INFO2120 – INFO2820 – COMP5138 Database Systems

Week 9: Transaction Management

(Kifer/Bernstein/Lewis – Chapter 18; Ramakrishnan/Gehrke – Chapter 16; Ullman/Widom – Chapter 6.6)

Dr. Uwe Röhm
School of Information Technologies

Learning Objectives for this Week

- **Understanding of Transaction Management**
  - What is a *transaction* and why are transactions important?
  - *ACID* guarantees given by databases
  - How to define *transactions in SQL*
- **The Theory behind: Correctness of Transaction Programs**
  - Idea of *conflict serializability*
  - how it avoids *update anomalies*
- **Overview of Concurrency Control techniques**
  - *2 Phase Locking* vs. *Snapshot Isolation*
  - *Deadlock* problem
  - Correctness vs. Performance: *SQL Isolation Levels*

Based on slides from Kifer/Bernstein/Lewis (2006) "Database Systems" and from Ramakrishnan/Gehrke (2003) "Database Management Systems" and also including material from Fekete and Röhm.

Transaction Concept

- Many enterprises and organisations 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)

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 transparent for the transaction by the DBMS

A - Atomicity

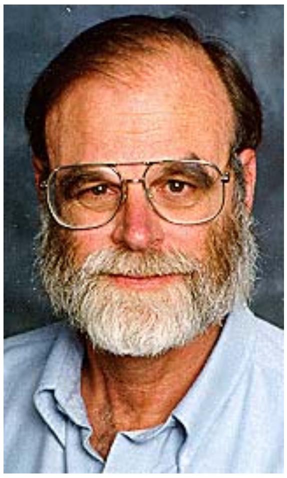
- A real-world event either happens or does not happen
  - Student either registers or does not register
- Similarly, the system must ensure that either the corresponding transaction runs to completion or, if not, it has no effect at all
  - a user can think of a transaction as always executing all its 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*.

Commit and Abort

- If the transaction successfully completes it is said to **commit**
  - The system is responsible for ensuring that all changes to the database have been saved
- If the transaction does not successfully complete, it is said to **abort**
  - The system is responsible for undoing, or **rolling back**, all changes – in the database! - that the transaction has made
  - Possible reasons for abort:
    - System crash
    - Transaction aborted by system
      - Execution cannot be made atomic (a site is down)
      - Execution did not maintain database consistency (integrity constraint violated)
      - Execution was not isolated
      - Resources not available (deadlock)
    - Transaction requests to roll back

Famous DB Researchers: Jim Gray

- One of the most influential database researchers and software designers
  - Co-authored the 'bible' on Transaction Mgmt: "*Transaction Processing: Concepts and Techniques*" (Gray/Reuter, 1993)
  - first Ph.D. from CS at UC Berkeley (1969)
  - worked for
    IBM (at System R)
    Tandem-Computers, DEC, …,
    Microsoft Research



Jim Gray (1944-2007)

- Turing Award (1998)
  - "for seminal contributions to database and ***transaction processing research*** and technical leadership in system implementation."

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. Sybase, 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 for one of the reasons on the previous slide.

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
   COMMIT
   send message to shipping department
 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
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 BEGIN TRANSACTION
   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;
}
```

Transactions in JDBC

- Some database APIs provide explicit functions for controlling the transaction semantics
- Example from Week 7: JDBC and Transactions
  - JDBC's Connection Class:
    - Provides **`commit()`** and **`rollback()`** methods
    - allows to set SQL Isolation levels (see later)
    - allows to set AutoCommit mode on/off
- By default, transactions are in **AutoCommit** mode
  - each SQL statement is considered its own transaction!
  - No explicit commit, no transactions with more than one statement…
- Hence: Have to set AutoCommit OFF first
  - `connection.setAutoCommit(false)`
  - Now a new transaction start implicit with first SQL statement, and explicit commit() or abort() calls are needed to end it

Transactions in JDBC - Example

```
public void bookFlight ( String flight_num, Date flight_date, Integer seat_no)
  {
    try {/* connect to the database */
      Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@oracle10…");

     /* set AUTOCOMMIT off; next SQL statement will start a transaction */
     conn.setAutoCommit(false);

     /* execute the SQL statements within the transaction */
      PreparedStatement stmt = conn.prepareStatement("SELECT occupied
                                                     FROM Flight
                                       WHERE flightNum=? AND flightDate=? AND seat=?");
      stmt.setString(1, flight_num); stmt.setDate(2, flight_date);  stmt.setInteger(3, seat_no);
      ResultSet rset = stmt.executeQuery();

     If ( !rset.empty() && rset.next().getInteger()==0 ) {
         stmt = conn.prepareStatement("UPDATE Flight SET occupied=TRUE
                                   WHERE flightNum=? AND flightDate=? AND seat=?");
         stmt.setString(1, flight_num); stmt.setDate(2, flight_date);  stmt.setInteger(3, seat_no);
         stmt.executeUpdate();
     }
     /* COMMIT the transaction */
     conn.commit();

     stmt.close();
     conn.close();
   }
  catch (SQLException sqle) {
     /* error handling */
  }
}
```

Transactions in PHP

- Most database extension of PHP rely solely on the database for transactions and assume every statement to be separate
  - one has to define transaction blocks using explicit SQL statements
  - Alternatively: Use of stored procedures!
    - Exemption: the Oracle 'oci8_' library works similar to JDBC with an autocommit flag and **oci8_commit()** and **oci8_rollback**() functions
- To define transactions in PHP for PostgreSQL, use explicit BEGIN TRANSACTION and COMMIT (or ROLLBACK) 'queries':

```
<?php
   $db = new PDO("dbname=somebankdb", $user, $pw);
   $db->beginTransaction();
     $stmt = $db->query("UPDATE Accounts
              SET balance = balance-100.00 WHERE name = 'Alice'")
     //… etc etc
   $db->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

C - 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* ≥ *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 via Intermediate 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
- Note that this is mainly the responsibility of the application developer!
  - database cannot 'fix' the correctness of a badly coded transaction

Integrity Constraints and Transactions

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

- Remember from week 05:
  - 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 UnitOfStudy (
   uos_code      VARCHAR(8),
   title         VARCHAR(220),
   lecturer      INTEGER,
   credit_points INTEGER,
   CONSTRAINT UnitOfStudy_PK PRIMARY KEY (uos_code),
   CONSTRAINT UnitOfStudy_FK FOREIGN KEY (lecturer)
     REFERENCES Lecturer DEFERABBLE INITIALLY IMMEDIATE
);

BEGIN TRANSACTION;
  SET CONSTRAINTS UnitOfStudy_FK DEFERRED;
  INSERT INTO Teaching VALUES('info1000',2009,'S1',42);
  INSERT INTO Lecturer VALUES(42,'Steve McQueen', …);
COMMIT;
```

lecturer 42 has to exist for the FK to be OK

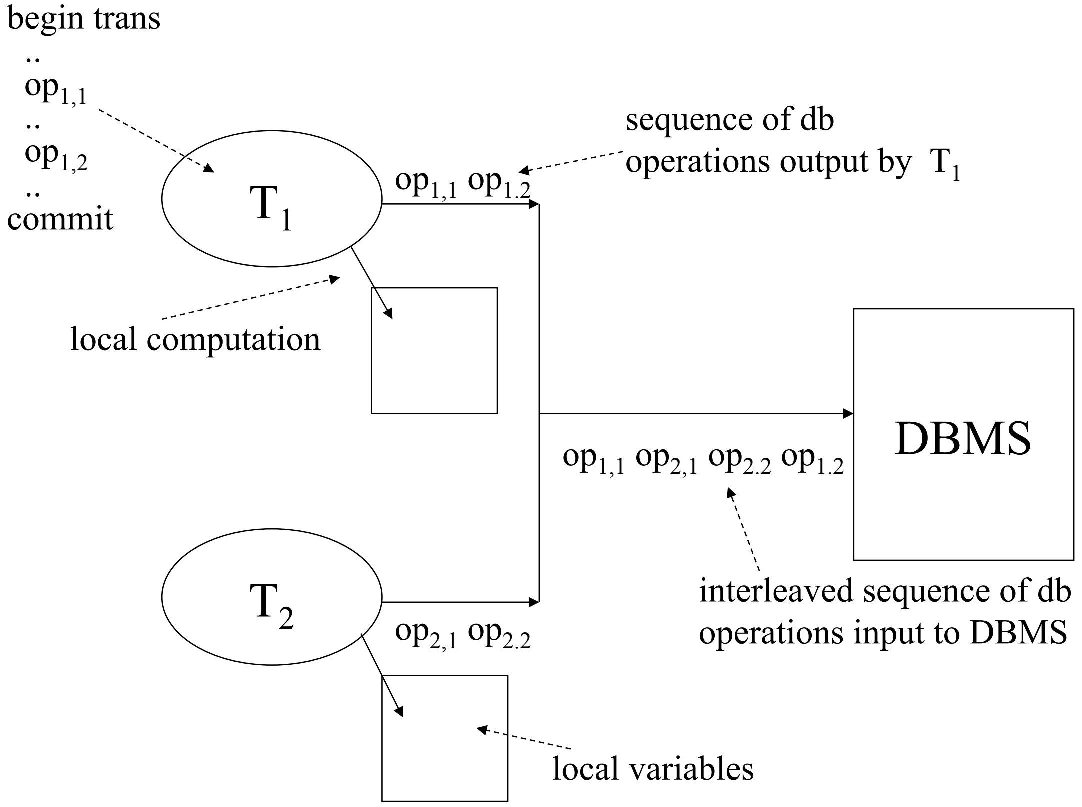
D - Durability

- 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…

Concurrent Execution



Example

- Let‘s consider two transactions:

```
T1:  BEGIN   A=A-100,    B=B+100    END
T2:  BEGIN   A=1.05*A,   B=1.05*B   END
```

  - Transaction T1 is transferring $100 from account *A* to account *B*.
    The second transaction credits both accounts with a 5% interest payment.
- *Atomicity requirement* — all updates are reflected in the db or none.
- *Consistency requirement* – T1 does not change the total sum of *A* and *B*, and after T2, this total sum is 5% higher.
- *Isolation requirement* — There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect must be equivalent to these transactions running serially in some order.
- *Durability requirement* — once a transaction has completed, the updates to the database by this transaction must persist despite failures

Example (cont' d)

- Consider a possible interleaving (**schedule**):

```
T1: A=A-100,             B=B+100
T2:           A=1.05*A,            B=1.05*B
```

- This is OK. But what about:

- The DBMS's view of the second schedule:

[text boxes to be filled-in in lecture]

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR conflicts, "**dirty reads**"):

```
T1: R(A),W(A),                 R(B),W(B),Abort
T2:           R(A),W(A),Commit
```

- **Unrepeatable Reads** (RW conflicts):

```
T1: R(A),                      R(A),W(A),Commit
T2:       R(A),W(A),Commit
```

- Overwriting Uncommitted Data (WW conflicts,„**lost updates**"):

```
T1:  W(A),                         W(B),Commit
T2:         W(A),W(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 sequence of database operations is **serializable** if it is equivalent to a serial execution of the involved transactions
- Central Theorem of concurrent execution in DBMS:
  - A **serializable execution guarantees correctness** in that it moves a database from one consistent state to another consistent state

This Week's Agenda

- **Transaction Management**
  - Transaction Concept
  - Serializability
- **Concurrency Control Mechanisms**
  - Locking
  - Snapshot Isolation
  - SQL Isolation Levels

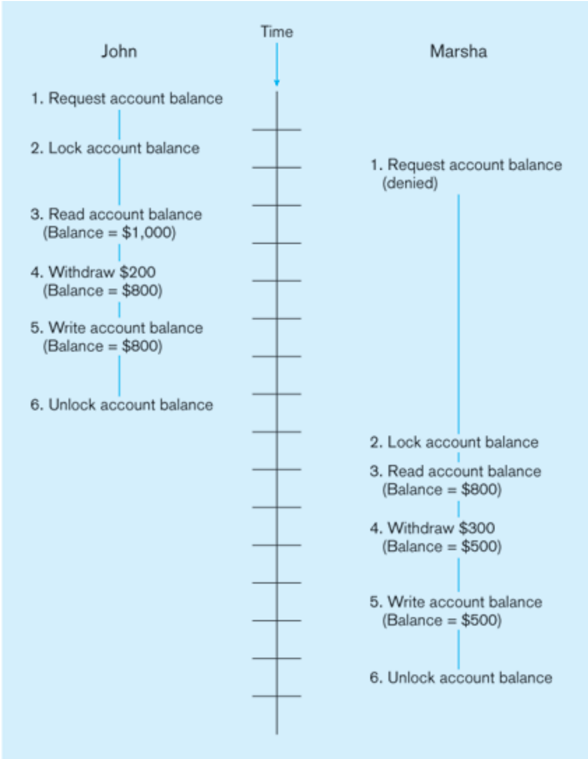
Concurrency Control vs. Serializability Tests

- Testing a schedule for serializability *after* it has executed is a little bit too late!
- **Concurrency Control:** The protocol that manages simultaneous operations against a database so that serializability *(actually: isolation levels)* is assured.
  - Such protocols will generally not examine the precedence graph as it is being created; instead a protocol will impose a discipline that avoids non-seralizable schedules.
  - Tests for serializability help understand why a concurrency control protocol is correct.
- Two important techniques:
  - Locking Protocol
  - Versioning (aka ‘Snapshot Isolation’)

Lock-based Concurrency Control

- **Two-phase Locking (2PL) Protocol:**
  - Locks are associated with each data item
  - A transaction must obtain a *S (shared) lock* on object before reading, and an *X (exclusive) lock* on object before writing.
    - *exclusive (X) lock*: Data item can be both read as well as written by just **one** transaction
    - *shared (S) lock*: Data item can only be read (but shared by transactions)
  - All locks held by a transaction are released when the transaction complete, and a transaction can not request additional locks once it releases any locks.
  - If a transaction holds an *X* lock on an object, no other transaction can get a lock (*S* or *X*) on that object.
    - Similar if a transaction requests a *X* lock of an already locked data object
    - Instead, such transactions must **wait** until the conflicting lock is released from the previous transaction(s)

Locking Example



Lock Compatibility Matrix and Lock Granularity

Held / Requested	Shared	Exclusive
Shared	OK	T2 wait on T1
Exclusive	T2 wait on T1	T2 wait on T1

- Locking Granularity: size of the database item locked
- database
- table / index
- page
- row
- column
- Tradeoff between waiting time and overhead

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

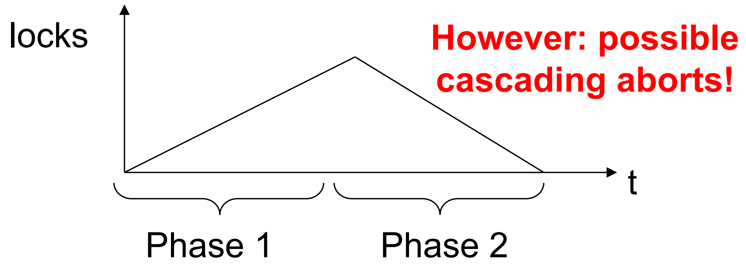
- Deadlock prevention
  - E.g. priorities based on timestamps
- Deadlock detection
  - A transaction in the cycle must be aborted by DBMS (since transactions will wait forever)
  - DBMS uses deadlock detection algorithms or timeout to deal with it
  - Most commonly used

Aborting a Transaction

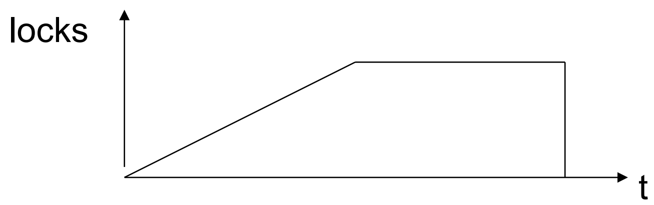
- If a transaction T_i is aborted, all its actions have to be undone.
  - if T_j reads an object last written by T_i, T_j must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If T_i writes an object, T_j can read it only after T_i commits.
- To undo the actions of an aborted transaction,DBMS maintains a log in which every write is recorded.
  - The log is also used to recover from system crashes:
    all active Xacts at the time of the crash are aborted when the system comes back up.

2PL versus Strict 2PL

- Two-Phase locking: All locks are acquired before any lock is released



Strict: Transaction holds all locks until completion



Isolation Levels

- Serializability provides a *conservative* definition of correctness
  - For a particular application there might be many acceptable *non-*serializable schedules
  - Requiring serializability might degrade performance
- In addition to *serializable*, every DBMS offers a variety of less stringent isolation levels
  - SERIALIZABLE is the most stringent (correct for *all* applications)
  - Lower levels of isolation give better performance
    - *Might* allow incorrect schedules
    - *Might* be adequate for some applications
    - Performance requirements might not be achievable if schedules are *serializable*
- Application programmer is responsible for choosing appropriate level!!!

Anomalies in Non-Serializable Schedules

- **Dirty read** (previous example – write lock given up early)

  $w_1(x)\ r_2(x)\ abort_1$
- **Non-Repeatable Read** (read lock given up early)

  $r_1(x)\ w_2(x)\ commit_2\ \ r_1(x)$
- **Lost Update** (result of non-repeatable read – read lock given up early)
  - Two transactions trying to deposit in the same bank account – the deposit of transaction 2 is lost

    $r_1(Bal)\ r_2(Bal)\ w_2(Bal)\ commit_2\ w_1(Bal)\ commit_1$

ANSI Standard Isolation Levels

- Defined in terms of anomalies
  - Anomaly prohibited at one level is also prohibited at all higher levels
  - **Serializable** — default according to SQL-standard…
    - In practice, most systems have weaker default level!
  - **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 record may return different (but committed) values. (*most common in practice!)*
  - **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.

Locks and Isolation Levels

- DBMS guarantees that each SQL statement is isolated
- Early (non-strict) lock release used to implement levels
  - Short-term locks - held for duration of single statement
  - Long-term locks - held until transaction completes (strict)
- At all levels, transactions obtain long-term write locks
- This means for isolation levels:
  - READ UNCOMMITTED - no read locks (dirty reads possible since transaction can read a write-locked item)
  - READ COMMITTED - short-term read locks on rows (non-repeatable reads possible since transaction releases read lock after reading)
  - REPEATABLE READ - long-term read locks on rows (phantoms possible)
  - SERIALIZABLE - combination of long-term table, row, and index locks

Snapshot Isolation

- Not an ANSI standard isolation level, but used in some major DBMS
  - Oracle, SQL Server since version 2005, PostgreSQL since version 8
- **Multiversion database**: The old value of an item is not overwritten when it is updated. Instead, a new version is created on update.
  - DBMS can construct, for any *i*, the state of an item as a result of the execution of the first *i* transactions to commit
  - Snapshot: The database state produced by the execution of the first *i* transactions to commit

Snapshot Isolation (cont'd)

- Transaction execution:
  - No read locks necessary: a transaction reads all values from the latest snapshot at the time it started.
  - Updates create new versions of data items (subject to commit).
- A transaction can successfully commit if
  - it was read-only, or
  - none of its updated data items were concurrently updated too (**disjoint write sets** with concurrent transactions), or
  - transaction *T* that has updated *x* can commit if no other transaction that concurrrently updated *x* has committed yet
    - **First-committer-wins rule**
    - after *T* committed, the other (concurrent) transactions that updated *x* too will need to abort.

Discussion

- **Locking** is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- **Snapshot Isolation**
  - Good performance (‚readers never block')
  - Implementation complicated by need to maintain multiversion DB. Eventually old versions must be discarded (creates problems for long-running transactions).
  - *Write-skew* anomaly is possible (disjoint writes based on shared reads) which can lead (in rare cases) to non-serialisable executions...
    - Example: $r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ w_1(x)\ w_2(y)$
    - Avoiding this is possible (only PostgreSQL 9.1 so far), but adds overhead.

Learning Outcomes:

- **Understanding: Transaction Concept**
  - What is a ***transaction*** and why are transactions important?
  - ***ACID*** guarantees given by databases
  - How to define ***transactions in SQL***
- **Theory: Correctness of Transaction Programs**
  - ***Serializability*** criterion for correctness to avoid ***update anomalies***
- **Practice: Concurrency Control techniques in DBMS's**
  - Concurrency control and recovery are among the most important functions provided by a DBMS. - Users need not worry about this.
  - General idea of ***2 Phase Locking*** and ***Snapshot Isolation***
  - ***Deadlock*** problem; ***SQL Isolation Levels***

References

- Kifer/Bernstein/Lewis (2nd edition)
  - Chapter 18 + parts of Chapter 20
  - *Kifer/Bernstein/Lewis, especially the 'complete' version, has lots of coverage of transactions in several chapters. For the purpose of INFO2120, we keep on the overview Chapter 18 with parts of Chapter Section 20; the introductionary version of this book combines this in one Chapter 13*
- Ramakrishnan/Gehrke (3rd edition - the 'Cow' book)
  - Chapter 16 (*without the recovery part*)
  - *Stick to the overview chapter of transaction management in the Cow book; this book also contains two very technical chapters on all the implementation details for CC and recovery which we will cover in the 3rd year INFO3404*
- Ullman/Widom (3rd edition - 'First Course in Database Systems')
  - Chapter 6.6
    only gives a high-level introduction to the transaction commands of SQL
- Silberschatz/Korth/Sudarshan (5th edition - 'sailing boat')
  - Chapter 15

Next Week

- Indexing
- Schema Tuning
  - Denormalization and Partitioning
- SQL Query Tuning

- Readings:
  - Kifer/Bernstein/Lewis book, Chapter 12
  - or alternatively (if you prefer those books):
    - Ramakrishnan/Gehrke (Cow book), Chapter 8
    - Ullman/Widom, Chapter 8.3 onwards