Transaction Management

Transaction Management

- Multiple clients accesses the same buffer concurrently can cause chaos:
  - One client can see different (and even inconsistent) values of a page each time
  - Two clients can unwittingly overwrite the values of each other
- A database system maintains order and ensure database integrity:
  - Recovery manager
    - Reads and writes records to the log
  - Concurrency manager
    - Regulates the execution of these transactions

Transaction Concept

- A transaction is a group of operations that behaves as a single operation.
- E.g., transaction to transfer $50 from account A to account B:

1. **read**(A)
2. $A := A – 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)

- **Atomicity requirement:**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
  - The system should ensure that updates of a partially executed transaction are not reflected in the database

Example of Fund Transfer (Cont.)

- Transaction to transfer $50 from account A to account B:
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)

- **Consistency requirement:**
  - A transaction, when starting to execute, must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent.
  - The sum of A and B is unchanged by the execution of the transaction

Example of Fund Transfer (Cont.)

- Transaction to transfer $50 from account A to account B:

	T1	T2
1.	**read**(*A*)	
2.	*A* := *A* – 50	
3.	**write**(*A*)	
		read(A), read(B), print(A+B)
4.	**read**(*B*)	
5.	*B* := *B* + 50	
6.	**write**(*B*)	

- **Isolation requirement:**
  - If between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

Example of Fund Transfer (Cont.)

- Transaction to transfer $50 from account A to account B:

	T1	T2
1.	**read**(*A*)	
2.	*A* := *A* – 50	
3.	**write**(*A*)	
		read(A), read(B), print(A+B)
4.	**read**(*B*)	
5.	*B* := *B* + 50	
6.	**write**(*B*)	

- **Isolation requirement:**
  - Isolation can be ensured trivially by running transactions **serially**
    - That is, one after the other.
  - However, executing multiple transactions concurrently has significant benefits.

Example of Fund Transfer (Cont.)

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

- **Durability requirement:**
  - Once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

The ACID Properties

- We require that the database system maintain the four ACID properties of the transactions to preserve the integrity of data:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

The ACID Properties (Cont.)

- **Atomicity:**
  - A transaction is ‘all or nothing’
  - Either all of its operations succeed (*commit)*, or they all fail (*rollback*)
- **Consistency:**
  - Every transaction leaves the database in a consistent state

The ACID Properties (Cont.)

- **Isolation:**
  - Transactions are isolated from one another
  - A transaction behaves as if it is the only thing running on the system
- **Durability:**
  - Changes made by a committed transaction are guaranteed to be permanent
  - Once a given transaction commits, its updates survive in the database, even if there is a subsequent system crash

Focus of Transaction Management

- Recovery Management
  - Transaction level, System level
- Concurrency Management

Recovery Manager

- Responsible for ensuring atomicity and durability
- Portion of the server that reads and processes the log
- Has three functions:
  - write log records
  - roll back a transaction
  - recover the database after a system crash

Recovery Manager: Log Records

- Writes a ***log record*** to the log in order to be able to roll back a transaction.
- Four basic kinds of log record:
  - **Start** record: when a transaction begins
  - **Commit/rollback** record: when a transaction completes
  - **Update** record: when a transaction modifies a value

Recovery Manager: Log Records

- Type of log record: START, SETINT, SETSTRING, COMMIT, or ROLLBACK

```
<START, 1>
<COMMIT, 1>
<START,  2>
<SETINT, 2, testfile, 1, 80, 1, 2>
<SETSTRING, 2, testfile, 1, 40, one, one!>
<COMMIT, 2>
<START, 3>
<SETINT, 3, testfile, 1, 80, 2, 9999>
<ROLLBACK, 3>
<START, 4>
<COMMIT, 4>
```

Recovery Manager: Log Records

- Transaction ID

```
<START, 1>
<COMMIT, 1>
<START,  2>
<SETINT, 2, testfile, 1, 80, 1, 2>
<SETSTRING, 2, testfile, 1, 40, one, one!>
<COMMIT, 2>
<START, 3>
<SETINT, 3, testfile, 1, 80, 2, 9999>
<ROLLBACK, 3>
<START, 4>
<COMMIT, 4>
```

Recovery Manager: Log Records

- Update records contain five additional values

```
<START, 1>
<COMMIT, 1>
<START,  2>
<SETINT, 2, testfile, 1, 80, 1, 2>
<SETSTRING, 2, testfile, 1, 40, one, one!>
<COMMIT, 2>
<START, 3>
<SETINT, 3, testfile, 1, 80, 2, 9999>
<ROLLBACK, 3>
<START, 4>
<COMMIT, 4>
```

Recovery Manager: Log Records

- File name

```
<START, 1>
<COMMIT, 1>
<START,  2>
<SETINT, 2, testfile, 1, 80, 1, 2>
<SETSTRING, 2, testfile, 1, 40, one, one!>
<COMMIT, 2>
<START, 3>
<SETINT, 3, testfile, 1, 80, 2, 9999>
<ROLLBACK, 3>
<START, 4>
<COMMIT, 4>
```

Recovery Manager: Log Records

- Block number of the modified file

```
<START, 1>
<COMMIT, 1>
<START,  2>
<SETINT, 2, testfile, 1, 80, 1, 2>
<SETSTRING, 2, testfile, 1, 40, one, one!>
<COMMIT, 2>
<START, 3>
<SETINT, 3, testfile, 1, 80, 2, 9999>
<ROLLBACK, 3>
<START, 4>
<COMMIT, 4>
```

Recovery Manager: Log Records

- Offset where the modification occurred

```
<START, 1>
<COMMIT, 1>
<START,  2>
<SETINT, 2, testfile, 1, 80, 1, 2>
<SETSTRING, 2, testfile, 1, 40, one, one!>
<COMMIT, 2>
<START, 3>
<SETINT, 3, testfile, 1, 80, 2, 9999>
<ROLLBACK, 3>
<START, 4>
<COMMIT, 4>
```

Recovery Manager: Log Records

- Old value at that offset

```
<START, 1>
<COMMIT, 1>
<START,  2>
<SETINT, 2, testfile, 1, 80, 1, 2>
<SETSTRING, 2, testfile, 1, 40, one, one!>
<COMMIT, 2>
<START, 3>
<SETINT, 3, testfile, 1, 80, 2, 9999>
<ROLLBACK, 3>
<START, 4>
<COMMIT, 4>
```

Recovery Manager: Log Records

- New value at that offset

```
<START, 1>
<COMMIT, 1>
<START,  2>
<SETINT, 2, testfile, 1, 80, 1, 2>
<SETSTRING, 2, testfile, 1, 40, one, one!>
<COMMIT, 2>
<START, 3>
<SETINT, 3, testfile, 1, 80, 2, 9999>
<ROLLBACK, 3>
<START, 4>
<COMMIT, 4>
```

Recovery Manager: Rollback

- The recovery manager ***rolls backs*** a transaction by reading the log backwards and undoing its modifications.

1. Set the current record to be the most recent log record.
2. Do until the current record is the start record for T:
    a) If the current record is an update record for T then:
        Write the saved old value to the specified location.
    b) Move to the previous record in the log.
3. Append a rollback record to the log.

Figure 14-5
The algorithm for rolling back transaction T

Recovery Manager: System Recovery

- Performed each time the database system starts up
- Purpose is to restore the database to a reasonable state
- “Reasonable state”
  - All uncompleted transactions should be rolled back.
  - All committed transactions should have their modifications written to disk.

Recovery Manager: Recovery

- Modifications made by uncompleted transactions:
  - Must be undone
- Modifications made by committed transactions:
  - Must be redone
- The recovery manager assumes that a transaction completed if the log file contains a commit or rollback record for it.

Recovery Manager: Recovery

- Modifications made by uncompleted transactions:
  - Must be undone

Undo-stage

- Modifications made by committed transactions:
  - Must be redone

Redo-stage

- The recovery manager assumes that a transaction completed if the log file contains a commit or rollback record for it.

Recovery Manager: Recovery Algorithms

- Undo-Redo Recovery Algorithm
- Undo-Only Recovery Algorithm
- Redo-Only Recovery Algorithm

Undo-Redo Recovery Algorithm

- Undoes the modifications made by uncommitted transactions
- Redoes the modifications made by committed transactions

// The undo stage

1. For each log record (reading backwards from the end):
   a) If the current record is a commit record then:
      Add that transaction to the list of committed transactions.
   b) If the current record is a rollback record then:
      Add that transaction to the list of rolled-back transactions.
   c) If the current record is an update record and that transaction is not on the committed or rollback list, then:
      Restore the old value at the specified location.

// The redo stage

2. For each log record (reading forwards from the beginning):
   If the current record is an update record and that transaction is on the committed list, then:
      Restore the new value at the specified location.

Figure 14-6
The undo-redo algorithm for recovering a database

Write-Ahead Logging

- Undo-Redo Recovery Algorithm assumption:
  - All updated records for an uncompleted transaction will be in the log file
- Problem:
  - System crash can occur in any time

Write-Ahead Logging

- Suppose that an uncompleted transaction modified a page and created a corresponding update log record.
- If the server crashes, there are four possibilities:
  a) Both the page and the log record got written to disk.
  b) Only the page got written to disk.
  c) Only the log record got written to disk.
  d) Neither got written to disk.

Write-Ahead Logging

- If the server crashes, there are four possibilities:

  a) **Both the page and the log record got written to disk.**
     - The recovery algorithm will find the log record and undo the change to the data block on disk.
     - No problem.

  b) Only the page got written to disk.

  c) Only the log record got written to disk.

  d) Neither got written to disk.

Write-Ahead Logging

- If the server crashes, there are four possibilities:
  a) Both the page and the log record got written to disk.
  b) **Only the page got written to disk.**
     - The recovery algorithm won’t find the log record, and so it will not undo the change to the data block.
     - This is a serious problem.
  c) Only the log record got written to disk.
  d) Neither got written to disk.

Write-Ahead Logging

- If the server crashes, there are four possibilities:
  a) Both the page and the log record got written to disk.
  b) Only the page got written to disk.
  c) **Only the log record got written to disk.**
     - The recovery algorithm will find the log record and will undo the non-existent change to the block.
     - Since the block wasn't changed, this is a waste of time, but not incorrect.
  d) Neither got written to disk.

Write-Ahead Logging

- If the server crashes, there are four possibilities:
  a) Both the page and the log record got written to disk.
  b) Only the page got written to disk.
  c) Only the log record got written to disk.
  d) **Neither got written to disk**.
     - The recovery algorithm won't find the log record, but since there was no change to the block.
     - There is nothing to undo anyway; no problem.

Write-Ahead Logging

- If the server crashes, there are four possibilities:
  a) Both the page and the log record got written to disk.
  b) **Only the page got written to disk.**
     - The recovery algorithm won’t find the log record, and so it will not undo the change to the data block.
     - This is a serious problem.
  c) Only the log record got written to disk.
  d) Neither got written to disk.

Write-Ahead Logging

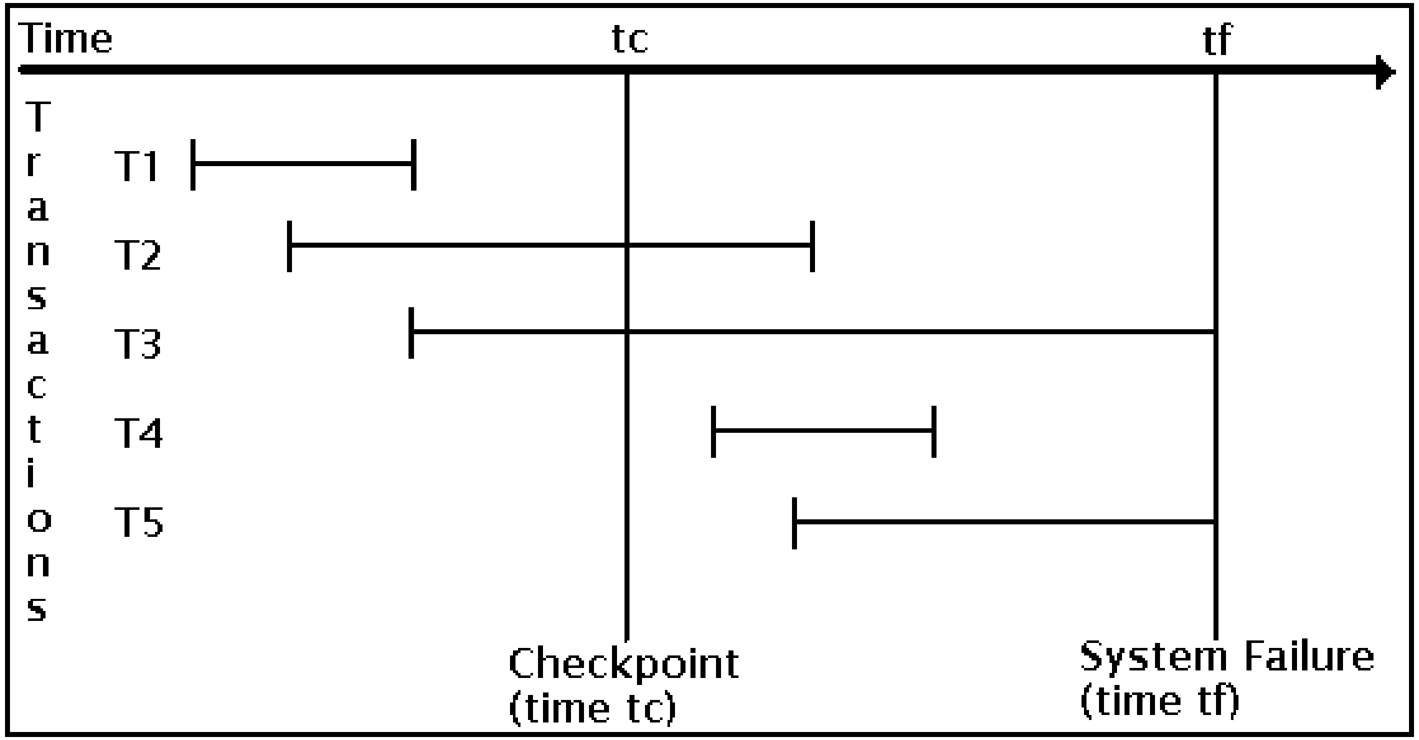
- A modified buffer can be written to disk only after all its update log records have been written to disk.
- Guarantees that modifications to the database will always be in the log and therefore will always be undoable.

Quiescent Checkpointing

- As time passes, the size of the log file can become very large.
- The recovery algorithm can stop searching the log as soon as it knows:
  - all earlier log records were written by completed transactions — Undo-stage
  - the buffers for those transactions have been flushed to disk — Redo-stage
- Checkpoint records are added to the log in order to reduce the portion of the log that the recovery algorithm needs to consider.

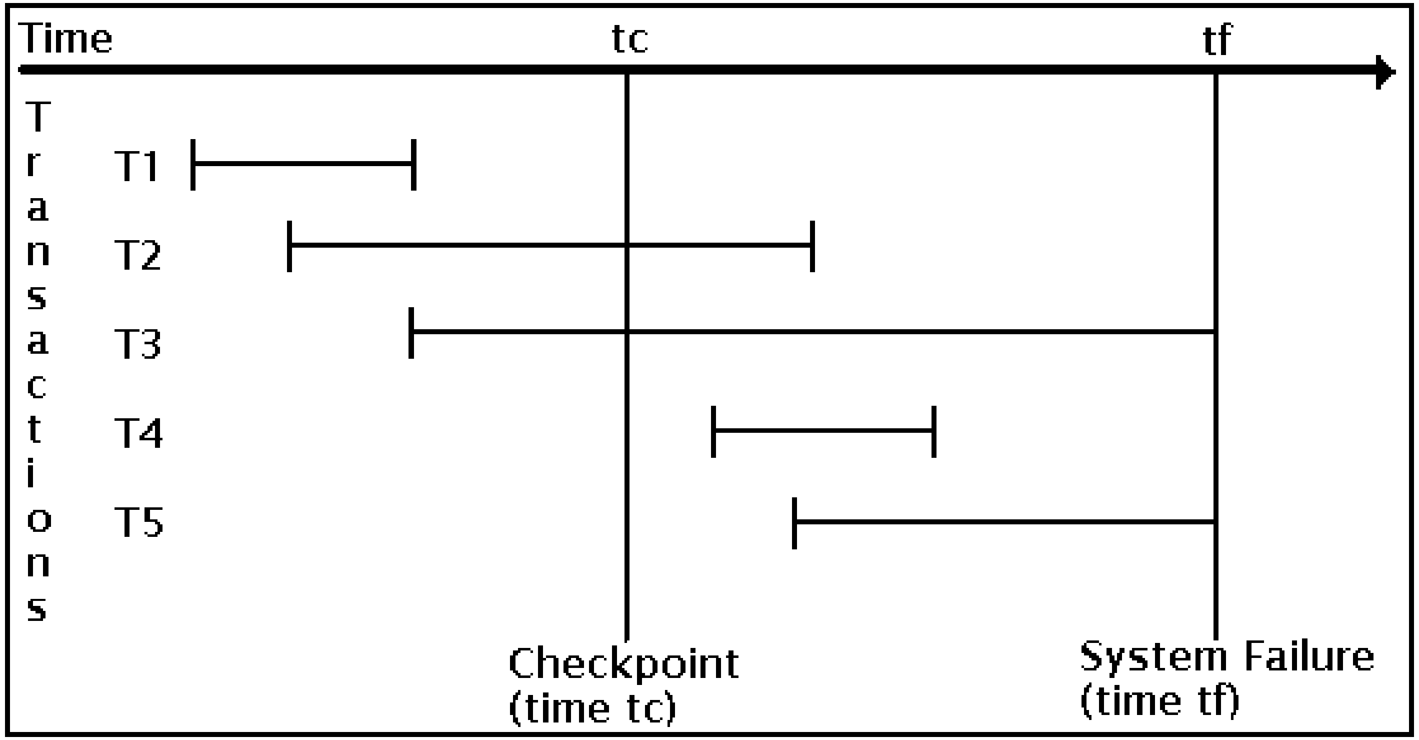
Example

- A System Failure has occurred at time *tf*.
- The most recent checkpoint prior to time *tf* was taken at time *tc*.



Example (Cont.)

- When the system is restarted, transactions of type T3 an T5 must be undone, and transactions of type T2 and T4 must be redone.
- Note, that transaction of type T1 do not enter into the restart process at all, because their updates were forced to the database at time *tc* as a part of the checkpoint process.



Quiescent Checkpointing

1. Stop accepting new transactions.
2. Wait for existing transactions to finish.
3. Flush all modified buffers.
4. Append a quiescent checkpoint record to the log and flush it to disk.
5. Start accepting new transactions.

Figure 14-9
The algorithm for performing a quiescent checkpoint

- The recovery algorithm never needs to look at the log records prior to a ***quiescent checkpoint*** record.

A Log Using Quiescent Checkpointing

```
<START, 0>
<SETINT, 0, junk, 33, 8, 542, 543>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
    //The quiescent checkpoint procedure starts here
<SETSTRING, 0, junk, 33, 12, joe, joseph>
<COMMIT, 0>
    //tx 3 wants to start here, but must wait
<SETINT, 2, junk, 66, 8, 0, 116>
<COMMIT, 2>
<CHECKPOINT>
<START, 3>
<SETINT, 3, junk, 33, 8, 543, 120>
```

Undo-Redo Recovery Algorithm

// The undo stage

1. For each log record (reading backwards from the end):

   a) If the current record is a commit record then:

      Add that transaction to the list of committed transactions.

   b) If the current record is a rollback record then:

      Add that transaction to the list of rolled-back transactions.

   c) If the current record is an update record and that transaction is not on the committed or rollback list, then:

      Restore the old value at the specified location.

// The redo stage

2. For each log record (reading forwards from the beginning):

   If the current record is an update record and that transaction is on the committed list, then:

      Restore the new value at the specified location.

Figure 14-6
The undo-redo algorithm for recovering a database

- Undoes the modifications made by uncommitted transactions
- Redoes the modifications made by committed transactions

Nonquiescent Checkpointing

- The recovery algorithm never needs to look at the log records prior to the start record of the earliest transaction listed in a ***nonquiescent checkpoint*** record.

1. Stop accepting new transactions.
2. Let $T_1,\ldots,T_k$ be the currently running transactions.
3. Flush all modified buffers.
4. Write the record `<NQCKPT, T1, ..., Tk>` into the log.
5. Start accepting new transactions.

Figure 14-11
The algorithm for adding a nonquiescent checkpoint record

A Log Using Nonquiescent Checkpointing

```
<START, 0>
<SETINT, 0, junk, 33, 8, 542, 543>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
<NQCKPT, 0, 2>
<SETSTRING, 0, junk, 33, 12, joe, joseph>
<COMMIT, 0>
<START, 3>
<SETINT, 2, junk, 66, 8, 0, 116>
<SETINT, 3, junk, 33, 8, 543, 120>
```

Nonquiescent Checkpointing and Recovery Algorithm

- Stage 1 of the algorithm reads the log backwards as before and keeps track of the completed transactions.
- When it encounters a nonquiescent checkpoint record <NQCKPT T_1,. . .,T_k>, it determines which of these transactions are still running.
- It can then continue reading the log backwards until it encounters the start record for the earliest of those transactions.
- All log records prior to this start record can be ignored.

A Log Using Nonquiescent Checkpointing:

- When it encounters the <SETINT, 3, junk, 33, 8, 543, 120> log record
  - It will check to see if transaction 3 was on the list of committed transactions
  - Since that list is currently empty, the algorithm will perform an undo, writing the integer 543 to offset 8 of block 3
- The log record <SETINT, 2, junk, 66, 8, 0, 116> will be treated similarly, writing the integer 0 to offset 8 of block 66 of “junk”.
- The <COMMIT, 0> log record will cause 0 to be added to the list of committed transactions.
- The <SETSTRING, 0, ...> log record will be ignored, because 0 is in the committed transaction list.

Nonquiescent Checkpointing

- When it encounters the <NQCKPT 0, 2> log record
  - It knows that transaction 0 has committed, and thus it can ignore all log records prior to the start record for transaction 2.
- When it encounters the <START, 2> log record, it enters stage 2 and begins moving forward through the log.
- The <SETSTRING, 0, ...> log record will be redone, because 0 is in the committed transaction list.
  - The value ‘joseph’ will be written to offset 12 of block 33 of “junk”.

Data Item Granularity

- A recovery manager can choose to log values, records, pages, files, and so on.
- The unit of logging used by the recovery manager is called a ***recovery data item.***
- The size of a data item is called its ***granularity***.
- Recovery data item granularity levels:
  - Values
  - Blocks
  - Files
- Tradeoff: A large-granularity data item will require fewer update log records, but each log record will be larger.

A Recovery Manager’s Decisions

- Undo-only recovery or undo only recovery or redo only recovery
- Value-granularity data items
- Quiescent checkpointing or nonquiscent checkpointing

References

- Edward Sciore, Database Design and Implementation, Wiley, 2009.
- Edward Sciore, Database Design and Implementation, Second Edition, Springer, 2020.
- A. Silberschatz, HF. Korth, S. Sudarshan, Database System Concepts, 7th Ed., McGraw-Hill, 2019.
  - Chapter 17, https://www.db-book.com/db7/slides-dir/PPTX-dir/ch17.pptx (modified)
- C.J. Date, An Introduction to Database Systems, 8th Ed., 2003.
  - Chapter 15