after update was performed
  - Makes recovery independent amongst objects
    - Recovery system does not need to know what type of page it deals with: data pages, index pages, ...
  - Contrast to **logical** redo ("set tuple with *rid* to value *v*"):
    - Redo requires accessing and changing indices, may cause tuple relocation to another page, etc.

Undo Information

- ARIES does support **logical undo**:
  - Page-oriented undo can cause cascading rollback situations.
    - Even if a transaction T_1 does not directly inspect tuples written by another transaction T_2, it may still see T_2's effects on, e.g., page layout.
    - T_1 could then not commit before T_2 commits.
  - Logical undo, therefore, increases concurrency.
  - By using operation logging, transactions can even run fully concurrent if their actions are semantically compatible.
    - E.g., increment and decrement operations.
    - This is particularly interesting in indices and meta data.

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).
- Committed transaction = a transaction all of whose log records, including a commit record have been written to stable storage.
- The log is an ever-growing file.

Normal Processing

- During normal transaction processing, keep two pieces of information in each transaction control block:
  - 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.

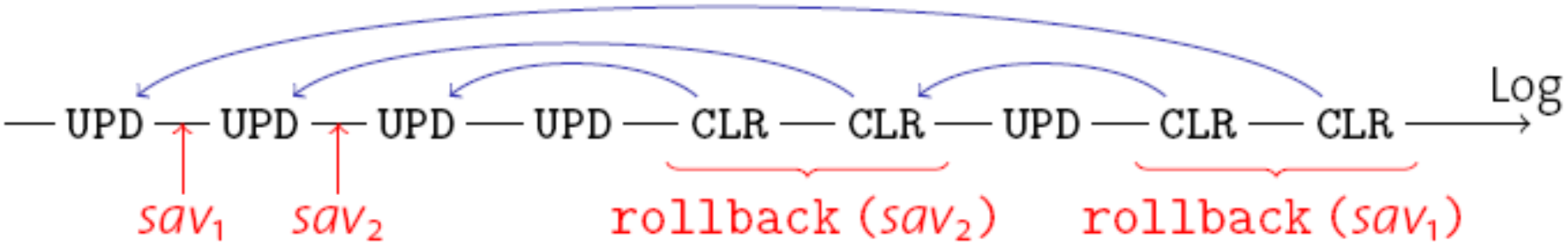
- Undo operations modify pages, too!
  - Log all undo operations to the WAL.
  - Use compensation log records (CLRs) for this purpose.
  - Note: We never undo an undo action, but we might need to redo an undo action.

Transaction Rollback

```
1  Function: rollback (SaveLSN, T)
2  UndoNxt ← T.UNxt ;
3  while SaveLSN < UndoNxt do
4      LogRec ← read log entry with LSN UndoNxt ;
5      switch LogRec.Type do
6          case UPD
7              perform undo operation LogRec.Undo on page LogRec.Page ;
8              LSN ← write log entry
                       ⟨CLR, T, T.LastLSN, LogRec.Page, LogRec.Prev, · · · , ∅⟩ ;
9              set LSN = LSN in page header of LogRec.Page ;
10             T.LastLSN ← LSN ;
11         case CLR
12             UndoNxt ← LogRec.UNxt ;
13     T.UNxt ← UndoNxt ;
```

Transaction Rollback

- Transaction can be rolled back partially (back to SaveLSN).
- The UNxt field in a CLR points to the log entry before the one that has been undone.



Summary

- Failure Recovery
  - Ensure Atomicity & Durability under different failure types, in presence of different storage types
  - Undo, Redo
- Shadow Paging
  - Force/No-force, Steal/No-steal
- ARIES
  - Write-Ahead Log
  - Transaction rollback

ARIES 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

```
1  Function: analyze ()
2  foreach log entry record LogRec do
3      switch LocRec.Type do
4          create transaction control block for LogRec.TX if necessary ;
5          case UPD or CLR
6              LogRec.TX.LastLSN ← LogRec.LSN ;
7              if LocRec.Type = UPD then
8                  LogRec.TX.UNxt ← LogRec.LSN ;
9              else
10                 LogRec.TX.UNxt ← LogRec.UNxt ;
11         case EOT
12             delete transaction control block for LogRec.TX ;
```

Redo Phase

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

- System crashes can occur **during** recovery!
  - Undo and redo of a transaction *T* must be **idempotent**:
    - undo(undo(*T*)) = undo(*T*)
    - redo(redo(*T*)) = redo(*T*)
- 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. All others would have to be undone anyway.
- We'll have to undo all effects of loser transactions afterwards.
- As an optimization, the analysis 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).

Undo Phase

```
1  Function: undo ()
2  while transactions (i.e., TCBs) left to roll back do
3      T ← TCB of loser transaction with greatest UNxt ;
4      LogRec ← read log entry with LSN T.UNxt ;
5      switch LogRec.Type do
6          case UPD
7              perform undo operation LogRec.Undo on page LogRec.Page ;
8              LSN ← write log entry
                       ⟨CLR, T, T.LastLSN, LogRec.Page, LogRec.Prev, · · · , ∅⟩ ;
9              set LSN = LSN in page header of LogRec.Page ;
10             T.LastLSN ← LSN ;
11         case CLR
12             UndoNxt ← LogRec.UNxt ;
13     T.UNxt ← UndoNxt ;
14     if T.UNxt = '–' then
15         write EOT log entry for T ;
16         delete TCB for T ;
```

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.

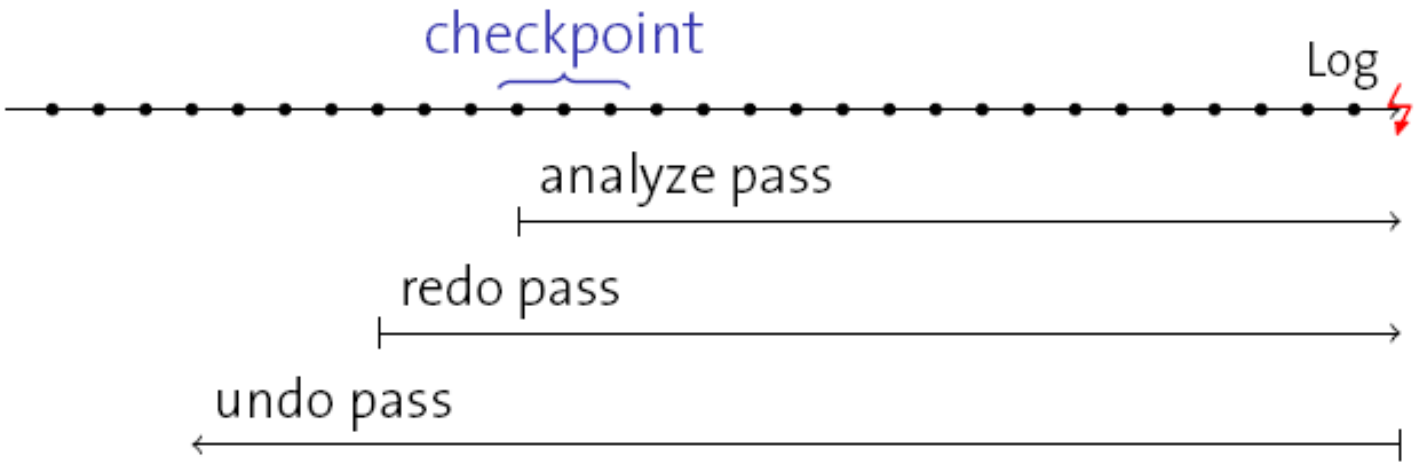
Fuzzy Checkpointing

- 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 Checkpointing

- During crash recovery
  - Start analyze pass at the most recent 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).

Wrap-up

- **Failure Recovery**
  - Goal: Guarantee Atomicity and Durability in case of failures.
- **Shadow Paging**
  - Can be with Force/No force and Steal/No steal
  - Has several drawbacks
- **ARIES Algorithm**
  - Uses a write-ahead log (WAL)
  - Commonly used by all major industrial implementations
- **Checkpointing**
  - Periodically taking database snapshots
  - Goal: to reduce recovery time during a system crash