Databases 2
Lecture VI

Alessandro Artale

Faculty of Computer Science – Free University of Bolzano

Room: 221

artale@inf.unibz.it

http://www.inf.unibz.it/∼artale/

2003/2004 – First Semester

Summary of Lecture VI

- **Notion of Transaction**
- Failure Recovery
  - Failure Classification
- Logging Techniques for System Failure Recovery
  1. Undo Logging
  2. Redo Logging
  3. Undo/Redo Logging
- Recovery from Media Failure: Archiving.

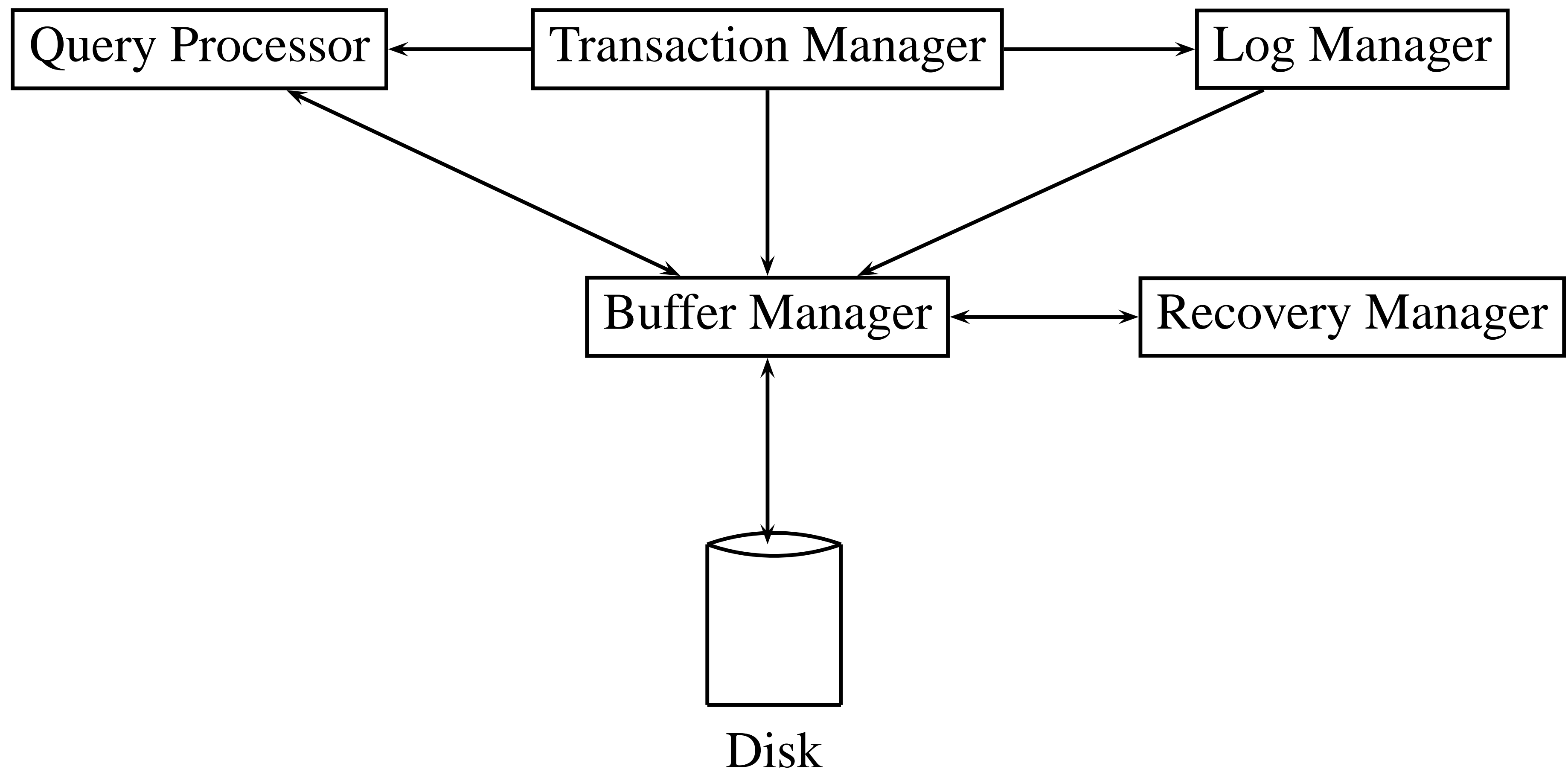
Recovery and Data Integrity

There are two fundamental issues in the way a DBMS controls access to data.

1. Data must be **recovered** from a system failure.
2. Data **integrity** must be preserved as a consequence of a concurrent access (i.e., the DBMS should avoid the introduction of inconsistent data like booking the same seat to two different users).

Transactions

- A **Transaction** is a collection of DBMS operations that form a single logical unit of execution—e.g., a query or a modification statement is a transaction.
- The *Transaction Manager* assures that Transactions are executed correctly.



Transactions: Basic Notions

- A DB is composed by **elements**: Unit of data accessed by transactions (e.g., tuples, disk blocks, etc.).
- A DB has a **state**: Values of all DB elements.
- A state is **consistent** if it meets all the DB constraints – e.g., a key identifies exactly one tuple.

Correct Transactions

- **Correctness Principle.**

  Atomicity. A transaction is *atomic*. If only part of its operations execute an error is issued since the DBMS will (possibly) be in an inconsistent state.

  Consistency. A transaction, if completed will take the DB from a *consistent* state to another consistent state—transactions preserve data integrity.

  Isolation. Even though many transactions may execute concurrently the DBMS guarantees that the computation is equivalent to a sequential (i.e., not concurrent) processing. Thus, each transaction runs in *isolation*.

  Durability. After a transaction completes successfully, the changes it has made to the database persist.

These properties are called **ACID** properties – acronym from the first letters of each of the above properties.

Address Spaces

There are three spaces that are of importance when dealing with transactions.

1. The *Disk space* storing he DB elements;
2. The *Main Memory space* managed by the Buffer Manager;
3. The *Local Address space* of the Transaction.

Primitive DB Operations of Transactions

Basic operations that move data between the different address spaces.

1. `INPUT(X)`. Copy the disk block containing database element X to a memory buffer.
2. `READ(X, t)`. If the block containing database element X is not in a memory buffer then `INPUT(X)`. Next, assign the value of X to the transaction's local variable t.
3. `WRITE(X, t)`. If the block containing database element X is not in a memory buffer then `INPUT(X)`. Next, copy the value of t to X in the buffer.
4. `OUTPUT(X)`. Copy the buffer containing X to disk.

- `READ` and `WRITE` are issued by transactions;
- `INPUT` and `OUTPUT` are issued by the buffer manager.

Transaction Example

- A, B are database elements with the constraint $A = B$.
- Transaction $\texttt{T} : \texttt{A} := \texttt{A} * 2; \quad \texttt{B} := \texttt{B} * 2.$
- Execution of T involves reading A, B from disk, performing the arithmetic operations in main memory, and writing new values for A, B back to disk.

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

System Failure: An Example

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

- If a system failure happens before `OUTPUT(A)` is executed there is no effect to the DB: It is as if `T` never ran.

Problem: **If a system failure happens just after `OUTPUT(A)` but before `OUTPUT(B)`. The DB is left in an inconsistent state where $A \neq B$.**

Summary

- Notion of Transaction
- **Failure Recovery**
  - **Failure Classification**
- Logging Techniques for Failure System Recovery
  1. Undo Logging
  2. Redo Logging
  3. Undo/Redo Logging
- Recovery from Media Failure: Archiving.

Failure Classification

- **Erroneous data entry (mistyping)**
  - Database Constraints: key constraints, value constraints that associate a type to each attribute, etc. Triggers can check that constraints are satisfied.
- **Media failures**
  - Local failure of a disk that changes few bits: parity checks on sectors.
  - Disk crash: handled by RAID (Redundant Array of Independent disks) techniques that automatically "mirror" data on separate disks.
  - Catastrophic Events: Maintain an external archive stored at a safe distance from the DB itself.
- **System failures**
  - Power loss and Software errors: The content of main memory is lost while DB *Transactions* already made some modifications to the DB. *Solution*: Maintain a *Log File* that records the changes—*Logging Techniques*.

Logging Techniques

- A **Log** is a file containing a sequence of **Log Records** describing what transactions have done.
- In case of a system failure the log is consulted to re-create a consistent state of the database.
- The Log can also be used together with an archive in case of a Media failure.
- Two possibilities to recover from system failure:
  1. **Un-Do** the work done by some transactions: It appears like they never executed.
  2. **Re-Do** the work done by some transactions: The new values they wrote to the DB are written again.

Summary

- Notion of Transaction
- Failure Recovery
  - Failure Classification
- **Logging Techniques for System Failure Recovery**
  1. **Undo Logging**
  2. Redo Logging
  3. Undo/Redo Logging
- Recovery from Media Failure: Archiving.

Undo Logging

- The **Undo Logging** technique restores a DB state by **undoing** any database change made by pending transactions.
- The log manager records in the log file each important event of a transaction execution.
- There are various types of log records:
  1. <`START T`>. Transaction `T` started.
  2. <`COMMIT T`>. Transaction `T` has completed, and all its changes have been output to the disk.
  3. <`ABORT T`>. Transaction `T` could not complete successfully, it does not write any of its DB modifications to disk.
  4. <`T`, `X`, `v`> – called *update record*. Database element `X` has been modified by Transaction `T`; it former value was `v`.

The Undo-Logging Rules

- **Main Idea**. If there is a crash before a transaction commits, the log will tell us how to restore old values for any DB elements changed on disk by the transaction.
- The Undo-Logging Rules control the data flow from Main Memory to Disk.
- **Undo-Logging Rules:**
  1. Log records $< \mathtt{T}, \mathtt{X}, \mathtt{v} >$ for DB element X must be written to disk—i.e., the log file should be updated on disk—*before* the new value for X is written to disk.
  2. Before writing the commit record on the log file on disk, all DB modifications made by the transaction must appear on disk.

The Undo-Logging Rules (cont.)

Given a transaction then under the undo-logging policy illustrated before the following disk writing occur in this order:

1. The log update records indicating changed elements;
2. The changed elements themselves;
3. The `COMMIT` log record.

Transaction Example

Action	t	Mem A	Mem B	Disk A	Disk B	Log
READ(A,t)	8	8		8	8	<START T>
`t := t*2`	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)	8	16	8	8	8	
`t := t*2`	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	<COMMIT T>
FLUSH LOG						

Note: `FLUSH LOG` is an action that writes any log record currently in Main Memory to disk – this command updates the Log file.

Recovery With Undo Logging

In case of a system failure such that only part of the changes made by a transaction have been written to disk, then the *recovery manager* uses the log file (stored on Disk) to restore the DB – in our example a system failure occurs just before OUTPUT(B).

- **To recover a crash using a Undo-Logging do:**
  1. Examine the log to identify all transactions T such that $<\mathtt{START\ T}>$ appears in the log, but neither $<\mathtt{COMMIT\ T}>$ nor $<\mathtt{ABORT\ T}>$ does. In this case, T is an **incomplete transaction** and must be **undone**.
  2. Examine each log entry $<\mathtt{T}, \mathtt{X}, \mathtt{v}>$ from most recent to earliest log records and:
     (a) If T is not an incomplete transaction, do nothing.
     (b) If T is incomplete, do: WRITE(X, v); OUTPUT(X).
  3. For each incomplete transaction T add $<\mathtt{ABORT\ T}>$ to the log, and flush the log.

System Failure Example

Action	t	Mem A	Mem B	Disk A	Disk B	Log
READ(A,t)	8	8		8	8	<START T>
`t := t*2`	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)	8	16	8	8	8	
`t := t*2`	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	

- Our hypothetical crash before OUTPUT(B) would result in T being identified as incomplete and the DB in an inconsistent state.
  - We would find in the log both <T, B, 8> and <T, A, 8>, and write both A = 8; B = 8 to the DB: The DB is now in a consistent state.
- **Problem:** What happens if there is a system error during recovery?

Checkpointing

- **Problem:** In principle recovery requires looking at entire log.
- Log cannot be truncated after a transaction commits since many inter-living transactions may execute, and log records pertaining to other active transactions might be lost.
- **Simple solution: occasional *checkpoint* operation:**
  1. Stop accepting new transactions;
  2. Wait until all current transactions commit or abort;
  3. Flush the log to disk;
  4. Enter a <CKPT> record in the log, and flush again the log to disk;
  5. Resume accepting transactions.
- **If recovery is necessary, we know that all transactions prior to a recorded checkpoint have committed and they do not need be undone.**
- **The log before a <CKPT> record can deleted safely.**

Nonquiescent Checkpointing

- **Problem:** we may not want to stop transactions from entering the system, since this means the system must shut down.
- **The steps in a *nonquiescent checkpointing* are:**
  1. Write $<\texttt{START CKPT}(\texttt{T}_1;\ldots;\texttt{T}_\texttt{k})>$ record to log, where T_i are all the active (uncommitted) transactions;
  2. Wait until all transactions $T_1;\ldots;T_k$ commit or abort, *but do not prohibit new transactions*;
  3. Write $<\texttt{END CKPT}>$ record to log, and flush the log.

Recovery With Nonquiescent Checkpoints

Rules to recover a crash using Undo Nonquiescent Checkpoints.

Scanning the log from the end do:

- If we first meet $<$ $\texttt{END CKPT}$ $>$ then we can restrict to transactions that began after the $<$ $\texttt{START CKPT}(\texttt{T}_1;\ldots;\texttt{T}_\texttt{k})$ $>$. The log before $<\texttt{START CKPT}(\texttt{T}_1;\ldots;\texttt{T}_\texttt{k})>$ is useless and can be deleted.

- If we first meet $<\texttt{START CKPT}(\texttt{T}_1,\ldots,\texttt{T}_\texttt{k})>$, then the crash occurred during the checkpoint. We need to undo:

  1. All those transactions $\texttt{T}$ with $<\texttt{START T}>$ after the $<\texttt{START CKPT}>$ but no $<\texttt{COMMIT T}>$;

  2. All transactions $\texttt{T}_\texttt{i}$ on the list associated with $<\texttt{START CKPT}>$ with no $<\texttt{COMMIT T}_\texttt{i}>$. The log before the start of the earliest of these incomplete transactions can be deleted.

Nonquiescent Checkpoints: An Example

Suppose we start with the following log:

$$< \mathtt{START}\ \mathtt{T_1} >$$
$$< \mathtt{T_1}, \mathtt{A}, \mathtt{5} >$$
$$< \mathtt{START}\ \mathtt{T_2} >$$
$$< \mathtt{T_2}, \mathtt{B}, \mathtt{10} >$$

Now we decide to do a Nonquiescent Checkpoint:

$$< \mathtt{START}\ \mathtt{T_1} >$$
$$< \mathtt{T_1}, \mathtt{A}, \mathtt{5} >$$
$$< \mathtt{START}\ \mathtt{T_2} >$$
$$< \mathtt{T_2}, \mathtt{B}, \mathtt{10} >$$
$$< \mathtt{START}\ \mathtt{CKPT}(\mathtt{T_1}; \mathtt{T_2}) >$$

Nonquiescent Checkpoints: An Example (cont.)

Suppose another transaction, T_3 starts while waiting for T_1 and T_2 to complete. After a crash we have the following log:

$< \mathtt{START}\ T_1 >$

$< T_1, A, 5 >$

$< \mathtt{START}\ T_2 >$

$< T_2, B, 10 >$

$< \mathtt{START\ CKPT}(T_1, T_2) >$

$< T_2, C, 15 >$

$< \mathtt{START}\ T_3 >$

$< T_1, D, 20 >$

$< \mathtt{COMMIT}\ T_1 >$

$< T_3, E, 25 >$

$< \mathtt{COMMIT}\ T_2 >$

$< \mathtt{END\ CKPT} >$

$< T_3, F, 30 >$

T_3 is the only uncompleted transaction that has to be undone:

$$\mathtt{WRITE}(F, 30); \mathtt{WRITE}(E, 25); \mathtt{OUTPUT}(F), \mathtt{OUTPUT}(E); \mathtt{ABORT}(T_3).$$

The log before $< \mathtt{START\ CKPT}(T_1; T_2) >$ can be deleted.

Nonquiescent Checkpoints: An Example (cont.)

Suppose the crash happens during the Checkpoint and the log is:

$< \mathtt{START\ T_1} >$

$< \mathtt{T_1, A, 5} >$

$< \mathtt{START\ T_2} >$

$< \mathtt{T_2, B, 10} >$

$< \mathtt{START\ CKPT(T_1, T_2)} >$

$< \mathtt{T_2, C, 15} >$

$< \mathtt{START\ T_3} >$

$< \mathtt{T_1, D, 20} >$

$< \mathtt{COMMIT\ T_1} >$

$< \mathtt{T_3, E, 25} >$

- Scanning backwards, we identify $\mathtt{T_2}$ and $\mathtt{T_3}$ as incomplete transactions;
- When we reach $< \mathtt{START\ CKPT(T_1, T_2)} >$ we know that the only other possible incomplete transaction is $\mathtt{T_1}$, but $\mathtt{T_1}$ committed;
- When we reach $< \mathtt{START\ T_2} >$ we undo both $\mathtt{T_2}$ and $\mathtt{T_3}$:
  $\mathtt{WRITE(E, 25); WRITE(C, 15); WRITE(B, 10);}$
  $\mathtt{OUTPUT(E); OUTPUT(C); OUTPUT(B); ABORT(T_2); ABORT(T_3)}$.
  The DB is now in a consistent state, and the log before $< \mathtt{START\ T_2} >$ can be deleted.

Summary

- Notion of Transaction
- Failure Recovery
  - Failure Classification
- **Logging Techniques for System Failure Recovery**
  1. Undo Logging
  2. **Redo Logging**
  3. Undo/Redo Logging
- Recovery from Media Failure: Archiving.

The Undo-Logging Rules

- **Main Idea**. If there is a crash before a transaction commits, the log will tell us how to restore old values for any DB elements changed on disk by the transaction.

1. Log records $< \mathtt{T}, \mathtt{X}, \mathtt{v} >$ for DB element `X` must be written to disk—i.e., the log file should be updated on disk—*before* the new value for `X` is written to disk.
2. Before writing the commit record on the log file on disk, all DB modifications made by the transaction must appear on disk.

Given a transaction then under the undo-logging policy the following disk writing occur in this order:

1. The log update records indicating changed elements;
2. The changed elements themselves;
3. The `COMMIT` log record.

Problem of the Undo Logging

- **Problem.** Undo Logging forbids to commit a transaction without first writing all its changed data to disk.
- There is an increase in the number of Disk I/O due to the *immediate backup* of DB elements required by the Undo Logging technique. It's desirable to output multiple log records at once.
- We can save disk I/O if DB changes reside in main memory for a while.

Redo-Logging Vs. Undo-Logging

1. While *Undo-Logging* cancels the effect of incomplete transactions and ignores committed ones, *Redo-Logging* ignores incomplete transactions and repeats the changes made by committed transactions.

2. While *Undo-Logging* writes changed DB elements to the Disk before Commit, *Redo-Logging* Commit before writing data to disk.

3. *Redo-log* record entries contain new values (instead of old values): $<\mathtt{T}, \mathtt{X}, \mathtt{v}>$ = "transaction T modified X and the *new* value is v".

Redo-Logging Rules

1. Before modifying the DB element X on disk, the transaction `T` must be committed, and the <`COMMIT T`> record written to the log in main memory.

2. Before modifying DB element X on disk, write to disk all the log records involving X, including both the update records <`T`, `X`, `v`> and the <`COMMIT T`> records.

Given a transaction then under the redo-logging policy the following disk writing occur in this order:

1. The log update records indicating changed elements;

2. The `COMMIT` log record.

3. The changed elements themselves;

Undo Vs. Redo Logging Rules

Disk writing sequence:

Undo		Redo	
1.	LOG	1.	LOG
2.	**X**	2.	**COMMIT**
3.	**COMMIT**	3.	**X**

Transactions under the Redo-Logging Policy: An Example

Action	t	Mem A	Mem B	Disk A	Disk B	Log
READ(A,t)	8	8		8	8	< START T >
t := t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	< T, A, 16 >
READ(B,t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	< T, B, 16 >
						< COMMIT T >
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Note 1. The < COMMIT T > comes now **before** the DB elements get updated.

Note 2. OUTPUT(A) and OUTPUT(B) can be delayed until necessary (waiting for other DB elements to be written to the Disk).

Recovery for Redo Logging

- While incomplete transactions did not change the DB, committed transactions cause problems since we don't know which of their changes have been written to disk.
- **To recover a crash using a Redo-Logging we do:**
  1. Find the set of committed transactions from the log;
  2. Scan the log *forward* from the beginning and for each $<\texttt{T}, \texttt{X}, \texttt{v}>$ do:
     - If `T` is committed, write value v for X to disk, i.e., do:
       `WRITE(X, v); OUTPUT(X)`;
     - Otherwise, do nothing.
  3. For each incomplete transaction `T` add $<\texttt{ABORT T}>$ to the log, and flush the log.

System Failure: An Example

Action	t	Mem A	Mem B	Disk A	Disk B	Log
READ(A,t)	8	8		8	8	<START T>
t := t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

If a system failure happens after <COMMIT T> then T is recognized as a committed transaction and will be redone:

WRITE(A, 16); WRITE(B, 16); OUTPUT(A); OUTPUT(B).

Nonquiescent Checkpointing a Redo Log

- **Key Action 1:** Since changes by committed transactions can be written to disk much later than the time the transactions commit, then we need to write to disk all those changes before ending the checkpoint.
- **Key Action 2:** Checkpoint can now end **without** waiting for active transactions to either commit or abort.
- **Steps in a *nonquiescent checkpointing* under a redo-logging policy:**
  1. Write $<\mathtt{START\ CKPT}(\mathtt{T_1};\ldots;\mathtt{T_k})>$ record to log, where T_i are all the active (uncommitted) transactions;
  2. **Write to disk DB elements written to buffers, but not yet to disk, by transactions that committed before `START CKPT`;**
  3. Write $<\mathtt{END\ CKPT}>$ record to log, and flush the log.

Nonquiescent Checkpointing a Redo Log: A Log Example

$<\mathtt{START\ T}_1>$

$<\mathtt{T}_1, \mathtt{A}, 5>$

$<\mathtt{START\ T}_2>$

$<\mathtt{COMMIT\ T}_1>$

$<\mathtt{T}_2, \mathtt{B}, 10>$

$<\mathtt{START\ CKPT}(\mathtt{T}_2)>$

$<\mathtt{T}_2, \mathtt{C}, 15>$

$<\mathtt{START\ T}_3>$

$<\mathtt{T}_3, \mathtt{D}, 20>$

$<\mathtt{END\ CKPT}>$

$<\mathtt{COMMIT\ T}_2>$

$<\mathtt{COMMIT\ T}_3>$

- When we start checkpointing:
  - Only $\mathtt{T}_2$ is active;
  - Only $\mathtt{T}_1$ committed.

  Then, before writing $<\mathtt{END\ CKPT}>$ into the log we must be sure that the new value for the element A (changed by the committed T_1) has been written to disk.

- Note that $<\mathtt{END\ CKPT}>$ is before $<\mathtt{COMMIT\ T}_2>$.

Recovery with a Nonquiescent Checkpointed Redo Log

The strategy depends on whether the last checkpoint record is START or END:

- **If we first meet** $< \texttt{END CKPT} >$, then we can restrict to transactions that committed after the $< \texttt{START CKPT}(\texttt{T}_1; \ldots; \texttt{T}_k) >$ (since transactions that committed before have their value written to disk).
  - We need to redo all the committed transactions that are either among the T_i or started after the START CKPT with the redo-logging strategy. Log previous to the earliest of the $< \texttt{START T}_\texttt{i} >$ is useless and can be deleted.
- **If we first meet** $< \texttt{START CKPT}(\texttt{T}_1, \ldots, \texttt{T}_k) >$, then the crash occurred during the checkpoint. *Committed transactions before this start could have not written their changes to the disk.*
  - Search back to the previous $< \texttt{END CKPT} >$ together with its matching $< \texttt{START CKPT}(\texttt{S}_1, \ldots, \texttt{S}_\texttt{m}) >$, and redo all the committed transactions that are either among the S_i or started after that $< \texttt{START CKPT}(\texttt{S}_1, \ldots, \texttt{S}_\texttt{m}) >$ record;
  - The log before the earliest of the $< \texttt{START S}_\texttt{i} >$ and can be deleted.

Recovery with a Nonquiescent Checkpointed Redo Log: An Example

Suppose there is a crash and the log is:

$<\mathtt{START}\ \mathtt{T}_1>$

$<\mathtt{T}_1, \mathtt{A}, 5>$

$<\mathtt{START}\ \mathtt{T}_2>$

$<\mathtt{COMMIT}\ \mathtt{T}_1>$

$<\mathtt{T}_2, \mathtt{B}, 10>$

$<\mathtt{START}\ \mathtt{CKPT}(\mathtt{T}_2)>$

$<\mathtt{T}_2, \mathtt{C}, 15>$

$<\mathtt{START}\ \mathtt{T}_3>$

$<\mathtt{T}_3, \mathtt{D}, 20>$

$<\mathtt{END}\ \mathtt{CKPT}>$

$<\mathtt{COMMIT}\ \mathtt{T}_2>$

$<\mathtt{COMMIT}\ \mathtt{T}_3>$

Since we meet $<\mathtt{END}\ \mathtt{CKPT}>$ we need to check only $\mathtt{T}_2$ (present in the checkpoint list) and $\mathtt{T}_3$ (started after checkpoint). Since both committed they are redone:

$\mathtt{WRITE}(\mathtt{B}, 10); \mathtt{WRITE}(\mathtt{C}, 15); \mathtt{WRITE}(\mathtt{D}, 20); \mathtt{OUTPUT}(\mathtt{B}); \mathtt{OUTPUT}(\mathtt{C}); \mathtt{OUTPUT}(\mathtt{D})$

The log before $<\mathtt{START}\ \mathtt{T}_2>$ can be deleted.

Recovery with a Nonquiescent Checkpointed Redo Log: An Example (Cont.)

Suppose there is a crash and the log is:

$$< \texttt{START } T_1 >$$
$$< T_1, A, 5 >$$
$$< \texttt{START } T_2 >$$
$$< \texttt{COMMIT } T_1 >$$
$$< T_2, B, 10 >$$
$$< \texttt{START CKPT}(T_2) >$$
$$< T_2, C, 15 >$$
$$< \texttt{START } T_3 >$$
$$< T_3, D, 20 >$$

Since we meet $< \texttt{START CKPT}(T_2) >$ and there is no previous $< \texttt{START CKPT} >$ a normal Redo policy is applied. T_1 is the only committed transaction:

$$\texttt{WRITE}(A, 5); \texttt{OUTPUT}(A); \texttt{ABORT}(T_2); \texttt{ABORT}(T_3)$$

Summary

- Notion of Transaction
- Failure Recovery
  - Failure Classification
- **Logging Techniques for System Failure Recovery**
  1. Undo Logging
  2. Redo Logging
  3. **Undo/Redo Logging**
- Recovery from Media Failure: Archiving.

Undo/Redo Logging

Problems.

- Undo Logging forbids to commit a transaction without first writing all its changed data to disk, perhaps increasing the number of disk I/O required.
- Redo Logging requires keeping all modified blocks buffered until commit, perhaps increasing the number of buffers required.

The Undo/Redo Logging technique provides an increased flexibility to order the actions at the expense of more information in the log.

The Undo/Redo Logging Rule

- The *update record* has a new form: $< \mathtt{T}, \mathtt{X}, \mathtt{v}, \mathtt{w} >$, i.e., transaction T updated DB element X from old value v to new value w.

The rule for the Undo/Redo Logging is:

1. Before writing a DB element X to the Disk because of changes made by T_i every update record $< \mathtt{T_i}, \mathtt{X}, \mathtt{v}, \mathtt{w} >$ must appear on Disk.

The Undo/Redo Logging Rule (cont.)

- The Undo/Redo rule enforces ONLY the **common** constraints between Undo and Redo techniques.
- In particular, there is no constraint about whether DB elements are written to disk before or after the <`COMMIT`> point.

Undo	Redo	Undo/Redo
1. LOG	1. LOG	1. LOG
2. X	2. COMMIT	2. X
3. COMMIT	3. X	

Transactions under the Undo/Redo Logging Policy: An Example

Action	t	Mem A	Mem B	Disk A	Disk B	Log
READ(A,t)	8	8		8	8	<START T>
t := t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8, 16>
READ(B,t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Recovery with Undo/Redo Logging

- In the log we have the information either to *Undo* a transaction (by restoring the *old* values), or to *Redo* a transaction (by writing the *new* values).
- **The Undo/Redo recovery policy is:**
  1. Redo all the committed transactions from the first to the most recent;
  2. Undo all the incomplete transactions from the most recent back to the first.
- **Note:** With the Undo/Redo technique *all* the transactions have to be either undone or redone.

Recovery with Undo/Redo Logging: An Example

Action	t	Mem A	Mem B	Disk A	Disk B	Log
READ(A,t)	8	8		8	8	< START T >
t := t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	< T, A, 8, 16 >
READ(B,t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	< T, B, 8, 16 >
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						< COMMIT T >
OUTPUT(B)	16	16	16	16	16	

If the crash occurs just after the flush of < COMMIT T >, then T is recognized as a committed transaction: We *redo* the changes writing the new value **16** for both A, B to the disk: WRITE(A, 16); WRITE(A, 16); OUTPUT(A); OUTPUT(B).

Recovery with Undo/Redo Logging: An Example (Cont.)

Action	t	Mem A	Mem B	Disk A	Disk B	Log
READ(A,t)	8	8		8	8	<START T>
t := t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8, 16>
READ(B,t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

If the crash occurs before <COMMIT T>, then T is recognized as an incomplete transaction: We *undo* the changes writing the old value **8** for both A, B to the disk: WRITE(A, 8); WRITE(A, 8); OUTPUT(A); OUTPUT(B); ABORT(T).

Nonquiescent Checkpointing with Undo/Redo Logging

- The steps in a **Nonquiescent Checkpointing** under the Undo/Redo logging policy are:
  1. Write a $<\mathtt{START\ CKPT}(\mathtt{T_1};\ldots;\mathtt{T_k})>$ record to log, where T_i are all the active (uncommitted) transactions, and flush the log;
  2. Write to disk all the DB elements written to buffers, but not yet to disk – unlike redo logging, we flush *all* buffers, not just the ones written by committed transactions;
  3. Write $<\mathtt{END\ CKPT}>$ record to log, and flush the log.

Summary

- Notion of Transaction
- Failure Recovery
  - Failure Classification
- Logging Techniques for System Failure Recovery
  1. Undo Logging
  2. Redo Logging
  3. Undo/Redo Logging
- **Recovery from Media Failure: Archiving.**

Recovery from Media Failure: Archiving + Logging

- Using just the Log is not sufficient:
  - Also if the Log is still available we would need the full Redo Log;
  - The log would tell us the history of the DB since he was populated (no deletions after checkpoints);
  - This is not practicable: the size of such a log would exceed that one of the DB.

Recovery from Media Failure: Archiving + Logging

- **Archiving:** Assumes that there is a backup copy (possibly made with an automatic RAID procedure).
- **Main Idea:** Use backup copies of the DB + the Log that records the full history since the backup archive was created.
- There are two levels of Archiving:
  1. **Full Dump:** The whole DB is copied;
  2. **Incremental Dump:** Only the DB elements changed since the previous dump are copied (such incremental dumps can be created faster than full dumps).

Nonquiescent Archiving

- **Problem.** A DB cannot shut down during the creation of the backup copy.
- **Solution.** Allowing transactions to execute during backup: DB elements can change (and possibly written back to disk) during the backup.
  - Use the Log entries generated during the backup time to restore a consistent DB state.
- **Steps in Nonquiescent Archiving:**
  1. A Nonquiescent archive tries to make a backup copy of the DB when the dump began;
  2. Transactions can modify the DB state;
  3. The Log entries recorded during the time spent for the backup will help to restore a consistent DB state.

Nonquiescent Archiving: An Example

- **Example.** Suppose we have a DB made by 4 elements A, B, C, D whose values are $1, 2, 3, 4$ and we have the following events during a backup:

Disk	Archive
	Copy A
A := 5	
	Copy B
C := 6	
	Copy C
	Copy D

Then, while at the beginning the DB had values $(1, 2, 3, 4)$, and at the end of the dump has values $(5, 2, 6, 4)$, the dump copy has values $(\mathbf{1}, 2, 6, 4)$: **The value for A is wrong!**

Recovery a Consistent Dump with Nonquiescent Archiving

To restore a consistent dump using a nonquiescent archiving under a redo-logging technique we do:

1. Write a <`START DUMP`> record to the Log;
2. Perform a checkpoint in the redo style;
3. Start a full/incremental dump;
4. If the dump completed successfully, including a copy of the Log, write <`END DUMP`> record to the Log;
5. Use a REDO policy to restore a consistent dump.

Recovery a Consistent Dump: An Example

Suppose the changes to the simple DB in the previous example were caused by transactions $\texttt{T}_1, \texttt{T}_2$ active when the dump began. We have the following log:

$<\texttt{START T}_1>$	$<\texttt{T}_2, \texttt{A}, \texttt{1}, \texttt{5}>$
...	$<\texttt{T}_2, \texttt{C}, \texttt{3}, \texttt{6}>$
$<\texttt{START T}_2>$	$<\texttt{COMMIT T}_2>$
...	$<\texttt{T}_1, \texttt{B}, \texttt{2}, \texttt{7}>$
$<\texttt{START DUMP}>$	$<\texttt{END CKPT}>$
$<\texttt{START CKPT}(\texttt{T}_1;\texttt{T}_2)>$	$<\texttt{END DUMP}>$

Since we are in a REDO policy and we first meet $<\texttt{END CKPT}>$ we need to redo only the committed transactions that are in the $<\texttt{START CKPT}>$ list and that started after the checkpoint. In this case only $\texttt{T}_2$ has to be redone.

While before the REDO the backup had the values $(1, 2, 6, 4)$, after the REDO the backup has the values $(5, 2, 6, 4)$.

Recovery the DB with Nonquiescent Archiving + Logging

Suppose we have a full dump, then the steps are:

1. Find the most recent full dump and restore it to a consistent state using the nonquiescent archiving technique;
2. Find possibly later incremental dumps and restore them;
3. Modify the (restored) full dump using the (restored) incremental dump, starting from the earliest to the most recent;
4. Use the log associated to the more recent dump to advance to a more recent state.

Summary of Lecture VI

- Notion of Transaction
- Failure Recovery
  - Failure Classification
- Logging Techniques for System Failure Recovery
  1. Undo Logging
  2. Redo Logging
  3. Undo/Redo Logging
- Recovery from Media Failure: Archiving.