Transactions

Peter Scheuermann

Transaction Concept

- Many enterprises and organizations use databases to store information about their state
  - e.g., Balances of all depositors at a bank
- When an event occurs in the real world that changes the state of the enterprise, a program is executed to change the database state in a corresponding way
  - e.g., Bank balance must be updated when deposit is made
- Such a program is called a **transaction**:
  a collection of one or more operations on one or more databases, which reflects a discrete unit of work
  - In the real world, this happened (completely) or it didn't happen at all (Atomicity)

What Does a Transaction Do?

- Return information from the database
  - RequestBalance transaction:
    Read customer's balance in database and output it

  => transactions can be read-only
- *Update the database to reflect the occurrence of a real world event*
  - *Transfer* money between accounts
    - Update customers' balances in database(s)
  - Purchase a group of products
  - Students enrolling in an unit of study
- Cause the occurrence of a real world event
  - Withdraw transaction:
    Dispense cash (and update customer's balance in database)

Transaction Concept (continued)

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. 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)*
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

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)*
- **Atomicity requirement**
  - if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- 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** in above example:
  - the sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement — if** between steps 3 and 6, another transaction T2 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).

T1	**T2**
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- **Isolation can be ensured trivially by running transactions serially**
  - that is, one after the other.
- **However, executing multiple transactions concurrently has significant benefits, as we will see later.**

Transactional Guarantees

- The execution of each transaction must maintain relationship between the database state and the enterprise state
  - correctness and consistency of the database is paramount!
- Therefore additional requirements are placed on the execution of transactions beyond those placed on ordinary programs:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

ACID properties

A C I D Properties

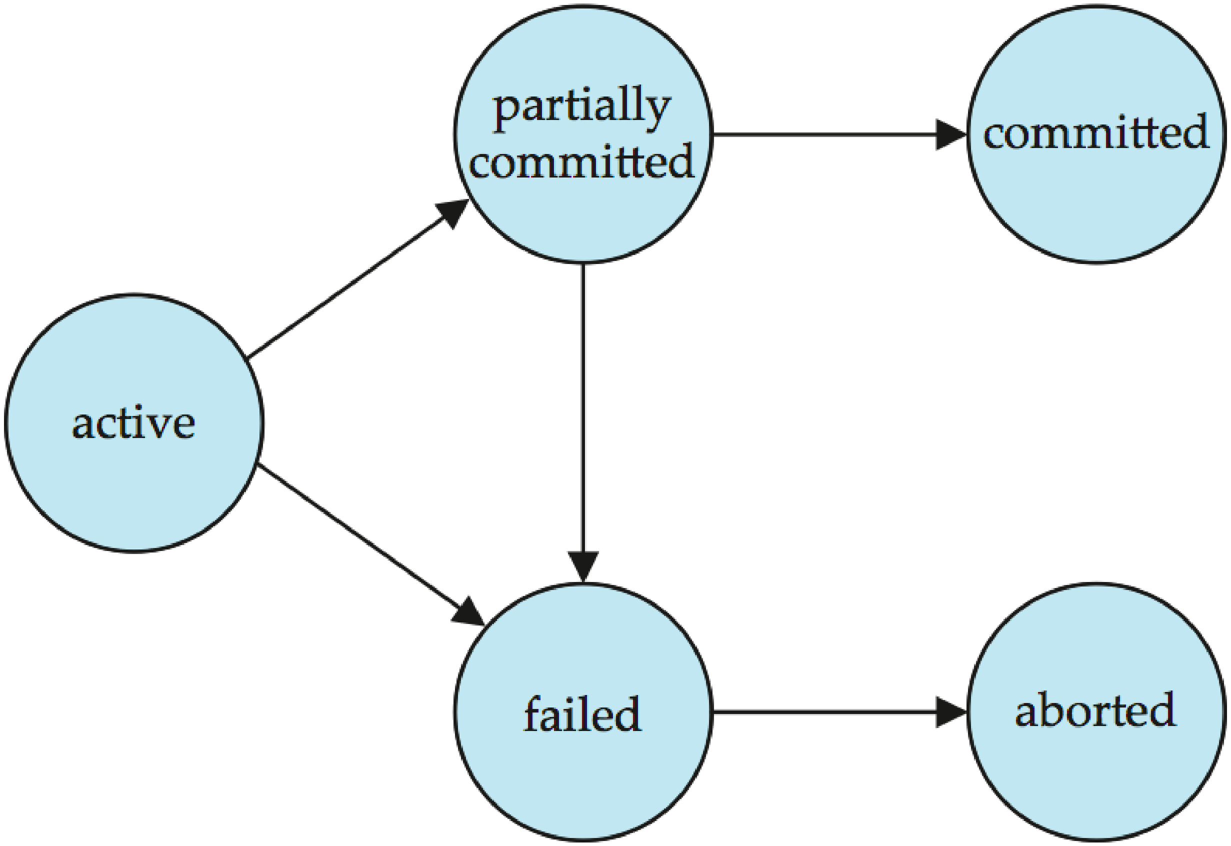
- **Atomicity**. Transaction should either complete or have no effect at all
  - In case of a failure, all effects of operations of not-completed transactions are undone.
- **Consistency**. Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
  - Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability**. The effect of a transaction on the database state should not be lost once the transaction has committed

ACID properties handled transparently for the transaction by the DBMS

Transaction State

- **Active –** the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed –** after successful completion.

Transaction State (Cont.)



A - Atomicity

- A real-world event either happens or does not happen
  - Student either registers or does not register
- Similarly, the system must ensure that either the corresponding transaction runs to completion or, if not, it has no effect at all
  - a user can think of a transaction as always executing all its actions in one step, or not executing any actions at all.
  - Not true of ordinary programs. A crash could leave files partially updated on recovery.
    - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.
    - Also, in case of a failure, all actions of not-committed transactions are *undone*.

API for Transactions

- Data manipulation language must provide commands for setting transaction boundaries. For example:
  - begin transaction
  - commit ; rollback
- In many DBMS such as Oracle, a transaction begins implicitly
  - Some other DBMS (eg. SQL Server or PostgreSQL) provide a **BEGIN TRANSACTION** command
- A transaction ends by:
  - **COMMIT** *requests* to **commit** current transaction
    - The system might commit the transaction, or it might abort if needed.
  - **ROLLBACK** causes current transaction to **abort** - *always satisfied.*
- The commit command is a request
  - The system might commit the transaction, or it might abort it

Transaction Example

- Pseudocode for a product order transaction:

```
display greeting
get order request
 BEGIN TRANSACTION
 SELECT  product record
  IF product is available THEN
     UPDATE quantityOnOrder of product record
 INSERT order record
    send message to shipping departement
    COMMIT
 ELSE
    ROLLBACK
 END IF
```

Another Transaction Example

- Transaction in Embedded SQL

```
EXEC SQL BEGIN DECLARE SECTION
   int flight;
   char  date[10]
   char seat [3]
   int occ;
EXEC SQL END DECLARE SECTION
START TRANSACTION
Void chooseSeat()  {
/* C code to prompt the user to enter flight, date, and seat and store these in the three variables
with those name */
EXEC SQL Select occupied into :occ
           From Flights
          Where fltNum=:flight and fltDate=:date and fltSeat=:seat;
If (!occ) {
  EXEC SQL Update Flights
              Set occupied = true
            Where  fltNum=:flight and fltDate=:date and fltSeat=:seat;
/*C and SQL code to record the seat assignment and inform the user of the assignment  */
}
Else /* C code to notify user of unavailability and ask for another seat selection */
} EXEC COMMIT;
```

Database Consistency

- **Enterprise (Business) Rules** limit the occurrence of certain real-world events
  - Student cannot register for a course if the current number of registrants equals the maximum allowed
- Correspondingly, allowable database states are restricted
  - *cur_reg* <= *max_reg*
- These limitations are called (static) **integrity constraints**: assertions that must be satisfied by the database state

- **Database is consistent** if all static integrity constraints are satisfied

Transaction Consistency

- A consistent database state does not necessarily model the actual state of the enterprise
  - A deposit transaction that increments the balance by the wrong amount maintains the integrity constraint $balance \geq 0$, but does not maintain the relation between the enterprise and database states
  - Dynamic Integrity Constraints: Some constraints restrict allowable state transitions
    - A transaction might transform the database from one consistent state to another, but the transition might not be permissible
    - **Example:** Students can only progress from Junior to the Senior year, but can never be degraded.
- A consistent transaction maintains database consistency *and* the correspondence between the database state and the enterprise state (implements its specification)
  - Specification of deposit transaction includes

$$balance = balance' + amt\_deposit\ ,$$

($balance'$ is the initial value of $balance$)

Transaction Consistency (cont'd)

- A **transaction is consistent** if, assuming the database is in a consistent state initially, when the transaction completes:
  - All static integrity constraints are satisfied (but constraints might be violated in intermediate states)
    - Can be checked by examining snapshot of database
  - New state satisfies specifications of transaction
    - Cannot be checked from database snapshot
  - No dynamic constraints have been violated
    - Cannot be checked from database snapshot

Integrity Constraints and Transactions

- When do we check integrity constraints?
  - *Immediate* after an SQL statement or *at the end* of a transaction?

- Remember from last week:
  - Integrity constraints may be declared:
    - **NOT DEFERRABLE**
      The default. It means that every time a database modification occurs, the constraint is checked immediately afterwards.
    - **DEFERRABLE**
      Gives the option to wait until a transaction is complete before checking the constraint.

Deferrable Integrity Constraints Example

```
CREATE TABLE COURSE (
   c_code          VARCHAR(8),
   title           VARCHAR(220),
   lecturer        INTEGER,
   credit_points   INTEGER,
   CONSTRAINT COURSE_PK PRIMARY KEY (c_code),
   CONSTRAINT COURSE_FK FOREIGN KEY (lecturer)
     REFERENCES Lecturer DEFERABBLE INITIALLY IMMEDIATE
);//this is Oracle syntaz

;
  INSERT INTO Course  VALUES('EECS339','DBMS', 42, 1);
  INSERT INTO Lecturer VALUES(42,'Charley Sheen', …);
   COMMIT;
```

lecturer 42 has to exist for the FK to be OK

D - Durability

- The system must ensure that once a transaction commits, its effect on the database state is not lost in spite of subsequent failures
  - Not true of ordinary programs. A media failure after a program successfully terminates could cause the file system to be restored to a state that preceded the program’s execution

- Implementing Durability:
  - Database is stored redundantly on mass storage devices to protect against media failure
  - Architecture of mass storage devices affects type of media failures that can be tolerated
  - Related to **Availability**: extent to which a (possibly distributed) system can provide service despite failure
    - Non-stop DBMS (mirrored disks)
    - Recovery based DBMS (log)

I - Isolation

- **Serial** Execution: transactions execute in sequence
  - Each one starts after the previous one completes.
    - Execution of one transaction is not affected by the operations of another since they do not overlap in time
  - The execution of each transaction is **isolated** from all others.
- If the initial database state and all transactions are consistent, then the final database state will be consistent and will accurately reflect the real-world state, *but*
  - Serial execution is inadequate from a performance perspective
- **Concurrent execution** offers performance benefits:
  - A computer system has multiple resources capable of executing independently (e.g., cpu's, I/O devices), *but*
  - A transaction typically uses only one resource at a time
  - *but* might not be correct…

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- **A serial schedule in which T_1 is followed by T_2 :**

T_1	T_2
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A)
	$temp$:= A * 0.1
	A := A - $temp$
	write (A)
	read (B)
	B := B + $temp$
	write (B)
	commit
read (A)	
A := A – 50	
write (A)	
read (B)	
B := B + 50	
write (B)	
commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of (*A* + *B*).

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **The Issue:** Maintaining database correctness when many transactions are accessing the database concurrently
  - Basic Assumption: *Each transaction preserves database consistency*.
  - Thus serial execution of a set of transactions preserves database consistency.
- **Serializability**:
  A schedule is **serializable** if it is equivalent to a serial schedule
  - Different notions of schedule equivalence…
  - This lecture concentrates on **conflict serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflict Serializability

- Two schedules are *conflict serializable* if:
  - They involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way

- Two actions a_i and a_j of transactions T_i and T_j **conflict** if and only if they access the same data X and at least one of these actions wrote X. (a_i, a_j) are called a **conflict pair**.
  1. a_i = read(X), a_j = read(X). don't conflict.
  2. a_i = read(X), a_j = write(X). they conflict.
  3. a_i = write(X), a_j = read(X). they conflict
  4. a_i = write(X), a_j = write(X). they conflict

- **Note**: With SQL - Select corresponds to read,
  Insert, Delete, Update correspond to write

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Serializable Schedule Example 1

- The concurrent schedule

  $$S: r_1(x)\ w_2(z)\ w_1(y)$$

  is equivalent to the serial schedules of T_1 and T_2 in either order:
  - T1, T2: $r_1(x)\ \ w_1(y)\ w_2(z)$ and
  - T2, T1: $w_2(z)\ r_1(x)\ \ w_1(y)$
- Reason: operations of distinct transactions on <u>*different*</u> data items commute.
- Hence, *S* is a *serializable* schedule

Serializable Schedule Example 2

- The concurrent schedule

  S: r1(z) r2(q) w2(z) r1(q) w1(y)

  is equivalent to the serial schedule *T1,T2*:

  r1(z) r1(q) w1(y) r2(q) w2(z)

  since <u>read operations</u> of distinct transactions on the same data item <u>commute</u>.
- Hence, *S* is a serializable schedule
- However, *S* is not equivalent to *T2,T1* since read and write operations (or two write operations) of distinct transactions on the same data item do not commute.

Non-Serializable Schedule Example

- Example: course registration; *cur_reg* is the number of current registrants

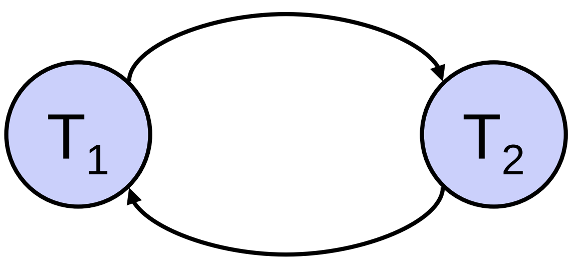
$T1: r(cur\_reg : 29) \qquad\qquad\qquad\qquad\qquad\qquad w(cur\_reg : 30)$

$T2: \qquad\qquad\qquad r(cur\_reg : 29)\ w(cur\_reg : 30)$

- Schedule not equivalent to *T1,T2* or *T2,T1*
- Database state no longer corresponds to real-world state, integrity constraint violated

Testing for Conflict Serializability

- Consider some schedule of a set of transactions $T_1, T_2, \ldots, T_n$
- **Precedence graph:**
  - direct graph where the vertices are the transactions.
  - edge from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- Example:
  T_1 and T_2 have
  2 conflict pairs



- Central Theorem:
  A schedule is ***conflict serializable*** if and only if its *precedence graph is acyclic*.
  - The serializability order can be obtained by a *topological sorting* of the graph.