Transactions — An Introduction

Tony Printezis

tony@dcs.gla.ac.uk

Dept of Computing Science

University of Glasgow

17 Lilybank Gardens

Office G103, x6043

Motivation — Scenario 1

- One way to populate a `JTable` with the contents of the **Band** table
  - count how many rows there are in **Band**
    - `SELECT COUNT(*) FROM ...`
  - create a `JTable` with that many rows
  - populate the `JTable` after getting the actual data from **Band**
    - `SELECT Name, Country, WebSite FROM ...`
- **Does this work?**

Motivation — Scenario 2

- Imagine I'm transferring 50 pounds from account A to account B
- The updates necessary to reflect this are
  1. balance of A -= 50
  2. balance of B += 50
- After update 1 has been propagated to the DB, a *system failure* prevents update 2 to be propagated to the DB
- **Is this correct?**

Motivation — Scenario 3

- Consider if we add a row to the **Release** table that contains a **bid** field that does not appear in the **Band** table
  - (this is a valid insert to the **Release** table, if no constraints have been defined)
- **Is this correct?**

Transactional Programming

- Purpose of Transactions (Tx)
  - DB usage is essentially **concurrent**
  - *Isolation* gives the **illusion** of a single user
    - implies much easier application programming
  - Requirements
    - **stability**: data shouldn't change while you're using it
    - **isolation**: your logic should not be corrupted by others' logic
    - **reliability**: when you you've done an update, it should persist
    - **fairness**: you should be able to make reasonable progress

ACID Transactions

- DB Community invented **ACID Transactions**
  - ***A****tomic*
    - all or nothing updates
  - ***C****onsistent*
    - takes the database from one consistent state to another one
  - ***I****solated*
    - it is possible to write an application ignoring the possibility of concurrent applications
  - ***D****urable*
    - once committed, reliably persistent
- . . . as well as *Undoable*
  - voluntary *abort (rollback)* of the Tx

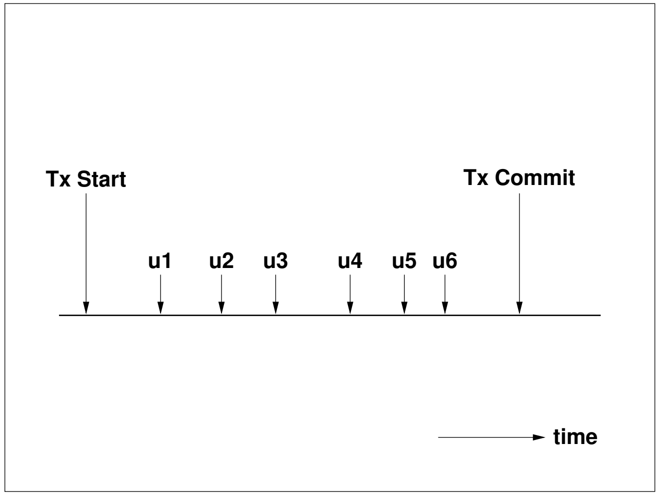
Transactions

- A *Transaction* is essentially a series of actions against a DB
  - **updates** *and* **reads**
- These actions are performed
  - atomically and durably,
  - isolated from other transactions,
  - while preserving the consistency of the data in the DB

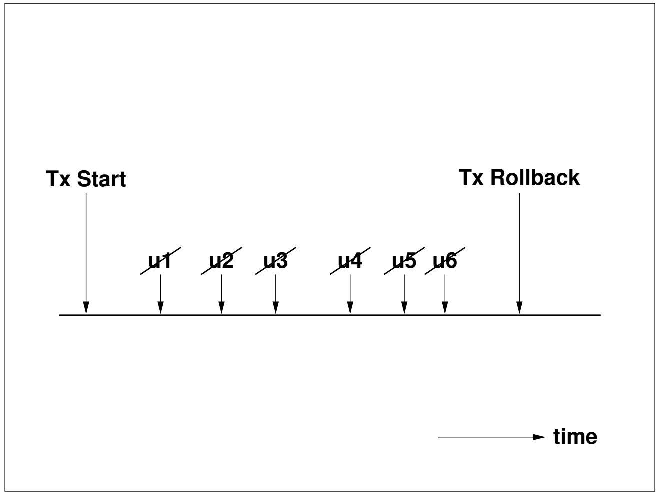
Atomicity

- ***All*** *of the effects of the operations within the Tx are preserved in the database*, or
- ***None*** *of the effects of the operations within the Tx are preserved in the database*
- Complications
  - delimiting the Tx
    - e.g. Tx **begin**, Tx **commit**, Tx **rollback**
  - synchronizing with external actions
    - e.g. issuing money

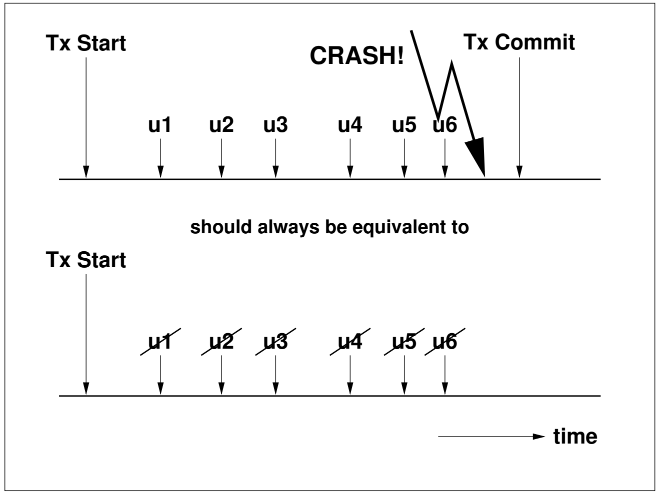
Atomicity — Commit



Atomicity — Rollback (Tx Undone)



Atomicity — System Crash



Atomicity — Delimiting Tx

- Who decides which updates should be performed atomically?
  - The DBMS?
    - **No.**
    - the DBMS does not know anything about the *application logic*
  - The Application Programmer?
    - **Yes.**
    - only the programmer knows about the application logic
    - only they can decide which updates should be part of one Tx

Consistency

- Internal Consistency
  - required by DBMS to operate
- Logical Consistency
  - required by Applications to operate
  - ideally *"Does the data make sense?"*
  - in practice
    - "Are all constraints & assertions satisfied?"
    - if not **force** the Tx to rollback
    - issue error information to the application

Consistency — When?

- When should we perform the consistency checks?
  1. Per update?
  2. Per commit?
- A single update might violate a constraint. . .
  - "add a row to the **Release** table with a **bid** 6"
  - when 6 does not exist in the **Band** table
- . . . but it may not, if it is part of a group of updates
  - "add a row to the **Release** table with a **bid** 6" and
  - "add a row to the **Band** table with a **bid** 6"
  - (*atomicity* — remember?)
- Cosistency can *only* be checked at *commit time*
  - only then enough information is available to do so

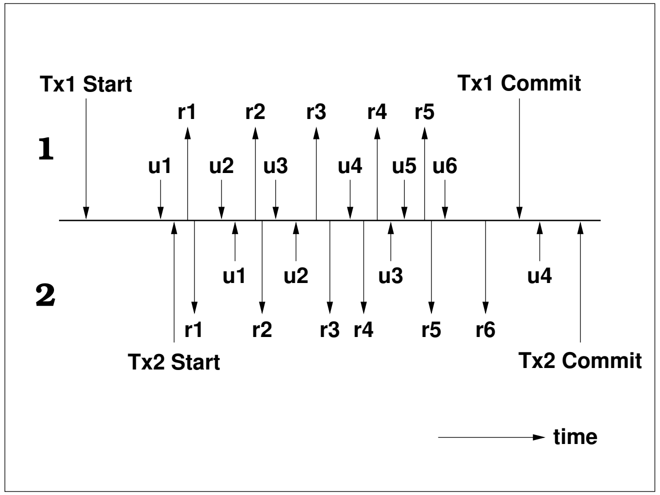
Consistency — How?

- Again, DBMS *cannot* decide what a consistent state of the data is
  - only the application programmer can do so
- *Trigger*
  - application-level code invoked when an particular events occurs
  - e.g. commit
- To perform consistency checks, the programmer registers
  - assertions
  - triggers

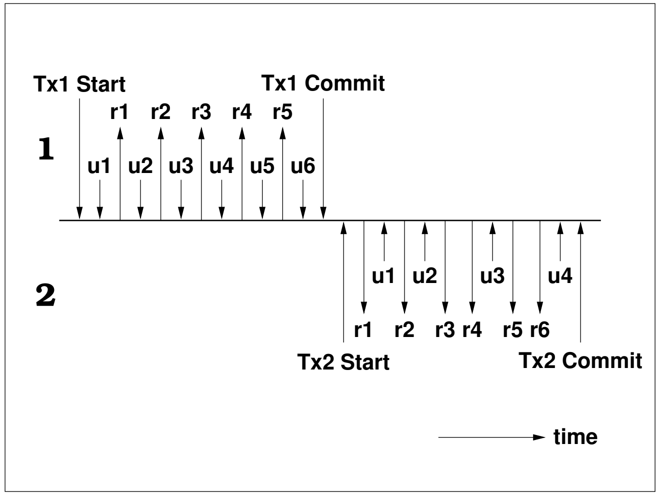
Isolation

- Informally
  - *An application developer writes code as if they are the only one coding & only one instance of one application runs at once*
- Formally
  - The set of Tx that run must be *serialisable*
    - *i.e. their effects (on the database) must be equivalent to some serial sequence of the individual Tx running one at a time.*
    - assumption of independence
    - avoid non-commutative operations interfering

Multiple Concurrent Tx



Should have the same effect as. . .



Erroneous Interleaving

Tx1	Tx2	Account Balance
start Tx 1	start Tx 2	100
read balance (100)	read balance (100)	100
pay in 100 (200)	pay out 50 (50)	100
write back (200)		200
commit Tx 1	write back (50)	**50**
	commit Tx 2	**50**

What You Actually Want. . .

Tx1	Tx2	Account Balance
start Tx 1	start Tx 2	100
read balance (100)	*wait*	100
pay in 100 (100)	*wait*	100
write back (200)	*wait*	200
commit Tx 1	*wait*	200
	read balance (200)	200
	pay out 50 (150)	200
	write back (150)	150
	commit Tx 2	150

Isolation

- In Practice
  - application programmers must . . .
    - keep Tx short
    - otherwise they delay other Tx (holding locks)
  - . . . and must take over I/O & GUI actions
    - otherwise can introduce delays and
    - cause irreversible external state change

Isolation Locking

- Two Popular Methods
  - **Locking**
    - stake claim before use
      - i.e. take a *lock*
    - hold it until the end of Tx
  - **Optimistic Concurrency**
    - assume "collisions" hardly ever happen
    - track the *Read Set* (*RS*) and the *Write Set* (*WS*)
    - at commit time check that
      - no WS_i intersects with $RS \cup WS$, and
      - no RS_i intersects with *WS*
    - if condition fails
      - **abort and retry!**

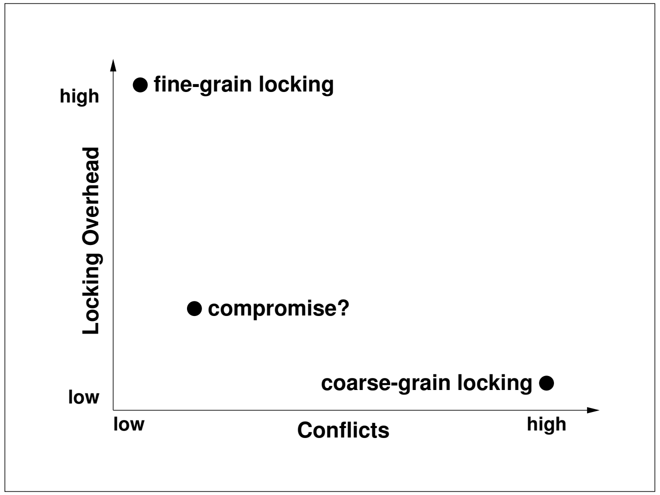
Locks

- Also referred to as
  - **Mutexes**
  - **Latches**
- A lock *guards* a data structure from being manipulated by more then one thread (or process)
  - only one thread can *take* the lock
  - the others have to *wait* until lock is released
- *Critical Region*
  - code that updates the data structure
  - only one thread can *enter* it
- Java has locks!
  - `synchronized` methods or statements

Locking Granularity

- Locks claimed *implicitly* as needed
  - e.g. as an object is about to be read or updated
    - physical locking (e.g. per page)
    - logical locking
      - e.g. per DB, Cluster, Catalog, Schema, Table, Row
- Trade-offs
  - coarse locking granularity
    - low locking overhead
    - more conflicts
  - fine locking granularity
    - less conflicts
    - high locking overhead

Locking Granularity

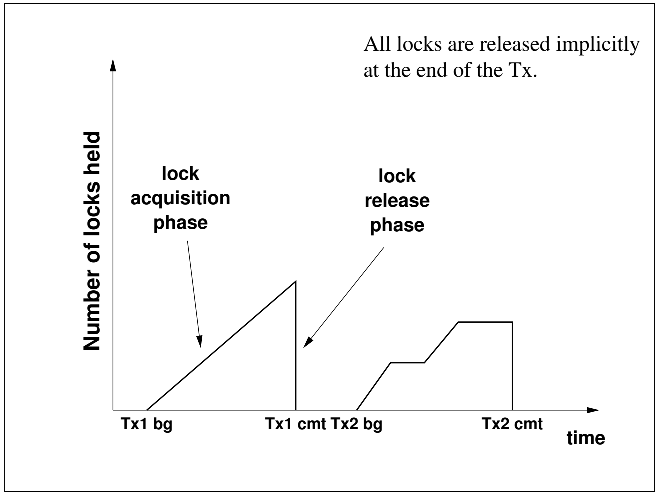


Two-Phase Locking

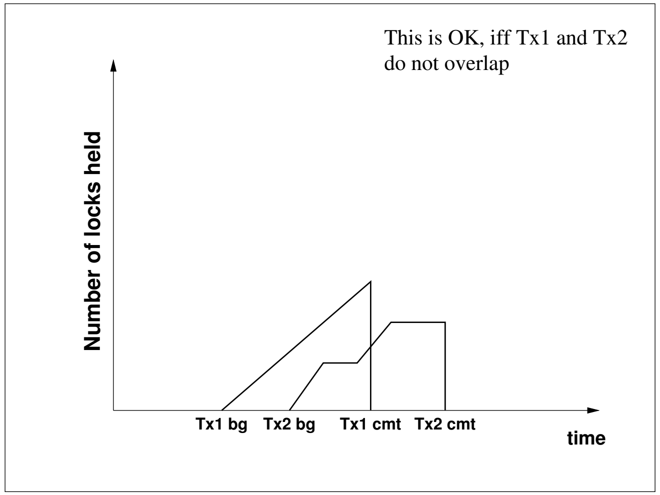
- **Lock Acquisition Phase**
  - locks are taken as data is accessed
- **Lock Release Phase**
  - locks are implicitly released at the end of Tx
  - (either at commit or rollback)

- **Cannot release and then retake the same lock during a Tx**
  - since somebody else might have taken it and updated the data
  - always keep all the locks until the end of the Tx

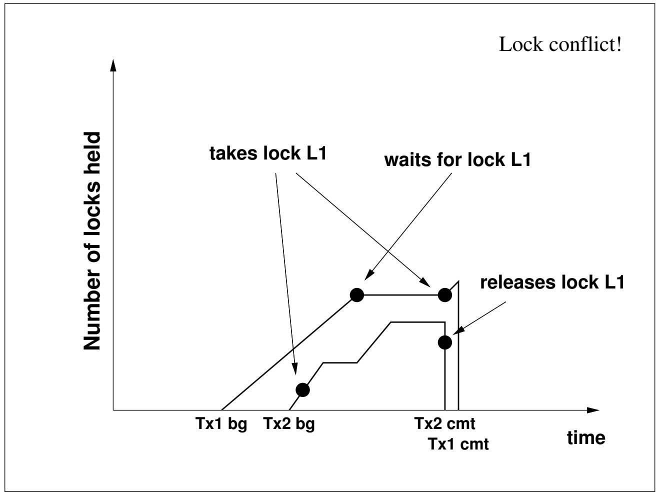
Acquisition of Locks (i)



Acquisition of Locks (ii)



Acquisition of Locks (iii)



Lock Conflict

- Two types of Lock
  - Read Locks (RL) & Write Locks (WL)
  - each object (subject to a lock) may have many readers
    - each RL can have one or more owners
  - each object may have only one writer
    - each WL can have *exactly* one owner
  - the owner of a RL may promote it to a WL
    - iff there are *no* other owners of that RL
- Denote possibilities by a *Conflict Matrix*

Conflict Matrix

Lock requested by Tx_k	Existing Lock held by Tx_i: RL_i	Existing Lock held by Tx_i: WL_i
RL_k	**OK**	**OK**, iff k == i
WL_k	**OK**, iff k == i AND no other RL	**OK**, iff k == i

When a Lock is not Granted

- A lock is not *granted* because of a conflict
- When the current owner(s) end, the lock will become free
- If not *deadlock*
  - (i.e. *not* final link in a cycle of suspended requests)
  - suspend processing requestor Tx until requested lock is freed
- If deadlock
  - (i.e. it *is* the final link in a cycle of suspended requests)
  - force the requestor to rollback & retry later
    - may be approximated by a time out
    - depends on ability to rollback and retry without program action

Deadlocks — Example

"Tx1 is transferring money from Account A to Account B"
"Tx2 is transferring money from Account B to Account A"

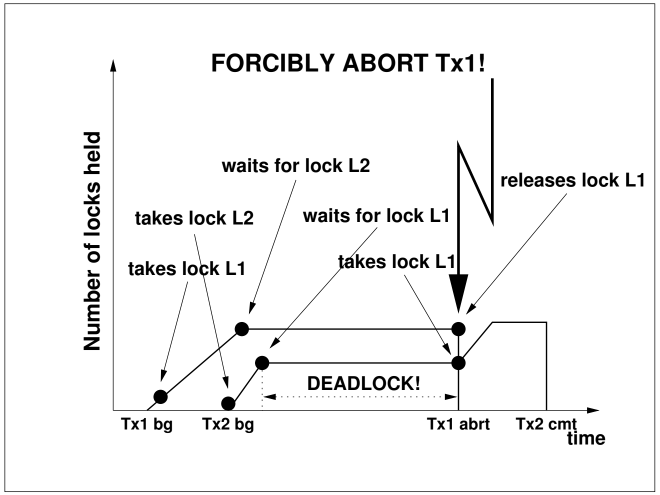
Tx1	**Tx2**	**Acc A**	**Acc B**
start Tx 1	start Tx 2		
lock account A	lock account B	locked(Tx1)	locked(Tx2)
read balance of A	read balance of B	locked(Tx1)	locked(Tx2)
calculate new sum	calculate new sum	locked(Tx1)	locked(Tx2)
try to lock account B	try to lock account A	locked(Tx1)	locked(Tx2)

DEADLOCK!

Deadlocks

- Those occur when there is a chain of lock requests of the form
  - Tx_1 has X and is waiting for A,
  - Tx_2 has A and is waiting for B,
  - Tx_3 has B and is waiting for C,
  - ...
  - Tx_n has W and is waiting for X
- Detect cycle, choose a *victim* Tx, and
  - **forcibly rollback victim**

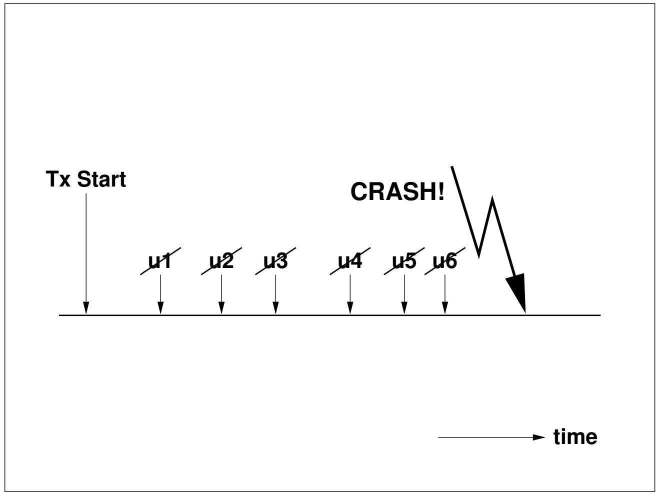
Acquisition of Locks — Deadlock



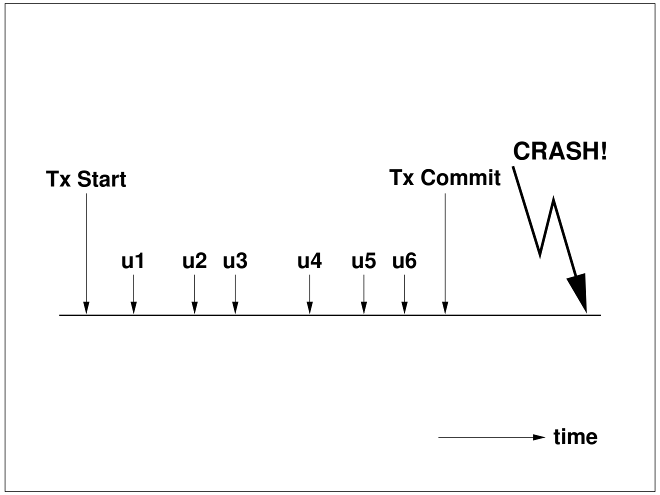
Durability

- Ensuring that a failure can't lose **committed changes**
  - software failures: DB system, OS, application, etc.
  - hardware failures: CPU, disk, etc.
- Your data can *never* be totally safe!
  - probability of losing it is always non-zero
    - you can never eliminate it
    - you can only decrease it to acceptable levels
      - e.g. less than the probability of all life on Earth being wiped out by an asteroid impact

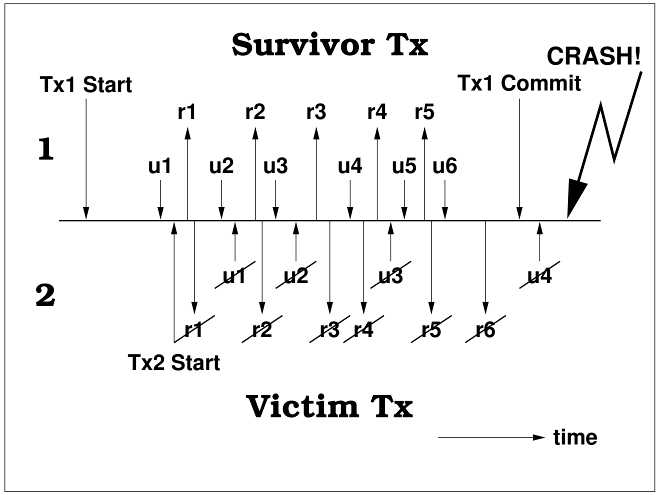
Atomicity & Durability (i)



Atomicity & Durability (ii)



Multiple Tx Durability



Implementation Principles of Durability

- *Logging*
  - ensures durability and guards against most *software* failures
  - all updates to the database are recorded in a *Log*
    - log resides on disk too
  - if a crash occurs, the log has enough information to
    - *redo* committed updates, if necessary
    - *undo* uncommitted updates, if necessary
- A log is a series of *Log Records*
  - each log record
    - represents a single update to the database
    - contains a *before-image* so we can undo the update
    - contains an *after-image* so we can redo the update

Write-Ahead Logging (WAL)

- Most widely used logging protocol
- Always record an update in the log *before* you write it to the database
- This guarantees that
  - if a committed update is not written to the database, the log contains enough information to be able to redo it
  - if an uncommitted update is written to the database, the log contains enough information to be able to undo it
- Not action needs to be taken for the other two cases
  - if a committed update is written to the database
  - if an uncommitted update is not written to the database

Archiving

- The presence of a log does not guard data against disk failures, even if the log and the database are stored on different physical disks
  - without the log, the rest of the database cannot operate as the log might contain essential data to bring it to a consistent state
  - without the rest of the database, the log itself cannot operate as it only contains the latest updates
- The database must be frequently *archived*
  - possibly large storage requirements
  - can do this incrementally by reading the log
    - note: the log contains a complete history of all updates!
- RAID arrays are also typically used to provide higher fault-tolerance and availability

Transactions and JDBC

- Whenever you perform actions against a database using JDBC
  - read-only queries
    - `executeQuery`
  - updates
    - `executeUpdate`

  these are performed in terms of a Tx
  - either implicitly or explicitly
- Remember: every action against a database *has* to be performed in terms of a Tx
  - otherwise, you cannot take advantage of the ACID properties of Tx

AutoCommit Mode

- Each JDBC connection can operate in two modes
  - **AutoCommit On**
    - (default mode)
    - every statement is executed in its own Tx
  - **AutoCommit Off**
    - there is a Tx associated with each connection and the programmer has to explicitly commit or abort it
    - the Tx begin is implicit
- (of course, different connections within the same client do not have to operate in the same mode)

JDBC Transaction API

- On the `Connection` interface
  - `void setAutoCommit(boolean mode)`
    - sets the AutoCommit mode on and off for that connection
    - this can be changed several times within the same application
  - `void commit()`
    - when AutoCommit mode is off, it commits all the updates that took place through that connection
  - `void rollback()`
    - when AutoCommit mode is off, it aborts all the updates that took place through that connection

AutoCommit Mode On

- Default Mode
- Every statement executed against the database is run inside a new Tx automatically
  - each invocation of `executeQuery` and `executeUpdate`...
  - ...either on `Statement` or `PreparedStatement`
- The Tx commits
  - when the `ResultSet` that was returned from `executeQuery` is either `close()`ed, or when the last row has been read
  - when `executeUpdate` returns succesfully
- In this mode, all Tx are assumed to commit
  - ...but might not due to a problem in the database
  - `SQLException`

AutoCommit Mode Off

- When AutoCommit mode is off, it is up to the programmer to explicitly commit or rollback a Tx
- A Tx remains active until
  - `commit` or `rollback` is called, or
  - the connection is terminated
- If `commit` is not called before the connection is terminated, the Tx is automatically **aborted**!
  - no implicit `commit`
- Programming with AutoCommit off is considerably more error-prone
  - *Use it only when you have to!*

Releasing Resources

- Whenever you've finished with a
  - `Connection`,
  - `Statement`,
  - `PreparedStatement`, and
  - `ResultSet`

  you are recommended to invoke `close()` on the object
- This releases resources held by
  - the client application, and
  - the DBMS

  (these resources are also released when the client terminates)
- You cannot use any of the above objects after calling `close()` on it