Extensions to E-R Model

- Basic E-R model is good for many uses
- Several extensions to the E-R model for more advanced modeling
  - Generalization and specialization
  - Aggregation
- These extensions can also be converted to the relational model
  - Introduces a few more design choices
- Will only discuss specialization today
  - See book §7.8.5 for details on aggregation (material will be included with Assignment 5 too)
Specialization

- An entity-set might contain distinct subgroups of entities
  - Subgroups have some different attributes, not shared by the entire entity-set
- E-R model provides **specialization** to represent such entity-sets
- **Example**: bank account categories
  - Checking accounts
  - Savings accounts
  - Have common features, but also unique attributes
Generalization and Specialization

- Generalization: a “bottom up” approach
  - Taking similar entity-sets and unifying their common features
  - Start with specific entities, then create generalizations from them

- Specialization: a “top down” approach
  - Creating general purpose entity-sets, then providing specializations of the general idea
  - Start with the general notion, then refine it

- Terms are basically equivalent
  - Book refers to generalization as the overarching concept
Bank Account Example

- Checking and savings accounts both have:
  - account number
  - balance
  - owner(s)

- Checking accounts also have:
  - overdraft limit and associated overdraft account
  - check transactions

- Savings accounts also have:
  - minimum balance
  - interest rate
Bank Account Example (2)

- Create entity-set to represent common attributes
  - Called the **superclass**, or higher-level entity-set
- Create entity-sets to represent specializations
  - Called **subclasses**, or lower-level entity-sets
- Join superclass to subclasses with hollow-head arrow(s)

- **account**
  - **acct_id**
  - **balance**

- **checking**
  - **overdraft_limit**

- **savings**
  - **min_balance**
  - **interest_rate**
Inheritance

- Attributes of higher-level entity-sets are inherited by lower-level entity-sets
- Relationships involving higher-level entity-sets are also inherited by lower-level entity-sets!
  - Lower-level entity-sets can also participate in their own relationship-sets, separate from higher-level entity-set
- Usually, entity-sets inherit from one superclass
  - Entity-sets form a hierarchy
- Can also inherit from multiple superclasses
  - Entity-sets form a lattice
  - Introduces many subtle issues, of course
Specialization Constraints

- Can an account be both a savings account and a checking account?
- Can an account be neither a savings account nor a checking account?
- Can specify constraints on specialization
  - Enforce what “makes sense” for the enterprise
Disjointness Constraints

- “An account cannot be both a checking account and a savings account.”
- An entity may belong to at most one of the lower-level entity-sets
  - Must be a member of checking, or a member of savings, but not both!
  - Called a “disjointness constraint”
  - A better way to state it: a disjoint specialization
- If an entity can be a member of multiple lower-level entity-sets:
  - Called an overlapping specialization
How the arrows are drawn indicates whether the specialization is disjoint or overlapping

Bank account example:
- One arrow split into multiple parts indicates a disjoint specialization
- An account may only be a checking account, or a savings account, not both
Another example from the book:

- Specialization hierarchy for people at a university

- Multiple separate arrows indicates an overlapping specialization
  - A person can be an employee of the university and a student

- One arrow split into multiple parts is a disjoint specialization
  - An employee can be an instructor or a secretary, but not both
Completeness Constraints

- “An account must be a checking account, or it must be a savings account.”
- Every entity in higher-level entity-set must also be a member of at least one lower-level entity-set
  - Called **total** specialization
- If entities in higher-level entity-set aren’t required to be members of lower-level entity-sets:
  - Called **partial** specialization
- **account** specialization is a **total** specialization
Completeness Constraints (2)

- Default constraint is \textit{partial} specialization
- Specify total specialization constraint by annotating the specialization arrow(s)
- Updated bank account diagram:
Completeness Constraints (3)

- Same approach with overlapping specialization
- Example: people at a university
  - Every person is an employee or a student
  - Not every employee is an instructor or a secretary
- Annotate arrows pointing to person with “total” to indicate total specialization
  - Every person must be an employee, a student, or both
Account Types?

- Our bank schema so far:

```
account
 acct_id
  balance
```

- How to tell whether an account is a checking account or a savings account?
  - No attribute indicates type of account
Membership Constraints

- **Membership constraints** specify which lower-level entity-sets each entity is a member of
  - e.g. which accounts are checking or savings accounts
- **Condition-defined** lower-level entity-sets
  - Membership is specified by a predicate
  - If an entity satisfies a lower-level entity-set’s predicate then it is a member of that lower-level entity-set
  - If all lower-level entity-sets refer to the same attribute, this is called **attribute-defined** specialization
    - e.g. account could have an account_type attribute set to “c” for checking, or “s” for savings
Membership Constraints (2)

- Entities may also simply be assigned to lower-level entity-sets by a database user
  - No explicit predicate governs membership
  - Called user-defined membership
- Generally used when an entity’s membership could change in the future
Would also create relationship-sets against various entity-sets in hierarchy
- associate customer with account
- associate check weak entity-set with checking
Mapping to Relational Model

- Mapping generalization/specialization to relational model is straightforward
- Create relation schema for higher-level entity-set
  - Including primary keys, etc.
- Create schemas for lower-level entity-sets
  - Subclass schemas include superclass’ primary key attributes!
  - Primary key is same as superclass’ primary key
    - Subclasses can also contain their own candidate keys!
    - Enforce these candidate keys in implementation schema
  - Foreign key reference from subclass schemas to superclass schema, on primary-key attributes
Mapping Bank Account Schema

- **Schemas:**
  
  - `account(acct_id, acct_type, balance)`
  - `checking(acct_id, overdraft_limit)`
  - `savings(acct_id, min_balance, interest_rate)`

  - Could use **CHECK** constraints on SQL tables for membership constraints, other constraints (although it may be expensive)
Alternative Schema Mapping

- If specialization is disjoint and complete, could convert only lower-level entity-sets to relational schemas
  - Every entity in higher-level entity-set also appears in lower-level entity-sets
  - Every entity is a member of exactly one lower-level entity-set
- Each lower-level entity-set has its own relation schema
  - All attributes of superclass entity-set are included on each subclass entity-set
  - No relation schema for superclass entity-set
Schemas, take 2:

checking(\text{acct\_id}, \text{acct\_type}, \text{balance}, \text{overdraft\_limit})

savings(\text{acct\_id}, \text{acct\_type}, \text{balance}, \text{min\_balance}, \text{interest\_rate})
Alternative Account Schema (2)

- Alternative schemas:
  
  **checking**((acct_id, acct_type, balance, overdraft_limit))
  **savings**((acct_id, acct_type, balance, min_balance, interest_rate))

- Problems?
  
  - Enforcing uniqueness of account IDs!
  - Representing relationships involving both kinds of accounts

- Can solve by creating a simple relation:
  
  **account**((acct_id))
  
  - Contains *all* valid account IDs
  - Relationships involving accounts can use account
  - Need foreign key constraints again…
Generating primary key values is actually the easy part

Most databases provide sequences
- A source of unique, increasing INTEGER or BIGINT values
- Perfect for primary key values
- Multiple tables can use the same sequence for their primary keys

PostgreSQL example:
```
CREATE SEQUENCE acct_seq;

CREATE TABLE checking (  
    acct_id INT PRIMARY KEY DEFAULT nextval('acct_seq');
    ...  
);

CREATE TABLE savings (  
    acct_id INT PRIMARY KEY DEFAULT nextval('acct_seq');
    ...  
);
```
Alternative Schema Mapping

- Alternative mapping has serious drawbacks
  - Doesn’t actually give many benefits in general case

- Fewer drawbacks if:
  - Total, disjoint specialization
  - No relationships against superclass entity-set

- If specialization is overlapping, some details will be stored multiple times
  - Unnecessary redundancy, and consistency issues

- Also limits future schema changes
  - Should always think about this when creating schemas
Recap: Weak Entity-Set Example

- **account** schema:
  \[
  \text{account}(\text{account\_number}, \text{balance})
  \]

- **check** schema:
  - Discriminator is \text{check\_number}
  - Primary key for check is: \((\text{account\_number, check\_number})\)
  \[
  \text{check}(\text{account\_number}, \text{check\_number}, \text{check\_date}, \text{recipient}, \text{amount}, \text{memo})
  \]
Relationship between weak entity-set and strong entity-set doesn’t need represented separately
- Many-to-one relationship
- Weak entity-set has total participation
- Weak entity-set’s schema already captures the identifying relationship

Can apply this technique to other relationship-sets:
- One-to-many mapping, with total participation on the “many” side

<table>
<thead>
<tr>
<th>account</th>
</tr>
</thead>
<tbody>
<tr>
<td>account_number</td>
</tr>
<tr>
<td>balance</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_number</td>
</tr>
<tr>
<td>check_date</td>
</tr>
<tr>
<td>recipient</td>
</tr>
<tr>
<td>amount</td>
</tr>
<tr>
<td>memo</td>
</tr>
</tbody>
</table>
```
Entity-sets $A$ and $B$, relationship-set $AB$
- Many-to-one mapping from $A$ to $B$
- $A$’s participation in $AB$ is total

Generates relation schemas $A$, $B$, $AB$
- Primary key of $A$ is $\text{primary\_key}(A)$
- Primary key of $AB$ is also $\text{primary\_key}(A)$
  - ($A$ is on “many” side of mapping)
- $AB$ has foreign key constraints on both $A$ and $B$
- There is one relationship in $AB$ for every entity in $A$

Can combine $A$ and $AB$ relation schemas
- Primary key of combined schema still $\text{primary\_key}(A)$
- Only requires one foreign-key constraint, to $B$
In this case, when relationship-set is combined into the entity-set, the entity-set’s primary key doesn’t change!

If A’s participation in AB is partial, can still combine schemas
- Must store null values for primary_key(B) attributes when an entity in A maps to no entity in B

If AB is one-to-one mapping:
- Can also combine schemas in this case
- Could incorporate AB into schema for A, or schema for B
- Don’t forget that AB has two candidate keys…
  - The combined schema must still enforce both candidate keys
Manager to worker mapping is one-to-many

Relation schemas were:

employee(employee_id, name)
works_for(employee_id, manager_id)

Could combine into:

employee(employee_id, name, manager_id)

(A very common schema combination)

Need to store *null* for employees with no manager
One-to-one mapping between customers and loans

- `customer(cust_id, name, street_address, city)`
- `loan(loan_id, amount)`
- `borrower(cust_id, loan_id)` – `loan_id` also a candidate key

Could combine `borrower` schema into `customer` schema or `loan` schema
- Does it matter which one you choose?
Participation of loan in borrower will be total
- Combining borrower into customer would require null values for customers without loans

Better to combine borrower into loan schema
- customer(cust_id, name, street_address, city)
- loan(loan_id, cust_id, amount)
- No null values!
Schema Combination Example (4)

- **Schema:**
  
  ```
  customer(cust_id, name, street_address, city)
  loan(loan_id, cust_id, amount)
  ```

- What if, after a while, we wanted to change the mapping cardinality?
  - Schema changes would be significant
  - Would need to migrate existing data to a new schema
Benefits of schema combination:
- Usually eliminates one foreign-key constraint, and the associated performance impact
  - Constraint enforcement
  - Extra join operations in queries
- Reduces storage requirements

Drawbacks of schema combination:
- May necessitate the use of null values to represent the absence of relationships
- Makes it harder to change mapping cardinality constraints in the future