Chapter 10: Failure Recovery

Zhaonian Zou

Massive Data Computing Research Center
School of Computer Science and Technology
Harbin Institute of Technology, China
Email: znzou@hit.edu.cn

Spring 2019

Outline[1]

1. 10.1 Transactions
2. 10.2 Overview of Failure Recovery
3. 10.3 Write-Ahead Logging (WAL)
    - 10.3.1 Undo Logging
    - 10.3.2 Redo Logging
    - 10.3.3 Undo/Redo Logging
4. 10.4 Archiving
5. 10.5 The ARIES Recovery Algorithm

[1] Updated on April 15, 2019

10.1 Transactions

Transactions (事务)

A transaction is a collection of one or more operations on the database that must be executed atomically (原子地); that is, either all operations are performed or none are (全做或全不做)

Example (Fund-Transfer Transaction)

```
EXEC SQL BEGIN DECLARE SECTION;
  int acct1, acct2, balance1, amount;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT balance INTO :balance1
  FROM Accounts WHERE acctNo = :acct1;
if (balance1 > amount) {
  EXEC SQL UPDATE Accounts SET balance = balance + :amount
    WHERE acctNo = :acct2;
  EXEC SQL UPDATE Accounts SET balance = balance - :amount
    WHERE acctNo = :acct1;
  COMMIT;
}
ROLLBACK;
```

The Commands of Transactions

- START TRANSACTION or BEGIN (事务启动)
  - The transaction starts
- COMMIT (事务提交)
  - The transaction is committed (completes successfully)
  - All changes to the database are installed permanently in the database (事务提交则全部落实)
- ROLLBACK (事务回滚)
  - The transaction is aborted (terminates unsuccessfully, 中止[2])
  - Any changes made by the transaction are undone (撤销), so they no longer appear in the database (事务中止则全部撤销)

[2] 不是终止

The ACID Properties of Transactions

A DBMS must ensure four important properties of transactions to maintain data in the face of concurrent execution and system failures

- Atomicity (原子性)
- Consistency (一致性)
- Isolation (隔离性)
- Durability (持久性)

Atomicity (原子性)

- Users should be able to regard the execution of each transaction as atomic (原子的): Either all actions are carried out or none are
- Users should not have to worry about the effect of incomplete transactions (不完整事务)
- Transactions can be incomplete for three kinds of reasons
  - A transaction can be aborted, or terminated unsuccessfully, by the DBMS because some internal anomaly arises during execution (e.g., division by 0)
  - The system may crash while one or more transactions are in progress
  - A transaction may encounter an unexpected situation (e.g., read an unexpected data value or be unable to get some resources) and decide to abort itself

Consistency (一致性)

- Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database
  - The consistency of the database is further beyond the integrity constraints in semantics
  - Example: Fund transfers between bank accounts should not change the total amount of money in the accounts
- The DBMS assumes that consistency holds for each transaction
- Users are responsible for ensuring transaction consistency
- An incomplete transaction may leave the database in an inconsistent state

Isolation (隔离性)

- Transactions are isolated, or protected, from the effects of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons
- Even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order
- If each transaction changes a consistent database instance to another consistent database instance, executing several transactions on a consistent initial database instance results in a consistent final database instance

Durability (持久性)

- Once the DBMS informs the user that a transaction has been successfully completed (committed), its effects should persist even if the system crashes before all its changes are reflected on disk

Transaction Management (事务管理)

Transaction management ensures the ACID properties of transactions in face of concurrent execution (并发执行) and system failures (系统故障)

Concurrency Control (并发控制) ⟶ consistency & isolation

- For performance reasons, a DBMS has to interleave the actions of several transactions
- Requirement: The result of a concurrent execution (并发执行) of transactions is equivalent (in its effects on the database) to some serial execution (串行执行) of the same set of transactions
- How does a DBMS handle concurrent executions of transactions?

Failure Recovery (故障恢复) ⟶ atomicity & durability

- A transaction is partial, or incomplete, if it is interrupted before it runs to normal completion
- Requirement: The changes made by partial transactions are not seen by other transactions
- How does a DBMS remove the effects of partial transactions?

The Simple Model of Transactions

A transaction is seen by the DBMS as a series of actions or operations

- The transaction reads database elements (objects)
- The transaction writes database elements
- The transaction commits (completes successfully)
- The transaction aborts (terminates and undo all the actions carried out thus far)

Assumptions

- Transactions interact with each other only via database read and write operations; they are not allowed to exchange messages
- A database is a fixed collection of independent objects. When objects are added to or deleted from a database or there are relationships between database objects that we want to exploit for performance, some additional issues arise.

Address Spaces (地址空间)

The database elements (objects) are placed in three address spaces

- The space of disk blocks holding database elements
- The main memory address space managed by the buffer manager
- The local address space of the transaction

The Primitive Read/Write Operations of Transactions

- `INPUT(X)`: Copy the disk block containing database element X to a main-memory buffer
- `READ(X, t)`: Copy database element X to the transaction's local variable t. If the block containing X is not in a memory buffer, then first execute INPUT(X). Next, assign the value of X to t.
- `WRITE(X, t)`: Copy the value of local variable t to database element X in a memory buffer. If the block containing X is not in a memory buffer, then execute INPUT(X). Next, copy the value of t to X in the buffer.
- `OUTPUT(X)`: Copy the block containing X from its buffer to disk

The Primitive Read/Write Operations of Transactions (Cont'd)

Example (Transaction Primitives & Address Spaces)

Step	Action	t	M_A	M_B	D_A	D_B
0					8	8
1	`READ(A, t)`	8	8		8	8
2	`t := t * 2`	16	8		8	8
3	`WRITE(A, t)`	16	16		8	8
4	`READ(B, t)`	8	16	8	8	8
5	`t := t * 2`	16	16	8	8	8
6	`WRITE(B, t)`	16	16	16	8	8
7	`OUTPUT(A)`	16	16	16	16	8
8	`OUTPUT(B)`	16	16	16	16	16

10.2 Overview of Failure Recovery

Recovery Manager (恢复管理器)

Function 1 (Recovery): The recovery manager is responsible for ensuring the atomicity and durability of transactions in the face of system failures

- When a DBMS is restarted after crashes, the recovery manager is given control and must bring the database to a consistent state
- It ensures atomicity by undoing the actions of transactions that do not commit
- It ensures durability by making sure that all actions of committed transactions survive system crashes and media failures

Function 2 (Rollback): The recovery manager is also responsible for executing the `ROLLBACK` command, that is, undoing the actions of an aborted transaction

Recovery Manager (Cont'd)

To implement a recovery manager, it is necessary to understand how database objects are written during normal execution of transactions

- (Steal or not?) Can the changes made to an object X in the buffer pool by a transaction T be written to disk before T commits?
  Such writes are executed when another transaction wants to bring in a page and the buffer manager chooses to replace the frame containing X (of course, this page must have been unpinned by T)
- (Force or not?) When a transaction commits, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk?

Steal/No-Steal Approaches

(Steal or not?) Can the changes made to an object X in the buffer pool by a transaction T be written to disk before T commits?

- Steal Approach: Such writes are allowed (The second transaction "steals" the frame containing X from T)
- No-Steal Approach: Such writes are not allowed

- Advantage of No-Steal: (中止事务无需undo) We do not have to undo the changes of an aborted transaction because these changes have not been written to disk
- Drawback of No-Steal: (内存开销大) The no-steal approach assumes that all pages modified by ongoing transactions can be accommodated in the buffer pool, and in the presence of large transactions (typically run in batch mode, e.g., payroll processing), this assumption is unrealistic
- Advantage of Steal: (充分利用内存)
- Drawback of Steal: (中止事务需undo)

Force/No-Force Approaches

(Force write or not?) When a transaction commits, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk?

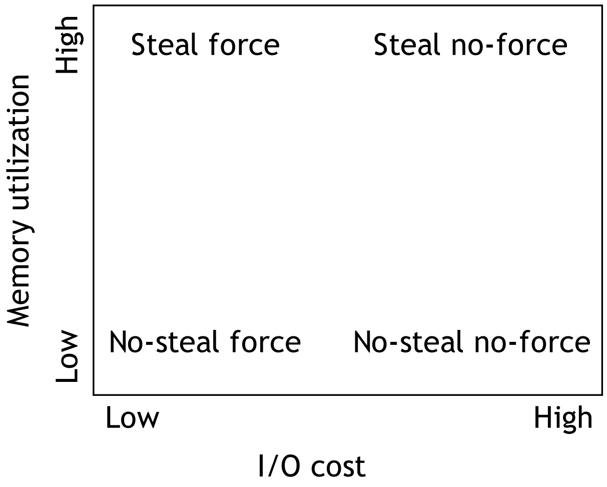
- Force Approach: such writes are forced
- No-Force Approach: such writes are not forced

- Advantage of Force: (已提交事务无需redo) We do not have to redo the changes of a committed transaction if there is a subsequent crash because all these changes are guaranteed to have been written to disk at commit time
- Drawback of Force: (I/O代价高) If a highly used page is updated in succession by n transactions, it would be written to disk n times
- Advantage of No-Force: (I/O代价低)
- Drawback of No-Force: (已提交事务需redo)

Four Approaches to Writing Database Elements

	Force	No-Force
Steal	Steal Force	Steal No-Force
No-Steal	No-Steal Force	No-Steal No-Force

- A no-steal force approach is the simplest one to implement, but has bad efficiency
- Most DBMSs use a steal no-force approach



Recovery-Related Steps during Normal Execution

[3]

"No lunch for free"

The recovery manager maintains some information during normal execution of transactions to enable recovery in the event of a failure

- Log (日志)
- Checkpoints (检查点)

[3] http://ws3.sinaimg.cn/bmiddle/9150e4e5ly1flvtk73zzuj20cc07m3yl.jpg

Log (日志)

- During normal execution of transactions, a log of all modifications to the database is saved on stable storage (稳定存储器), which is assumed to survive crashes
- Write-Ahead Logging (WAL, 预写日志): The log records describing a change to the database must be written to stable storage before the change is made; otherwise, the system might crash just after the change, leaving us without a record of the change
- The log enables the recovery manager to undo (撤销) the actions of aborted and incomplete transactions and redo (重做) the actions of committed transactions
  - Incomplete transactions, a steal approach → undo the transactions
  - Committed transactions, a no-force approach → redo the transactions

		Redo committed transactions	
		No	Yes
Undo incomplete transactions	Yes	Steal Force	Steal No-Force
	No	No-Steal Force	No-Steal No-Force

Checkpointing (检查点)

- The amount of work involved during recovery is proportional to the changes made by committed transactions that have not been written to disk at the time of the crash
- To reduce the time to recover from a crash, the DBMS periodically forces buffer pages to disk during normal execution using a background process called checkpointing

Rollback

- The recovery manager is also responsible for executing the `ROLLBACK` command, which aborts a single transaction
- The logic (and code) involved in undoing a single transaction is identical to that used during the Undo phase in recovering from a system crash
- All log records for a given transctction are organized in a linked list and can be efficiently accessed in reverse order to facilitate transaction rollback

10.3 Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL, 预写日志技术)

- In-Place Updating (原地更新): Writing a page in the buffer pool back to the same original disk block, thus overwriting the old value of any changed database elements on disk
- Write-Ahead Logging (WAL, 预写日志技术): When in-place updating is used, it is necessary to use a log (日志) for recovery. The log records describing a change to the database are written to stable storage before the change is made; otherwise, the system might crash just after the change, leaving us without a record of the change

10.3 Write-Ahead Logging (WAL)
10.3.1 Undo Logging

Undo Logging (Undo日志技术)

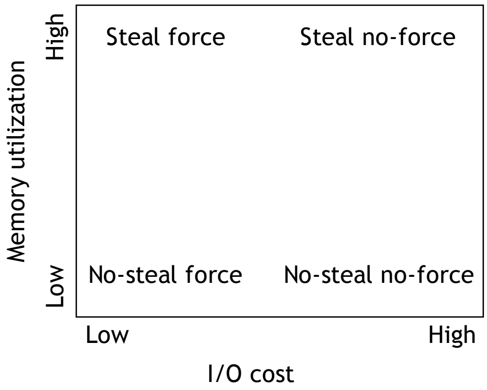
- $< \texttt{START}\ T >$: Transaction T has begun
- $< \texttt{COMMIT}\ T >$: Transaction T has completed successfully and will make no more changes to database elements
- $< \texttt{ABORT}\ T >$: Transaction T has aborted, and no changes it made can have been written to disk

The Undo-Logging Rules (Undo日志规则)

U_1: If transaction T modifies database element X, the log record of the form $< T, X, v >$ must be written to disk before the new value of X is written to disk, where v is the old value of X

U_2: If transaction T commits, a log record $< \texttt{COMMIT}\ T >$ must be written to disk only after all database elements changed by T have been written to disk, but as soon as possible

The undo logging is a steal force approach



The Undo-Logging Rules (Cont'd)

The log records and the database elements associated with one transaction T must be written to disk in the following order:

1. $<$ START $T >$
2. The log records $< T, X, v >$ indicating changed database elements X
3. The changed database elements X themselves
4. $<$ COMMIT $T >$

The Undo-Logging Rules (Cont'd)

Example (Undo Logging)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START $T >$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 8 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	$< T, B, 8 >$
8	`FLUSH LOG`						
9	`OUTPUT(A)`	16	16	16	16	8	
10	`OUTPUT(B)`	16	16	16	16	16	
11							$<$ COMMIT $T >$
12	`FLUSH LOG`						

Types of Transactions

Committed Transactions (已提交事务)

- A transaction T is committed if both $<$ START $T>$ and $<$ COMMIT $T>$ can be found on the log
- A committed transaction T must not be undone due to rule U_2 (all database elements changed by T have been written to disk before $<$ COMMIT $T>$ was written to disk)

Aborted Transactions (已中止事务)

- A transaction T is aborted if both $<$ START $T>$ and $<$ ABORT $T>$ can be found on the log
- An aborted transaction T must not be undone because all database elements changed by T have already been undone

Types of Transactions (Cont'd)

Incomplete Transactions (不完整事务)

- A transaction T is incomplete if we can find $<$ START $T>$ but neither $<$ COMMIT $T>$ nor $<$ ABORT $T>$ on the log
- An incomplete transaction T must be undone because several or all database elements changed by T were not written to disk
- How to undo? Rule U_1 assures us that if T changed database element X on disk before the crash, there will be a $< T, X, v >$ record on the log. Thus, during the recovery, we must write the value v for X.

Recovery using Undo-Logging

Scan the log backward (反向扫描), i.e., from the latest record to the earliest record

- $<$ COMMIT $T >$: Remember T, and T must not be undone
- $<$ ABORT $T >$: Remember T, and T must not be undone
- $< T, X, v >$: Do nothing if T is a committed transaction or an aborted transaction; otherwise, change the value of X in the database to v
- $<$ START $T >$: Write a log record $<$ ABORT $T >$ and flush the log if T is incomplete

Recovery using Undo-Logging (Cont'd)

Example (Recovery using Undo-Logging 1)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START $T >$
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	$< T, A, 8 >$
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	$< T, B, 8 >$
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
11							$<$ COMMIT $T >$
12	FLUSH LOG						
	System crash						

Recovery using Undo-Logging (Cont'd)

Example (Recovery using Undo-Logging 2)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$< \text{START } T >$
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	$< T, A, 8 >$
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	$< T, B, 8 >$
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
11							$< \text{COMMIT } T >$
	System crash						
12	FLUSH LOG						

Recovery using Undo-Logging (Cont'd)

Example (Recovery using Undo-Logging 3)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$< \text{START } T >$
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	$< T, A, 8 >$
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	$< T, B, 8 >$
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
	System crash						
11							$< \text{COMMIT } T >$
12	FLUSH LOG						

Recovery using Undo-Logging (Cont'd)

Example (Recovery using Undo-Logging 4)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START $T >$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 8 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	$< T, B, 8 >$
8	`FLUSH LOG`						
	System crash						
9	`OUTPUT(A)`	16	16	16	16	8	
10	`OUTPUT(B)`	16	16	16	16	16	
11							$<$ COMMIT $T >$
12	`FLUSH LOG`						

Recovery using Undo-Logging (Cont'd)

Example (Recovery using Undo-Logging 5)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START $T >$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 8 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	$< T, B, 8 >$
	System crash						
8	`FLUSH LOG`						
9	`OUTPUT(A)`	16	16	16	16	8	
10	`OUTPUT(B)`	16	16	16	16	16	
11							$<$ COMMIT $T >$
12	`FLUSH LOG`						

Recovery using Undo-Logging (Cont'd)

Example (Recovery using Undo-Logging 6)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START $T >$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 8 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	
	System crash						
8	`FLUSH LOG`						$< T, B, 8 >$
9	`OUTPUT(A)`	16	16	16	16	8	
10	`OUTPUT(B)`	16	16	16	16	16	
11							$<$ COMMIT $T >$
12	`FLUSH LOG`						

Checkpointing (检查点)

Problem: Can we recover the database without scanning the entire log?

Log	Action
$<$ START $T_1 >$	Useless
$< T_1, A, 5 >$	Useless
$<$ START $T_2 >$	Useless
$< T_2, B, 10 >$	Useless
$< T_2, C, 15 >$	Useless
$<$ COMMIT $T_2 >$	Useless
$<$ START $T_3 >$	
$< T_1, D, 20 >$	Useless
$< T_3, E, 25 >$	
$<$ COMMIT $T_1 >$	Useless
$< T_3, F, 30 >$	

Solution: Checkpointing!

Simple Checkpoint

1. Stop accepting new transactions
2. Wait until all currently active transactions commit or abort and have written a COMMIT or ABORT record on the log
3. Flush the log to disk
4. Write a log record $<$ CKPT $>$ and flush the log again
5. Resume accepting new transactions

Simple Checkpointing (Cont'd)

Example (Simple Checkpointing)

Without checkpoints	Using checkpoints
$<$ START T_1 $>$	$<$ START T_1 $>$
$< T_1, A, 5 >$	$< T_1, A, 5 >$
$<$ START T_2 $>$	$<$ START T_2 $>$
$< T_2, B, 10 >$	$< T_2, B, 10 >$
$< T_2, C, 15 >$	Make a checkpoint
$<$ COMMIT T_2 $>$	$< T_2, C, 15 >$
$<$ START T_3 $>$	$<$ COMMIT T_2 $>$
$< T_1, D, 20 >$	$< T_1, D, 20 >$
$< T_3, E, 25 >$	$<$ COMMIT T_1 $>$ // flush log
$<$ COMMIT T_1 $>$	$<$ CKPT $>$ // flush log
$< T_3, F, 30 >$	$<$ START T_3 $>$ // accept new transactions
	$< T_3, E, 25 >$
	$< T_3, F, 30 >$

Recovery with a Checkpointed Undo Log

Scan the log from the end to the latest $< \texttt{CKPT} >$ and do the undo-style recovery

Example (Recovery with a Checkpointed Undo Log)

Log	Action
$< \texttt{START}\ T_1 >$	
$< T_1, A, 5 >$	
$< \texttt{START}\ T_2 >$	
$< T_2, B, 10 >$	
Make a checkpoint	
$< T_2, C, 15 >$	
$< \texttt{COMMIT}\ T_2 >$	
$< T_1, D, 20 >$	
$< \texttt{COMMIT}\ T_1 >$	
$< \texttt{CKPT} >$	Stop recovery
$< \texttt{START}\ T_3 >$	$\uparrow$ Write $< \texttt{ABORT}\ T_3 >$ and flush the record
$< T_3, E, 25 >$	$\uparrow$ Restore the value of E on disk to 25
$< T_3, F, 30 >$	$\uparrow$ Restore the value of F on disk to 30

Nonquiescent Checkpointing (不停机检查点) / Fuzzy Checkpointing (模糊检查点)

Problem: Can new transactions be allowed to enter the system during the checkpoint?
Solution: Nonquiescent checkpointing

1. Write a log record $< \texttt{START CKPT}\ (T_1, \ldots, T_k) >$ and flush the log, where $T_1, \ldots, T_k$ are the identifiers of all the active transactions
2. Wait until all of $T_1, \ldots, T_k$ commit or abort, but do not prohibit other transactions from starting
3. When all of $T_1, \ldots, T_k$ have completed, write a log record $< \texttt{END CKPT} >$ and flush the log

Nonquiescent Checkpointing (Cont'd)

Example (Nonquiescent Checkpointing)

Without checkpoints

$< \mathtt{START}\ T_1 >$

$< T_1, A, 5 >$

$< \mathtt{START}\ T_2 >$

$< T_2, B, 10 >$

$< T_2, C, 15 >$

$< \mathtt{COMMIT}\ T_2 >$

$< \mathtt{START}\ T_3 >$

$< T_1, D, 20 >$

$< T_3, E, 25 >$

$< \mathtt{COMMIT}\ T_1 >$

$< T_3, F, 30 >$

Using checkpoints

$< \mathtt{START}\ T_1 >$

$< T_1, A, 5 >$

$< \mathtt{START}\ T_2 >$

$< T_2, B, 10 >$

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

$< T_2, C, 15 >$

$< \mathtt{COMMIT}\ T_2 >$

$< \mathtt{START}\ T_3 >$

$< T_1, D, 20 >$

$< T_3, E, 25 >$

$< \mathtt{COMMIT}\ T_1 >$

$< \mathtt{END\ CKPT} >$ // flush log

$< T_3, F, 30 >$

Recovery with a Checkpointed Undo Log

Case 1: When scanning the log backwards, we first meet an $< \mathtt{END\ CKPT} >$ record instead of a $< \mathtt{START\ CKPT}\ (T_1, \ldots, T_k) >$ record

- The crash occurs after the checkpoint
- All incomplete transactions began after the previous $< \mathtt{START\ CKPT}\ (T_1, \ldots, T_k) >$ record
- We may scan backwards as far as the previous $< \mathtt{START\ CKPT}\ (T_1, \ldots, T_k) >$ record, and then stop
- All records prior to the previous $< \mathtt{START\ CKPT}\ (T_1, \ldots, T_k) >$ record are useless and may have been discarded

Recovery with a Checkpointed Undo Log (Cont'd)

Example (Recovery with a Checkpointed Undo Log)

Log	Action
$<$ START $T_1 >$	
$< T_1, A, 5 >$	
$<$ START $T_2 >$	
$< T_2, B, 10 >$	
$<$ START CKPT $(T_1, T_2) >$	Stop recovery
$< T_2, C, 15 >$	↑
$<$ COMMIT $T_2 >$	↑ Remember T_2 as a complete transaction
$<$ START $T_3 >$	↑ Write $<$ ABORT $T_3 >$ and flush the record
$< T_1, D, 20 >$	↑
$< T_3, E, 25 >$	↑ Restore the value of E on disk to 25
$<$ COMMIT $T_1 >$	↑ Remember T_1 as a complete transaction
$<$ END CKPT $>$	↑
$< T_3, F, 30 >$	↑ Restore the value of F on disk to 30

Recovery with a Checkpointed Undo Log (Cont'd)

Case 2: When scanning the log backwards, we first meet a $<$ START CKPT $(T_1, \ldots, T_k) >$ record instead of an $<$ END CKPT $>$ record

- The crash occurs during the checkpoint
- The incomplete transactions are those we met scanning backwards before we reached the START CKPT and those of $T_1, \ldots, T_n$ that did not comlplete before the crash
- We may scan backwards as far as the start of the earliest of these incomplete transactions, and then stop
- The previous START CKPT record is certainly prior to the start of the earliest of these incomplete transactions
- All records prior to the previous START CKPT record are useless and may have been discarded

Recovery with a Checkpointed Undo Log (Cont'd)

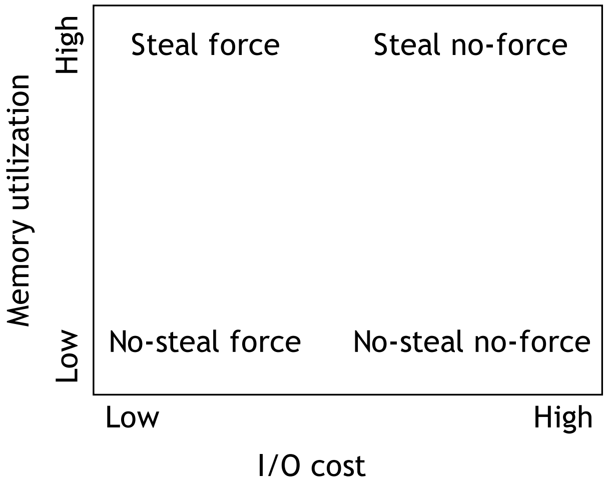
Example (Recovery with a Checkpointed Undo Log)

Log	Action
$< \texttt{START}\ T_1 >$	$\uparrow$ Write $< \texttt{ABORT}\ T_1 >$ and flush the record
$< T_1, A, 5 >$	$\uparrow$ Restore the value of A on disk to 5
$< \texttt{START}\ T_2 >$	$\uparrow$
$< T_2, B, 10 >$	$\uparrow$
$< \texttt{START CKPT}\ (T_1, T_2) >$	$\uparrow$ Continue undoing T_1
$< T_2, C, 15 >$	$\uparrow$
$< \texttt{COMMIT}\ T_2 >$	$\uparrow$ Remember T_2 as a complete transaction
$< \texttt{START}\ T_3 >$	$\uparrow$ Write $< \texttt{ABORT}\ T_3 >$ and flush the record
$< T_1, D, 20 >$	$\uparrow$ Restore the value of D on disk to 20
$< T_3, E, 25 >$	$\uparrow$ Restore the value of E on disk to 25

10.3 Write-Ahead Logging (WAL)
10.3.2 Redo Logging

Problems with Undo-Logging

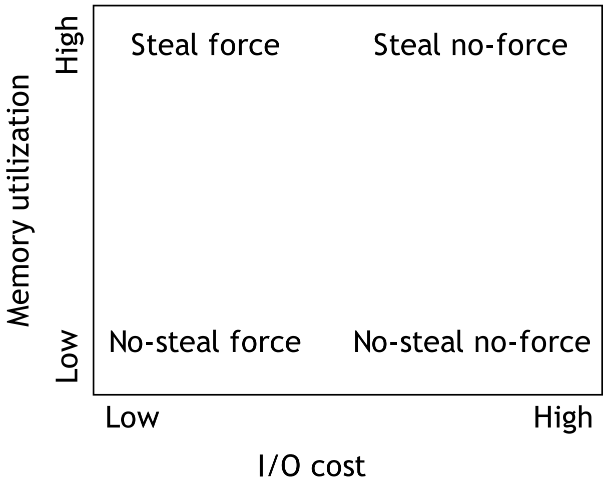
- The undo logging is a steal force approach
- We cannot commit a transaction without first writing all its changed database elements to disk
- Sometimes, we can save disk I/O's if we let changes to the database reside only in main memory for a while (writing eagerly → wrting lazily)



The Redo-Logging Rule (Redo日志规则)

R_1: Before modifying any database element X on disk, both the update record $< T, X, v >$ and the $<$ COMMIT $T >$ record must appear on disk, where v is the new value of X

The redo logging is a no-steal no-force approach



The Redo-Logging Rules (Cont'd)

The log records and the database elements associated with one transaction T must be written to disk in the following order:

1. $<$ START T $>$
2. The log records $< T, X, v >$ indicating changed database elements X
3. $<$ COMMIT T $>$
4. The changed database elements X themselves

The Redo-Logging Rules (Cont'd)

Example (Redo Logging)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START T $>$
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	$< T, A, 16 >$
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	$< T, B, 16 >$
8							$<$ COMMIT T $>$
9	FLUSH LOG						
10	OUTPUT(A)	16	16	16	16	8	
11	OUTPUT(B)	16	16	16	16	16	

Types of Transactions

Aborted Transactions (已中止事务)

- A transaction T is aborted if both $<$ START $T >$ and $<$ ABORT $T >$ can be found on the log
- An aborted transaction T must not be redone due to rule R_1 (no database elements changed by T have been written to disk before $<$ COMMIT $T >$ was written to disk)

Incomplete Transactions (不完整事务)

- A transaction T is incomplete if we can find $<$ START $T >$ but neither $<$ COMMIT $T >$ nor $<$ ABORT $T >$ on the log
- An incomplete transaction T must not be redone due to rule R_1

Types of Transactions (Cont'd)

Committed Transactions (已提交事务)

- A transaction T is committed if both $<$ START $T >$ and $<$ COMMIT $T >$ can be found on the log
- A committed transaction T must be redone because several or all database elements changed by T were not written to disk
- How to redo? Rule R_1 assures us that if T changed database element X before the crash, there will be a $< T, X, v >$ record on the log. Thus, during the recovery, we must write the value v for X.

Recovery using Redo-Logging

1. Identify the committed transactions
2. Scan the log from the earliest record to the latest record (前向扫描)
   $< T, X, v >$: Change the value of X in the database to v if T is a committed transaction; otherwise, do nothing
3. For each incomplete transaction T, write an $<$ ABORT $T >$ record and flush the log

Recovery using Redo-Logging (Cont'd)

Example (Recovery using Redo-Logging 1)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START $T >$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 16 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	$< T, B, 16 >$
8							$<$ COMMIT $T >$
9	`FLUSH LOG`						
10	`OUTPUT(A)`	16	16	16	16	8	
11	`OUTPUT(B)`	16	16	16	16	16	

System crash

Recovery using Redo-Logging (Cont'd)

Example (Recovery using Redo-Logging 2)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START T $>$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 16 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	$< T, B, 16 >$
8							$<$ COMMIT T $>$
9	`FLUSH LOG`						
	System crash						
10	`OUTPUT(A)`	16	16	16	16	8	
11	`OUTPUT(B)`	16	16	16	16	16	

Recovery using Redo-Logging (Cont'd)

Example (Recovery using Redo-Logging 3)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START T $>$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 16 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	$< T, B, 16 >$
8							$<$ COMMIT T $>$
	System crash						
9	`FLUSH LOG`						
10	`OUTPUT(A)`	16	16	16	16	8	
11	`OUTPUT(B)`	16	16	16	16	16	

Recovery using Redo-Logging (Cont'd)

Example (Recovery using Redo-Logging 4)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START T $>$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 16 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	$< T, B, 16 >$
	System crash						
8							$<$ COMMIT T $>$
9	`FLUSH LOG`						
10	`OUTPUT(A)`	16	16	16	16	8	
11	`OUTPUT(B)`	16	16	16	16	16	

Checkpointing a Redo Log

1. Write a log record $<$ START CKPT $(T_1, \ldots, T_k)$ $>$ and flush the log, where $T_1, \ldots, T_k$ are all the active transactions
2. Write to disk all dirty database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log
3. Write an $<$ END CKPT $>$ record and flush the log

Checkpointing a Redo Log (Cont'd)

Example (Checkpointing a Redo Log)

Without checkpoints

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

Using checkpoints

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

Recovery with a Checkpointed Redo Log: Case 1

Case 1: When scanning the log backwards, we first meet an $< \texttt{END CKPT} >$ record instead of a $< \texttt{START CKPT}\ (T_1, \ldots, T_k) >$ record

- The crash occurs after the checkpoint
- The new values of the database elements changed by all transactions committed prior to the START CKPT record must have been written to disk
- The new values of some or all database elements changed by the transactions committed after the START CKPT record may not have been written to disk
- Any transaction committed after the START CKPT record is either one of the T_i's mentioned in the START CKPT record or started after the START CKPT record
- We may scan forward from the earliest of the $< \texttt{START}\ T_i >$ records
- All records prior to the earliest of the $< \texttt{START}\ T_i >$ records are useless and may have been discarded

Recovery with a Checkpointed Redo Log: Case 1 (Cont'd)

$< \texttt{START}\ T >$

$\ldots$

$< \texttt{COMMIT}\ T >$

$\ldots$

$< \texttt{START CKPT}\ (T_1, \ldots, T_k) >$

$\ldots$

$< \texttt{END CKPT} >$

$\ldots$

All the changes made by T have been written to the disk permanently, so T must not be redone

Recovery with a Checkpointed Redo Log: Case 1 (Cont'd)

$< \texttt{START}\ T >$

$\ldots$

$< \texttt{START CKPT}\ (T_1, \ldots, T_k) >$

$\ldots$

$< \texttt{COMMIT}\ T >$

$\ldots$

$< \texttt{END CKPT} >$

$\ldots$

We have $T \in \{T_1, T_2, \ldots, T_k\}$, and T must be redone

Recovery with a Checkpointed Redo Log: Case 1 (Cont'd)

$< \mathtt{START}\ T >$

$\ldots$

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

$\ldots$

$< \mathtt{END\ CKPT} >$

$\ldots$

$< \mathtt{COMMIT}\ T >$

$\ldots$

We have $T \in \{T_1, T_2, \ldots, T_k\}$, and T must be redone

Recovery with a Checkpointed Redo Log: Case 1 (Cont'd)

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

$\ldots$

$< \mathtt{START}\ T >$

$\ldots$

$< \mathtt{END\ CKPT} >$

$\ldots$

$< \mathtt{COMMIT}\ T >$

$\ldots$

We have $T \notin \{T_1, T_2, \ldots, T_k\}$, and T must be redone

Recovery with a Checkpointed Redo Log: Case 1 (Cont'd)

$<\texttt{START CKPT}\ (T_1, \ldots, T_k)>$
$\ldots$
$<\texttt{END CKPT}>$
$\ldots$
$<\texttt{START}\ T>$
$\ldots$
$<\texttt{COMMIT}\ T>$
$\ldots$
We have $T \notin \{T_1, T_2, \ldots, T_k\}$, and T must be redone

Recovery with a Checkpointed Redo Log: Case 1 (Cont'd)

Example (Recovery with a Checkpointed Redo Log: Case 1)

Log	Action
$<\texttt{START}\ T_1>$	
$<T_1, A, 5>$	
$<\texttt{START}\ T_2>$	↓ Start redoing T_2
$<\texttt{COMMIT}\ T_1>$	↓
$<T_2, B, 10>$	↓ Rewrite the value of B on disk to 10
$<\texttt{START CKPT}\ (T_2)>$	↓
$<T_2, C, 15>$	↓ Rewrite the value of C on disk to 15
$<\texttt{START}\ T_3>$	↓ Start redoing T_3
$<\texttt{END CKPT}>$	↓
$<T_3, D, 20>$	↓ Rewrite the value of D on disk to 20
$<\texttt{COMMIT}\ T_2>$	↓ Stop redoing T_2
$<\texttt{COMMIT}\ T_3>$	↓ Stop redoing T_3
	Stop recovery

Recovery with a Checkpointed Redo Log: Case 1 (Cont'd)

Example (Recovery with a Checkpointed Redo Log: Case 1)

Log	Action
$<$ START T_1 $>$	
$< T_1, A, 5 >$	
$<$ START T_2 $>$	↓ Start redoing T_2
$<$ COMMIT T_1 $>$	↓
$< T_2, B, 10 >$	↓ Rewrite the value of B on disk to 10
$<$ START CKPT (T_2) $>$	↓
$< T_2, C, 15 >$	↓ Rewrite the value of C on disk to 15
$<$ START T_3 $>$	↓
$<$ END CKPT $>$	↓
$< T_3, D, 20 >$	↓
$<$ COMMIT T_2 $>$	↓ Stop redoing T_2
	↓ Write $<$ ABORT T_3 $>$ and flush the log
	Stop recovery

Recovery with a Checkpointed Redo Log: Case 2

Case 2: When scanning the log backwards, we first meet a $<$ START CKPT $(T_1, \ldots, T_k)$ $>$ record instead of an $<$ END CKPT $>$ record

- The crash occurs during the checkpoint
- The new values of the database elements changed by all transactions committed prior to the START CKPT record may not have been written to disk
- The new values of the database elements changed by all transactions committed prior to the previous $<$ START CKPT $(S_1, \ldots, S_m)$ $>$ record must have been written to disk
- We may scan forward from the earliest of the $<$ START S_i $>$ records
- All records prior to the earliest of the $<$ START S_i $>$ records are useless and may have been discarded

Recovery with a Checkpointed Redo Log: Case 2 (Cont'd)

$< \texttt{START}\ T >$

$\ldots$

$< \texttt{COMMIT}\ T >$

$\ldots$

$< \texttt{START CKPT}\ (S_1, \ldots, S_m) >$

$\ldots$

$< \texttt{END CKPT} >$

$\ldots$

$< \texttt{START CKPT}\ (T_1, \ldots, T_k) >$

$\ldots$

All the changes made by T have been written to the disk permanently, so T must not be redone

Recovery with a Checkpointed Redo Log: Case 2 (Cont'd)

$< \texttt{START}\ T >$

$\ldots$

$< \texttt{START CKPT}\ (S_1, \ldots, S_m) >$

$\ldots$

$< \texttt{COMMIT}\ T >$

$\ldots$

$< \texttt{END CKPT} >$

$\ldots$

$< \texttt{START CKPT}\ (T_1, \ldots, T_k) >$

$\ldots$

We have $T \in \{S_1, S_2, \ldots, S_m\}$ and $T \notin \{T_1, T_2, \ldots, T_k\}$. One or more changes made by T might not have been written to the disk permanently.

Recovery with a Checkpointed Redo Log: Case 2 (Cont'd)

$< \texttt{START}\ T >$
$\ldots$
$< \texttt{START CKPT}\ (S_1, \ldots, S_m) >$
$\ldots$
$< \texttt{END CKPT} >$
$\ldots$
$< \texttt{COMMIT}\ T >$
$\ldots$
$< \texttt{START CKPT}\ (T_1, \ldots, T_k) >$
$\ldots$

We have $T \in \{S_1, S_2, \ldots, S_m\}$ and $T \notin \{T_1, T_2, \ldots, T_k\}$. One or more changes made by T might not have been written to the disk permanently.

Recovery with a Checkpointed Redo Log: Case 2 (Cont'd)

$< \texttt{START}\ T >$
$\ldots$
$< \texttt{START CKPT}\ (S_1, \ldots, S_m) >$
$\ldots$
$< \texttt{END CKPT} >$
$\ldots$
$< \texttt{START CKPT}\ (T_1, \ldots, T_k) >$
$\ldots$
$< \texttt{COMMIT}\ T >$
$\ldots$

We have $T \in \{S_1, S_2, \ldots, S_m\} \cap \{T_1, T_2, \ldots, T_k\}$. One or more changes made by T might not have been written to the disk permanently.

Recovery with a Checkpointed Redo Log: Case 2 (Cont'd)

$<$ START CKPT $(S_1, \ldots, S_m) >$

$\ldots$

$<$ START $T >$

$\ldots$

$<$ END CKPT $>$

$\ldots$

$<$ COMMIT $T >$

$\ldots$

$<$ START CKPT $(T_1, \ldots, T_k) >$

$\ldots$

We have $T \notin \{T_1, T_2, \ldots, T_k\}$. One or more changes made by T might not have been written to the disk permanently.

Recovery with a Checkpointed Redo Log: Case 2 (Cont'd)

$<$ START CKPT $(S_1, \ldots, S_m) >$

$\ldots$

$<$ END CKPT $>$

$\ldots$

$<$ START $T >$

$\ldots$

$<$ COMMIT $T >$

$\ldots$

$<$ START CKPT $(T_1, \ldots, T_k) >$

$\ldots$

We have $T \notin \{T_1, T_2, \ldots, T_k\}$. One or more changes made by T might not have been written to the disk permanently.

Recovery with a Checkpointed Redo Log: Case 2 (Cont'd)

$< \texttt{START CKPT } (S_1, \ldots, S_m) >$

$\ldots$

$< \texttt{END CKPT} >$

$\ldots$

$< \texttt{START } T >$

$\ldots$

$< \texttt{START CKPT } (T_1, \ldots, T_k) >$

$\ldots$

$< \texttt{COMMIT } T >$

$\ldots$

We have $T \in \{T_1, T_2, \ldots, T_k\}$. One or more changes made by T might not have been written to the disk permanently.

Recovery with a Checkpointed Redo Log: Case 2 (Cont'd)

Example (Recovery with a Checkpointed Redo Log: Case 2)

Log	Action
$< \texttt{START } T_1 >$	↓ Start redoing T_1
$< T_1, A, 5 >$	↓ Rewrite the value of A on disk to 5
$< \texttt{START } T_2 >$	↓
$< \texttt{START CKPT } (T_1, T_2) >$	↓
$< \texttt{COMMIT } T_1 >$	↓ Stop redoing T_1
$< T_2, B, 10 >$	↓
$< \texttt{END CKPT} >$	↓
$< T_2, C, 15 >$	↓
$< \texttt{START CKPT } (T_2) >$	↓
$< \texttt{START } T_3 >$	↓
$< T_3, D, 20 >$	↓
	↓ Write $< \texttt{ABORT } T_2 >$ and flush the log
	↓ Write $< \texttt{ABORT } T_3 >$ and flush the log
	Stop recovery

10.3 Write-Ahead Logging (WAL)
10.3.3 Undo/Redo Logging

Problems with Redo Logging

- The redo logging is a no-steal no-force approach
- Redo logging requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed, perhaps increasing the average number of buffers required by transactions
- Both undo and redo logs may put contradictory requirements on how buffers are handled during a checkpoint

A block in buffer: $\boxed{X, Y}$

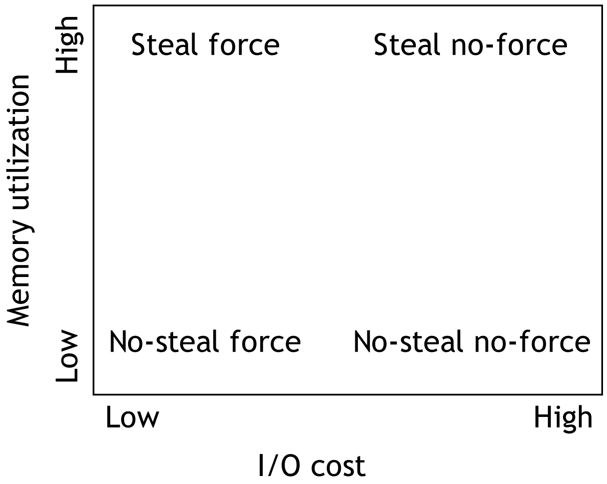
- Transaction T_1 changes database element X, and transaction T_2 changes database element Y
- We use undo logging for T_1 and use redo logging for T_2
- Both T_1 and T_2 commit

What will happen if T_1 commits before T_2?

The Undo/Redo-Logging Rule (Undo/Redo日志规则)

UR$_1$: Before modifying any database element X on disk, the update record $< T, X, v, w >$ must appear on disk, where v is the old value of X and w is the new value of X

The undo/redo logging is a steal no-force approach



The Undo/Redo-Logging Rule (Cont'd)

Example (Undo/Redo Logging)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START $T >$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 8, 16 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	$< T, B, 8, 16 >$
8	`FLUSH LOG`						
9	`OUTPUT(A)`	16	16	16	16	8	
10							$<$ COMMIT $T >$
11	`OUTPUT(B)`	16	16	16	16	16	

Recovery using Undo/Redo-Logging

1. Redo Phase: Redo all the committed transactions in the order of earliest-first
2. Undo Phase: Undo all the uncommitted transactions in the order latest-first

Recovery using Undo/Redo-Logging (Cont'd)

Example (Recovery using Undo/Redo-Logging 1)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<\texttt{START}\ T>$
2	`READ(A, t)`	8	8		8	8	
3	`t := t * 2`	16	8		8	8	
4	`WRITE(A, t)`	16	16		8	8	$< T, A, 8, 16 >$
5	`READ(B, t)`	8	16	8	8	8	
6	`t := t * 2`	16	16	8	8	8	
7	`WRITE(B, t)`	16	16	16	8	8	$< T, B, 8, 16 >$
8	`FLUSH LOG`						
9	`OUTPUT(A)`	16	16	16	16	8	
10							$<\texttt{COMMIT}\ T>$
11	`OUTPUT(B)`	16	16	16	16	16	
	System crash						

Recovery using Undo/Redo-Logging (Cont'd)

Example (Recovery using Undo/Redo-Logging 2)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START T $>$
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	$< T, A, 8, 16 >$
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	$< T, B, 8, 16 >$
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10							$<$ COMMIT T $>$
	System crash						
11	OUTPUT(B)	16	16	16	16	16	

Recovery using Undo/Redo-Logging (Cont'd)

Example (Recovery using Undo/Redo-Logging 3)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							$<$ START T $>$
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	$< T, A, 8, 16 >$
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	$< T, B, 8, 16 >$
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
	System crash						
10							$<$ COMMIT T $>$
11	OUTPUT(B)	16	16	16	16	16	

Checkpointing an Undo/Redo Log

1. Write a log record $<$ START CKPT $(T_1, \ldots, T_k) >$ and flush the log, where $T_1, \ldots, T_k$ are all the active transactions
2. Write to disk all dirty buffers, i.e., they contain one or more changed database elements. Unlike redo logging, we flush all buffers, not just those written by committed transactions.
3. Write an $<$ END CKPT $>$ record and flush the log

Checkpointing an Undo/Redo Log (Cont'd)

Example (Checkpointing an Undo/Redo Log)

Without checkpoints

$<$ START $T_1 >$
$< T_1, A, 4, 5 >$
$<$ START $T_2 >$
$<$ COMMIT $T_1 >$
$< T_2, B, 9, 10 >$
$< T_2, C, 14, 15 >$
$<$ START $T_3 >$
$< T_3, D, 19, 20 >$
$<$ COMMIT $T_2 >$
$<$ COMMIT $T_3 >$

Using checkpoints

$<$ START $T_1 >$
$< T_1, A, 4, 5 >$
$<$ START $T_2 >$
$<$ COMMIT $T_1 >$
$< T_2, B, 9, 10 >$
$<$ START CKPT $(T_2) >$
$< T_2, C, 14, 15 >$
$<$ START $T_3 >$
$< T_3, D, 19, 20 >$
$<$ END CKPT $>$
$<$ COMMIT $T_2 >$
$<$ COMMIT $T_3 >$

Recovery with a Checkpointed Undo/Redo Log (Cont'd)

Example (Recovery with a Checkpointed Undo/Redo Log 1)

Log	Action[a]
$< \texttt{START}\ T_1 >$	
$< T_1, A, 4, 5 >$	
$< \texttt{START}\ T_2 >$	↓ We don't have to start redoing T_2 from here
$< \texttt{COMMIT}\ T_1 >$	
$< T_2, B, 9, 10 >$	
$< \texttt{START CKPT}\ (T_2) >$	↓ Start redoing T_2
$< T_2, C, 14, 15 >$	↓ Rewrite the value of C on disk to 15
$< \texttt{START}\ T_3 >$	↓ Start redoing T_3
$< T_3, D, 19, 20 >$	↓ Rewrite the value of D on disk to 20
$< \texttt{END CKPT} >$	
$< \texttt{COMMIT}\ T_2 >$	↓ Stop redoing T_2
$< \texttt{COMMIT}\ T_3 >$	↓ Stop redoing T_3

[a] The actions shown in black color are performed during the redo-style recovery, and the actions shown in blue color are performed during the undo-style recovery

Recovery with a Checkpointed Undo/Redo Log (Cont'd)

Example (Recovery with a Checkpointed Undo/Redo Log 2)

Log	Action[a]
$< \texttt{START}\ T_1 >$	
$< T_1, A, 4, 5 >$	
$< \texttt{START}\ T_2 >$	
$< \texttt{COMMIT}\ T_1 >$	
$< T_2, B, 9, 10 >$	
$< \texttt{START CKPT}\ (T_2) >$	↓ Start redoing T_2
$< T_2, C, 14, 15 >$	↓ Rewrite the value of C on disk to 15
$< \texttt{START}\ T_3 >$	↑ Write $< \texttt{ABORT}\ T_3 >$ and flush the log
$< T_3, D, 19, 20 >$	↑ Rewrite the value of D on disk to 19
$< \texttt{END CKPT} >$	
$< \texttt{COMMIT}\ T_2 >$	↓ Stop redoing T_2

[a] The actions shown in black color are performed during the redo-style recovery, and the actions shown in blue color are performed during the undo-style recovery

Recovery with a Checkpointed Undo/Redo Log (Cont'd)

Example (Recovery with a Checkpointed Undo/Redo Log 3)

Log	Action[a]
$<$ START T_1 $>$	
$< T_1, A, 4, 5 >$	
$<$ START T_2 $>$	
$<$ COMMIT T_1 $>$	
$<$ START T_3 $>$	↑ Write $<$ ABORT T_3 $>$ and flush the log
$< T_2, B, 9, 10 >$	
$<$ START CKPT (T_2, T_3) $>$	↓ Start redoing T_2
$< T_2, C, 14, 15 >$	↓ Rewrite the value of C on disk to 15
$< T_3, D, 19, 20 >$	↑ Rewrite the value of D on disk to 19
$<$ END CKPT $>$	
$<$ COMMIT T_2 $>$	↓ Stop redoing T_2

[a] The actions shown in black color are performed during the redo-style recovery, and the actions shown in blue color are performed during the undo-style recovery

10.4 Archiving

Archiving (转储)

Archive: a copy of the database separate from the database itself

Types of Archiving

- Full Dump (全量备份): The entire database is copied
- Incremental Dump (增量备份): Only those database elements changed since the previous full or incremental dump are copied

Nonquiescent Archiving (不停机转储)

Example (Nonquiescent Archiving)

Step	Action	D_A	D_B	D_C	D_D	Archive	Log
1							$<$ START DUMP $>$
2		1	2	3	4	$A = 1$	$<$ START CKPT$(T_1, T_2) >$
3	A := 5	5	2	3	4		$< T_1, A, 1, 5 >$
4		5	2	3	4	$B = 2$	
5	C := 6	5	2	6	4		$< T_2, C, 3, 6 >$
6							$<$ COMMIT T_2 $>$
7		5	2	6	4	$C = 6$	
8	B := 7	5	7	6	4		$< T_1, B, 2, 7 >$
9	D := 8	5	7	6	8		$< T_1, D, 4, 8 >$
10		5	7	6	8	$D = 8$	$<$ END CKPT $>$
11		5	7	6	8		$<$ END DUMP $>$

Recovery using an Archive and a Log

Example (Recovery using an Archive and a Log)

Log	D_A	D_B	D_C	D_D	Action[a]
< START DUMP >	1	2	6	8	
< START CKPT (T_1, T_2) >	1	2	6	8	
$< T_1, A, 1, 5 >$	1	2	6	8	↓ Restore A to 1
$< T_2, C, 3, 6 >$	1	2	6	8	↑ Rewrite C to 6
< COMMIT $\mathtt{T}_2$ >	1	2	6	8	
$< T_1, B, 2, 7 >$	1	2	6	8	↓ Restore B to 2
$< T_1, D, 4, 8 >$	1	2	6	4	↓ Restore D to 4
< END CKPT >	1	2	6	4	
< END DUMP >	1	2	6	4	

[a] The actions shown in black color are performed during the redo-style recovery, and the actions shown in blue color are performed during the undo-style recovery

10.5 The ARIES Recovery Algorithm

ARIES

ARIES is designed to work with a steal no-force approach

1. Analysis Phase: Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash
2. Redo Phase: Repeats ALL actions[4], starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash
3. Undo Phase: Undoes the actions of transactions that did not commit, so that the database reflects only the actions of cormmitted transactions

[4] In undo/redo logging, only the actions of the committed transactions are undone

Three Main Principles behind ARIES

1. Write-Ahead Logging (WAL)
2. Repeating History During Redo: On restart following a crash, ARIES retraces ALL actions of the DBMS before the crash and brings the systern back to the exact state that it was in at the time of the crash. Then, it undoes the actions of transactions still active at the time of the crash
3. Logging Changes During Undo: Changes made to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated (failures causing) restarts

Structures Related to Recovery

1. Log (日志)
2. Transaction table (活跃事务表)
3. Dirty page table (脏页表)

Transaction Table

trxID	status	lastLSN
T1	in progress	103
T2	in progress	102

Dirty Page Table

pageID	recLSN
P500	100
P505	103
P600	101

Log

LSN	prevLSN	trxID	type	pageID	length	offset	old-value	new-value
100		T1	update	P500	3	21	ABC	DEF
101		T2	update	P600	3	40	HIJ	KLM
102	101	T2	update	P500	3	20	GDE	QRS
103	100	T1	update	P505	3	20	TUV	WXY

Log

- Log Sequence Number (LSN): Every log record is given a unique id called the log sequence number (LSN), which should be assigned in monotonically increasing order
- Every log record has certain fields:
  - `LSN`: the LSN of the log record
  - `prevLSN`: the LSN of the previous log record for the transaction
  - `trxID`: the id of the transaction generating the log record
  - `type`: the type of the log record
- pageLSN: For recovery purposes, every page in the database contains the LSN of the most recent log record that describes a change to this page. This LSN is called the pageLSN

LSN	prevLSN	trxID	type	pageID	length	offset	old-value	new-value
100		T1	update	P500	3	21	ABC	DEF
101		T2	update	P600	3	40	HIJ	KLM
102	101	T2	update	P500	3	20	GDE	QRS
103	100	T1	update	P505	3	20	TUV	WXY

Types of Log Records

- Update Log Records (更新日志记录): $< T, X, v, w >$
- Commit Log Records (提交日志记录): $<$ `COMMIT` $T >$
- Abort Log Records (中止日志记录): $<$ `ABORT` $T >$
- End Log Records (终止日志记录)
- Compensation Log Records (CLR, 补偿日志记录)

Types of Log Records

Update Log Records (更新日志记录)

- After modifying a page, an update log record is appended to the log tail in the log buffer
  - `pageID`: the id of the modified page
  - `length`: the length in bytes of the change
  - `offset`: the offset of the change
  - `old-value`: the value of the changed bytes before the change
  - `new-value`: the value of the changed bytes after the change
- The pageLSN of the page is set to the LSN of the update log record
- The page must be pinned in the buffer pool while these actions are carried out

LSN	prevLSN	trxID	update	pageID	length	offset	old-value	new-value

Types of Log Records (Cont'd)

Commit Log Records (提交日志记录)

- When a transaction decides to commit, it force-writes a commit log record to stable storage
- The transaction is considered to have committed at the instant that its commit log record is written to stable storage
- Some additional steps must be taken following the writing of the commit log record, e.g., removing the transaction's entry in the transaction table

LSN	prevLSN	trxID	commit

Types of Log Records (Cont'd)

Abort Log Records (中止日志记录)

- When a transaction is aborted, an abort log record containing the transaction id is appended to the log
- Undo is initiated for this transaction

LSN	prevLSN	trxID	abort

End Log Records (终止日志记录)

- When a transaction is aborted or committed, some additional actions must be taken beyond writing the abort or commit log record
- After all these additional steps are completed, an end log record containing the transaction id is appended to the log

LSN	prevLSN	trxID	end

Types of Log Records (Cont'd)

Compensation Log Records (补偿日志记录)

- Just before the change recorded in an update log record U is undone (when the transaction writing U is aborted or during recovery from a crash), a compensation log record C is appended to the log tail, which describes the action taken to undo the actions recorded in U
- The CLR C also contains a field called `undoNextLSN`, which is is set to the value of `prevLSN` in U
- Unlike an update log record, a CLR describes an action that will never be undone (why?)
- The number of CLRs that can be written during Undo is no more than the number of update log records for active transactions at the time of the crash

LSN	prevLSN	trxID	CLR	pageID	length	offset	old-value	undoNextLSN

Transaction Table (活跃事务表)

- The transaction table contains one entry for each active transaction:

  `(trxID, status, lastLSN)`

  - `trxID`: the id of the transaction
  - `status`: the status of the transaction (in-progress, committed, or aborted)
  - `lastLSN`: the LSN of the most recent log record for this transaction
- The entry for a committed or aborted transaction will be removed from the transaction table once certain "clean up" steps are completed
- The transaction table is held in main memory
- During normal execution, the transaction table is maintained by the transaction manager
- During recovery after a crash, the transaction table is reconstructed in the analysis phase

Structures Related to Recovery

Transaction Table

trxID	status	lastLSN
T1	in progress	103
T2	in progress	102

Dirty Page Table

pageID	recLSN
P500	100
P505	103
P600	101

Log

LSN	prevLSN	trxID	type	pageID	length	offset	old-value	new-value
100		T1	update	P500	3	21	ABC	DEF
101		T2	update	P600	3	40	HIJ	KLM
102	101	T2	update	P500	3	20	GDE	QRS
103	100	T1	update	P505	3	20	TUV	WXY

Dirty Page Table (脏页表)

- The dirty page table contains one entry for each dirty page in the buffer pool

  `(pageID, recLSN)`

  - `pageID`: the id of the page
  - `recLSN`: the LSN of the first log record that caused the page to become dirty, which identifies the earliest log record that might have to be redone for this page during recovery from a crash
- The dirty page table is held in main memory
- During normal execution, the dirty page table is maintained by the buffer manager
- During recovery after a crash, the dirty table is reconstructed in the analysis phase

The Write-Ahead Logging Protocol

- Before writing a page to disk, all log records up to and including the one with LSN equal to the pageLSN of the page to stable storage
- When a transaction is committed, the log tail (the most recent portion of the log kept in main memory) is forced to stable storage, even if a no-force approach is being used

Checkpointing

- The transaction table and the dirty page table are kept in main memory, which will be lost at the time of a crash
- Goal: To reduce the amount of work to reconstruct the tables during restart, the DBMS takes checkpoints periodically
- Checkpointing in ARIES has the following steps
    1. A begin_checkpoint log record is written to indicate when the checkpoint starts
    2. An end_checkpoint log record is constructed, including in it the current contents of the transaction table and the dirty page table, and appended to the log
- While the end_checkpoint record is being constructed, the DBMS continues executing transactions and writing other log records

Recovery from a Crash: Analysis Phase

Task 1: The Analysis phase reconstructs the transaction table at the time of the crash

- Analysis scans the log forward until it reaches the end of the log
- If an end log record for a transaction T is encountered, the entry for T is removed from the transaction table because it is no longer active
- If a log record other than an end log record for a transaction T is encountered, an entry for T is added to the transaction table if it is not already in the table
  - The `lastLSN` field of the entry is set to the LSN of this log record
  - If the log record is a commit log record, the status of the transaction is set to *committed*, otherwise it is set to *in-progress* (indicating that it is to be undone)

Recovery from a Crash: Analysis Phase

Task 2: The Analysis phase constructs a dirty page table, which includes all dirty pages at the time of the crash but may also contain some (clean) pages that were written to disk

- Analysis scans the log forward until it reaches the end of the log
- If a redoable log record (update or CLR) affecting page P is encountered, and P is not in the dirty page table, an entry for P is inserted into the table
  - The `pageID` field of the entry is the page id of P
  - The `recLSN` field of the entry is the LSN of this log record. This LSN identifies the oldest change affecting page P that may not have been written to disk.

Recovery from a Crash: Redo Phase

Task: The Redo phase reapplies the updates of all transactions and brings the database to the same state at the time of the crash

- Where to start: Redo begins with the log record that has the smallest `recLSN` of all pages in the dirty page table (why?)
- Where to end: Redo scans forward until the end of the log
- Conditions for redoing: For each redoable log record (update or CLR) encountered, Redo reapplies the logged action unless one of the following conditions holds (why?)
  - The affected page is not in the dirty page table
  - The affected page is in the dirty page table, but the `recLSN` for the entry is greater than to the LSN of the log record being checked
  - The pageLSN stored on the affected page (which must be retrieved from disk to check this condition) is greater than or equal to the LSN of the log record being checked

Recovery from a Crash: Undo Phase

Task: The Undo phase undoes the actions of all imcomplete transactions, that is, transactions active at the time of the crash

- *ToUndo*: the set of `lastLSN` values for all incomplete transactions
- Undo repeatedly chooses the largest (the most recent) LSN value in the set *ToUndo* and processes[5] the log record, until $ToUndo = \emptyset$
  - Case 1 (the log record is a CLR): A CLR needs not to be undone. If the `undoNextLSN` value is not null, the `undoNextLSN` value is added to the set *ToUndo*; if the `undoNextLSN` value is null, an end log record for the transaction is written to the log because it is completely undone
  - Case 2 (the log record is update): A CLR is written to the log, and the action recorded in the update log record is undone. Then, the `prevLSN` value in the update log record is added to the set *ToUndo*
- When $ToUndo = \emptyset$, the recovery is now coplete, and the system can proceed with normal operations

[5] Why don't we say "undoes"?

Summary

1. 10.1 Transactions
2. 10.2 Overview of Failure Recovery
3. 10.3 Write-Ahead Logging (WAL)
   - 10.3.1 Undo Logging
   - 10.3.2 Redo Logging
   - 10.3.3 Undo/Redo Logging
4. 10.4 Archiving
5. 10.5 The ARIES Recovery Algorithm