Transaction

NGUYEN HongPhuong
Email: phuongnh@soict.hust.edu.vn
Site: http://users.hust.edu.vn/phuongnh
Face: https://www.facebook.com/phuongnhbk
Hanoi University of Science and Technology

Contents

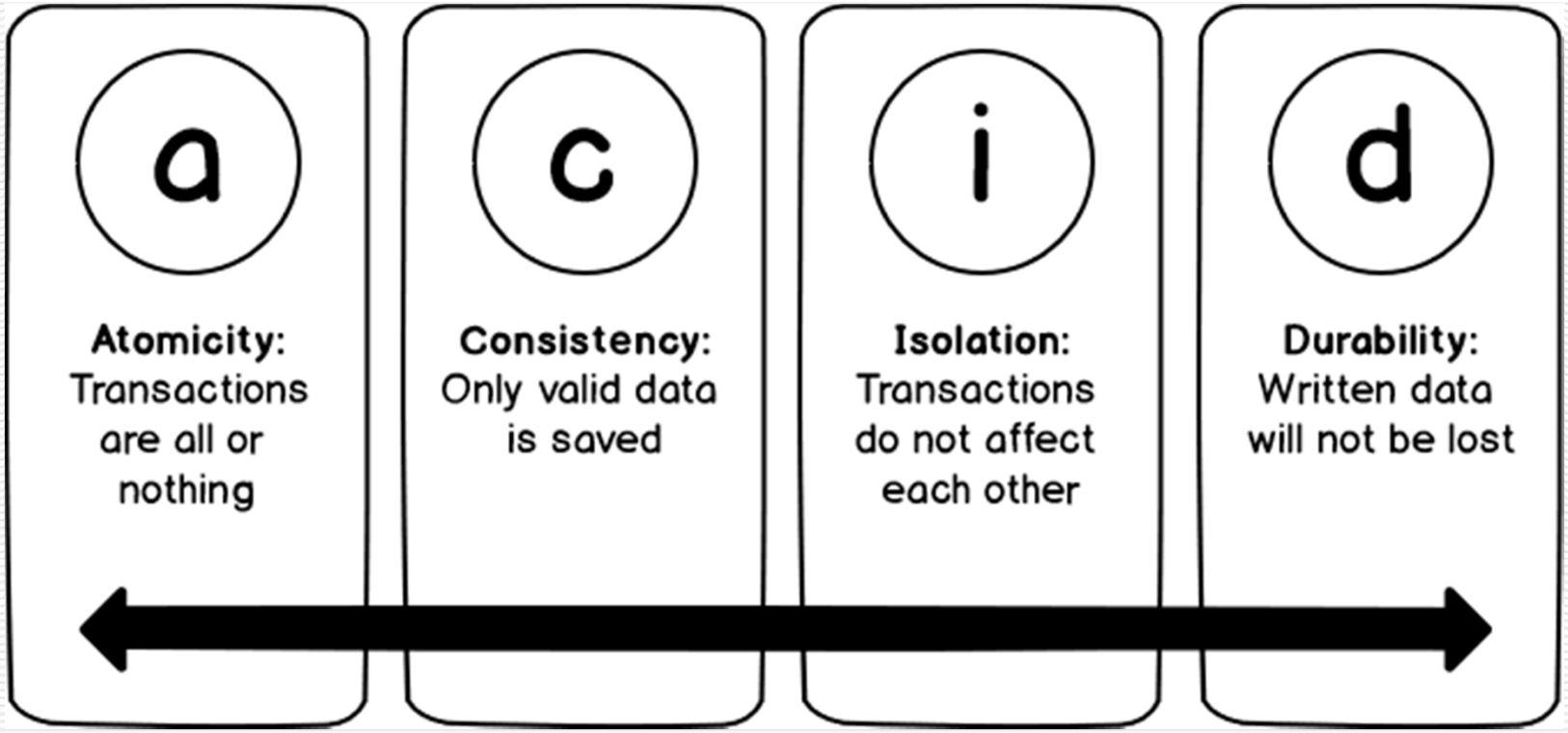
- Introduction
- Properties of a transaction
- Transaction states
- Processing a transaction
- Transactions, read and write operation, DBMS buffer
- Some examples
- Transaction best practices

Introduction

- **Transactions in SQL** are a group of SQL statements. If a transaction is made successfully, all data changes made in the transaction are saved to the database. If a transaction fails and is rolled back, all data modifications will be deleted (data is restored to the state before the transaction was executed).

Properties of a transaction

- The transaction has 4 standard properties, referenced by ACID



Properties of a transaction (2)

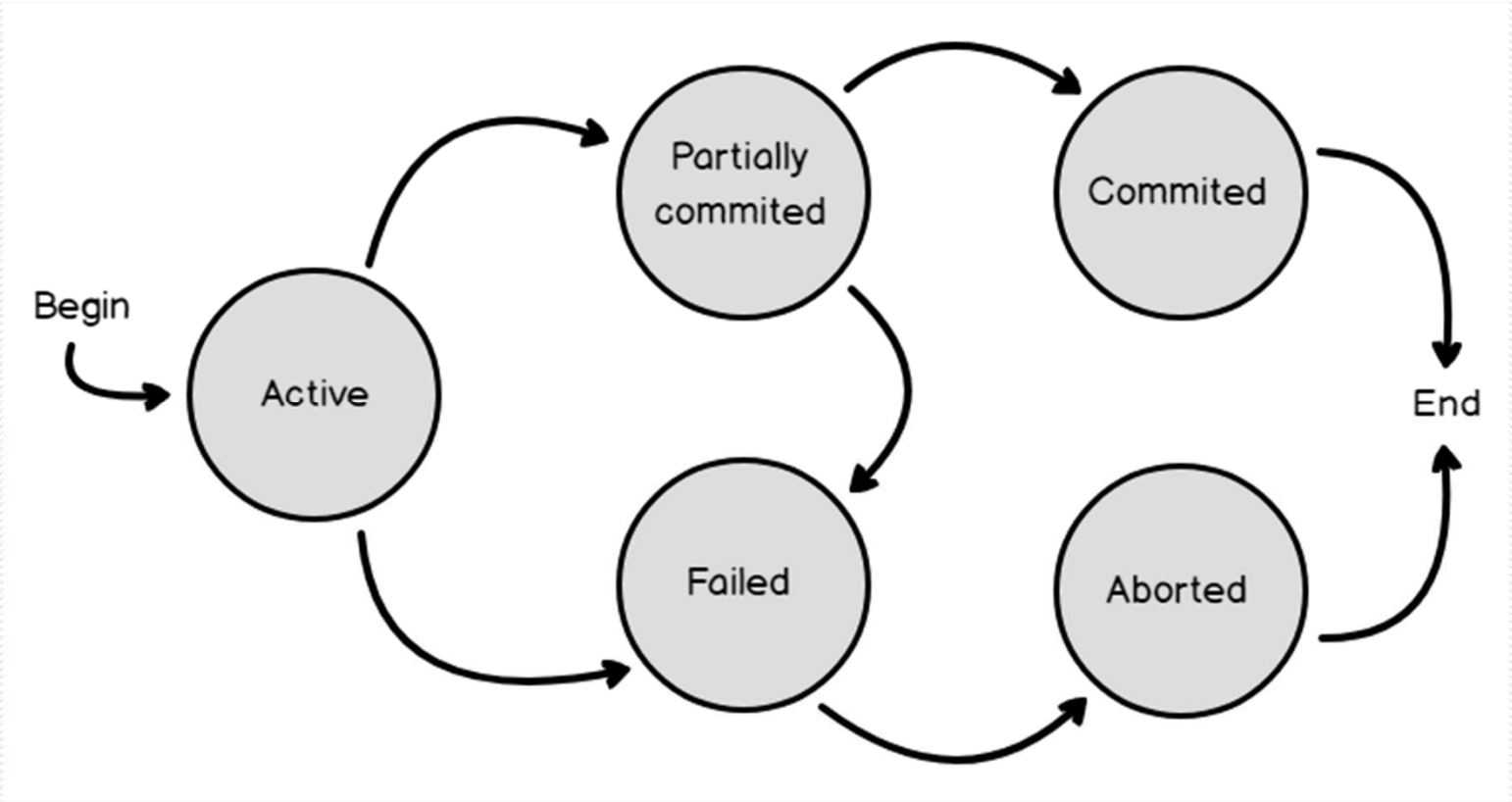
- Atomicity: ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
- Consistency: ensures that the database properly changes states upon a successfully committed transaction.
- Isolation: enables transactions to operate independently of and transparent to each other.
- Durability: ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction states

- A transaction is an atomic unit of work that is either completed in its entirety or not done at all.
- For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

Transaction states (2)

A state transition diagram



Transaction states (3)

- The states of the transaction can be summarized as follows:
  - The running transaction is referred to as the **Active** transaction
  - The transaction that completes its execution successfully without any error is referred to as a **Committed** transaction
  - The transaction that does not complete it is execution successfully is referred to as an **Aborted** transaction

Transaction states (4)

- The transaction that is not fully committed yet is referred to as a **Partially Committed** transaction
- If the transaction does not complete its execution, it is referred to as a **Failed** transaction, that is **Aborted** without being committed
- If the **Partially Committed** transaction completes its execution successfully, it will be **Committed**, otherwise it will be **Failed** then **Aborted**

Processing a transaction

- The following commands are used to process transactions.
  - COMMIT: to save the changes.
  - ROLLBACK: to return to the previous state before changing.
  - SAVEPOINT: create points within the transaction group to ROLLBACK, i.e. to return to that status point.
  - SET TRANSACTION: give a name to a transaction.
- These commands are only used with DML: INSERT, UPDATE and DELETE.

COMMIT command

- Used to save the changes invoked by a transaction to the database.
- Stores all transactions in the Database since the last COMMIT or ROLLBACK command.
- The basic syntax of a COMMIT command is as follows:

COMMIT;

ROLLBACK command

- Used to return transactions to a state before changes have not been saved to the database.
- Can only be used to undo transactions from the last COMMIT or ROLLBACK command.
- The basic syntax:

ROLLBACK;
ROLLBACK TO SavePointName;

SAVEPOINT

- A SAVEPOINT is a point in a transaction when you can undo the transaction to a specific point without having to roll it back to the first state before that change.
- The basic syntax of the SAVEPOINT is as follows:

```
SAVEPOINT  SAVEPOINT_NAME;
```

RELEASE SAVEPOINT command

- Used to delete a SAVEPOINT that you have created.
- The basic syntax

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

- Once a SAVEPOINT has been deleted, you can no longer use the ROLLBACK command to undo the transaction to that SAVEPOINT.

SET TRANSACTION command

- Can be used to initiate a Database Transaction. This command is used to characterize the transaction.
- For example, you can specify a transaction as read only or read write.
- The basic syntax

SET TRANSACTION [READ WRITE | READ ONLY];

Transactions, read and write operation, DBMS buffer

- The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.
- One way of specifying the transaction boundaries is by specifying explicit begin transaction and end transaction statements in an application program

- A read-only transaction
  - do not update the database but only retrieve data
- To simplify, the basic database access operations that a transaction can include are as follows:
  - **read_item(X):** Reads a database item named X into a program variable also named X.
  - **write_item(X):** Writes the value of program variable X into the database item named X

- The basic unit of data transfer from disk to main memory is one block
- Executing a read_item(X) command includes the following steps:
  - Find the address of the disk block that contains item X
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer)
  - Copy item X from the buffer to the program variable named X

- Executing a write_item(X) command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).

- A transaction includes read_item and write_item operations to access and update the database.
- The read-set of a transaction is the set of all items that the transaction reads
- The write-set is the set of all items that the transaction writes.

T_1	T_2
read_item *(X);* *X:=X-N;* write_item *(X);* read_item *(Y);* *Y:=Y+N;* write_item *(Y);*	read_item *(X);* *X:=X+M;* write_item *(X);*

Why Concurrency Control Is Needed

- Problem
  - The Lost Update
  - The Temporary Update (Dirty Read)
  - The Incorrect Summary
  - The Unrepeatable Read

The Lost Update Problem

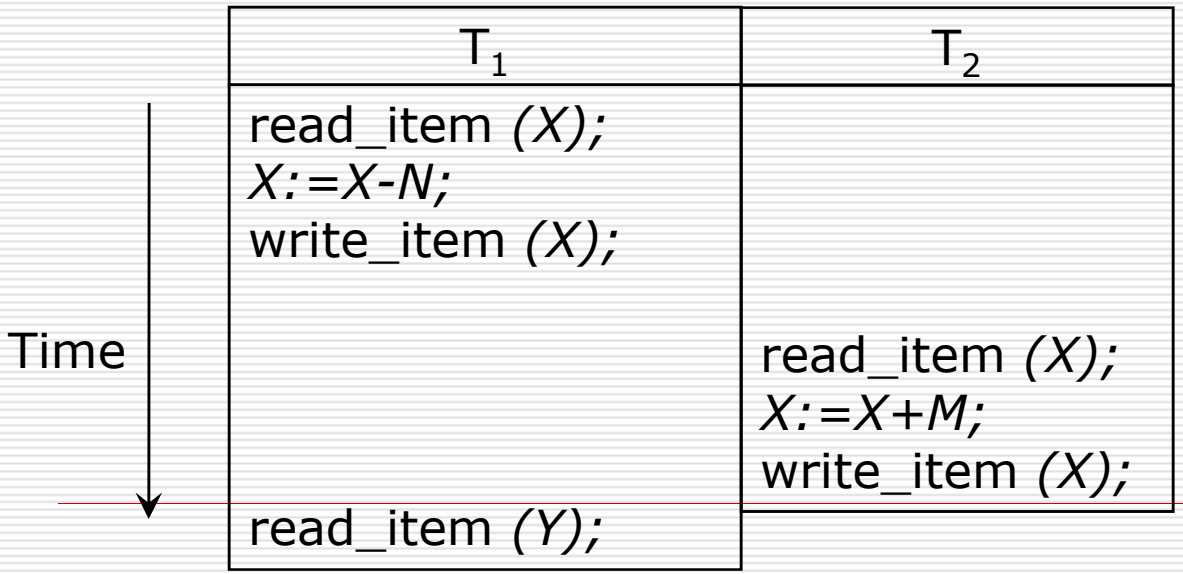
- Occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.

Time

T_1	T_2
read_item *(X);*	
X:=X-N;	
	read_item *(X);*
	X:=X+M;
write_item *(X);*	
read_item *(Y);*	
	write_item *(X);*
Y:=Y+N;	
write_item *(Y);*	

The Temporary Update Problem

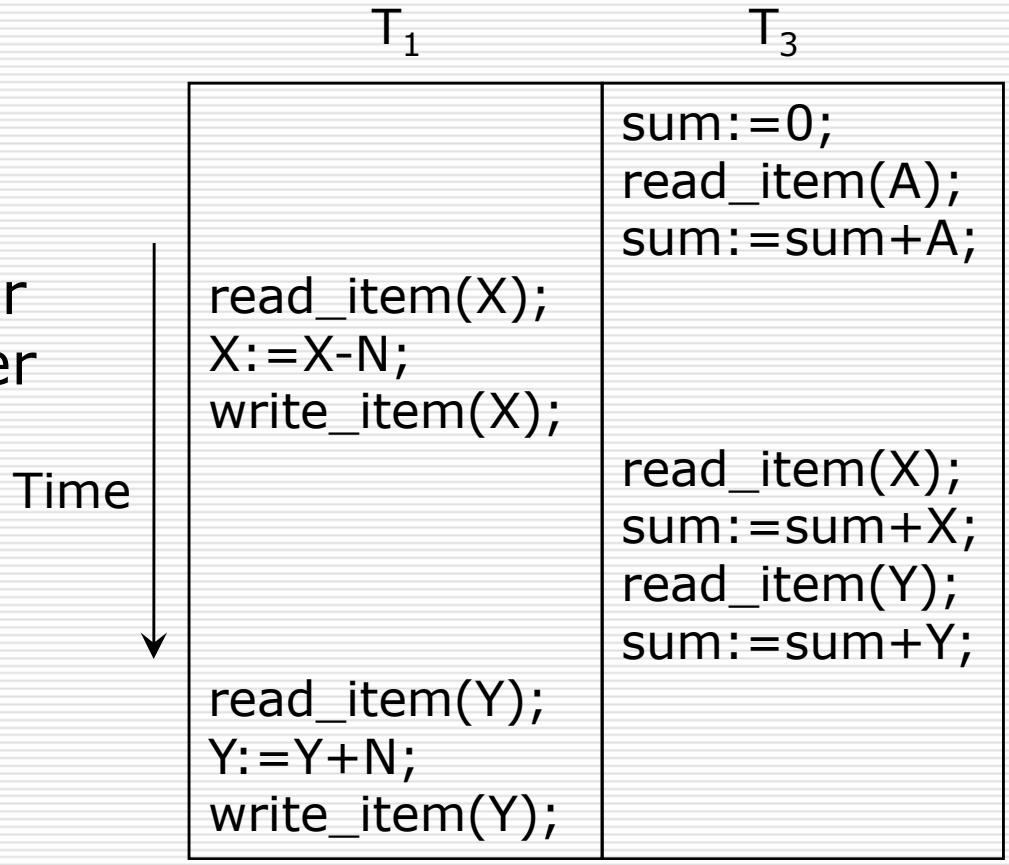
- Occurs when one transaction updates a database item and then the transaction fails for some reason.
- The updated item is accessed by another transaction before it is changed back to its original value.



Transaction *T1 fails and must change the value* of *X back to its old value; meanwhile T2* has read the "temporary" incorrect value of X.

The Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated



T3 reads X after N is subtracted and reads Y before N is added

The Unrepeatable Read Problem

- A transaction T reads an item twice and the item is changed by another transaction T' between the two reads
- Hence, T receives different values for its two reads of the same item.

Why Recovery Is Needed

- The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not. This may happen if a transaction fails after executing some of its operations but before executing all of them.
- There are several possible reasons for a transaction to fail in the middle of execution:
  - A computer failure (system crash)
  - A transaction or system error
  - Local errors or exception conditions detected by the transaction
  - Concurrency control enforcement
  - Disk failure
  - Physical problems and catastrophes

Some examples

- Using an explicit transaction

```
CREATE TABLE ValueTable (id int);
INSERT INTO ValueTable VALUES(1);
INSERT INTO ValueTable VALUES(2);

BEGIN TRANSACTION;
DELETE FROM ValueTable
   WHERE id = 2;
COMMIT;
```

- Rolling back a transaction

```
BEGIN TRANSACTION;
INSERT INTO ValueTable VALUES(3);
INSERT INTO ValueTable VALUES(4);
ROLLBACK;
```

Some examples (2)

- Naming a transaction

```
DECLARE @TranName VARCHAR(20);
SELECT @TranName = 'MyTransaction';
BEGIN TRANSACTION @TranName;
DELETE FROM ValueTable WHERE id = 1;
COMMIT TRANSACTION @TranName;
```

Some examples (3)

- Marking a transaction

```
BEGIN TRANSACTION Del
WITH MARK N'Deleting a row';
DELETE FROM ValueTable WHERE id = 1;
COMMIT TRANSACTION Del;
```

Transaction best practices

- Using the SQL Server transaction helps maintaining the database integrity and consistency. On the other hand, a badly written transaction may affect the overall performance of your system by locking the database resources for long time. To overcome this issue, it is better to consider the following points when writing a transaction:

Transaction best practices (2)

- Narrow the scope of the transaction
- Retrieve the data from the tables before opening the transaction if possible
- Access the least amount of data inside the transaction body
- Do not ask for user input inside the body of the transaction
- Use a suitable mode of transactions
- Use as suitable Isolation Level for the transaction