Concurrent and inconsistency

Once we have these databases all setup, we hope people will want to use them. We begin to run into problems when a lot of them want to use the database at the same time.

And it might not be people at all. Multiple continuously running applications may be simultaneously maintaining connections with the database.

As the value of the information in the database rises, so does the number of users and their dependence (or expectation) of its availability. Thus, in addition to conflicts among simultaneous access by clients and applications, you also encounter system needs such as replication, backup/restore, and maintenance.

As we saw earlier in this and the term, database consistency is very important in most applications.

Recall the example of the ATM machine and to bank customers trying to withdraw from the same account at nearly the same time. That was an example of what Widom calls attribute-level inconsistency.

```sql
/* Client 1 - Deposit */
UPDATE CheckingAccount
SET Balance = Balance+100 WHERE AccountNumber=123456;
--------
/* Client 2 - Deposit */
UPDATE CheckingAccount
SET Balance = Balance+150 WHERE AccountNumber=123456;
```

these two clients initiate their transactions almost simultaneously, the steps of the two transactions can actually occur in different sequences.

* Client 1 might complete first, and then Client 2 would complete, and everything's as expected.
* Alternatively, the steps might interleave:
  1. Client 1 might run first by reading the Balance,
  2. Client 2 runs and reads the Balance.
3. Then Client 1 runs again, does the +100, and stores Balance.
4. Client 2 runs again, does the +150, and stores Balance.
Thus there are three different possible values for the Balance after these steps.

There are other levels of inconsistency: **Relation-level inconsistency**

```sql
/* Client 1 - Give High Cost Area increases to top performers */
UPDATE Employees
SET Hours = Hours * 1.25
WHERE Rating>90;

/* Client 2 - Tri-State hourly rate Increases for full-time workers */
UPDATE Paychecks
SET Rate = Rate * 1.05
WHERE (State=NY OR State=CT OR State=NJ) AND Empid IN
  (SELECT Empid FROM Employees WHERE Hours>=40);
```

some employees would get both increases when they only deserve one.

Finally, there is **Multiple-Statement inconsistency**

```sql
/* Client 1 - promote students based on hours earned */
INSERT INTO Seniors (SELECT * FROM Juniors WHERE Hours > 90);
DELETE FROM Juniors WHERE Hours > 90;
...

/* Client 2 - Calculate class sizes */
SELECT COUNT(*) FROM Seniors;
SELECT COUNT(*) FROM Juniors;
```

's unclear which students will be counted where, or even whether they will be counted once!

As we just demonstrated, half-completed or non-sequential transactions can wreak havoc with a database’s consistency and integrity.

So is the answer to force all transactions to be serialized (isolated)?
* That sort of defeats the purpose of a large database serving many users.
* We want concurrency so we can have the highest possible performance

What about multi-threaded or multi-core solutions which could potentially runn against the same database? We don’t want to serialize these!

**What about system failures**

Stuff breaks ... networks, computers, disks, ... it happens.
Sometimes someONE breaks stuff: disgruntled employee, hacker, terrorist, ...
We also want the database to have as good a chance of surviving such attacks as possible.

* So many stock changes happening so fast after XYZ corporation announced their plans to acquire WVU Industries that the database server crashed.

* The power failed just as the archive of the day’s retail sales transactions process began.

* The system might crash in the middle of our update to promote Juniors to Seniors, missing the DELETE to remove the promoted students from Juniors/

* The disgruntled employee logged into the database server as root and ran the command `rm -rf /`

Transactions to the rescue

A **Transaction** is a Logical unit of work that must be entirely completed or completely aborted.

Our goal is the ability to execute Transactions (SELECT, UPDATE, INSERT, DELETE, etc.) while maintaining our carefully designed and managed DB in its consistent state with all data-integrity constraints satisfied.

In general SQL, a Transaction begins automatically as the first SQL statement is run. A **COMMIT** ends the Transaction and may begin a new one.

- Think of it as a timeline: … COMMIT | SELECT | UPDATE | DELETE | COMMIT …

Transaction works because it tells the DB that it needs some peace and quiet - some uninterrupted access. This is vital to the concept. Just make sure that you never start a transaction and then wait for some unpredictable amount of time (synchronization with some other process, external state input from a sensor like thermometer, user input, etc.). It would tie up the DB !!!

So this example from Widom would be bad:

```sql
BEGIN TRANSACTION;
  -- get input from something or someone
  Do some SQL commands using that input;
  -- Confirm the results with something or someone
  IF (OK?) THEN Commit; ELSE Undo;

MySQL does **AutoCommit** by default, which means anytime a table is changed it is written to disk. You can turn this off: `SET AUTOCOMMIT = 0;`

MySQL 5.6+ has **START TRANSACTION READ ONLY** which promises that the actions taken by this script do not modify the database - they are read only.

* helpful for concurrency

What do transactions need?

**Atomicity**

All operations of a transaction must be completed

If not, the transaction is aborted

If things don’t go well with the Transaction, we want it to abort and clean up everything it changed.
In support of Atomicity, the system maintains a transaction log.
* Keeps track of all transactions that update the database
* DBMS uses the information stored in a log for:
  * Recovery requirement triggered by a ROLLBACK statement
  * A program’s abnormal termination
  * A system failure

In SQL, we use `ROLLBACK` to clean up *almost* everything that changed.
We say almost because `ROLLBACK` doesn’t cleanup:
* SQL variables that might have been changed,
  * out-of-band actions such as
    * dropping the soda can into the delivery area
    * creation/update of a file or a different database.

**Consistency**

Permanence of database’s consistent state.

DB designers spend a lot of time setting up integrity constraints to insure the DB is always in a valid state.

The Consistency goal means that any transaction can assume that the DB is in a consistent (valid) state when it starts, and the system will ensure that the DB is in a consistent state when the transaction finishes.

**Isolation**

Data used during transaction cannot be used by second transaction until the first is completed

We want each client/application to think it’s running fully isolated from all others, but without really doing it (i.e., running them serially)

We strive for isolation by by analyzing the potentially concurrent actions to determine an interleaving that has the same results as if the actions were run serially.

**Durability**

Ensures that once transactions are committed, they cannot be undone or lost.

They will persist across transaction execution, application executions, and even system crashes.

**Serializability**

Ensures that the schedule for the concurrent execution of several transactions should yield consistent results.

We can see that the examples we did earlier could be remedied via serializability.

Serializability means that the actions of transactions may be interleaved, but the end result must be the same as if the transactions were run in some sequential order. This can bring DB overhead and can reduce concurrency.

A common approach to implementing serialization is via locks within the database to prevent two transactions (or two functions within two transactions) from accessing the same data elements at the same time.
For example, when buying tickets online for a Revolution game, once you choose the seats you want there is a little countdown window that appears, showing how long you have to complete the transaction. This is a means of serialization on the Seats resource. Note that it is a lock only on the seats you have begun to buy, not all the seats.

This example shows how seat selection was allowed to continue, due to the support for transactions and serialization, resulting in the appearance of concurrency.

**Isolation levels**

If a transaction has written some data to the DB but not yet committed it, that data is *Dirty Data*.

If then another transaction reads that data, that is called a *Dirty Read*.

You can imagine the issues here.

Since serializable has costs, SQL lets us specify when to use weaker isolation levels that modify how READs are done. These specifications are Transaction based, implying they only affect the transaction they're part of.

1. **READ UNCOMMITTED** - Dirty Reads are ok; In general, SQL assumes transactions are READ WRITE, except when you allow Dirty Reads. Since that's so risky, SQL assume the transaction is READ ONLY unless you specifically override it.
2. **READ COMMITTED** - Forbids reading Dirty Data
3. **REPEATABLE READ** - Implies that repeated reads of the same tuple will have identical results. Reasonable, except that the query might return *phantom* tuples due to some other updates to the database.

Perhaps the Revolution Ticket purchase scheme uses READ COMMITED:

1. Transactions with READ COMMITTED begins
2. Customer chooses seats
3. Transaction commits the selection with the PENDING value set TRUE.
4. A trigger fires because of the COMMIT and PENDING=TRUE that sets a timer to run while updating the little “Time left” window.
5. Other transactions cannot see those seats because of the COMMIT.
6. If the timer completes, another TRIGGER is initiated which reverses the transaction and COMMITS.

So in summary (From Garcia-Molina):

---

Real Incident: *in the middle of a transaction, after* selecting seats, my WiFi router pooped out. I quickly rest
<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Reads</th>
<th>NonRepeatable Reads</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>NOT OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>NOT OK</td>
<td>NOT OK</td>
<td>OK</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>NOT OK</td>
<td>NOT OK</td>
<td>NOT OK</td>
</tr>
</tbody>
</table>

1. Lecture notes based on texts by Coronel, Widom, Ullman, Jukic, and Silberschatz.