Chapter 11
Database Recovery

Failures in DBMS

- Two common kinds of failures
- **System failure** (e.g. power outage)
  - affects all transactions currently in progress but does not physically damage the data (soft crash)
- **Media failures** (e.g. Head crash on the disk)
  - damage to the database (hard crash)
  - need backup data
- Recovery scheme responsible for handling failures and restoring database to consistent state
Recovery

- Recovering the database itself
- Recovery algorithm has two parts
  - Actions taken during normal operation to ensure system can recover from failure (e.g., backup, log file)
  - Actions taken after a failure to restore database to consistent state
- We will discuss (briefly)
  - Transactions/Transaction recovery
  - System Recovery

Transactions

- A database is updated by processing *transactions* that result in changes to one or more records.
- A user’s program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned with data read/written from/to the database.
- The DBMS’s abstract view of a user program is a sequence of transactions (*reads* and *writes*).
- To understand database recovery, we must first understand the concept of *transaction integrity*. 
Transactions

- A transaction is considered a logical unit of work
  - START Statement: `BEGIN TRANSACTION`
  - END Statement: `COMMIT`
  - Execution errors: `ROLLBACK`

- Assume we want to transfer $100 from one bank (A) account to another (B):
  
  ```
  UPDATE Account_A SET Balance = Balance -100;
  UPDATE Account_B SET Balance = Balance +100;
  ```

- We want these two operations to appear as a **single atomic action**

Transactions

- We want these two operations to appear as a **single atomic action**
  - To avoid **inconsistent states** of the database **in-between** the two updates
  - And obviously we cannot allow the first UPDATE to be executed and the second not or vice versa.

- Transactions guarantee that, if a failure occurs before the transaction reaches its planned termination, then those previous transaction updates will be **undone**.
**Pseudocode Transaction**

```
BEGIN TRANSACTION
    UPDATE ACCOUNT_A {BALANCE = BALANCE -100}
    IF any error occurred then GO TO UNDO; END IF;
    UPDATE ACCOUNT_B {BALANCE = BALANCE +100}
    IF any error occurred then GO TO UNDO; END IF;
    COMMIT;
    GO TO FINISH;
UNTDO:
    ROLLBACK;
FINISH:
    RETURN;
```

**Transaction Recovery**

- **COMMIT** establishes a Commit Point or Synch Point
  - A point at which we assume the database in a correct state

- **ROLLBACK** has to roll back the database to the state it had before the Transaction started.
Transaction **ACID** Properties

- **Atomicity**
  - Transactions are atomic (all or nothing)

- **Consistency**
  - Transaction transform the DB from one correct state to another correct state

- **Isolation**
  - Transactions are isolated from each other

- **Durability**
  - Once a transaction commits, changes are permanent: no subsequent failure can reverse the effect of the transaction.

---

**Atomicity**

```
UPDATE Account_A SET Balance= Balance -100;
1. Read(A)
2. A = A - 100
3. Write(A)
UPDATE Account_B SET Balance= Balance +100;
4. Read(B)
5. B = B + 100
6. write(B)
```

- Transaction may fail after step 3 and before step 6 (failure could be due to software or hardware)
- The system should ensure that updates of a partially executed transaction are not reflected in the database
Consistency

```
UPDATE Account_A SET Balance= Balance -100;
1.   Read(A)
2.   A = A - 100
3.   Write(A)
UPDATE Account_A SET Balance = Balance +100;
4.   Read(B)
5.   B = B + 100
6.   write(B)
```

- In this example, the sum of A and B is unchanged by the execution of the transaction

Isolation

```
<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Read(A)</td>
<td>Read(A), Read(B), Write(A+B)</td>
</tr>
<tr>
<td>2. A = A - 100</td>
<td></td>
</tr>
<tr>
<td>3. Write(A)</td>
<td></td>
</tr>
<tr>
<td>4. Read(B)</td>
<td></td>
</tr>
<tr>
<td>5. B = B + 100</td>
<td></td>
</tr>
<tr>
<td>6. write(B)</td>
<td></td>
</tr>
</tbody>
</table>
```

- What will T2 “see”? Database changes not revealed to users until after transaction has completed
- Isolation can be ensured trivially by running transactions *serially*
- However, executing multiple transactions concurrently has significant benefits
  - Keep CPU humming when disk I/O takes place.
**Durability**

```
UPDATE Account_A SET Balance= Balance -100;
1.  Read(A)
2.  A = A - 100
3.  Write(A)
UPDATE Account_A SET Balance= Balance +100;
4.  Read(B)
5.  B = B + 100
6.  write(B)
```

- Database changes are permanent (once the transaction was committed).

---

**Passing the ACID Test**

- **Logging and Recovery**
  - Guarantees Atomicity and Durability
  - Log file based recovery techniques

- **Concurrency Control**
  - Guarantees Consistency and Isolation, given Atomicity

- **We’ll do Recovery Methods first**
  - Assume no concurrency and study recovery methods – Log based recovery
  - Concurrency control methods will then generate schedules that are “recoverable”
Concept: Log-based Recovery

- Write to a log file before writing to database
  - Enter log records
- Transaction states:
  - Start, Abort, Commit

Example log file

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Log File</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 start</td>
<td>&lt;T1,Start&gt;</td>
</tr>
<tr>
<td>Read (A)</td>
<td></td>
</tr>
<tr>
<td>A = A - 100</td>
<td></td>
</tr>
<tr>
<td>Write (A)</td>
<td>&lt;T1,A,1000,900&gt;</td>
</tr>
<tr>
<td>T1 end</td>
<td>&lt;T1,Commit&gt;</td>
</tr>
<tr>
<td>T2 start</td>
<td>&lt;T2,Start&gt;</td>
</tr>
<tr>
<td>Read (B)</td>
<td></td>
</tr>
<tr>
<td>B = B + 100</td>
<td></td>
</tr>
<tr>
<td>Write (B)</td>
<td>&lt;T2,B,2000,2100&gt;</td>
</tr>
<tr>
<td>T2 end</td>
<td>&lt;T2,Commit&gt;</td>
</tr>
</tbody>
</table>
Recovery using Log File

- When has a transaction committed?
  - When <Ti, start> and <Ti, Commit> is in the log file

- When has a transaction failed/aborted?
  - When <Ti, Start> is in the log file but no <Ti, Commit>

- Backward Recovery:
  - Undo transaction (restore old values) if no Commit

- Forward Recovery:
  - Start with an earlier copy of the database
  - Redo transaction (write new values) if Commit

Checkpoints

- Recovery system consults log file when failure occurs
  - Search entries in log file
  - Time depends on number of entries

- Why redo transactions already written to disk?
  - Introduce checkpointer records into log file

- Only transactions committed after last checkpoint need to be redone after a failure
At restart, transactions 2 and 4 must be redone.

Concurrency in Transaction Processing
Concurrency Control

- Typically a DBMS allows many different transactions to access the database at the same time.
- This may result in data inconsistency.
- **Solution—Concurrency Control**
  - The process of managing simultaneous operations against a database so that ACID properties are maintained.

Figure 11-10  Lost update (no concurrency control in effect)

Simultaneous access causes updates to cancel each other.
Concurrent Control Techniques

- Serializability
  - Finish one transaction before starting another

- Locking Mechanisms (Pessimistic Approach)
  - The most common way of achieving serialization
  - Data that is retrieved for the purpose of updating is locked for the updater
  - No other user can perform update until unlocked

- Versioning (Optimistic Approach)
  - Newer approach to concurrency control

Figure 11-11: Updates with locking (concurrency control)

This prevents the lost update problem
Locking Mechanisms

- **Locking level:**
  - Database—used during database updates
  - Table—used for bulk updates
  - Block or page—very commonly used
  - Record—only requested row; fairly commonly used
  - Field—requires significant overhead; impractical

- **Types of locks:**
  - **Shared lock**—Read, but no update, permitted. Used when just reading to prevent another user from placing an exclusive lock on the record
  - **Exclusive lock**—No access permitted. Used when preparing to update.

Deadlock

- An impasse that results when two or more transactions have locked common resources, and each waits for the other to unlock their resources.

*John and Marsha will wait forever for each other to release their locked resources!*
Another Deadlock Example

- Unless DBMS intervenes, both users will wait indefinitely!

![Deadlock Diagram]

Managing Deadlock

- **Deadlock Prevention:**
  - Lock all records required at the beginning of a transaction
  - Two-phase locking protocol
    - Growing phase: all necessary locks acquired
    - Shrinking phase: all locks released
  - May be difficult to determine all needed resources in advance

- **Deadlock Resolution:**
  - Allow deadlocks to occur
  - Mechanisms for detecting and breaking them
  - Simple hack: *timeouts*. T1 made no progress for a while? Shoot it!
Versioning

- Optimistic approach to concurrency control
- Replaces locking
- Assumption is that simultaneous updates will be infrequent
- Each transaction can attempt an update as it wishes
- The system will reject an update when it senses a conflict
- Use of rollback and commit for this

Figure 11-14 The use of versioning

Better performance than locking
End of Lecture

Three things in life that are certain:
Death
Taxes
And
Lost Data

As computer scientists, at least we may try to do something against the latter.