CS 121: Relational Databases

The Entity-Relationship Model II

Agenda

N-ary relationships

Converting the E-R model to the relational model

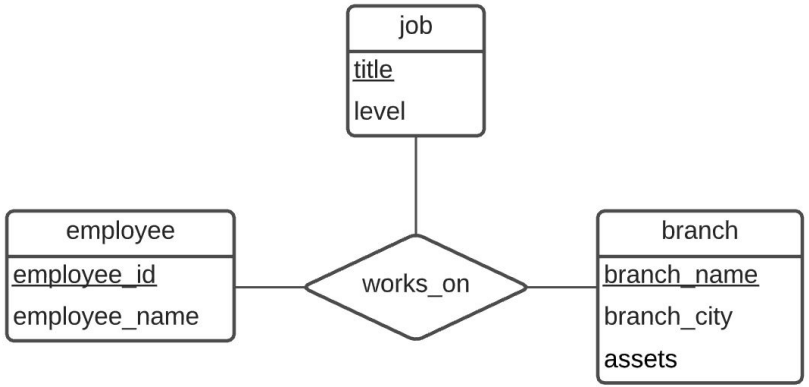
Specialization with superclasses and subclasses

Schema combinations for relationship-sets

N-ary Relationships

Can specify relationships of degree > 2 in E-R model

Example:



Employees are assigned to jobs at various branches (manager, bank teller, etc.)

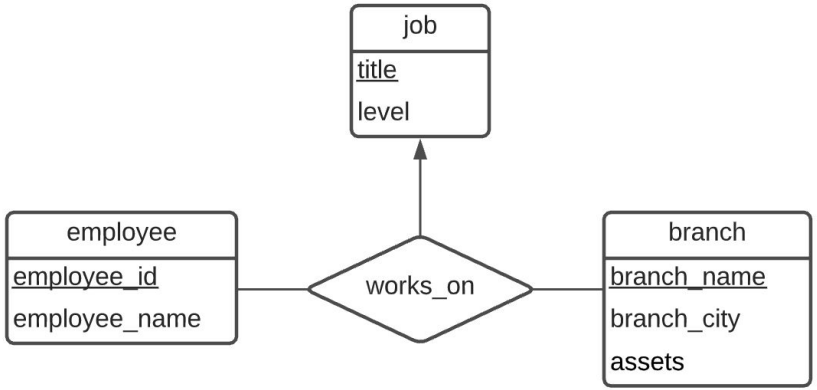
Many-to-many mapping: any combination of employee, job, and branch is allowed

An employee can have several jobs at one branch

N-ary Mapping Cardinalities

Can specify *some* mapping cardinalities on relationships with degree > 2

Each combination of employee and branch can only be associated with one job:



In other words, each employee can have only one job at each branch

N-ary Mapping Cardinalities (2)

For degree > 2 relationships, we only allow at most one edge with an arrow

Reason: multiple arrows on N-ary relationship-set is ambiguous

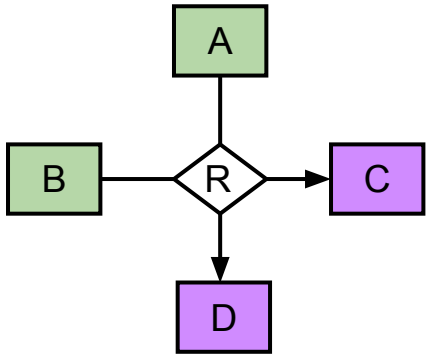
- (several meanings have been defined for this in the past)

Consider relationship-set R associating entity-sets $A_1, A_2, \ldots, A_n$

- No arrows on edges $A_1, \ldots, A_i$
- Arrows are on edges to $A_{i+1}, \ldots, A_n$

Meaning 1 (the simpler one):

- A particular combination of entities in $A_1, \ldots, A_i$ can be associated with at most one set of entities in $A_{i+1}, \ldots, A_n$
- Primary key of R is union of primary keys from set $\{ A_1, A_2, \ldots, A_i \}$



Primary key for R:
(a, b)

Meaning 1:
(a_1, b_2, c_3, d_5)
(a_3, b_4, c_3, d_2)
~~(a_3, b_4, c_6, d_9)~~
…

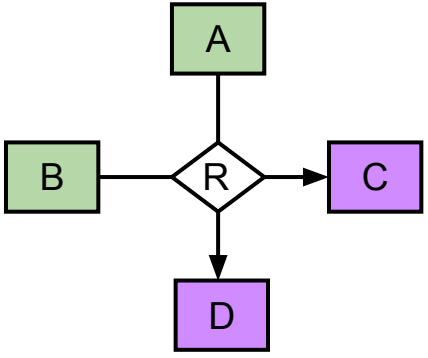
N-ary Mapping Cardinalities (3)

Consider same relationship-set R associating entity-sets $A_1, A_2, \ldots, A_n$

- No arrows on edges $A_1, \ldots, A_i$; arrows on edges to $A_{i+1}, \ldots, A_n$

Meaning 2 (the insane one):

- For each entity-set A_k $(i < k \leq n)$, a particular combination of entities from *all other* entity-sets can be associated with at most one entity in A_k
- R has a candidate key for each arrow in N-ary relationship-set
- For each k $(i < k \leq n)$, another candidate key of R is union of primary keys from entity-sets $\{ A_1, A_2, \ldots, A_{k-1}, A_{k+1}, \ldots, A_n \}$



Two candidate keys:
(a, b, c), (a, b, d)

Meaning 2:
(a_1, b_2, c_3, d_5)
(a_3, b_4, c_3, d_2)
(a_1, b_2, c_1, d_4)
(a_3, b_4, c_5, d_7)
~~(a_1, b_2, c_3, d_6)~~
~~(a_3, b_4, c_8, d_2)~~
…

All disallowed by meaning 1!

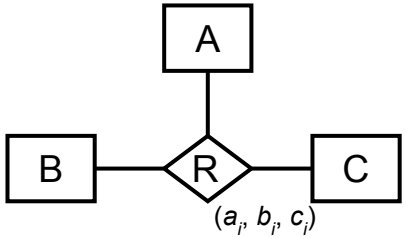
N-ary Mapping Cardinalities (4)

Both interpretations of multiple arrows have been used in books and papers…

If we only allow one edge to have an arrow, both definitions are equivalent

- The ambiguity disappears

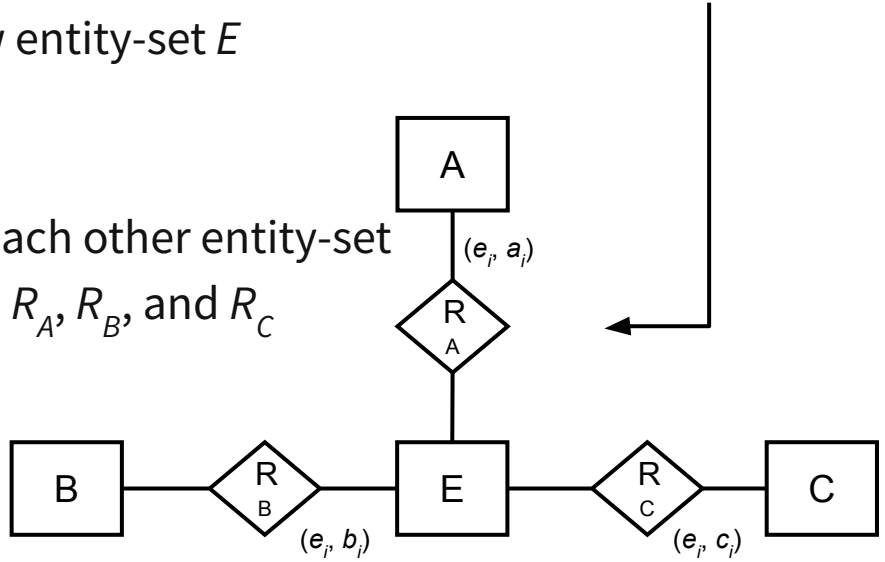
Binary vs. N-ary Relationships



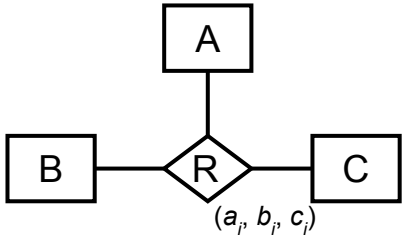
Often have only binary relationships in DB schemas

For degree > 2 relationships, *could* replace with binary relationships

- Replace N-ary relationship-set with a new entity-set *E*
  - Create an identifying attribute for *E*
  - e.g. an auto-generated ID value
- Create a relationship-set between *E* and each other entity-set
- Relationships in *R* must be represented in R_A, R_B, and R_C



Binary vs. N-ary Relationships (2)



Are these representations identical?

Example: Want to represent a relationship between entities a_5, b_1 and c_2

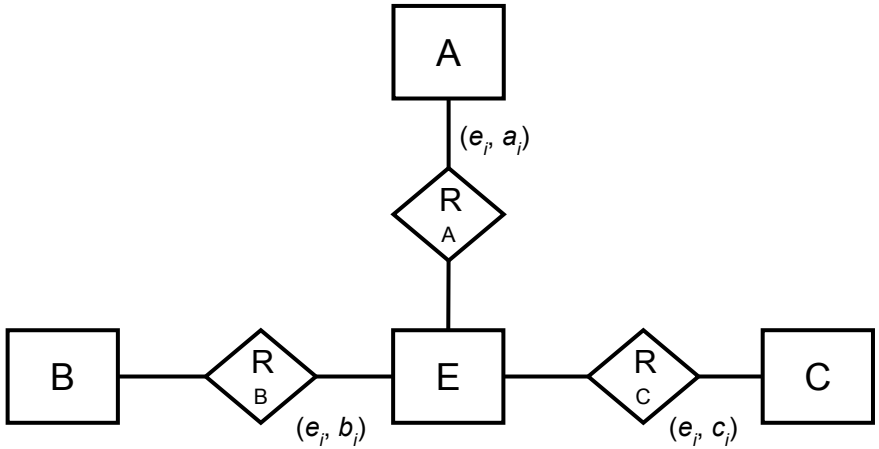
- How many relationships can we actually have between these three entities?

Ternary relationship set:

- Can only store one relationship between a_5, b_1 and c_2, due to primary key of R (a_i, b_i, c_i)

Alternate approach:

- Can create <u>many</u> relationships between these entities, due to the entity-set E !
  - $(a_5, e_1), (b_1, e_1), (c_2, e_1)$
  - $(a_5, e_2), (b_1, e_2), (c_2, e_2)$
  - …
- Can't constrain in exactly the same ways



Binary vs. N-ary Relationships (3)

So, sometimes a relationship is best represented with a ternary relationship

However, using binary relationships is sometimes more intuitive for particular designs

Example: office-equipment inventory database

- Consider ternary relationship-set *inventory*, associating *department*, *machine*, and *vendor* entity-sets

What if vendor info is unknown for some machines?

- For ternary relationship, must use *null* values to represent missing vendor details
- With binary relationships, can simply not have a relationship between *machine* and *vendor*

For cases like these, use binary relationships

- If it makes sense to model as separate binary relationships, do it that way!

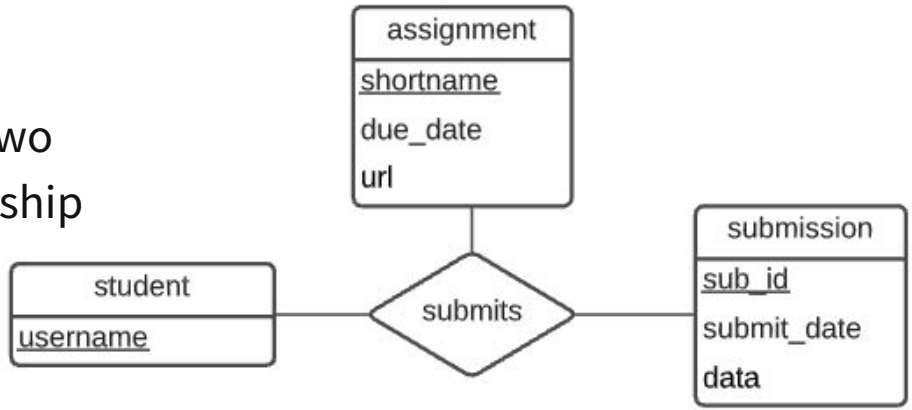
Course Database Example

What about this scenario?

- Ternary relationship between *student*, *assignment*, and *submission*
- Need to allow multiple submissions for a particular assignment, from a particular student

In this case, it could make sense to represent as a ternary relationship

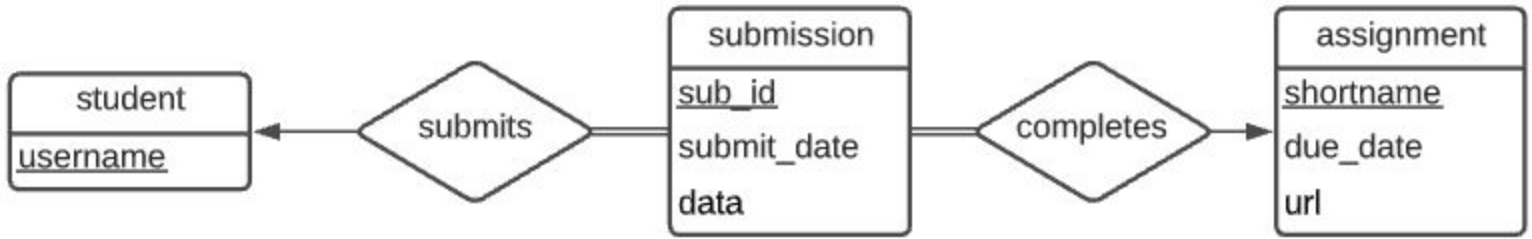
- Doesn't make sense to have only two of these three entities in a relationship



Course Database Example (2)

Other ways to represent students, assignments and submissions?

Can also represent as two binary relationships



Note the total participation constraints!

- Required to ensure that every *submission* has:
  - an associated *student*, and
  - an associated *assignment*

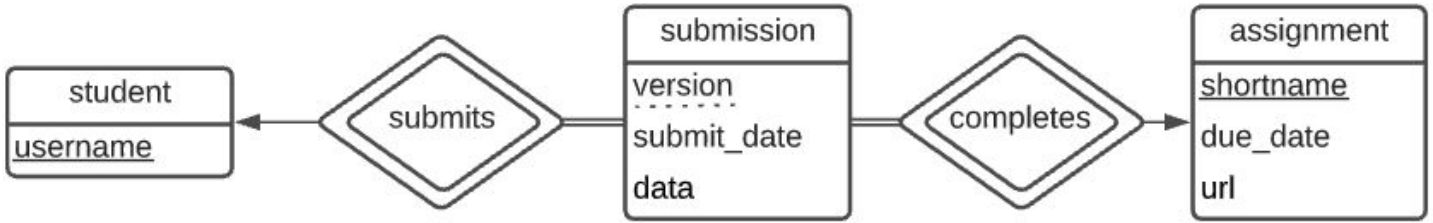
Also, two one-to-many constraints

If we wanted to support partner assignments, how would we change this?

Course Database Example (3)

Could even make *submission* a weak entity-set

- Both *student* and *assignment* are identifying entities!



Discriminator for *submission* is version number

Primary key for *submission* ?

- Union of primary keys from all owner entity-sets, plus discriminator
- (*username*, *shortname*, *version*)

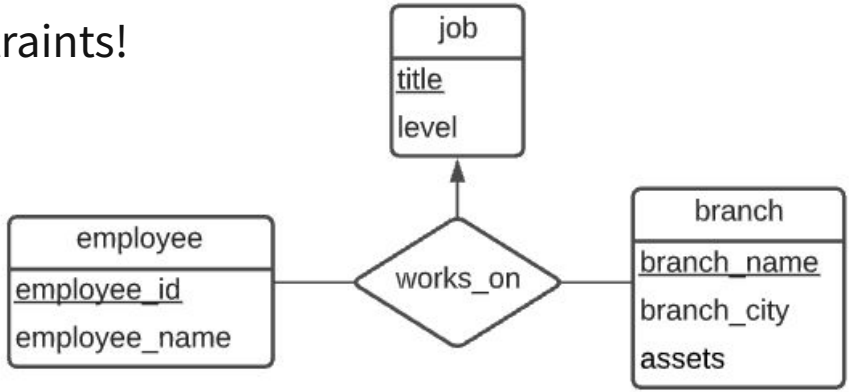
What is an advantage of this design?

Binary vs. N-ary Relationships

Sometimes ternary relationships are best

- Clearly indicates all entities involved in relationship
- Only way to represent certain constraints!

Bank jobs example:



Each (*employee*, *branch*) pair can have only one job

Simply <u>cannot</u> construct the same constraint using only binary relationships

- *(Reason is related to issue identified on slide 9)*

E-R Model and Real Databases

For E-R model to be useful, need to be able to convert diagrams into an implementation schema

Turns out to be very easy to do this!

- Big overlaps between E-R model and relational model
- Biggest difference is E-R composite/multivalued attributes, vs. relational model atomic attributes

Three components of conversion process:

1. Specify schema of the relation itself
2. Specify primary key on the relation
3. Specify any foreign key references to other relations

Strong Entity-Sets

Consider strong entity-set E with attributes $a_1, a_2, \ldots, a_n$

- Assume simple, single-valued attributes for now

Create a relation schema with same name E, and same attributes $a_1, a_2, \ldots, a_n$

Primary key of relation schema is same as primary key of entity-set

- Strong entity-sets require no foreign keys to other things

Every entity in E is represented by a tuple in the corresponding relation

Entity-Set Examples

Geocache location E-R diagram:

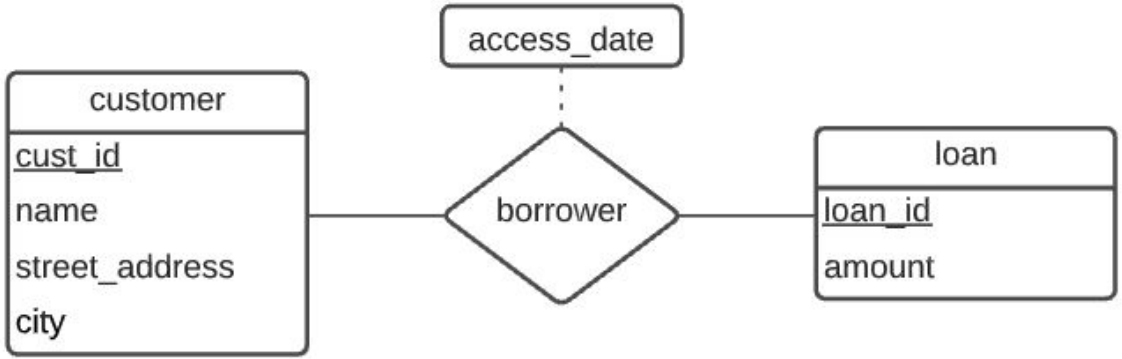
- Entity-set named *location*

Convert to relation schema:

location(*latitude*, *longitude*, *description*, *last_visited*)

Entity-Set Examples (2)

E-R diagram for customers and loans:



Convert *customer* and *loan* entity-sets:

customer(*cust_id*, *name*, *street_address*, *city*)

loan(*loan_id*, *amount*)

Relationship-Sets

Consider relationship-set R

- For now, assume that all participating entity-sets are strong entity-sets
- $a_1, a_2, \ldots, a_m$ is the union of all participating entity-sets' primary key attributes
- $b_1, b_2, \ldots, b_n$ are descriptive attributes on R (if any)

Relational model schema for R is:

- $\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$

$\{a_1, a_2, \ldots, a_m\}$ is a superkey, but not necessarily a candidate key

- Primary key of R depends on R's mapping cardinality

Relationship-Sets: Primary Keys

For binary relationship-sets:

- e.g. between strong entity-sets *A* and *B*
- If many-to-many mapping:
  - Primary key of relationship-set is union of all entity-set primary keys (from last time)
  - *primary_key*(*A*) ∪ *primary_key*(*B*)
- If one-to-one mapping:
  - Either entity-set's primary key is acceptable
  - *primary_key*(*A*), or *primary_key*(*B*)
  - Enforce both candidate keys in DB schema (e.g. with **`UNIQUE`**)!

Relationship-Sets: Primary Keys (2)

For many-to-one or one-to-many mappings:

- e.g. between strong entity-sets *A* and B
- Primary key of entity-set on “many” side is primary key of relationship

Example: relationship *R* between *A* and *B*

- One-to-many mapping, with *B* on “many” side
- Schema contains *primary_key*(*A*) $\cup$ *primary_key*(*B*), plus any descriptive attributes on *R*
- *primary_key*(*B*) is primary key of *R*
  - Each $a \in A$ can map to many $b \in B$
  - Each value for *primary_key*(*B*) can appear only once in *R*

Relationship-Set Foreign Keys

Relationship-sets associate entities in entity-sets

- We need foreign-key constraints on relation schema for *R* !

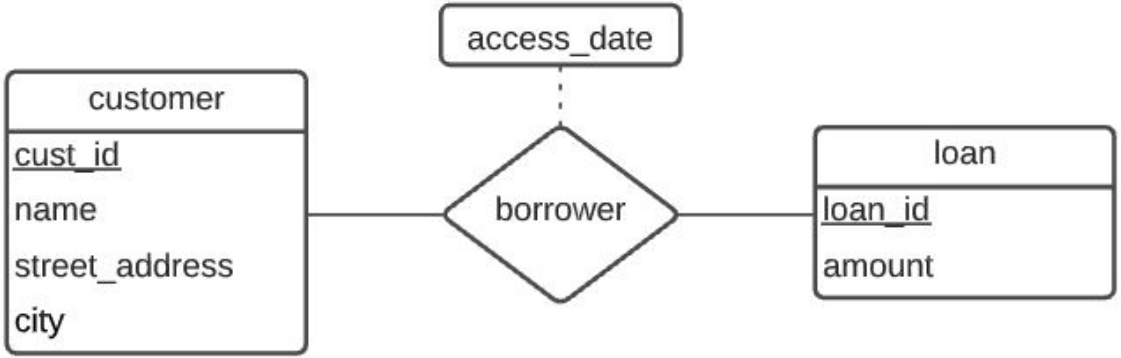
For each entity-set E_i participating in *R* :

- Relation schema for *R* has a foreign-key constraint on E_i relation, for *primary_key*(E_i) attributes

Relation schema notation doesn't provide mechanism for indicating foreign key constraints

- Don't forget about foreign keys and candidate keys!
    - Making notes on your relational model schema is a very good idea
- Can specify both foreign key constraints and candidate keys in the SQL DDL

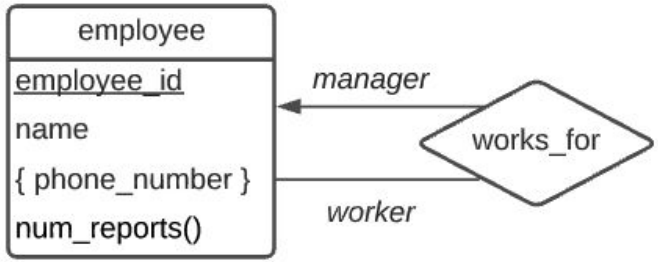
Relationship-Set Example



Relation schema for *borrower*:

- Primary key of *customer* is *cust_id*
- Primary key of *loan* is *loan_id*
- Descriptive attribute *access_date*
- *borrower* mapping cardinality is many-to-many
- Result: *borrower*(*cust_id*, *loan_id*, *access_date*)

Relationship-Set Example (2)



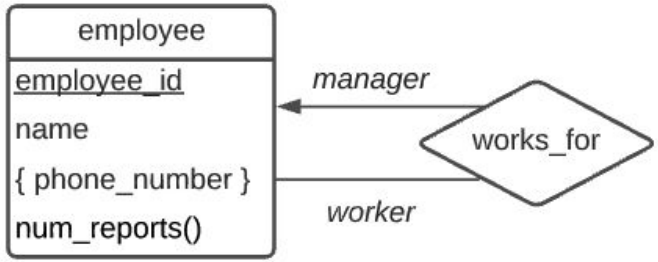
In cases like this, must use roles to distinguish between the entities involved in the relationship-set

- *employee* participates in *works_for* relationship-set twice
- Can’t create a schema (*employee_id*, *employee_id*) !

Change names of key-attributes to distinguish roles

- e.g. (*manager_employee_id*, *worker_employee_id*)
- e.g. (*manager_id*, *employee_id*)

Relationship-Set Example (2)



Relation schema for *employee* entity-set:

- (For now, ignore *phone_number* and *num_reports*…)

  employee(*employee_id*, *name*)

Relation schema for *works_for*:

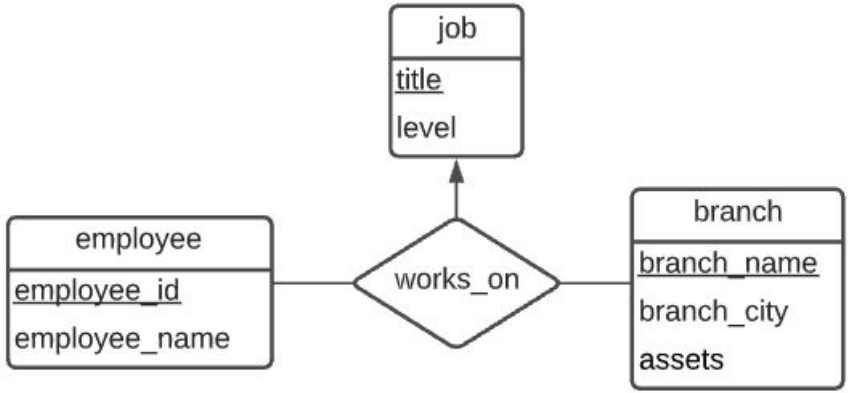
- One-to-many mapping from *manager* to *worker*
- “Many” side is used for primary key
- Result: *works_for*(*employee_id*, *manager_id*)

N-ary Relationship Primary Keys

For degree > 2 relationship-sets:

- If no arrows (“many-to-many” mapping), relationship-set primary key is union of all participating entity-sets’ primary keys
- If one arrow (“one-to-many” mapping), relationship-set primary key is union of primary keys of entity-sets without an arrow
- Don’t allow more than one arrow for relationship-sets with degree > 2

N-ary Relationship-Set Example



Entity-set schemas:

job(*title*, *level*)

employee(*employee_id*, *employee_name*)

branch(*branch_name*, *branch_city*, *assets*)

Relationship-set schema:

- Primary key includes entity-sets on non-arrow links

works_on(*employee_id*, *branch_name*, *title*)

Weak Entity-Sets

Recall that weak entity-sets depend on at least one strong entity-set

- The identifying entity-set, or owner entity-set
- Relationship between the two is called the identifying relationship

Weak entity-set *A* owned by strong entity-set *B*

- Attributes of *A* are $\{a_1, a_2, \ldots, a_m\}$
  - Some subset of these attributes comprises the discriminator of *A*
- *primary_key*(*B*) = $\{b_1, b_2, \ldots, b_n\}$
- Relation schema for *A*: $\{a_1,a_2,\ldots,a_m\} \cup \{b_1,b_2,\ldots,b_n\}$
- Primary key of *A* is *discriminator*(*A*) $\cup$ *primary_key*(*B*)
- *A* must also have a foreign key constraint on *primary_key*(*B*), to *B*
- In DDL, **`ON CASCADE DELETE`** is also useful

Identifying Relationship?

The identifying relationship is many-to-one, with no descriptive attributes

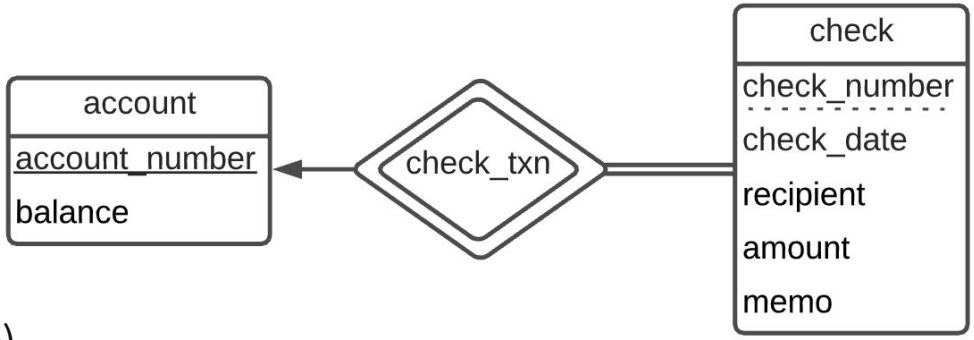
Relation schema for weak entity-set already includes primary key for strong entity-set

- Foreign key constraint is imposed, too

No need to create relational model schema for the identifying relationship

- Would be redundant to the weak entity-set’s relational model schema!

Weak Entity-Set Example



account schema:

account(*account_number*, *balance*)

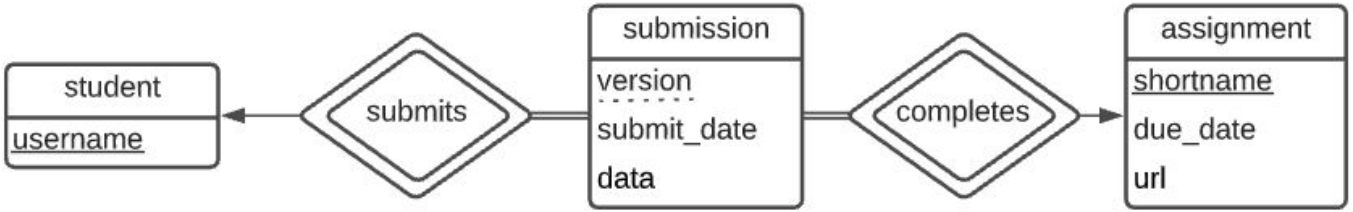
check schema:

- Discriminator is *check_number*
- Primary key for *check* is: (*account_number*, *check_number*)

check(*account_number*, *check_number*, *check_date*, *recipient*, *amount*, *memo*)

Review: What does the cardinality and participation tell us about the relationship?

Weak Entity-Set Example (2)



Schemas for strong entity-sets:

student(*username*)

assignment(*shortname*, *due_date*, *url*)

Schema for *submission* weak entity-set:

- Discriminator is *version*
- Both *student* and *assignment* are owners!

  submission(*username*, *shortname*, *version*, *submit_date*, *data*)
- Two foreign keys in this relation as well; again, no relation for *submits* or *completes*

Practice: Weak Entities in Assignment 7

Problem 1 (flights and airplanes) motivates a weak entity set.

First, let's identify the entity sets

1. Flights
2. Airplanes (aircraft types/models)
3. Seats on a specific airplane

What are the relationships?

- Relationships between flight and aircrafts
- Relationship between seats and aircrafts (weak)

(LucidChart demo)

Composite Attributes

Relational model simply doesn’t handle composite attributes

- All attribute domains are *atomic* in the relational model

When mapping E-R composite attributes to relation schema: simply flatten the composite

- Each component attribute maps to a separate attribute in relation schema
- In relation schema, simply can’t refer to the composite as a whole
- (Can adjust this mapping for databases that support composite types)

Composite Attribute Example

Customers with addresses:

Each component of *address* becomes a separate attribute

customer(*cust_id*, *name*, *street*, *city*, *state*, *zip_code*)

Multivalued Attributes

Multivalued attributes **require** a separate relation

- Again, no such thing as a multivalued attribute in the relational model
- E-R constraint on multivalued attributes: in a specific entity's multivalued attribute, each value may only appear once

For a multivalued attribute *M* in entity-set *E*

- Create a relation schema *R* to store *M*, with attribute(s) *A* corresponding to the single-valued version of *M*
- Attributes of *R* are: *primary_key*(*E*) ∪ *A*
- Primary key of *R* includes all attributes of *R*
  - Each value in *M* for an entity *e* must be unique
- Foreign key from *R* to *E*, on *primary_key*(*E*) attributes

Multivalued Attribute Example

Change our E-R diagram to allow customers to have multiple addresses:

Now, must create a separate relation to store the addresses

customer(*cust_id*, *name*)

cust_addrs(*cust_id*, *street*, *city*, *state*, *zipcode*)

- Large primary keys aren't ideal – tend to be costly

Extensions to the E-R Model

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 is included with Assignment 6 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)

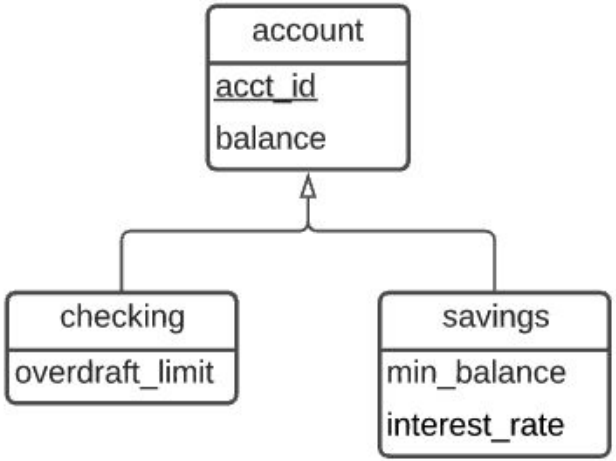
Create entity-set to represent common attributes

- 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)



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

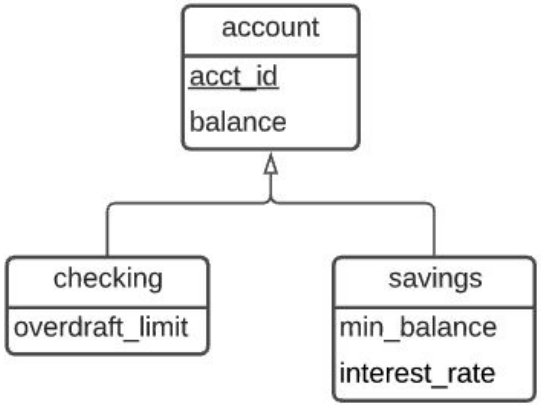
Usually, entity-sets inherit from one superclass

- 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

Practice: Specialization in Assignment 6

Problem 2 (airline customers) motivates a weak entity set.

First, let's identify the entity sets (yours may slightly differ)

1. Purchase info
2. People
3. ???

What are the relationships?

Where does specialization come in hand?

(LucidChart demo)

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:

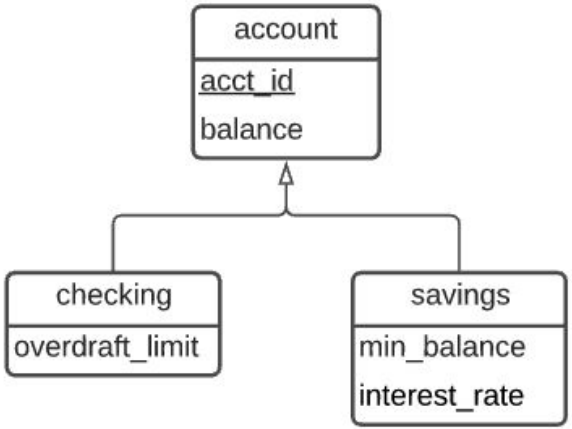
- Called an **overlapping specialization**

Disjointness Constraints (2)

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



Disjointness Constraints (3)

Another example:

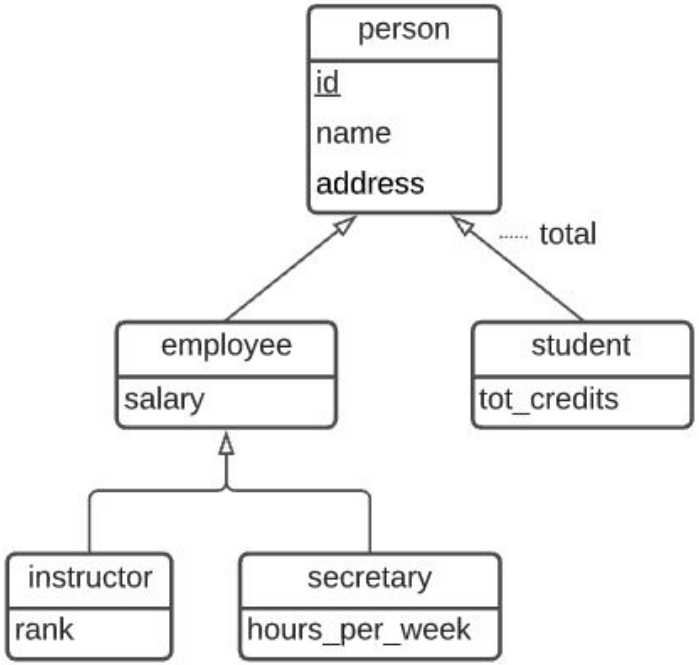
- Specialization hierarchy for people at a university

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

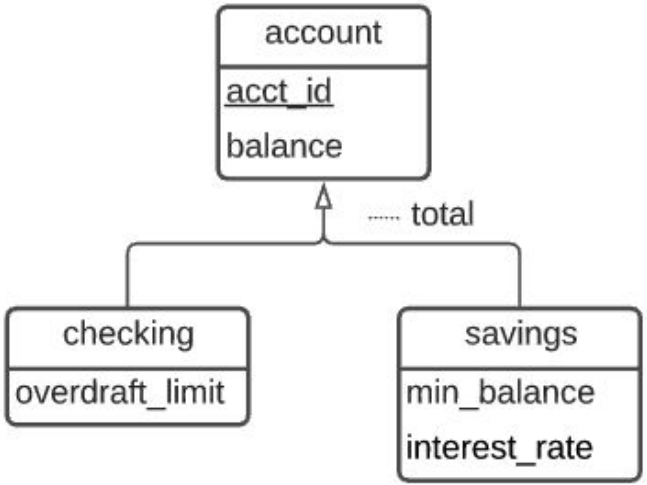
account specialization is a total specialization

Completeness Constraints (2)

Default constraint is partial specialization

Specify total specialization constraint by annotating the specialization arrow(s)

Updated bank account diagram:



Completeness Constraints (3)

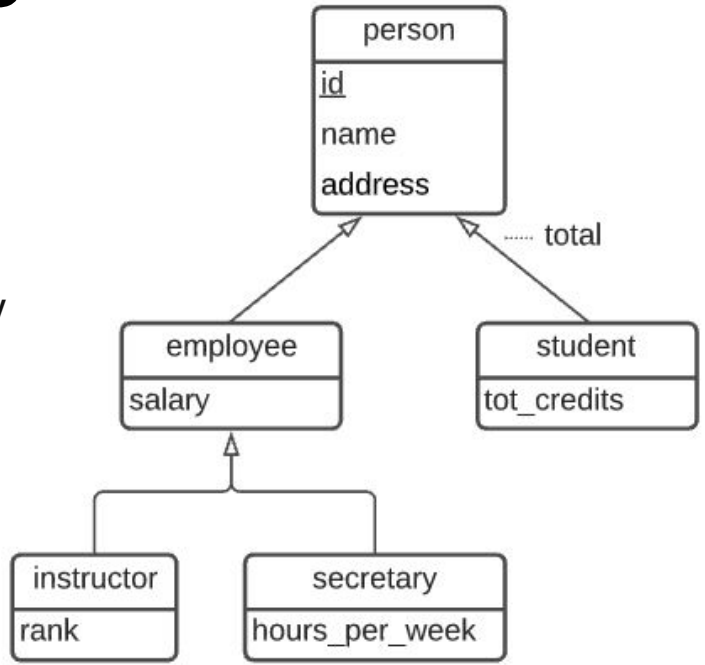
Same approach with overlapping specialization

Example: people at a university

- Every person is an employee or a student
- Not every employee is an instructor or a secretary

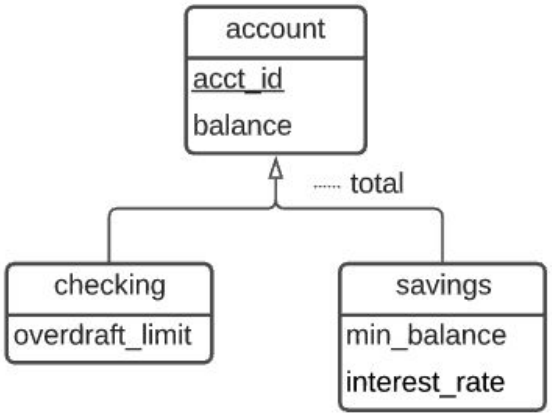
Annotate arrows pointing to person with “total” to indicate total specialization

- Every person must be an employee, a student, or both



Account Types?

Our bank schema so far:



How to tell whether an account is a checking account or a savings account?

- No attribute indicates type of account

Practice: Specialization in Assignment 6

Problem 2 (airline customers) motivates a weak entity set.

First, let's identify the entity sets (yours may slightly differ)

1. Purchase info
2. **Customers**
    - **Purchasers**
    - **Travelers**

Where does specialization come in hand? **Do we disjointness or completeness constraints?**

(LucidChart demo)

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

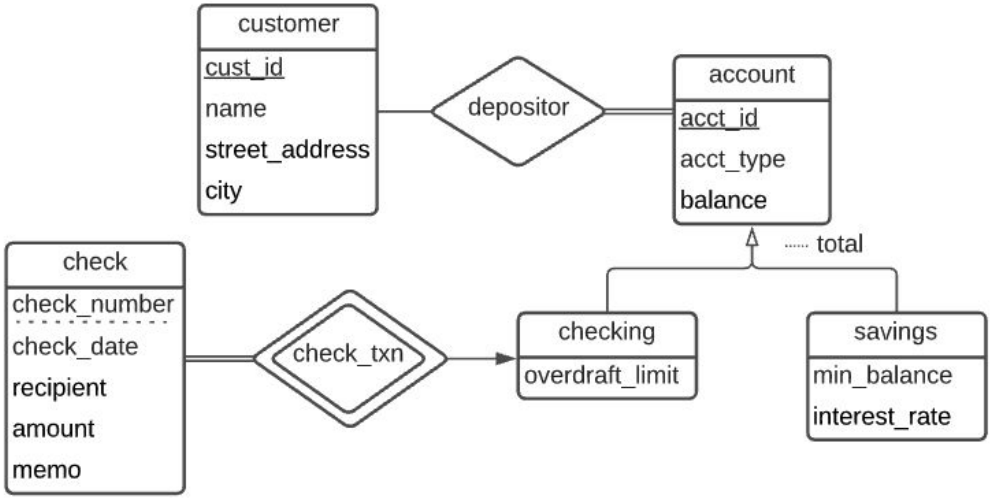
Membership Constraints (2)

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

Final Bank Account Diagram



Would also create relationship-sets against various entity-sets in hierarchy

- associate *customer* with *account*
- associate *check* weak entity-set with *checking*

Translating ER to DDL

Mapping to Relational Model

Mapping generalization/specialization to relational model is straightforward

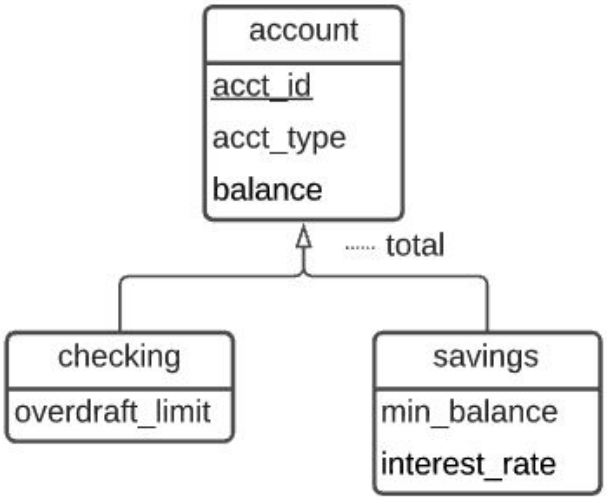
Create relation schema for higher-level entity-set

- 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

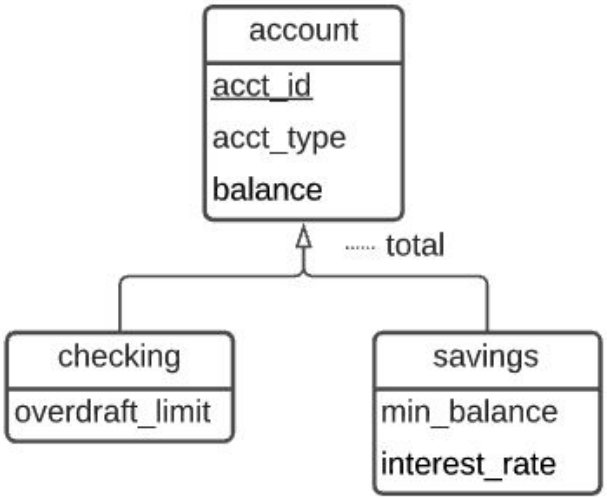
If specialization is disjoint and complete, could convert only lower-level entity-sets to relational schemas

- 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

Alternative Account Schema



Schemas, take 2:

checking(<u>*acct_id*</u>, *acct_type*, *balance*, *overdraft_limit*)

savings(<u>*acct_id*</u>, *acct_type*, *balance*, *min_balance*, *interest_rate*)

Alternative Account Schema (2)

Alternative schemas:

checking(<u>*acct_id*</u>, *acct_type*, *balance*, *overdraft_limit*)

savings(<u>*acct_id*</u>, *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(<u>*acct_id*</u>)

- Contains *all* valid account IDs
- Relationships involving accounts can use *account*
- Need foreign key constraints again…

Generating Primary Keys

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 from second example has serious drawbacks

- Doesn’t actually give many benefits in general case

Fewer drawbacks if:

- Total, disjoint specialization
- No relationships against superclass entity-set

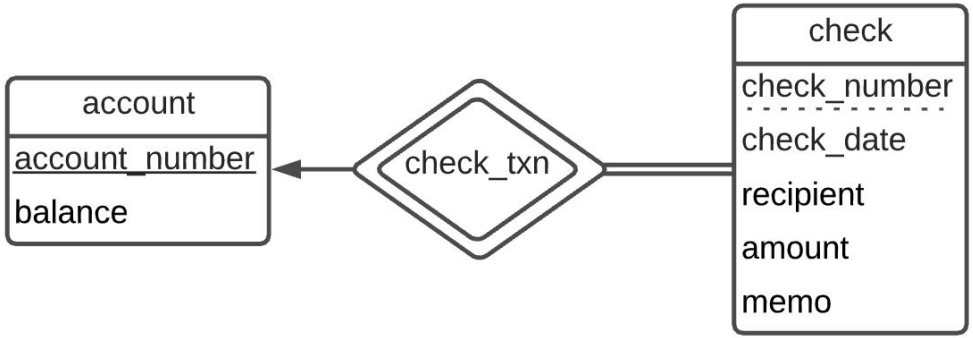
If specialization is overlapping, some details will be stored multiple times

- 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:

account(*account_number*, *balance*)

check schema:

- Discriminator is *check_number*
- Primary key for *check* is: (*account_number*, *check_number*)

check(*account_number*, *check_number*, *check_date*, *recipient*, *amount*, *memo*)

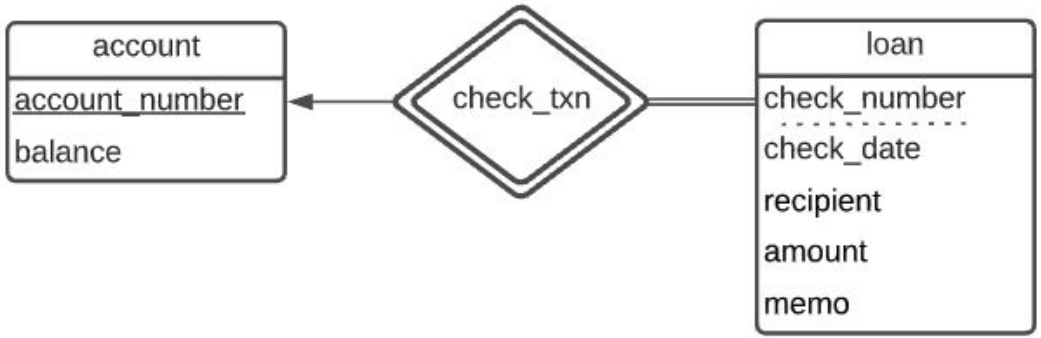
Schema Combination

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:

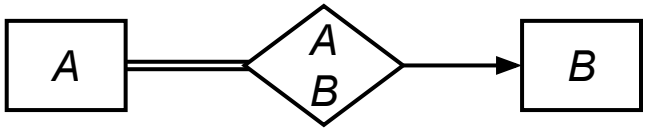
- One-to-many mapping, with total participation on the "many" side



Schema Combination (2)

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 *primary_key*(*A*)
- Primary key of *AB* is also *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*

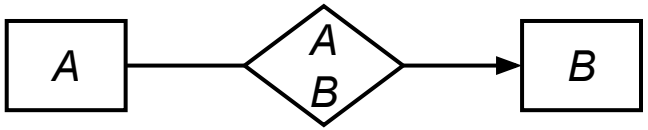
Can combine *A* and *AB* relation schemas

- Primary key of combined schema still *primary_key*(*A*)
- Only requires one foreign-key constraint, to *B*

Schema Combination (3)

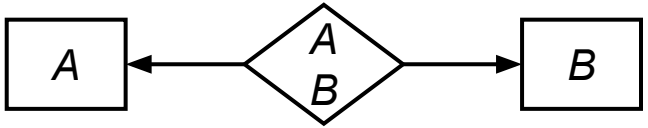
In this case, when relationship-set is combined into the entity-set, the entity-set's primary key *doesn't change!*

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 <u>two</u> candidate keys…
  - The combined schema must still enforce both candidate keys

Schema Combination Example

Manager to worker mapping is one-to-many

Relation schemas were:

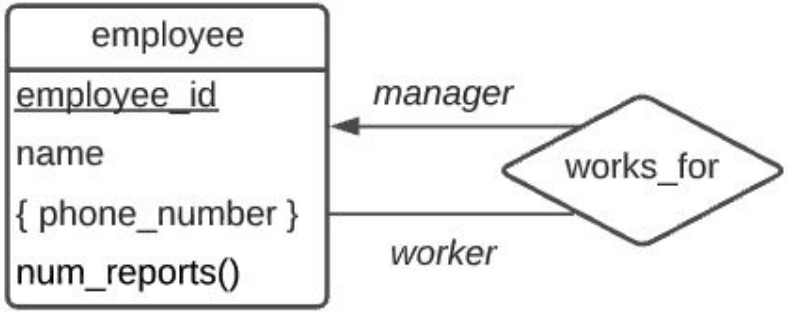
employee(*employee_id*, *name*)

works_for(*employee_id*, *manager_id*)

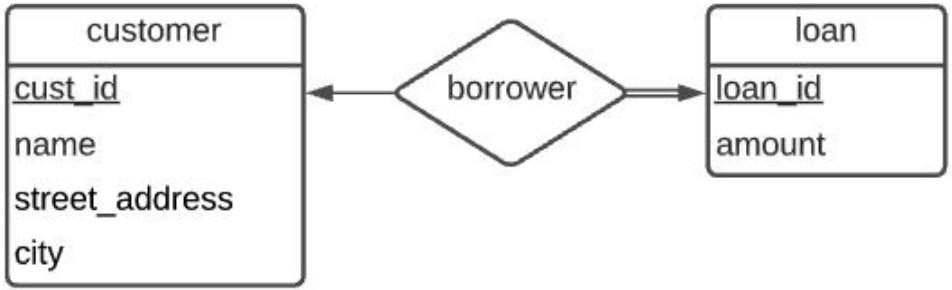
Could combine into:

employee(*employee_id*, *name*, *manager_id*)

- (A very common schema combination)
- Need to store *null* for employees with no manager



Schema Combination Example (2)



One-to-one mapping between customers and loans

customer(*cust_id*, *name*, *street_address*, *city*)

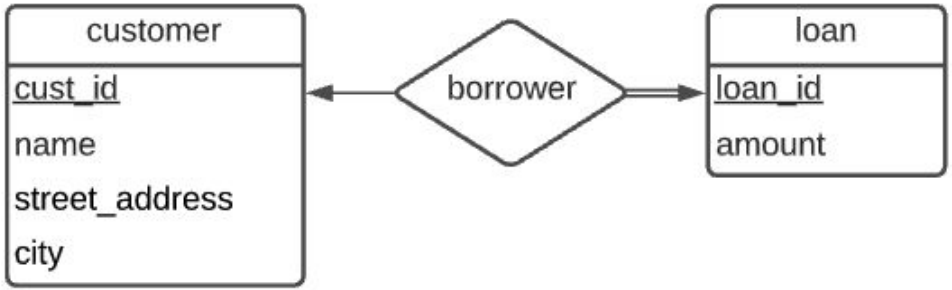
loan(*loan_id*, *amount*)

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?

Schema Combination Example (3)



Participation of *loan* in *borrower* will be total

- Combining *borrower* into *customer* would require *null* values for customers without loans

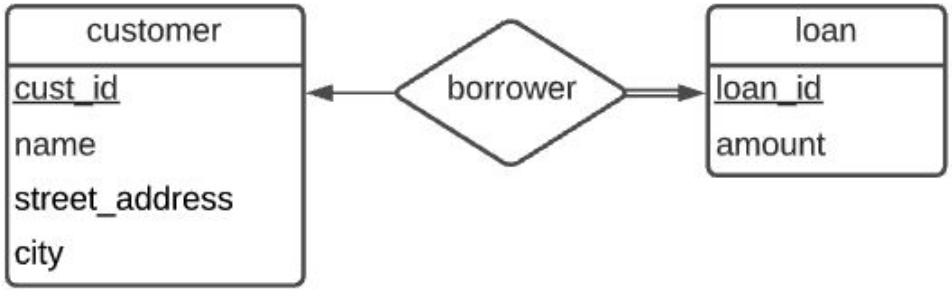
Better to combine *borrower* into *loan* schema

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

Schema Combination Notes

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