Advanced Databases

Optimistic Concurrency Control and Advanced Transaction Models

Dr. George Mertzios
Michaelmas Term

george.mertzios@durham.ac.uk
Room 2066, MCS Building
Tel: 42 429

Course Outline

- Enhanced Entity-Relationship (EER) Model
- Semistructured Databases - XML
- XML Data Manipulation - XPath, XQuery
- Transactions and **Concurrency Control**
- Distributed Transactions
- Distributed Concurrency Control

Concurrency control

Two main approaches to ensure consistency when executing transactions concurrently:

- Conservative (pessimistic) methods:
  - prevent conflicts (i.e. before they arise)
  - delay (or restart) conflicting transactions
- We have seen:
  - Locking: grant a “lock” on specific data items to avoid conflicting transaction operations
  - Timestamping: roll back and restart conflicting transactions

Optimistic concurrency control

- <u>Optimistic</u> methods:
  - assume that transactions are rarely in conflict

  $\Rightarrow$ more efficient to allow transactions proceed
  - check for conflicts at the end
    - just before the transaction commits
    - if there is evidence for possible conflict: abort / restart
- Two (or three) phases:
  - Read phase
  - Validation phase
  - Write phase (only for write operations)

Optimistic concurrency control

- Read phase:
  - runs from the beginning until just before commit
  - read all items from the database
  - store them in local variables
  - update only the local copies
    - not the real data items !

- In the read phase:
  - both read and write operations occur
  - but only in a virtual database !

Optimistic concurrency control

- Validation phase:
  - after the read phase
  - check: will serializability be violated if you commit ?
  - if something looks suspicious: abort and restart
    - i.e. just discard the local copies of the data items

- For read-only transactions:
  - check: are the read values still current ?
  - if yes: commit   (reading was correct)
  - if not:
    - somebody updated a value in the meanwhile

  $\Longrightarrow$ abort and restart

Optimistic concurrency control

- For transactions that update values:
  - check whether the transaction: how to check ?
    - will leave the database in a consistent state
    - with serializability maintained
      (i.e. equivalent to *some* serial schedule)
  - if yes: proceed to write phase
  - if not (or not sure): abort and restart
- Write phase (for updating transactions):
  - only after successful validation
  - apply the local copies to the database

Validation phase

- Each transaction *T* gets three timestamps:
  - at the start of its execution: *start(T)*
  - at the start of the validation phase: *validation(T)*
  - at its finish time (incl. writing phase): *finish(T)*
- First check to pass validation:
  - all transactions *S* with earlier start-timestamps finished before *T* started: *finish(S)* < *start(T)*
- In this case: clearly no conflicts
  - all items *T* has read are still current
  - all items *T* has written were not overridden

Validation phase

- Second check to pass validation:
  - suppose *T* started before *S* finished,
    i.e. *start(T) < finish(S)*
- Then check that both:
  a) all data items written by *S* were not read by *T*
  b) *S* completes its write phase before *T* enters its validation phase,
     i.e. *start(T) < finish(S) < validation(T)*

a) guarantees that *T* has read current values

b) guarantees that writes of *T* are done serially, i.e. with no conflicts with *S* can ever arise

Validation phase

Note: these are too strong guarantees for serializability

- Example for condition (a):
  - if *T* reads only after *finish(S)* an item that *S* wrote

$\Longrightarrow$ *S* and *T* still not in conflict

- Example for condition (b):
  - assume *validation(T)* < *finish(S)*
  - if *S* wrote different items than *T*

$\Longrightarrow$ *S* and *T* still not in conflict

Validation phase

Note: these are too strong guarantees for serializability

$\Longrightarrow$ transactions restart more often than needed

However:

- roll back involves only a local copy of data

  $\Longrightarrow$ no cascading rollback

- more accurate validation tests would:
  - imply more processing time for each transaction
  - be useless, assuming conflicts are rare

- If conflicts are not so rare:
  - conservative methods may be faster
  - locking (just wait): faster than repeating transactions

A bank analogy

Pessimistic method:

- when you enter the bank, a guard at the door checks your account number
- if someone else (e.g. your spouse) is already in the bank accessing your account:
  - you cannot enter until the current transaction finishes

Optimistic method:

- you can always enter the bank to do your business
- when you walk out, the guard may suspect that your transaction conflicted with another one
  - then you go back and do it again

Advanced transaction models

So far, we considered flat transactions:

- with a single start-point and a single termination-point
- suitable for traditional databases:
  - simple nature of data (integers, decimal, short strings…)
  - short duration of transactions (seconds / minutes)

In modern database applications:

- transactions can have a long duration

$\Longrightarrow$ more susceptible to failure

$\Longrightarrow$ loss of significant amount of work (in case of a rollback)

$\Longrightarrow$ unacceptably long delays

$\Longrightarrow$ ideally: recover to a state shortly before the crash

Advanced transaction models

Further problems of flat transactions:

- large number of data items accessed

$\Rightarrow$ to preserve transaction isolation:

  - many items inaccessible (locked) for extended periods

$\Rightarrow$ limited concurrency

- deadlocks are more likely
  - experimentally shown: frequency of deadlocks increases to the 4th power of transaction size

- To mitigate these problems:
  - nested transaction model
  - dynamic restructuring
  - sagas
  - workflow models

Nested transaction model

- Nested transaction model:
  - permit a transaction to contain other transactions
  - a tree (or hierarchy) of (nested) sub-transactions
  - each with own start / termination
  - only leaf sub-transactions perform database operations

- Bottom-up execution of transactions:
  - child starts before and finishes after the parent

- Abort / Commit at top level: the usual semantics
  - abort undoes all updates
  - commit permanently records updates

$\Rightarrow$ top-level: ACID properties of flat transactions

Nested transaction model

- Updates of a committed sub-transaction:
  - visible only to immediate parent
  - commit is conditional to commit/abort of ancestors
  - sub-transactions are not durable *(ACI<u>D</u>)*
    - updates become permanent only when top-level transaction commits

- During a sub-transaction execution:
  - updated items are isolated

$\Longrightarrow$ visible only to parent, not to other (sub-)transactions

Nested transaction model

- If a sub-transaction aborts:
  - just informs the parent node
  - no effect on the progress at higher levels
  - parent can choose how to proceed

- Possible reactions of parent node:
  1. retry the sub-transaction
  2. ignore the failure $\Longrightarrow$ non-vital sub-transaction
     (e.g. no rental car when booking a flight)
  3. run a “contingency” (alternative) sub-transaction
     (e.g. book *Sheraton* if booking at *Hilton* fails)
  4. Abort

Nested transaction model

Example:

```
begin_transaction T1                                  Complete Reservation
    begin_transaction T2                              Airline_reservation
        begin_transaction T3                          First_flight
            reserve_airline_seat(London, Paris);
        commit T3;
        begin_transaction T4                          Connecting_flight
            reserve_airline_seat(Paris, New York);
        commit T4;
    commit T2;
    begin_transaction T5                              Hotel_reservation
        book_hotel(Hilton);
    commit T5;
    begin_transaction T6                              Car_reservation
        book_car();
    commit T6;
commit T1;
```

Nested transaction model

- Advantages
  - modularity and finer level of granularity:
    - easier to handle for concurrency / recovery
  - sub-transactions can execute concurrently
  - aborted sub-transactions can roll back without side-effects to each other

$\Longrightarrow$ mitigates the effects of a long-duration transaction

- Simulating nested transactions with savepoints:
  - identifiable points in flat transactions, representing a partially consistent state
  - can be used as internal restart points

$\Longrightarrow$ again finer unit of recovery than a whole transaction

Sagas

- Saga:
  - an ordered sequence of flat transactions that can be interleaved with each other
  - for every sub-transaction T_i there is a compensating transaction C_i which undoes the effects of T_i

- The effects of the saga $T_1, T_2, \ldots, T_n$:
  - $T_1, T_2, \ldots, T_n$, if it completes successfully
  - $T_1, T_2, \ldots, T_i, C_i, C_{i-1}, \ldots, C_1$, if sub-transaction T_i fails/aborts

- Saga for the flight reservation example:
  - T_3, T_4, T_5, T_6 : the leaf nodes of the top-level transaction
  - easy to derive compensating transactions C_3, C_4, C_5, C_6

Sagas

- In contrast to flat transactions, sagas:
  - relax the Isolation property *(ACID)*
  - reveal partial results to other transactions (before they completes)
- Sagas are generally useful when:
  - sub-transactions are relatively independent
  - compensating transactions can be produced
- Compensating transactions not always easy:
  - e.g. when dispensing cash from an automatic teller machine

Dynamic restructuring

- Split-transaction operation:
  - transaction T is split into two serializable transactions A, B
    - only possible if serializability is guaranteed
  - their actions and resources are divided (e.g. locked data items)
  - the new transactions proceed independently

$\Longrightarrow$ partial results of T become visible by other transactions

- Example scenario:

Programmer Bob edits and recompiles module F, and then works for several days on other modules. Another programmer Alice wants to test her own changes to a module G, but has to wait until Bob commits his work, in order to be able to read module F's code to build the system executable.

The length of Bob's transaction may prevent Alice from carrying out productive work for unacceptably long time.

Dynamic restructuring

- Three conditions for split-transaction:
  1. A-Write-Set $\cap$ B−Write−Set $\subseteq$ B-Write-Last
     - if both A and B write to the same object, then B's write operations follow A's write operations
  2. A-Read-Set $\cap$ B−Write−Set $= \emptyset$
     - A cannot see (read) any results of B
  3. B-Read-Set $\cap$ A−Write−Set $=$ Share-Set
     - B may see (read) the results of A
- Then A is serializable before B

$\Rightarrow$ if A aborts then B must also abort

- If B-Write-Last $=$ Share-Set $= \emptyset$

$\Rightarrow$ A and B are serialized in any order

Dynamic restructuring

- Join-transaction operation:
  - the reverse of split-transaction
  - merges the ongoing work of two transactions, as though they have been a single transaction
- Combination of the operations:
  - split into transactions A, B
  - then join of B with C

$\Longrightarrow$ transfer resources among transactions, without making them available to other transactions

Dynamic restructuring

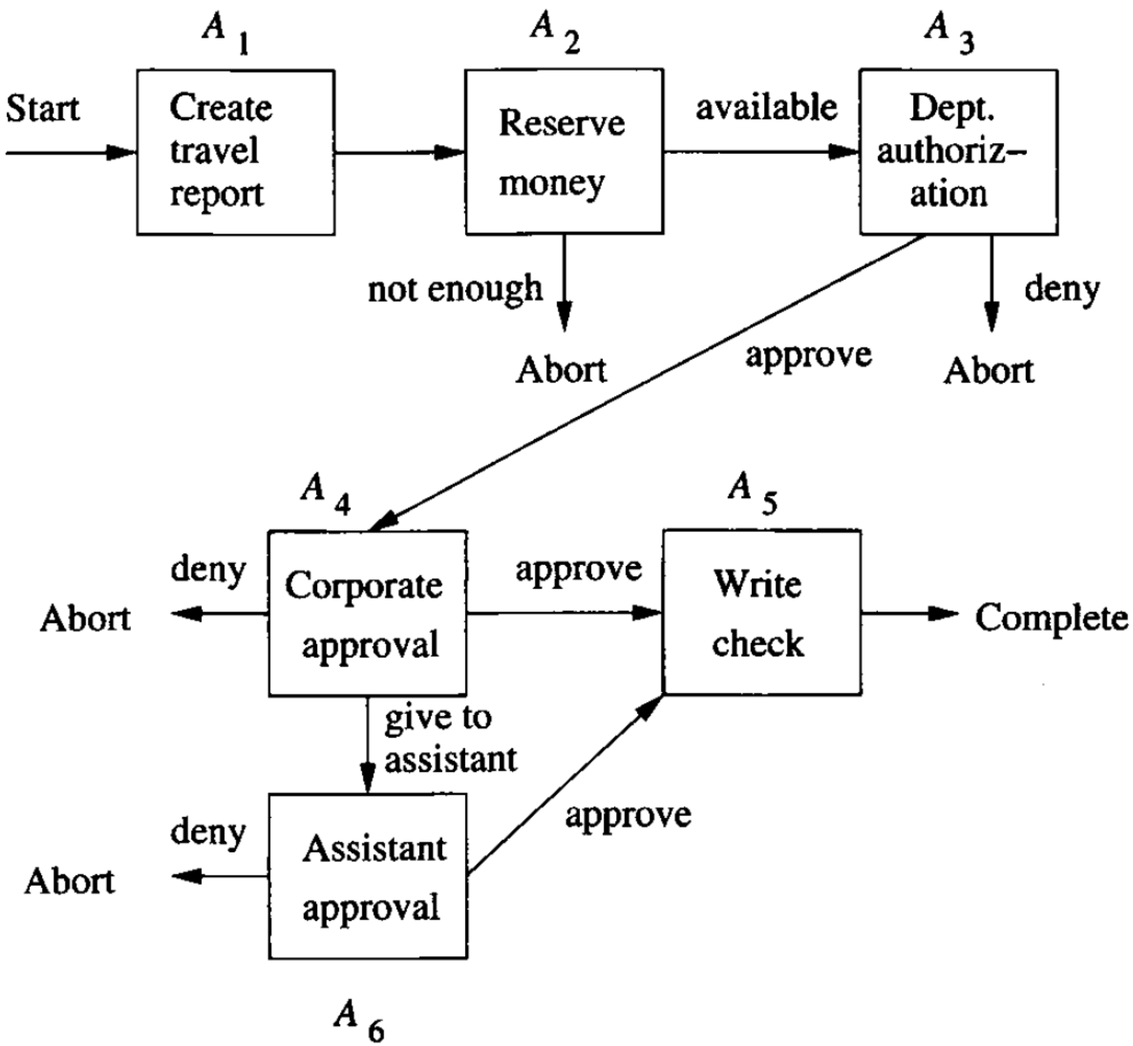
- Advantages
  - adaptive recovery:
    allows part of the work of a transaction to commit,
    $\Rightarrow$ not affected by subsequent failures
  - reduced Isolation: *(ACID)*
    resources are released by committing part of a transaction

$\Rightarrow$ mitigates the effects of a long-duration transaction

Workflow models

- All above models:
  - overcome some limitations of long-lived flat transactions
  - but still not powerful to model some real-world business
- Workflow:
  - an activity for coordinated execution of multiple tasks by people / processing systems (DBMS, applications, …)
- Workflow models:
  - complex models for specific business applications
  - hardly conform to any ACID property

Workflow models - Example



Workflow for a traveler requesting expense reimbursement

- Process may take several days
- If we implement this workflow with locking:
  - company account unavailable since action A_2, until transaction either aborts, or action A_5 completes

Additional Slides

Recovery from failures

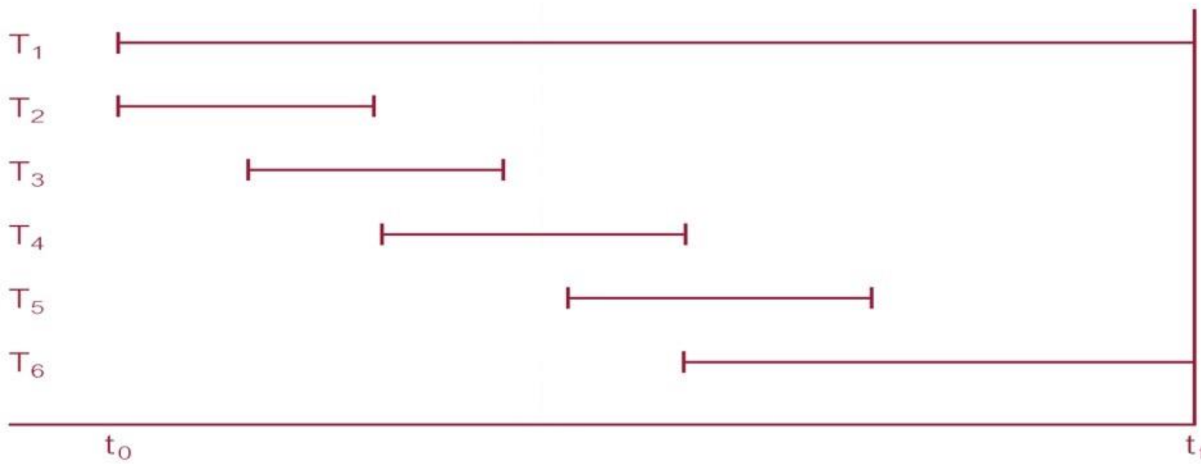
- <u>Database recovery:</u> the process of restoring the database to a correct state in the event of failure
- Possible failures:
  - system crashes (hardware / software errors)
  - media failures (e.g. head crashes / unreadable media)
  - application software errors (logical program errors)
  - natural physical disasters (fire / power failure)
  - carelessness / unintentional destruction of data
  - sabotage / intentional corruption or destruction of data

Recovery from failures

- At the time of failure:
  - if a transaction has not committed, recovery manager must undo (rollback) any effects of the transaction
    - for Atomicity *(<u>A</u>CID)*
  - if a transaction has committed, recovery manager must redo (rollforward) any effects of the transaction
    - for Durability *(ACI<u>D</u>)*

Recovery from failures

Example:



- DBMS starts at time t_0 but fails at time t_f
- T_1, T_6: not committed at the time of failure
- T_2, T_3, T_4, T_5: committed at the time of failure

$\Longrightarrow$ Undo T_1, T_6,   Redo T_2, T_3, T_4, T_5

Log file

- In order to recover from failures that affect transactions, the system maintains a log:
  - keep track of all update operations of the database
- Log is periodically backed up to *tape*
  - for the case of *catastrophic failure*
  - tapes are cheaper / more reliable than disk
  - nowadays: log is stored *online*, on a fast direct-access storage device
- Log records are written to the disk
  - it is only affected in case of catastrophic failure

Log file

Transaction identifiers

Pointers to previous / next log records in the same transaction

time

0	Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
1	T1	10:12	START				0	2
2	T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
3	T2	10:14	START				0	4
4	T2	10:16	INSERT	STAFF SG37		(new value)	3	5
5	T2	10:17	DELETE	STAFF SA9	(old value)		4	6
6	T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
7	T3	10:18	START				0	11
8	T1	10:18	COMMIT				2	0
		10:19	CHECKPOINT	T2, T3				
9	T2	10:19	COMMIT				6	0
10	T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
11	T3	10:21	COMMIT				11	0

3 transactions T_1, T_2, T_3

Commit point of a transaction

- A transaction *T* reaches its **commit point** when:
  - all its operations that access the database have been executed successfully and
  - the effect of all the transaction operations on the database have been recorded in the log
- After the commit point:
  - the transaction is said to be **committed**
  - a *commit record* is added into the log
  - its effect is assumed to be *permanently recorded* in the database
- In case of failure we use the log:
  - to roll back all started but not committed transactions
  - to redo all committed transactions (if needed)