-- CS 457
-- Lecture: Transactions
--
-- In this lecture you need to run sqlite from TWO terminals; see below

-- 1 - MOTIVATION FOR TRANSACTIONS
--
-- suppose we have a database for flights reservations
--
-- Run sqlite3 txn.db in terminal 1  -- run here all commands marked "User 1"
-- Run sqlite3 txn.db in terminal 2  -- run here all commands marked "User 2"

-- User 1:
.headers on
.mode columns
create table Flights(seat int, status int);

insert into Flights values(22,0); -- seat 22 is available
insert into Flights values(23,1); -- seat 23 is occupied
insert into Flights values(24,0);
insert into Flights values(25,0);
insert into Flights values(26,1);

-- User 1 and User 2 want to choose a seat, at about the same time:

-- User 1:
select * from Flights;

-- User 2:
.headers on
.mode columns
select * from Flights;

-- User 1: seat 22 is available, grab it:
update Flights set status = 1 where seat = 22;

-- User 2: seat 22 is available, grab it:
update Flights set status = 1 where seat = 22;

-- User 1:
.quit
-- User 2:
.quit

-- *** In class: what is wrong ?
-- The challenge:
--- For performance, want to execute many
-- applications concurrently. All
-- these applications read and write data.
--
-- But for correctness, multiple operations
-- often need to be executed as an atomic
-- transaction over the database.
-- In our example, both users have reserved the
-- same seat, and they are unhappy
--
-- Other possible problems when executing two
-- applications concurrently that read and write
-- to the same database:
-- WRITE-READ conflict ("Dirty read" or "Inconsistent Read")

-- One application is in the middle of performing some changes:
-- (a) A Manager is re-balancing budget and is moving
-- money between projects:
-- Step 1: Remove $10K from project 1
-- Step 2: Add $7K to project 2
-- Step 3: Add $3k to project 3
--
-- (b) The CEO wants to see the total balance and runs this:
--
-- select sum(money) from Budget
--
-- The CEO sees "inconsistent" data
--

-- READ-WRITE conflict ("Unrepeatable read")

-- An application reads the value of some database item: e.g., inventory.
-- Another application updates that value: e.g., someone else buys the
-- last book and now the inventory is zero.
-- The first application re-reads the value and finds that it has
-- changed... the inventory is now at zero.
-- This leads to "Unrepeatable read" or other anomalies

-- WRITE-WRITE conflict ("Lost update")

-- Account 1 = $100
-- Account 2 = $100
-- Total = $200
--
-- Application 1 writes $200 to account 1 (without reading its balance).
-- Application 1 writes $0 to account 2
--
-- Application 2 writes $200 to account 2
-- Application 2 writes $0 to account 1
--
-- Final state: one account has $200, the other one has $0
-- Total = $200 (unchanged)
--
-- What if the applications executed concurrently:
-- Application 1 writes $200 to account 1
-- Application 2 writes $200 to account 2
-- Application 1 writes $0 to account 2
-- Application 2 writes $0 to account 1
--
-- Where did the money go?

--That's not all! What if a failure happens while an application
--is updating the database? This can also create problems:
--e.g., What if your browser crashes while you are purchasing a $1K gift
--for your pet? What do you do?

----------------------------------------------------------------

-- 2 - Definitions and Properties

-- Transaction = a collection of statements that are executed atomically
--
-- it looks like this:
-- begin transaction;
-- . . .
-- commit; -- or rollback;
-- Start from scratch:
-- rm txn.db
-- sqlite3 txn.db

-- User 1:

-- Rerun the first example as follows:

-- User 1:
.headers on
.mode columns
create table Flights(seat int, status int);
insert into Flights values(22,0); -- seat 22 is available
insert into Flights values(23,1); -- seat 23 is occupied
insert into Flights values(24,0);
insert into Flights values(25,0);
insert into Flights values(26,1);

begin transaction;

-- User 2:
.headers on
.mode columns
begin transaction;

-- User 1:
select * from flights;

-- User 2:
select * from flights;

-- User 1:
update flights set status = 1 where seat = 22;

-- User 2:
update flights set status = 1 where seat = 22; -- DENIED !

-- User 2:
commit; -- (or rollback) CAN'T ASSIGN SEAT

-- User 1:
commit;

---------------------------------------------
-- Definition: a SERIAL execution of the transactions is one in which
-- transactions are executed one after the other, in serial order
-- Fact: nothing can go wrong if the system executes transactions serially
-- *** In Class: the database system could execute all transactions
-- serially, but it doesn't do that. WHY NOT?
-- Definition: a SERIALIZABLE execution of the transactions is one
-- that is equivalent to a serial execution
-- *** In class: repeat the two transactions by user 1 and 2, but
-- switch the commit order. That is, user 1 commits while user 2
-- continues the transaction. But user 1 receives an error when
-- attempting to commit. WHY ???

-- User 1:
begin transaction;

-- User 2:
begin transaction;
-- User 1:
select * from flights;

-- User 2:
select * from flights;

-- User 1:
update flights set status = 1 where seat = 24;

-- User 2:
update flights set status = 1 where seat = 24;  -- DENIED !!

-- User 1:
commit;  -- ERROR -- WHY??

-- User 2:
commit;  -- (or rollback)

-- User 1:
commit;  -- OK

-- sqlite ensures serializable execution of transactions
-- for that it may have to deny some requests

-- WARNING: You can see somewhat different behaviors with different DBMSs (more next lecture).

---------------------------------------------
-- ACID Properties:
---------------------------------------------
-- A DBMS guarantees the following properties of transactions:
-- (we will see next lecture that we can relax these properties a bit
-- to get higher performance)
--
-- -- Atomic
-- State shows either all the effects of txn, or none of them
--
-- -- Consistent
-- Txn moves from a state where integrity holds, to another where integrity holds
--
-- -- Isolated
-- Effect of txns is the same as txns running one after another (ie looks like batch mode)
--
-- -- Durable
-- Once a txn has committed, its effects remain in the database

---------------------------------------------
-- ACID: Atomicity
---------------------------------------------
-- Definition: each transaction is ATOMIC meaning that all its updates
-- must happen or not at all. Important for recovery and if
-- we need to abort a transaction in the middle.
--
-- Example: move $100 from account 1 to account 2
--
-- update Accounts
-- set balance = balance - 100
-- where account = 1
--
-- update Accounts
-- set balance = balance +100
-- where account = 2
--
-- If the system crashes between the two updates, then we are in trouble.
-- begin transaction
-- update Accounts
-- set balance = balance - 100
-- where account = 1
--
-- update Accounts
-- set balance = balance +100
-- where account = 2
-- commit
--
-- Now all updates happen atomically, when the commit is done.
--
-- begin transaction
--
-- read the balance in account 1
-- if ( balance < 100) ROLLBACK // Any update already performed is undone
-- else
--   update the two bank accounts
--   ...
--
-- commit
--
-- -------------------------------------------------------------
-- ACID: Isolation
-- - A transaction executes concurrently with other transaction
-- - Isolation: the effect is as if each transaction executes in isolation of the others
--
-- -------------------------------------------------------------
-- ACID: Consistency
--
-- The state of the tables is restricted by integrity constraints
--
-- How consistency is achieved:
-- - Programmer makes sure a txn takes a consistent state to a consistent state
-- - The system makes sure that the tnx is atomic
--
-- When defining integrity constraints, it is possible to specify
-- whether constraints can be delayed and checked only at the END
-- of the transaction instead of being checked after each statement.
--
-- -------------------------------------------------------------
-- ACID: Durability
--
-- The effect of a transaction must continue to exists after the transaction, or the whole
-- program has terminated
--
-- Means: write data to disk
--
-- -------------------------------------------------------------
-- 3 - More about aborting transactions
--
-- ROLLBACK
-- - If the app gets to a place where it can't
-- complete the transaction successfully, it can
-- execute ROLLBACK
-- - This causes the system to "abort" the
-- transaction
-- - The database returns to the state without any of
-- the previous changes made by activity of the
-- transaction
-- Reasons for Rollback
-- - User changes their mind ("ctl-C"/cancel)
-- - Explicit in program, when app program finds a problem
-- E.g. when the # of rented movies > max # allowed
-- - System-initiated abort
-- - System crash
-- - Housekeeping, e.g. due to timeouts
--
-- ==============================================================
-- 4 - Comments
--
-- -- Consider how ACID transactions help application development.
--
-- -- By default, when using a DBMS, each statement is its own
-- transaction!
--
-- -- Turing Awards to Database Researchers
-- - Charles Bachman 1973 for CODASYL (data model before relational model)
-- - Edgar Codd 1981 for relational databases
-- - Jim Gray 1998 for transactions!
-- - Michael Stonebreaker 2014 for PostgreSQL
--
-- ==============================================================
-- Implementation of Transactions
-- sqlite = one lock for the entire database
-- (SQL Server, DB2 = one lock per record (and index entry, etc) -- next lecture)
-- for more information on sqlite: http://www.sqlite.org/atomiccommit.html
--
-- Step 1: every transaction acquires a READ LOCK (aka "SHARED" lock)
-- all these transactions may read happily
--
-- Step 2: when one transaction wants to write
-- acquire a RESERVED LOCK
-- this lock may coexists with many READ LOCKs (but not other RESERVED LOCKS)
-- writer transaction may now write (others don't see the updates)
-- reader transactions continue to read, new readers accepted
--
--
-- Step 3: when writer transaction wants to commit
-- try to acquire an EXCLUSIVE LOCK: no READ LOCKs may coexist
-- 1. acquire a PENDING LOCK: coexists with READ LOCKs
-- no new READ LOCKs are accepted during a PENDING LOCK
-- wait for all read locks to be released
-- 2. acquire an EXCLUSIVE LOCK:
-- all updates are written permanently to the database and log
--
--
-- create a new database:
-- rm txn.db
-- sqlite3 txn.db
--
create table r(a int, b int);
insert into r values (1,10);
insert into r values (2,20);
insert into r values (3,30);
-- Run the script below in three different windows, T1, T2, T3:
--
-- T1:
begin transaction;
select * from r;
-- T1 has a READ LOCK
-- T2:
  begin transaction;
  select * from r;
  -- T2 has a READ LOCK

-- T1:
  update r set b=11 where a=1;
  -- T1 has a RESERVED LOCK

-- T2:
  update r set b=21 where a=2;
  -- T2 asked for a RESERVED LOCK: DENIED

-- T3:
  begin transaction;
  select * from r;
  commit;
  -- everything works fine, could obtain READ LOCK

-- T1:
  commit;
  -- SQL error: database is locked
  -- T1 asked for PENDING LOCK -- GRANTED
  -- T1 asked for EXCLUSIVE LOCK -- DENIED

-- T3': (run T3' in the same window as T3)
  begin transaction;
  select * from r;
  -- T3 asked for READ LOCK -- DENIED (because of T1)

-- T2:
  commit;
  -- releases the RESERVED LOCK

-- T1:
  commit;
  -- T1 asked for EXCLUSIVE LOCK -- GRANTED

-- T3':
  select * from r;
  -- T3 asked for READ LOCK -- GRANTED
  commit;