Fakultas Informatika
School of Computing
Telkom University

CSH2D3 - Database System

09 | Transactions (1)

Outline

Transaction Concept

Transaction State

Concurrent Executions

In most large database systems, many users and application programs will be (and must be) accessing the database at the same time. Having concurrent users updating the database raises a number of problems that, if not properly dealt with by the DBMS, could leave the database in an inconsistent state, even if all users "did the right thing".

Example 1 : Centralized Database

Ilustration:

- A bank application can be used by several users on centralized database
- The tables are:
  - Account (account_no, customer_name, balance)
  - Deposit (deposit_no, deposit_date, account_no, deposit_amount)
  - Withdrawal (witdhdrawal_no, withdrawal_date, account_no, withdrawal_amount)
- For example: IDR 1,000,000 is taken from account_no “007” which have balance IDR 1,500,000 by two different users.

... Program Written ...

1. **Read account data**
2. **Check balance**
   SELECT balance FROM account WHERE account_no = “007”;
3. **If balance is sufficient, update balance; otherwise process end.**
   UPDATE account SET balance = balance – 1000000 WHERE account_no = “007”;
4. **Save data to withdrawal table.**
   INSERT INTO withdrawal VALUES (“W324”, “March-14-2017”, “007”, 1000000);
5. **COMMIT;**

Concurrent Processing Mechanism

User #1	User #2
T1: read data	
T2: select	
	T3: read data
	T4: select
T5: update	
T6: insert	
	T7: update
	T8: insert
	T9: commit
T10: commit	

If the process sequence is described as following table, then both (user) action can be executed, although when the user #2 is withdrawing, the balance account is no longer sufficient.
WHY?
in T4, balance is still read as 1500000, because update process is executed when T5
THIS CAN HAPPEN because the processes are not treated as "a transaction"

it should be

User #1	User #2
T1: read data T2: select update insert commit	T3: read data T4: select update insert commit

What is Transaction (Tx)?

- Collection, set or group of SQL statement will be treated as an **atomic process**. When a transaction is executed, one or more data items which stored on database tables will be updated.
- A **transaction** is a sequence of SQL statements that DBMS treats as a single unit of work.
- As soon as you connect to the database with sqlplus, a transaction begins.
- Once the transaction begins, every SQL DML (Data Manipulation Language) statement you issue subsequently becomes a part of this transaction.

Tx – Starts

- ✓ By the SQL standard, transactions consisting of more than one statement are started by the command START TRANSACTION.
- ✓ Each statement issued at the generic query interface is a transaction by itself.
- ✓ In programming interfaces like Embedded SQL or PSM, a transaction begins the first time an SQL statement is executed.

Tx – Ends

- ✓ With the SQL COMMIT; statement
- ✓ With the SQL ROLLBACK; statement
- ✓ When the user ends the session normally, disconnect from the database (implicit commit)
- ✓ When the user process crashes (implicit rollback)
- ✓ An exception is generated in the processing of the statements (implicit rollback)

Tx – Commit and Rollback

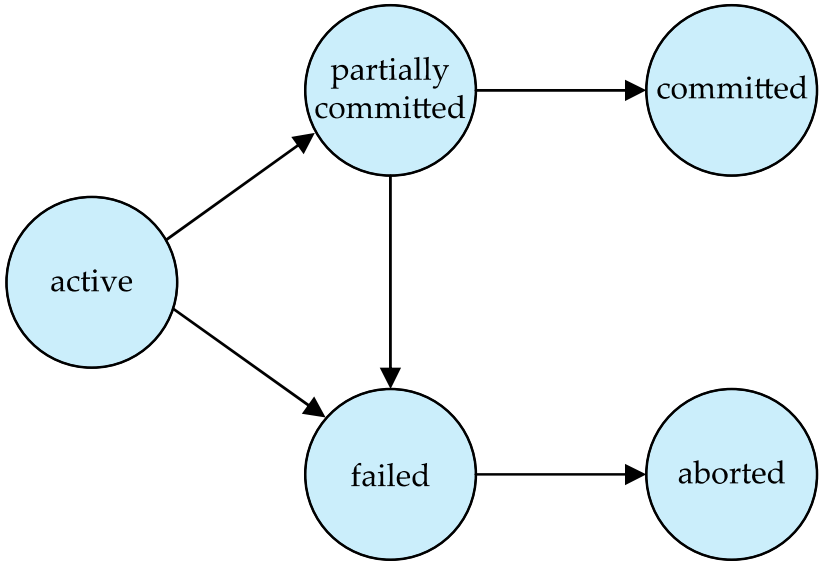
- After the current transaction has ended with a COMMIT or ROLLBACK, the first executable SQL statement that you subsequently issue will automatically begin another transaction.
- COMMIT makes permanent any database changes you made during the current transaction. Normally, other users cannot see them until you commit your changes. But this depends on the isolation level of DBMS (will be discussed in next chapter).
- ROLLBACK ends the current transaction and undoes any changes made since the transaction began.

Example 2:

For example, the following SQL commands have the final effect of inserting into table R the tuple (3, 4), but not (1, 2):

```
insert into R values (1, 2);
rollback;
insert into R values (3, 4);
commit;
```

Transaction State



- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

ACID Properties

To preserve the integrity of data, the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- 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*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Example of Fund Transfer

- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - 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
- **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.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **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).

	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 can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.
- A serial schedule

	T_1	T_2	
1	read (A)		1
2	$A := A - 50$		2
3	write (A)		3
4	read (B)		4
5	$B := B + 50$		5
6	write (B)		6
7	commit		7
8		read (A)	8
9		$temp := A * 0.1$	9
10		$A := A - temp$	10
11		write (A)	11
12		read (B)	12
13		$B := B + temp$	13
14		write (B)	14
15		commit	15

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1. When a concurrent schedule equivalent to a serial schedule, the concurrent schedule is **serializable**.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of (*A* + *B*). When a concurrent schedule not-equivalent to all serial schedules, the concurrent schedule is **nonserializable**.

T_1	T_2
read (*A*)	
A := *A* – 50	
	read (*A*)
	temp := *A* * 0.1
	A := *A* - *temp*
	write (*A*)
	read (*B*)
write (*A*)	
read (*B*)	
B := *B* + 50	
write (*B*)	
commit	
	B := *B* + *temp*
	write (*B*)
	commit

Exercise

Consider the following two transactions:

```
T1: read(A);
    read(B);
    if A = 0 then B := B + 1;
    write(B)
T2: read(B);
    read(A);
    if B = 0 then A := A + 1;
    write(A)
```

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ as the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.

b. Show a concurrent execution of *T*1 and *T2* that produces a nonserializable schedule.

c. Is there a concurrent execution of *T*1 and *T2* that produces a serializable schedule?

References

Silberschatz, Korth, and Sudarshan. *Database System Concepts* – 7th Edition. McGraw-Hill. 2019.

Slides adapted from Database System Concepts Slide.

Source: https://www.db-book.com/db7/slides-dir/index.html