Database Systems

Transactions, Concurrency Control and Recovery

Jing Sun and Miao Qiao

The University of Auckland

Transactions

- Transaction : a **logical unit** of database processing that must be completed in its entirety to ensure correctness

Examples: Relation *Accounts*(*acctNo*, *balance*)

- UPDATE Accounts SET balance = balance + 100 WHERE accNo = 456;
- UPDATE Accounts SET balance = balance - 100 WHERE accNo = 123;

Other applications:

- Airline reservations
- Banking (credit card transaction)
- Online retail system
- ...

Reading material: Chapter 20 of the textbook.

Transactions

- Transaction : a **logical unit** of database processing that must be completed in its entirety to ensure correctness

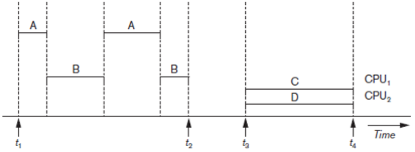
Write transactions in host languages:

- BEGIN TRANSACTION
- READ / WRITE OPERATIONS
  - READ: data retrievals
  - WRITE: updates, insertions and deletions
- COMMIT: the transaction completes successfully, all the updates are permanently made to the database
- ROLLBACK: the transaction ends unsuccessfully, undo the operations in the transaction
- END TRANSACTION

Why we have commit and rollback operations?

Transaction: Concurrency Control

- Transaction processing systems : systems with large databases and hundreds of concurrent users
- Consider two types of operations:
  - computation
  - input and output (I/O)
- One cpu, how to serve multiple users concurrently to minimize the average delay?
  - Interleaved concurrency, e.g., $[t_1, t_2]$, when a process requires I/O, one keeps the cpu busy by switching to execute another process instead of waiting.
- Multiple cpus, how to serve multiple users concurrently to minimize the average delay? E.g., $[t_3, t_4]$. parallel processing + interleaved concurrency



Transaction: Recovery

Types of failures:

- Computer failure (system crash)
- Concurrency control enforcement
- Disk failure
- Physical problems and catastrophes

Transaction: ACID Properties

THE UNIVERSITY OF AUCKLAND Te Whare Wānanga o Tāmaki Makaurau NEW ZEALAND | SCIENCE DEPARTMENT OF COMPUTER SCIENCE

■ Atomicity	Transaction performed in its entirety or not at all
■ Consistency preservation	Takes database from one consistent state to another
■ Isolation	Not interfered by other transactions
■ Durability or permanency	Changes must be persist in the database

Transactions

How to achieve ACID?

- BEGIN TRANSACTION
- READ / WRITE OPERATIONS
  - READ: data retrievals
  - WRITE: updates, insertions and deletions
- COMMIT: the transaction completes successfully, all the updates are permanently made to the database
- ROLLBACK: the transaction ends unsuccessfully, undo the operations in the transaction
- END TRANSACTION

Transaction

Recall that a **database** is a collection of **named data items**.

- A file
- A subtree on the B^+-tree
- A disk block
- A database record
- A field

The size of a data item is called its **granularity**. We assume that each data item has a unique name.

Transaction

For a data item (named) X, a transaction T may carry out two database access operations:

- **read(X)**: Reads a database item named X into a program variable named X_T
- **write(X)**: Write a program variable named X_T to a database item named X

When the context is clear, we omit the subscription T of a program variable.

Discussion: consider the **buffer manager**, what are the steps of read(X) and write(X), respectively?

Transaction

Read(X)

- Find the address of the disk block of X
- Copy the disk block of X to the memory buffer
- Copy X from the buffer to the program variable X_T

Write(X)

- Find the address of the disk block of X
- Copy the disk block of X to the memory buffer
- Copy X from the program variable X_T to the buffer
- Write the updated block from the buffer back to the disk

controlled by buffer manager

Transaction: Concurrency Issues

Given two transactions T_1 and T_2,

(a)

T_1
read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

(b)

T_2
read_item(X);
$X := X + M$;
write_item(X);

The interleaved processing of T_1 and T_2 may lead to two problems:

- **Lost update**
- **Dirty read**

Transaction: Concurrency Issues

Lost update.

(a)

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);	

Plug $X = 10$, $M = 2$ and $N = 3$ in the execution, what will X be after T_1 and T_2?

Transaction: Concurrency Issues

Lost update.

(a)

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);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

Transaction: Concurrency Issues

Dirty read.

(b)

Time	T_1	T_2

↓	read_item(X);
	$X := X - N$;
	write_item(X);	

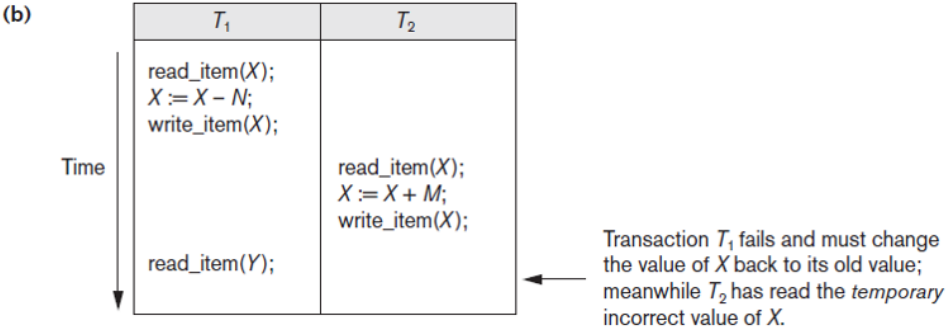
		read_item(X);
		$X := X + M$;
		write_item(X);

	read_item(Y);	

Plug $X = 10$, $M = 2$ and $N = 3$ in the execution, what will happen in T_2 if T_1 rolls back after read_item(Y)?

Transaction: Concurrency Issues

Dirty read. Accessing an updated value that has not been committed is considered a dirty read because it is possible for that value to be rolled back to its previous value. If you read a value that is later rolled back, you will have read an invalid value.



Transaction: Concurrency Issues

Unrepeatable read. Always read committed data items but get two different values in reading the same data item.

T_1	T_2
Read(X)	
	Read(X)
...	
	...
	Write(X)
	Commit
Read(X)	
...	

Transaction: Concurrency Issues

Phantom record. Always read committed values and there is no "unrepeatable read", the query result may have a phantom record t.

- select * from R where c_1: read all database records that satisfy c_1
- insert a record t that satisfies c_1 to R

Time Order	T_1	T_2
1	select * from R where c_1	
		insert a record t that satisfies c_1

Time Order	T_1	T_2
2		insert a record t that satisfies c_1
	Select * from R where c_1	

Transaction: System Log

System log: a sequential, append-only file that tracks the transaction operations.

Associate each transaction with an ID, e.g., T
System log includes the following items:

- [start_transaction, T]
- [write, T, X, old_value, new_value]
- [read, T, X]
- [commit, T]
- [abort,T] (rollback)

Transaction: System Log

System log: to ensure that the database is not affected by failure

- Log buffer
- Log file is backed up periodically
- Commit point, undo and redo operations

Transaction: Commit Point

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

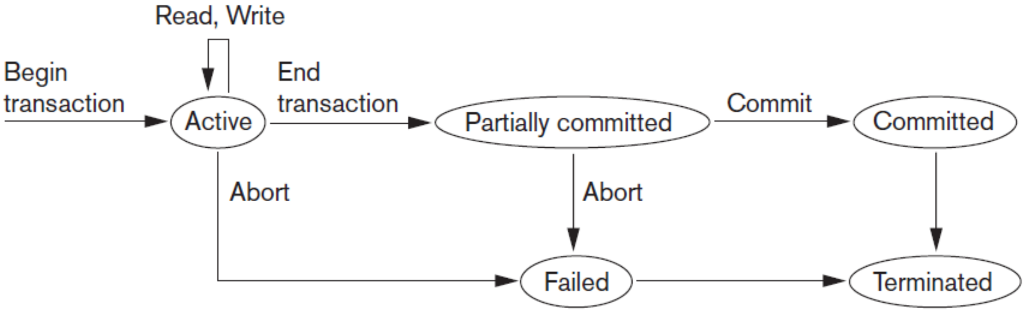
A transaction reaches its **commit point** if:

- All of its database access operations have been executed successfully,
- The effect of the transaction operations to the database has been recorded (**force-write** log buffer before commit point) in the log.

Beyond the commit point, the transaction is called **committed**: its effect must be permanently recorded in the database.

Transaction: States

State transitions via operations:



Properties of Transactions (ACID)

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Atomicity

- A transaction should either be performed in its entirety or not performed at all

Consistency

- Database state: a snapshot of the database (the values of stored data items) at a point in time.
- Consistent state: the database satisfies all the integrety constraints.
- A transaction transforms a database state from a consistent state to another consistent state.

Isolation

- A transaction, though may execute concurrently, should appear as though it is executed in isolation from other transactions.

Durability

- A *committed* transaction has a *permanent* impact on the database. The changes must be kept in case of any failure.

Schedules of Transactions

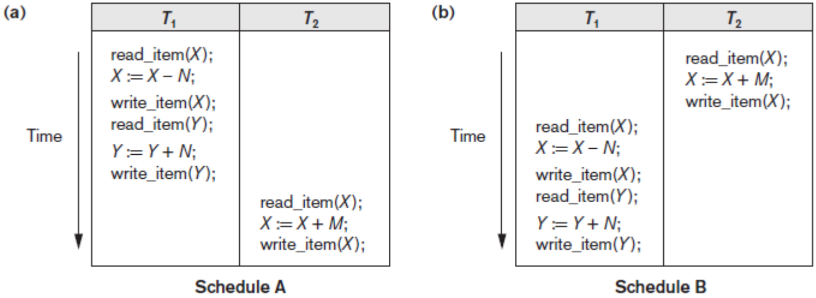
A **schedule** S of n transactions $T_1, T_2, ..., T_n$ is an ordering of **all** the operations of these transactions. For transaction T_i, we denote by

- s_i : [start_transaction, T_i]
- $w_i(X)$: [write, T_i, X, old_value, new_value]
- $r_i(X)$: [read, T_i, X]
- c_i: [commit, T_i]
- a_i: [abort, T_i]

Serial Schedules

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

- Serial Schedules: place simultaneous transactions in series (T_1, T_2 or T_2, T_1)



- Problem: Limit concurrency by prohibiting interleaving of operations
- Solution: determine which schedules are equivalent to a serial schedule

Serializable Schedules

Serializable schedule

Given n transactions $T_1, T_2, \cdots, T_n$, S is a serializable schedule of the n transactions if it is **equivalent** to some serial schedule of same n transactions.

Discussion: Can we define equivalent schedules as the schedules that produce the same final state of the database? Why?

S_1

read_item(X);
$X := X + 10$;
write_item(X);

S_2
read_item(X); $X := X * 1.1$; write_item (X);

Conflicting Operations

Conflicting Operations

Two operations are conflicting if changing their order can result in a different outcome.

Two operations in a schedule are said to conflict if

- They belong to different transactions
- They access the same item X
- At least one of the operations is write(X)

Conflicting operations (read-write conflict and write-write conflict):

- T_1 reads an item written by T_2 : T_2 should terminate before T_1 ($T_2 < T_1$)
- T_1 writes an item read by T_2: $T_2 < T_1$
- T_1 writes an item written by T_2: $T_2 < T_1$

Serializable Schedules

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Conflict equivalence

Schedule S is conflict equivalent with schedule S' if the relative order of any two conflicting operations is the same in both schedules

A schedule S is **serializable** if it is **conflict equivalent** to some **serial schedule** S'.

Precedence Graph

Find conflict operations in a schedule:

- For each write operation, find the closest (both sides) read/write operations on the same data item from other transactions.

Discuss: Are the following schedules serializable?

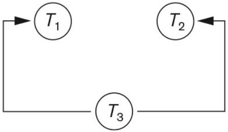
- $S_a : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y)$
  - $r_2(X); w_1(X);$ on X
  - $w_1(X); w_2(X);$ on X
- $S_b : r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); w_1(Y)$
  - $w_1(X); r_2(X);$ on X
  - $w_1(X); w_2(X);$ on X
  - remark: equivalent to the serial schedule of T_1, T_2.

Precedence Graph

The **Precedence Graph** is a directed graph that consists of a set of nodes $V = (T_1, T_2, \cdots, T_n)$ and a set E of edges: an edge from T_i to T_j exists if there are two conflicting operations o, o' in the schedule with

- o from T_i
- o' from T_j

$$S : r_3(X); w_3(X); r_1(X); r_2(X);$$



Equivalent serial schedules

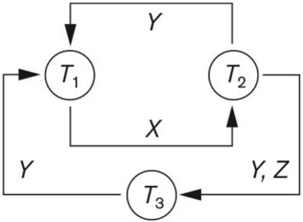
$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

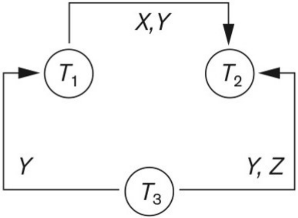
Precedence graph

A schedule is serializable if and only if its precedence graph has no cycle.

Discussion: Are the schedules serializable? What serial schedules are they equivalent to?



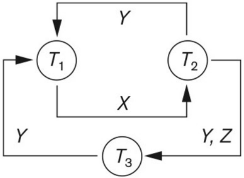
(a)



(b)

Precedence graph

Discussion: Are the schedules serializable? What serial schedules are they equivalent to?

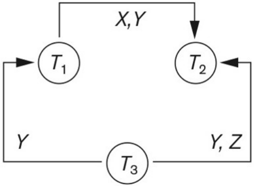


Equivalent serial schedules

None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Schedules of Transactions

(b)

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Time

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

What is the problem of the following schedule (with commit and abort operations)?

- $r_1(X); w_1(X); r_2(X); w_2(X); c_2; a_1$

THE UNIVERSITY OF AUCKLAND Te Whare Wānanga o Tāmaki Makaurau NEW ZEALAND | SCIENCE DEPARTMENT OF COMPUTER SCIENCE

Recoverable Schedule

A schedule is **recoverable** if

- Each transaction commits only after all transactions whose changes they read commit. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i

Is there an order of the terminations (commit/abort operations) of T_1 and T_2 such that S_a (S_b) is recoverable?

- $S_a : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y).$
- $S_b : r_1(X); w_1(X); r_2(Y); r_2(X); w_2(Y); r_1(Y).$

Isolation Level

- Dirty read, Nonrepeatable read, Phantoms

Table 20.1 Possible Violations Based on Isolation Levels as Defined in SQL

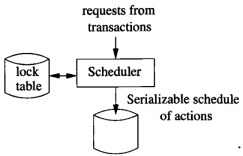
	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

SQL:

- SET TRANSACTION READ ONLY ISOLATION LEVEL READ COMMITTED;
- SET TRANSACTION READ WRITE ISOLATION LEVEL READ UNCOMMITTED;

Enforce Serializability by Locks

- Lock: a variable associated with a data item describing the status.
- Lock table: a table of locked items (normally implemented with a hash table)



- Consistency of Transactions: Actions and locks must relate in the expected ways:
  - A transaction can only read or write an element if it previously was granted a lock on the element and hasn't yet released the lock.
  - If a transaction locks an element, it must later unlock that element.
- Legality of Schedules (applied to binary locks with two status, locked and unlocked): Locks must have their intended meaning: no two transactions may have locked the same element without one having first released the lock.

Binary Locks

Two states (values) lock(X) =

- Locked (1) — Item cannot be accessed
- Unlocked (0) — Item can be accessed when requested

Two operations

- lock_item(X)
- unlock_item(X)

- The lock/unlock operations are atom operations.
- Transactions hold the lock on one data item in a mutually exclusive way.
- At most one transaction can hold the lock on an item at a given time — too restrictive for database items

Shared/Exclusive (Read/Write) Locks

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Shared/exclusive or read/write locks:

- Read operations on the same item are not conflicting
- Must have an exclusive lock to write

Three locking operations in a transaction T:

- read_lock(X): attempt to request a read lock
- write_lock(X): attempt to request a write lock
- unlock(X): release the lock T currently holds on X

Serializability

Using locking techniques does not guarantee serilizability.

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

Two-Phase Locking

Two-phase locking protocol: All locking operations precede the first unlock operation in the transaction.

Phases

- Expanding (growing) phase — New locks can be acquired but none can be released
- Shrinking phase — Existing locks can be released but none can be acquired

Two-Phase Locking

T_1'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);

T_2'

read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
$Y := X + Y$;
write_item(Y);
unlock(Y);

Any schedule produced by two-phase locking protocol is serializable.

How to prove this?

Two-Phase Locking

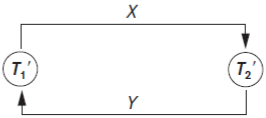
Wait-for graph: An edge from T_i to T_j if T_i is waiting for a lock currently hold by T_j.
Deadlock: There is a cycle in the wait-for graph.

(a)

Time ↓

T_1'	T_2'
read_lock(Y); read_item(Y);	
	read_lock(X); read_item(X);
write_lock(X);	
	write_lock(Y);

(b)



Solutions:

- Detect the cycles in wait-for graph and choose a victim to break a cycle — the victim transaction will be undone and then restart.
- Set up a time limit L — if a transaction waits longer than L seconds then undo the transaction and restart.

Starvation.

Two-Phase Locking

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Any schedule produced by two-phase locking protocol is serializable.

How to prove this?

Prove with the following properties:

- There must be a moment (m_i) when a transaction T_i holds all of its locks.
- T_i holds Lock(X) of item X for a consecutive period of time $[start_i(X), end_i(X)]$.
- $m_i \in [start_i(X), end_i(X)]$.
- If there are two conflicting operations on X from T_i and T_j, then $[start_i(X), end_i(X)]$ and $[start_j(X), end_j(X)]$ are mutual exclusive.
- In the precedence graph: if there is an edge from T_i to T_j, then $m_i < m_j$

there is no cycle in the precedence graph.

To ensure a serializable schedule — locking techniques

Recall that a **database** is a collection of **named data items** with different **granularities**.

- A file
- A subtree on the B^+-tree
- A disk block
- A database record
- A field

Thank you for your attention!

Any questions?