Anna Östlin Pagh and Rasmus Pagh
IT University of Copenhagen

Advanced Database Technology

March 14, 2005

CONCURRENCY CONTROL

Lecture based on [GUW, 18.0-8, 19.3]

Today: Concurrency control

- Serial schedules and consistency.
- Conflict-serializability.
- Locking schemes:
  - 2-phase locking.
  - Lock modes and compatibility matrices.
- Optimistic concurrency control.
  - Timestamping.
  - Multi-version timestamps.

Problem session

- Come up with examples of undesired behaviour that could 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.

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.

Today: Atomicity and isolation

- Later lecture deals with durability.
- This lecture is 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

- 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).
  - Some transactions may take very long, e.g. when external input is needed during the transaction.

Interleaving schedulers

- Good 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. Today: **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 transaction(s) doing the reads.
- In other cases, changing the order **may** give a different result - there is a **conflict**.

What operations conflict?

- By considering all cases, it can be seen that two operations conflict if and only if:
  1) They involve the same DB element, and
  2) 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 can obtain **conflict-equivalent** schedules by swapping adjacent operations that do not conflict.
- If a schedule is conflict-equivalent to a serial schedule, it is serializable.
- The converse is not true (ex. in GUW).

Testing conflict serializability

- Suppose we have a schedule for the operations of current transactions.
- If there is a conflict between operations of two transactions, T_1 and T_2, 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

- Find a criterion on the precedence graph which is equivalent to conflict-serializability, i.e.:
  - 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 DB systems 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.

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

- **Question**: Why not upgrade shared locks?

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 in GUW.

Problem session

- 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)?

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.

Warning locks

- An exclusive lock on a tuple should prevent an exclusive lock on the whole relation.
- To implement this, all locks must be accompanied by **warning locks** at higher levels of the hierarchy of database elements.
- We denote shared/exclusive warning locks by IS/IX, where I stands for the *intention* to shared/exclusively lock database elements at a lower level.

Compatibility of warning locks

		Lock requested			
Lock held		IS	IX	S	X
	IS	Yes	Yes	Yes	No
	IX	Yes	Yes	No	No
	S	Yes	No	Yes	No
	X	No	No	No	No

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.
- In SQL, the programmer may choose to either allow phantoms in a transaction or insist they should not occur.

SQL isolation levels

A transactions 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 implementation

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

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

- **Idea:**
  - Only 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.
- More general scheme, the **tree protocol**, discussed in GUW.

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 after every lock request 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.

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.

Problems with locking

- Locking prevents 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.
- Next we will consider an **optimistic** concurrency control that works better when transactions don’t conflict much.

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, it must be rolled back.
- Write a DB element with a lower read timestamp, it must be rolled back.

Rolling back means getting a **new** timestamp.

Problem session

- 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.
- Optimistic concurrency control is best if conflicts are rare, e.g., if there are many read-only transactions.
  - Highest level of concurrency.
- Hybrid solutions often used in practice.

Why this is useful knowledge

- Knowledge on concurrency control of a DBMS is useful:
  - If you encounter concurrency problems, e.g., many deadlocks or long response times.
  - When choosing a suitable DBMS for a particular application (optimistic or pessimistic concurrency control?).

Summary

- Concurrency control is crucial in a DBMS.
- The schedule of operations of transactions should be **serializable**.
- The scheduler may use locks or timestamps.
  - Locks allow less concurrency, and may cause deadlocks.
  - Timestamps may cause more aborts.