Review

• Suppose we have
  – a students table with a primary key of sid
  – a faculty table with a primary key of fid

• How would we implement a many-to-many advisor mapping from students to faculty?
Many-to-Many

- Many-to-many requires a mapping table
  - Note that the primary key constraint on (sid, fid) disallows two rows with the same pair
  - But not two rows with the same sid or two rows with the same fid

```sql
CREATE TABLE advisor_map (  
sid INTEGER REFERENCES students(sid),
fid INTEGER REFERENCES faculty(fid),
PRIMARY KEY (sid, fid)
);
```
One-to-Many

• What if we wanted to make the advisor relationship one-to-many
  – That is, each student has one faculty as advisor but each faculty can have multiple students as advisees
One-to-Many

• Option 1: Use a mapping table (like with many-to-many) but add a uniqueness constraint on sid
  – Disallows two rows with the same sid

```
CREATE TABLE advisor_map (
    sid INTEGER UNIQUE REFERENCES students(sid),
    fid INTEGER REFERENCES faculty(fid),
    PRIMARY KEY (sid, fid)
);
```
One-to-Many

- Option 2: Add a foreign key to faculty(fid) in the students table
  - Allows multiple students to have the same fid, but each student can have only one fid

```sql
CREATE TABLE students (  
sid INTEGER PRIMARY KEY,  
...,  
fid INTEGER REFERENCES faculty(fid)  
);
```
One-to-One

• What if we wanted to make the advisor relationship one-to-one?
  – So each student has only one faculty advisor and each faculty advises only one student
    • (Doesn't make much sense in real life, but it's just an example)
One-to-One

• Option 1: Use a mapping table (like with many-to-many) but add uniqueness constraints on both sid and fid

CREATE TABLE advisor_map (  
sid INTEGER UNIQUE REFERENCES students(sid),
fid INTEGER UNIQUE REFERENCES faculty(fid),
PRIMARY KEY (sid, fid)
);
One-to-One

• Option 2: Add a foreign key to faculty(fid) in the students table and make it unique
  – Alternatively, could add a unique foreign key to students(sid) in the faculty table

```sql
CREATE TABLE students (  
sid INTEGER PRIMARY KEY,  
...,
  fid INTEGER UNIQUE REFERENCES faculty(fid)  
);
```
Total Participation

• Assume we have a one-to-many advisor relationship between students and faculty
  – How would we guarantee that every student has an advisor?
Total Participation

• If we use the option where we put the foreign key in the students table then it is easy
  – Make the foreign key column NOT NULL

  CREATE TABLE students (  
    sid  INTEGER PRIMARY KEY,  
    ... ,  
    fid  INTEGER NOT NULL REFERENCES faculty(fid)  
  );
Total Participation

• What if we're using a mapping table?
  – Difficult. Need some way to guarantee that every sid in the students table has an entry in the mapping table.
  – Cannot be done with just UNIQUE and NOT NULL constraints.
Example

• Consider the database design for a shopping website
  – Need to store the address of a customer. Where should it go?

• Option 1: Put the address in the customer table

<table>
<thead>
<tr>
<th>cust_id</th>
<th>last_name</th>
<th>first_name</th>
<th>...</th>
<th>street</th>
<th>city</th>
<th>state</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

• But what if we want to store both a shipping address and a billing address?
  – Doable, but what if most customers use the same address for shipping and billing?
    • Either have to duplicate data in the shipping and billing columns or leave one set blank
      – Lots of columns which are mostly NULL can lead to performance problems

<table>
<thead>
<tr>
<th>customers</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_id</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Example

• Worse yet, what if a customer wants to use a different address for a particular order?
  – No good way in a single table to store multiple shipping or billing addresses for a customer
    • Could add yet more columns, but would still need a predetermined limit
Example

• Option 2: Use a separate addresses table
  – But what should the relationship between customers and addresses be?
    • One-to-one, one-to-many, many-to-one?

<table>
<thead>
<tr>
<th>customers</th>
<th>addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_id</td>
<td>street</td>
</tr>
<tr>
<td>last_name</td>
<td>city</td>
</tr>
<tr>
<td>first_name</td>
<td>state</td>
</tr>
<tr>
<td></td>
<td>zip</td>
</tr>
</tbody>
</table>
Example

• One-to-one?
  – Probably not, as the point of using separate tables was so customers could have multiple addresses.

• One-to-many?
  – Possibly. But what about families and roommates? We need some way for two or more customers to have the same address.

• Many-to-many?
  – Possibly, considering families or roommates, but it's probably better to keep data for different customers completely separate.
One-to-Many

• We need a way for two customers to have the same address while having different rows in the address table.
  – Effectively, we want a one-to-many relationship from customers to addresses (one customer can have many addresses) while allowing multiple address rows to have the same values for street, city, state, and zip, without having duplicate rows.
One-to-Many

- We need extra columns in the address table.
- How do we implement one-to-many?
One-to-Many

• How to we implement one-to-many?
  – The primary key from the "one" side of the relationship becomes a foreign key on the "many" side.
  • Create a cust_id column in the addresses table and make it a foreign key to the cust_id column in the customers table.
One-to-Many

- We can now have
  - Multiple addresses per customer (as long as cust_id in the addresses table is not declared as UNIQUE)
  - Two or more customers with the same [street, city, state, zip]
- We cannot have two addresses with the same [street, city, state, zip] for the same cust_id
  - But that's probably OK

<table>
<thead>
<tr>
<th>customers</th>
<th>addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_id</td>
<td>last_name</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

• Now that a customer can have multiple addresses, how do we know which is their current shipping and billing addresses?

<table>
<thead>
<tr>
<th>customers</th>
<th>addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_id</td>
<td>cust_id</td>
</tr>
<tr>
<td>last_name</td>
<td>street</td>
</tr>
<tr>
<td>first_name</td>
<td>city</td>
</tr>
<tr>
<td></td>
<td>state</td>
</tr>
<tr>
<td></td>
<td>zip</td>
</tr>
</tbody>
</table>
Example

• Option 1: Add `cur_ship` and `cur_bill` columns to the addresses table
  – Can enforce single current billing and single current shipping address for a customer by making `[cust_id, cur_ship]` and `[cust_id, cur_bill]` unique
Example

• Option 2: Add foreign keys to the customers table for the current shipping and billing addresses.
  – But what is the primary key on the addresses table?

<table>
<thead>
<tr>
<th>customers</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_id</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_id</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Natural Keys

• Natural Key
  – A natural key is a key defined over the meaningful columns of a table
    • Given the way we've defined it, the primary key on the addresses table would need to include all of the columns
    • So, if we wanted to include a foreign key to the addresses table in the customers table, we would need five extra columns
Synthetic Keys

• Synthetic (a.k.a. Surrogate) Key
  – A synthetic key is a unique identifier for a row that has no meaning outside of the database model
    • We can add an automatically generated addr_id column to the addresses table and use that as the primary key
    • (Actually, cust_id might be a synthetic key for the customers table, too)

<table>
<thead>
<tr>
<th>addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr_id</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Synthetic Keys

- We can now use `addr_id` as a foreign key in the `customers` table for the current shipping and billing addresses
Synthetic Keys

• Another benefit of having a synthetic addr_id is that it is easier to allow editing of addresses
  – In general, it's better to avoid changing the value of a primary key field of a row

• Without the addr_id, if we changed the street field of an address then we've changed the value of the primary key for that row
  – Must make sure we change all referring rows

• With addr_id, we're free to change any of the other fields of a row
Synthetic Keys

• Two general approaches:
  – Database generated
  – Application generated

• Two general types:
  – Integer valued
  – UUID (Universally Unique Identifiers)
Database vs Application

• Database generated
  – Easy to implement
    • Database handles concurrency issues

• Application generated
  – If universally unique then easy to use across databases or applications/devices
Integer vs UUID

• Integer
  – Fast to generate and search (especially if database generated)
  – Difficult to use across multiple applications or databases

• UUID
  – Slower to generate and search
  – Easy to use across multiple applications or databases
Integer Keys

• Most DBMSs have a way to auto-generate sequential integer values
  – CREATE SEQUENCE

• Most also have shorthand for using sequences as keys
  – PostgreSQL has SERIAL
  – MS SQL Server has IDENTITY
SERIAL

• PostgreSQL

```sql
CREATE TABLE addresses (  
    addr_id SERIAL,
    ...
);  

is short for

CREATE SEQUENCE addresses_addr_id_seq;
CREATE TABLE addresses (  
    addr_id integer NOT NULL DEFAULT nextval('addresses_addr_id_seq'),
    ...
);
ALTER SEQUENCE addresses_addr_id_seq
OWNED BY addresses.addr_id;
```
UUIDs (a.k.a. GUIDs)

- Usually application generated
  - MS SQL Server can generate UUIDs
  - PostgreSQL has an extension
- Standard for UUIDs
  - 128 bit value
    - So $2^{128}$, or about $3 \times 10^{28}$, possible values
  - Represented as 32 hexadecimal digits
    - 550e8400-e29b-41d4-a716-446655440000
  - Different variants allow construction from
    - MAC address, random values, MD5 hash, ...
CREATE TABLE customers (  
cust_id SERIAL PRIMARY KEY,  
last_name TEXT,  
first_name TEXT
);

CREATE TABLE addresses (  
addr_id SERIAL PRIMARY KEY,  
cust_id INTEGER  
    REFERENCES customers(cust_id)  
street TEXT,  
city TEXT,  
state TEXT,  
zip TEXT
);

Example
Large Integers

• INTEGERs are 4-byte signed values
  – -2,147,483,648 to +2,147,483,647

• BIGINTs are 8-byte signed values

• BIGSERIAL is an 8-byte SERIAL