ACID, Transaction Management and Conccurrency Control

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Lecture topics

- ACID.
- Transaction Management.
- Concurrency Control.

Transactions and Concurrency

Reasons

- *Concurrent* execution essential for good DBMS performance.
- Disk accesses frequent, relatively slow. Important to keep the cpu busy by working on several user programs concurrently.
- Client application may carry out many operations on the data retrieved from the database.
- DBMS is only concerned about what data is read/written from/to the database.
- A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes .

Transactions and Concurrency

Transactions and Concurrency

- Users submit transactions, and can think of each transaction as executing by itself.
  - *Concurrency* is achieved by the DBMS: reads/writes of DB objects of various transactions.
  - transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
  - DBMS does not need to understand the semantics of the data. (e.g. how the interest on a bank account is computed).
- Issues: Effect of interleaving transactions, and crashes.

Transactions and Concurrency

The ACID properties

- A RDBS ensures this four properties of a transaction:
  - **Atomicity**: states that database modifications must follow an all or nothing rule.

Transactions and Concurrency

The ACID properties

- A RDBS ensures this four properties of a transaction:
    - **Atomicity**: states that database modifications must follow an all or nothing rule.
    - **Consistency**: states that only valid data will be written to the database.

Transactions and Concurrency

The ACID properties

- A RDBS ensures this four properties of a transaction:
  - **Atomicity**: states that database modifications must follow an all or nothing rule.
  - **Consistency**: states that only valid data will be written to the database.
  - **Isolation**: requires that multiple transactions occurring at the same time not impact each others execution.

Transactions and Concurrency

The ACID properties

- A RDBS ensures this four properties of a transaction:
    - **Atomicity**: states that database modifications must follow an all or nothing rule.
    - **Consistency**: states that only valid data will be written to the database.
    - **Isolation**: requires that multiple transactions occurring at the same time not impact each others execution.
    - **Durability**: ensures that any transaction committed to the database will not be lost.

Transactions and Concurrency

Atomicity of Transactions

- A transaction either commit after completing all its actions or abort after executing some actions.
- A user can think of a transaction as always executing all its actions in one step, or not executing any actions at all.
- DBMS logs all actions so that it can undo the actions of aborted transactions.

Transaction

```
BEGIN;
UPDATE ships SET name = 'Alpha'
        WHERE name = 'Oleg';
SAVEPOINT my_savepoint;
UPDATE ships SET integrity = 30
        WHERE name = 'Alpha';
-- oops ... forget that and update Beta
ROLLBACK TO my_savepoint;
UPDATE ships SET integrity = integrity + 10
        WHERE name = 'Beta';
COMMIT;
```

When are those updates valid states for other actions?

Example of a Transaction

Transactions

- We simplify transaction queries for readability in our next example

Example of a Transaction

Transactions

- We simplify transaction queries for readability in our next example
- Consider those two transactions:
  - T1: BEGIN ship1.energy = ship1.energy - 10, ship2.shields = ship2.shields + 10 COMMIT
  - T2: BEGIN ship1.energy = 1.05 * ship1.energy, ship2.shields = 1.05 * ship2.shields COMMIT
- The first transaction is using `ship1` energy to recharge `ship2` shields.
- Both the energy and the shields are being recharged by a nearby generator which recharge by 5% of the total amount.
- How are those transaction scheduled?

Example of a Transaction

Transactions

- Possibility 1:

T1	T2
s1.energy = s1.energy - 10	
	s1.energy = 1.05 * s1.energy
s2.shield = s2.shields + 10	
Commit	
	s2.shields = 1.05 * s2.shields
	Commit

Example of a Transaction

Transactions

- Possibility 2:

T1	T2
s1.energy = s1.energy - 10	
	s1.energy = 1.05 * s1.energy
	s2.shields = 1.05 * s2.shields
	Commit
s2.shield = s2.shields + 10	
Commit	

HOGESCHOOL ROTTERDAM

Example of a Transaction

DBMS interleaved schedule

- DBMS View of the first schedule:

T1	T2
R(s1)	
W(s1)	
	R(s1)
	W(s1)
R(s2)	
W(s2)	
Commit	
	R(s2)
	W(s2)
	Commit

- DBMS View of the second schedule:

T1	T2
R(s1)	
W(s1)	
	R(s1)
	W(s1)
	R(s2)
	W(s2)
	Commit
R(s2)	
W(s2)	
Commit	

Conflicts

Non-equivalent transaction results

- Assume `s1.energy = 50`, `s2.shields = 70`.
- At the end of the first schedule `s1.energy = 52.5` and `s2.shields = 84`.
- At the end of the second schedule `s1.energy = 40` and `s2.shields = 83.5`
- The two schedules are not equivalent.

Conflicts

Conflicts of interleaved execution

- The different schedules of transactions might ignore the operations finalized by other transactions.
- Reason: conflicts with Read/Write operations
- Conflicts are caused by the isolation property of transactions.
- The isolation property might cause an anomaly in the database.
- Four possible combination of Read/Write operations: Read/Read, Write/Read, Read/Write, Write/Write

Conflicts

Read/Read

- Reading does not alter the state of the database.
- Just reading is always safe!

Conflicts

Write/Read

- Dirty read:

T1	T2
A = s1.energy	
s1.energy = A - 10	
	A = s1.energy
	s1.energy = A * 0.5
	B = s2.shields
	s2.shields = B * 0.5
	Commit
B = s2.shields	
s2.shields = B * 0.5	
Commit	

Conflicts

Write/Read (Dirty read)

- Transaction 1 reads and writes the energy. Does not commit
- Meanwhile Transaction 2 reads the energy and the shields, writes them, and then commit.
- Transaction 1 reads and writes the shields and commits.
- **Problem:** Transaction 2 reads the value of the energy before Transaction 1 commits.
- The changes of Transaction 2 are overwritten by Transaction 1.

Conflicts

Read/Write

- Unrepeatable reads:

T1	T2
A = s1.energy do something else with A...	

	A = s1.energy
	s1.energy = A * 0.5
	Commit

A = s1.energy
s1.energy = A - 10
Commit	

Conflicts

Read/Write (Unrepeatable reads)

- Transaction 1 reads the energy and computes the increment, saving it in variable B.
- Transaction 2 reads the energy, writes it and commits.
- Transaction 1 reads the energy again getting a different result.
- **Problem:** Transaction 2 reads the same value and gets two different results.

Conflicts

Write/Write

- Lost update:

T1	T2
	s1.energy = 1000
s2.shields = 2000	
	s2.shields = 1000
	Commit
s1.energy = 2000	
Commit	

Conflicts

Write/Write (Lost update)

- T2 commits first and overwrites T1 changes to shields.
- **Problem:** the update made by one of the transactions is lost.

Conflicts

Conflicting operations

- Read/Read non conflicting. R(A)-R(A)
- Write/Read on the same object conflicting. W(A)-R(A)
- Read/Write on the same object conflicting. R(A)-W(A)
- Write/Write on the same object conflicting. W(A)-W(A)

Serializability

Serializability

- We want to get rid of the side effects of the conflicts.
- Serial executions never conflict.
- Transform an interleaved execution into an execution that is equivalent to a serial execution.

Serializability

Conflict serializability

- Two schedules are conflict equivalent if we can transform to one another by swapping non-conflicting operations.
- A schedule is conflict serializable if it is conflict equivalent to a serial schedule.
- The schedule below is conflict equivalent to the serial execution of T1;T2, by making the read/write on B in T1 before the read/write on A in T2.

T1	T2
A = s1.energy	
s1.energy = s1.energy - 10	
	A = s1.energy
	s1.energy = s1.energy * 0.5
B = s2.shields	
s2.shields = s2.shields + 10	
	B = s2.shields
	s2.shields = 0.5 * s2.shields
Commit	
	Commit

Serializability

Conflict Serializability

- Conflict serializability is not always achievable.
- The following schedule is not conflict serializable. We cannot swap any of the operations to obtain one of the serial executions T1;T2 or T2;T1 because they conflict.

T1	T2
A = s1.energy	
	s1.energy = 1000
s1.energy = 2000	
Commit	
	Commit

Locking

Strict Two Phase-Locking (2PL)

- We define two locks on objects, *Shared* and *Exclusive*.
- We denote with X(A) an exclusive lock on an object A. We denote with S(A) a shared lock.
- The Strict 2PL scheduler ensures that no anomalies arise from the interleaved executions of the transactions.

Locking

2PL rules

- A transaction that wants to read an object requests a shared lock.
- A transaction that wants to write an object requests an exclusive lock.
- A transaction is allowed to release all the locks only after it commits or aborts (so rollback is executed before another transaction can use the data).

Locking

2PL lock manager

- Locks are managed with the following rules

Request	Free	Shared	Exclusive
Shared	Shared	Shared	Denied
Exclusive	Exclusive	Denied	Denied
Unlock	Free	Maybe*	Yes

* It is freed only if all the transactions sharing the read lock can release it.

Locking

2PL Example

- 2PL schedule of RW conflict

T1	T2
X(s1)	
A = s1.energy	
s1.energy = A - 10	
	RequestLock(s1)
X(s2)	
B = s2.shields	
s2.shields = B * 0.5	
Commit	
Unlock(s1)	
Unlock(s2)	
	X(s1)
	A = s1.energy
	s1.energy = A * 0.5
	X(s2)
	B = s2.shields
	s2.shields = B * 0.5
	Commit
	Unlock(s1)
	Unlock(s2)

Locking on tables

- Query for T1:

```
SELECT type
FROM ships
WHERE firepower = 1500
```

- Query for T2

```
UPDATE ships
SET type = "Star Destroyer"
WHERE firepower >= 1500
```

- Set a shared lock for T1 on the entire table, or
- set a shared lock only on the rows with `firepower = 1500` (more concurrency).

Phantom update

- Shared lock on rows do not prevent another transaction to add data to a table.
- Query for T3

```
INSERT INTO ships
VALUES ('Alecto', 'Star Destroyer',
    1500, 5000, 5000)
```

- T1 does not see the row added by T3 (phantom update).

Locking

Transaction isolation

- Allows to control the concurrency level and exposure to other transactions

Level	**Dirty Read**	**Unrepeatable read**	**Phantom update**
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Transaction in PostgreSQL

- BEGIN TRANSACTION begins a transaction.
- ISOLATION LEVEL sets the isolation level as seen above.
- COMMIT commits the transaction.
- **Example:**

```
BEGIN TRANSACTION ISOLATION LEVEL
    REPEATABLE READ;
SELECT name
FROM ships
WHERE firepower = 1500;
COMMIT;
```

Locking

Deadlocks

- Consider this interleaved schedule

T1	T2
W(A)	
	W(B)
W(B)	
	W(A)
	Commit
Commit	

- A strict 2PL schedule is the following:

T1	T2
X(A)	
W(A)	
	X(B)
	W(B)
WaitLock(B)	
	WaitLock(A)
Oh noes! A deadlock!	

- Both transaction are stuck waiting for the other lock to be released.
- This phenomenon is called *deadlock*.

Locking

Deadlock prevention

- Build a graph where each node is mapped to a transaction
- An arrow starts from T1 and points to T2 if T1 is waiting for a lock to be released by T2.
- If we have a cycle in such graph then there is a deadlock.
- The DBMS scheduler aborts transactions involved in a deadlock to free their locks.

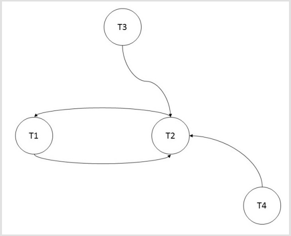

Figure: Wait graph with a deadlock.

Tasks

Task 1

- List all the possible conflicts arising in the transaction schedule below.
- Convert the schedule into an equivalent strict 2PL schedule.

T1	T2
R(A)	
R(C)	
W(C)	
	R(B)
	W(B)
	R(C)
	W(C)
Commit	
	Commit

Tasks

Solution

- The only conflict is a dirty read on C in T2.

T1	T2
S(A)	
R(A)	
S(C)	
R(C)	
X(C)	
W(C)	
	S(B)
	R(B)
	X(B)
	W(B)
	WaitLock(C)
Commit	
Unlock(A)	
Unlock(C)	
	S(C)
	R(C)
	X(C)
	W(C)
	Commit
	Unlock(B)
	Unlock(C)

Tasks

Task 2

- List all the possible conflicts arising in the transaction schedule below.
- Convert the schedule into an equivalent strict 2PL schedule.

T1	T2	T3
W(A)		
	R(A)	
	W(A)	
	W(B)	
	Commit	
		W(A)
		R(B)
		W(C)
W(B)		
Commit		
		R(C)
		W(A)
		Commit

Tasks

Solution

- There is a dirty read conflict on A between T1 and T2.
- There is a lost updated conflict on A between T1 and T3.
- There is an unrepeatable read conflict on B between T3 and T1.
- Note that there is NO dirty read conflict on B between T2 and T3 because T2 commits before T3 reads B.

Tasks

T1	T2	T3
X(A)		
W(A)		
	WaitLock(A)	
		WaitLock(A)
X(B)		
W(B)		
Commit		
Unlock(A)		
Unlock(B)		
	S(A)	
	R(A)	
	X(A)	
	W(A)	
	X(B)	
	W(B)	
	Commit	
	Unlock(A)	
	Unlock(B)	
		X(A)
		W(A)
		S(B)
		R(B)
		X(C)
		W(C)
		R(C)
		W(A)
		Commit
		Unlock(A)
		Unlock(B)
		Unlock(C)