Mapping Data Models to a Relational Database

- Time to dovetail two prior topics: data modeling (E-R, UML) and the relational data model (structure, algebra)

- To recap, this is how they relate:

  ◇ The data model (E-R, UML) specifies the conceptual schema of your database

  ◇ This conceptual schema needs to be implemented as a logical internal schema in terms of your chosen database software — in our case, this logical internal schema would conform to the relational data model

Ground Rules

- Entity sets or classes in the conceptual schema become relations in the logical internal schema

- Relationship sets or associations in the conceptual schema become relations in the logical internal schema

- Attributes "carry over" directly into their corresponding relations

  ◇ Primary keys in the conceptual schema are also primary keys in the logical internal schema

Mapping Relationship Sets/ Associations

Relationship sets or associations become relations with the following attributes:

- Union of the primary keys for each entity set or class participating in the relationship set/association — these become foreign keys into the relations that represent the entity set or class

- Additional attributes, if the relationship itself has them (or, in UML terminology, if there is an association class in the conceptual schema)

Cardinality Considerations

The cardinality and "n-arity" of a relationship set or association influences the primary key of the resulting mapped relation

Cardinality	Primary Key
binary many-to-many	union of all primary key attributes
binary one-to-one	primary key of either entity set or class (take your pick)
binary many-to-one or binary one-to-many	primary key of the "many" side of the relationship
n-ary many-to-many	union of all primary key attributes
n-ary one-to-many (-to-many-to-many…)	union of all primary key attributes except for the primary key of the "one" entity set or class

Possible Reductions

Depending on the specifics of your relationship sets/ associations, the "cookbook" mapping may be simplified

- *Total participation*: if the "many" side of a 1-to-many or many-to-1 relationship *always* participates in this relationship — e.g., the "many" side never "stands alone" — then the intermediate "relationship relation" may be combined with the "many" relation

- *One-to-one*: the intermediate "relationship relation" may also be dropped in this case, and combined with either entity set/class relation

- Foreign key constraints transfer to the combined relations, except for the one between the combined relations themselves

- The primary key of the combined relation is the primary key of the entity set/class relation that was combined

- In the absence of total participation, this schema combination may still be performed by allowing the foreign key in the combined relation to be *null* — with the usual caveats and cautions that accompany the use of *null* values

Mapping Weak Entity Sets (E-R) and Compositions (UML)

- Weak entity sets in E-R and composition in UML are two conceptual frameworks that don't have exact equivalents, but they *do* have similar enough structure that their relational mapping approaches correspond

- The idea behind both of these constructs is that there is an "owning" ("pwning"?) or "master" strong entity set or container class, upon which the weak entity set or contained/component class completely depends for its existence and meaning

- When mapped to the relational data model, these ideas result in the same structure:

  - The weak entity set or component class maps into a relation whose attributes are the union of the "owned" entity/class and the primary key of the "owning" entity/class

  - The primary key of the resulting relation is the primary key of the "owning" entity/class combined with the *discriminator* of the "owned" entity/class (i.e., the set of attributes that differentiates individual weak entities among their "siblings")

- Note how, in the end, this is just a special case of binary, 1-to-many relationship mapping and reduction

Mapping Composite and Multivalued Attributes

Composite and multivalued attributes (available in both E-R and UML) are explicitly *not* supported by the relational data model — recall that attributes in relations are defined as atomic:

- Composite attributes in the relational model simply aren't — they aren't allowed, period — so anything modeled as a composite attribute just breaks down into its atomic sub-attributes (e.g., an *address* consisting of a *street*, *city*, *state*, and *zip* would simply break down into those four attributes at the relational level)

- Multivalued attributes end up in their own relation

  ◇ The relation schema is the union of the primary key of the entity/class that has the multivalued attribute and a single instance of the attribute itself

  ◇ The primary key of this schema is usually the entire tuple, though a smaller primary key may exist if the multivalued attribute were also composite

  ◇ The attributes that were taken from the primary key of the "owning" entity/class comprise a foreign key into that entity/class's relation — but at this point, you knew that already, right?

- Note how this is just another incarnation of the "owner/owned" relation pattern

Mapping Generalization

- Two methods for mapping generalization (a.k.a. superclasses and subclasses) onto relations, depending on the characteristics of the generalization

- Caveat here is: we mustn't forget that the relational data model does not have "native" support for generalization in the first place, so neither of these mapping techniques is completely airtight

  ◇ While they can represent generalization, they cannot strictly *enforce* it — each approach has loopholes that a natively object-oriented view wouldn't allow

  ◇ Neither the relational algebra or SQL have explicit support for generalization/ specialization, so you have to "build those in" with explicit joins or other expressions

1. Create a relation for the generalizing entity or class, with one relation each for the specializations

   ◇ The specialization relations have the primary key of the generalization relation plus the additional attributes from the specialization entity/class

   ◇ Note that this implementation can't inherently enforce "single inheritance" — that is, an entity or instance may be a member of more than one specialization relation

2. If the generalization is *disjoint* (i.e. "single inheritance") and *complete* (i.e. the generalization is *abstract* — every member of the generalization must also be a specialization), then you *may* be able to ditch the generalized relation — just create a relation for each specialization, with *each* of these relations having the primary key and attributes of the generalization

   ◇ Implementation issue: the primary keys have to be unique across relations, and the only way to enforce that is to *still* create a generalization relation consisting only of the primary key attributes, with foreign key constraints on the specialization relations

Mapping Aggregation

- Aggregations are ultimately special-case relationship sets or associations, and so they also map into their own relations, consisting of the primary keys of the entity sets/classes that participate in the aggregation, with the corresponding foreign key constraints

  ◇ In E-R, the aggregation is captured the relationship set that "enters" the aggregated (boxed) region, so no additional relations are created explicitly to express the aggregation; in UML, aggregation has its own special association notation, which is what "maps" into the final aggregation relation

- Then, add the primary key of the *aggregator* entity set/class to that aggregation relation, which of course is a foreign key to that relation as well