Pre-lecture video

http://www.youtube.com/watch?v=G3xH2SoMOF0
Lecture 11: Transactions

Rasmus Pagh

Literature: KBL Appendix A.1
Today’s lecture

• Transactions: Motivation
• Conflicts and serializability
• Locking
• Isolation levels in SQL
Transaction

A transaction is a sequence of operations on a database that *belong together*.

**Example:**
Two persons with a shared bank account try to withdraw 100 kr at the same time.

**Transaction:**
1) read balance and store in variable B
2) if B$\geq$100 then B:=B-100
3) write B to balance
Another example, in SQL

Consider the following three transactions on the relation accounts(no,balance,type):

**Transaction A**

UPDATE accounts SET balance=balance*1.02 WHERE type='savings';  
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance<0;  
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance>0;

**Transaction B**

UPDATE accounts SET type='salary' WHERE no=12345;

**Transaction C**

UPDATE accounts SET balance=balance-1000 WHERE no=12345;
Autocommit

• Default in many DBMS interfaces is to make each SQL statement a transaction.
  - "Autocommit".

• In MySQL this is turned off by:
  - "Maximizing query area" in Query Browser
  - Typing `set autocommit=0;` (Q. Browser)
  - Calling `conn.setAutoCommit(false)` in JDBC.
Rollback

• Sometimes it is useful to be able to regret making the changes carried out by a transaction
  – Example: Booking several resources on the same day, but find one is not available.

• Commit means “no regret” anymore.

• DBMS needs rollback internally – might as well make it available for users.
ACID Properties

**Atomicity**: Transaction runs to completion or has no effect at all

**Consistency**: After a transaction completes, the integrity constraints are satisfied

**Isolation**: Transactions executed in parallel have the same effect as if they were executed sequentially

**Durability**: The effect of a committed transaction remains in the database even if the computer crashes.
Durability in a nutshell

• There exist disk systems that are highly reliable (e.g. still functions if one or two disks fail).
  – Trade-off: Redundancy vs reliability

• A database transaction is only really committed when the actions made by the transaction have all been written to the log on disk.
  – In case of crash, the log is used to reverse the state to the one implied by committed transactions.

• More info in KBL Appendix A.2.
Today: Atomicity and isolation

• This lecture is mainly concerned with atomicity and isolation.
• Consistency is a consequence of atomicity and isolation + maintaining any declared DB constraint (not discussed in this course).
Isolation and serializability

• Want transactions to satisfy serializability:
  – The state of the database should always look as if the committed transactions in a serial schedule.

• The scheduler of the DBMS is allowed to choose the order of transactions:
  – It is not necessarily the transaction that is started first, which is first in the serial schedule. (MySQL gives priority to writes.)
  – The order may even look different from the viewpoint of different users.
  – Demo in MySQL...
A simple scheduler

- A simple scheduler would maintain a queue of transactions, and carry them out in order.

- Problems:
  - Transactions have to wait for each other, even if unrelated (e.g. requesting data on different disks).
  - Smaller throughput. (Why?)
  - Some transactions may take very long, e.g. when external input or remote data is needed during the transaction.
Interleaving schedulers

• Most DBMSs have schedulers that allow the actions of transactions to interleave.
• However, the result should be as if some serial schedule was used.
• Such schedules are called serializable.
• In practice schedulers do not recognize all serializable schedules, but allow just some.

Next: Conflict serializable schedules.
Simple view on transactions

• We regard a transaction as a sequence of reads and writes of DB elements, that may interleave with sequences of other transactions.
• DB elements could be e.g. a value in a tuple, an entire tuple, or a disk block.
• \( r_T(X) \), shorthand for “transaction T reads database element X”.
• \( w_T(X) \), shorthand for “transaction T writes database element X”.
Conflicts

• The order of some operations is of no importance for the final result of executing the transactions.

• **Example:** We may interchange the order of any two read operations without changing the behaviour of the transactions doing the reads.

• In other cases, changing the order may give a different result - there is a conflict.
What operations conflict?

- It can easily be seen that two operations can conflict only if:
  - They involve the same DB element, and
  - At least one of them is a write operation.
- Note that this is a conservative, but safe, rule.

\[ r_{T1}(A) \rightarrow w_{T1}(A) \rightarrow r_{T2}(A) \rightarrow r_{T2}(A) \]

\( r_{T1}(A) \)

\( r_{T2}(A) \)

\( w_{T1}(A) \)

\( r_{T2}(A) \)

\( \text{time} \)
Conflict serializability

• Suppose we have a schedule for the operations of several transactions.
• We obtain conflict-equivalent schedules by swapping adjacent operations that do not conflict (any number of times).
• If a schedule is conflict-equivalent to a serial schedule, it is serializable.
• (The converse is not true.)
Testing conflict serializability

- Suppose we have a schedule for the operations of current transactions.
- If there is a conflict between operations of two transactions, T1 and T2, we know which of them must be first in a conflict-equivalent serial schedule.
- This can be represented as a directed edge in a graph with transactions as vertices.
Problem session

1 Concurrency control (15%)

a) Determine which of the following schedules are conflict-serializable:

1. \( r_2(A); w_1(A); w_2(A); r_1(A) \)

2. \( w_1(A); w_2(B); r_1(A); r_3(B); w_3(A); r_1(B) \)

Justify your answer by drawing the corresponding precedence graphs, with indication of the actions behind each arc.
Enforcing serializability

• Knowing how to recognize conflict-serializability is not enough.
• We will now study a mechanism that enforces serializability: Locking.

• Other methods exist: Time stamping / optimistic concurrency control.
  – Out of scope for this course.
Locks

• In its simplest form, a lock is a right to perform operations on a database element.

• Only one transaction may hold a lock on an element at any time.

• Locks must be requested by transactions and granted by the locking scheduler.
Two-phase locking

• Commercial DBMSs widely use two-phase locking, satisfying the condition:
  – In a transaction, all requests for locks precede all unlock requests.

• If two-phase locking is used, the schedule is conflict-equivalent to a serial schedule in which transactions are ordered according to the time of their first unlock request. (Why?)
Strict two-phase locking

• In strict 2PL all locks are released when the transaction completes.

• This is commonly implemented in commercial systems, since:
  – it makes transaction rollback easier to implement, and
  – avoids so-called cascading aborts (this happens if another transaction reads a value by a transaction that is later rolled back)
Lock modes

• The simple locking scheme we saw is too restrictive, e.g., it does not allow different transactions to read the same DB element concurrently.
• **Idea**: Have several kinds of locks, depending on what you want to do. Several locks on the same DB element may be ok (e.g. two read locks).
Shared and exclusive locks

- Locks for reading can be shared (S).
- Locks needed for writing must be exclusive (X).
- Compatibility matrix says which locks are granted:

<table>
<thead>
<tr>
<th>Lock held</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>Yes</td>
</tr>
<tr>
<td>X</td>
<td>No</td>
</tr>
</tbody>
</table>
Locks via B-trees

- If it is known that tuples in a relation are only accessed through a B-tree structure, an efficient way of locking many tuples (e.g. in a range) is to lock the corresponding B-tree nodes.
- This is known as **index locking**.
- MySQL manual: “A locking read, an UPDATE, or a DELETE generally set record locks on every index record that is scanned in the processing of the SQL statement.”
Phantom tuples

- Suppose we lock tuples where A=42 in a relation, and subsequently another tuple with A=42 is inserted.
- For some transactions this may result in unserializable behaviour, i.e., it will be clear that the tuple was inserted during the course of a transaction.
- Such tuples are called phantoms.
Avoiding phantoms

- Phantoms can be avoided by putting an exclusive lock on a relation before adding tuples. (However, this gives poor concurrency.)
- Index locking can be used to prevent other transactions from inserting phantom tuples, but allow most non-phantom insertions.
- In SQL, the programmer may choose to either allow phantoms in a transaction or insist they should not occur.
SQL isolation levels

- A transaction in SQL may be chosen to have one of four isolation levels:
  - **READ UNCOMMITTED**: Dirty reads are possible.
  - **READ COMMITTED**: Dirty reads are not permitted (but nonrepeatable reads and phantoms are possible).
  - **REPEATABLE READ**: Nonrepeatable and dirty reads are not permitted (but phantoms are possible).
  - **SERIALIZABLE**: Transaction execution must be serializable (above anomalies not allowed).
SQL isolation levels

• Possible implementations:
  – READ UNCOMMITTED:
    “No locks are obtained.”
  – READ COMMITTED:
    “Read locks are immediately released - read values may change during the transaction.” (But should ensure “statement level read consistency”.)
  – REPEATABLE READ:
    “2PL but no lock when adding new tuples.”
  – SERIALIZABLE:
    “2PL with lock when adding new tuples.”
Consider the following three transactions on the relation accounts\((no, balance, type)\):

**Transaction A**

- UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
- UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance<0;
- UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance>0;

**Transaction B**

- UPDATE accounts SET type='salary' WHERE no=12345;

**Transaction C**

- UPDATE accounts SET balance=balance-1000 WHERE no=12345;

The purpose of Transaction A is to add interest to the balance of accounts, depending on the type and balance. Transaction B changes the type of a particular account. Transaction C makes a withdrawal from a particular account.

\[ a \] Suppose that the transactions are run more or less simultaneously at isolation level \textsc{read commited}. What results of running the transactions are possible in this case, but not if the transactions had been run at isolation level \textsc{serializable}?
Isolation level syntax

• Begin transaction with:

```sql
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED |
  READ COMMITTED |
  REPEATABLE READ |
  SERIALIZABLE }```

ACID testing MySQL

- Most storage engines are made with only very simple concurrency control in mind.
- InnoDB supports the standard SQL isolation concurrency control features, and more.
- Beware: Using SQL isolation levels with another storage engine will probably just have no effect.
Be careful with SERIALIZABLE

• Some common implementations of “SERIALIZABLE” allows, e.g., the following:
  – Suppose we have a relation R with a tuple for each reserved seat in a plane.
  – Transactions A and B simultaneously read R and find that seat 13A is free.
  – Transaction A and B both insert a tuple indicating that seat 13A has been booked.

• Such conflicts can be stopped by a lock or by a database constraint.
Explicit row locking

• Many DBMSs allow transactions to explicitly lock a set of tuples.

• Example:
  ```sql
  SELECT * FROM seats
  WHERE seat = '13A'
  FOR UPDATE;
  ```

• Can be used to control a resource, e.g. the right to insert a reservation tuple for seat 13A in another table.
Snapshot isolation

• Some DBMSs implement **snapshot isolation**, an isolation level that gives a stronger guarantee than **READ COMMITTED**.

• Each transaction T executes against the version of the data items that was committed “when the T started”.
  – Oracle, option in MySQL, SQL server,…

• Possible implementation:
  – No locks for read, locks for writes.
  – Store old versions of data (costs space).
Granularity of locks

• So far we did not discuss what DB elements to lock: Atomic values, tuples, blocks, relations?
• What are the advantages and disadvantages of fine-grained locks (such as locks on tuples) and coarse-grained locks (such as locks on relations)?
• The following advice can be found in the book by Shasha and Bonnet: "Long transactions should use table locks, short transactions should use record locks".
Granularity of locks

- Fine-grained locks allow a lot of concurrency, but may cause problems with *deadlocks*.
- Coarse-grained locks require fewer resources by the locking scheduler.
- We want to allow fine-grained locks, but use (or switch to) coarser locks when needed.
- Some DBMSs switch automatically - this is called lock escalation. The downside is that this easily leads to *deadlocks*. 
Granularity in MySQL

- InnoDB uses row-level locking by default.
  - No lock escalation.
- Table locking can be done manually:
  - LOCK TABLES T1 READ, T2 WRITE,...
  - UNLOCK TABLES
Locks and deadlocks

- The DBMS sometimes must make a transaction wait for another transaction to release a lock.
- This can lead to deadlock if e.g. A waits for B, and B waits for A.
- In general, we have a deadlock exactly when there is a cycle in the waits-for graph.
- Deadlocks are resolved by aborting (rolling back) some transaction involved in the cycle.
Simple deadlock prevention

• 2001 MySQL manual:

All locking in MySQL is deadlock-free.
This is managed by always requesting all needed locks at once at the beginning of a query and always locking the tables in the same order.

• 2008 MySQL manual:

All locking in MySQL is deadlock-free, except for InnoDB and BDB type tables.

– Explanation? Why use InnoDB and BDB?

• Problem session: Why does “always locking tables in the same order” never lead to deadlock?
Summary

• Concurrency control mechanisms give various trade-offs between isolation and performance.
  – Safe choice is SERIALIZABLE (well...)
  – Sometimes lower SQL isolation levels suffice – difficult to analyze in general
  – Manual efforts may sometimes be better: Table locking, explicit row locking,...
  – Deadlocks happen. A simple cure is lock acquisition in fixed order.
Course goals

After the course the students should be able to:

- identify possible problems in transaction handling, related to consistency, atomicity, and isolation.
- apply a simple technique for avoiding deadlocks