TRANSACTIONS FOR CONCURRENCY CONTROL (ISOLATION)
[based on excellently short lecture notes of Prof. Jeff Ullman]

MOTIVATION:
* Database systems are normally being accessed by many users or processes at the same time. Both queries and modifications.

**Concurrency**
* Unlike operating systems, which support interaction of processes, a DBMS needs to keep processes from troublesome interactions.

>> EXAMPLE of bad interaction:
You and your domestic partner each take $100 from different ATM’s at about the same time. The DBMS better make sure one account deduction doesn’t get lost.

Contrast: An OS allows two people to edit a document at the same time. If both save, someone’s changes get lost.

>>TRANSACTIONS (abbreviation Xacts)<<
Avoid such problems, by providing, among others, for ISOLATION: the appearance that a group of SQL statements are executed as an uninterrupted whole.

>>SQL TRANSACTIONS
SQL supports Xacts, often behind the scenes
* Each statement issued at the generic query interface is an Xact by itself.

* In programming interfaces like Embedded SQL, a transaction begins the first time a SQL statement is executed and ends with the program or an explicit transaction-end: COMMIT/ROLLBACK

<> SQL COMMIT indicates that the transaction has completed so the modifications can be made permanent.
<> SQL ROLLBACK terminates the Xact by aborting it and leaving no traces in the database.

>>EXAMPLE
Serves(bar,beer,price) relation
(Sam,Bud,2.50)
(Sam,Miller,3.00)

Dianne -- query: Price for the highest and lowest price Sam charges.
SELECT MAX(price) FROM Serves; [query (max)]
SELECT MIN(price) FROM Serves; [query (min)]

Dianne -- decides to stop serving Bud and Miller, but to sell only Heineken at $3.50
DELETE FROM Serves WHERE bar='Sams'; [del]
INSERT INTO Serves VALUES (Sams,heineken,3.50); [ins]

Sam -- decides to stop serving Bud and Miller, but to sell only Heineken at $3.50
DELETE FROM Serves WHERE bar='Sams'; [del]
INSERT INTO Serves VALUES (Sams,heineken,3.50); [ins]

Concurrent execution allows any shuffling of Dianne [[(max),(min)]] and Sam [[(del),(ins)]]

PROBLEM: consider the trace
D: (max) (del) (ins)
S: (max) (del) (ins)

PRICE column
2.50
3.00

(max) compute 3
(del) <empty>
(ins) 3.50
(min) compute 3.5

unpleasant consequence: Dianne sees MAX=3.00 < MIN=3.50

SOLUTION: group Dianne’s statements into an Xact;
they are executed together and so they see Sam’s prices either
- before the first modification // ^ (del)
- between the first and second modification // (del) ^ (ins)
- after the second modification // (del)(ins) ^

But in all cases, she sees a *consistent* set of values.
(What are the possible answers seen by Dianne?)

>> ANOTHER PROBLEM <<

Suppose Sam executes (del) (ins), not in a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement.

If Dianne executes her statements after (ins) but before the ROLLBACK, she sees a value, 3.50, that never existed in the database.

SOLUTION: have Sam execute (ins) (del) as an Xact; then its effects cannot be seen until the Xact issues a COMMIT; so executing ROLLBACK means its effects can never be seen by others.

>> SERIALIZABLE TRANSACTIONS <<
If Dianne = (max)(min) and Sam = (del)(ins) are each transactions, and Dianne runs with isolation level SERIALIZABLE, then she will see the database either before or after Sam runs, but not in the middle.

It’s up to the DBMS vendor to figure out how to do that, e.g.:
- True isolation in time.
- Keep Sam’s old prices around to answer Dianne’s queries.

DEFINITION: AN INTERLEAVED EXECUTION OF T1 and T2 IS *SERIALIZABLE*
IF ITS EFFECT IS EQUIVALENT TO EITHER T1;T2 or T2;T1, which are called “serial” executions.

>NOTE: the effect of T1;T2 might be different
from T2; T1 (try above eg)

>> SQL ISOLATION LEVELS <<
In order to allow even more concurrency, SQL defines four ISOLATION LEVELS = choices about what interactions are allowed by transactions that execute at about the same time.

At the beginning of a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL X
where X =
SERIALIZABLE
REPEATABLE READ
READ COMMITTED
READ UNCOMMITTED

>> NOTE: ISOLATION LEVEL IS AN INDIVIDUAL/LOCAL CHOICE
Your choice, e.g., run serializable, affects only how you see the database, not how others see it. If you use weaker levels, you may get some non-serializable effect, but if you only want approximate answers (e.g., with aggregates), you may be happy.

Example: If Sam runs serializable, but Dianne doesn’t, then Dianne might see no prices for beers in Sam’s Bar. i.e., it looks to Dianne as if she ran in the middle of Sam’s transaction.

>> 3. READ COMMITTED LEVEL
If Dianne runs with isolation level READ COMMITTED, then she can see only committed data, but not necessarily the same data each time.
Example: Under READ COMMITTED, the interleaving \((\max)(\del)(\ins)(\min)\) is allowed, as long as Sam commits. Dianne sees \(\max < \min\).

>> 2. REPEATABLE READ LEVEL
Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time.
But the second and subsequent reads may see more tuples as well. (Called phantom reads.)
Example: Suppose Dianne runs under REPEATABLE READ, and the order of execution is \((\max)(\del)(\ins)(\min)\). \((\max)\) sees prices 2.50 and 3.00. \((\min)\) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by \((\max)\).

>> 4. READ UNCOMMITTED
A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed, and may never do so. (Called dirty read)
Example: If Dianne runs under READ UNCOMMITTED, she could see a price 3.50 even if Sam later aborts.

-------------------------------
>> "ACID" TRANSACTIONS <<
Deal with additional problems, not just concurrency:
* possible system crashes, network failures can leave db in inconsistent state
* need to suspend integrity constraint checking in the middle of a procedure; e.g., if \(\text{spouse}\) in table \(\text{Person(name, spouse references Person)}\) must be symmetric, and we have two people, Marie and Don, then after the first update of Marie’s spouse to Don, the system is in an inconsistent intermediate state until Don’s spouse is set to Marie too.

Atomic : Either the whole process is done or none is. (rollback, crash 1)
Consistent : Database constraints are preserved at the end!
Isolated : It appears to the user as if only one process executes at a time. (deals with concurrency issue)
Durable : Effects of a process do not get lost if the system. (crash 2)

>> Additional info:
* ROLLBACKs can be caused by system exceptions (division by 0) as well as concurrency control discovering deadlock and restarting transactions. (See below)

THEOREM The following protocol (strict 2-phase) is guaranteed to allow only serializable schedules: i) locks must be acquired as per above; ii) once a lock is acquired it is not

-------------------------------
>> IMPLEMENTATION OF ISOLATION:
2 kinds of LOCKS
* S (shared) lock; needed to \(READ\) data
* X (exclusive) lock; needed \(BEFORE\) \(WRITING\)
Each kind of lock can be requested (or released) on i) a single tuple, ii) a range of tuples/entire table.
So we now view transaction schedules as also containing requests/releases for locks:

Diane:
ask \(S(\text{entire serves}.price)\)
max
min
release \(S(...)\)
Sam:
ask \(X(\text{entire serves})\)
del
ins
release \(X\)

RULE: although two different transactions may proceed if they request S locks on the same data, any other lock requests involving X locks conflict, and result in the second requestor being blocked, and having to wait.
released till the end of the transaction (commit or rollback).

PROBLEM: Suppose T1 wants to do W(A); W(B); and T2 wants to do W(B); W(A). The following is a possible serial schedule:

T1: X-lock(A) W(A) X-lock(B) {blocked} W(B)
T2: X-lock(B) W(B) X-lock(A) {blocked} W(A)

Deadlock! (DBMS must detect and restart one of the transactions)

Explanation of SQL isolation levels using locks.

"Read uncommitted"
X-lock acquired and held
no S-locking
allows "dirty reads"

"Read committed"
X-lock acquired and held
S-lock obtained for tuples read but released right away
allows "unrepeatable reads"

"Repeatable read"
X-lock acquired and held
S-lock obtained for tuples read and kept till end of Xact
allows "phantoms" (locking at tuple level not enough)

" Serializable"
all locks acquired and kept till end
+ lock the indexes used by the query
(thereby holding off updates that might affect its answer)

>> IMPLEMENTATION IDEA OF ATOMICITY
logging of all updates, commits, etc (See Ramakrishnan & Gehrke summary)