atomicity consistency

Database Transactions

a.k.a. ACID that's good for you

isolation DURABILITY

- Many real-world data tasks involve multiple database operations—multiple inserts, multiple updates, multiple deletes
- Textbook example: a bank transfer

```
transfer_funds(source, destination, amount):

  UPDATE account SET balance = balance - amount
  WHERE id = source

  UPDATE account SET balance = balance + amount
  WHERE id = destination
```

- What if something goes wrong between the two updates? —the transfer must take place completely, or else it must not happen at all

Enter the Transaction

- Recognizing that a single meaningful database operation (to an application) may require multiple CRUD operations, the idea of a transaction came into being—essentially, a composite CRUD
- Another major four-letter database acronym—ACID—describes a correctly-implemented transaction
- Fortunately, databases do implement transactions for you: knowing ACID is about understanding transactions and not programming them

(unless you actually do try to program a database)

  - Atomicity means that a transaction behaves as a single operation—either all of it happens, or none
  - Consistency means that a transaction maintains a database's integrity: it cannot break any rules or constraints (e.g., uniqueness, references, etc.)
  - Isolation means that a transaction should behave like it took place while nothing else was happening—i.e., concurrent transactions should behave like they happened sequentially, without interleaving
  - Durability means that a transaction, once finished, cannot be undone—in other words, a completed transaction is permanent, even if something fails later

CrOSsover!

If transactions and ACID sound vaguely familiar, your déjà vu is déjà true—database transactions are extremely related to concurrency control in operating systems:

- Concurrency solutions rely on the atomicity of certain operations—that is, they must not be interruptible, either taking place completely or not at all (e.g., setting locks or semaphores)
- Improper concurrency over critical sections will ruin the consistency of a system

- Critical sections must execute as if in isolation—that is, proper concurrency means that they behave as if they are the only one running on the system
  - This relates to serialization or serializability: the result of multiple concurrent operations should be identical to some sequence of those operations performed one after the other
- Perhaps the one item that doesn't have an exact analog is durability because, after all, software and data in main memory can be highly dynamic and rapidly changing—still, concurrent threads can be said to be durable in that, upon execution, their effects should influence all future operations

Transactions in SQL

- Transactions are straightforward in SQL—by default, every single SQL statement constitutes a transaction of its own (think about how a single `INSERT` statement, for example, would fulfill ACID)
- To initiate a multi-operation transaction, just say:

```
BEGIN;
```

- You can also say `BEGIN TRANSACTION` or `BEGIN WORK`, but that's just syntactic sugar—`BEGIN` suffices (all with that semicolon, of course)

- To end a transaction, you either `COMMIT;` (save it) or `ROLLBACK;` (cancel it)—this ensures atomicity
- An error encountered during a transaction will cancel it—none of the activities since the transaction began will "stick," nor can you do anything else until you `ROLLBACK` (thus ensuring consistency)
- For isolation, multiple transactions will behave as if they happened strictly one after another and if this is impossible, transactions may be rolled back (try it—this one is fun to see; just run multiple psqls)
- Behind the scenes, a committed transaction is stored in a way that keeps it durable—subsequent crashes or failures won't lose them

Transactions in the DAL

- One doesn't copy-paste `BEGIN`s, `COMMIT`s, or `ROLLBACK`s all over DAL code (whether as SQL or as function calls)—this is unnecessarily repetitive and error-prone
- As such, the general practice is to write DAL code without express references to transactions
- Instead, the framework or environment within which a DAL is written is expected to set up well-defined transaction boundaries so that you can concentrate on your application's specific database operations

- This provides excellent background on the subject:

  https://docs.sqlalchemy.org/en/13/orm/session_basics.html#when-do-i-construct-a-session-when-do-i-commit-it-and-when-do-i-close-it

  (the "session" language here is SQLAlchemy-specific but the way it should live outside DAL code is not)
- In line with this, the DAL code we see in this course tries to keep the idea of transactions outside of DAL functions—that part of the code may need to change depending on the environment: a web service framework may wrap a single request around a transaction automatically; annotations, decorators, or other mechanisms may mark transaction boundaries