Transaction and Concurrency Control

Elmasri/Navathe ch 21
Padron-McCarthy/Risch ch 23 and 24

DATABASE DESIGN I 192300 spring 2013
Selhan Badaroumari
Department of Information Technology, Upsala University

Why concurrency control is needed?

To address the following problems:

- The Lost Update Problem
- The Temporary Update (or Dirty Read) Problem
- The Incorrect Summary Problem
Transaction and Concurrency Control

Elmasri/Navathe ch 21
Padron-McCarthy/Risch ch 23 and 24

DATABASE DESIGN - 192300 spring 2013
Sahhak Badakzamery
Department of Information Technology, Upsala University

Why concurrency control is needed?

To address the following problems:

• The Lost Update Problem
• The Temporary Update (or Dirty Read) Problem
• The Incorrect Summary Problem
Multi-user System:

- Many users can access the system concurrently.
- Concurrency: several computations executing simultaneously.

Transaction:

- Logical unit of database program that includes one or more access operations.
- read - retrieval
- write - insert or update
- delete

Transaction boundaries:

- Begin and End transaction
Users can access the system concurrently.

Transaction: several computations executing simultaneously.

Transaction:
- Logical unit of database processing that includes one or more access operations:
  - read - retrieval
  - write - insert or update
  - delete

Transaction boundaries:
- Begin and End transaction
Read/write operations in transactions

Read_item(X):
  • Reads a database item named X into a program variable.

--> A copy is made during read.

Write_item(X):
  • Writes the value of program variable X into the database item named X.
Multi-user System:
- Many users can access the system simultaneously.
- Concurrency: several computations take place simultaneously.

Advantage:
- Transactions can run in parallel.
- [In many cases] no transaction has to wait.

Disadvantage:
- Consistency problems might appear if concurrent execution of transactions is uncontrolled.

But suddenly Alice's transaction fails in the middle, because of [e.g.] a network problem.

While Alice is transferring to Tom, John is sending her 50 SEK.

Alice has 300 SEK.
Alice wants to transfer 200 SEK to Tom.

If Alice and Bob don't make the transactions at the same time, e.g., Bob transfers first and Alice transfers next, then everything is OK.

Bob wants to transfer 100 SEK to Tom.

Tom needs some money urgently, his friends are sending him some.

Tom already has 1500 SEK --> we expect Tom to have 1800 SEK after the two transactions.
Tom needs some money urgently, his friends are sending him some.

Tom already has 1500 SEK --> we expect Tom to have 1800 SEK after the two transactions.

The auditing officer at bank wants to know the sum of money. So she writes an aggregate query that sums up all account balances.

Again, if she waits until all transfers are done, there will be no problem. BUT she can't wait forever! there are always ongoing transactions...

If Alice and Bob don't make the transactions at the same time, e.g. Bob transfers first and Alice transfers next, then everything is OK.

Bob wants to transfer 100 SEK to Tom.

Bob has 2000 SEK, so bob has 2000 - 100 = 1900 after the transfer.

Alice has 300 SEK.

Alice wants to transfer 200 SEK to Tom.

At this point, Alice has 300 - 200 = 100 SEK left.

The temporary update problem.

Again, in concurrent execution of transactions, any order of interleaving is possible.
Alice wants to transfer 200 SEK to Tom.

If Alice and Bob don't make the transactions at the same time, e.g. Bob transfers first and Alice transfers next, then everything is OK.

Concurrent execution is uncontrolled:
(a) The lost update problem.

But in concurrent execution of transactions, any order of interleaving is possible.

Here is one possible interleaving of operations in the two transactions:

<table>
<thead>
<tr>
<th>Alice’s transaction</th>
<th>Bob’s transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Concurrent execution is uncontrolled: (a) The lost update problem.

But in concurrent execution of transactions, any order of interleaving is possible.

Here is one possible interleaving of operations in the two transactions:

<table>
<thead>
<tr>
<th>Alice’s transaction</th>
<th>Bob’s transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Tom’s balance into t_balance&lt;br&gt;Read t_balance;</td>
<td>Read Tom’s balance into t_balance&lt;br&gt;Read t_balance;</td>
</tr>
<tr>
<td>Add 200 to Tom’s balance&lt;br&gt;t_balance= t_balance + 200;</td>
<td>Add 100 to Tom’s balance&lt;br&gt;t_balance= t_balance + 100;</td>
</tr>
<tr>
<td>Write Tom’s balance back to the database&lt;br&gt;Write t_balance;</td>
<td>Write Tom’s balance back to the database&lt;br&gt;Write t_balance;</td>
</tr>
</tbody>
</table>

Instead of 1800, Tom will see 1600 in his account after the these transactions. Alice's update is lost/over-written in this interleaving.

The lost update problem.
But in concurrent execution of transactions, any order of interleaving is possible.

Here is one possible interleaving of operations in the two transactions:

<table>
<thead>
<tr>
<th>Alice’s transaction</th>
<th>Bob’s transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_balance is 1500</td>
<td>t_balance is 1500</td>
</tr>
<tr>
<td>t_balance is 1700</td>
<td>t_balance is 1600</td>
</tr>
<tr>
<td>1700 is written</td>
<td>1600 is over-written</td>
</tr>
<tr>
<td>Read Tom’s balance into t_balance</td>
<td>Read Tom’s balance into t_balance</td>
</tr>
<tr>
<td>Read \texttt{t_balance};</td>
<td>Read \texttt{t_balance};</td>
</tr>
<tr>
<td>Add 200 to Tom’s balance</td>
<td>Add 100 to Tom’s balance</td>
</tr>
<tr>
<td>\texttt{t_balance} = \texttt{t_balance} + 200;</td>
<td>\texttt{t_balance} = \texttt{t_balance} + 100;</td>
</tr>
<tr>
<td>Write Tom’s balance back to the database</td>
<td>Write Tom’s balance back to the database</td>
</tr>
<tr>
<td>Write \texttt{t_balance};</td>
<td>Write \texttt{t_balance};</td>
</tr>
</tbody>
</table>

Instead of 1800, Tom will see 1600 in his account after the these transactions. Alice's update is lost/over-written in this interleaving.

The lost update problem.
But suddenly Alice's transaction fails in the middle, because of [e.g.] a network problem.

While Alice is transferring to Tom, John is sending her 50 SEK.

Tom needs some money urgently, his friends are sending him some.

Alice has 300 SEK.

Alice wants to transfer 200 SEK to Tom.

If Alice and Bob don't make the transactions at the same time, e.g. Bob transfers first and Alice transfers next, then everything is OK.
Concurrent execution is uncontrolled:
(b) The temporary update problem.

Again, in concurrent execution of transactions, any order of interleaving is possible.

Alice is going to send money to Tom, so the program should first reduce the money from her account.

Here is one possible interleaving of operations in the two transactions:

<table>
<thead>
<tr>
<th>Alice's transaction</th>
<th>John's transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_balance is 300</td>
<td>a_balance is 100</td>
</tr>
<tr>
<td>Read a_balance</td>
<td>Read a_balance</td>
</tr>
<tr>
<td>Deduct 200 from</td>
<td>Add 50 to Alice's</td>
</tr>
<tr>
<td>a_balance = a_balance - 200;</td>
<td>balance</td>
</tr>
<tr>
<td>Write Alice's balance back to the database</td>
<td>Write Alice's balance back to the database</td>
</tr>
<tr>
<td></td>
<td>a_balance = a_balance + 50;</td>
</tr>
<tr>
<td></td>
<td>Write a_balance;</td>
</tr>
</tbody>
</table>

Suddenly a network problem happens for Alice's computer, so the rest of the steps never happen.

Transfer 200 to Tom (multiple steps)

A. transaction fail and must role back, i.e. change a_balance back to 300. meanwhile John's transaction has read temporary incorrect value of a_balance.

The temporary update problem.
### Alice's transaction

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read Alice’s balance into a_balance</td>
</tr>
<tr>
<td>2</td>
<td>Deduct 200 from Alice’s balance</td>
</tr>
<tr>
<td>3</td>
<td>Write Alice’s balance back to the database</td>
</tr>
</tbody>
</table>

### John's transaction

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read Alice’s balance into a_balance</td>
</tr>
<tr>
<td>2</td>
<td>Add 50 to Alice’s balance</td>
</tr>
<tr>
<td>3</td>
<td>Write Alice’s balance back to the database</td>
</tr>
</tbody>
</table>

Suddenly a network problem happens for Alice's computer, so the rest of the steps never happen.

Transfer 200 to Tom (multiple steps)

---

Alice's transaction fail and must role back, i.e. change a_balance back to 300. meanwhile John's transaction has read temporary incorrect value of a_balance.

**The temporary update problem.**
The auditing officer at bank wants to know the sum of money. So she writes an aggregate query that sums up all account balances.

Again, if she waits until all transfers are done, there will be no problem. BUT she can't wait forever! there are always ongoing transactions...
Concurrent execution is uncontrolled:

(c) The incorrect summary problem.

Alice is going to send money to Tom, so the program should first reduce the money from her account. Here is one possible interleaving of operations in the two transactions:

<table>
<thead>
<tr>
<th>Alice’s transaction</th>
<th>Bank officer’s transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Alice’s balance into a_balance</td>
<td>Initialize sum</td>
</tr>
<tr>
<td>Read a_balance;</td>
<td>sum = 0;</td>
</tr>
<tr>
<td>Deduct 200 from Alice’s balance</td>
<td>Read a_balance;</td>
</tr>
<tr>
<td>a_balance = a_balance - 200;</td>
<td>Add Alice’s balance to sum</td>
</tr>
<tr>
<td>Write Alice’s balance back to the database</td>
<td>sum = sum + a_balance;</td>
</tr>
<tr>
<td>Write a_balance;</td>
<td>Read Tom’s balance into t_balance</td>
</tr>
<tr>
<td></td>
<td>Read t_balance;</td>
</tr>
<tr>
<td>Read Tom’s balance into t_balance</td>
<td>Add Alice’s balance to sum</td>
</tr>
<tr>
<td>Read t_balance;</td>
<td>sum = sum + a_balance;</td>
</tr>
<tr>
<td>Add 200 to Tom’s balance</td>
<td>Read Tom’s balance into t_balance</td>
</tr>
<tr>
<td>t_balance = t_balance + 200;</td>
<td>Add Tom’s balance to sum</td>
</tr>
<tr>
<td>Write Alice’s balance back to the database</td>
<td>sum = sum + t_balance;</td>
</tr>
<tr>
<td>Write t_balance;</td>
<td></td>
</tr>
</tbody>
</table>

The sum transaction reads Alice's balance after 200 is subtracted, and reads Tom's balance before 200 is added. Summary is of by 200.
<table>
<thead>
<tr>
<th>Alice’s transaction</th>
<th>Bank officer’s transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Alice’s balance into a_balance</td>
<td>Initialize sum</td>
</tr>
<tr>
<td>Read a_balance;</td>
<td>sum = 0;</td>
</tr>
<tr>
<td>Deduct 200 from Alice’s balance a_balance = a_balance - 200;</td>
<td></td>
</tr>
<tr>
<td>Write Alice’s balance back to the database Write a_balance;</td>
<td>Read Alice’s balance into a_balance Read a_balance;</td>
</tr>
<tr>
<td></td>
<td>Add Alice’s balance to sum sum = sum + a_balance;</td>
</tr>
<tr>
<td></td>
<td>Read Tom’s balance into t_balance Read t_balance;</td>
</tr>
<tr>
<td></td>
<td>Add Tom’s balance to sum sum = sum + t_balance;</td>
</tr>
</tbody>
</table>

The sum transaction reads Alice's balance after 200 is subtracted, and reads Tom's balance before 200 is added. Summary is off by 200.

The incorrect summary problem
Why concurrency control is needed?

To address the following problems:

- The Lost Update Problem
- The Temporary Update (or Dirty Read) Problem
- The Incorrect Summary Problem
Transaction states

**Figure 21.4**
State transition diagram illustrating the states for transaction execution.

**Active**: The initial state; the transaction stays in this state while it is being executed.

**Partially committed**: When transaction ends, after the final statement has been executed, it goes into the partially committed state.

**Committed**: After successful completion.

**Failed**: After the discovery that normal execution can no longer proceed or if it has been aborted in its active state. Rollback might be necessary.

**Abort**: When a transaction is terminated.

**Terminated**: Corresponds to the transaction leaving the system.

---

1. **A computer failure (system crash)**
   - A hardware or software error occurs in the computer system during transaction execution.
   - The hardware crashes, or the contents of the computer’s internal memory may be lost.

2. **A transaction or system error**
   - Some operation in the transaction such as integer overflow or division by zero.
   - Errors due to parameter values or compiler execution.
   - The user may interrupt the transaction during its execution.

3. **Local errors or exception conditions detected by the transaction**
   - Examples:
Active: The initial state; the transaction stays in this state while it is being executed.
Partially committed:
When transaction ends, after the final statement has been executed, it goes into the partially committed state.
Committed:
After successful completion.
Failed:
After the discovery that normal execution can no longer proceed or if it has been aborted in its active state. Rollback might be necessary.
Terminated:
Corresponds to the transaction leaving the system.
What does cause a transaction to fail?

1. A computer failure (system crash)
   • A hardware or software error occurs in the computer system during transaction execution.
   • If the hardware crashes, the contents of the computer's internal memory may be lost.

2. A transaction or system error
   • Some operation in the transaction such as integer overflow or division by zero.
   • Erroneous parameter values or because of a logical programming error.
   • The user may interrupt the transaction during its execution.

3. Local errors or exception conditions detected by the transaction

   Examples:
   • Data for the transaction may not be found.
   • A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

4. Concurrency control enforcement

   The concurrency control method may decide to abort the transaction, to be restarted later, because it violates consistency or because several transactions are in a state of deadlock (see Chapter 22).

5. Disk failure

   Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes

   This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.
1. A computer failure (system crash)
   - A hardware or software error occurs in the computer system during transaction execution.
   - If the hardware crashes, the contents of the computer’s internal memory may be lost.

2. A transaction or system error
   - Some operation in the transaction such as integer overflow or division by zero.
   - Erroneous parameter values or because of a logical programming error.
   - The user may interrupt the transaction during its execution.

3. Local errors or exception conditions detected by the transaction
   
   Examples:
   - Data for the transaction may not be found.
   - A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

4. Concurrency control enforcement
4. Concurrency control enforcement

The concurrency control method may decide to abort the transaction, to be restarted later, because it violates consistency or because several transactions are in a state of deadlock (see Chapter 22).

5. Disk failure

Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes

This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.
4 Desirable Properties of Transactions

ACID

Atomicity Consistency Isolation Durability

If all 4 properties are hold, transactions can safely run concurrently.

The approach
- All operations are executed in a single unit of work (atomicity)
- Transactions are isolated from each other (isolation)
- Durability ensures that transactions are committed or rolled back (durability)
- If a transaction is interrupted, it is restarted (consistency)
- Improved performance and utilization of resources
Atomicity (atomic or indivisible):
- A transaction, including all operations in it, is a logical processing unit.
- It should be is carried out in its whole or not at all.
Example:
a transaction to transfer 400 SEK from Erik to Jack.

1. read(Erik's_balance);

2. Erik's_balance := Erik's_balance - 400;

3. write(Erik's_balance);

4. read(Jack's_balance);

5. Jack's_balance := Jack's_balance + 400;

6. write(Jack's_balance);

Atomicity requirement:
if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
Consistency:
A correct execution of the transaction must take the database from one consistent state to another.

Example:
a transaction to transfer 400 SEK from Erik to Jack.

1. read(Erik's_balance);
2. Erik's_balance := Erik's_balance - 400;
3. write(Erik's_balance);
4. read(Jack's_balance);
5. Jack's_balance := Jack's_balance + 400;
6. write(Jack's_balance);
Example:
a transaction to transfer 400 SEK from Erik to Jack.

1. read(Erik's_balance);
2. Erik's_balance := Erik's_balance - 400;
3. write(Erik's_balance);
4. read(Jack's_balance);
5. Jack's_balance := Jack's_balance + 400;
6. write(Jack's_balance);

Consistency requirement:
the sum of Erik's_balance and Jack's_balance is unchanged by the execution of the transaction.
Isolation:
A transaction should not make its updates visible to other transactions until it is committed.

Example:
a transaction to transfer 400 SEK from Erik to Jack.

1. read(Erik's_balance);
2. Erik's_balance := Erik's_balance - 400;
3. write(Erik's_balance);
Example:
a transaction to transfer 400 SEK from Erik to Jack.

1. read(Erik's_balance);
2. Erik's_balance := Erik's_balance - 400;
3. write(Erik's_balance);
4. read(Jack's_balance);
5. Jack's_balance := Jack's_balance + 400;
6. write(Jack's_balance);

Isolation requirement:
Between steps 3 and 6, Other transactions should not be able to see Erik's balance and/or Jack's balance, because it is an inconsistent database state.
(Jack's balance + Erik's Balance will be 400 less than it should be)
Durability:
Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Example:
a transaction to transfer 400 SEK from Erik to Jack.

1. read(Erik's_balance);

2. Erik's_balance := Erik's_balance - 400;
Example:
a transaction to transfer 400 SEK from Erik to Jack.

1. read(Erik's_balance);

2. Erik's_balance := Erik's_balance - 400;

3. write(Erik's_balance);

4. read(Jack's_balance);

5. Jack's_balance := Jack's_balance + 400;

6. write(Jack's_balance);

Durability requirement:
Once the user has been notified that the transaction has completed (ie. the transfer of the 400 SEK has taken place), the updates to the database by the transaction must persist despite later failures.
4 Desirable Properties of Transactions

ACID

Atomicity
Consistency
Isolation
Durability

If all 4 properties are hold, transactions can safely run concurrently.

The approach

- All requirements can be excecuted orderly by nesting transactions serially, that is, one after the other.
- No set of actions should be allowed to be undone by multiple transactions occurring concurrently.
- Because concurrent execution.
- Increase processor and disk utilization, leading to better transaction throughput.
- New transactions are loading the database, another is reading those or writing to the disk.
- Conflicts are resolved using a concurrency control protocol.
- Short transaction nested well last ending.
-
The approach

• ACID requirement can be ensured trivially by running transactions serially, that is, one after the other.
• But, we would like to accomplish the same benefits for multiple transactions executing concurrently.

Because concurrent execution:

– Increases processor and disk utilization, leading to better transaction throughput:
  • one transaction can be using the CPU while another is reading from or writing to the disk.
– Reduces average response time for transactions:
  • short transactions need not wait behind long ones.
Concurrency control

Transaction schedule
- A transaction schedule is a set of ordered operations involving the execution of a set of transactions.
- How do we determine the order of transactions?

Serializable schedule
- A transaction schedule for a set of transactions is said to be serializable if it has the effect of a serial transaction schedule, i.e., it can be executed in serial without violating the ACID properties.
- How do we determine if a schedule is serializable?

Controlling serializability of schedules
- Tracking schedule serializability is crucial for ensuring data consistency.
- What factors influence the serializability of a schedule?

Recovery
- Recovery is needed after aborted transactions.
- The goal is to restore the database to an earlier consistent state.

concurrency control approaches

Pessimistic concurrency control
- Based on the pessimistic assumption that concurrency produces conflicting side effects, pessimistic concurrency control locks shared data and enforces serializability.

Optimistic concurrency control
- Based on the optimistic assumption that concurrency is safe and does not lead to conflicts, optimistic concurrency control allows multiple transactions to execute concurrently and detects conflicts only after the execution is complete.
Transaction schedule

There exist a number of different execution orders that can be scheduled for the operations in a set of transactions.
  - But which of these execution orders are acceptable?

A transaction schedule for a set of transactions:

describes in what order the operations (Read Write etc.) in the transactions should be performed.

serial schedule

A transaction schedule where the operations for each transaction uninterruptedly follow each other.

T1: read(x); write(y); read(z);
T2: read(u); write(v); read(w);
A serial schedule S:

S: read(x); write(y); read(z); read(u); write(v); read(w);
Serializable schedule

A transaction schedule for a number of transactions is said to be serializable,

if its effect is “equivalent”

to the effect of a serial transaction schedule incorporating the same transactions.

To be able to judge if a transaction schedule is correct we must prove that the schedule is serializable.
Controlling serializability of schedules

- Testing a schedule for serializability after it has executed is a little too late!

Goal:
- To develop concurrency control protocols that will assure serializability;
  - Instead of analyzing if a schedule is serializable, we will instead impose a protocol that avoids nonserializable schedules.
Recovery

- Recovery is needed after aborted transactions.
- The goal is to restore the database to an earlier and consistent state.
concurrency control approaches

**Pessimistic concurrency control**
Based on the pessimistic assumption that concurrency problems will happen, these methods try to prevent them.

**Optimistic concurrency control**
- Let transactions execute concurrently and control if they have interacted in a non-serializable manner when they are to be finished.
- If conflicts occur, one of the transactions can be aborted.
Pessimistic concurrency control

Based on the pessimistic assumption that concurrency problems will happen, these methods try to prevent them.

**Locking**

- Locking is one of the main mechanisms to handle concurrent transactions (based on a pessimistic assumption that conflicts will appear).
- A lock is the access right for an item and a program, the lock manager decides which transactions should be granted the access right for an item.
- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules. Only one transaction is allowed to lock an item at a certain point of time.

**Locking example**

**Without locks**

- T1: Read A; A = A + 1; Write A
- T2: Read A; A = A + 1; Write A

**What can go wrong here?**
Risk of lost updates

**The solution is to introduce locks**

- T1: Lock A; Read A; A = A + 1; Write A; Unlock A
- T2: Lock A; Read A; A = A + 1; Write A; Unlock A
Locking

- Locking is one of the main mechanisms to handle concurrent transactions (is based on a pessimistic assumption that conflicts will will appear)

- A lock is the access right for an item and a program, the lock manager, decides which transaction should be granted the access right for an item.

- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Only one transaction is allowed to lock an item at a certain point of time.
## Locking example

### Without locks

**T1**
- Read A;
- A := A + 1;
- Write A;

**T2**
- Read A;
- A := A + 1;
- Write A;

What can go wrong here?
Risk of lost updates

### The solution is to introduce locks

Main concept:
- Only one transaction can access the shared variable at any given time.

**T1**
- Lock A;
- T1 gets the lock first.
- Read A;
- A := A + 1;
- Write A;
- Unlock A;

**T2**
- Lock A;
- T2 should wait here:
- Read A;
- A := A + 1;
- Write A;
- T2 can resume only after T1
  - Releases the lock on A
- Unlock A;
Without locks

T1
Read A;
A:=A+1;
Write A

T2
Read A;
A:=A+1;
Write A

What can go wrong here?
Risk of lost updates
The solution is to introduce locks

Main concept:
- only one transaction can access the shared variable at any given time.

T1 gets the lock first.

T2 should wait here:

T2 can resume only after T1 releases the lock on A.
Optimistic concurrency control

- Let transactions execute concurrently and control if they have interacted in a non-serializable manner when they are to be finished.
- If conflicts occur, one of the transactions can be aborted.
in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.

- In SQL, a transaction begins implicitly.
  - however some DBMS's support explicit begin/end transaction
- A transaction in SQL ends by:
  - Commit: commits current transaction and begins a new one.
  - Rollback: causes current transaction to abort.

Levels of consistency specified by SQL-92:
- Serializable — default \(\rightarrow\) Very safe, one does not have to worry at all.
- Repeatable read
- Read committed
- Read uncommitted
in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.

- In SQL, a transaction begins implicitly.
  - however some DBMS's support explicit begin/end transaction

- A transaction in SQL ends by:
  - Commit: commits current transaction and begins a new one.
  - Rollback: causes current transaction to abort.
Levels of consistency specified by SQL-92:
- Serializable — default  --> Very safe, one does not have to worry at all.
- Repeatable read
- Read committed
- Read uncommitted
Transaction and Concurrency Control

Elmasri/Navathe ch 21
Padron-McCarthy/Risch ch 23 and 24

DATABASE INRTN T - 192300 spring 2013
Saktan Bedi/currancy
Department of Information Technology, Upsala University

Why concurrency control is needed?
To address the following problems:

- The Lost Update Problem
- The Temporary Update (or Dirty Read) Problem
- The Incorrect Summary Problem