CS 325 - Reading Packet: "Database design, part 2"

Sources:

* Ricardo, "Databases Illuminated", Jones and Bartlett.
* Kroenke, "Database Processing: Fundamentals, Design, and Implementation", 7th edition, Prentice Hall, 1999.
* Rob and Coronel, "Database Systems: Design, Implementation, and Management", 3rd Edition, International Thomson Publishing, 1997.
* Connolly and Begg, "Database Systems: A Practical Approach to Design Implementation and Management", 3rd Edition, Addison-Wesley.
* Korth and Silberschatz, "Database System Concepts"
* Sunderraman, "Oracle 9i Programming: A Primer", Addison-Wesley.
* Ullman, "Principles of Database Systems", 2nd Edition, Computer Science Press.

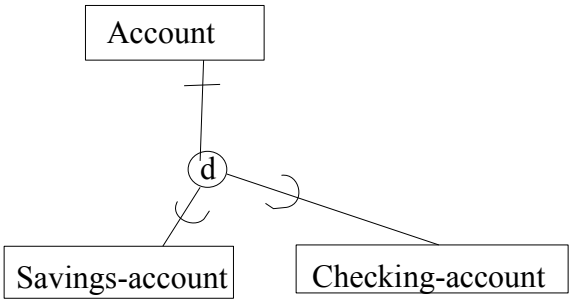
BASICS of DATABASE DESIGN, Part 2

What we have discussed so far is the large majority of what one would encounter in converting database models to database designs/schemas. Today we add a few more pieces that you should also know how to handle during the database design process.

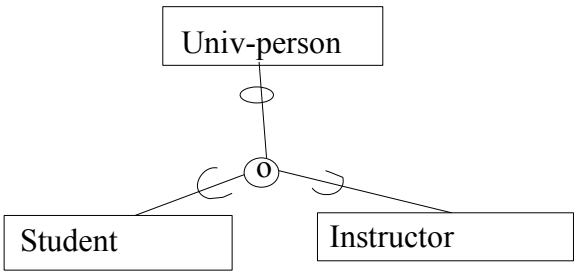
Handling Supertype/Subtype Entity Classes

As supertype/subtype entity classes are not infrequent, it is useful to know how to convert them into appropriate tables in a database design/schema.

Recall these examples of supertype and subtype entity classes from earlier in the semester (now with attribute information included):




```
Account         Savings-Account          Checking-Account
-------         ---------------          ----------------
ACCT_NUM        Interest_rate            Min_balance
Date_opened     Bonus_feature (MV)       Per_ck_charge
Balance
```




```
Univ-person          Student              Instructor
-----------          -------              ----------
UNIV_ID              GPA                  is_W4_on_file
Last_name            Semester_matricd     Date_first_hired
First_name           Semester_graduated
Campus_email
```

Here are the steps for handling supertype/subtype entity classes in the database design process:

* create a "base" table for each entity class (the supertype AND each subtype)

* determine an appropriate primary key for the supertype entity's "base" table

* add the supertype entity's "base" table's primary key to EACH of the subtype entity's "base" tables, making it the primary key for those tables as well, AND making it a foreign key referencing the supertype entity's "base" table

* to make certain queries easier:
    * if the subtype entities are **disjoint** subtypes (**d** on the circle in the ER Model), then each supertype instance is at most one of the subtype instances. Add a *_type attribute to the supertype entity's "base" table, whose domain is some code or value indicating of WHICH subtype a supertype instance is.

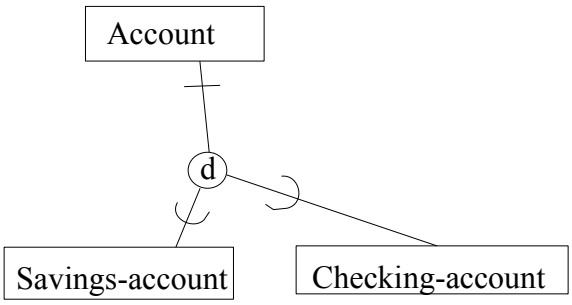
    * if the subtype entities are **overlapping** subtypes (**o** on the circle in the ER Model), then each supertyle instance can be more than one of the subtype instances. Add an attribute is_* to the supertype entity's "base" table for *each* subtype entity, a yes/no, 1/0, or otherise "binary"-domained attribute that indicates if the supertype instance is of that subtype as well.

* handle the remaining attributes and any other relationships involving these entity classes following the normal design process

Let's walk through these steps for our two example models.

Accounts-model example

Consider the Account/Savings-Account/Checking-Account scenario first.




```
Account         Savings-Account          Checking-Account
-------         ---------------          ----------------
ACCT_NUM        Interest_rate            Min_balance
Date_opened     Bonus_feature (MV)       Per_ck_charge
Balance
```

First we create "base" tables for each supertype and subtype entity class:

```
Account(
Savings_Account(
Checking_Account(
```

Then we determine an appropriate primary key for the supertype entity's "base" table:

```
Account(ACCT_NUM,
Savings_Account(
Checking_Account(
```

... and add the supertype entity's "base" table's primary key to EACH of the subtype entity's "base" tables, making it the primary key for those tables as well, AND making it a foreign key referencing the supertype entity's "base" table:

```
Account(ACCT_NUM,

Savings_Account(ACCT_NUM,
    foreign key (acct_num) references Account

Checking_Account(ACCT_NUM,
    foreign key (acct_num) references Account
```

Finally, since these are **disjoint** subtypes (note the **d** in the circle in its model), add an **acct_type**

attribute to Account's "base" table, whose domain will indicate whether a particular Account row is a Savings account or a Checking account:

```
Account(ACCT_NUM, acct_type

Savings_Account(ACCT_NUM,
    foreign key (acct_num) references Account

Checking_Account(ACCT_NUM,
    foreign key (acct_num) references Account
```

The remaining single-valued attributes should be placed in the associated entity's "base" table:

```
Account(ACCT_NUM, acct_type, date_opened, balance,

Savings_Account(ACCT_NUM, interest_rate,
    foreign key (acct_num) references Account

Checking_Account(ACCT_NUM, min_balance, per_ck_charge,
    foreign key (acct_num) references Account
```

...which leaves entity Savings-account's multi-valued attribute Bonus_features. But one just follows the normal design process for this at this point, creating a new table for this multi-valued attribute, whose primary key is a composite primary key consisting of the primary key of the entity's "base" table and the multi-valued attribute, and with that primary key from the entity's "base" table defined to be a foreign key back to the entity's "base" table:

```
Account(ACCT_NUM, acct_type, date_opened, balance)

Savings_Account(ACCT_NUM, interest_rate)
    foreign key (acct_num) references Account

Savings_Acct_Bonuses(ACCT_NUM, BONUS_FEATURE)
    foreign key (acct_num) references Savings

Checking_Account(ACCT_NUM, min_balance, per_ck_charge)
    foreign key (acct_num) references Account
```

In particular, note that new table Savings_Acct_Bonuses's foreign key acct_num references the **Savings** table, not the Account table; it is quite all right to reference a primary key that is itself a foreign key. It would *not* be the same to reference the Account table instead, in this case -- a savings account bonus feature *must* be an attribute of a savings account in particular, not just any account. Having the foreign key reference the **Savings** table will ensure that this is so, thanks to the **referential integrity** checking enforced by the DBMS by declaring the foreign key in this way.

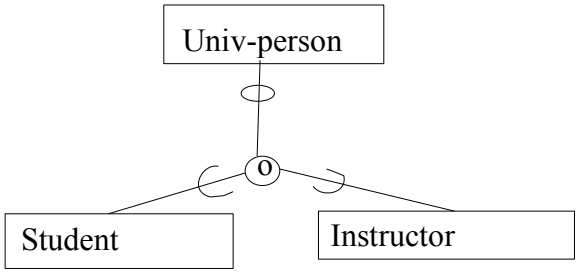
(The same principle would apply if, for example, Savings were a parent (the 1 side) in a 1:N

relationship -- then its primary key Acct_num would be placed in the child entity's "base" table as a foreign key referencing Savings. And if Checking were involved in a N:M relationship with another entity, then its primary key Acct_num would be placed in the new intersection table created as a result, also as a foreign key referencing Checking. Make sure that this is clear to you!)

If we had further relationships between these entities and other entities, we would handle each of them in turn. But as there are no other entities here, the database design/schema is complete (in terms of the table structures and relationships, anyway).

Univ-person model example

Now, for our other example supertype/subtypes model:




```
Univ-person          Student              Instructor
-----------          -------              ----------
UNIV_ID              GPA                  is_W4_on_file
Last_name            Semester_matricd     Semester_hired (MV)
First_name           Semester_graduated
Campus_email
```

First we create "base" tables for each supertype and subtype entity class:

```
Univ_person(
Student(
Instructor(
```

Then we determine an appropriate primary key for the supertype entity's "base" table:

```
Univ_person(UNIV_ID,
Student(
Instructor(
```

... and add the supertype entity's "base" table's primary key to EACH of the subtype entity's "base" tables, making it the primary key for those tables as well, AND making it a foreign key referencing the supertype entity's "base" table:

```
Univ_person(UNIV_ID,
```

```
Student(UNIV_ID,
    foreign key (univ_id) references Univ_person

Instructor(UNIV_ID,
    foreign key (univ_id) references Univ_person
```

Finally, since these are **overlapping** subtypes (note the **o** in the circle in its model), add **is_student** and **is_instructor** attributes to Univ_person's "base" table, whose domains are 'y'/'n' or 1/0 or some other suitable domain to indicate whether each Univ_person row is also a student, and whether each Univ_person row is also an instructor:

```
Univ_person(UNIV_ID, is_student, is_faculty,

Student(UNIV_ID,
    foreign key (univ_id) references Univ_person

Instructor(UNIV_ID,
    foreign key (univ_id) references Univ_person
```

The remaining single-valued attributes should be placed in the associated entity's "base" table:

```
Univ_person(UNIV_ID, is_student, is_faculty, last_name,
            first_name, campus_email,

Student(UNIV_ID, gpa, semester_matricd, semester_graduated,
    foreign key (univ_id) references Univ_person

Instructor(UNIV_ID, is_W4_on_file,
    foreign key (univ_id) references Univ_person
```

...which leaves entity Instructor's multi-valued attribute Semester_hired. But one just follows the normal design process for this at this point, creating a new table for this multi-valued attribute, whose primary key is a composite primary key consisting of the primary key of the entity's "base" table and the multi-valued attribute, and with that primary key from the entity's "base" table defined to be a foreign key back to the entity's "base" table:

```
Univ_person(UNIV_ID, is_student, is_faculty, last_name,
            first_name, campus_email)

Student(UNIV_ID, gpa, semester_matricd, semester_graduated)
    foreign key (univ_id) references Univ_person

Instructor(UNIV_ID, is_W4_on_file)
    foreign key (univ_id) references Univ_person

Instructor_Semesters(UNIV_ID, SEMESTER_HIRED)
```

```
    foreign key (univ_id) references Instructor
```

And so the new Instructor_Semesters table's foreign key univ_id references the **Instructor** table, since semester_hired is an attribute of the entity class Instructor.

If we had further relationships between these entities and other entities, we would handle each of them in turn. But as there are no other entities here, the database design/schema is complete (in terms of the table structures and relationships, anyway).

Comments on a Supertype/Subtypes variant: UNIONS

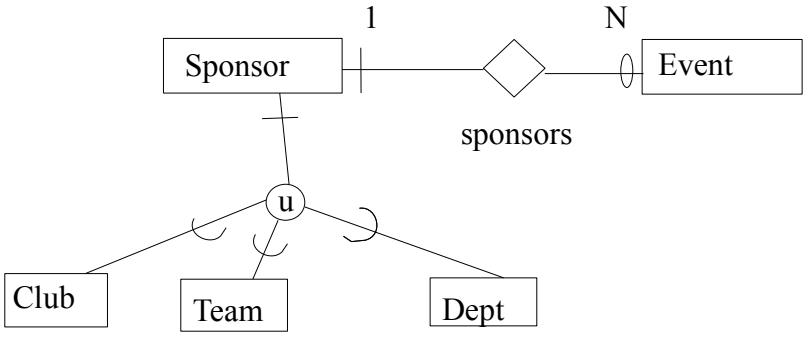
Ricardo (p. 384) mentions a potentially-useful supertype/subtypes variant that would be transformed into a design in a slightly different fashion.

Sometimes you have entity classes that are really quite distinct -- they have separate identifying attributes, for example, and perhaps no overlap in terms of attributes -- that nevertheless share a relationship. Ricardo calls this a Union -- it could be "drawn" like a supertype/subtype, except with a **U** on the circle. You'd notice here that the "subtype" entity classes feature identifying attributes, and that the "supertype" entity class has few or no attributes of its own, but participates in one or more relationships.

How would this be turned into a design differently? Since the "subtype" entities here really each have their own identity, then the primary keys for their "base" tables should be distinct from each other and from the "base" table for the "supertype". But the primary key for the "supertype's" "base" table would still be placed in each of the "subtypes" "base" tables, but *just* as a foreign key back to the "supertype's" "base" table (not as part of the "subtypes" "base" tables' primary keys). And, like for the overlapping subtypes case, you would add a *_type attribute to the "supertype" "base" table as well.

Univ-sponsor model example

As an example, consider a university-based scenario in which there are entity classes for campus clubs, campus teams, campus departments, and campus events, and a business rule noting that campus events must be sponsored by either a club, a team, or a department. You have a situation where each event needs to be related to a club *or* a team *or* a department, which is awkward to model until you consider this Union approach -- but with that approach, you can recognize that a Sponsor entity class would make things much cleaner:



```
Sponsor     Club        Team        Dept          Event
-------     --------    --------    ---------     -----
            CLUB_NUM    TEAM_CODE   DEPT_CODE     EVENT_NUM
            Club_Name   Sport       Dept_title    Event_title
            Is_active   Season      Office_num    Event_date
```

First we create "base" tables for each supertype and subtype entity class (and for Event, too, while we're at it):

```
Sponsor(
Club(
Team(
Dept(
Event(
```

Then we determine an appropriate primary keys for these. The **U** in the circle and the identifying attributes for the subtype entities lead us to give each subtype's "base" table its "own" primary key, and to create a primary key for Sponsor distinct from these primary keys. (We also determine an appropriate primary key for Event while we're at it).

```
Sponsor(SPONS_NUM,
Club(CLUB_NUM,
Team(TEAM_CODE,
Dept(DEPT_CODE,
Event(EVENT_NUM,
```

Because this is a Union situation, we will add Sponsor's primary key to the subtypes' "base" tables, but only as a foreign key (not also as their primary key, as would be the case for **d** or **o** supertype/subtype entity classes).

```
Sponsor(SPONS_NUM,
Club(CLUB_NUM, spons_num,
    foreign key (spons_num) references Sponsor
Team(TEAM_CODE, spons_num,
    foreign key (spons_num) references Sponsor
Dept(DEPT_CODE, spons_num,
    foreign key (spons_num) references Sponsor
Event(EVENT_NUM,
```

Because this is a Union situation, then as in the case of **disjoint** subtypes, we add a **spons_type** attribute to Sponsor's "base" table, whose domain will indicate whether a particular Sponsor is a Club or a Team or a Dept:

```
Sponsor(SPONS_NUM, spons_type,
Club(CLUB_NUM, spons_num,
    foreign key (spons_num) references Sponsor
```

```
Team(TEAM_CODE, spons_num,
    foreign key (spons_num) references Sponsor
Dept(DEPT_CODE, spons_num,
    foreign key (spons_num) references Sponsor
Event(EVENT_NUM,
```

All of the remaining attributes in the model happen to be single-valued, so each should be placed in the associated entity's "base" table. (If there were multi-valued attributes, they would each be the basis of a new table, as usual.)

```
Sponsor(SPONS_NUM, spons_type,
Club(CLUB_NUM, spons_num, club_name, is_active,
    foreign key (spons_num) references Sponsor
Team(TEAM_CODE, spons_num, sport, season,
    foreign key (spons_num) references Sponsor
Dept(DEPT_CODE, spons_num, dept_title, office_num,
    foreign key (spons_num) references Sponsor
Event(EVENT_NUM, event_title, event_date,
```

And of course we must remember to handle the relationship between Event and Sponsor -- since Sponsor is the parent in the Sponsor-sponsors-Event relationship, Sponsor's primary key will be added to Event's "base" table as a foreign key:

```
Sponsor(SPONS_NUM, spons_type)

Club(CLUB_NUM, spons_num, club_name, is_active)
    foreign key (spons_num) references Sponsor

Team(TEAM_CODE, spons_num, sport, season)
    foreign key (spons_num) references Sponsor

Dept(DEPT_CODE, spons_num, dept_title, office_num)
    foreign key (spons_num) references Sponsor

Event(EVENT_NUM, event_title, event_date, spons_num)
    foreign key (spons_num) references Sponsor
```

Since we've now taken care of all relationships and all entity classes, the database design/schema is complete (in terms of the table structures and relationships, anyway).

Handling Association Entities

Sometimes you find, in a model, an entity that *associates* two or more entities in a transaction "central" or important to that scenario that has attributes of its own, but doesn't seem to have any identifying attributes, usually because it really gets its identity from the two or more entities involved in that transaction. Such an entity is called an **association entity**. It isn't depicted any differently from other

entities within an ER diagram, except perhaps for its lack of any identifying attributes within its attribute list. (Remember, **entity attribute lists do not contain *any* relationship-related information at the model level**.)

So, why mention association entities here? Because there are two common options for handling their primary key at the database design stage:

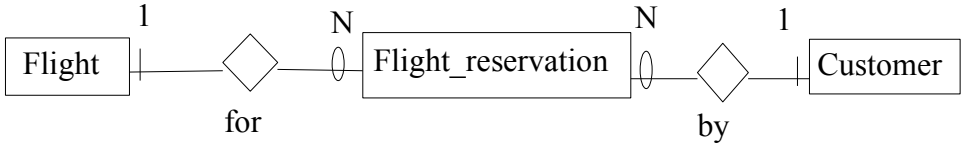
1) you might introduce a primary key into the "base" table for this entity (as is always an option); or,

2) if the entities that seem to give an instance of this association entity its "identity" are parents in 1:N relationships with this association entity, then those parents' primary keys that would already be added to the association entity's "base" table as foreign keys because of those 1:N relationships might also be made a composite primary key for the association entity's "base" table.

Why might you choose one option over another? If the transaction is very "central" to the scenario, you might lean toward giving the association entity's "base" table its "own" primary key. If that association entity is the parent in any 1:N relationships, that would be further encouragement to give the association entity's "base" table its "own" primary key, because single attribute foreign keys are just a little more convenient to work with. (Recall that, in the design process for a 1:N relationship, the primary key of the parent's "base" table is placed in the child's "base" table as a foreign key. You can have multi-attribute foreign keys, but they aren't very common in practice.) But for an association entity that is more minor, not involved in other relationships, it would also be a reasonable choice to simply go with option 2), with the two-attribute primary key. The resulting table looks very much like an intersection table with one or more extra attributes.

Flight model example

Imagine a scenario such as the following:




```
Flight          Flight_reservation        Customer
-----------     ---------------------     -------------
FLIGHT_NUM      Date_made                 CUST_NUM
Flight_date                               Cust_lname
Exp_departure                             Cust_fname
Exp_arrival
Actl_departure
Actl_arrival
```

Using Option 1) above, you would end up with the tables:

```
Flight(FLIGHT_NUM, flight_date, exp_departure, exp_arrival,
       actl_departure, actl_arrival)
```

```
Customer(CUST_NUM, cust_lname, cust_fname)

Flight_reservation(RES_NUM, date_made, flight_num, cust_num)
    foreign key (flight_num) references flight,
    foreign key (cust_num) references customer
```

Using Option 2) above, you would end up with the tables:

```
Flight(FLIGHT_NUM, flight_date, exp_departure, exp_arrival,
       actl_departure, actl_arrival)

Customer(CUST_NUM, cust_lname, cust_fname)

Flight_reservation(FLIGHT_NUM, CUST_NUM, date_made)
    foreign key (flight_num) references flight,
    foreign key (cust_num) references customer
```

Handling Weak Entities

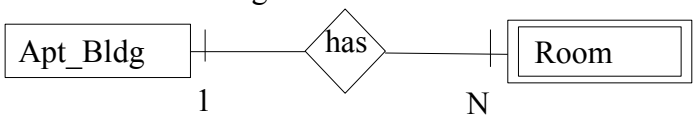
The design process for weak entities is really mostly a special case of handling 1:1 or 1:N relationships. (Since weak entities tend to be, by definition, identity-dependent upon the entity class they are related to, it would be odd to have one involved in an N:M relationship.)

Just following the normal "rules" for handling 1:N and 1:1 relationships, you will usually end up putting the primary key of the entity's "base" table that this weak entity is dependent upon into the weak entity's "base" table as a foreign key -- in the 1:N case, the weak entity is usually the child, and in the 1:1 case, the weak entity usually must be involved in the relationship.

So, why mention this at all? Because, since a weak entity is so dependent on another entity, it is quite common for the weak entity's "base" table to have a primary key based on that other entity's "base" table's primary key. If the relationship between the weak entity and the other entity is 1:1, and the weak entity doesn't have suitable identifying attributes, then the other entity's "base" table's primary key that is inserted into the weak entity's "base" table to serve as a foreign key may also be used as the primary key of the weak entity's "base" table. And if the relationship between the weak entity and the other entity is 1:N, then the other entity's "base" table's primary key that is inserted into the weak entity's "base" table to serve as a foreign key may also be used *as part of* the primary key of the weak entity's "base" table, along with some other attribute or attributes of the weak entity. (Since, in the 1:N case, one instance of the parent entity may be related to several children instances of the weak entity, using just the parent's primary key as the child weak entity's primary key would not be sufficient.)

Apartment model example

For example, consider the following:



```
Apt_bldg            Room
---------           ---------
BLDG_NUM            Room_num
Bldg_name           Num_bedrooms
Bldg_street_addr    Num_baths
```

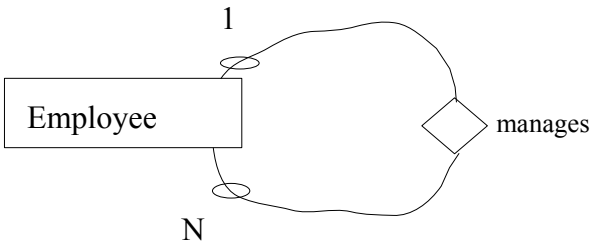
Using the above guidelines, this ER diagram could result in the tables:

```
Apt_bldg(BLDG_NUM, bldg_name, bldg_street_addr)

Room(BLDG_NUM, ROOM_NUM, num_bedrooms, num_baths)
    foreign key (bldg_num) references Apt_bldg
```

Handling Recursive Relationships

Recall that a recursive relationship is a relationship line that begins and ends at the same entity class:



Some other examples of recursive relationships might be course-is-a-prerequisite-of-course, student-rooms-with-student, and employee-is-spouse-of-employee.

Again, really you handle this like any other relationship, based on whether it is 1:1, 1:N, or N:M. The only additional design-stage consideration is that a primary keys placed in a table as a foreign key may need to be given a different name than the original primary key, to avoid having two attributes with the same name (which is of course forbidden in a relation!) And so you will need to use the "full" foreign key syntax necessary when the foreign key name is different than the primary key attribute it references, including the referenced attribute.

Employee-manager model example

Consider the above ER model. Based on the 1:N relationship, you know you would put the primary key of the parent's/the 1 side's "base" table into the child's/N side's "base" table as a foreign key. But, say that the Employee "base" table's primary key is EMPL_NUM.

```
Employee(EMPL_NUM,
```

Here, the "base" table of the parent and the child in this 1:N relationship are the same table, the Employee table! So you cannot add a second attribute named empl_num to Employee!

However, you can add an attribute whose name is based on the recursive relationship -- here, for example, mgr_num. And you can make this new attribute a foreign key by using the "full" foreign key

syntax, where you specify the attribute name in the referenced table as well:

```
Employee(EMPL_NUM, mgr_num,
    foreign key (mgr_num) references Employee(empl_num)
```

(Quick population note: I don't really care what order you list your tables in your database design/schema. However, when you then create a SQL script to create your tables based on your database design/schema, note that the table-creation order does matter at that point -- a table with a foreign key cannot be created unless the table it references already exists, to meet the requirements of referential integrity. So, you will need to create tables without foreign keys first, and then those that reference those tables, and then those that reference those tables, and so on.

There is another referential integrity implication directly related to situations such as the Employee table above. As you cannot create a table with a foreign key until the table it references exists, so you cannot insert a row containing a foreign key unless there is a row with that foreign key's value in the corresponding parent table. But child and parent are the same table for the Employee table! So, how does one handle this? By inserting the first row into Empl with a null value for mgr_num -- a foreign key can be null (unless it is explicitly declared to be **not null**). Then employees who have that first row's employee as their manager can be inserted, then those managed by those employees, and so on.

If every employee has a manager, and employee A manages employee B, and employee B manages employee A, then you can insert a row for employee A with a mgr_num of null, then insert a row for employee B with A's mgr_num, and then use the SQL UPDATE command to update just the mgr_num attribute of A's row to now have a mgr_num of B, which is now just fine, as B's row exists.

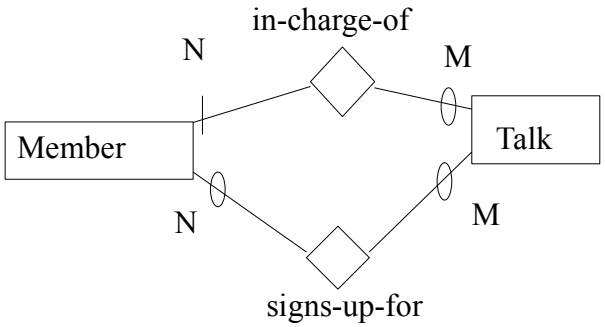
Happily, in this kind of case, the awkwardness tends to only be at the beginning -- over time, new rows usually reference already-existing rows, and there is little problem in day-to-day operations. After all, new employees tend to be managed by already-existing employees.)

Handling multiple relationships between the same entities

As in the recursive case just described, the main design issue in this case is that you may need to be careful how you name things -- you may need to rename a primary key inserted into another "base" table as a foreign key, and if you need to do so that foreign key should be named in a way related to the relationship, and should use the "full" foreign key syntax required when the foreign key and the primary key it references have different names. And in the case of two N:M relationships between the same two entities, you just need to make sure you give the two intersection tables names that make their meanings (which relationships they represent) clear.

Member-talk model example

For example, consider the following:



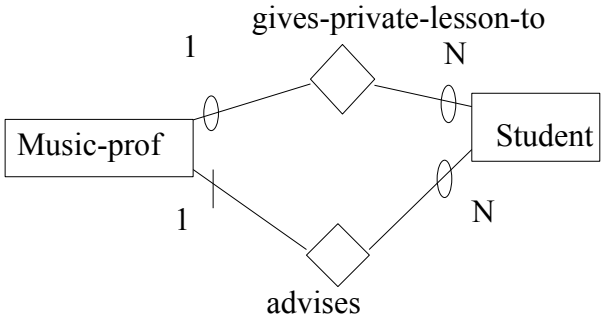
(Note that this is a slightly different scenario than that depicted in the earlier homework -- here, more than one member can share the task of being in charge of a single talk, and a talk doesn't have to have any members signed up...)

When you find you need two intersection tables for the two N:M relationships between Member and Talk, you just have to remember that it would be a poor choice to call either of those tables Member_Talk or Talk_Member -- the purpose of such a table would not be clear. But table names such as In_charge and Sign_up could work just fine:

```
Member(MEM_NUM, ...
Talk(TALK_NUM, ...
In_charge(MEM_NUM, TALK_NUM)
    foreign key (mem_num) references Member,
    foreign key (talk_num) references Talk
Sign_up(MEM_NUM, TALK_NUM)
    foreign key (mem_num) references Member,
    foreign key (talk_num) references Talk
```

Music-prof model example

You might get the dueling-foreign-key-names situation if, for example, there are two 1:N relationships between two entities, and the parent is the same in both:



So, you will want to put Music-prof's "base" table's primary key into the Student's "base" table twice, once for the gives-private-lesson-to relationship, and once for the advises relationship. Say that the

Music-prof "base" table's primary key is PROF_NUM. I would argue that, in this case, it would be confusing to use prof_num as the name of the foreign key in Student's "base" table for either relationship -- priv_teacher_num and advisor_num might be less confusing names, in this case:

```
Music_prof(PROF_NUM, ...)
Student(STU_NUM, ..., priv_teacher_num, advisor_num)
    foreign key (priv_teacher_num) references Music_prof(prof_num),
    foreign key (advisor_num) references Music_prof(prof_num)
```

What if the 1's are on *different* sides in two 1:N relationships between two entities? Then there is no name collision, and indeed no problem in the database design/schema stage -- say your tables are A and B. A will have B's primary key as a foreign key, and B will have A's primary key as a foreign key. Where a kluge will be necessary would be at table creation time -- since referential integrity constraints require that a table cannot be created until the table it references is created, this kind of **circular** situation is awkward. There is a solution:

* create table A without the foreign key;
* create table B with the foreign key referencing A;
* now use SQL's ALTER command to change the structure of table A, adding in the foreign key referencing B.

And at table population, a row may need to be added to table A with a null foreign key, then rows can be added to table B referencing that row in A, and then A's row can be UPDATE'd to now reference a row in B.

Again, happily, in this kind of case, the awkwardness tends to only be at the beginning -- over time, new rows usually reference already-existing rows, and there is little problem in day-to-day operations.