Database Transaction Abstraction I

Modeling DB Systems
(Based on BHG 1987)
Model vs. Reality

• We make simplifying assumptions
  – We introduce ‘discreteness’ of events and actions
  – We impose ‘structure’ on a family of systems
  – We postulate ‘correct behavior’ for users
• We can then argue mathematically and prove algorithms correct
• In reality, some of our assumptions are usually relaxed
Abstract Database System Model - Transactions

• Transaction manager (TM): preprocessing
• Scheduler: relative order of operations
• Data manager (DM):
  – Recovery manager (RM)
  – Cache manager (CM)
• Variations:
  – centralized
  – multiprocessor
  – distributed
Centralized DBS

Transaction Manager

Scheduler

Recovery Manager

Cache Manager

Database

Data Manager

T1
T2

Tm
CM

- Cache manager (CM): controls transfers between volatile memory and stable storage.
- Uses Fetch(x) and Flush(x).
- May initiate actions on its own.
RM

• Ops: Start, Commit, Abort, Read, Write.
• Uses Fetch and Flush (CM).
• Recovers after System Failure = loss of volatile memory.
• Need to restore to consistent state, may need to control the CM.
• Recovers after Media Failure = loss of portions of stable storage.
• Uses redundancy on independent devices.
• DM interface = RM interface.
Scheduler

• Controls order of DM ops.
• Needs to ensure useful properties such as “strict” or “avoid cascading abort”.
• Uses execute, delay, reject ops.
• Uses limited information in decision making.
• All knowledge comes from the operations themselves.
• The TM does bookkeeping and makes some decisions (e.g., which copy to read in a distributed system).
How are operations processed by modules?

• A module may execute any of its unexecuted ops, independently of submission order.
• Op issuer is responsible to ensure order.
• Handshake: pass op and wait for acknowledgement.
• How about using queues?
  – too much unneeded ordering
  – If op needs op1 in module m1 and *afterwards* op2 in module m2:
    • putting op1 on Q1 and op2 on Q2 does not have the right effect.
    • having a single queue q12 does not have the right effect.
    • Need some kind of handshake, e.g., tagging op1 with op2...
Distributed System Architecture

- Sites connected via a network.
- Processes can exchange messages.
- Each site is a centralized DBS.
- For now, each data item is in a unique site.
- Transaction = one or more processes executing at one or more site.
- A transaction is controlled by a unique TM.
- TMs communicate with local or remote schedulers in order to process a transaction’s reads and writes.
Distributed DBS

Network

Centralized DBS

Centralized DBS

Centralized DBS
Transactions

• An execution of a program accessing shared data.

• Informally, a transaction should execute atomically, that is not interfering with other transactions and either:
  – terminate normally and have its effects made permanent, or
  – terminate abnormally and have no effect.

• We start formalizing these concepts.
Database System

- A database is a set of named distinct data items having values. Usually denoted by x,y,z ...
- DBS: system supporting ops such as Read[x] and Write[x,val].
- The DBS overall effect is as if ops execute atomically, in reality they may execute concurrently.
- For example, operations on two different data items done concurrently have the same effect as if done sequentially.
- Ops on transactions: Start (id is generated), Commit, Abort.
- A transaction may result from concurrently executing two or more programs.
  - In this case, a transaction may submit an operation to the DBS prior to the DBS having responded to the previous one.
- The last operation of a transaction is either commit or abort.
Commit and Abort

- Aborts may be generated by the transaction or the system; an abort belongs to the transaction.
- Commit: a guarantee by the system that the transaction will not be aborted and that its effects will be made permanent.
Model assumptions

• Interactions between different transactions (e.g., messages) must be controlled by the DBS.
• Transactions may communicate using output (of one transaction to terminal) and input (based on this to another transaction).
• User should not trust terminal output until commit is processed successfully.
• System may defer outputs until commit.
• In general, if a transaction is aborted and resubmitted its old terminal input is no longer valid.
Recoverability

• Abort: effects need be eliminated:
  – effects on data items: need to restore a value $x$ would have if $T$ never ran.
  – effects on other transactions (they should be aborted, may lead to cascading aborts).
  – $x=y=1$, $w_1[x,2]$ $r_2[x]$ $w_2[y,3]$ if $T_1$ aborts $X$ is restored to 1, $T_2$ that read $x=2$ is aborted, $y$ is restored to 1.
  – Executions must be recoverable, that is cannot commit $T$ until all those transactions that $T$ read from are committed.
  – $x=y=1$, $w_1[x,2]$ $r_2[x]$ $w_2[y,3]$ $c_2$ is not recoverable. $T_2$ reads from $T_1$ and commits prior to $T_1$’s commit.
Avoid Cascading Aborts

• To ensure recoverability we sometimes use aborts which may lead to cascading aborts.
• Aborting transactions wastes work.
• A DBS avoids cascading aborts if it ensures that only data written by committed transactions is read.
  – This ensures that the commit of a transaction happens after the commit of all transactions it read from.
Undoing effects of aborted transactions

- There is a problem in undoing the effects of an aborted transaction.
- Suppose T wrote into x and aborted, if the execution avoids cascading abort, no other transaction need be aborted.
- How do we undo T’s effect on x?
- Erase T and its ops from the execution.
- The value of x should be the value due to the modified execution. This is the meaning of “as if T never happened”.
- But what is this value?
The right value for x

- $w_1[x,1] \ w_1[y,3] \ w_2[y,1] \ c_1 \ r_2(x) \ a_2$ now $y=1$.
- T2 now aborts, the modified execution is:
  - $w_1[x,1] \ w_1[y,3] \ c_1$
- The value of $y$ should be 3.
- The **before image** of $w[y,val]$ is the value $y$ had immediately prior to the op.
- For $w_2[y,1]$ the before image is 3.
- It’s convenient to undo by restoring to before images.
- But, it is not always possible to undo this way.
Before images are not always the right answer

- \( x=1, \ w1[x,2] \ w2[x,3] \ a1 \ x=3 \)
- The modified execution is \( x=1, \ w2[x,3] \ x=3 \)
- The before image of \( w1[x,2] \) is \( x=1 \).
- Restoring to \( x=1 \) is wrong!
- Consider \( x=1, \ w1[x,2] \ w2[x,3] \ a1 \ a2 \ x=3 \)
- The modified execution is: \( x=1 \)
- So, we need restore to \( x=1 \) which was twice overwritten ...
Strict executions

- The source of our problem: two transactions that have not committed wrote into x.
- Can require that (*) w[x, val] is delayed until all transactions that previously wrote x are committed or aborted.
- Similar to (**) r[x] is delayed until all transactions that previously wrote x are committed or aborted (used in preventing cascading aborts).
- Strict executions: those satisfying (*) and (**).
- So, no cascading aborts and restoration with before images (and also recoverable).
- Strict executions correspond to 02 (lecture 3).
Problem: Inconsistent Retrieval

- $r_1[s]$
  - $s := s - 1000$
  - $w_1[s]$

- $s = 1000, c = 500$
  - $r_1[c]$
    - $c := c + 1000$
    - $w_1[c]$

- $r_2[s]$
  - $s = 2000, c = 500$
  - $s = 1000, c = 500$

- $r_2[c]$
  - $s = 1000, c = 500$
  - $s = 1000, c = 1500$

Print $s + c$ (1500)
Problem: Lost Update

r1[c]
c := c + 1000
w1[c]
c = 1500
r1[c]
c := c + 2000
w1[c]
c = 2500

c = 500
Serializable execution

- Serial executions are correct because each transaction is assumed to be a correct database transformation.
- True serial executions are inefficient.
- A serializable execution is one that has the same effect as that of a serial execution.
- So, serializable executions are correct.
- Serializability is our notion of correctness.
Example of a Serializable execution

• $L = w0[x] w0[y] w0[z] r1[x] r1[z] r3[x] r2[z]$
  $w1[x] w3[y] w2[y] w2[z]$ 
• $L' = w0[x] w0[y] w0[z] r3[x] w3[y] r1[x] r1[z]$
  $w1[x] r2[z] w2[y] w2[z]$ 
• $L'$ is serial and $L$ is therefore serializable.
Consistency Preservation

• Consistency is defined by a set of predicates that need hold on database instances.
• Transactions are required to preserve database consistency.
• So, transactions perform consistent transformations.
Ordering transactions

• To obtain a specific ordering of operations, users need ensure it by submitting one operation after the other is complete.
• This does not always work.
• Consider a scheduler producing
  – r3[x] r1[x] w1[x] c1 r2[z] w2[z] c2 w3[z] c3
• T1 and T2 are not interleaved.
• The only equivalent serial order is:
  – r2[z] w2[z] c2 r3[x] w3[z] c3 r1[x] w1[x] c1
• So, if T2 is submitted after T1 commits this does not ensure that in the resulting equivalent serial order T1 precedes T2.
• If 2PL is used then this behavior cannot happen.