Relational Model

- Table = relation.
- Column headers = attributes.
- Row = tuple

<table>
<thead>
<tr>
<th>name</th>
<th>manf</th>
</tr>
</thead>
<tbody>
<tr>
<td>WinterBrew</td>
<td>Pete’s</td>
</tr>
<tr>
<td>BudLite</td>
<td>A.B.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Beers

- Relation schema = name(attributes).
  Example: Beers(name, manf).
  
  ✦ Order of attributes is arbitrary, but in practice we need to assume the order given in the relation schema.

  ✦ Relation instance is current set of rows for a relation schema.

- Database schema = collection of relation schemas.
Keys in Relations

An attribute or set of attributes $K$ is a *key* for a relation $R$ if we expect that in no instance of $R$ will two different tuples agree on all the attributes of $K$.

- Indicate a key by underlining the key attributes.

- Example: If *name* is a key for *Beers*:

  \[ \text{Beers}(\text{name}, \text{manf}) \]
Why Relations?

- Very simple model.
- *Often* a good match for the way we think about our data.
- Abstract model that underlies SQL, the most important language in DBMS's today.
  - And even influential in competitors like OQL.
Abstract Vs. Concrete Relations

The relational model implemented in SQL differs slightly from the abstract notion of relations that we shall learn first.

- Big difference: abstract relations are sets of tuples; SQL relations are bags of tuples (i.e., duplicates allowed).

- Abstract relations vital for foundation:
  - Semantics of SQL statements.
  - Formal meaning of functional dependencies, normalization.
Relational Design

- Relations are closer to real storage structures than the concepts of E/R or ODL.
  - Thus, going from E/R or ODL designs to relational often requires some additional intellectual input.

Easiest Case: Entity Set \(\rightarrow\) Relation

E. S. attributes become relational attributes.

Becomes:

\[ \text{Beers}(\text{name}, \text{manf}) \]
Slightly Harder: ODL Class Without Relationships

- Problem: ODL allows attribute types build from structures and collection types.
- Structure: Make one attribute for each field.
- Set: make one tuple for each member of the set.
  - More than one set attribute? Make tuples for all combinations.
- Problem: ODL class may have no key, but we should have one in the relation to represent “OID.”
Example

interface Drinkers (key name) {
    attribute string name;
    attribute Struct Addr
    {string street, string city,
    int zip} address;
    attribute Set<string> phone;
}

<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$</td>
<td>$s_1$</td>
<td>$c_1$</td>
<td>$z_1$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>$s_1$</td>
<td>$c_1$</td>
<td>$z_1$</td>
<td>$p_2$</td>
</tr>
</tbody>
</table>

- Surprise: the key for the class (name) is not the key for the relation (name, phone).
  - **name** in the class determines a unique object, including a unique *set* of phones.
  - **name** in the relation does not determine a unique tuple.
  - Since tuples are not identical to objects, there is no inconsistency!
Decompose Relations?

One option is to get phone into a separate relation (with name). The database would look like:

<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$</td>
<td>$s_1$</td>
<td>$c_1$</td>
<td>$z_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>$p_2$</td>
</tr>
</tbody>
</table>

- **Advantages:**
  1. Avoids redundancy in address components.
  2. Handles the case where someone has *no* phone.

- **Disadvantage:** Harder to answer queries that jump between two relations, e.g., “in what city is phone 650-725-4802?”
A Design Problem

interface Family {
    attribute Set<string> parents;
    attribute Set<string> children;
}

1. What is the key?

2. How should we represent a family with two parents and three children?

3. Would you favor decomposition into several relations?
E/R Relationships → Relations

Relation has attribute for *key* attributes of each E.S. that participates in the relationship.

- Key of relation excludes attributes from the “one” side if relationship is many-one.
- For a one-one relationship, choose which side provides the key of the relation.
- Renaming attributes OK.
  ✦ Essential if multiple roles for an E.S.
Likes\texttt{(drinker, beer)}
Favorite\texttt{(drinker, beer)}
Married\texttt{(husband, wife)}
Buddies\texttt{(name1, name2)}

- For one-one relation Married, we can choose either husband or wife as key.
Weak Entity Sets, Relationships → Relations

hosts\( (\text{hostName}) \)
logins\( (\text{loginName}, \text{hostName}) \)
\( \text{At}(\text{loginName}, \text{hostName}, \text{hostName2}) \)

- In At, hostName and hostName2 must be the same host, so delete one of them.
- Then, Logins and At become the same relation; delete one of them.
- In this case, Hosts’ schema is a subset of Logins’ schema. Delete Hosts?
- General rule: Delete the relation that comes from a many-one relationship supporting a weak entity set.
ODL Relationships

Pick one direction, say $A \rightarrow B$.

- Put key of $B$ attributes in the relation for class $A$.
- Prefer to make $A$ the “many,” if relationship is many-one.
- If relationship is many-many, we’ll have to duplicate $A$-tuples as in E/R.
Example

interface Drinkers {
    attribute string name;
    attribute string addr;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Beers favorite
        inverse Beers::realFans;
    relationship Drinkers husband
        inverse wife;
    relationship Drinkers wife
        inverse husband;
    relationship Set<Drinkers> buddies
        inverse buddies;
}

Drinkers(name, addr, wifeName, buddyName, beerName, favoriteBeer)