Recap: Why Databases?
Transactions: A Useful Abstraction for DB Access

Natural match with web apps: can say transaction = handler logic for one HTTP request. In general, follows some natural unit of work in a program.

A classic acronym for features that transactions should provide:

Atomic
Consistent
Isolated
Durable

What problems does ACID solve?
Programming simplicity
Performance (esp. with parallelism)
Fault tolerance
To handle a single request, server runs a number of database operations.

Transactions are all or nothing. Afterward, looks like either all steps happened or none did. (i.e., the set of operations runs atomically.)
Consistent

Shouldn't be able to follow a user that doesn't exist.

Database enforces invariants for all possible simultaneous interactions with servers.
With multiple transactions running at once, it appears as though they run in some serial order. It's not possible for any transaction to observe intermediate states of any other.
Once a transaction finishes, the changes it has made remain available from the database, until overwritten, even in the face of common kinds of hardware failures, etc.
Important to Remember

The database engine doesn't need to implement all of these rules literally! E.g., transactions don't really run one-at-a-time. Optimizations can allow simultaneous transactions to appear to run serially. (in any serious implementation of ACID, at least)
**MongoDB Scorecard?**

Transaction = a single operation on a single document

- **Atomic?** Yes
- **Consistent?** No; Mongo doesn't know your invariants.
- **Isolated?** Yes
- **Durable?** Yes

Kind of a cop-out, since it's hard to make documents the natural unit of work for most applications!
Example: Fritter's Revenge

Please add this message and all its hashtag records, atomically.

Actually, **indexes** can at least partially overcome those two performance issues, but we'd still run into trouble if we needed more complicated queries.

“Which messages used this hashtag?”

“Which hashtags appear in this message?”

**Mongo ideas**

Use two separate collections? No transactions across documents.

Nest hashtags inside messages? Need expensive linear scan to answer 1st question below.

Nest messages inside hashtags? Need expensive linear scan to answer 2nd question below.

Actually, **indexes** can at least partially overcome those two performance issues, but we'd still run into trouble if we needed more complicated queries.
Code demo: MongoDB Fritter--
Relational Databases

Database is a set of **tables**, each with a schema assigning **types** to columns.

**Messages**

<table>
<thead>
<tr>
<th>usr</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Followers**

<table>
<thead>
<tr>
<th>follower</th>
<th>followed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Only primitive datatypes allowed. Build up aggregate values with relations! (Sound familiar?) → Avoid needing to decide on representation at schema definition.

Syntax in SQL (most popular relational lang.):

```sql
CREATE TABLE messages(
    usr TEXT NOT NULL,
    text TEXT NOT NULL
);
```

```sql
CREATE TABLE followers(
    follower TEXT NOT NULL,
    followed TEXT NOT NULL
);
```
Joins: Using Linkages Between Columns

**Informal query:** find all messages written by users that user 'myname' is following.

**SQL syntax:**
```
SELECT usr, text
FROM messages, followers
WHERE usr = followed
AND follower = 'myname'
```

In general, FROM considers Cartesian product of table lists. E.g., here we consider all pairs of messages and follower records.

**Big idea:** write data schemas & queries using natural mathematical notation, and let the DB engine find optimized code.
Indexes

CREATE INDEX ON messages(usr);

Ask DB engine to maintain data structures to support fast lookup of messages by user.

**Big idea**: give optimization hints that exist independently of your main schema/program.
Code demo: PostgreSQL Fritter--
Explicit Transactions

Start.

Run a series of queries & updates atomically & without interference by other transactions.

Make results visible to transactions that start “later.”

BEGIN TRANSACTION;
SELECT ... FROM ... WHERE ...;
INSERT INTO messages(usr, text)
    VALUES ('user1', 'blah blah');
INSERT INTO followers(follower, followed)
    VALUES ('user1', 'user2');
SELECT ... FROM ... WHERE ...;
COMMIT; ROLLBACK;

**Big idea:** write queries & updates assuming a very simple concurrency semantics, and let the DB engine find optimized code.
Keys

CREATE TABLE messages(
    id INTEGER NOT NULL PRIMARY KEY,
    usr TEXT NOT NULL,
    text TEXT NOT NULL
);

General meaning of declaring $K_1$, $\ldots$, $K_n$ as a key: no combination can be duplicated across multiple rows.
Constraints (E.g., Foreign Keys)

CREATE TABLE hashtags(
    tag TEXT NOT NULL,
    message INTEGER NOT NULL REFERENCES messages(id),
    PRIMARY KEY (tag, message)
);

Foreign-key constraints ("REFERENCES") force a column (or a group of them) to be a valid reference to another table.
Code demo: PostgreSQL Fritter-- with hashtags