DATABASE TECHNOLOGY - 1MB025

(also 1DL029, 1DL300+1DL400)

Spring 2008

An introductory course on database systems

http://user.it.uu.se/~udbl/dbt-vt2008/
alt. http://www.it.uu.se/edu/course/homepage/dbastekn/vt08/

Kjell Orsborn
Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden

Introduction to Transactions & Concurrency Control

Elmasri/Navathe ch 17 and 18
Padron-McCarthy/Risch ch 23 and 24

Kjell Orsborn

Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden

The transaction concept

- We have earlier assumed that only one program (or DML query) at a time accesses and performs operations on a database (i.e. we have assumed serial access).
- In general several programs work on the same database.
  - This results in that simultaneous access and updates must be controlled by means of transactions management (e.g. seat booking, ATM systems)
- In a DBMS context, a **transaction** is an atomic and logic *unit* of database processing that accesses and possibly updates various data items.
  - A simple query in the DML of the DBMS.
  - A program written in the host language with one or several calls to DML. If several users execute the same program every execution constitute a transaction in their own.

Transaction concept cont'd . . .

- A transaction must see a consistent state
- During transaction execution the database may be inconsistent
- When a transaction is committed, the database must be consistent
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Transaction management

- Transaction management aims at handling transactions as indivisible sets of operations; that is a transaction is either performed as a whole or not at all.
- In reality, a transaction consist of a sequence of more elementary steps (operations) such as read and write of database items.
- At the same time as we want to admit time sharing for these elementary operations, we want to keep the requirement of indivisibility.

Example of a transaction

Ex. two transactions:

T1:	T2:
Read X	Read X
X:=X-N	X:=X+M
Write X	Write X
Read Y	
Y:=Y+N	
Write Y	

Without time sharing (serial)

T1:	T2:
Read X	
X:=X-N	
Write X	
Read Y	
Y:=Y+N	
Write Y	
	Read X
	X:=X+M
	Write X

Problem 1: simultaneous transactions
(with time sharing - scheduled)

- Problem with lost updates

T1:	T2:
Read X	
X:=X-N	
	Read X
	X:=X+M
Write X	
Read Y	
	Write X
Y:=Y+N	
Write Y	

The last operation in T2 writes a wrong value in the database.

Problem 2: simultaneous transactions

(with time sharing - scheduled)

- Problem with temporary updates

T1:	T2:
Read X	
X:=X-N	
Write X	
	Read X
	X:=X+M
	Write X
Read Y	
Y:=Y+N	
Write Y	

T1 failured before it was finished. The system must eliminate (“undo”) the effects of T1. However, T2 has already read the wrong value for X and will also write that wrong value in the database.

Problem 3: simultaneous transactions

(with time sharing - scheduled)

- Problem with incorrect summation

T1:	T2:
	Sum:=0
	Read A
	Sum:=Sum+A
Read X	...
X:=X-N	...
Write X	...
	Read X
	Sum:=Sum+X
	Read Y
	Sum:=Sum+Y
Read Y	
Y:=Y+N	
Write Y	

T2 performs an aggregation operation while T1 modifies some of the relevant items.

Do we get the correct sum in Sum?

ACID properties

- To preserve the integrity of data, the DBMS must ensure:
  - **Atomicity** (atomic or indivisible): a logic processing unit (all operations of the transaction) is carried out in its whole or not at all.
  - **Consistency** (preservation): a correct execution of a transaction in isolation should preserve the consistency of the database (from one consistent state to another).
  - **Isolation**: Although multiple transactions may execute concurrently, each transaction must be unaware of of other concurrently executing transactions. The updates of a transaction shall be isolated from other transactions until after the commit point.
  - **Durability** (or permanency): If a transaction completes successfully, the changes it has made to the database must persist and should not be lost in a later system failure.

Example of fund transfer

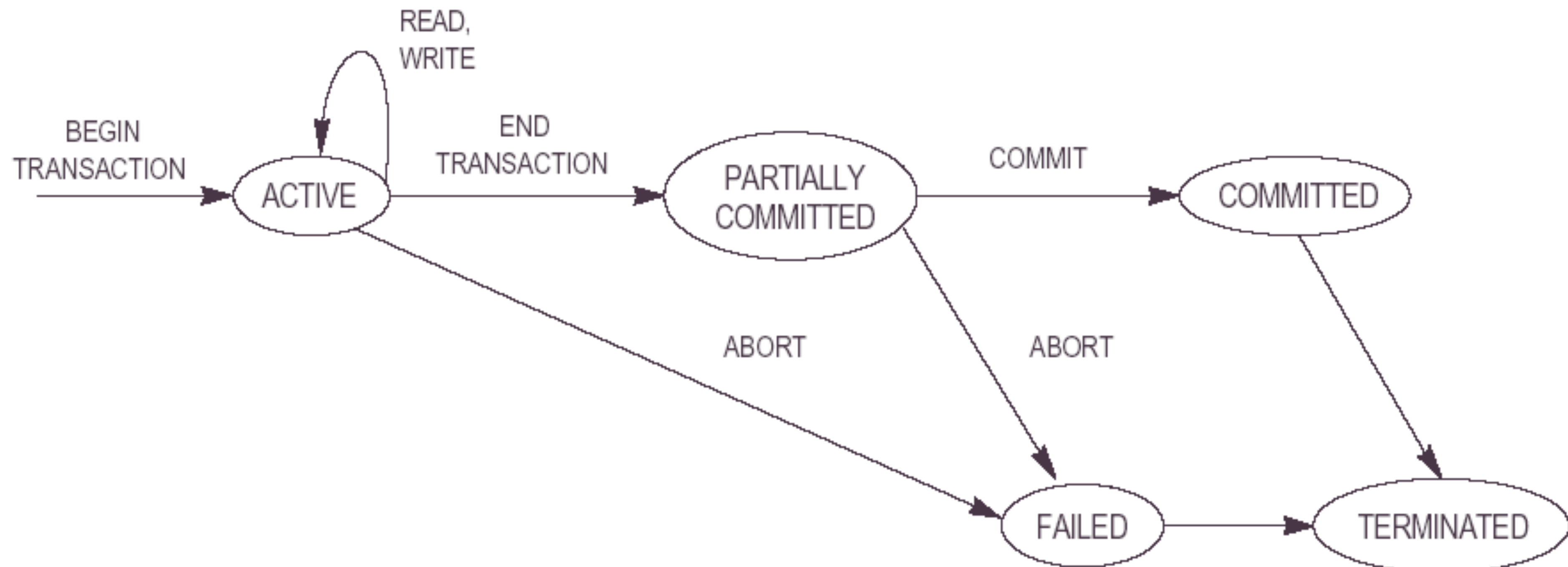
- Transaction to transfer $50 from account A to account B :
  1. read(A)
  2. A := A - 50
  3. write(A)
  4. read(B)
  5. B := B + 50
  6. write(B)
- Consistency requirement — the sum of A and B is unchanged by the execution of the transaction.
- Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

Example of fund transfer cont'd

- Durability requirement — once the user has been notified that the transaction has completed (ie. the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum A +B will be less than it should be).

These requirement can be ensured trivially by running transactions serially, that is, one after the other. However, we would like to accomplish the same benefits for multiple transactions executing concurrently.

Transaction state cont'd



Transaction state

- **Active**, the initial state; the transaction stays in this state while it is executing
- **Partially committed**, when transaction ends, after the final statement has been executed, it goes into the partially commited state.
- **Committed**, after *successful* completion.
- **Failed**, after the discovery that normal execution can no longer proceed or if it has been aborted in its active state. Rollback might be necessary.
- **Terminated**, corresponds to the transaction leaving the system.
  After the transaction has been rolled back and the database is restored to its state prior to the start of the transaction. A failed or aborted transaction can be restarted either automatically or manually.

Concurrent executions

- Multiple transactions are allowed to run concurrently in the system.
  Advantages are :
  - increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk
  - reduced average response time for transactions: short transactions need not wait behind long ones
- Concurrency control schemes – mechanisms to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Transaction schedule

- There exist a number of different execution orders that can be scheduled for the operations in a set of transactions.
  - But which of these execution orders are acceptable?
- We will assume that the intention, when transactions are implemented, is that they should be executed in serial.
- A **transaction schedule** for a set of transactions describes in what order the operations (Read Write etc.) in the transactions should be performed.
- OBS!: the relative order among single operations in a transaction is kept in the transaction schedule.

Serial and serializable transaction schedules

- A transaction schedule where the operations for each transaktion uninteruptedly follow each other is called a ***serial*** schedule.
- For example the transaction schedule S for transactions T1 and T2:

  T1: o11, o12,..., o1m

  T2: o21, o22,..., o2n

  S: o11, o12,..., o1m, o21, o22,..., o2n
- A transaction schedule for a number of transactions is said to be ***serializable*** if its effect is "equivalent" to the effect of a serial transaction schedule incorporating the same transactions.
- To be able to judge if a transaction schedule is correct we must prove that the schedule is serializable.

Controlling serializability of schedules

- Testing a schedule for serializability after it has executed is a little too late!
- Goal – to develop concurrency control protocols that will assure serializability. The idea is that instead of analyzing if a schedule is serializable, they will instead impose a protocol that avoids nonserializable schedules.
- There are algorithms that can control serializability such as two-phase locking protocols (see concurrency control).

Classes of recoverable schedules

- recoverable schedule
  - committed transactions never need to rolled back
- cascadeless schedule
  - recoverable
  - transactions read only items written by committed transactions
- strict schedule
  - recoverable
  - cascadeless
  - no read/write item X until the last transaction that wrote X has committed
  - simplifies the recovery process

Transaction definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - Commit work commits current transaction and begins a new one.
  - Rollback work causes current transaction to abort.
- Levels of consistency specified by SQL-92:
  - Serializable — default
  - Repeatable read
  - Read committed
  - Read uncommitted

Levels of consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of a record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.
- Lower degrees of consistency useful for gathering approximate information about the database, e.g. statistics for query optimizer.

Concurrency control

- Concurrency control handles the execution of concurrent transactions.
- There are two main techniques for concurrency control:
  - pessimistic concurrency control - locking of data items
  - optimistic concurrency control - shadow paging

Items and data granularity

- Units of data that are operated on by transactions are called **items**.
- The size, or **data granularity**, of such an item is determined by the database designer (and capabilities of the DBMS).
- The term item can mean different things:
  - a record (or tuple in a relation)
  - a data field value
  - a disc block
  - a whole file
  - or the whole database

Pessimistic techniques - Locking

- Locking is one of the main mechanisms to handle concurrent transactions (is based on a pessimistic assumption that conflicts will appear)
- A **lock** is the access right for an item and a program, the **lock manager**, decides which transaction that should be granted the access right for an item.
- The lock manager stores information about locked items in a table that consists of records of the form:
  - (<item>, <lock-type>, <transaction>)
  - A record (I,L,T) means that the transaction T has locked item I with a lock of type L.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Binary locks

- A **binary** lock only has two states: *locked/unlocked*.
- Transactions must comply with the following rules:
  - 1. The transaction T must perform Lock X before it performs any Read X or Write X operation.
  - 2. T must perform Unlock X after all Read X and Write X operations are finished.
  - 3. T shall not perform Lock X if T already has locked X.
  - 4. T should only perform Unlock X if T has locked X at the same moment.
- During the time between a Lock X and Unlock X in T, T locks the item X (or T controls item X.)
- Only <u>one</u> transaction is allowed to lock an item at a certain point of time.

Example

- Two transactions T1 and T2 that both perform:

  Read A;
  A:=A+1;
  Write A

- The solution is to introduce locks,

  Lock A;
  Read A;
  A:=A+1;
  Write A;
  Unlock A

A in DB	5	5	5	5	6	6
T1:	ReadA		A:=A+1			WriteA
T2:		ReadA		A:=A+1	WriteA	
A in T1:s working area	5	5	6	6	6	6
A in T2:s working area		5	5	6	6	

- T2 can not any longer reach A before T1 is finished to operate on A.

Other types of locks

- Binary locks are very restrictive. For that reason one has adapted locking systems that e.g. grant read access to several transactions simultaneously. However, write access is only granted to one transaction.
- The following conditions must be fulfilled:
  1. T must perform Readlock X or Writelock X before Read X.
  2. T must perform Writelock X before Write X.
  3. T must perform Unlock X after all Read X an Write X is finished.
  4. T shall not perform Readlock X if T already has locked X.
  5. T shall not perform Writelock X if T already has write access to X.
  6. T shall only perform Unlock X if T has locked X at the moment.

The two-phase locking protocol

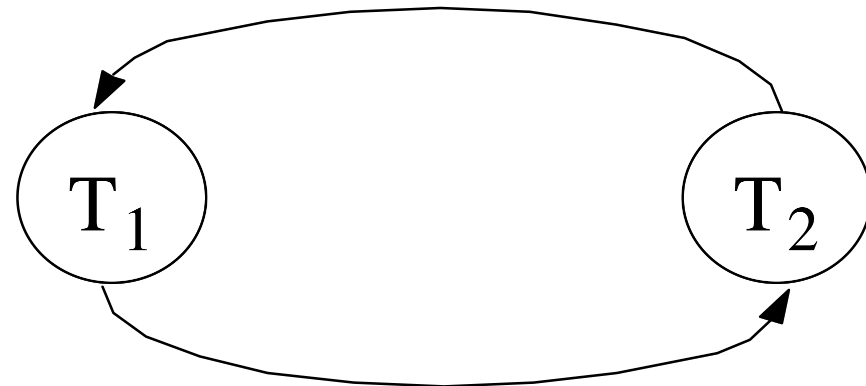
- This is a protocol which ensures conflict-serializable schedules.
  - Phase 1: Growing Phase
    - transaction may obtain locks
    - transaction may not release locks
  - Phase 2: Shrinking Phase
    - transaction may release locks
    - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions are serializable in the same order as they acquired its final lock.
- Two-phase locking *does not* ensure freedom from deadlocks

Two-phase locking protocols

- Two-phase locking protocol
  - expanding phase - shrinking phase
  - guaranteeing serilizable schedules
- Basic 2PL
  - All locking operations precedes the the first unlock operation
- Conservative (static) 2PL
  - Dead-lock free
  - Difficult in practice
- Strict 2PL
  - Guarantees strict schedules
  - Not dead-lock free
  - No release of write-locks until commit/abort
- Rigorous 2PL
  - Also guarantees strict schedules
  - No release of locks until commit/abort

Deadlock - a locking problem

- T1: Lock A; Lock B; ...;Unlock A; Unlock B
- T2: Lock B; Lock A; ...;Unlock B; Unlock A
- A **deadlock** is a situation where every member in S (a set of at least two transactions) waits for the privilige to lock an item that already has been locked by another transaction.
- We have a deadlock (or circular lock) if there is a cycle in the dependency graph that shows which transactions that wait for locks to be realeased.



Dependency graph

To solve a deadlock

Deadlock prevention protocols (impractical)

1. Every transaction should, before it starts, lock all items it needs at the same time. Partial locks result in a failured transaction.
2. Introduce an arbitrary linear order between the items and demand that the locking of these items should be performed according to this order.

   T1: Lock A; Lock B; ...

   T2: Lock A; Lock B; ...
3. Use time stamps to create priorities between transactions.

Deadlock detection and timeouts (practical)

4. Create a wait-for graph that keeps track of the transactions that other transactions are waiting for. Then check periodically if there is any circularity in the graph. Stop transactions that causes deadlock.
5. Timeouts - abort transactions that have been waiting for a defined threshold time.

Other pitfalls of lock-based protocols

- The potential for deadlock exists in most locking protocols - deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an write-lock on an item, while a sequence of other transactions request and are granted an read-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation using some form of prioritization.

Optimistic techniques - shadow paging

- Is based on an optimistic assumption that conflicts are seldom.
- Let transactions execute concurrently and control if they have interacted in a non-serializable manner when they are to be finished.
- If conflicts occur, one of the transactions can be aborted.
- The technique is combined with *shadow paging* where each transaction make updates on their own copy of data in a way that concurrent transactions do not “see” updates from another transaction.