Relational Database System Implementation

CS122 – Lecture 16

Winter Term, 2018-2019

Write-Ahead Logging

- Last time, introduced write-ahead logging (WAL) as a mechanism to provide atomic, durable transactions
- Log records:
  - $<T_i$: start>, $<T_i$: commit>, $<T_i$: abort>
  - Updates: $<T_i, X_j, V_1, V_2>$
    - Records that transaction T_i changed data-item X_j from V_1 to V_2
  - Redo-only updates: $<T_i, X_j, V>$
    - Records that data-item X_j was rolled back to value V
- Must always follow the write-ahead logging rule:
  - All database state changes must be recorded to the log on disk, before any table-files are changed on disk

Write-Ahead Logging (2)

- Recovery processing involves two phases
- Redo phase:
  - Replay all state changes recorded in the write-ahead log
  - Find the set of all incomplete transactions
- Undo phase:
  - Roll back all incomplete transactions, updating the log as with a normal transaction rollback
- Once recovery is completed:
  - All table files are in sync with the write-ahead log
  - All transactions are in a completed state

Recovery Processing

- The write-ahead log records every transaction, and every state-change performed against the database
- Recovery processing scans this entire log file…
  - Unless database is crashing all the time (unlikely), most transactions in log file won't actually require recovery!
- Introduce a *checkpoint* mechanism that allows us to pinpoint where to start recovery processing from
- Goal: As with recovery, checkpoint procedure should bring all table files into sync with the write-ahead log
  - Extent of write-ahead log required for recovery processing will be constrained by most recent checkpoint

Checkpoint Procedure

- Constraint: while checkpoint is being performed, no other update operations are allowed
  - Imposes a small but definite performance impact
- Checkpoint procedure (each step performed in order):
  - Output all log records currently in memory to disk, and sync the log file
  - Output all modified table-pages from memory to disk, and sync each table-file as well
  - Output a log record of the form <checkpoint *L*>, where *L* is the set of all active transactions at time of checkpoint

Checkpoint Records

- When we see a <checkpoint *L*> record in the log:
  - The table state on disk reflects <u>all</u> changes recorded in the write-ahead log on disk, up to that checkpoint record
- Once the checkpoint is performed, can the database ignore all log records before <checkpoint *L*> record?
  - NO: If one of the transactions in *L* was not completed before recovery, it must be aborted during recovery
  - Database must keep all logs up to earliest <T_i: start> record, over all transactions T_i that appear in the set *L*
  - Logs before this earliest <T_i: start> record may be discarded

Checkpoint Recovery

- Recovery processing changes slightly with new checkpoint mechanism
- Database must initially scan backward through logs, to find the last checkpoint record
- Perform redo phase first, as before
  - This time, the set of incomplete transactions is initialized to contain all *L* txns specified in <checkpoint: *L*> record
- Can ignore all *redo* operations before this checkpoint:
  - All logs were flushed, then all modified table-pages were flushed, before the <checkpoint: *L*> record was output
  - (May still need to undo operations before checkpoint…)

Checkpoint Recovery (2)

- Second, perform the undo phase:
  - At end of redo phase, the set of incomplete transactions will specify which transactions must be rolled back
  - Undo processing doesn't change at all
- Traverse transaction-log backwards, undoing state-changes of transactions in the incomplete set
  - For a transaction T_i in the incomplete set, remove it from the set when a <T_i: start> record is reached
- Don't forget:
  - Txns in set L from <checkpoint: L> record may also need to be rolled back; their records will be *before* checkpoint record
  - Just scan backward in logs until incomplete-set is empty

Checkpoint Performance

- During a checkpoint, no other updates are allowed
- Example:
  - A transaction is updating every record in a very large table
  - Lots of dirty buffer-pages in memory, logs being generated!
  - Changes aren't required to be flushed to disk until commit
- …and then a checkpoint occurs.
  - All in-memory write-ahead log records must be output
  - All dirty table-pages must be output
- During this time, all other write operations will be blocked until the checkpoint completes
  - Even small transactions that only need to update one value!
  - If a lot of data must be output, this will take some time

Checkpoint Performance (2)

- Can perform checkpoints based on how many logs (or dirty table-pages) have accumulated in memory
  - Impose an upper-bound on the time a checkpoint takes
- Can also implement *fuzzy checkpoints*
  - Allows transactions to perform updates even during the fuzzy checkpoint procedure
- Modify the checkpoint procedure slightly:
  - Still require that updates are blocked while WAL records are output to disk, and while <checkpoint> record is written
  - Allow modified table-pages to be written to disk *after* the <checkpoint> record has been written
  - After <checkpoint> is written, allow *some* updates to proceed

Fuzzy Checkpoints

- Fuzzy checkpoint procedure (performed in order):
  - Output all write-ahead log records currently in memory to disk, and sync the log file
  - Output a log record of the form <checkpoint *L*>, where *L* is the set of all active transactions at time of checkpoint
    - *(This used to be done <u>after</u> writing all table-files to disk…)*
  - Output all modified table-pages from memory to disk, and sync each table-file as well
- Checkpoint record isn't quite as strong now…
  - The <checkpoint *L*> record no longer guarantees that table-files are in sync with log records up to that point

Fuzzy Checkpoints (2)

- When a fuzzy checkpoint is performed:
  - The <checkpoint *L*> record no longer guarantees that table-files are in sync with log records up to that point
- Checkpoint is still incomplete at the time the record is written to the log!
- Database maintains an additional value on the disk:
  - A *last-checkpoint* value, recording the record-location of the last successfully completed checkpoint in the log file
  - (Also want to avoid having to scan backward through log just to find the most recent checkpoint record…)

Fuzzy Checkpoints (3)

- Database maintains a *last-checkpoint* value on disk
- When <checkpoint *L*> record is written, DB must collect all dirty table-pages at time of checkpoint
  - Iterate through this collection, writing each page to disk
  - When all table-pages have been successfully written, update *last-checkpoint* to point to most recent <checkpoint> record
- Database can choose the order that it writes dirty pages out to disk
  - e.g. write them in sequential order for very fast output

Fuzzy Checkpoints (4)

- The database must still follow write-ahead logging rule…
- Example:
  - During a fuzzy checkpoint, a dirty buffer-page *B* is collected to be output to disk…
  - …then, a transaction wants to write to the data in page *B*
- Cannot allow this write until after page *B* is written to disk
  - Otherwise, will violate semantics of the checkpoint procedure
- With fuzzy checkpoints, txns will still block on individual pages being output during the fuzzy-checkpoint process
  - Ideally this will be infrequent, and the delay will be short

Long-Running Transactions

- Long-running transactions still cause issues:
  - A checkpoint marks a point-in-time when table files and write-ahead log are in sync with each other, on disk
  - Don't need to redo anything before the checkpoint…
  - May still need to undo operations before the checkpoint, if a transaction extended well before that point
- As described, the write-ahead logging mechanism doesn't handle this situation very well
  - May still need to traverse log far into the past, just to undo operations before the checkpoint
  - *(In fact, undos may already be reflected in the table files!)*

ARIES Logging/Recovery

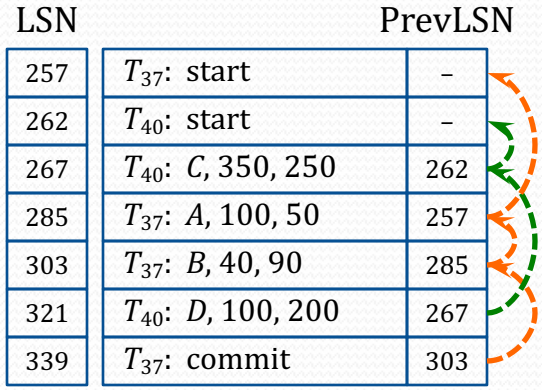
- ARIES: Algorithms for Recovery and Isolation Exploiting Semantics
  - A no-force, steal logging/recovery system
  - Designed at IBM in 1992
  - Used in IBM DB2, MS SQLServer, NTFS, and *many* others
- Has many transaction-processing features, such as:
  - Row-level locking instead of page-level locking
  - Fuzzy checkpoints that allow updates during checkpoint
  - Nested transactions and savepoints
  - Parallel recovery processing
- Will do a *very* basic overview of ARIES mechanisms
  - *(See the research papers for all the details!)*

Log Sequence Numbers

- Many features in ARIES depend on assigning each log record a unique *log sequence number* (LSN)
  - Can simply use the record's offset in the log file
- Can refer to individual log records using their LSNs
- Write-ahead logging can generate very large logs…
  - Place an upper bound on log file sizes
  - When current log file hits upper bound, begin another file
  - Assign each log file its own number
  - Fully specified LSN would be *logfile_number.offset*
  - Within a single log file, just use a record's offset for the LSN

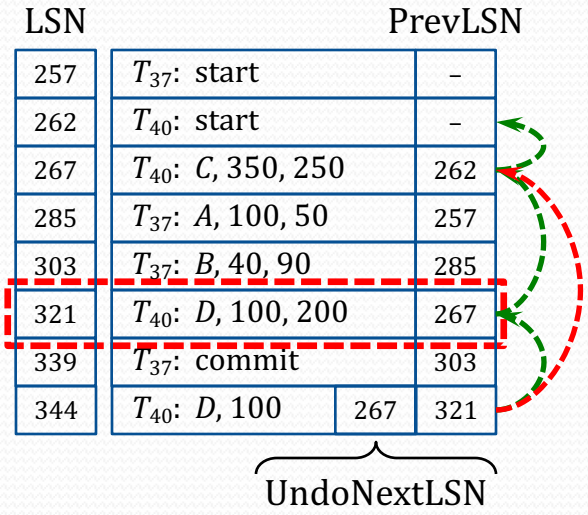
ARIES Log Records

- Example: multiple concurrent transactions
  - T_{37}: Transfer \$50 from account A to B (committed)
  - T_{40}: Transfer \$100 from account C to D (active)
- Each record has an (*implicit*) log sequence number
  - Not stored; can infer from the record's position in the file
- All log records contain a PrevLSN field, specifying the LSN for the previous record for that transaction
- Records store the LSN of previous log-record in the same transaction



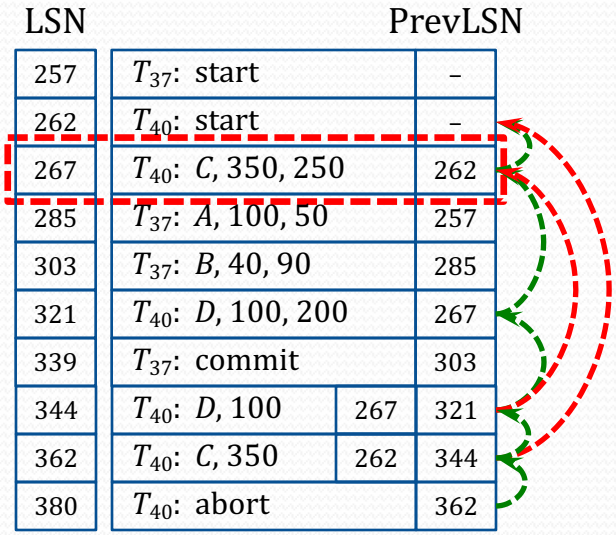
ARIES Log Records (2)

- PrevLSN field makes it *very* easy to trace through logs for a specific transaction
- Continuing example: Abort T_{40}, transfer from C to D
  - Need to roll back all state-changes in T_{40}
  - Start with last log-record recorded for T_{40}: LSN 321
    - DB also tracks the most recent LSN for every active transaction in a table
- As before, ARIES uses compensation log records (CLRs) during rollback
  - These records have an extra field UndoNextLSN, specifying the next operation to undo in the transaction



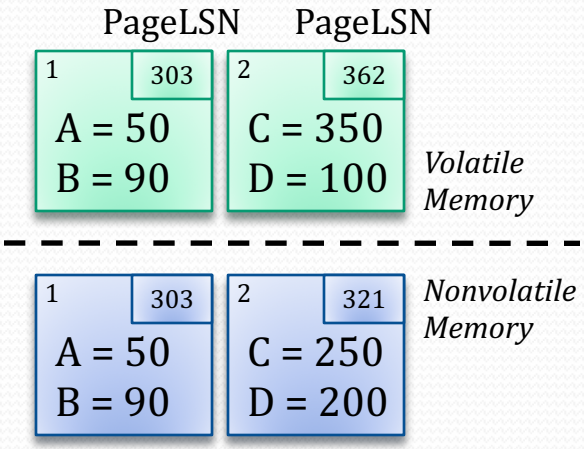
ARIES Log Records (3)

- Continuing example: Abort T_{40}, transfer from C to D
  - After performing compensating action for LSN 321, roll back LSN 267
- As before, continue process until reaching T_{40}'s start log-record
- Using PrevLSN and UndoNextLSN entries, ARIES can provide very fast rollback, even during periods of heavy concurrent usage



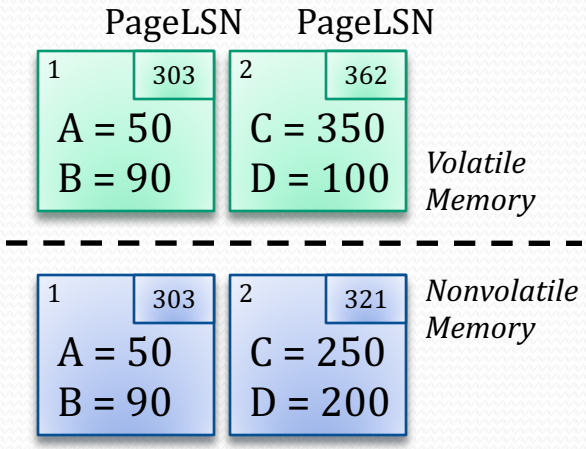
ARIES Data Pages

- Every data page is also given a PageLSN field
  - Specifies the most recent write-ahead log record that has been applied to the page
  - When pages are flushed to disk, PageLSN on disk is also updated (obvious)
- DB buffers data pages in memory…
  - PageLSN value in the page on disk may differ from value in memory
  - Simply indicates that data page on disk doesn't yet have all updates applied
  - Similarly, if PageLSN values match, we know the disk page is up to date



ARIES Data Pages (2)

- PageLSN field specifies the most recent write-ahead log record that has been applied to the page
- During recovery processing:
  - For a given page, only apply records with a LSN *larger than* the page's current PageLSN
  - Unnecessary to apply earlier records; data page on disk already reflects the earlier updates
- ARIES also has some update records that can only be applied once!
  - Applying to a page multiple times produces incorrect results
  - PageLSN value is *essential* to maintain idempotence of recovery processing



Dirty Page Table

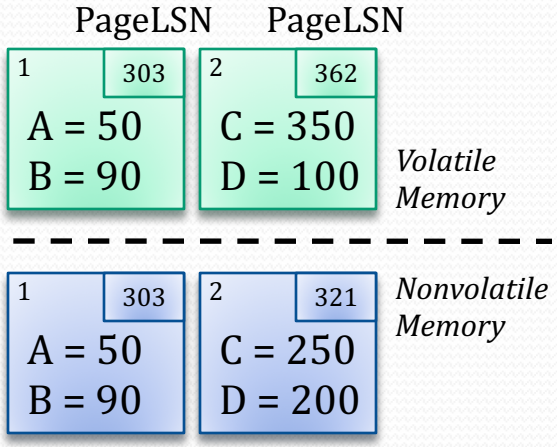
- ARIES also keeps a *dirty page table*, recording extra details about all dirty pages in the buffer manager
- Dirty page table entries hold three values:
  - The dirty page's ID (e.g. filename and block-number)
  - The current in-memory PageLSN value for the dirty page
  - Also, an additional *RecLSN* field:
    - Set to the "current LSN" at the time the page became dirty
- When a data page is flushed from buffer manager back to disk, its entry in the dirty page table is removed

Dirty Page Table (2)

- Continuing previous example…
- Dirty page table contains one entry
  - Data page 2 is dirty
- PageLSN value is easy. RecLSN is more challenging…
  - Page 2 in memory reflects state after aborting T_{40}
  - Page 2 on disk has two initial writes from T_{40}
  - Page 2 became dirty again from update recorded in LSN 344

Dirty Page Table:

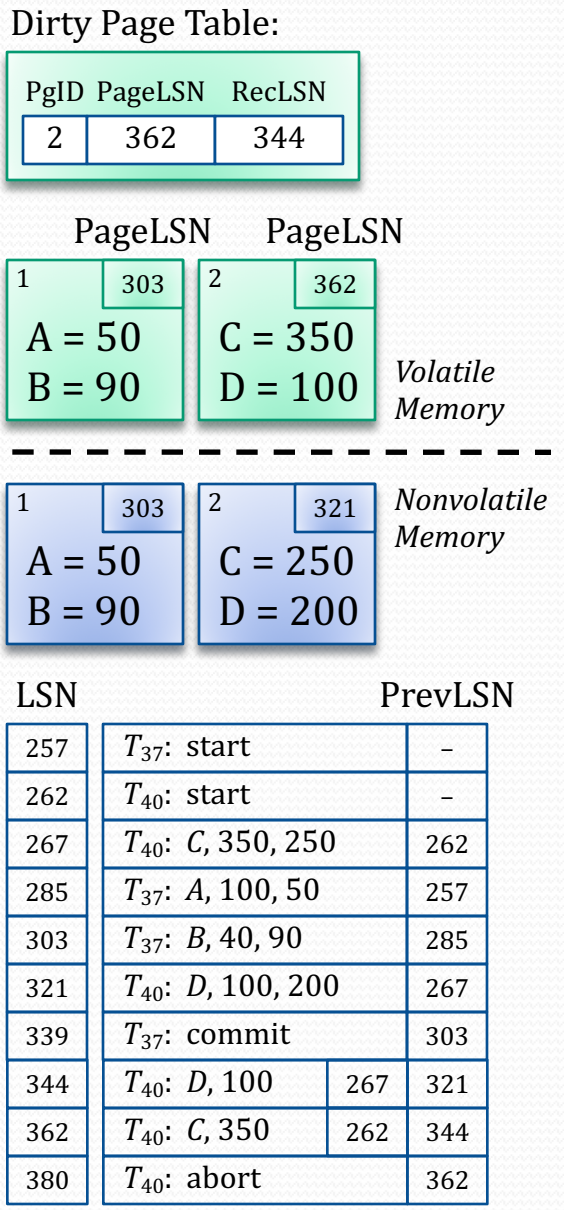
PgID	PageLSN	RecLSN
2	362	344



LSN			PrevLSN
257	T_{37}: start		–
262	T_{40}: start		–
267	T_{40}: C, 350, 250		262
285	T_{37}: A, 100, 50		257
303	T_{37}: B, 40, 90		285
321	T_{40}: D, 100, 200		267
339	T_{37}: commit		303
344	T_{40}: D, 100	267	321
362	T_{40}: C, 350	262	344
380	T_{40}: abort		362

Dirty Page Table (3)

- What can we discern from the RecLSN value recorded in the dirty page table?
- RecLSN tells us where to start in the log, to bring the disk-version of the data page into synchronization with the in-memory version of the page
- Very useful information to have during recovery processing…

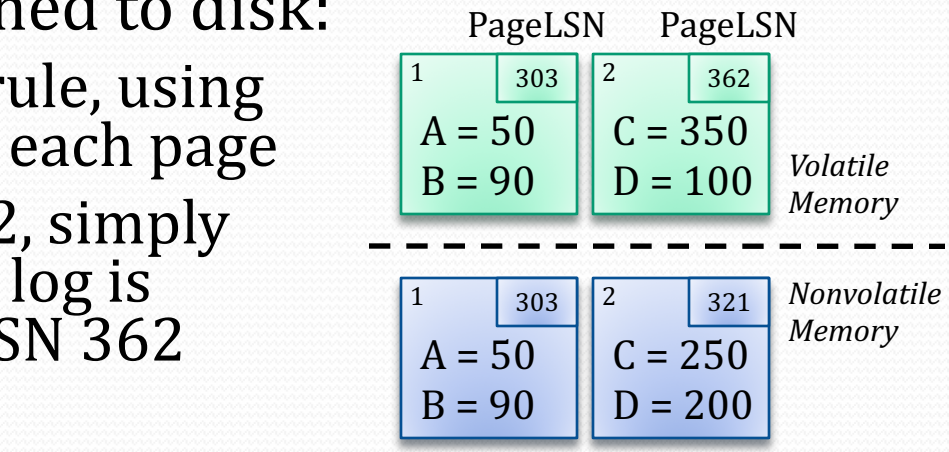


ARIES Checkpoints

- ARIES also supports fuzzy checkpoints
- An ARIES checkpoint record includes:
  - A table of all active transactions, including the ID of each transaction, and its LastLSN value:
    - The log sequence number of the last WAL record generated for that transaction
  - The dirty page table, specifying all dirty pages at time of checkpoint, along with their PageLSN and RecLSN values
- The fuzzy checkpoint procedure is similar to before:
  - Write and flush all in-memory log records to disk
  - Write and flush the checkpoint record to disk

ARIES Checkpoints (2)

- ARIES doesn't require flushing all dirty pages to disk as a part of the fuzzy checkpoint!
  - Makes checkpointing very unintrusive to other transaction-processing operations
- Rather, buffer manager writes out dirty pages on a regular basis, during normal execution
  - Can schedule dirty page writes to minimize disk seeks
- When a dirty page is flushed to disk:
  - Easy to follow the WAL rule, using PageLSN value stored in each page
  - e.g. when flushing page 2, simply ensure that write-ahead log is flushed to record with LSN 362



ARIES Recovery

- ARIES recovery-processing involves three phases:
  - Analysis phase, redo phase, undo phase
- Analysis phase performs three important tasks:
  - Determine the starting point (LSN) in the write-ahead log where redo processing must commence from
  - Determine the set of data pages that must be brought into sync with the write-ahead log state
  - Determine the set of incomplete transactions that must be rolled back
    - (In ARIES paper, these are called *loser transactions*)

Recovery: Analysis Phase

- Checkpoint record contains a copy of dirty page table
  - Use this table to determine initial value of *RedoLSN*, the LSN where the redo phase should start from
- e.g. given this dirty page table:
  - RedoLSN should be 345
- General rule:
  - Set RedoLSN to smallest value of RecLSN that appears in dirty page table
  - If checkpoint's dirty page table is empty, set RedoLSN to the LSN of the checkpoint record

Dirty Page Table:

PgID	PageLSN	RecLSN
14	403	389
3	521	484
9	505	456
11	398	345
6	584	577
7	522	490

Recovery: Analysis Phase (2)

- Next, analysis phase determines the set of dirty pages, and the set of incomplete transactions
- Initial dirty page table, and initial set of incomplete transactions are both taken from the checkpoint state
- Start scanning forward through write-ahead log, starting with the checkpoint record (not RedoLSN!)

Recovery: Analysis Phase (3)

- Scan forward from the checkpoint record:
  - If a <T_i: begin> record is seen, add T_i to set of incomplete transactions, with LastLSN set to LSN of record
    - Allows us to know where to start undoing T_i, if we have to
  - If a <T_i: commit> or <T_i: abort> record is seen, remove T_i from set of incomplete transactions
  - For T_i update records: <T_i: X_j, V_1, V_2> or <T_i: X_j, V>
    - Update transaction T_i's LastLSN to LSN of the update record
    - If X_j's data page is not in the dirty page table, add it to the table with RecLSN set to the LSN of the update record, and PageLSN taken from the data page on disk
- When this scan is complete, analysis phase is complete

Recovery: Redo Phase

- Database now has an up-to-date dirty page table
- From earlier:
  - ARIES uses each page's PageLSN value to ensure that each update is only applied to a page once
  - Required to make recovery processing idempotent, since some ARIES redo-operations may only be applied once
- Redo phase: repeat history!
- Recall: Found *RedoLSN* value during analysis phase (*RecLSN* in Dirty Page Table that is furthest in the past)

Dirty Page Table:

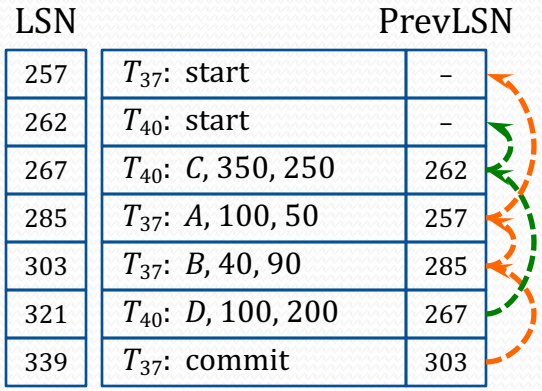
PgID	PageLSN	RecLSN
14	403	389
3	521	484
9	505	456
11	398	345
6	584	577
7	522	490

Recovery: Redo Phase (2)

- For each update record <T_i: X_j, V_1, V_2> or <T_i: X_j, V>:
  - If X_j's page is in dirty page table then apply the update
  - If not, we know update already hit the data page on disk; no need to apply the update twice!
- At end of redo phase, database will be in exactly the same state as when the system crash occurred
- The redo phase only loads data pages that need to be updated. No updates are applied multiple times!
  - Makes for a much faster recovery process than before
- After redo phase, there will be incomplete transactions
  - Roll these back in the undo phase

Recovery: Undo Phase

- Undo phase must roll back all incomplete transactions
- At end of analysis phase, had a table of all incomplete transactions, along with LastLSN of each transaction
- Can roll back these transactions in any order
  - Could roll each transaction back separately, or could roll all transactions back in a single pass
  - Each log record specifies the LSN of the previous operation in that txn, so rolling back these operations is easy
- As before, must write Compensation Log Records to WAL when undoing an update of the form <T_i: X_j, V_1, V_2> !



ARIES Logging/Recovery

- Checkpoints are often performed during recovery to guard against further crashes during recovery
  - Next recovery attempt will begin further down the line…
- ARIES is a very complex logging/recovery system…
  - Provides many txn-processing features and benefits
  - Checkpoints and recovery processing are both very fast
  - Definitely worth the added complexity!
  - Extensive use of ARIES demonstrates this very clearly