Transactions

CSC365

Transaction

A **transaction** is a collection of operations on a database that must be executed *atomically*; that is, either all operations are performed or none.

This is an important concept when considering database failure, as well as concurrent usage of a single database (for example, many users interacting with a bank's database, a flight reservation system, etc.)

ACID Properties of Transactions

Atomic	Each transaction must be "all or nothing." if one part of the transaction fails, then the *entire* transaction fails, and the database state is left unchanged.
Consistent	Any transaction will bring the database from one valid state to another (valid meaning that no constraints are violated)
Isolated	Concurrent execution of transactions results in a system state that would be obtained if transactions were executed in isolation (**serially)**.
Durable	Once a transaction has been committed, it will remain so.

Haerder, T.; Reuter, A. (1983). "Principles of transaction-oriented database recovery". *ACM Computing Surveys.* https://dl.acm.org/citation.cfm?doid=289.291

Transaction Example

Time

User *1* searches for open seats on flight *1234*, finds seat *22A* empty	
	User *2* searches for open seats on flight *1234*, finds seat *22A* empty
User *1* chooses seat *22A*, application sets this seat's status to occupied in database	
	User *2* chooses seat *22A*

Implementation Details

To support atomicity and durability:

Write-ahead logging: all changes are written to a log before they are applied to the database. In case of failure, the system can determine which operations succeeded (perhaps partially) or failed.

Copy on write (shadow paging): When a page of data needs to be modified, a copy is made and changes are applied. When ready, all references to the original page are changed to the new page.

Implementation Details

Two fundamental approaches to handling isolation and concurrency:

Locking: A transaction gains a lock on data to be updated (eg. table or row(s)) which prevents other transactions from making changes until the lock is released.

Multiversion Concurrency Control (MVCC): A transaction sees the state of the data as it was when the transaction started. Database maintains multiple versions of data items. Typically implemented using timestamps or transaction IDs.

Transactions in SQL

SQL includes a few commands used to define **transaction boundaries**:

START TRANSACTION - begin a transaction, which continues until a COMMIT or ROLLBACK (Note: some RDMSs use BEGIN [TRANSACTION])

COMMIT - end a transaction, making changes permanent

ROLLBACK - revert any changes made during the current transaction

The Sequence of Operations in a Transaction

Transaction Example

<table>
<tr><th>Client 1</th><th>Client 2</th></tr>
<tr><td>
<pre>
START TRANSACTION;

SELECT SUM(amount) AS Balance FROM cust_transfers
WHERE cust_id = 1234 AND account_num = 8001;

INSERT INTO cust_transfers (cust_id, account_num, date,
amount, memo) VALUES (1234, 8001, CURRENT_DATE, -450,
'Vacation time');

UPDATE cust_balance SET balance = (SELECT SUM(amount)
FROM cust_transfers WHERE cust_id = cust_balance.cust_id
AND account_num = cust_balance.account_num);

SELECT SUM(amount) AS Balance FROM cust_transfers
WHERE cust_id = 1234 AND account_num = 8001;

-- COMMIT;
-- ROLLBACK;
</pre>
</td><td>
<pre>
SELECT SUM(amount) AS Balance FROM cust_transfers
WHERE cust_id = 1234 AND account_num = 8001;

INSERT INTO cust_transfers (cust_id, account_num, date,
amount, memo) VALUES (1234, 8001, CURRENT_DATE, -600, 'Cash
Out!');

UPDATE cust_balance SET balance = (SELECT SUM(amount) FROM
cust_transfers WHERE cust_id = cust_balance.cust_id AND
account_num = cust_balance.account_num);

SELECT * FROM cust_transfers WHERE cust_id = 1234;
SELECT * FROM cust_balance;
</pre>
</td></tr>
</table>

Set AutoCommit to False

By default, MySQL sets the autocommit to ON.

- Set it to False:
    - mysql> SET AUTOCOMMIT = 0;
- In Java
    - Call setAutoCommit method of Connection object
        - **connection**.setAutoCommit(**false**);

DBMS Scheduler

A scheduler manages the sequencing of read & write requests. Goals:

1. Maintain **serializable** schedule
2. Allow the greatest possible **concurrency**

Serial versus Serializable

It is impractical to require that transactions run **serially** (one at time, with no time overlap)

Instead, databases support parallelism by assuring **serializability**: even if individual operations in transactions are interleaved, the result *looks to users* as if the transactions were executed serially.

Two Sample Transactions

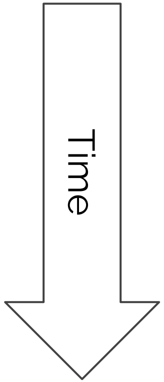
T_1	T_2

`Read(A)`
`A = A - 50`
`Write(A)`
`Read(B)`
`B = B - 50`
`Write(B)`
	`Read(A)`
	`A = A / 2`
	`Write(A)`
	`Read(B)`
	`B = B / 2`
	`Write(B)`

Each transaction (individually) preserves a consistency requirement: A = B

Serial Schedule

A = 250, B = 250



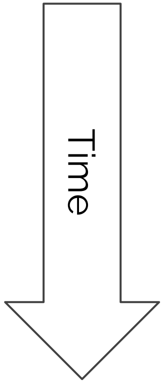
T_1	T_2

`Read(A)`
`A = A - 50`
`Write(A)`
`Read(B)`
`B = B - 50`
`Write(B)`	

	`Read(A)`
	`A = A / 2`
	`Write(A)`
	`Read(B)`
	`B = B / 2`
	`Write(B)`

A = 100, B = 100

Interleaved, Serializable



A = 250, B = 250

T_1	T_2

`Read(A)`
`A = A - 50`
`Write(A)`	

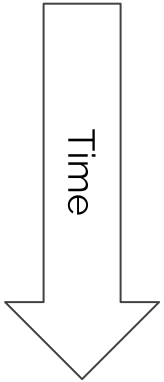
	`Read(A)`
	`A = A / 2`
	`Write(A)`

`Read(B)`
`B = B - 50`
`Write(B)`	

	`Read(B)`
	`B = B / 2`
	`Write(B)`

A = 100, B = 100

Interleaved, NOT Serializable



A = 250, B = 250

T_1	T_2

`Read(A)`
`A = A - 50`
`Write(A)`	

	`Read(A)`
	`A = A / 2`
	`Write(A)`

	`Read(B)`
	`B = B / 2`
	`Write(B)`

`Read(B)`
`B = B - 50`
`Write(B)`	

A = 100, B = 75

Transaction Notation

Transaction: sequence of reads and writes (insert, update, *or* delete) on database objects. Example: `r(A); w(A); r(B); w(B);`

Schedule : sequence of reads/writes performed by a *collection* of transactions.

Serial schedule involving two transactions:

$r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);\ r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);$

Given a non-serial schedule, how do we determine whether it is serializable?

Conflict Serializable

Determining whether a schedule is serializable is an NP-complete problem.

Conflict serializability ia an easier-to-determine special case of serializability. Any schedule that is conflict serializable is also serializable. However, a schedule may be serializable but *not* conflict serializable..

We define this in terms of **conflicts**. What constitutes a conflict?

Conflicts

Two actions in a schedule conflict if *all* of these are true:

1. They are from **different transactions**
2. One (or both) of the actions is a **write** operation
3. Access to the **same object**

Actions	Conflict?
$r_1(A);\ r_2(B)$	False
$r_1(A);\ w_2(B)$	False
$r_1(A);\ w_1(A)$	False
$r_1(A);\ w_2(A)$	True
$r_1(A);\ r_2(A)$	False
$w_1(A);\ w_2(B)$	False

Conflict Equivalence

We need a way to test whether a schedule is conflict serializable (ie. equivalent in its effect to a serial schedule)

How to test? Repeatedly swap contiguous, non-conflicting actions, try to arrive at a serial schedule.

$r_1(A); \ w_1(A); \ r_1(B); \ r_2(A); \ w_2(A); \ w_1(B); \ r_2(B); \ w_2(B);$

Is the schedule above conflict serializable?

Important: when performing a "swap test," changing the order of actions within a single transaction is never permitted!

Precedence Graph

Another option: a schedule is conflict-serializable if and only if its **precedence graph** of committed transactions is *acyclic*.

Nodes are transactions (T_1, T_2, … T_n)

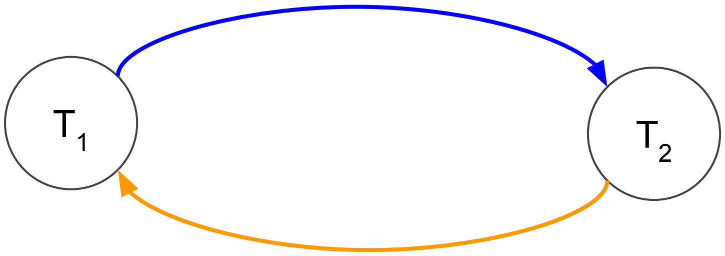
Directed edge from T_i to T_j if T_i <u>precedes</u> and <u>conflicts</u> with one of T_j's actions.
Note: "precede" includes *non-contiguous actions*.

Precedence Graph Example

Schedule: $r_1(A);\ r_2(B);\ \underline{w_1(A);\ r_2(A);}\ \underline{w_2(B);}\ w_2(A);\ \underline{r_1(B);}\ w_1(B);$

Edge from T_1 to T_2 due to these two conflicting actions.

Edge from T_2 to T_1 due to these two conflicting actions.

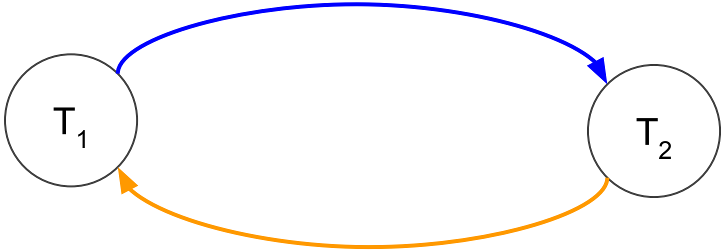


Resulting precedence graph contains a cycle, therefore the schedule is **not conflict serializable.**

$r_1(A); r_2(B); w_1(A); r_2(A); w_2(B); w_2(A); r_1(B); w_1(B);$

Time

T1	T2
r(A)	
	r(B)
w(A)	
	r(A)
	w(B)
	w(A)
r(B)	
w(B)	



Precedence Graphs

$r_1(A); \ r_2(B); \ w_1(A); \ r_2(A); \ w_2(B); \ w_2(A); \ r_1(B); \ w_1(B);$

$w_3(A); \ r_1(A); \ w_1(B); \ r_2(B); \ w_2(C); \ r_3(C);$

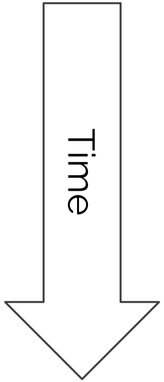
$w_1(B); \ w_2(B); \ w_2(A); \ w_1(A); \ w_3(A);$

Is this schedule serializable?

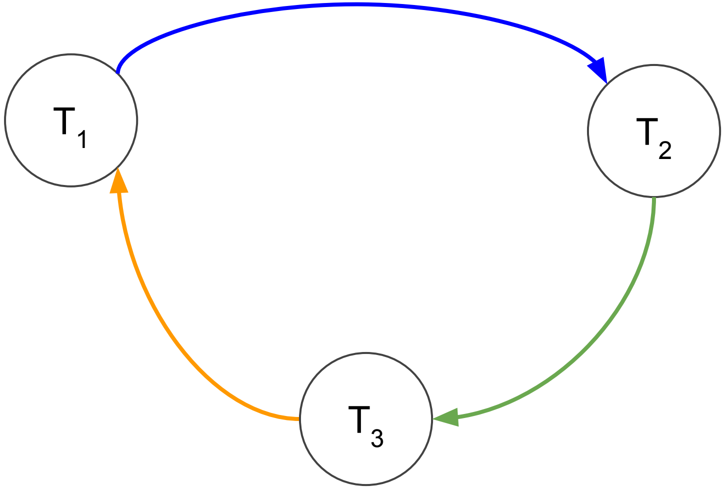
If precedence graph contains a cycle, the schedule is not conflict serializable.
In other words, the schedule is not equivalent to *any* serial schedule.

Is this schedule serializable?

$w_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_2(C)$; $r_3(C)$;

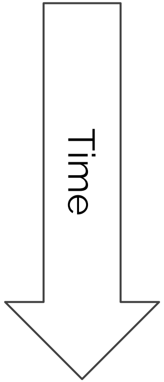


T1	T2	T3
		w(A)
r(A)		
w(B)		
	r(B)	
	w(C)	
		r(C)

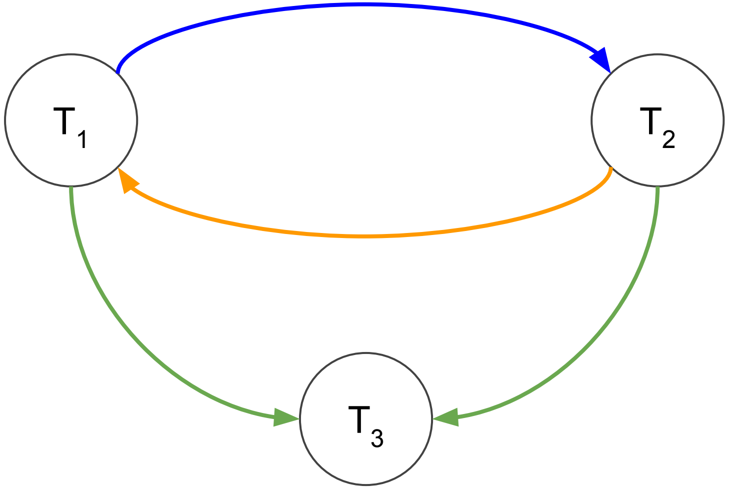


Is this schedule serializable?

$w_1(B)$; $w_2(B)$; $w_2(A)$; $w_1(A)$; $w_3(A)$;



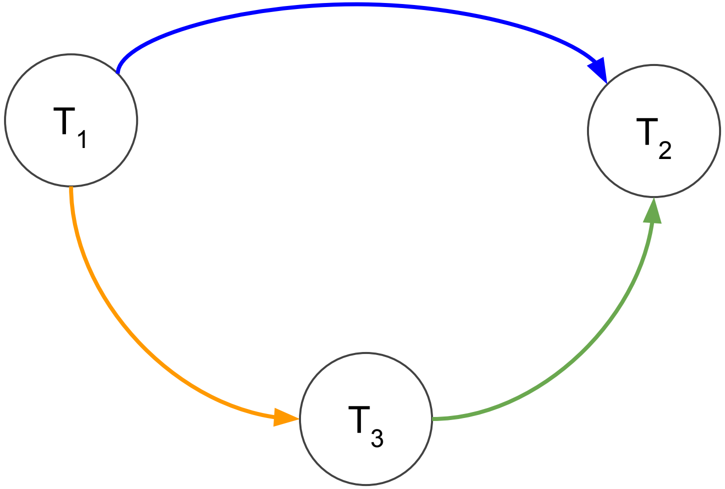
T1	T2	T3
w(B)		
	w(B)	
	w(A)	
w(A)		
		w(A)



Is this schedule serializable?

Time

T1	T2	T3
r(A)		
r(B)		
	r(A)	
	r(C)	
w(B)		
		r(B)
		r(C)
		w(B)
	w(A)	
	w(C)	



Conflict Serializable

Testing for Conflict Equivalence

Two possible methods to determine whether a schedule is conflict equivalent to a serial schedule:

1. Trial and error: repeatedly swap pairs of non-conflicting actions until you arrive at a serial schedule
2. Build a precedence graph. If a cycle is present, you do not have a conflict serializable schedule.

Relatively easy with a small number of transactions / actions. Rather expensive as the complexity grows.

Two-Phase Locking

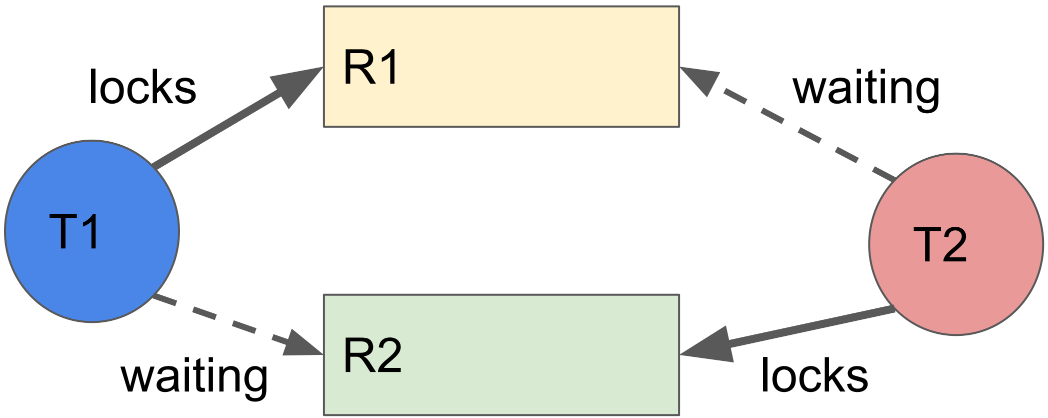
As an alternative to potentially expensive tests for conflict equivalence, many database implementations support locking to enforce conflict serializability

Specifically, **two-phase locking** in which each data item is locked by a transaction before access. Every transaction has a locking phase and an unlocking phase; once the unlocking phase has begun, *no additional locks may be obtained*.

Disambiguation: the term *two-phase commit* (2PC) refers to a different concept which applies in distributed systems

Deadlocks

If two transactions are halted, waiting for each other, a **deadlock** occurs. The DBMS (or DBA) must terminate one of the transactions to allow others to proceed. The terminated transaction is then typically restarted.



MySQL Specifics

- One of the strengths of MySQL is that it supports more than one storage engine.
- Storage engines you can use with MySQL (more engines available)
  - MyISAM
    - The original storage engine used by MySQL - caches only index
    - Supports table lock but not record lock
    - Default storage engine before v5.5
  - INNODB
    - Relatively new - caches both index and data
    - Supports record lock
    - Default storage engine since v5.5 (if you do not specify, this is the engine used)

Optimistic Concurrency Control

In situations where conflicts are rare (ie. users do not often attempt to access the same data) the expense of locks can be avoided using a technique known as "optimistic concurrency control"

1. Record a timestamp/row version before making changes
2. While applying changes, determine whether a conflict exists by comparing expected versus actual timestamp/version
3. Rollback in case of conflict, commit otherwise

Optimistic CC Example

Simple SQL implementation of optimistic concurrency control: add a `version` column that holds an integer. With every modification, confirm the correct version and increment the version;

```
UPDATE equipment_rental
SET checked_out_by = 'OP', version = version + 1
WHERE item = 'Skis' AND version = 2;
```

Optimistic Concurrency Control

Optimistic concurrency control is used in situations where contention is low. Many database mapping frameworks offer built-in support for optimistic concurrency control.

In cases where contention is high, however, the cost of frequently restarting transactions can outweigh the benefits of an optimistic approach.

Concurrency Control / Transactions: Summary

- ACID properties
- Schedules: serial, serializable, conflict serializable
- Testing for conflict serializability
  - Swap actions
  - Precedence graph
- Locking
  - Two-phase locking (guarantees conflict serializability)
  - Deadlocks
  - Optimistic locking
- MVCC: improve concurrency of reads