CS143: Transactions

Professor Junghoo "John" Cho

Motivation (1)

- Crash recovery
  - Example: Transfer $1M from Susan to Jane

    S1: UPDATE Account SET balance = balance - 1000000 WHERE owner = `Susan‘
    S2: UPDATE Account SET balance = balance + 1000000 WHERE owner = `Jane‘

    System crashes after S1 but before S2. What now?

Motivation (2)

T1	T2
A = balance	
A = A - 10	
Give out $10	
	B = balance
	B = B - 20
	Give out $20
	balance = B
balance = A	

balance
100

- Q: How can DBMS guarantee that these “bad” scenarios will never happen?

Transaction

- A sequence of SQL statements that are executed as "one unit"
- DBMS guarantees ***ACID*** property on all transactions
  - <u>A</u>tomicity: "all or nothing"
    - Either ALL OR NONE of the operations in a transaction is executed
    - If system crashes in the middle of a transaction, all changes are "undone"
  - <u>C</u>onsistency
    - If the database was in a "consistent" state before transaction, it is still in a consistent state after the transaction
  - <u>I</u>solation
    - Even if multiple transactions run concurrently, the final result is the same as each transaction runs in isolation in a sequential order
  - <u>D</u>urability
    - All changes made by "committed" transaction will remain even after system crash

Transactions in SQL

- Two basic commands
  - **COMMIT**: all changes made by the transaction is stored permanently
  - **ROLLBACK**: Undo all changes made by the transaction
- AUTOCOMMIT mode
  - When ON: every SQL statement becomes one transaction
  - When OFF:
    - All SQL commands through COMMIT/ROLLBACK become one transaction

Setting Autocommit Mode

- Oracle: SET AUTOCOMMIT ON/OFF (default is off)
- MySQL: SET AUTOCOMMIT = {0|1} (default is on. InnoDB only)
- MS SQL Server: SET IMPLICIT_TRANSACTIONS OFF/ON (default is off)
  - IMPLICIT_TRANSACTION ON means AUTOCOMMIT OFF
- DB2: UPDATE COMMAND OPTIONS USING c ON/OFF (default is on)
- In JDBC: connection.setAutoCommit(true/false) (default is on)
- In Oracle, MySQL, and MS SQL Sever, “BEGIN TRANSACTION” command temporarily disables autocommit mode until COMMIT or ROLLBACK

SQL Isolation Levels

- By default, RDBMS guarantees ACID for transactions
- Some applications may not need ACID and may want to allow minor "bad scenarios" to gain more "concurrency"
- By specifying "SQL Isolation Level," app developer can specify what type of "bad scenarios" can be allowed for their apps
    - Dirty read, non-repeatable read, and phantom

Dirty Read

name	salary
Amy	1000
Eddie	1000
Esther	1000
John	1000
Melanie	1000

- T1: UPDATE Employee SET salary = salary + 100;
  T2: SELECT salary FROM Employee WHERE name = ‘Amy’;
- Q: Under ACID, once T1 update Amy’s salary, can T2 read Amy’s salary?
- Some applications may be OK with *dirty read*
  - Among 4 SQL isolation levels, READ UNCOMMITTED allows dirty read

SQL Isolation Levels

	Dirty read		
Read uncommitted	Y		
Read committed	N		
Repeatable read	N		
Serializable	N		

Non-repeatable Read

- T1: UPDATE Employee SET salary = salary + 100 WHERE name = ‘John’;

  T2: (S1) SELECT salary FROM Employee WHERE name = ‘John’;

  ...

  (S2) SELECT salary FROM Employee WHERE name = ‘John’;
- Q: Under ACID, can T2 get different values for S1 and S2?
- ***Non-repeatable read***: When Ti reads the same tuple multiple times, Ti may get different value
- SQL isolation levels, READ UNCOMMITTED and READ COMMITTED, allow non-repeatable read

SQL Isolation Levels

	Dirty read	Non-repeatable read	
Read uncommitted	Y	Y	
Read committed	N	Y	
Repeatable read	N	N	
Serializable	N	N	

Phantom

- T1: INSERT INTO Employee VALUES (Beverly, 1000), (Zack, 1000);
  T2: SELECT SUM(salary) FROM Employee;

name	salary
Amy	1000
Eddie	1000
Esther	1000
John	1000
Melanie	1000

- Q: Under ACID, what may T2 return?

Phantom

- ***Phantom***: When new tuples are inserted, statements may or may not see (part of) them
  - Preventing phantom can be very costly
  - Exclusive lock on the entire table or a range of tuples
- Except the isolation level SERIALIZABLE, phantoms are allowed

SQL Isolation Levels

	Dirty read	Non-repeatable read	Phantom
Read uncommitted	Y	Y	Y
Read committed	N	Y	Y
Repeatable read	N	N	Y
Serializable	N	N	N

Access Mode

- A transaction can be declared to be ***read only***, when it has SELECT statements only (no INSERT, DELETE, UPDATE)
- DBMS may use this information to optimize for more concurrency

Declaring SQL Isolation Level

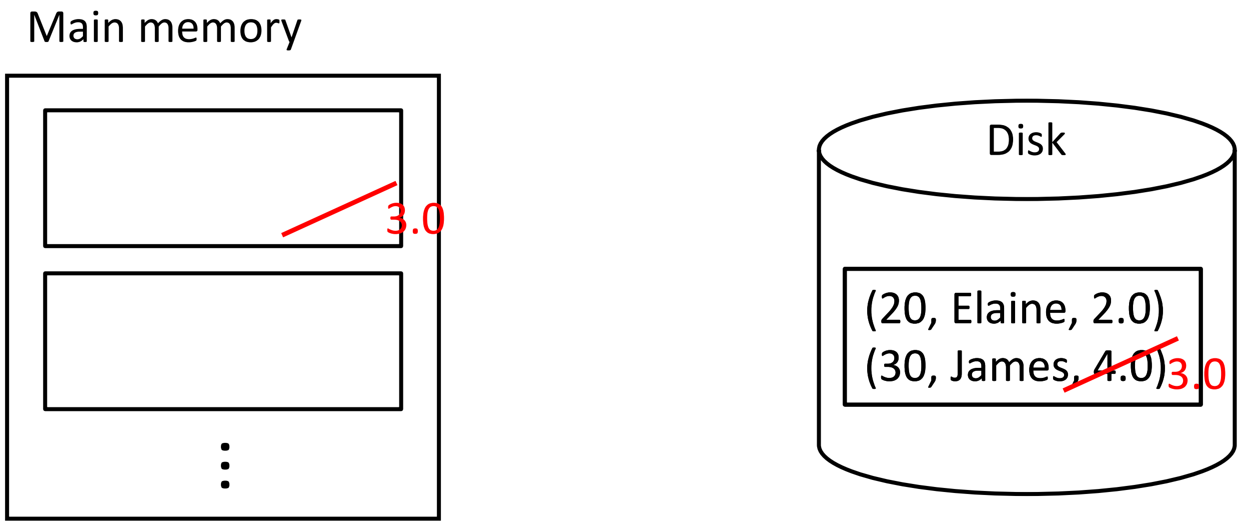
- SET TRANSACTION [READ ONLY] ISOLATION LEVEL <level>
  - e.g., SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
- More precisely “SET TRANSACTION [access mode,] ISOLATION LEVEL <level>”
  - access mode: READ ONLY/READ WRITE (default: READ WRITE)
  - level:
    - READ UNCOMMITTED
    - READ COMMITTED (default in Oracle, MS SQL Server)
    - REPEATABLE READ (default in MySQL, IBM DB2)
    - SERIALIZABLE
  - READ UNCOMMITED is allowed only for READ ONLY access mode
- Isolation level needs to be set before every transaction

Mixing Isolation Levels

- John’ initial salary = 1000
  T1: UPDATE Employee SET salary = salary + 100; ROLLBACK;
  T2: SELECT salary FROM Employee WHERE name = ‘John’;
- Q: T1: SERIALIZABLE and T2: SERIALIZABLE. What may T2 return?
- Q: T1: SERIALIZABLE and T2: READ UNCOMMITTED. What may T2 return?
- Isolation level is in the eye of the beholding operation
  - Global ACID is guaranteed only when ***all*** transactions are SERIALIZABLE

Guaranteeing ACID

- T1: UPDATE Student SET GPA = 3.0 WHERE sid = 30;



- DBMS does not immediately writes the updated disk block back to disk for performance reasons
  - Q: What happens if the system crashes before the block is written back?

Rolling Back to Earlier State

- T: $\mathrm{read(A)}\ \mathrm{write(A)}\ \mathrm{read(B)}\ \mathrm{write(B)}$

  Q: What if we execute up to “$\mathrm{read(A)}\ \mathrm{write(A)}\ \mathrm{read(B)}$” and decide to ROLLBACK? How can we go back to the “old value” of A?

Partial Execution

- T: $\text{read(A) write(A) read(B) write(B)}$

  Q: What if system executes up to “read(A) write(A)”, and system crashes? What should the system do when it reboots? How does the system know whether T did not finish?

Logging: Intuition

- In a separate log file, save the following log records before T_i takes any action:

Log record	When
<T_i, start>	Before transaction T_i starts
<T_i, commit/abort>	Before transaction T_i is committed/aborted
<T_i, X, old-value, new-value>	Before a statement in T_i changes value of X from “old-value” to “new-value”

- These records are used during ROLLBACK or during crash recovery

Logging Example

T1	T2
x = read(A)	
x = x - 50	
write(A, x)	
	z = read(C)
	z = z * 2
	write(C, z)
	commit
y = read(B)	
y = y + 50	
write(B, y)	
commit	

Memory

A: ~~100~~ 50 | B: ~~100~~ 150 | C: ~~100~~ 200

Log file

1 <T1, start>
2 <T1, A, 100, 50>
3 <T2, start>
4 <T2, C, 100, 200>
5 <T2, commit>
6 <T1, B, 100, 150>
7 <T1, commit>

Disk

A: 100 | B: 100 | C: 100

Log file

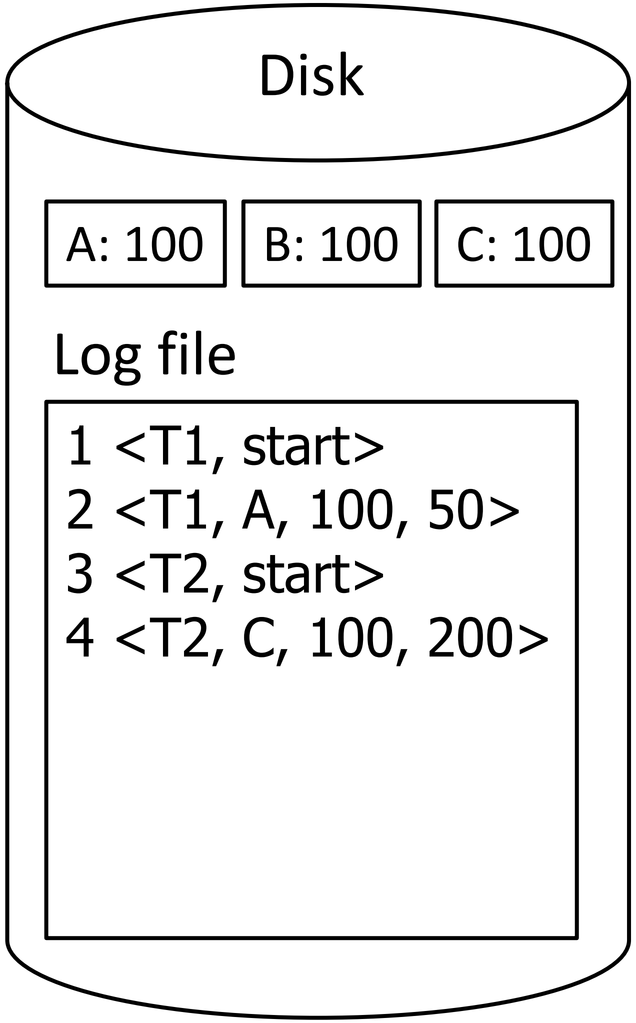
1 <T1, start>
2 <T1, A, 100, 50>
3 <T2, start>
4 <T2, C, 100, 200>
5 <T2, commit>
6 <T1, B, 100, 150>
7 <T1, commit>

Rules for Log-Based Recovery

1. DBMS generates a log record before start and end and modification by T_i
2. Before T_i is committed, all log records until T_i's commit must be flushed to disk
3. Before any modified tuple is written back to disk, all log records through the tuple modification must be flushed to disk first
   - Example: the log record <T_i, A, 5, 10> should be written to the disk before the tuple A is updated to 10 in disk
4. During ROLLBACK, DBMS reverts to old values of tuples using log records
5. During crash recovery, DBMS does:
   a) "re-execute" all actions in the log file from the beginning to the end and
   b) "rolls back" all actions from non-committed transactions in the reverse order

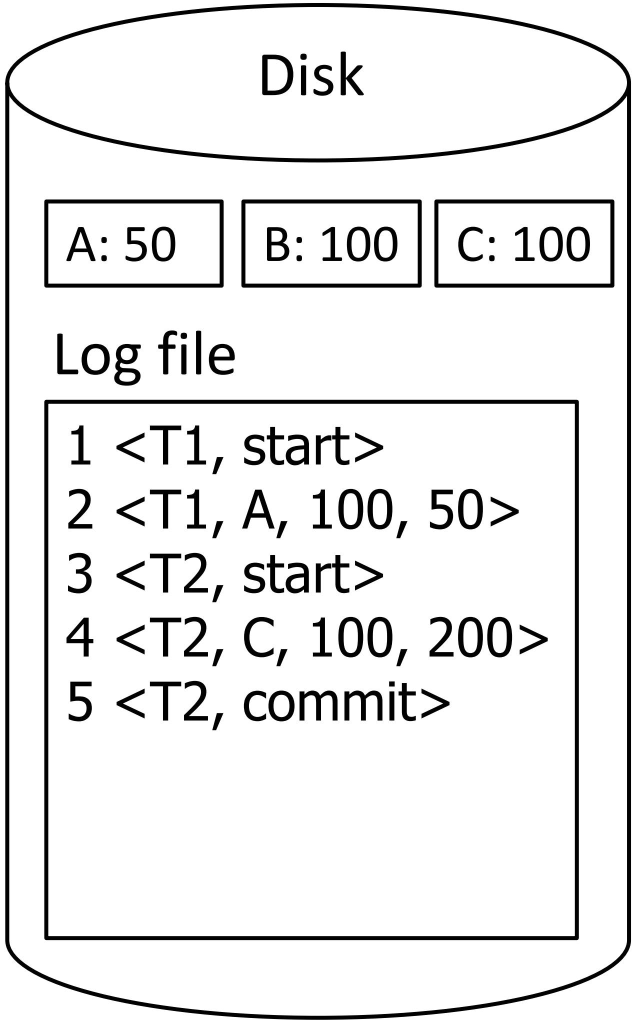
Example: Recovery

T1	T2
x = read(A)	
x = x - 50	
write(A, x)	
	z = read(C)
	z = z * 2
	write(C, z)
	commit
y = read(B)	
y = y + 50	
write(B, y)	
commit	



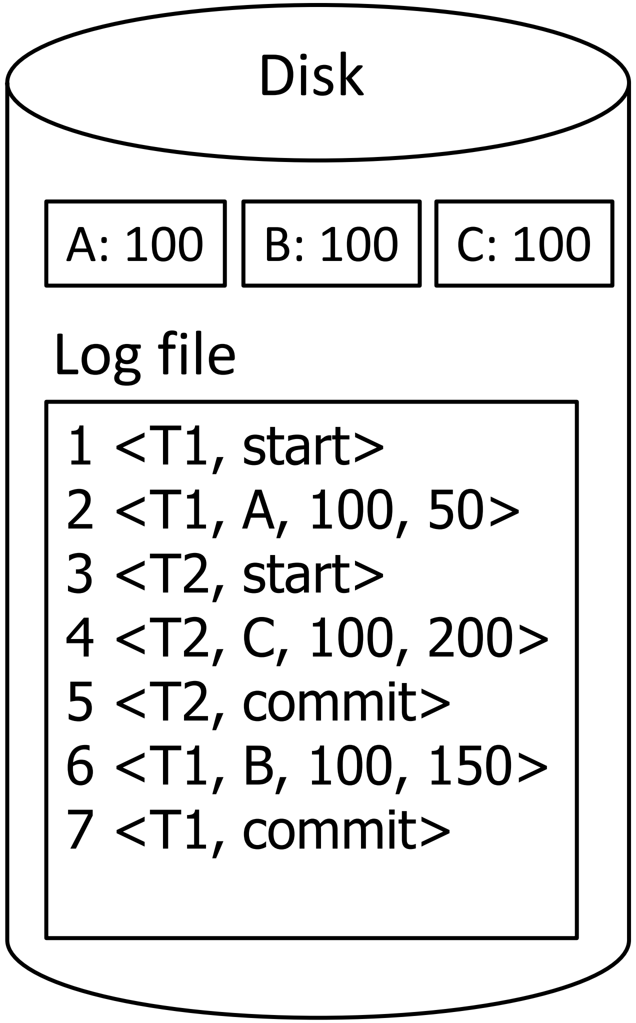
Example: Recovery

T1	T2
x = read(A)	
x = x - 50	
write(A, x)	
	z = read(C)
	z = z * 2
	write(C, z)
	commit
y = read(B)	
y = y + 50	
write(B, y)	
commit	



Example: Recovery

T1	T2
x = read(A)	
x = x - 50	
write(A, x)	
	z = read(C)
	z = z * 2
	write(C, z)
	commit
y = read(B)	
y = y + 50	
write(B, y)	
commit	



Summary

- DBMS uses a log file to ensure ACID for transactions
  - Helps rolling back partially executed transactions
  - Helps recovery after crash
- Before modifying any data, DBMS generates a log record
- Before commit, DBMS flushes log records to disk to ensure durability
- During recovery, records in the log file are “replayed” to put the system in the supposed state