Database Design Guidelines

• Every table should have a primary key
  – Since a primary key constraint guarantees uniqueness, this implies no duplicate rows

• Every primary key should consist of as few columns as possible
  – Ideally, each primary key would consist of only one column (except for many-to-many mapping tables)

• Ideally, the key values for any given row will not change for the lifetime of that row
  – Changing a row's key values requires changing all referencing row's foreign key values
Database Design Guidelines

• Every non-key column should depend directly on the entire key
  – If there is a multi-column key and one or more other columns depend on only part of the key then they should be split out into another table
  – If one or more columns depend on the key only indirectly then they should be split out into another table
Database Design Guidelines

• One-to-one mappings
  – If there is a one-to-one mapping between two tables then consider combining into a single table unless
    • Doing so would create an indirect dependence, or
    • There is a natural separation between the two tables, or
    • One table is accessed much less frequently than the other table
  – The foreign key relationship should be one way
    • Only one table should have a foreign key referencing the other table
    • Use a uniqueness constraint to guarantee one-to-one
Database Design Guidelines

• One-to-many mappings
  – Do *not* combine into a single table
  – Create a foreign key constraint in the "many" table that references the primary key of the "one" table
    • e.g., if a department can have many employees but an employee can be in only one department then the employee table (the "many" side) should have a foreign key reference to the department table (the "one" side)
Database Design Guidelines

• One-to-many mappings
  – If the "many" table depends on only part of the primary key of the "one" table then consider splitting the "one" table
  – If there is extra data that depends on the combination of the keys from the two tables (and not just one or the other) then use a separate mapping table (as with many-to-many)
    • Use a uniqueness constraint on the "many" foreign key to guarantee one-to-many
      – Consider accomplishing this by making the "many" foreign key the primary key of the mapping table
Database Design Guidelines

• Many-to-many mappings
  – Do not combine into a single table
  – Use a separate table with foreign key references to both of the primary tables
    • Use a uniqueness constraint on the combination of the foreign keys to avoid duplicates
      – Consider accomplishing this by making the combination of foreign keys the primary key of the mapping table
  – Extra data that depends on the combination of the keys from the two tables (and not just one or the other) should go in this table
Database Design Guidelines

• Synthetic keys can be useful, but be careful of using them to break other rules
  – e.g., having rows which are duplicates except for just the synthetic key
• Avoid using synthetic keys for many-to-many (or one-to-many) mapping tables unless
  – The mapping table contains other columns and could naturally be considered a separate entity
  – Other considerations (such as an object-relational mapping tool) require a simple serial key
CREATE TABLE users (  
  user_id BIGSERIAL PRIMARY KEY,  
  username TEXT UNIQUE,  
  password TEXT  
);

Note: for security, passwords should not be stored in plain text in the database. Generally, passwords should be salted and hashed. Although many DBMSs have libraries to help with this, there is no SQL standard. DBMS independence can be achieved by hashing at the application level.

<table>
<thead>
<tr>
<th>user_id</th>
<th>username</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bugs</td>
<td>######</td>
</tr>
<tr>
<td>2</td>
<td>daffy</td>
<td>######</td>
</tr>
<tr>
<td>3</td>
<td>porky</td>
<td>######</td>
</tr>
<tr>
<td>4</td>
<td>petunia</td>
<td>######</td>
</tr>
</tbody>
</table>
Example

• Questions
  – Should the username field be unique?
    • What would happen if two users had the same username?
  – If the username field is unique then should that be the primary key (and get rid of the user_id column)?
    • What happens if a user wants to change his username?
Example

CREATE TABLE books (  
isbn TEXT PRIMARY KEY,  
title TEXT,  
description TEXT,  
price INTEGER  );

Note: floating-point types such as real and double should not be used to store monetary values. Postgresql supports the NUMERIC datatype for exact, arbitrary precision values, but INTEGER works as well for this example.

<table>
<thead>
<tr>
<th>isbn</th>
<th>title</th>
<th>description</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>9780743273565</td>
<td>Great Gatsby, The</td>
<td>…</td>
<td>900</td>
</tr>
<tr>
<td>9781400079988</td>
<td>War and Peace</td>
<td>…</td>
<td>1360</td>
</tr>
<tr>
<td>9780486432151</td>
<td>Moby-Dick</td>
<td>…</td>
<td>500</td>
</tr>
<tr>
<td>9781613823590</td>
<td>Ulysses</td>
<td>…</td>
<td>1795</td>
</tr>
</tbody>
</table>
Example

• Questions
  – Is ISBN a reasonable primary key?
    • String keys are less efficient than integer keys
  – Numeric or Integer for price?
    • Numeric allows for exact, arbitrary-precision numbers, but arithmetic on numerics is slow
      – But we're generally returning the values to the application and doing the arithmetic there
  – Why no author(s) column?
CREATE TABLE authors (  
author_id BIGSERIAL PRIMARY KEY,  
last_name TEXT,  
first_name TEXT  
);

<table>
<thead>
<tr>
<th>author_id</th>
<th>last_name</th>
<th>first_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fitzgerald</td>
<td>F. Scott</td>
</tr>
<tr>
<td>2</td>
<td>Tolstoy</td>
<td>Leo</td>
</tr>
<tr>
<td>3</td>
<td>Melville</td>
<td>Herman</td>
</tr>
<tr>
<td>4</td>
<td>Joyce</td>
<td>James</td>
</tr>
</tbody>
</table>
Example

• Questions
  – What should we do about authors whose names don't fit the last_name, first_name pattern?
    • e.g., F. Scott Fitzgerald
  – Should we have single name field instead?
  – Why no book(s) column?
Example

CREATE TABLE books_authors_map (  
isbn TEXT REFERENCES books(isbn),  
author_id BIGINT REFERENCES authors(author_id),  
CONSTRAINT books_authors_key  
    PRIMARY KEY (isbn, author_id)  
);

<table>
<thead>
<tr>
<th>isbn</th>
<th>author_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>9780743273565</td>
<td>1</td>
</tr>
<tr>
<td>9781400079988</td>
<td>2</td>
</tr>
<tr>
<td>9780486432151</td>
<td>3</td>
</tr>
<tr>
<td>9781613823590</td>
<td>4</td>
</tr>
</tbody>
</table>
Example

• Questions
  – Why use a composite key? Why not just use isbn or author_id alone as the primary key?
  – Should we create a synthetic key instead?
CREATE TABLE shopping_carts (  
  user_id BIGINT REFERENCES user(user_id) PRIMARY KEY,  
  last_modified TIMESTAMP WITHOUT TIME ZONE  
);
Example

• Questions
  – There is a one-to-one relationship between users and shopping carts. Should we combine the tables into one?
  – Should we create a cart_id as a synthetic key instead of using user_id as the primary key?
  – PostgreSQL has "timestamp with time zone" and "timestamp without time zone" data types. Which should we use?
Timestamps

• Date/times are complicated by time zones and especially Daylight Saving Time
• PostgreSQL has two different data types for storing timestamps
  – "timestamp without time zone"
    • stores a moment in time without any time zone information
  – "timestamp with time zone"
    • stores a moment in time together with time zone information
• (Other databases such as MySQL have similar types with different names)
Timestamps

• **Design Guideline:** *always store timestamps in UTC (Coordinated Universal Time)*
  – Makes comparison between timestamps much easier
  – Conversion to/from various time zones also much easier
Timestamps

• Problem: users generally do not want to see date/times in UTC
  – Option 1
    • Use "timestamp without time zone"
    • Convert all timestamps from user's time zone to UTC at the application level before sending to database
    • Convert all timestamps from UTC to the user's time zone at the application level after retrieving from database
Timestamps

– Option 2

• Set the default time zone in the PostgreSQL configuration to UTC
• Use "timestamp with time zone"
• Always specify a time zone when inserting or updating
• Use "AT TIME ZONE" in select to convert to user's time zone

SELECT last_modified AT TIME ZONE tz
    FROM shopping_carts
WHERE user_id = 1;
Note that for both options we need to know the user's time zone.
- Might be useful to store a user's preferred time zone

**Design Guideline:** *store time zones in name form, not offset*
- e.g., use "US/Central" not ",-500"
  - Does ",-500" mean US/Central without DST or US/Eastern with DST?
Example

CREATE TABLE shopping_cart_items (  
cart_id BIGINT REFERENCES shopping_carts(user_id),  
isbn TEXT REFERENCES books(isbn),  
quantity INTEGER,  
CONSTRAINT shopping_cart_items_key  
PRIMARY KEY (cart_id, isbn)  
);

<table>
<thead>
<tr>
<th>cart_id</th>
<th>isbn</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9780743273565</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>9780486432151</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>9780486432151</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: shopping_cart_items is actually a many-to-many mapping between shopping_carts and books.
Example

• Questions
  – Why use a composite key? Why not just use cart_id or isbn alone as the primary key?
  – Should we create a synthetic key instead?