Transaction Processing
Recovery

Hans Philippi

December 4, 2025

Crash recovery in databases

Sun Tzu:
To ... not prepare [for war] is the greatest of crimes; to be prepared beforehand for any contingency is the greatest of virtues
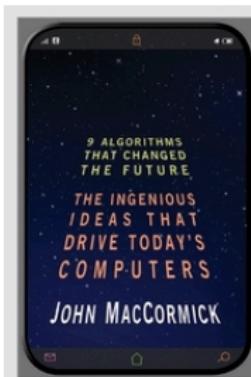
Crash recovery in databases

- Data might get lost
- We don't like that, because . . .
- Solutions are based on logging techniques
- General term: *write ahead logging*

Write ahead logging is mentioned as one of the algorithms in:

Nine Algorithms That Changed the Future: The Ingenious Ideas That Drive Today's Computers
John MacCormick
With a foreword by Chris Bishop

Honorable Mention for the 2012 Award for Best Professional/Scholarly Book in Computing & Information Sciences, Association of American Publishers

Paperback | 2013 | **$16.95** | **£11.95** | ISBN: 9780691158198
232 pp. | 6 x 9 | 5 halftones. 98 line illus. 1 table.
Add to Shopping Cart
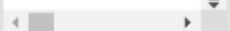
eBook | ISBN: 9781400839568 |
Our eBook editions are available from these online vendors

(Princeton L & L Lecture)

Crash recovery in databases

- We will focus on crashes where loss of main memory occurs
- We will also have a short look on situations involving loss of data on disk

Recall: the ACID properties

- Atomicity
- Consistency
- Isolation
- Durability

ACID proerties

- Atomicity: transactions must either be completely finished (commit) or completely rolled back (abort)
- Consistency: each transaction must obey all integrity constraints; if not, it must be aborted
- Isolation: the concurrent execution of transactions may not be visible; transactions should be processed as if they run exclusively on the database
- Durability; the persistence of data that is updated by a committed transaction must be guaranteed

Architectural model
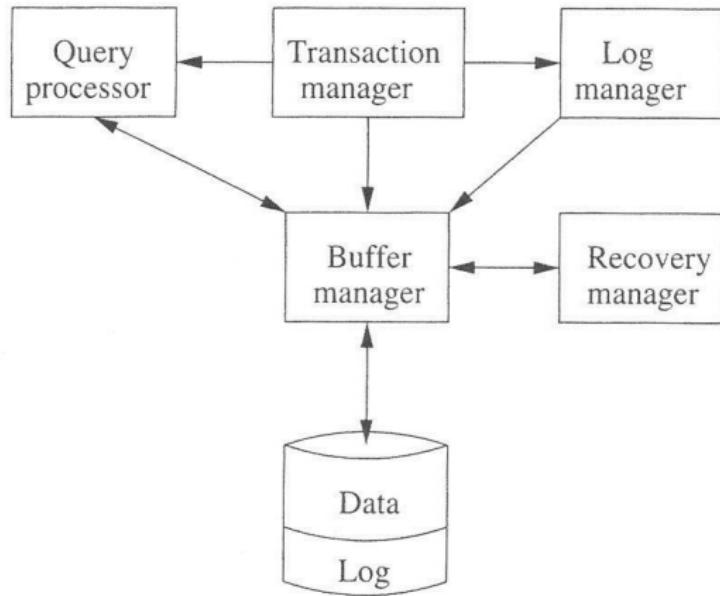
illustration from: *Database Systems, the Complete Book*

Architectural model: buffer manager

- IO: traffic disk $\leftrightarrow$ memory

- Unit size: page or block

- Typically 8-32 kbyte

- Access time: 5 - 10 msec

- A block copied in memory is called *buffered*
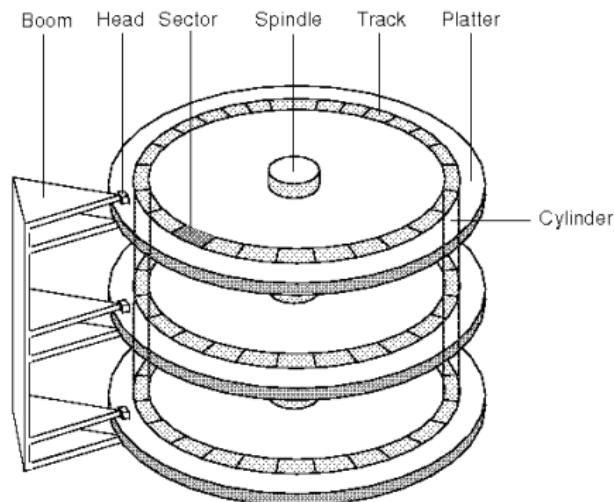
illustration from: *Database Systems, the Complete Book*

Architectural model: log manager

- Log file is a separate file, containing information to support database reconstruction
- Entries have a record structure
- Entries are (in most cases) related to a specific transaction
- Good practice: log file on separate disk
- Recent development: log files on SSD

Main memory vs disk: buffer manager

- INPUT(X): copy block X from disk to memory
- READ(X,t): assign value of X to variable t; if necessary, this implicitly forces an INPUT(X)
- WRITE(X,t): copy value of t into X in memory
- OUTPUT(X): copy block X from memory to disk . . .
- . . . also called flushing X
- Example (financial transaction):

```
INPUT(Account1); READ(Account1, v1); v1 := v1 - 200;
WRITE(Account1, v1); OUTPUT(Account1);
INPUT(Account2); READ(Account2, v2); v2 := v2 + 200;
WRITE(Account2, v2); OUTPUT(Account2);
```

Partial execution by failure:

```
INPUT(Account1); READ(Account1, v1);
v1 := v1 - 200;
WRITE(Account1, v1); OUTPUT(Account1);
```

>>> CRASH <<<

```
INPUT(Account2); READ(Account2, v2);
v2 := v2 + 200;
WRITE(Account2, v2); OUTPUT(Account2);
```

Problem 1: crash on the fly

Partial execution by failure:

```
INPUT(Account1); READ(Account1, v1);
v1 := v1 - 200;
WRITE(Account1, v1); OUTPUT(Account1);
```

$>>>$ CRASH $<<<$

```
INPUT(Account2); READ(Account2, v2);
v2 := v2 + 200;
WRITE(Account2, v2); OUTPUT(Account2);
```

Solution:

- Rollback transaction (undo partial changes)
- As soon as possible: restart transaction

Principles of UNDO logging

- Old values of each data element X must be written to the log file: `<T, X, oldvalue>`
- T denotes the transaction identifier
- `oldvalue` is often called the *before image* of X
- Before doing an OUTPUT(X), the log record for this X must be flushed
- The `<T, commit>` record is written to the log after all database elements have been updated on disk

Step	Action	t	M-A	M-B	D-A	D-B	Log
1	Start T				3	7	$<$START T$>$

- Transaction T will double the values of A and B
- D-A and D-B are the values of A and B on disk
- M-A and M-B are the values of A and B in memory
- t is a variable in the programming environment
- Log entries initially are collected in memory
- Action FLUSH LOG forces the output of the log entries to the log file on disk

Step	Action	t	M-A	M-B	D-A	D-B	Log
1	Start T				3	7	<START T>
2	READ(A,t)	3	3		3	7	
3	t = t*2	6	3		3	7	
4	WRITE(A,t)	6	6		3	7	<T, A, 3>
5	READ(B,t)	7	6	7	3	7	
6	t = t*2	14	6	7	3	7	
7	WRITE(B,t)	14	6	14	3	7	<T, B, 7>
8	FLUSH LOG	14	6	14	3	7	>>> log
9	OUTPUT(A)	14	6	14	6	7	
10	OUTPUT(B)	14	6	14	6	14	
11	Commit T	14	6	14	6	14	<COMMIT T>
12	FLUSH LOG	14	6	14	6	14	> log

- Real world effects of commitment may take place after step 12

Step	Action	t	M-A	M-B	D-A	D-B	Log
1	Start T				3	7	<START T>
2	READ(A,t)	3	3		3	7	
3	t = t*2	6	3		3	7	
4	WRITE(A,t)	6	6		3	7	<T, A, 3>
5	READ(B,t)	7	6	7	3	7	
6	t = t*2	14	6	7	3	7	
7	WRITE(B,t)	14	6	14	3	7	<T, B, 7>
8	FLUSH LOG	14	6	14	3	7	>>> log
9	OUTPUT(A)	14	6	14	6	7	
10	OUTPUT(B)	14	6	14	6	14	
11	Commit T	14	6	14	6	14	<COMMIT T>
12	FLUSH LOG	14	6	14	6	14	> log

- Step 8 must be done before steps 9 and 10 to enable undoing!
- Steps 9 and 10 must be done before step 12!

Recovery using UNDO logging

- Check the log file for uncommitted transactions
- Rollback these transactions using the before images
- The order of undoing transactions is essential
- By the way: restart the transactions that are rolled back

- What about ... a crash during the recovery process?

Checkpointing

- Dealing with a complete transaction log since the DB became operational (possibly a few years ago) may be less desirable
- Solution: regular checkpointing, for instance every night or every hour
- Only unfinished transactions since last checkpoint must be rolled back
- Checkpoint steps:
    - No new transactions accepted, for the moment
    - Ensure that all active transactions are finished (COMMIT or ABORT) and the log records have been flushed
    - Write a $<$CKPT$>$ record into the log and flush the log
    - Resume transaction processing

Problem 2: loss of committed data

```
INPUT(Account1); READ(Account1, v1);
v1 := v1 - 200;
WRITE(Account1, v1);
INPUT(Account2); READ(Account2, v2);
v2 := v2 + 200;
WRITE(Account2, v2);
COMMIT;

>>> CRASH <<<

OUTPUT(Account1);
OUTPUT(Account2);
```

But why would we want to commit before having flushed all
written data?

Loss of committed data!

- Note that processing speed and memory access are several orders of magnitude faster than disk access
- If the disk arm just left in another direction, it might take a long time before it visits us again ...
- ... while we are so eager to commit and release the locks!

Principles of REDO logging

- New values of each data element X will be written to the log file: `<T, X, newvalue>`
- T denotes the transaction identifier
- `newvalue` is often called the *after image* of X
- After logging (and flushing) all after images, the `<T, commit>` record is flushed to the log

Step	Action	t	M-A	M-B	D-A	D-B	Log
1	Start T				3	7	<START T>

- Transaction T will double the values of A and B
- D-A and D-B are the values of A and B on disk
- M-A and M-B are the values of A and B in memory
- t is a variable in the programming environment
- Log entries are initially collected in memory
- FLUSH LOG forces the output of the log entries to the log file on disk

Step	Action	t	M-A	M-B	D-A	D-B	Log
1	Start T				3	7	<START T>
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							

REDO logging: example of transaction run

Step	Action	t	M-A	M-B	D-A	D-B	Log
1	Start T				3	7	<START T>
2	READ(A,t)	3	3		3	7	
3	t = t*2	6	3		3	7	
4	WRITE(A,t)	6	6		3	7	<T, A, 6>
5	READ(B,t)	7	6	7	3	7	
6	t = t*2	14	6	7	3	7	
7	WRITE(B,t)	14	6	14	3	7	<T, B, 14>
8	Commit T	14	6	14	3	7	<COMMIT T>
9	FLUSH LOG	14	6	14	3	7	>>>>> log
10	OUTPUT(A)	14	6	14	6	7	
11	OUTPUT(B)	14	6	14	6	14	

- Steps 4 and 7 must be done before step 9
- Real world effects may take place after step 9

Recovery using REDO logging

- Check the log file for committed transactions
- Redo these transactions using the after images
- The order of redoing transactions is essential
- Of course, we can apply checkpointing in this case as well

- What about ... a crash during the recovery process?

Combined UNDO-REDO logging

- Log both before image and after image:
  <T, X, oldvalue, newvalue>
- This approach gives optimal freedom to the buffer manager
  - The UNDO logging gives the buffer manager the freedom to flush new data before the COMMIT
  - The REDO logging gives the buffer manager the freedom to flush new data after the COMMIT
- Flushing the <T, X, oldvalue, newvalue> record must preceed both writing the data and committing the transaction

Failure of stable storage

- Up till now, we have discussed system failures:
  - Memory is lost; data on disk is ok
- What to do in case of disk failure?
- Archiving: always keep a copy of your entire database
- Full dump or a series of incremental dumps
- Keep logs since last dump
- Archive copy + log = actual database
- ... that is why you *must* keep your log file on a separate disk!

Inference between transactions

- Care must be taken with respect to recoverability issues due to possible interference between transactions
- Look at this scenario (with UNDO recovery):

T1	T2
W(x)	
	R(x)
	COMMIT
ABORT	

- Obviously it is risky to read uncommitted values

- *Definition "reads from"*: T_j RF T_i if there is an X such that $W_i(X)$ is the last write on X before $R_j(X)$
- *Definition*:   if T_j RF T_i and T_i is not yet committed, then the corresponding read by T_j is called a *dirty read*
- *Definition*:   a schedule is *recoverable (RC)* if for each pair of transactions the following property holds:

  T_j RF T_i $\implies$ COMMIT$_i$ < COMMIT$_j$

- The symbol $<$ denotes time order
- In other words: a schedule is *RC* if it does not contain dirty reads

- Uncommitted data is often called *dirty data*

More interference issues

- Recoverability ensures a minimum level of safety, but is not without troubles
- The schedule below is RC, but leads to cascading aborts
- Time order is from left to right

$$W_1(x)\ R_2(x)\ W_2(y)\ R_3(y)\ W_3(z)\ R_4(z)\ ABORT_1$$

- Note that abortion of transactions T_2, T_3 and T_4 must be enforced
- We are looking for a stricter notion than RC to prevent cascading aborts

- *Definition:* A schedule is *strict (ST)* if: [3]

  $W_i(X) < O_j(X) \implies$

  $COMMIT_i < O_j(X)$ or $ABORT_i < O_j(X)$

  for each read or write operation O_j by T_j on X

- Implementation of strictness can be combined with 2PL: a transaction must hold all its write locks until COMMIT/ABORT

- The notions of RC and ST are related to the SQL Isolation Levels

---

[3]The symbol $<$ denotes time order

SQL Isolation Levels

Most DBMS's provide the possibility to choose between several levels of isolation

- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
  - Corresponds to strict serializability
  - Increases the probability of worse performance and deadlocks
- SET TRANSACTION ISOLATION LEVEL READ COMMITTED
  - Guarantees RC
- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
  - No restriction; allows dirty reads
  - Application: creating snapshots for data warehouse construction ...
  - ... 100% consistency of data is not required for statistical data analysis