Recap

- Data-savviness is the future!
- Notion of a DBMS
- The relational data model and algebra: bags and sets
- SQL Queries, Modifications, DDL
- Database Design
- Views, constraints, and triggers
- Indexes
- Query processing & optimization
- Next: transactions

So far…

- We have explored how to interact with a DBMS using SQL…
- However, at any time, many users can be interacting with the DBMS via various modalities…

- Imagine a bank… activities include:
  - Users may be making changes via mobile apps, websites, ATMs, …
    - These changes may span multiple SQL statements (e.g., move money from one account to the next)
  - Banks may be updating investments, interests rates, …
- Things could go wrong:
  - Users change their mind mid-sequence of changes
  - ATMs, mobile apps or even backend servers can crash

What are ways things can go wrong?

- **Example 1: interference**
- Flight Booking Program:
  - A: Check if seat is available: SELECT-FROM-WHERE
  - B: Book seat (change availability of seat from T to F, and assign passenger name to that seat): UPDATE-SET…
- Two simultaneous runs 1 & 2

A1

A2

B1

B2

- Two executions could lead to same seat being assigned to both passengers (but only user 2 has the seat actually)

What is the solution?

- Group the "SFW" (which verified availability) and "USW" (which reserved the seat) into one unit, called a TRANSACTION (or txns)
- *A transaction is a sequence of operations that are considered a basic unit within a database*

- Typical DBMSs guarantee *serializability* of txns
  - The state at any point is equivalent to a *serial* order of the txns
  - That is, either user 1's txn executes first, and then user 2, or the other way around, rather than in parallel
  - In actuality, the DBMS doesn't force the txns to execute in strict sequence, but uses intelligence to ensure correct parallelization
    - e.g., if the two users are reserving two different seats, it's probably OK to parallelize

What else could go wrong?

- **Example 2: problems even with a single user**
- Even if one user is interacting with a DB at a time …
- Transfer $100 from checking to savings
  - Check if checking has >$100
  - Reduce checking balance by $100
  - Increase savings balance by $100

- What if there's a crash after the second step?

What is the solution?

- Databases will ensure that the txn happens all at once or not at all
    - So either balance has been correctly transferred or not at all

- The way to maintain this is to "log" all changes that are being made so that the database can correctly *recover* from failures
    - For example, if we log that we're going to make the change from checking to savings, we can consult the log to identify where we were when the program crashed
        - We can either "redo" the changes to persist them
        - Or "undo" the changes to reverse them

Declaring Txns

BEGIN;
SQL statement 1;
SQL statement 2;
…
SQL statement n;
COMMIT;

- SQL statements 1…n are an *atomic* unit.
- These atomic units are *serialized.*

Declaring Txns

```
BEGIN;
SQL statement 1;
SAVEPOINT s1;
…
ROLLBACK TO s1;
SQL statement i;
…
SQL statement n;
COMMIT;
```

- ROLLBACK TO s1 causes changes from s1 onwards to be rolled back.
- A complete ROLLBACK will ensure that no effects from the txn are present

Txn Manager & Log Manager

- Components of the DBMS
- Job: making sure that txns execute as expected

- Purpose 1: log changes so that you can recover from failures
  - The *write-ahead log* allows you to write what you're planning to do before you do it so that you don't mess up the data, and you can recover from errors

- Purpose 2: make sure that simultaneous txns don't interfere with each other
  - The *concurrency control* module is used to ensure that txns are correctly parallelized
    - Various methods: locks, timestamps, …

- Won't cover logging/recovery and concurrency control in this class…

Transaction Manager and Log Manager

- Together, achieve the so-called **ACID** properties
  - ATOMICITY: Each txn happens in entirety or not at all.
    - e.g., if a user's sequence of updates fails because their mobile app crashes, it either reflects entirely or not at all
    - *Via logging (and recovery)*
  - CONSISTENCY: Each txn leaves the database in a consistent state
    - e.g., if the database obeys constraints (ref integrity, NULL, PK etc.) prior to changes, they obey it after changes
    - *Via concurrency control (and checking of constraints prior to commit)*
  - ISOLATION: Each txn happens without interference from others
    - *Via concurrency control*
  - DURABILITY: Each txn's effects are durable, especially in the face of power or other types of failures
    - *Via logging (and recovery)*

Speeding Txns Up

- Maintaining these four properties over the course of many simultaneous txns is hard!
- Can cause delays …

- As a user of txns, there are several things we may do to speed things up by indicating properties of a txn or relaxing requirements …

I. Indicating Properties

- Can specify a txn as READ ONLY

  SET TRANSACTION READ ONLY;

- Benefit: DBMSs can use this to schedule this txn to happen in parallel with other READ ONLY txns, since there is no danger of conflicts (and therefore loss of isolation)

- Otherwise, default is READ WRITE

2. Relaxing Requirements

- Can relax requirements by setting an ISOLATION LEVEL for a txn.
- Default is SERIALIZABLE
  - Effect is equivalent to a serial schedule

- To motivate various ISOLATION LEVELS, let's see how we can relax requirements to read uncommitted data

2. Relaxing Requirements: Dirty Reads

- A *dirty read* is a read of *dirty data:* data written by a txn that hasn't committed yet.
- Flight Booking Program:
  - A: Check if seat is available: SELECT-FROM-WHERE
  - B: Book seat (change availability of seat from T to F, and assign passenger name to that seat): UPDATE-SET…
- Two simultaneous runs 1 & 2

  A1

  A2

  B1

  B2

Example with dirty reads: Implicit read with B2

- Here, if both txns were OK with dirty reads, this could lead to both passengers thinking they have the seat (while only B2 does) => bad!

2. Relaxing Requirements: Dirty Reads

- A *dirty read* is a read of *dirty data:* data written by a txn that hasn't committed yet.
- Slightly different booking program:
  - A: Check if seat is available: SELECT-FROM-WHERE
  - B: Book seat (change availability of seat from T to F, and assign passenger name to that seat): UPDATE-SET…
  - C: If passenger has been assigned a seat, then COMMIT, else go to step A.
- Does this fix the problem?
- Two simultaneous runs 1 & 2

A1

    A2

B1

    B2

C1 (returns to A)

    C2 (commits)

A1

Here, dirty reads are fixed later (via Step C)

2. Relaxing Requirements: Dirty Reads

- You can allow a transaction to read dirty data via the following:

```
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ UNCOMMITTED;
```

- Note that an ISOLATION LEVEL is from the perspective of a txn:
  - That is, the given txn is reading dirty data
  - Not a global property of all txns

Other ISOLATION LEVELs

```
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ UNCOMMITTED;
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION READ WRITE
    ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION READ WRITE
    ISOLATION LEVEL SERIALIZABLE; // Default
```

READ COMMITTED / REPEATABLE READ

- In the READ COMMITTED case, a txn will only see data of other txns that have been committed.
  - Will avoid a "reader program" interacting with with the bank transfer program from having problems
  - One issue is that over the course of a txn, it might see different instances of committed data for the same query (based on different txns that may have committed in the interim)

- In the REPEATABLE READ case, if the txn sees a certain tuple for a given query, it will continue to see the same tuple if query is issued again
  - One issue is that the query may also see new "phantom" data

Txns and Recovery: Summary

- The database handles many users concurrently interacting with the data as well as failures and errors in a seamless way
- However, this handling can come with overheads, reducing # of txns serviced per second
- Can regulate this by various things:
  - Setting txns as read-only
  - Keeping them "small"
  - Setting more tolerant isolation levels
  - Don't leave a txn "idle" longer than necessary
  - Use a connection pooling system (to reduce overhead)