Logs and Recovery

Lecture 8

ICS502 - Database Programming

Dr. Eng. Amal Yassien

German International University in Cairo

Office: A.216

Office Hours: Monday 2nd (or by appointment via email)

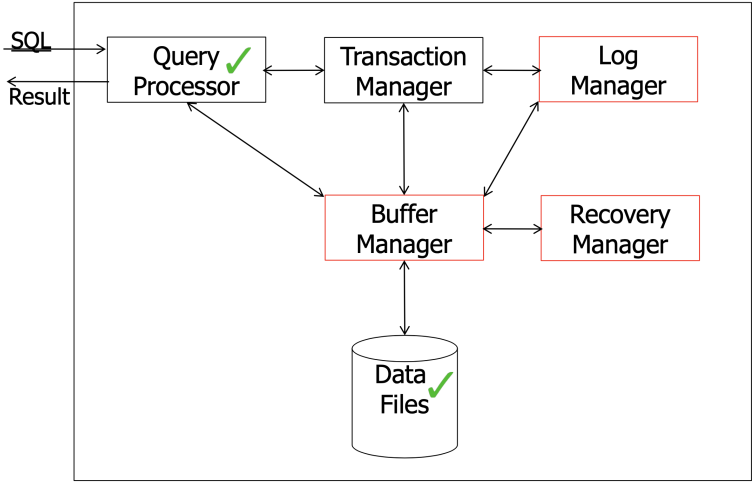
Email: amal.walied@giu-uni.de

Acknowledgment: These slides are based on the slides of Dr. Caroline Sabty, Dr. Wael Abouelsaadat and "Databases: The Complete Book - 2nd edition" by Prof. Gracia-Moline, Prof. Ullman, and Prof. Widom.

November 13, 2025

RDBMS Architecture

DBMS Architecture



Today's Agenda

- **Logs and Recovery - A Conceptual Overview**
- Undo Logging & Recovery
- Redo Logging & Recovery
- Undo/Redo Logging & Recovery

Toward Resilience and Durability in RDBMS

Coping with System Failure

In the face of system failures, RDBMS should:

1. adopt techniques that support *resilience* (i.e., integrity of data when the system fails)
2. ensure that data is not corrupted because of error-free queries or database modifications are being done at once

Toward Resilience and Durability in RDBMS

Coping with System Failure

In the face of system failures, RDBMS should:

1. adopt techniques that support *resilience* (i.e., integrity of data when the system fails) - Today's Topic
2. ensure that data is not corrupted because of error-free queries or database modifications are being done at once

What can go wrong? The **Expected** but *Undesired* Events

Erroneous Data Entry

Some data errors are impossible to detect. For example, if a clerk mistypes one digit of your phone number, the data will still look like a phone number that could be yours. On the other hand, if the clerk omits a digit from your phone number, then the data is evidently in error, since it does not have the form of a phone number. The principal technique for addressing data-entry errors is to write constraints and triggers that detect data believed to be erroneous.

Media Failures

A local failure of a disk, one that changes only a bit or a few bits, parity checks can be used for that (Media Courses), or major one, like head failures, where the disk is completely unreadable. To recover from that failure, database administrators aim to recover disk content, keep an *archive* of database content, or distribute several *copies* (Redundancy) of the database among several sites (copies need to be like the original!).

What can go wrong? The **Expected** but *Undesired* Events (Cont'd)

System Failure (what we are addressing today)

On the execution level, database modification commands and queries get translated to a series of steps which is called *transaction*. Each transaction has a *state* that indicates which steps have been executed and which has not, along of course with the modification this step entails. A system failure is when the *state* of a *transaction* is lost due to power loss or software errors.

What can go wrong? The **Unexpected** and *Undesired* Events

Catastrophic Failure

A catastrophic event in which the media holding the databases is completely destroyed (e.g. fires, or vandalism at the site of the database). In these events, archiving and redundancy of database copies across different sites come to the rescue.

Coping with System Failures: The Guarantees RDBMS Provide

Recall

A system failure is when the *state* of a *transaction* is lost due to power loss or software errors.

To correct system errors, how to ensure a transaction executed "Correctly"

To establish this, we need to ensure that a transaction modify database elements (e.g. tables, tuples, pages) in a way that transfers the database from an old "consistent" state to a new "consistent" state.

What is a consistent database state?

A state in which all database constraints are satisfied (e.g., schema related constraints, custom-constraints set by system designers).

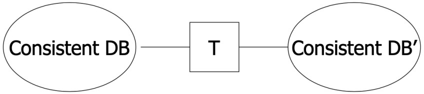
Coping with System Failures: The Correctness Principle

The Correctness Principle

If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transaction ends.

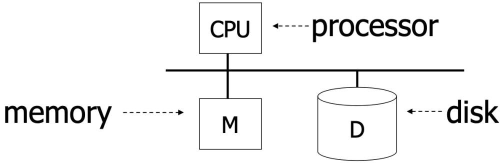
Enforcing Correctness - RDBMS ensures the following:

Atomicity	A transaction is either fully executed or not executed at all
Concurrency Control	Ensure that simultaneous modifications made by multiple transactions doesn't lead to any inconsistencies.



Flow of Information during Transaction Execution

Transactions are executed in the main memory relying on the CPU, but its effect or output needs to be stored in second storage to ensure durability of RDBMS systems.



Primitive Operations of Transactions

Main-Memory to Hard Disk

INPUT(X)	Copy the disk block containing database element X to a memory buffer
OUTPUT(X)	Copy the block containing X from its buffer to disk

Transaction Actions

READ(X,t)	Copy the database element X to the transaction's local variable t. More precisely, if the block containing database element X is not in a memory buffer then first execute INPUT(X). Next, assign the value of X to local variable t
WRITE(X,t)	Copy the value of local variable t to database element X in a memory buffer. More precisely, if the block containing database element X is not in a memory buffer then execute INPUT(X). Next, copy the value of t to X in the buffer. Then, the buffer should be saved to disk using OUTPUT(X).

Logs and Recovery: Key Components

Log

Records securely (we will see the protocols later ;)) the history of database changes.

Recovery

Uses the log to reconstruct what has happened to the database when there has been a failure.

Checkpointing

Limits the length of log that must be examined during recovery.

Logging Techniques

We shall discuss *three* logging styles:

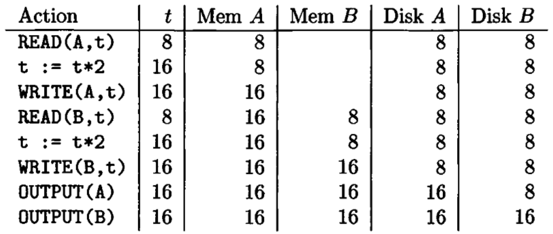
1. Undo Logging: which restores the database to its state "prior" to the system failure
2. Redo Logging: which re-executes the transactions that were not completed as a result of the failure (at least those whose full modifications were captured before failure!)
3. Undo/Redo Logging: which is a hybrid technique that capitalizes on the advantages of both undo and redo logging.

System Failure Impact on Executing Transactions State

Transaction T

Let T be a transaction that ensures that database element A is equal to database element B, while doing the below operations.

```
start transaction;
 update TableR set A = A * 2, B = B * 2;
end transaction;
```



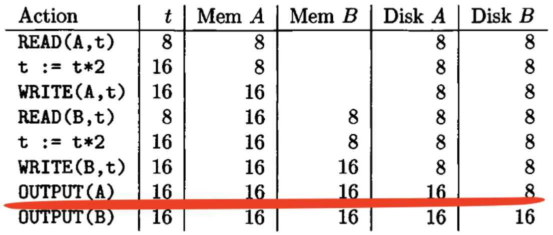
Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t := t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t := t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

System Failure Impact on Executing Transactions State (Cont'd)

Transaction T

Let T be a transaction that ensures that database element A is equal to database element B, while doing the below operations.

```
start transaction;
 update TableR set A = A * 2, B = B * 2;
end transaction;
```



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

Log Commands

<Start T>

log the start of transaction T

<T, X, value>

log that T (transaction) modified X (database element), the value being written defers based on the logging technique used.

<COMMIT T>

log the completion of transaction T. Note: recovery actions based on this command change based on the logging technique used.

Today's Agenda

- Logs and Recovery - A Conceptual Overview
- **Undo Logging & Recovery**
- Redo Logging & Recovery
- Undo/Redo Logging & Recovery

Undo Logging: The Approach

What is does?

undo any partial changes made by a transaction that was not completed before failure

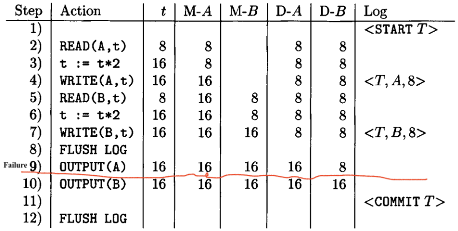
Log Records in Undo Logging

<T, X, v>	Transaction T modified database element X, and the the value of X before T modified it was v.

Undo Logging Steps

1. If transaction T modifies X, then the log record <T,X,v> must be written to disk before the new value of X is written to disk
2. If transaction T commits, then its COMMIT log record must be written to disk only after all database elements T modified written to disk

System Failure Scenario 1: 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						
Failure 9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

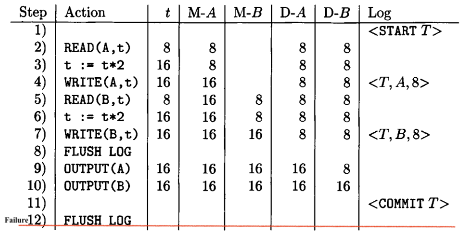
Flushed Log Content

<Start T>, <T, A, 8>, <T, B, 8>

Recovery Actions

Undo T, set A back to 8, set B back to 8

System Failure Scenario 2: 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>
Failure 12)	`FLUSH LOG`						

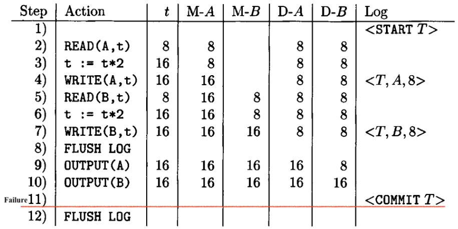
Flushed Log Content

<Start T>, <T, A, 8>, <T, B, 8>, <Commit T>

Recovery Actions

Do Nothing. Changes made by T were saved to disk before failure because it committed in log.

System Failure Scenario 3: 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	
Failure 11)							<COMMIT T>
12)	FLUSH LOG						

Flushed Log Content

<Start T>, <T, A, 8>, <T, B, 8>

Recovery Actions

Undo T, set A back to 8, set B back to 8, because <COMMIT T> was not flushed to log (log saved to disk) before failure

Undo Logging: The Approach

Log Records in Undo Logging

<T, X, v> Transaction T modified database element X, and the the value of X before T modified it was v.

Undo Logging Steps

1. If transaction T modifies X, then the log record <T,X,v> must be written to disk before the new value of X is written to disk
2. If transaction T commits, then its COMMIT log record must be written to disk only after all database elements T modified written to disk
3. A <C0MMIT T> record must be flushed to disk as soon as it appears in the log to minimize the frequency of Scenario 3.

Classical Checkpointing in Undo Logs

Motivation

During Recovery, the entire log is examined, in principle.

The Approach

Periodically place checkpoints in the log file to minimize the portion of log to be examined by recovery manager.

How Classical Checkpoints work?

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 transactions

Undo Logging and Recovery with Classical Checkpointing

Examples

The below shows the content of a log file, after a system failure, occurred. State how far in the log file does the recovery manager need to examine and the actions that the recovery manager would perform.

Solution

The recovery manager would start recovering after <CKPT> by undoing all actions made by T_3 because <COMMIT T_3> was not present in the log file.

```
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<T2, C, 15>
<T1, D, 20>
<COMMIT T1>
<COMMIT T2>
<CKPT>
<START T3>
<T3, E, 25>
<T3, F, 30>
```

Non-quiescent Checkpointing in Undo Logs

Motivation

Classical Checkpoints introduce bottlenecks in transaction execution flow.

The Approach

1. Write a log record <START CKPT($T_1, \ldots, T_k$)> and flush the log. Here, $T_1, \ldots, T_k$ are the names or identifiers for all the active transactions (i.e., transactions that have not yet committed and written their changes to disk).
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 <END CKPT> and flush the log.

Recovering using Undo Logs and Non-quiescent Checkpoints

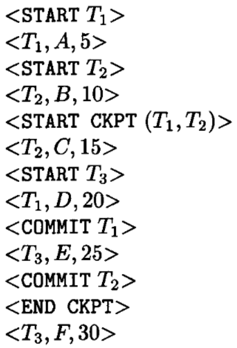
With a log of this type, we can recover from a system crash as follows. As usual, we scan the log from the end, finding all incomplete transactions as we go, and restoring old values for database elements changed by these transactions. While scanning backwards, we can meet either:

<END CKPT> If we first meet an <END CKPT> record, then we know that all incomplete transactions began after the previous <START CKPT $T_1, \ldots, T_k$ record. We may thus scan backwards as far as the next START CKPT, and then stop; previous log is useless and may as well have been discarded.

<START CKPT($T_1, \ldots, T_k$)> If we first meet a record <START CKPT($T_1, \ldots, T_k$)>, then the crash occurred during the checkpoint. However, the only incomplete transactions are those we met scanning backwards before we reached the START CKPT and those of $T_1, \ldots, T_k$ that did not complete before the crash. Thus, we need scan no further back than the start of the earliest of these incomplete transactions.

Undo Logging and Recovery using Non-quiescent Checkpointing - Case 1

After a system failure, these are the content of log files. When we scan the log backwards, we find <END CKPT>, which means all transactions within <Start CKPT> has committed. Therefore, the recovery manager scans the log till the <Start CKPT>. Then it would undo the changes made by T_3.



<START T_1>
<$T_1, A, 5$>
<START T_2>
<$T_2, B, 10$>
<START CKPT (T_1, T_2)>
<$T_2, C, 15$>
<START T_3>
<$T_1, D, 20$>
<COMMIT T_1>
<$T_3, E, 25$>
<COMMIT T_2>
<END CKPT>
<$T_3, F, 30$>

Undo Logging and Recovery using Non-quiescent Checkpointing - Case 2

After a system failure, these are the content of log files. When we scan the log backwards, we find <Start CKPT(T_1, T_2)>, this means that while the checkpoint has been placed T_1, T_2 didn't commit yet. So the recovery manager scans the log till the <Start T_x>, where x is the incomplete transaction that started the earliest, which is T_2, and undoes the changes made by T_2, *and* T_3, as neither of them committed in log.

<START T_1>
<$T_1, A, 5$>
<START T_2>
<$T_2, B, 10$>
<START CKPT (T_1, T_2)>
<$T_2, C, 15$>
<START T_3>
<$T_1, D, 20$>
<COMMIT T_1>
<$T_3, E, 25$>

Undo Logging: *The Cons*

Immediate backup of database elements to disk

Undo logging has a potential problem that we cannot commit a transaction without first writing all its changed data 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. As long as there is a log to fix things up in the event of a crash, it is safe to do so.

Today's Agenda

- Logs and Recovery - A Conceptual Overview
- Undo Logging & Recovery
- **Redo Logging & Recovery**
- Undo/Redo Logging & Recovery

Redo Logging: The Approach

What it does?

Uses write-ahead logging where all the actions and modification a transaction would make are written in the log first and then the actual database elements would be updated on disk afterwards.

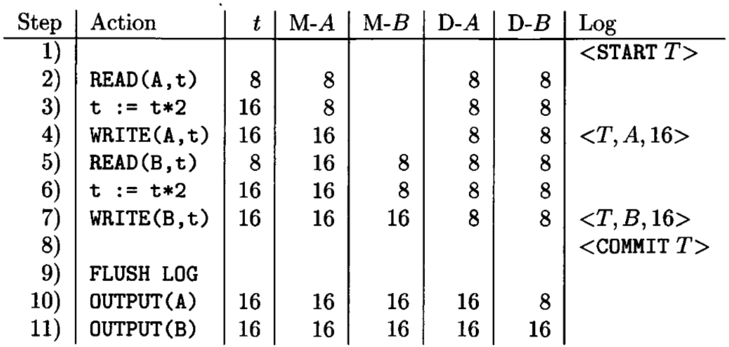
Log Records

In redo logging the meaning of a log record <T, X, v> is “transaction T wrote new value v for database element X”.

Redo Logging Steps

- Before modifying any database element X on disk, it is necessary that all log records pertaining to this modification of X, including both the update record <T,X,v> and the <COMMIT T> record, must appear on disk.

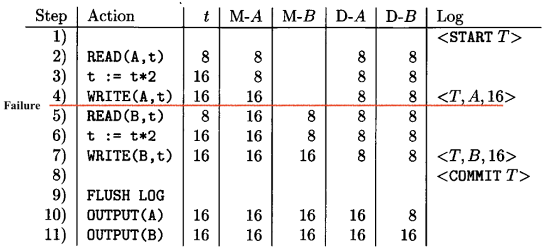
Redo Logging Example 1



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 Failure Scenario 1: Redo Logging

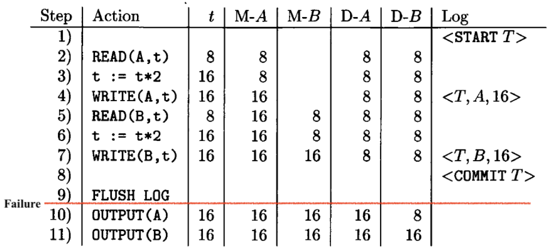
When the recovery manager inspects the log, it would see that there is nothing written in log regarding T. The state of database remains unchanged as all modifications T would make will become effective when it commits in log. Therefore, the recovery manager will ABORT T.



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>
Failure							
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 Failure Scenario 2: Redo Logging

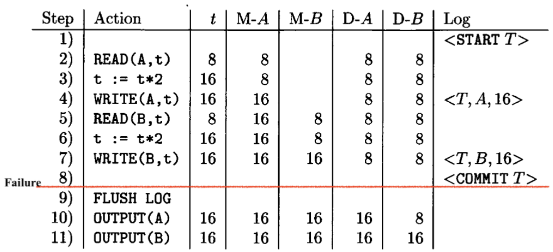
Here the failure happened after the log has been flushed to disk. Therefore, when the recovery manager looks at the log it would see <Start T>, <T, A, 16>, <T, B, 16>, <Commit T>. Therefore, it would know that all modifications T will make have been fully captured before the failure and will REDO T.



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`						
Failure							
10)	`OUTPUT(A)`	16	16	16	16	8	
11)	`OUTPUT(B)`	16	16	16	16	16	

System Failure Scenario 3: Redo Logging

When the recovery manager inspects the log, it would see that there is nothing written in log regarding T. The state of database remains unchanged as all modifications T would make will become effective when it commits in log and log is written to disk, which at this point, we are not sure it was written in time before the failure. Therefore, the recovery manager will ABORT T.



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	<$T, B, 16$>
Failure 8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

Recovery Steps using Redo Logging

To recover, using a redo log, after a system crash, we do the following:

1. Identify the committed transactions
2. Scan the log forward from the beginning. For each log record <T,X,v> encountered:
   - 2.1 If T is not a committed transaction, do nothing (ABORT)
   - 2.2 If T is committed, write value v for database element X (REDO)
3. For each incomplete transaction T, write an <ABORT T> record to the log and flush the log

Order of Redos

With redo logging, we focus on the committed transactions, as these need to be redone. It is quite normal for there to be two committed transactions, each of which changed the same database element at different times. Thus, order of redo is always important. Think which final value should be assigned to the database element?

Non-quiescent Checkpointing in Redo Logging

The Steps

1. Write a log record <START CKPT ($T_1, \ldots, T_k$)>, where $T_1, \ldots, T_k$ are all the active (uncommitted) transactions, and flush the log.
2. Write to disk all 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 to the log and flush the log.

Recovery using Redo Logging and Non-quiescent Checkpointing

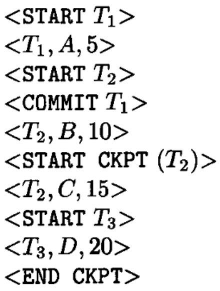
When a crash happens, the recovery manager might either of the below cases while scanning the log file backwards:

<END CKPT> Now, we know that every value written by a transaction that committed before the corresponding <START CKPT $(T_1, \ldots, T_k)$> has had its changes written to disk, so we need not concern ourselves with recovering the effects of these transactions. However, any transaction that is either among the T_i's is or that started after the beginning of the checkpoint can still have changes it made not yet migrated to disk, even though the transaction has committed.

<Start CKPT$(T_1, \ldots, T_k)$> We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk. Thus, we must search back to the previous <END CKPT> record, find its matching <START CKPT $(S_1, \ldots, S_n)$> record, and redo all those committed transactions that either started after that START CKPT or are among the S_i's.

Redo Logging and Recovery using Non-quiescent Checkpointing - Case 1

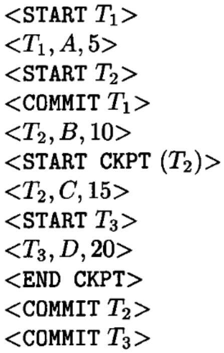
suppose that the crash occurs just prior to the <END CKPT> record. In principal, we must search back to the next-to-last START CKPT record and get its list of active transactions. However, in this case there is no previous checkpoint, and we must go all the way to the beginning of the log. Thus, we identify T_1 as the only committed transaction, redo its action $<T_1$,A, 5>, and write records <ABORT T_2> and <ABORT T_3> to the log after recovery.



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

Redo Logging and Recovery using Non-quiescent Checkpointing - Case 2

If a crash occurs at the end, we search backwards, finding the <END CKPT> record. We thus know that recovery actions will be related to redoing all those transactions that either started after the <START CKPT (T_2)> record was written or that are on its list. Thus, our candidate set is T_2, T_3. We find the records <COMMIT T_2> and <COMMIT T_3>, so we know that each must be redone. So, we rewrite the values 10, 15, and 20 for B, C, and D, respectively.



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

Undo Logging & Redo Logging: Can we do better?

Undo Logging Drawback

(1) Data is required to be written to disk immediately after a transaction finishes, perhaps increasing the number of disk I/O's that need to be performed. (2) Database is restored to its old consistent state rather than staying up-to-date.

Redo Logging Drawback

Keeping all modified blocks in buffers until the transaction commits and the log records have been flushed is required, perhaps increasing the average number of buffers required by transactions.

Undo/Redo Logging

Provides increased flexibility to order actions, at the expense of maintaining more information on the log.

Today's Agenda

- Logs and Recovery - A Conceptual Overview
- Undo Logging & Recovery
- Redo Logging & Recovery
- **Undo/Redo Logging & Recovery**

Undo/Redo Logging: The Approach

What it does?

Logs both the old value of a database element prior to its modification by a certain transaction along with the value the transaction will assign to this database element.

Log Records

Record <T,X,v,w> means that transaction T changed the value of database element X; its former value was v, and its new value is w.

Undo/Redo Steps

1. Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update record <T,X,v,w> appear on disk.
2. A <COMMIT T> record must be flushed to disk as soon as it appears in the log.

Undo/Redo Recovery Steps

When we need to recover using an undo/redo log, we have the information in the update records either to undo a transaction T by restoring the old values of the database elements that T changed, or to redo T by repeating the changes it has made. The undo/redo recovery policy is:

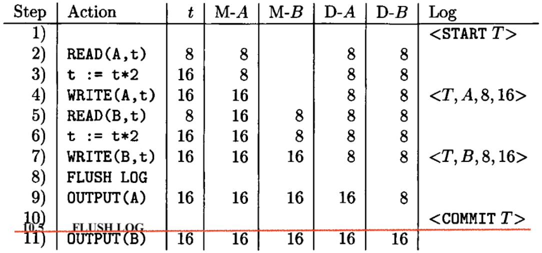
1. Redo all the committed transactions in the order earliest-first, and
2. Undo all the incomplete transactions in the order latest-first

Flexibility of Undo/Redo regarding Commit

Because it stores both the old and new value, the <COMMIT T> log record can precede or follow any of the changes to the database elements on disk.

System Failure Scenario 1: Undo/Redo Logging

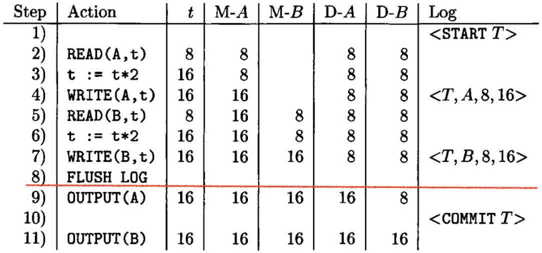
Suppose the crash occurs after the <COMMIT T> record is flushed to disk. Then T is identified as a committed transaction. We write the value 16 for both A and B to the disk.



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>
10.5	FLUSH LOG						
11)	OUTPUT(B)	16	16	16	16	16	

System Failure Scenario 2: Undo/Redo Logging

If the crash occurs prior to the <COMMIT T> record reaching disk, then T is treated as an incomplete transaction. The previous values of A and B, 8 in each case, are written to disk.



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	

Non-quiescent Checkpointing in Undo/Redo Logging

The Steps

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

Undo/Redo Logging using Non-quiescent Checkpointing - Case 1

Suppose the crash occurs just before the <COMMIT T_3> record is written to disk. Then we identify T_2 as committed but T_3 as incomplete. We redo T_2 by setting C to 15 on disk; it is not necessary to set B to 10 since we know that change reached disk before the <END CKPT>. However, unlike the situation with a redo log, we also undo T_3; that is, we set D to 19 on disk.

<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>

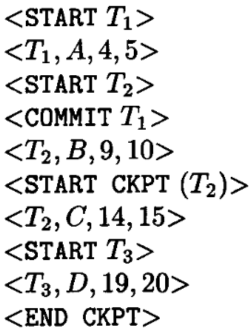
Undo/Redo Logging using Non-quiescent Checkpointing - Case 2

T_2 and T_3 are identified as committed transactions. Transaction T_1 is prior to the checkpoint. Since we find the <END CKPT> record on the log, T_1 is correctly assumed to have both completed and had its changes written to disk. We therefore redo both T_2 and T_3. However, when we redo a transaction such as T_2, we do not need to look prior to the <START CKPT (T_2)> record, even though T_2 was active at that time, because we know that T_2's changes prior to the start of the checkpoint were flushed to disk during the checkpoint.

<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>

Undo/Redo Logging using Non-quiescent Checkpointing - Case 3

Here the crash occurs before T_2 and T_3 commit. Once the log file is scanned backwards, we will meet <END CKPT>, which means that all changes made by committed transaction T_1 were written to disk. We go the matching <START CKPT(T_2)>, we see that T_2 started prior to <START CKPT(T_2)>, so we go and scan the log forward starting <Start T_2> and undo changes made T_2 before and after <Start CKPT(T_2)> and undo all changes made by T_3.



<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>

Takeaways I

- Logging and recovery is an important feature of a database engine which enables it to protect the integrity of the data from undesired expected events.
- Logging involves writing down all changes made by a transaction before actually changing values on disk. The log must be written first to disk.
- There are three logging techniques supported by database engines; undo, redo and undo/redo. Each has its own rules and own recovery policy.

Takeaways II

Logging Technique	**When DB Element Can Be Updated on Disk?**	**Recovery Policy**
Undo Logging	When $\langle T, X, v \rangle$ is written to the log and the log is flushed to disk	Undo uncommitted transactions; do nothing for committed transactions
Redo Logging	When $\langle \text{Commit } T \rangle$ is written to the log and the log is flushed to disk	Redo all committed transactions; Abort all uncommitted transactions (careful of order)
Undo/Redo Logging	When $\langle T, X, v, w \rangle$ is written to the log and the log is flushed to disk	Undo all uncommitted transactions; redo all committed transactions (careful of order)

Table: Comparison of Logging Techniques in Database Recovery.

Takeaways III

- We discussed two checkpointing techniques:

  Classical Checkpoints
  : Queues all new transactions until all currently executing transactions commit in log

  Non-quiescent Checkpoints
  : Allows new transactions to start by periodically keeping track of currently active transactions.

- For each of the *three* logging techniques we covered today, the non-quiescent checkpoints placement process is adapted to each logging technique's rules and recovery policies.

Next Time

- Transactions I

Thank You
Vielen Dank

Dr. Eng. Amal Yassien

amal.walied@giu-uni.de

A.216