Reliable storage; Concurrency control

Rasmus Pagh

Today's lecture

- Reliable storage (1 slide)
- Conflicts and serializability
- Locking
- Isolation levels in SQL
- *A music video!*
- Optimistic concurrency control
- Transaction tuning
- Transaction chopping

Generic motivatiom

- Undesired behaviour can occur if several transactions run in parallel such that the actions interleave. Consider e.g.:
  - Transferring money and adding interest in a bank's database.
  - Booking plane tickets.
  - Inconsistency in patient records.

SQL example

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;
```

ACID properties of transactions

- **A**tomicity: A transaction is executed either completely or not at all.
- **C**onsistency: Database constraints must be kept.
- **I**solation: Transactions must appear to execute one by one in isolation.
- **D**urability: The effect of a committed transaction must never be lost.

Durability in a nutshell

- There exist disk systems (RAID) 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.
  - In high-update situations, many transactions need to commit in 1 log I/O.

Today: Atomicity and isolation

- This lecture is mainly concerned with atomicity and isolation.
- More on durability can be found in RG 18 and SB 2.3.
- Consistency is a consequence of atomicity and isolation + maintaining any declared DB constraint (not discussed in this course).

Isolation and serializability

- We would like the DBMS to make transactions satisfy serializability:
  - The state of the database should always look as if the committed transactions were performed one by one in isolation (i.e., 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.
  - The order may even look different from the viewpoint of different users.

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

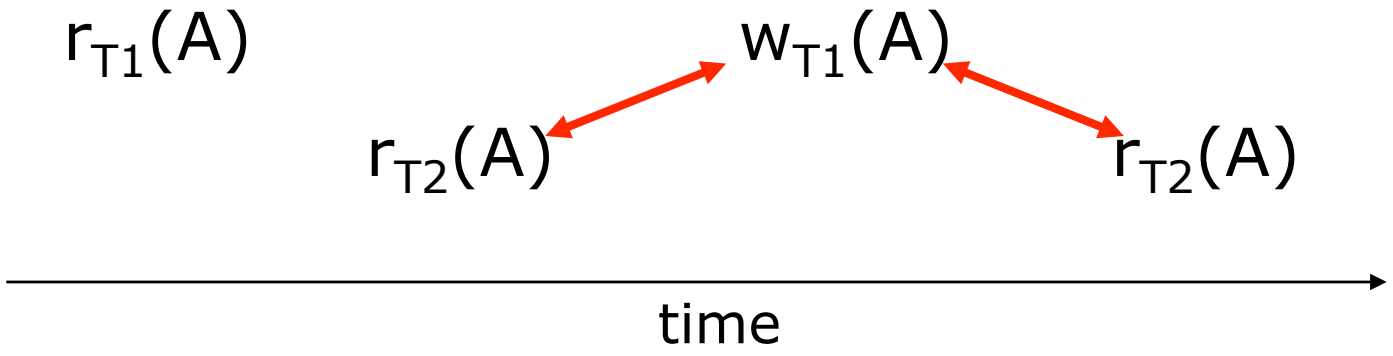
- We regard a transaction as a sequence of reads and writes of DB elements, that may interleave with sequences of other transactions.
- DB elements could be e.g. a value in a tuple, an entire tuple, or a disk block.
- $r_T(X)$, shorthand for ”transaction T reads database element X”.
- $w_T(X)$, shorthand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



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 (ex. in RG).

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.

- More generally, find a criterion on the precedence graph that is equivalent to conflict-serializability:
  - If the graph meets the criterion the schedule is conflict-serializable, and
  - If the schedule is conflict-serializable, the graph meets the criterion.

Enforcing serializability

- Knowing how to recognize conflict-serializability is not enough.
- We will now study mechanisms that **enforce** serializability:
  - Locking (pessimistic concurrency control).
  - Time stamping (optimistic concurrency control).

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:

	Lock requested		
Lock held		S	X
	S	Yes	No
	X	No	No

Update locks

- A transaction that will eventually write to a DB element, may allow other DB elements to keep read locks until it is ready to write.
- This can be achieved by a special update lock (U) which can be upgraded to an exclusive lock.

Compatibility for update locks

		Lock requested		
		S	X	U
Lock held	S	Yes	No	Yes
	X	No	No	No
	U	No (!)	No	No

- **Question**: Why not allow new shared locks when an upgrade lock is held?

The locking scheduler

- The locking scheduler is responsible for granting locks to transactions.
- It keeps lists of all currents locks, and of all current lock requests.
- Locks are not necessarily granted in sequence, e.g., a request may have to wait until another transaction releases its lock.
- Efficient implementation not discussed in this lecture.

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 SB: *"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.

Locks on indexes

- When updating a relation, any indexes on the table also need to be updated.
- Again, we must use locks to ensure serializability.
- B-trees have a particular challenge: The root may be changed by any update, so it seems concurrent updates can’t be done using locks.

Locks in B-trees

- **First idea**:
  - Put an exclusive lock on a B-tree node if there is a **chance** it may have to change.
  - Lock from top to bottom along the path.
  - Unlock from bottom to top as soon as nodes have been updated.
- **Refinement:**
  - First figure out what nodes need to be updates.
  - Lock these nodes top-down (why not any order?).

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**.

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`:
    ”No locks are obtained.”
  - `READ COMMITTED`:
    ”Read locks are immediately released - read values may change during the transaction.”
  - `REPEATABLE READ`:
    ”2PL but no lock when adding new tuples.”
  - `SERIALIZABLE`:
    ”2PL with lock when adding new tuples.”

SQL isolation levels

Isolation Level	Dirty Read	Unrepeatable read	Phantoms
Read uncommitted	Maybe	Maybe	Maybe
Read committed	No	Maybe	Maybe
Repeatable read	No	No	Maybe
Serializable	No	No	No

SQL implementation

- Note that implementation is not part of the SQL standards - they specify only semantics.
- A DBMS designer may choose another implementation than the mentioned locking schemes, as long as the semantics conforms to the standard.

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:

```
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.

Cursor stability

- Most DBMSs extend the guarantee of `READ COMMITTED` with **cursor stability**:
  - "Read locks are held for the duration of a single SQL statement."
  - Still, values seen may change from one statement to the next.

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".
- Possible implementation:
  - No locks for read, locks for writes.
  - Store old versions of data (costs space).

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 some transaction involved in the cycle.

Dealing with deadlocks

- Possibilities:
  - Examine the waits-for graph periodically to find any deadlock.
  - If a transaction lived for too long it may be involved in a deadlock - roll back.
  - Use timestamps, unique numbers associated with every transaction to prevent deadlocks.
- Deadlocks are less likely if we lock entire relations - but this decreases throughput.

Avoiding deadlocks by timestamp

- Two possible policies when T waits for U:
  - **Wait-die**.
    If T is youngest it is rolled back.
  - **Wound-wait**.
    If U is youngest it is rolled back.

- In both cases there can be no waits-for cycles, because transactions only wait for younger (resp. older) transactions.
- Why always roll back the youngest?

Summary of locking

- Locking can prevent transactions from engaging in non-serializable behaviour.
- However, it is sometimes a bit too strict and pessimistic, not allowing as much concurrency as we would like.

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

- Next we will consider optimistic concurrency control that works well when transactions don't conflict much.

Optimistic concurrency control

- **Basic idea:** Let transactions go ahead, and only at the end make sure that the behavior is serializable.
- Used in prototypes for very fast OLTP systems (H-store), with fallback on locking in high-update situations.
- Several ways of realizing:
  - Validation
  - Optimistic 2PL (private workspace)
  - Timestamping (next)

Timestamps

- **Idea**: Associate a unique number, the *timestamp*, with each transaction.
- Transactions should behave as if they were executed in order of their timestamps.
- For each database element, record:
  - The highest timestamp of a transaction that read it.
  - The highest timestamp of a transaction that wrote it.
  - Whether the writing transaction has committed.

Timestamp based scheduling

- If transaction T requests to:
  - Read a DB element written by an uncommitted transaction, it must wait for it to commit or abort.
  - Read a DB element with a higher write timestamp, T must be rolled back.
  - Write a DB element with a higher read timestamp, T must be rolled back.

- Rolling back means undoing all actions, getting a new timestamp, and restarting.

Question

- What should the scheduler do if a transaction requests to write a DB element with a higher write timestamp?

Multiversion timestamps

- In principle, all previous versions of DB elements could be saved.
- This would allow any read operation to be consistent with the timestamp of the transaction.
- Used in many systems for scheduling read only transactions. (In practice, only recent versions are saved.)

Optimism vs pessimism

- Pessimistic concurrency is best in high-conflict situations:
  - Smallest number of aborts.
  - No wasted processing (if no deadlocks).
- Optimistic concurrency control is best if conflicts are rare, e.g., if there are many read-only transactions.
  - Highest level of concurrency.
  - Time stamp methods used in some **distributed** databases.

Lock tuning

- SB offers this advice on locking:
  - Use special facilities for long reads.
  - Eliminate locking when unnecessary.
  - Use the weakest isolation guarantee the application allows.
  - Change the database schema only during ”quiet periods” (catalog bottleneck).
  - Think about partitioning (also hash!).
  - Select the appropriate granularity of locking. (Next: How-to in Oracle)
  - If possible, chop transactions into smaller pieces. (Later)

Locking in Oracle

- ”Oracle Database always performs necessary locking to ensure data concurrency, integrity, and statement-level read consistency. You can override these default locking mechanisms. For example, you might want to override the default locking of Oracle Database if:
  - You want transaction-level read consistency or "repeatable reads" - where transactions query a consistent set of data for the duration of the transaction, knowing that the data has not been changed by any other transactions. This level of consistency can be achieved by using explicit locking, read-only transactions, serializable transactions, or overriding default locking for the system.
  - A transaction requires exclusive access to a resource. To proceed with its statements, the transaction with exclusive access to a resource does not have to wait for other transactions to complete.”

Locking in Oracle, cont.

- Locking a table T:

```
LOCK TABLE T IN EXCLUSIVE MODE;
```

- Unlocking all locked tables:

```
commit;
```

- Other lock modes are also available.

Transaction tuning example

(slide (c) Shasha and Bonnet, 2001)

Simple purchases:

- Purchase item I for price X
  - 1. If cash < X then roll back transaction (constraint)
  - 2. Inventory(I) := inventory(I)+X
  - 3. Cash := Cash – X
- Two purchase transactions P1 and P2
  - P1 purchases item I for price 50
  - P2 purchases item I for price 75
  - Cash is 100

Example: Simple Purchases

(slide (c) Shasha and Bonnet, 2001)

- If steps 1-2-3 is one transaction then one of P1, P2 rolls back.
- If 1, 2, 3 are three distinct transactions, this may happen:
  - P1 checks that cash > 50. It is.
  - P2 checks that cash > 75. It is.
  - P1 completes. Cash = 50.
  - P2 completes. Cash = - 25.

Example: Simple Purchases

(slide (c) Shasha and Bonnet, 2001)

- Orthodox solution
  - Make whole program a single transaction
    - Cash becomes a bottleneck!
- Chopping solution
  - Find a way to rearrange and then chop up the transactions without violating serializable isolation level.

Example: Simple Purchases

(slide (c) Shasha and Bonnet, 2001)

- Chopping solution:
  - 1. If Cash < X then roll back.
    Cash := Cash – X.
  - 2. Inventory(I) := inventory(I) + X
- Chopping execution:
  - P11: 100 > 50. Cash := 50.
  - P21: 75 > 50. Rollback.
  - P12: inventory := inventory + 50.

Transaction Chopping

(slide (c) Shasha and Bonnet, 2001)

- Execution rules:
  - When pieces execute, they follow the partial order defined by the transactions.
  - If a piece is aborted because of a conflict, it will be resubmitted until it commits
  - If a piece is aborted because of an `abort` statement, no other pieces for that transaction will execute.
- A chopping is *rollback-safe* if
  - (a) There are no abort commands, or
  - (b) if the abort commands are in the first piece of the chopping.

Summary

- Concurrency control is crucial in a DBMS.
- Handling CC problems requires an understanding of the involved mechanisms.
- Mostly, you want the schedule of operations of transactions to be serializable.
- The scheduler may use locks or timestamps.
  - Locks allow less concurrency, and may cause deadlocks.
  - Timestamps may cause more aborts.

Exercises

Transaction case study

(Tangled transactions)