COMP9120

Week 9: Transactions

Semester 2, 2016

(Ramakrishnan/Gehrke – Chapter 16

Kifer/Bernstein/Lewis – Chapter 18;

Ullman/Widom – Chapter 6.6)

Slides may contain some content from Kifer/Bernstein/Lewis (2006) "Database Systems" and from Ramakrishnan/Gehrke (2003) "Database Management Systems", and also including material from Dr. Uwe Röhm.

THE UNIVERSITY OF SYDNEY

Copyright warning

Key concepts

- Transaction
  - COMMIT, ROLLBACK
- ACID properties
  - Atomicity, Consistency, Isolation, Durability
- Deferrable constraints
- Update anomalies
  - dirty reads, unrepeatable reads, lost updates
- Execution schedules
  - Serializable schedules
  - conflict-serializable schedules
- Concurrency control
  - Locking protocols: Strict 2-phase locking
    - shared and exclusive locks, deadlocks
  - Versioning protocols: Snapshot Isolation
- SQL Isolation Levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE

THE UNIVERSITY OF SYDNEY

Initial Motivation

Transfer Operation

Withdraw $100
Deposit $100

Account A

Withdraw $100

Account B

Deposit $100
System fails

Account Balance successfully updated for Account A

System recovers but database state no longer reflects amount of physical money available (short $100)

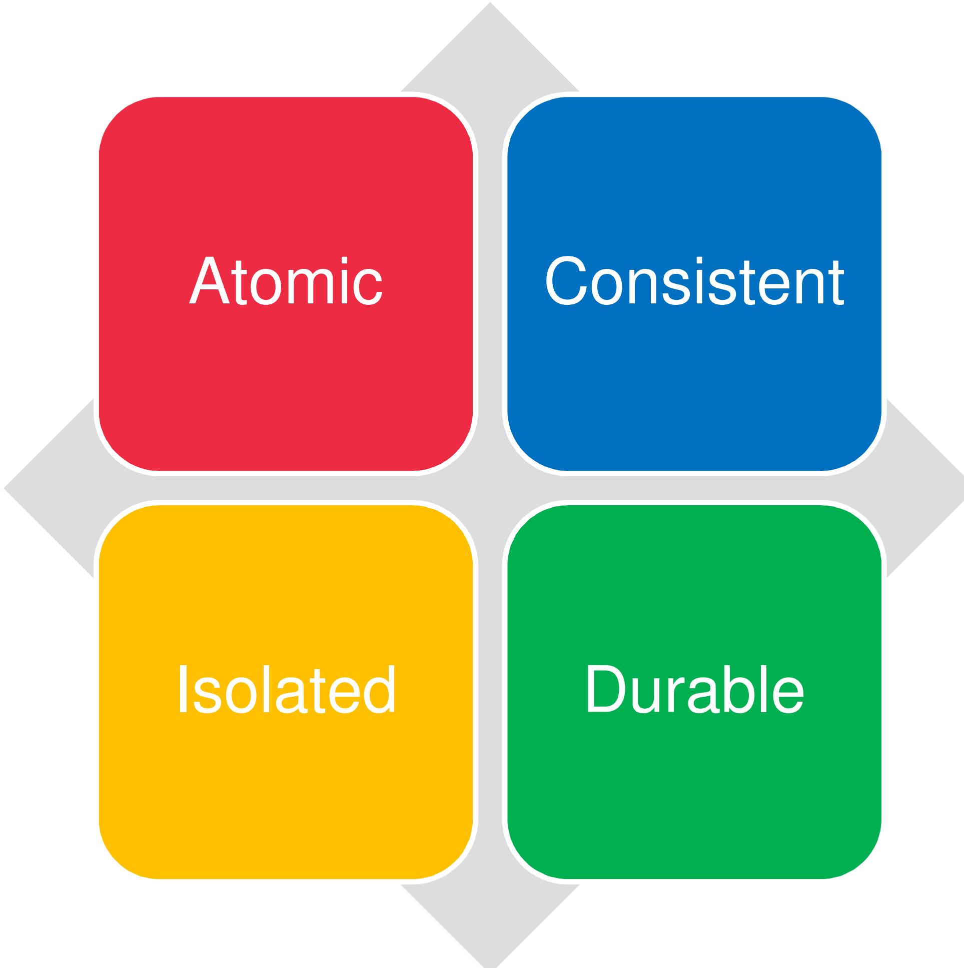
Should group withdraw & deposit operations together – so that they either both succeed or none happen at all

What are transactions?

- 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 on 2 accounts when a transfer is made
- Such a program is often modelled as a transaction:
  a collection of one or more operations on one or more databases, which reflects a discrete unit of work
  - Transactions should conform to certain requirements (***ACID properties***):
    Eg: In the real world, a transaction (completely) or it didn't happen at all

Required properties of a transaction

- The execution of each transaction must maintain relationship between the database state and the enterprise state.
- Therefore additional requirements are placed on the execution of transactions beyond those placed on ordinary programs.
  - ACID properties



Transactions should be Consistent (Consistency)

- Each transaction should preserve the consistency of the database.
- A **transaction is consistent** if, assuming the database is in a consistent state initially, when the transaction completes:
  - All database constraints are satisfied
    (but constraints might be violated in intermediate states)
  - New state satisfies specifications of transaction
- Note that this is mainly the responsibility of the application developer!
  - database cannot 'fix' the correctness of a badly coded transaction

Transactions should be Atomic (Atomicity)

- Every transaction should act as an atomic operation.
- A real-world event either happens or does not happen
  - Bank Transfer operation: either both withdrawal + deposit occur, or neither occurs.
- Similarly, the system must ensure that either the corresponding transaction runs to completion or, if not, it has no effect at all
  - a DBMS user can think of a transaction as always executing all its actions in one step, or not executing any actions at all.
    - 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, e.g.,
    - Transaction or connection hits time-out,
    - violation of constraint, etc
  - Transaction requests to roll back

Writing a transaction in SQL

- 3 new SQL commands to know:
  - **BEGIN TRANSACTION** (not required in Oracle)
  - **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.*

- Can also **SET AUTOCOMMIT OFF** or **SET AUTOCOMMIT ON**
  - With auto-commit on, each statement is its own transaction and 'auto-commits'
  - With auto-commit off, statements form part of a larger transaction delimited by the keywords discussed above.
  - different clients have different defaults for auto-commit.

What value should be returned?

<u>uosCode</u>	lecturerId
COMP5138	3456
COMP5338	4567

```
BEGIN;
UPDATE Course
SET lecturerId=1234
WHERE uosCode='COMP5138';
COMMIT;
SELECT lecturerId FROM Course
WHERE uosCode='COMP5138';
```

1. 1234
2. 3456
3. 4567

What value should be returned?

<u>**uosCode**</u>	**lecturerId**
COMP5138	3456
COMP5338	4567

```
BEGIN;
UPDATE Course
SET lecturerId=1234
WHERE uosCode='COMP5138';
ROLLBACK;
SELECT lecturerId FROM Course
WHERE uosCode='COMP5138';
```

1. 1234
2. 3456
3. 4567

Writing a transaction in JDBC

- APIs, like JDBC, often provide explicit functions for controlling the transaction semantics
  - (Otherwise you need to use explicit SQL commands)

- 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…

- 3 new methods to know (all for JDBC connection class):
  - **setAutoCommit(**false**)**
  - **commit()**
  - **rollback()**

JDBC Transaction example

```
public boolean bookFlight ( String flight_num, Date flight_date, Integer seat_no) {

  boolean booked = false;
    try {
      Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@oracle10…");
     conn.setAutoCommit(false); // next SQL statement will start a transaction

        /* Check whether there's a seat free for flight*/
      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 ) {

           /* reserve the seat – any issues here?? – see Isolation levels*/
           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();
            conn.commit();
           booked=true;
     } else { conn.rollback();    }
       /* close objects*/ …
    }
   catch (SQLException sqle) {
       /* error handling */ …
     } finally { … }
  return booked;
}
```

Note: In Oracle: should explicitly conn.rollback() if you want to rollback before calling conn.close()

When can a transaction be rolled back?

1. After COMMIT?

2. After an INSERT or UPDATE violates an IC?

3. After a database crashes?

Deferring Integrity Constraints

```
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
);
```

```
INSERT INTO UnitOfStudy VALUES('info1000','theTitle',42, 6);
INSERT INTO Lecturer VALUES(42,'Steve McQueen', …);
```

Transactions should be Durable (Durability)

- Once a transaction is committed, its effects should persist in a database, and these effects should be permanent even if the system crashes.

- A database should always be able to be recovered to the last consistent state

- Implementing Durability:
  - Database is stored redundantly on mass storage devices to protect against media failure (e.g., RAID)
  - **Write-Ahead Log**

What happens when we have more than one transaction running?

- Let's consider two transactions:
  - Transaction T1 is transferring $100 from account *A* to account *B*.
  - T2 credits both accounts with a 5% interest payment.

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

- Both transactions run concurrently
- At the database each operation must happen one after the other
  - Interleaved execution schedule

Execution Schedules

- 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:

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

- The DBMS's view of the second schedule:

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

Anomalies with Interleaved Execution

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

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

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

- **Unrepeatable Reads** (RW conflicts): may not read same value twice
  Phantom Reads: may not read same count of rows twice.

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

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

Transactions should be Isolated (Isolation)

- Transactions should be isolated from the effects of other concurrent transactions.
- Easiest implementation: **Serial Execution**
  - 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.
- **Concurrent execution** offers performance benefits:
  - A computer system has multiple resources capable of executing independently (e.g., CPUs, I/O devices), *but*
  - Must deal with concurrency anomalies saw on the previous slide

Isolation through 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.
- ***Schedule*** – sequence of operations that indicates the chronological order in which instructions of concurrent transactions are executed.
- ***Serial Schedule*** – A schedule in which all transactions run without interleaving, are executed from start to finish, one after the other.
- **Serializability**:
  A schedule is **serializable** if it is equivalent to a serial schedule

Conflict Serializability

- **Rule**: Two schedules are *conflict serializable* if:
  - 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, they come from different transactions, 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 Serializable Schedule Example 1

- **Notation**: *$r_1(x)$ means read by transaction 1 of object x*
  - also written as r1(x)

  $w_1(x)$ means write by transaction 1 on object x
  - also written as w1(x)

Eg: 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 *different* data items commute.
- Hence, *S* is a *serializable* schedule

Conflict 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-Conflict Serializable Schedule Example

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

$$
\begin{array}{llll}
T1: r(cur\_reg : 29) & & & w(cur\_reg : 30) \\
T2: & r(cur\_reg : 29) & w(cur\_reg : 30) &
\end{array}
$$

S: r1(c) r2(c) w2(c) w1(c)

T1, T2: r1(c) w1(c) r2(c) w2(c)
T2, T1: r2(c) w2(c) r1(c) w1(c)

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

Which of these schedules are conflict serializable?

- r1(x) r2(y) r1(z) r3(z) r2(x) r1(y)
  - all reads – no conflicts – hence serializable

- **r1(x)** <u>w2(y)</u> r1(z) r3(z) **w2(x)** <u>r1(y)</u>
  - non-serializable:

    -putting T1 before of T2 will make conflict w2(y) r1(y) violate conflict-serializability rule.

    - putting T2 before T1 will make conflict r1(x) w2(x) violate conflict-serializability rule

- **r1(x)** w2(y) r1(z) **r3(x) w2(x)** r2(y)
  - serializable: conflicts on x by T1/T3 and T2 can conform to conflict serializability rule

Isolation Levels

- DBMS' offers a variety of 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! (SET ISOLATION LEVEL ...)

ANSI Standard Isolation Levels

- Defined in terms of anomalies
- SET TRANSACTION ISOLATION LEVEL …
  - **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. *Phantom Reads Possible*
  - **Read committed** — only committed records can be read,
    Phantom + Unrepeatable Reads Possible.
    (*most common in practice!)*
  - **Read uncommitted** - even uncommitted records may be read
    Phantom + Unrepeatable + Dirty Reads Possible

Lock-based Concurrency Control

- **Strict Two-phase Locking (S2PL) 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 item before writing.
    - *exclusive (X) lock*: Data item can be accessed 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 completes.
  - If a transaction holds an *X* lock on an item, no other transaction can get a lock (*S* or *X*) on that item.
    - Similar if a transaction requests a *X* lock of an already locked item
    - Instead, such transactions must **wait** until the conflicting lock is released by the previous transaction(s)

Locking Example 1

- Consider:

```
T1: R(A),W(A),R(B),W(B),COMMIT
T2: R(A),W(A),R(B),W(B),COMMIT
```

- Complete a schedule starting as:

```
T1: S(A),R(A),X(A),W(A),S(B),R(B),X(B),W(B)
T2:                       (A locked, must wait)
```

- T2 must wait until T1 releases its lock on A (and B) – forces T2 to start only after T1 commits or aborts (rolls back)

Lock Compatibility Matrix and Lock Granularity

Held by T1 / T2 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

Locking Example 2

- Consider:

```
T1: R(A),W(A),R(B),W(B),COMMIT
T2: R(B),W(B),R(A),W(A),COMMIT
```

- Schedule with locking might start as:

```
T1: S(A),R(A),X(A),W(A)
T2:                     S(B),R(B),X(B),W(B)
```

- What happens next?
  - T1 waiting on T2 to release lock on A
  - T2 waiting on T1 to release lock on B
  - DEADLOCK

Deadlocks

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

Two ways of dealing with deadlocks: 17.4 [RG]

- 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

.

Thrashing

- Although a disadvantage of locking, deadlocks are only encountered by about 1% of transactions in practice. [RG16.5]
- A larger drawback of locking comes from the blocking that each transaction is faced with when they need to access an object.
- As number of clients (and transactions) increase, and there is more contention for locks, each transaction will have to wait longer to obtain a lock.
  - Eventually a critical point is reached, where adding another transaction to wait for the lock, would actually decrease the number of transactions served per unit time. This is when the system is said to experience "Thrashing". [RG 16.5]

What should happen with ACID transactions?

- Let's return to our two transactions:
  - Transaction T1 is transferring $100 from account *A* to account *B*.

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

- *Atomicity requirement* — all updates of a transaction 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, if both are submitted together. However, the actions of T1 should not affect those of T2, or vice-versa.
- *Durability requirement* — once a transaction has completed, the updates to the database by this transaction must persist despite failures

Key Skills

You should be able to:

- Explain how ACID properties define correct transaction behaviour
- Identify update anomalies when ACID properties aren't enforced
- Explain whether an execution schedule is serializable
- Use deferred integrity constraints in a transaction
- Implement appropriate transaction handling in client code (Java/JDBC)
- Explain how locking provides isolated transactions
- Select appropriate isolation level for a transaction

References

- [RG] Ramakrishnan /Gehrke – Chapter 16, gory details in Ch. 17 & 18
- Kifer/Bernstein/Lewis – Chapter 18
- Ullman/Widom – Chapter 6.6
- [JDBC] JDBC documentation
  - Docs for java.sql.connection (with commit, rollback and setAutoCommit) http://docs.oracle.com/javase/6/docs/api/java/sql/Connection.html
  - See also tutorial http://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html
- Oracle 12c Documentation
  - **Database Concepts**
    - Part III. Transaction Management http://docs.oracle.com/cd/E16655_01/server.121/e17633/part_txn.htm
  - **SQL Reference**: COMMIT http://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_4010.htm

Next Week (after study vac)

- Storage and Indexing
  - Storing data in a database
  - Retrieving records from a database
  - B+Tree and Hash indexes
- Kifer/Bernstein/Lewis
  - Chapter 9 (9.1-9.4)
- Ramakrishnan/Gehrke
  - Chapter 8
- Ullman/Widom
  - Chapter 8 (8.3 onwards)
- Silberschatz/Korth/Sudarshan (5th ed)
  - Chapter 11 and 12