DATABASE SCHEMA DESIGN
ENTITY-RELATIONSHIP MODEL

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 14

Designing Database Applications

- Database applications are large and complex
- A few of the many design areas:
  - Database schema (physical/logical/view)
  - Programs that access and update data
  - Security constraints for data access

- Also requires familiarity with the problem domain
  - Domain experts *must* help drive requirements

General Approach

- Collect user requirements
  - Information that needs to be represented
  - Operations to perform on that information
  - Several techniques for representing this info, e.g. UML
- Develop a <u>conceptual schema</u> of the database
  - A high-level representation of the database's structure and constraints
    - Physical *and* logical design issues are ignored at this stage
  - Follows a <u>specific</u> data model
  - Often represented graphically

Conceptual Schema

- Also need to create a <u>specification of functional requirements</u>
  - "What operations will be performed against the data?"
  - Updating data, adding data, deleting data, …
- Designer can use functional requirements to verify the conceptual schema
  - Is each operation possible?
  - How complicated or involved is it?
  - Performance or scalability concerns?

Implementation Phases

- Once conceptual schema and functional requirements are verified:
  - Convert conceptual schema into an <u>implementation data model</u>
  - Want to have a simple mapping from conceptual model to implementation model
- Finally: any necessary physical design
  - Not always present!
  - Smaller applications have few physical design concerns
  - Larger systems usually need additional design and tuning (e.g. indexes, disk-level partitioning of data)

Importance of Design Phase

- Not all changes have the same impact!
- Physical-level changes have the least impact
  - (Thanks, relational model!)
  - Typically affect performance, scalability, reliability
  - Little to no change in functionality
- Logical-level changes are typically *much* bigger
  - Affects how to interact with the data…
  - Also affects what is even *possible* to do with the data
- <u>Very important</u> to spend time up front designing the database schema

Design Decisions

- Many different ways to represent data
- Must avoid two major problems:
  - Unnecessary redundancy
    - Redundant information wastes space
    - Greater potential for inconsistency!
    - Ideally: each fact appears in exactly one place
  - Incomplete representation
    - Schema must be able to fully represent all details and relationships required by the application

More Design Decisions

- Even with correct design, usually many other concerns
  - How easy/hard is it to access useful information? (e.g. reporting or summary info)
  - How hard is it to update the system?
  - Performance considerations?
  - Scalability considerations?
- Schema design requires a good balance between aesthetic and practical concerns
  - Frequently need to make compromises between conflicting design principles

Simple Design Example

- Purchase tracking database
  - Store details about product purchases by customers
  - Actual purchases tracked in database
- Can represent sales as relationships between customers and products
  - What if product price changes? Where to store product sale price? Will this affect other recent purchases?
  - What about giving discounts to some customers? May want to give different prices to different customers.
- Can also represent sales as separate entities
  - Gives much more flexibility for special pricing, etc.

The Entity-Relationship Model

- A *very* common model for schema design
  - Also written as “E-R model”
- Allows for specification of complex schemas in graphical form
- Basic concepts are simple, but can also represent very sophisticated abstractions
  - e.g. type hierarchies
- Can be mapped very easily to the relational model!
  - Simplifies implementation phase
  - Mapping process can be automated by design tools

Entities and Entity-Sets

- An entity is any "thing" that can be uniquely represented
    - e.g. a product, an employee, a software defect
  - Each entity has a set of attributes
  - Entities are uniquely identified by some set of attributes
- An entity-set is a named collection of entities of the same type, with the same attributes
  - Can have multiple entity-sets with same entity type, representing different (possibly overlapping) sets

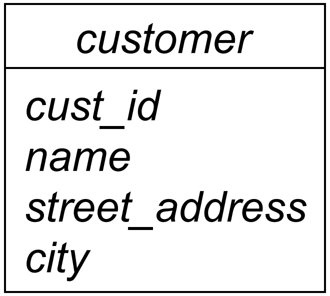
Entities and Entity-Sets (2)

- An entity has a set of <u>attributes</u>
  - Each attribute has a name and domain
  - Each attribute also has a corresponding value
- Entity-sets also specify a set of attributes
  - Every entity in the entity-set has the same set of attributes
  - Every entity in the entity-set has its own value for each attribute

Diagramming an Entity-Set

Example: a *customer* entity-set

- Attributes:
  - *cust_id*
  - *name*
  - *street_address*
  - *city*



- Entity-set is denoted by a box
- Name of entity-set is given in the top part of box
- Attributes are listed in the lower part of the box

Relationships

- A <u>relationship</u> is an association between two or more entities
  - e.g. a bank loan, and the customer who owns it
- A <u>relationship-set</u> is a named collection of relationships of the same type
  - i.e. involving the same entities
- Formally, a relationship-set is a mathematical relation involving n entity-sets, $n \geq 2$
  - $E_1, E_2, \ldots, E_n$ are entity sets; $e_1, e_2, \ldots$ are entities in $E_1, E_2, \ldots$
  - A relationship set R is a subset of:
    $\{ (e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n \}$
  - $(e_1, e_2, \ldots, e_n)$ is a specific relationship in R

Relationships (2)

- Entity-sets <u>participate</u> in relationship-sets
  - Specific entities participate in a <u>relationship instance</u>
- Example: bank loans
  - *customer* and *loan* are entity-sets

    (*555-55-5555*, Jackson, Woodside) is a *customer* entity

    (L-14, 1500) is a *loan* entity
  - *borrower* is a relationship-set
    - *customer* and *loan* participate in *borrower*
    - *borrower* contains a relationship instance that associates customer "Jackson" and loan "L-14"

Relationships and Roles

- An entity's <u>role</u> in a relationship is the function that the entity fills

  Example: a *purchase* relationship between a *product* and a *customer*
  - the product's role is that it was purchased
  - the customer did the purchasing
- Roles are usually obvious, and therefore unspecified
  - Entities participating in relationships are distinct…
  - Names clearly indicate the roles of various entities…
  - In these cases, roles are left unstated.

Relationships and Roles (2)

- Sometimes the roles of entities are *not* obvious
  - Situations where entity-sets in a relationship-set are *not* distinct

Example: a relationship-set named *works_for*, specifying employee/manager assignments
  - Relationship involves two entities, and both are *employee* entities

- Roles are given names to distinguish entities
  - The relationship is a set of entities ordered by role:
    (*manager*, *worker*)
  - First entity's role is named *manager*
  - Second entity's role is named *worker*

Relationships and Attributes

- Relationships can also have attributes!
  - Called descriptive attributes
  - They describe a particular relationship
  - They *do not* identify the relationship!
- Example:
  - The relationship between a software defect and an employee can have a *date_assigned* attribute
- Note: this distinction between entity attributes and relationship attributes is not made by relational model
  - Entity-relationship model is a higher level of abstraction than the relational model

Relationships and Attributes (2)

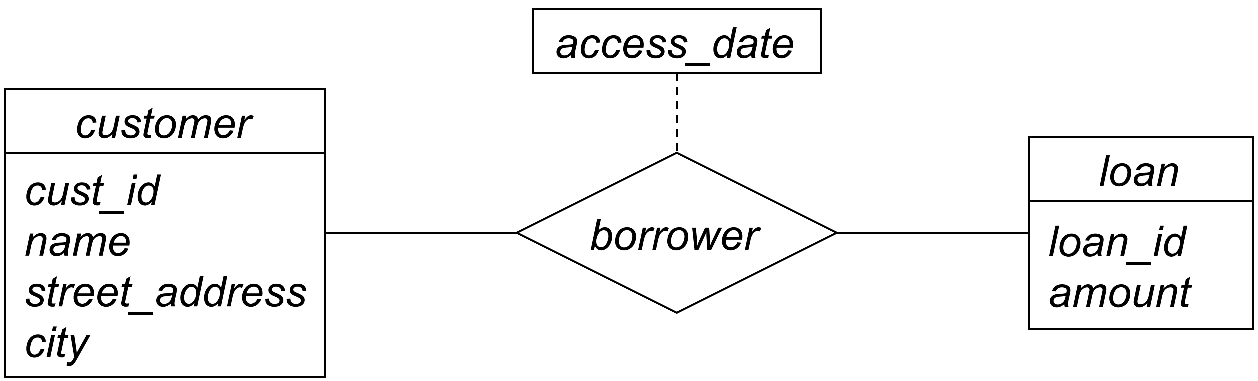
- Specific relationships are identified *only* by the participating entities
  - …not by any relationship attributes!
  - Different relationships are allowed to have the same value for a descriptive attribute
  - (This is why entities in an entity-set must be uniquely identifiable.)
- Given:
  - Entity-sets A and B, both participating in a relationship-set R
- Specific entities $a \in A$ and $b \in B$ can only have <u>one</u> relationship instance in R
  - Otherwise, we would require more than just the participating entities to uniquely identify relationships

Degree of Relationship Set

- Most relationships in a schema are <u>binary</u>
  - Two entities are involved in the relationship
- Sometimes there are ternary relationships
  - Three entities are involved
  - Far less common, but still useful at times
- The number of entity-sets that participate in a relationship-set is called its <u>degree</u>
  - Binary relationship: degree = 2
  - Ternary relationship: degree = 3

Diagramming a Relationship-Set

Example: the *borrower* relationship-set between the *customer* and *loan* entity-sets



- Relationship-set is a diamond
  - Connected to participating entity-sets by solid lines
- Relationship-set can have descriptive attributes
  - Listed in another box, connected with a dotted-line

Attribute Structure

- Each attribute has a domain or value set
  - Values come from that domain or value set
- Simple attributes are atomic – they have no subparts
  - e.g. *amount* attribute is a single numeric value
- Composite attributes have subparts
  - Can refer to composite attribute as a whole
  - Can also refer to subparts individually
  - e.g. *address* attribute, composed of *street*, *city*, *state*, *postal_code* attributes

Attribute Cardinality

- Single-valued attributes only store one value
  - e.g. a *customer*'s *cust_id* only has one value
- Multi-valued attributes store zero or more values
  - e.g. a *customer* can have multiple *phone_number* values
  - A multi-valued attribute stores a set of values, not a multiset
  - Different *customer* entities can have different sets of phone numbers
  - Lower and upper bounds can be specified too
    - Can set upper bound on *phone_number* to 2

Attribute Source

- Base attributes (aka source attributes) are stored in the database
  - e.g. the *birth_date* of a *customer* entity
- Derived attributes are computed from other attributes
  - e.g. the *age* of a *customer* entity would be computed from their *birth_date*

Diagramming Attributes

- Example: Extend customers with more detailed info

- Composite attributes are shown as a hierarchy of values
  - Indented values are components of the higher-level value
  - e.g. *name* is comprised of *first_name*, *middle_initial*, and *last_name*

customer

cust_id
name
    first_name
    middle_initial
    last_name
address
    street
    city
    state
    zip_code

Diagramming Attributes (2)

- Example: Extend customers with more detailed info

- Multivalued attributes are enclosed with curly-braces
  - e.g. each customer can have zero or more phone numbers

customer

cust_id
name
    first_name
    middle_initial
    last_name
address
    street
    city
    state
    zip_code
{ *phone_number* }

Diagramming Attributes (3)

- Example: Extend customers with more detailed info
- Derived attributes are indicated by a trailing set of parentheses ()
  - e.g. each customer has a base attribute recording their date of birth
  - Also a derived attribute that reports the customer's current age

customer

cust_id
name
    first_name
    middle_initial
    last_name
address
    street
    city
    state
    zip_code
{ *phone_number* }
birth_date
age ()

Representing Constraints

- E-R model can represent different kinds of constraints
  - Mapping cardinalities
  - Key constraints in entity-sets
  - Participation constraints
- Allows more accurate modeling of application's data requirements
  - Constrain design so that schema can only represent valid information
- Enforcing constraints can impact performance…
  - Still ought to specify them in the design!
  - Can always leave out constraints at implementation time

Mapping Cardinalities

- Mapping cardinality represents:

  "How many other entities can be associated with an entity, via a particular relationship set?"
- Example:
  - How many *customer* entities can the *borrower* relationship associate with a single *loan* entity?
  - How many *loans* can *borrower* relationship associate with a single *customer* entity?
  - Specific answer depends on what is being modeled
- Also known as the cardinality ratio
- Easiest to reason about with binary relationships

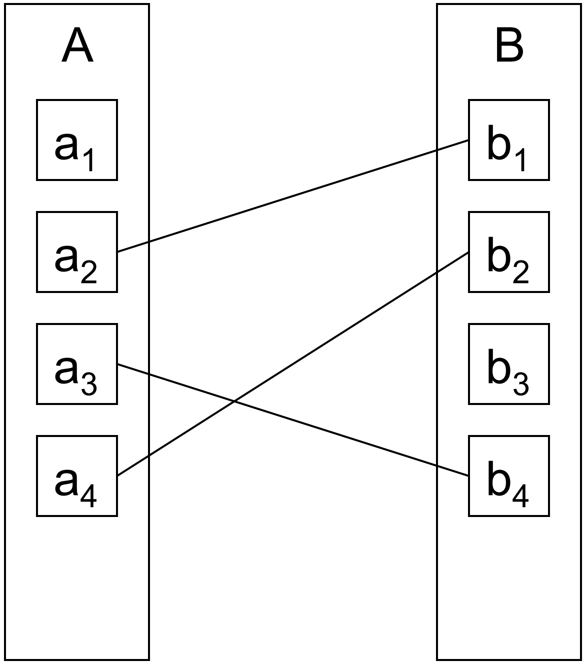
Mapping Cardinalities (2)

Given:

- Entity-sets *A* and *B*
- Binary relationship-set *R* associating *A* and *B*

One-to-one mapping (1:1)

- An entity in *A* is associated with *at most* one entity in *B*
- An entity in *B* is associated with *at most* one entity in *A*



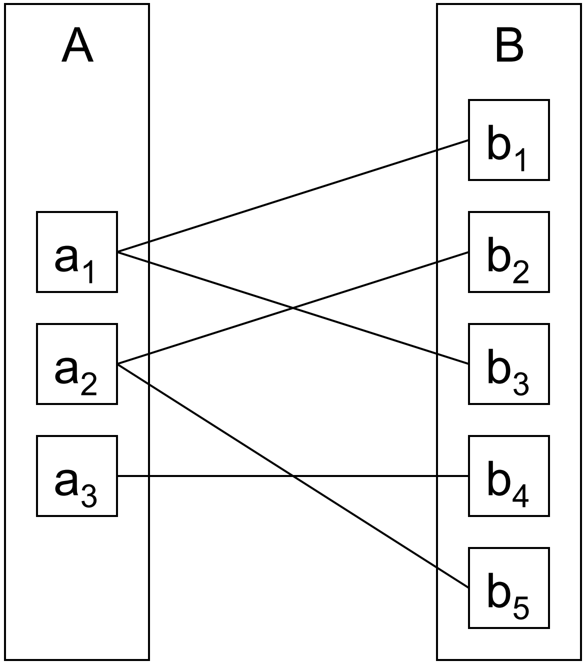
Mapping Cardinalities (3)

One-to-many mapping (1:*M*)

- An entity in *A* is associated with *zero or more* entities in *B*
- An entity in *B* is associated with *at most* one entity in *A*

Many-to-one mapping (*M*:1)

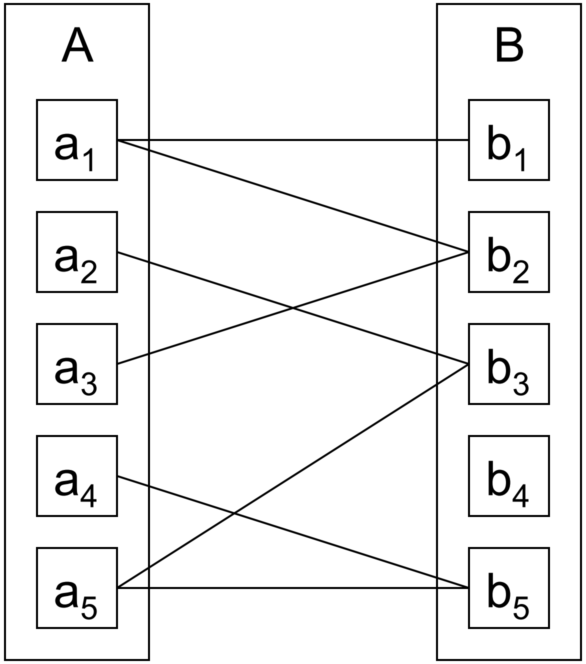
- Opposite of one-to-many
- An entity in *A* is associated with *at most* one entity in *B*
- An entity in *B* is associated with *zero or more* entities in *A*



Mapping Cardinalities (4)

Many-to-many mapping

- An entity in A is associated with *zero or more* entities in B
- An entity in B is associated with *zero or more* entities in A



Mapping Cardinalities (5)

- Which mapping cardinality is most appropriate for a given relationship?
  - Answer depends on what you are trying to model!
  - Could just use many-to-many relationships everywhere, but that would be dumb.
- Goal:
  - Constrain the mapping cardinality to most accurately reflect what should be allowed
  - Database can enforce these constraints automatically
  - Good schema design reduces or eliminates the *possibility* of storing bad data

Example: *borrower* relationship between *customer* and *loan*

One-to-one mapping:

- Each customer can have only one loan
- Customers can't share loans

  (e.g. with spouse or business partner)

One-to-many mapping:

- A customer can have multiple loans
- Customers still can't share loans

Many-to-one mapping:

- Each customer can have only one loan
- Customers can share loans

Many-to-many mapping:

- A customer can have multiple loans
- Customers can share loans too

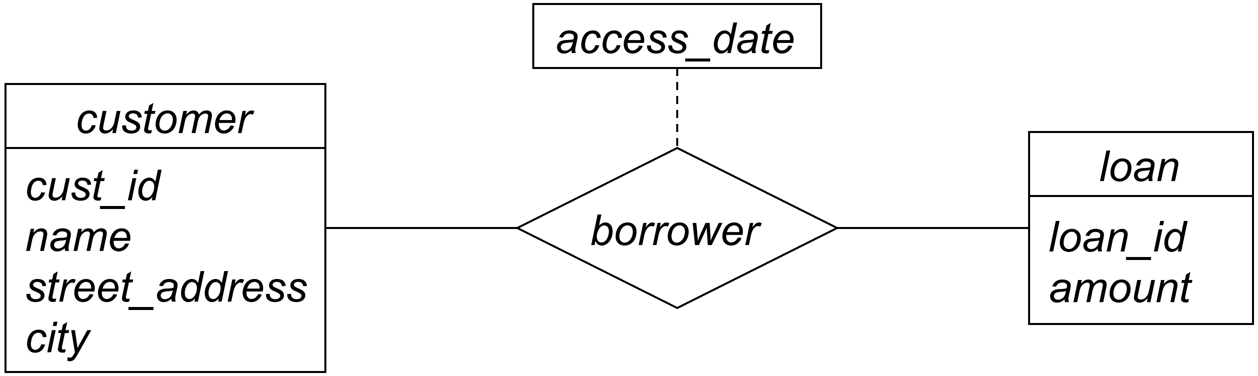
Best choice for *borrower* :

many-to-many mapping

Handles real-world needs!

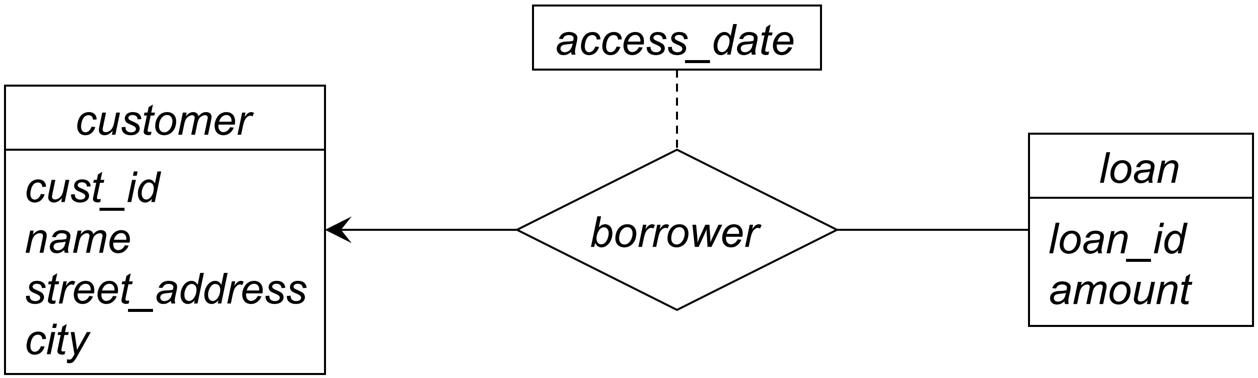
Diagramming Cardinalities

- In relationship-set diagrams:
  - an arrow towards an entity represents "one"
  - a simple line represents "many"
  - arrow is *always* towards the entity
- Many-to-many mapping between *customer* and *loan*:



Diagramming Cardinalities (2)

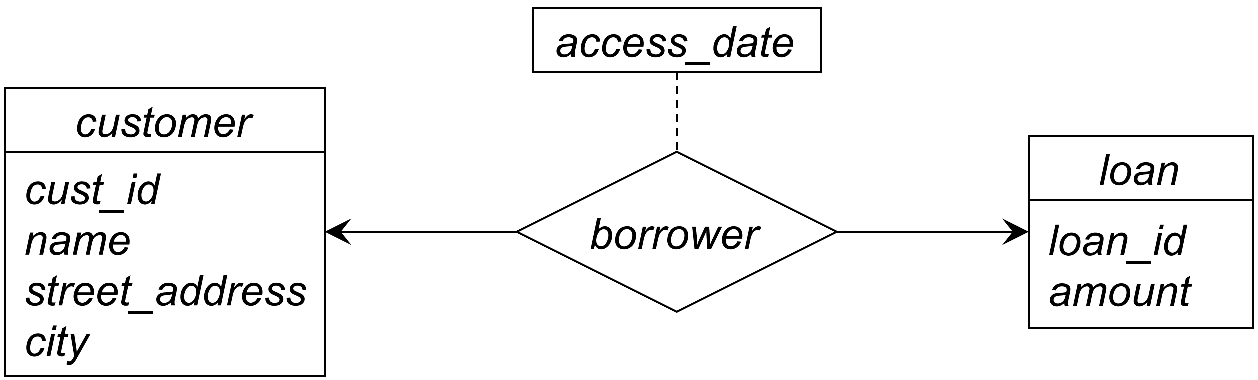
- One-to-many mapping between *customer* and *loan*:



  - Each customer can have multiple loans
  - A loan is owned by <u>exactly</u> one customer
    - (Actually, this is technically "<u>at most</u> one". Participation constraints will allow us to say "exactly one.")

Diagramming Cardinalities (3)

- One-to-one mapping between *customer* and *loan*:



  - Each customer can have only one loan
  - A loan is owned by exactly one customer

ENTITY-RELATIONSHIP MODEL II

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 15

Last Lecture

- Began to explore the Entity-Relationship Model
  - A visual representation of database schemas
  - Can represent entities and relationships
  - Can represent constraints in the schema
- Last time, left off with mapping cardinalities

Entity-Set Keys

- Entities in an entity-set must be uniquely distinguishable using their values
  - Entity-<u>set</u>: each entity is unique
- E-R model also includes the notion of keys:
  - Superkey: a set of one or more attributes that can uniquely identify an entity
  - Candidate key: a *minimal* superkey
  - Primary key: a candidate key chosen by DB designer as the primary means of accessing entities
- Keys are a property of the entity-set
  - They apply to *all* entities in the entity-set

Choosing Candidate Keys

- Candidate keys constrain the values of the key attributes
  - No two entities can have the same values for those attributes
  - Need to ensure that database can actually represent all expected circumstances
- Simple example: *customer* entity-set
  - Using customer name as a candidate key is bad design: different customers can have the same name

Choosing Primary Keys

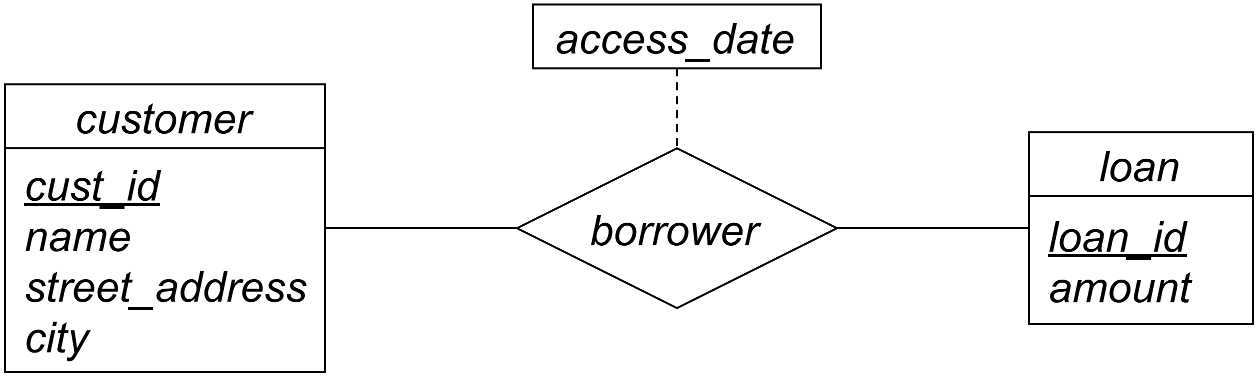
- An entity-set may have multiple candidate keys
- The primary key is the candidate key most often used to reference entities in the set
  - In logical/physical design, primary key values will be used to represent relationships
  - External systems may also use primary key values to reference entities in the database
- The primary key attributes should <u>*never*</u> change!
  - If ever, it should be *extremely* rare.

Choosing Keys: Performance

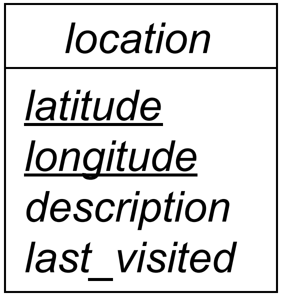
- Large, complicated, or multiple-attribute keys are generally slower
  - Use smaller, single-attribute keys
    - (You can always generate them…)
  - Use faster, fixed-size types
    - e.g. `INT` or `BIGINT`
- Especially true for primary keys!
  - Values used in both database and in access code
  - Use something small and simple, if possible

Diagramming Primary Keys

- In an entity-set diagram, all attributes in the primary key have an underlined name



- Another example: a geocache *location* entity-set



Keys and Relationship-Sets

- Need to be able to distinguish between individual relationships in a relationship-set as well
  - Relationships aren't distinguished by their descriptive attributes
  - (They might not even have descriptive attributes)
- Relationships are identified by the *entities* participating in the relationship
  - Specific relationship instances are uniquely identified by the primary keys of the participating entities

Keys and Relationship-Sets (2)

- Given:
  - R is a relationship-set with no descriptive attributes
  - Entity-sets E_1, E_2, …, E_n participate in R
  - $primary\_key(E_i)$ denotes set of attributes in E_i that represent the primary key of E_i
- A relationship instance in R is identified by $primary\_key(E_1) \cup primary\_key(E_2) \cup \ldots \cup primary\_key(E_n)$
  - This is a superkey
  - Is it a candidate key?
    - Depends on the *mapping cardinality* of the relationship set!

Keys and Relationship-Sets (3)

- If R also has descriptive attributes $\{a_1, a_2, \ldots\}$, a relationship instance is described by:

  $primary\_key(E_1) \cup primary\_key(E_2) \cup \ldots \cup primary\_key(E_n) \cup \{a_1, a_2, \ldots\}$

  - Not a minimal superkey!
  - By definition, there can only be one relationship between $\{E_1, E_2, \ldots, E_n\}$ in the relationship-set
    - i.e. the descriptive attributes do not identify specific relationships!
- Thus, just as before, this is also a superkey:

  $primary\_key(E_1) \cup primary\_key(E_2) \cup \ldots \cup primary\_key(E_n)$

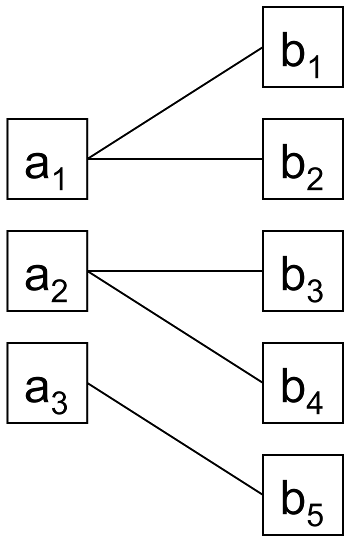
Relationship-Set Primary Keys

- What is the primary key for a binary relationship-set?
  - Must also be a candidate key
  - Depends on the mapping cardinalities
- Relationship-set *R*, involving entity-sets *A* and *B*
  - If mapping is many-to-many, primary key is:

    $primary\_key(A) \cup primary\_key(B)$
  - Any given entity's primary-key values can appear multiple times in *R*
  - We need both entity-sets' primary key attributes to uniquely identify relationship instances

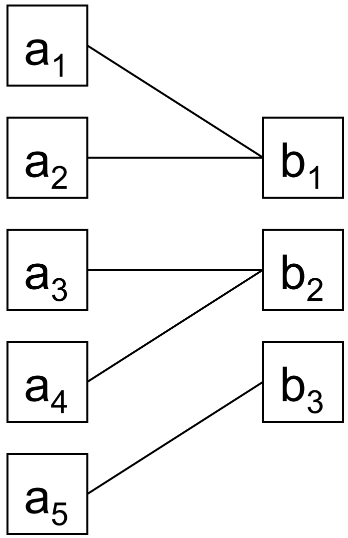
Relationship-Set Primary Keys (2)

- Relationship-set *R*, involving entity-sets *A* and *B*
  - Individual relationships are described by *primary_key*(*A*) ∪ *primary_key*(*B*)
- If mapping is one-to-many:
  - Entities in *B* associated with *at most* one entity in *A*
  - A given *primary_key*(*A*) value can appear in multiple relationships
  - Each value of *primary_key*(*B*) can appear <u>only once</u>
  - Relationships in *R* are uniquely identified by *primary_key*(*B*)
  - *primary_key*(*B*) is primary key of relationship-set



Relationship-Set Primary Keys (3)

- Relationship-set *R*, involving entity-sets *A* and *B*
- Many-to-one is exactly the opposite of one-to-many
  - *primary_key*(*A*) uniquely identifies relationships in *R*

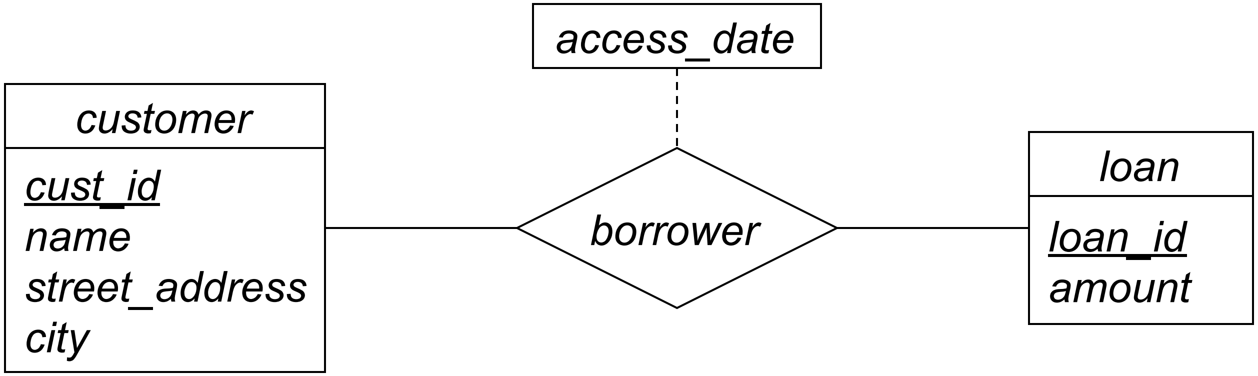


Relationship-Set Primary Keys (4)

- Relationship-set *R*, involving entity-sets *A* and *B*
- If mapping is one-to-one:
  - Entities in *A* associated with *at most* one entity in *B*
  - Entities in *B* associated with *at most* one entity in *A*
  - Each entity's key-value can appear only once in *R*
  - Either entity-set's primary key can be primary key of *R*
- For one-to-one mapping, *primary_key*(*A*) and *primary_key*(*B*) are both candidate keys
  - Make sure to enforce both candidate keys in the implementation schema!

Example

- What is the primary key for *borrower* ?



- *borrower* is a many-to-many mapping
  - Relationship instances are described by (*cust_id*, *loan_id*, *access_date*)
  - Primary key for relationship-set is (*cust_id*, *loan_id*)

Participation Constraints

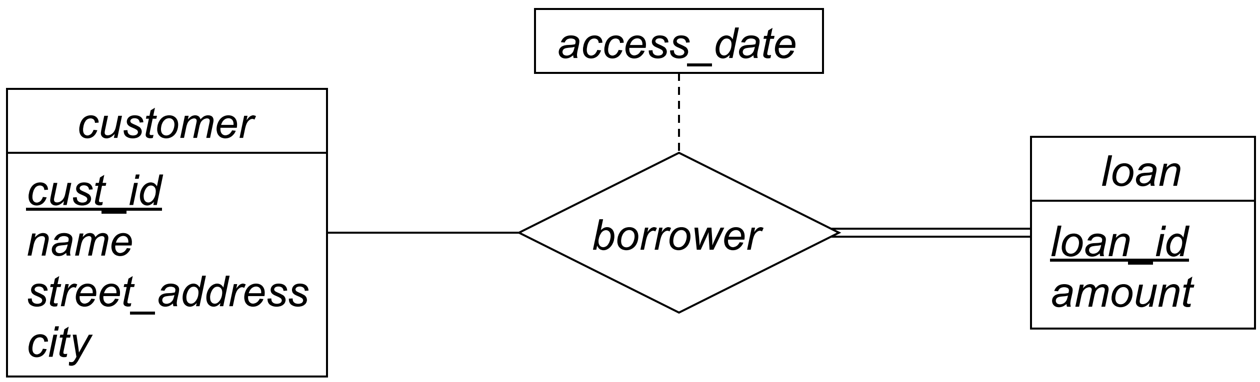
- Given entity-set *E*, relationship-set *R*
  - How many entities in *E* participate in *R* ?
  - In other words, what is minimum number of relationships that each entity in *E* *must* participate in?
- If every entity in *E* participates in at least one relationship in *R*, then:
  - *E*'s participation in *R* is total
- If only some entities in *E* participate in relationships in *R*, then:
  - *E*'s participation in *R* is partial

Participation Constraints (2)

- Example: *borrower* relationship between *customer* and *loan*
- A customer might not have a bank loan
  - Could have a bank account instead
  - Could be a new customer
  - Participation of *customer* in *borrower* is <u>partial</u>
- Every loan definitely has at least one customer
  - Doesn't make any sense not to!
  - Participation of *loan* in *borrower* is <u>total</u>

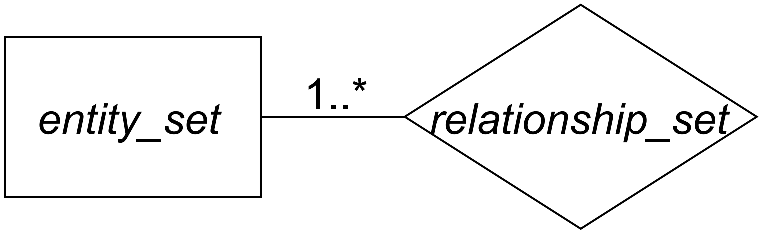
Diagramming Participation

- Can indicate participation constraints in entity-relationship diagrams
  - Partial participation shown with a single line
  - Total participation shown with a double line



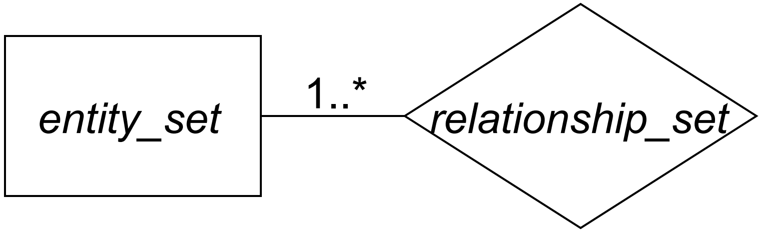
Numerical Constraints

- Can also state numerical participation constraints
  - Specifies how many different relationship instances each entity in the entity-set can participate in
  - Indicated on link between entity and relationship
- Form: lower..upper
  - * means "unlimited"
  - 1..* = one or more
  - 0..3 = between zero and three, inclusive
  - etc.



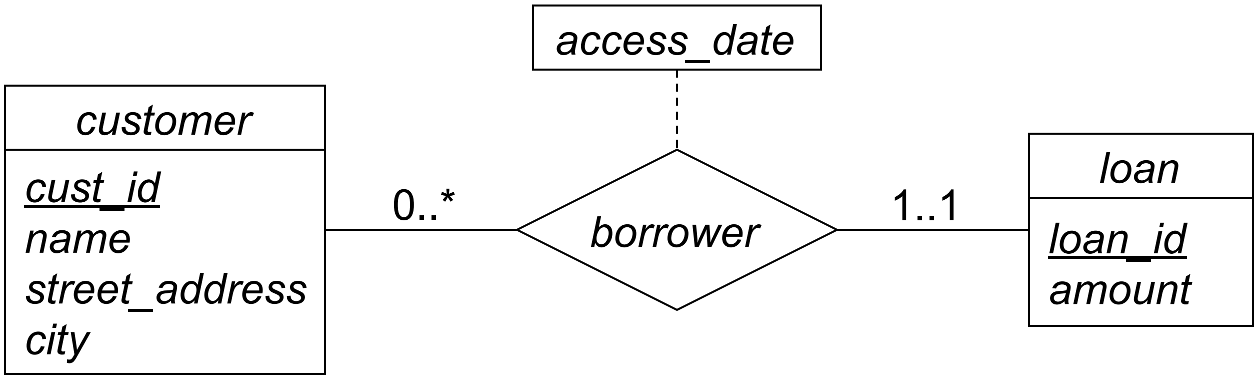
Numerical Constraints (2)

- Can also state mapping constraints with numerical participation constraints
- Total participation:
  - Lower bound at least 1
- Partial participation:
  - Lower bound is 0



Numerical Constraint Example

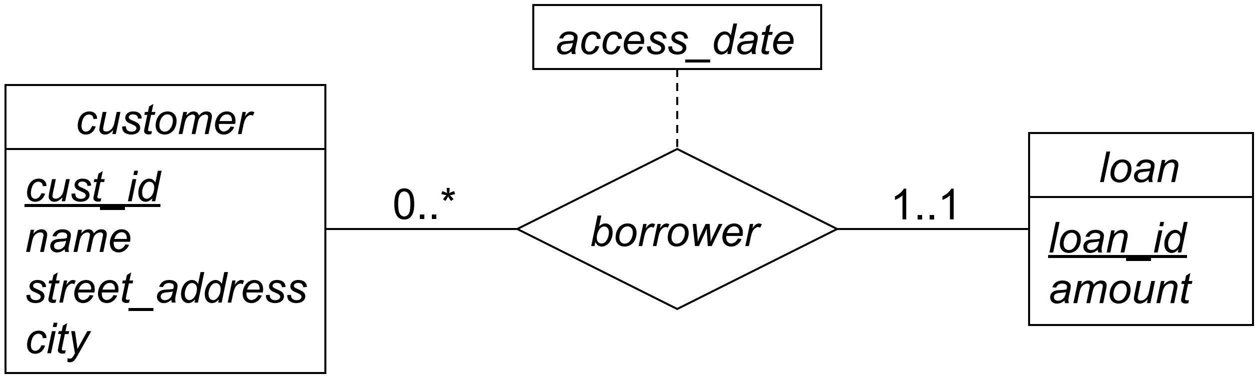
- What does this mean?



- Each *customer* entity may participate in zero or more relationships in this relationship-set
  - *A customer can have zero or more loans.*
- Each *loan* entity must participate in *exactly one* relationship (no more, no less) in this relationship-set
  - *Each loan must be owned by exactly one customer.*

Numerical Constraint Example (2)

- What is the mapping cardinality of *borrower* ?



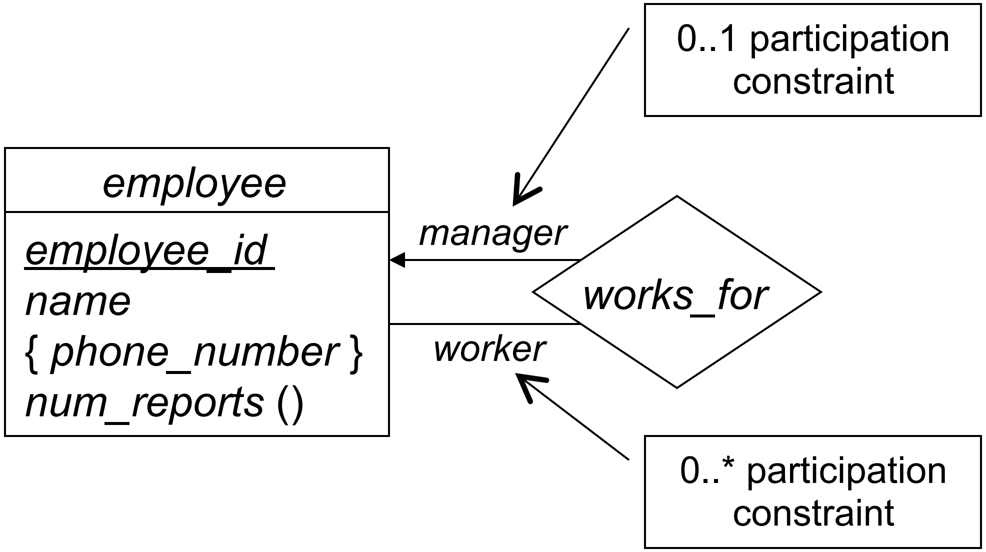
- From last slide:
  - *A customer can have zero or more loans*
  - *Each loan must be owned by exactly one customer.*
- This is a <u>one-to-many</u> mapping from *customer* to *loan*

Diagramming Roles

- Entities have roles in relationships
  - An entity's role indicates the entity's function in the relationship
  - e.g. role of customer in *borrower* relationship-set is that they own the loan
- Sometimes roles are ambiguous
  - e.g. when the same kind of entity is involved in a relationship multiple times
- Example: *works_for* relationship
  - Relationship is between two *employee* entities
  - One is the *manager*; the other is the *worker*

Diagramming Roles (2)

- If roles need to be indicated, put labels on the lines connecting entity to relationship



- *works_for* relationship-set is one-to-many from managers to workers

Weak Entity-Sets

- Sometimes an entity-set doesn't have distinguishing attributes
  - Can't define a primary key for the entity-set!
  - Called a <u>weak entity-set</u>
- Example:
  - Checking accounts have a unique account number
  - Checks have a check number
    - Unique for a given account, but not across all accounts!
    - Number only makes sense in context of a particular account
  - Want to store check transactions in the database

Weak Entity-Sets (2)

- Weak entity-sets *must* be associated with another (strong) entity-set
  - Called the <u>identifying entity-set</u>, or <u>owner entity-set</u>
  - The identifying entity-set <u>owns</u> the weak entity-set
  - Association called the <u>identifying relationship</u>
- Every weak entity *must* be associated with an identifying entity
  - Weak entity's participation in relationship-set is total
  - The weak entity-set is <u>existence dependent</u> on the identifying entity-set
  - If the identifying entity is removed, its weak entities should also cease to exist
  - *(this is where cascade-deletes may be appropriate…)*

Weak Entity-Set Keys

- Weak entity-sets don't have a primary key
  - Still need to distinguish between weak entities associated with a particular strong entity
- Weak entities have a discriminator
  - A set of attributes that distinguishes between weak entities associated with a strong entity
  - Also known as a partial key
- Checking account example:
  - The check number is the discriminator for check transactions

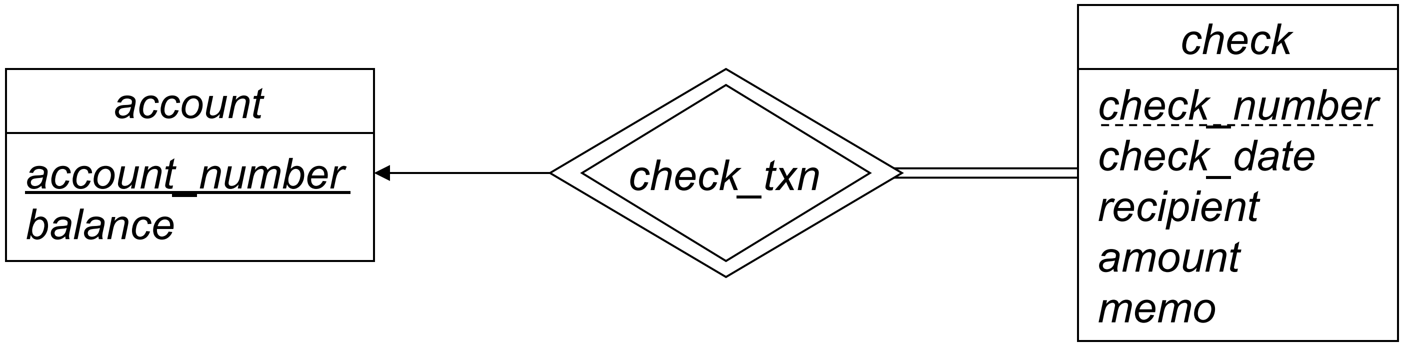
Weak Entity-Set Keys (2)

- Using discriminator, can define a primary key for weak entity-sets
- For a weak entity-set *W*, and an identifying entity-set *S*, primary key of *W* is:

  primary_key(*S*) ∪ *discriminator*(*W*)
- Checking account example:
  - *account_number* is primary key for checking accounts
  - *check_number* is discriminator (partial key) for checks
  - Primary key for check transactions would be (*account_number*, *check_number*)

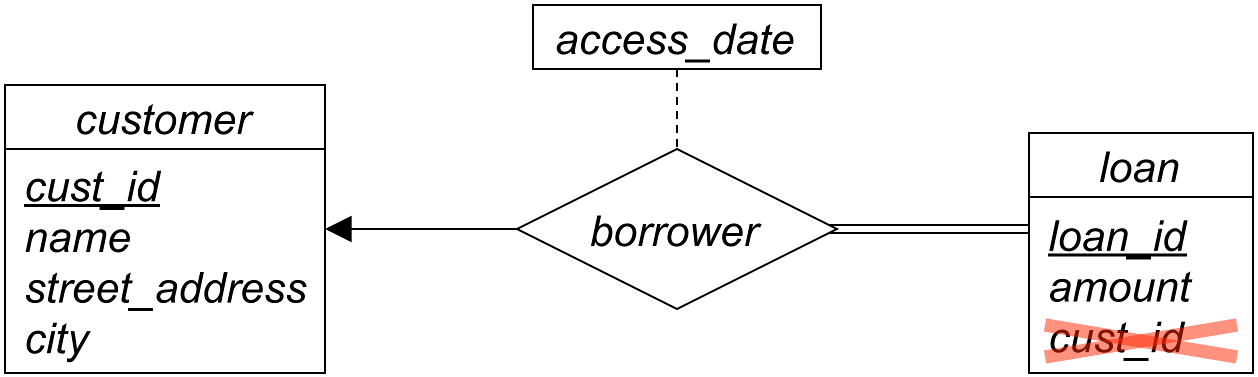
Diagramming Weak Entity-Sets

- Weak entity-sets drawn similarly to strong entity-sets
  - Difference: discriminator attributes are underlined with a dashed underline
- Identifying relationship to the owning entity-set is indicated with a double diamond
  - One-to-many mapping
  - Total participation on weak entity side



Common Attribute Mistakes

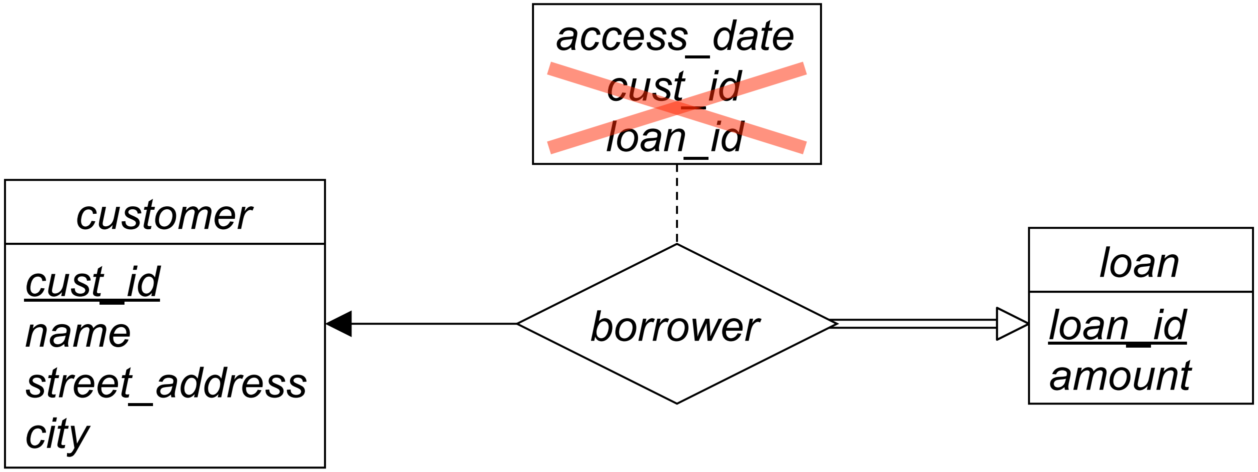
- Don't include entity-set primary key attributes on other entity-sets!
  - e.g. customers and loans, in a one-to-many mapping



- Even if every loan is owned by only one customer, this is still wrong
  - The association is recorded by the *relationship*, so specifying foreign key attributes on the entity-set is redundant

Common Attribute Mistakes (2)

- Don't include primary key attributes as descriptive attributes on relationship-set, either!
- This time, assume *borrower* is a 1:1 mapping
  - IDs used as descriptive attributes on *borrower*



- Again, this is implicit in the relationship

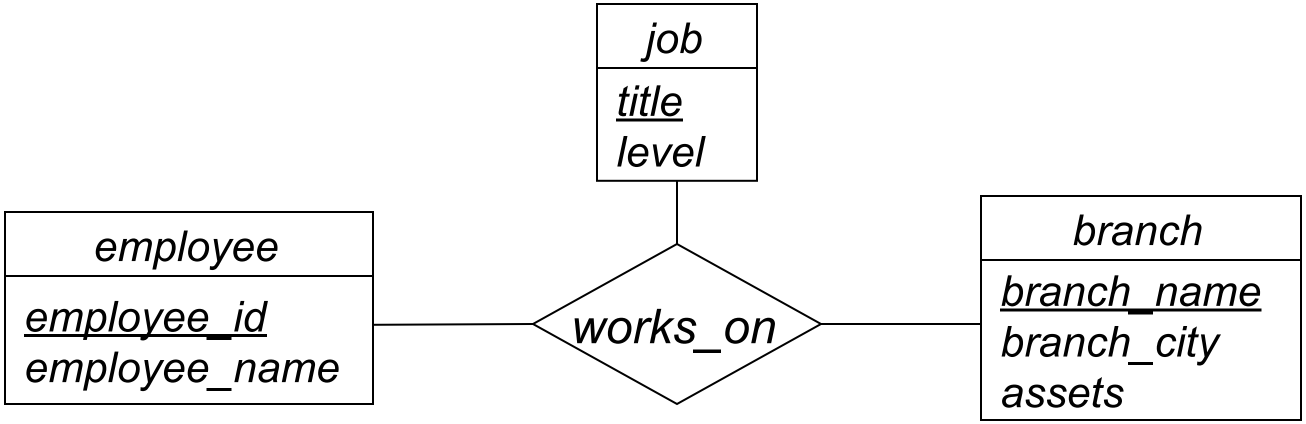
ENTITY-RELATIONSHIP MODEL III

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 16

N-ary Relationships

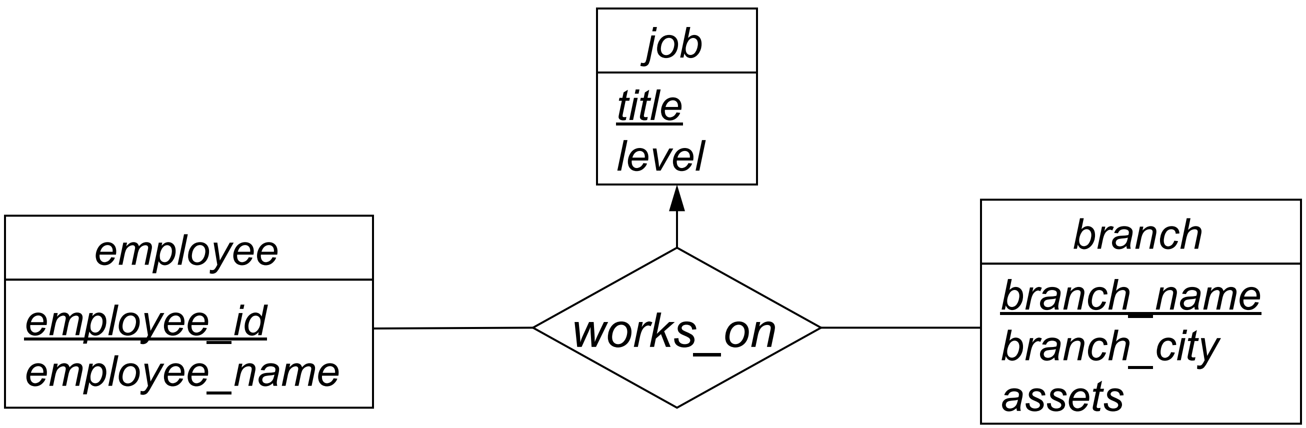
- Can specify relationships of degree > 2 in E-R model
- Example:



  - Employees are assigned to jobs at various branches
  - 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

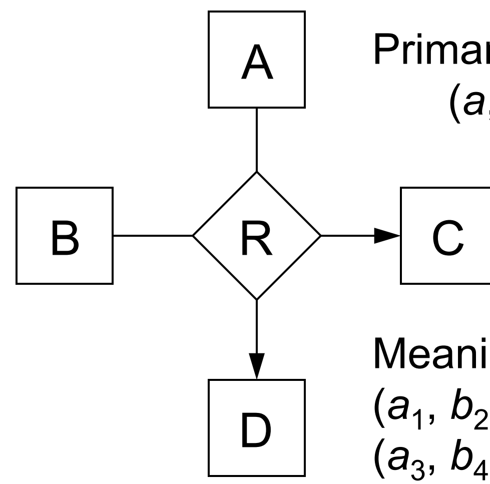
- Can specify *some* mapping cardinalities on relationships with degree > 2
- Each combination of employee and branch can only be associated with one job:



  - Each employee can have only one job at each branch

N-ary Mapping Cardinalities (2)

- For degree > 2 relationships, we only allow at most <u>one</u> edge with an arrow
- <u>Reason</u>: multiple arrows on N-ary relationship-set is ambiguous
  - (several meanings have been defined for this in the past)
- Relationship-set R associating entity-sets A_1, A_2, …, A_n
  - No arrows on edges A_1, …, A_i
  - Arrows are on edges to A_{i+1}, …, A_n
- Meaning 1 (the simpler one):
  - A particular combination of entities in A_1, …, A_i can be associated with at most one set of entities in A_{i+1}, …, A_n
  - Primary key of R is union of primary keys from set { A_1, A_2, …, A_i }

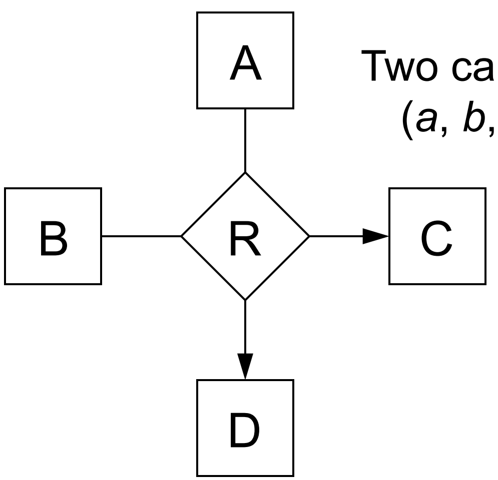


Primary key:
(a, b)

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)

- 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):
  - <u>For each</u> 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 <u>each</u> arrow in N-ary relationship-set
  - <u>For each</u> k ($i < k \leq n$), another candidate key of R is union of primary keys from entity-sets { $A_1, A_2, \ldots, A_{k\text{-}1}, A_{k+1}, \ldots, A_n$ }



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)~~
…

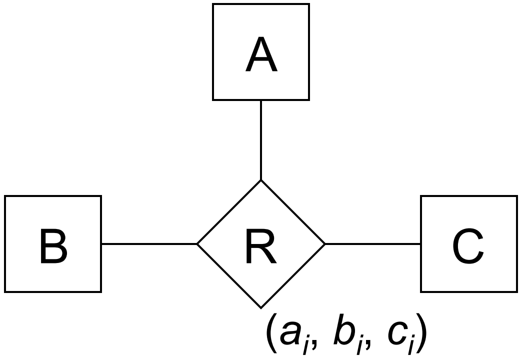
<u>All</u> disallowed by meaning 1!

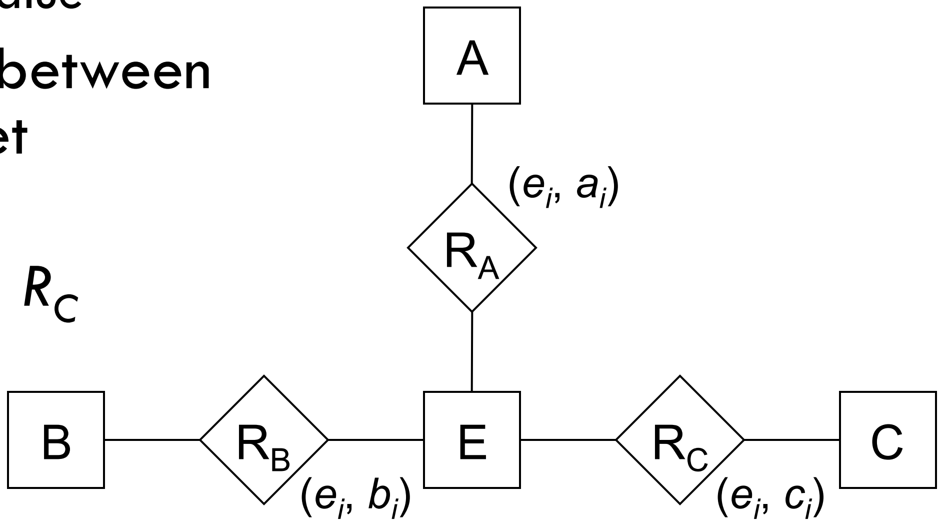
N-ary Mapping Cardinalities (4)

- 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

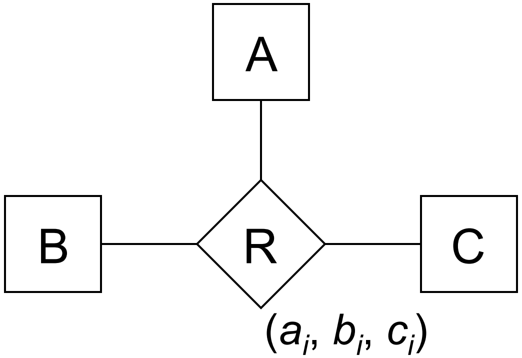
- Often have only binary relationships in DB schemas
- For degree > 2 relationships, *could* replace with binary relationships
  - Replace N-ary relationship-set with a new entity-set *E*
    - Create an identifying attribute for *E*
    - e.g. an auto-generated ID value
  - Create a relationship-set between *E* and each other entity-set
  - Relationships in *R* must be represented in R_A, R_B, and R_C

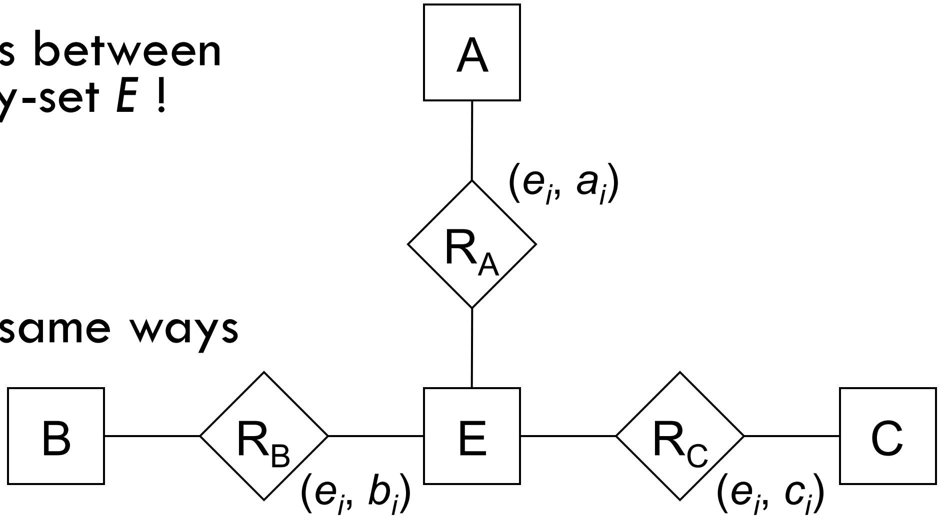




Binary vs. N-ary Relationships (2)

- **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*
- 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



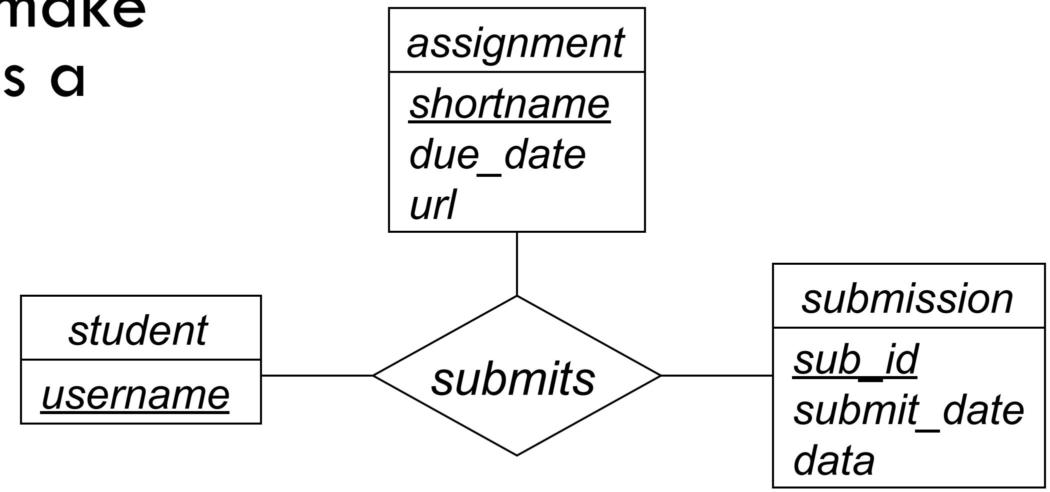


Binary vs. N-ary Relationships (3)

- Using binary relationships is sometimes more intuitive for particular designs
- Example: office-equipment inventory database
  - 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!

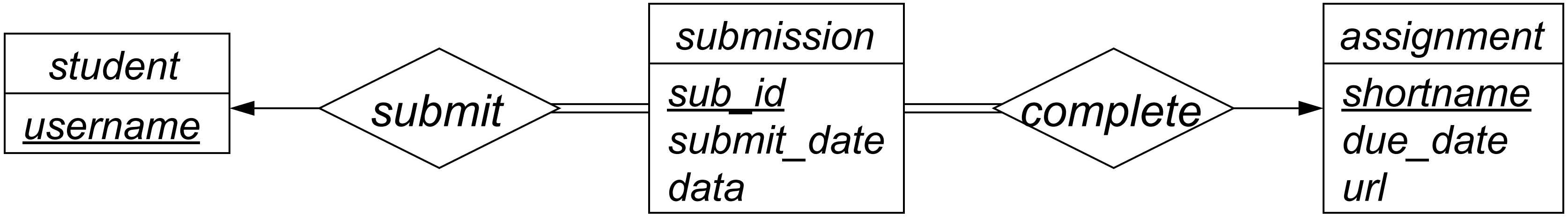
Course Database Example

- What about this case:
  - 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)

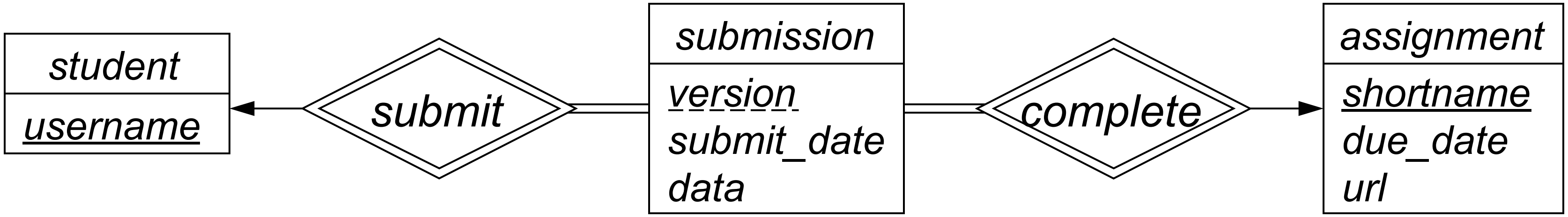
- Other ways to represent students, assignments and submissions?
- Can also represent as two binary relationships



- 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

Course Database Example (3)

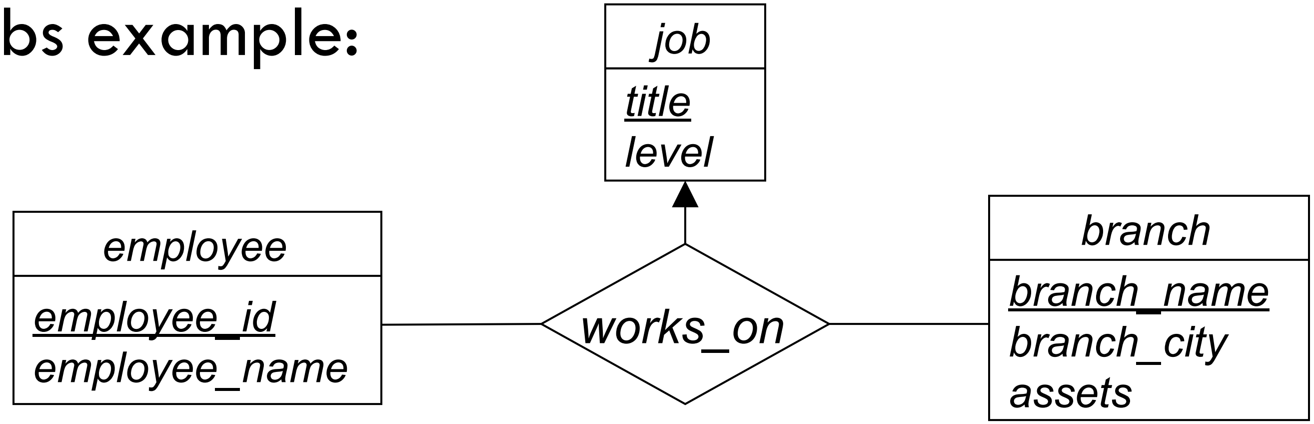
- Could even make *submission* a weak entity-set
  - Both *student* and *assignment* are identifying entities!



- Discriminator for *submission* is version number
- Primary key for *submission* ?
  - Union of primary keys from all owner entity-sets, plus discriminator
  - (*username*, *shortname*, *version*)

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 cannot construct the same constraint using only binary relationships
    - *(Reason is related to issue identified on slide 8)*

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:
  - Specify schema of the relation itself
  - Specify primary key on the relation
  - Specify any foreign key references to other relations

Strong Entity-Sets

- 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*

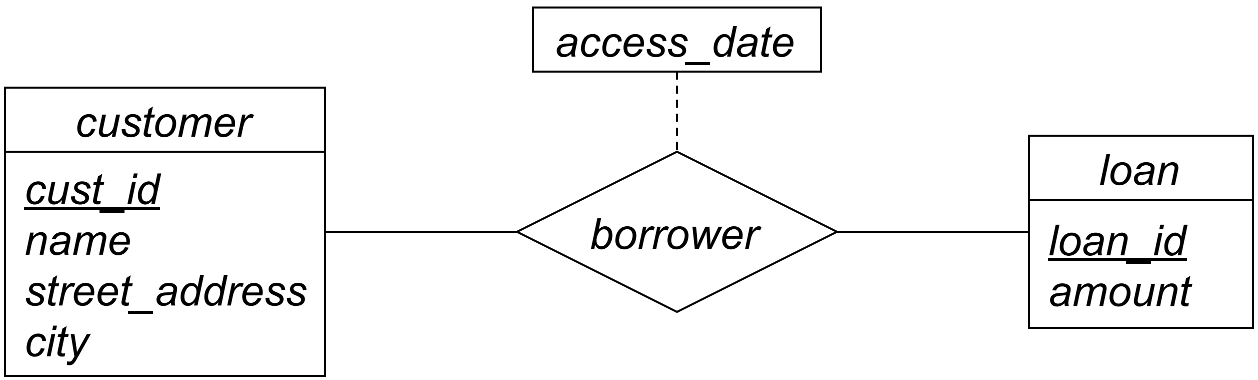
location
<u>*latitude*</u> <u>*longitude*</u> *description* *last_visited*

- Convert to relation schema:

location(<u>*latitude*</u>, <u>*longitude*</u>, *description*, *last_visited*)

Entity-Set Examples (2)

- E-R diagram for customers and loans:



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

  customer(<u>*cust_id*</u>, *name*, *street_address*, *city*)

  loan(<u>*loan_id*</u>, *amount*)

Relationship-Sets

- 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
    - *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 <u>both</u> candidate keys in DB schema!

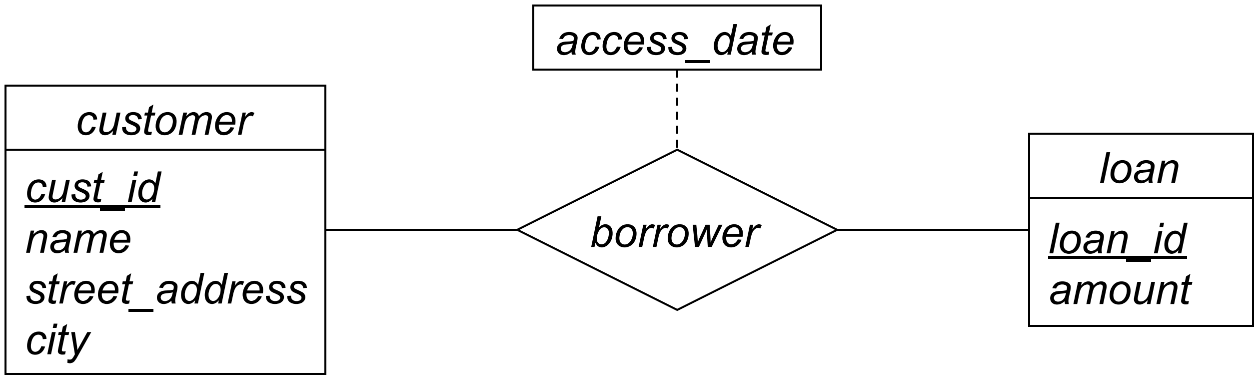
Relationship-Sets: Primary Keys (2)

- 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*) ∪ *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

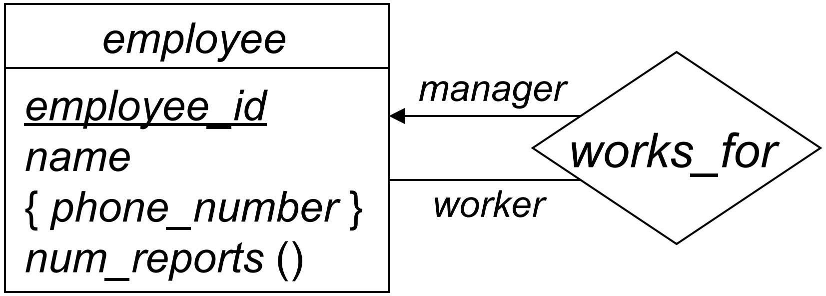
- Relationship-sets associate entities in entity-sets
  - We need foreign-key constraints on relation schema for *R* !
- For each entity-set E_i participating in *R* :
  - Relation schema for *R* has a foreign-key constraint on E_i relation, for *primary_key*(E_i) attributes
- Relation schema notation 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 <u>very</u> good idea
  - Can specify both foreign key constraints and candidate keys in the SQL DDL

Relationship-Set Example



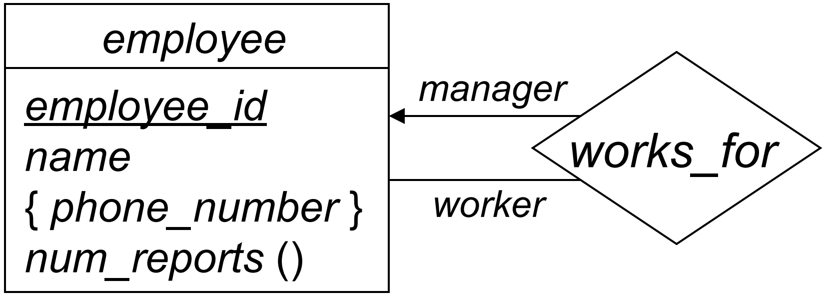
- Relation schema for *borrower*:
  - Primary key of *customer* is *cust_id*
  - Primary key of *loan* is *loan_id*
  - Descriptive attribute *access_date*
  - *borrower* mapping cardinality is many-to-many
  - Result: *borrower*(<u>*cust_id*</u>, <u>*loan_id*</u>, *access_date*)

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)



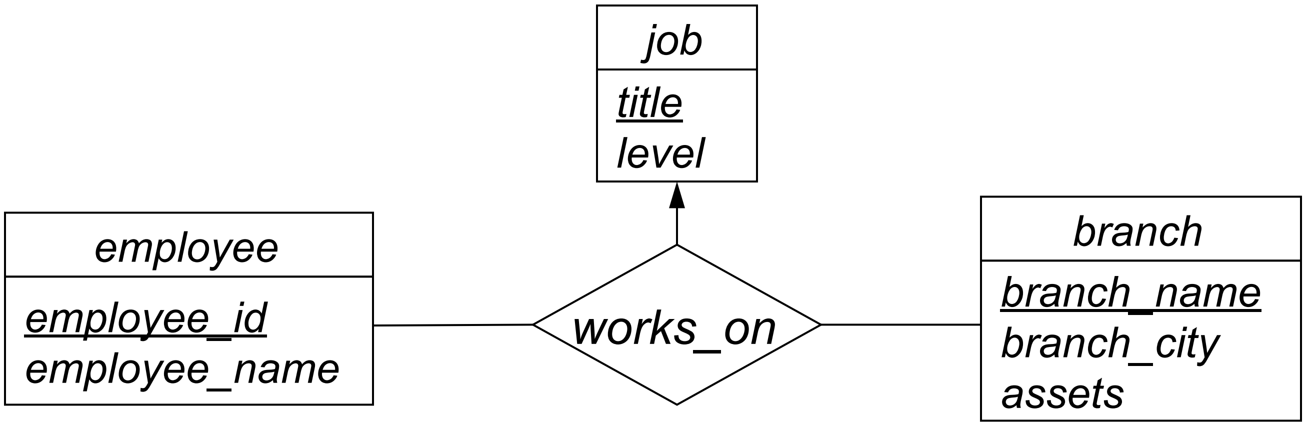
- Relation schema for *employee* entity-set:
  - (For now, ignore *phone_number* and *num_reports*…)

    employee(*<u>employee_id</u>*, *name*)
- Relation schema for *works_for*:
  - One-to-many mapping from *manager* to *worker*
  - "Many" side is used for primary key
  - Result: *works_for*(*<u>employee_id</u>*, *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 <u>all</u> 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*)

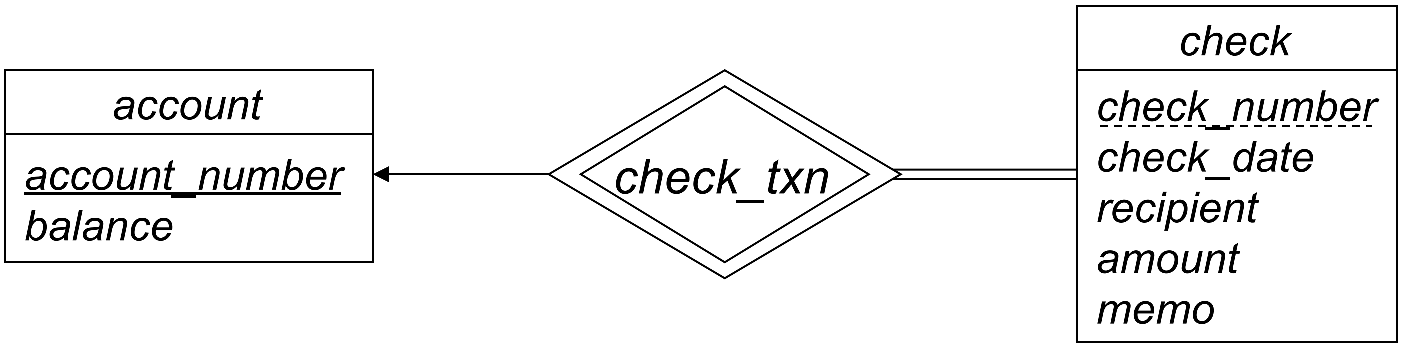
Weak Entity-Sets

- 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* has a foreign key constraint on $primary\_key(B)$, to *B*

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



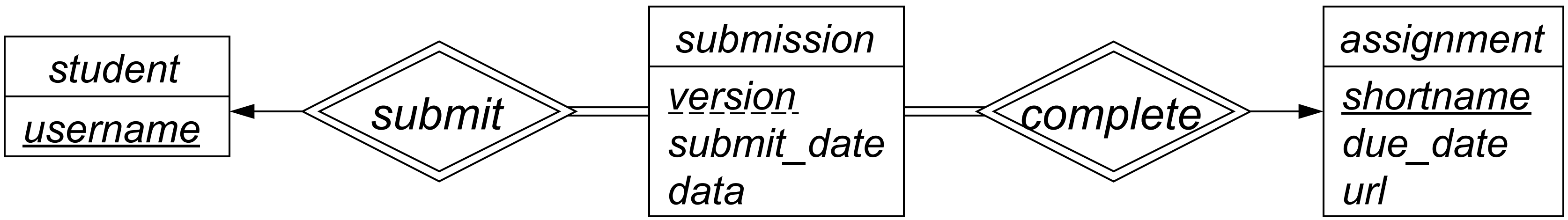
- *account* schema:

  account(<u>*account_number*</u>, *balance*)

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

  check(<u>*account_number*</u>, <u>*check_number*</u>, *check_date*, *recipient*, *amount*, *memo*)

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

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:

customer

cust_id
name
address
    street
    city
    state
    zip_code

- 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 <u>once</u>
- 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: $primary\_key(E) \cup A$
  - Primary key of *R* includes <u>all</u> 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:

customer

cust_id
name
{ *address*
    street
    city
    state
    zip_code }

- 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

ADVANCED E-R FEATURES

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 17

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 <u>specialization</u> to represent such entity-sets
- Example: bank account categories
  - Checking accounts
  - Savings accounts
  - Have common features, but also unique attributes

Generalization and Specialization

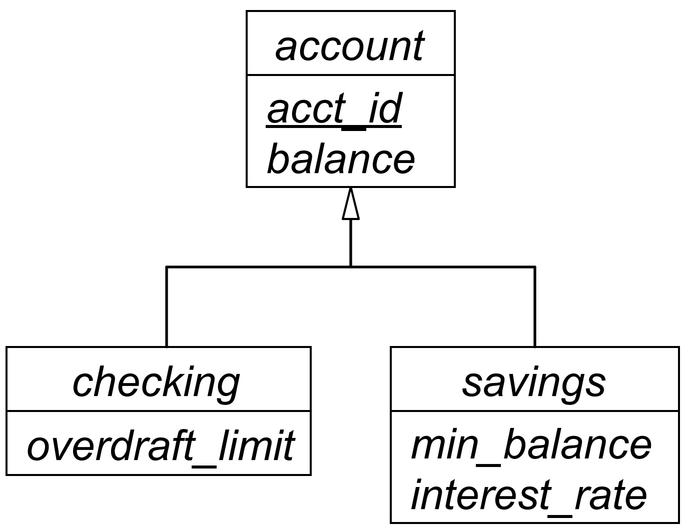
- Generalization: a “bottom up” approach
  - Taking similar entity-sets and unifying their common features
  - Start with specific entities, then create generalizations from them
- Specialization: a “top down” approach
  - Creating general purpose entity-sets, then providing specializations of the general 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)

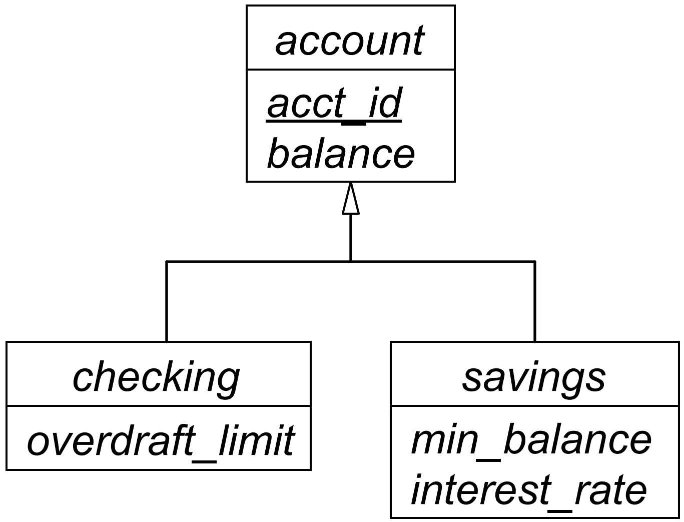
- Create entity-set to represent common attributes
  - Called the superclass, or higher-level entity-set
- Create entity-sets to represent specializations
  - Called subclasses, or lower-level entity-sets
- Join superclass to subclasses with hollow-head arrow(s)



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 <u>hierarchy</u>
- Can also inherit from multiple superclasses
  - Entity-sets form a <u>lattice</u>
  - 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

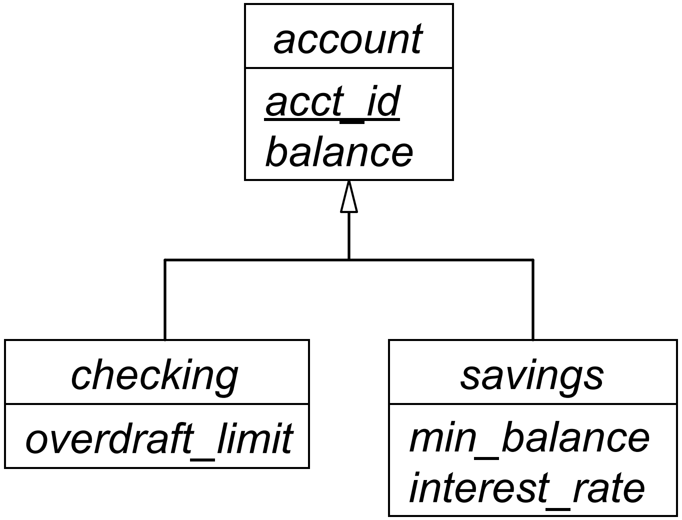
Disjointness Constraints

- "An account cannot be *both* a checking account and a savings account."
- An entity may belong to at most <u>one</u> 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 <u>disjoint specialization</u>
- If an entity can be a member of multiple lower-level entity-sets:
  - Called an <u>overlapping specialization</u>

Disjointness Constraints (2)

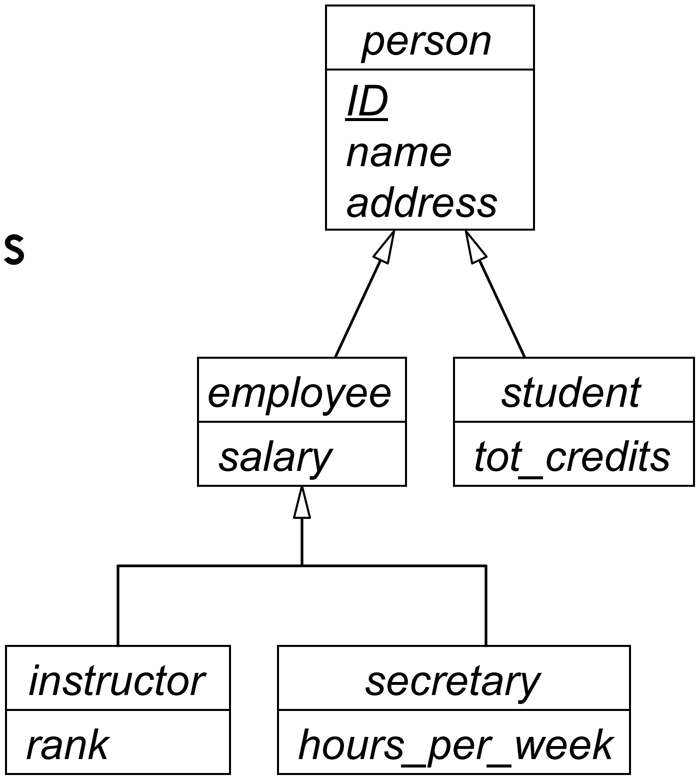
- How the arrows are drawn indicates whether the specialization is disjoint or overlapping

- 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 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

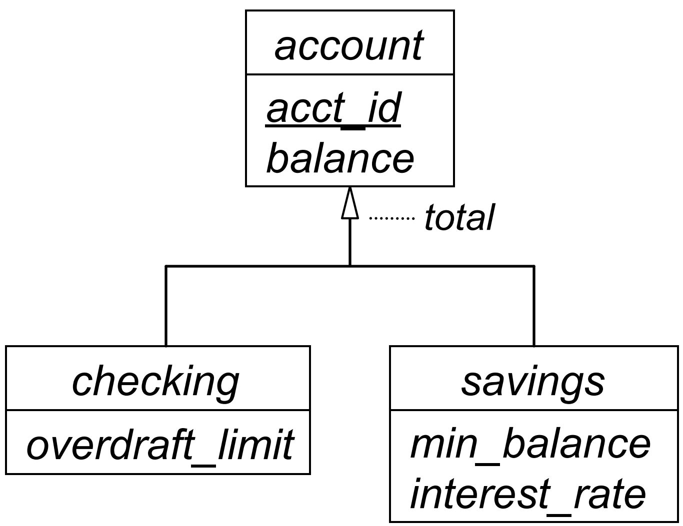


Completeness Constraints

- "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 <u>total</u> specialization
- If entities in higher-level entity-set aren't required to be members of lower-level entity-sets:
  - Called <u>partial</u> specialization
- *account* specialization is a <u>total</u> specialization

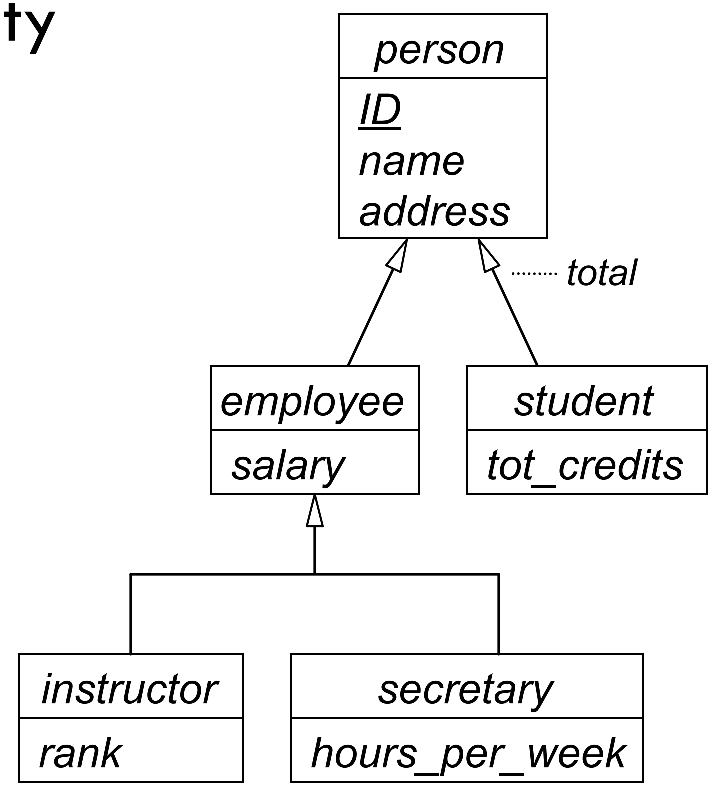
Completeness Constraints (2)

- Default constraint is <u>partial</u> specialization
- Specify total specialization constraint by annotating the specialization arrow(s)
- Updated bank account diagram:



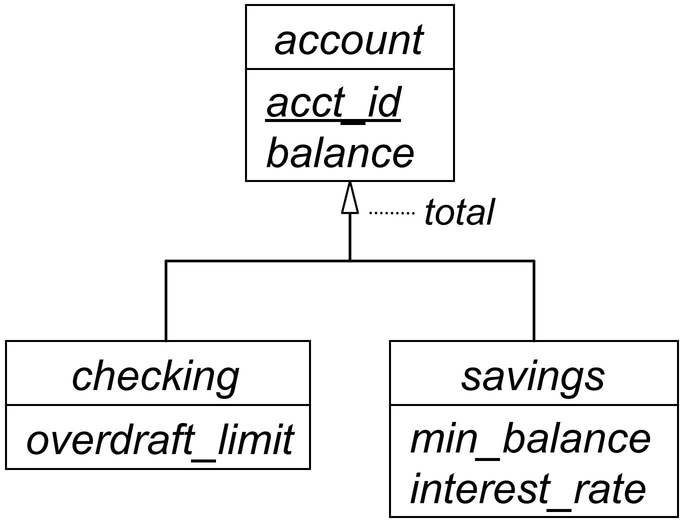
Completeness Constraints (3)

- 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

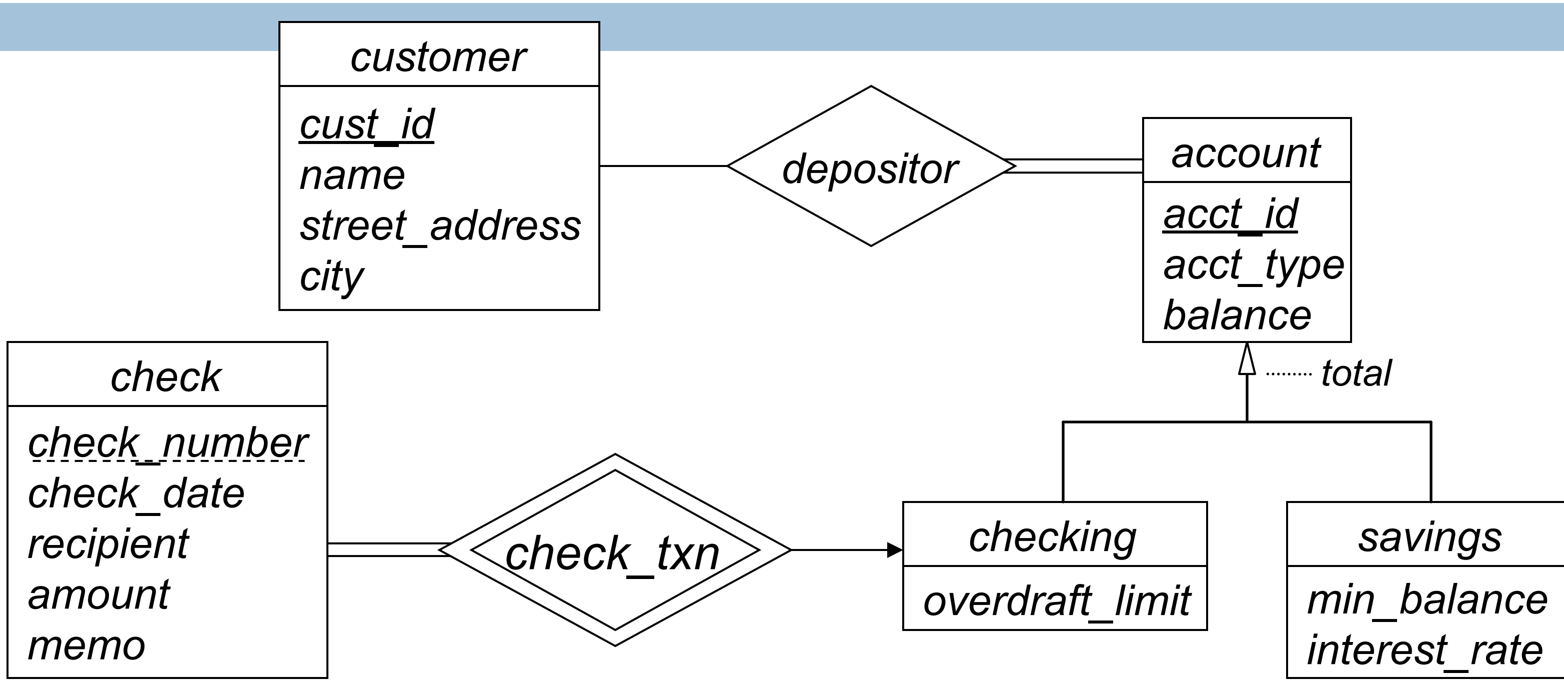
Membership Constraints

- 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 simply be assigned to lower-level entity-sets by a database user
  - No explicit predicate governs membership
  - Called <u>user-defined</u> membership
- Generally used when an entity's membership could change in the future

Final Bank Account Diagram

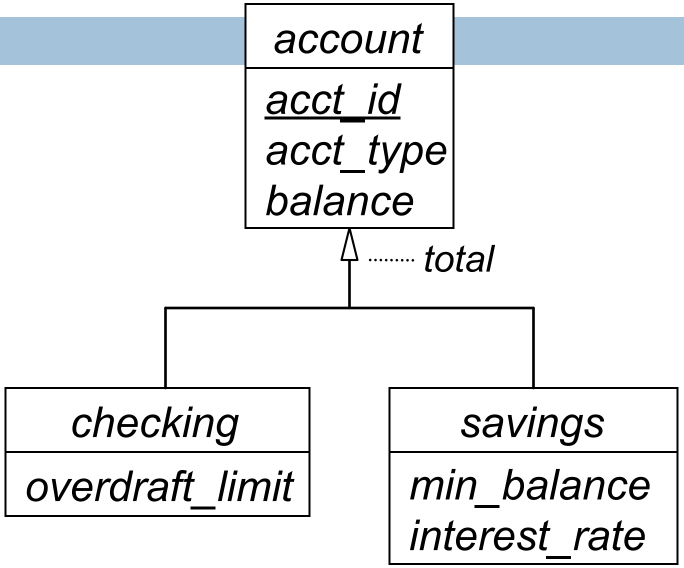


- Would also create relationship-sets against various entity-sets in hierarchy
  - associate *customer* with *account*
  - associate *check* weak entity-set with *checking*

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

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

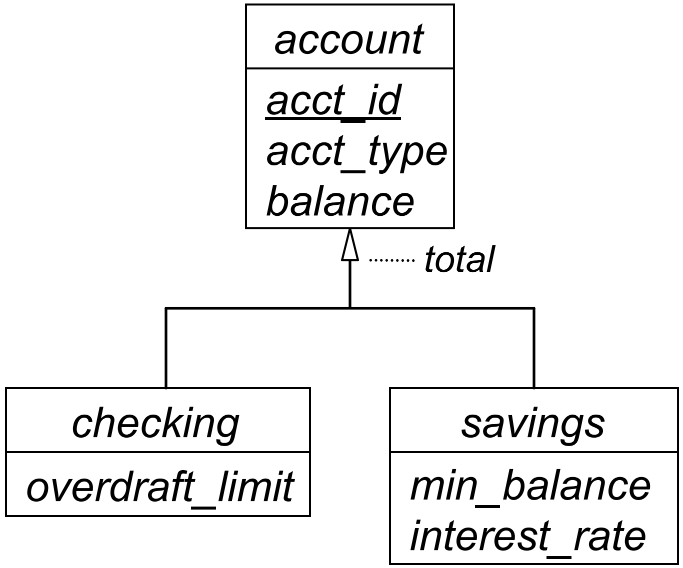
  savings(*<u>acct_id</u>*, *min_balance*, *interest_rate*)

  - 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(*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 Keys

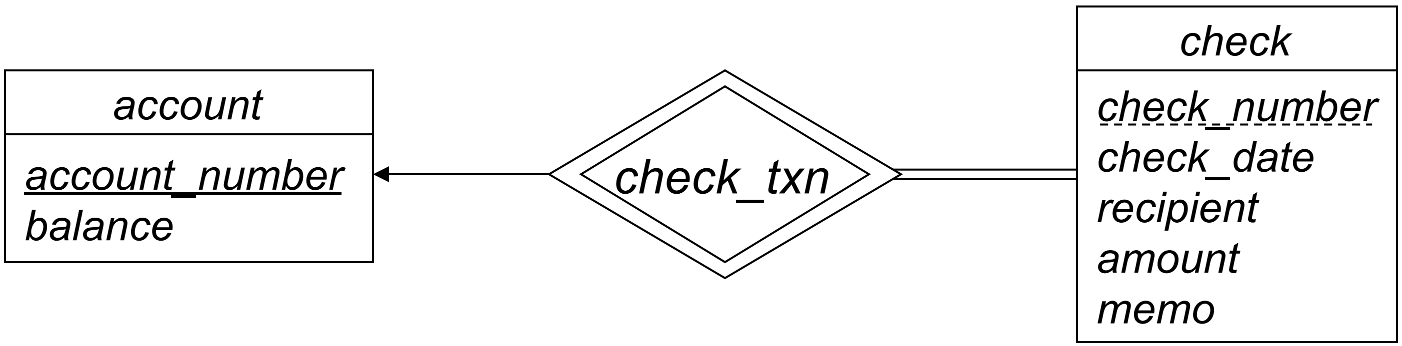
- *Generating* primary key values is actually the easy part
- Most databases provide sequences
  - A source of unique, increasing **INTEGER** or **BIGINT** values
  - Perfect for primary key values
  - Multiple tables can use the same sequence for their primary keys
- PostgreSQL example:

```
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 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:

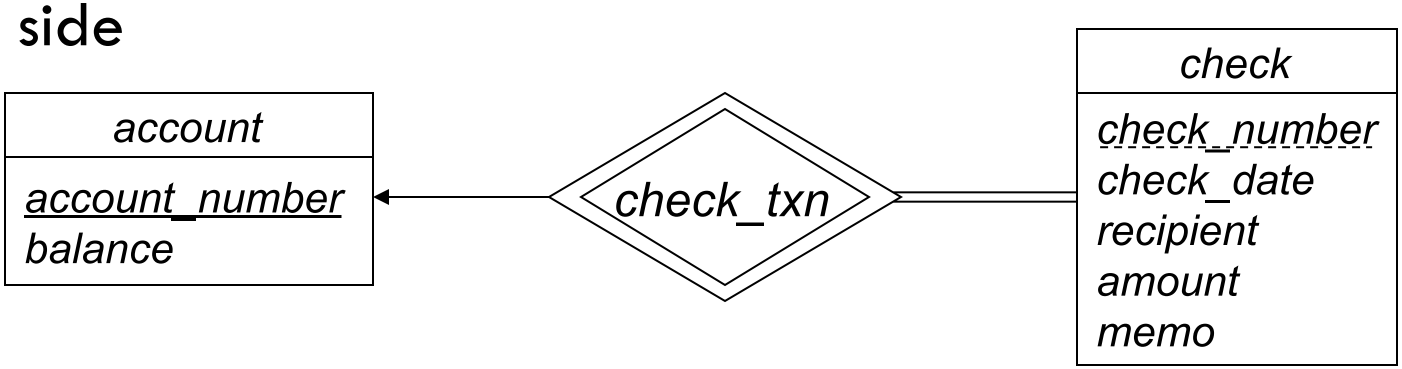
  account(*<u>account_number</u>*, *balance*)

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

  check(*<u>account_number</u>*, *<u>check_number</u>*, *check_date*, *recipient*, *amount*, *memo*)

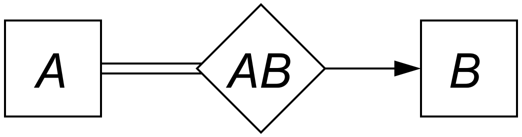
Schema Combination

- 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

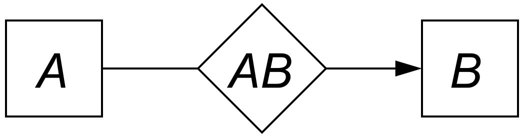


- Generates relation schemas *A*, *B*, *AB*
  - Primary key of *A* is *primary_key*(*A*)
  - Primary key of *AB* is also *primary_key*(*A*)
    - (*A* is on "many" side of mapping)
  - *AB* has foreign key constraints on both *A* and *B*
  - There is one relationship in *AB* for every entity in *A*
- Can combine *A* and *AB* relation schemas
  - Primary key of combined schema still *primary_key*(*A*)
  - Only requires one foreign-key constraint, to *B*

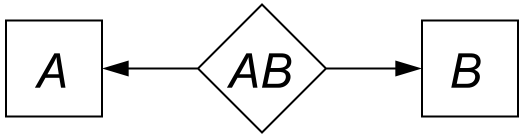
Schema Combination (3)

- 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

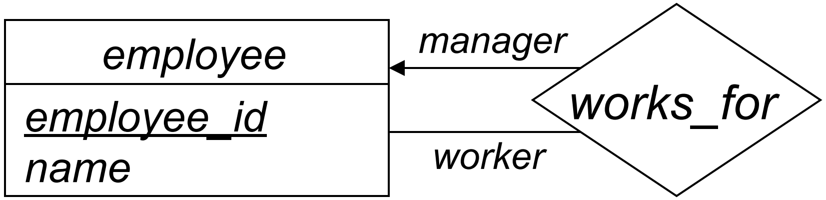


  - Must store *null* values for *primary_key*(*B*) attributes when an entity in *A* maps to no entity in *B*
- If *AB* is one-to-one mapping:



  - 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



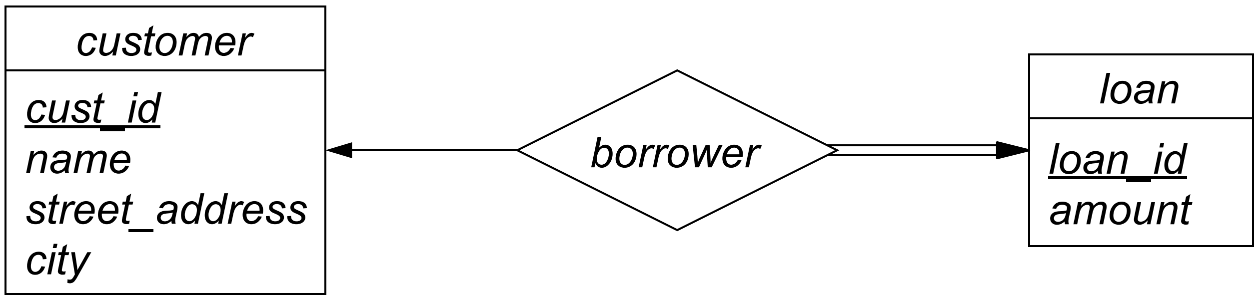
- Manager to worker mapping is one-to-many
- Relation schemas were:

  employee(*employee_id*, *name*)

  works_for(*employee_id*, *manager_id*)
- Could combine into:

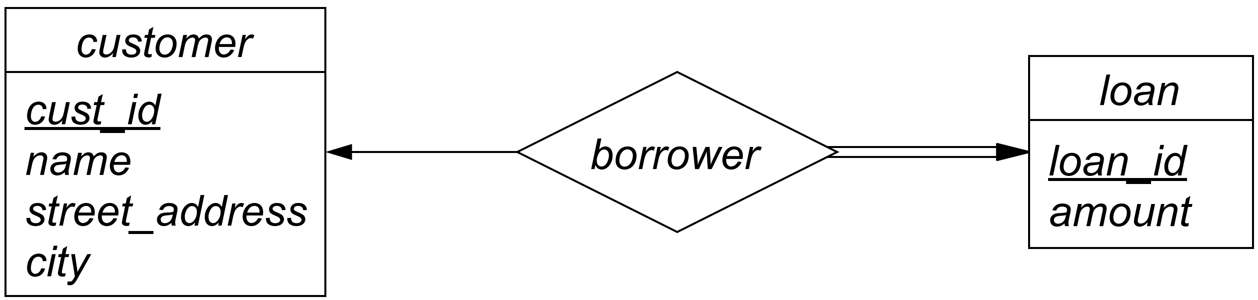
  employee(*employee_id*, *name*, *manager_id*)
  - (A very common schema combination)
  - Need to store *null* for employees with no manager

Schema Combination Example (2)



- One-to-one mapping between customers and loans
  - *customer*(<u>*cust_id*</u>, *name*, *street_address*, *city*)
  - *loan*(<u>*loan_id*</u>, *amount*)
  - *borrower*(<u>*cust_id*</u>, *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)

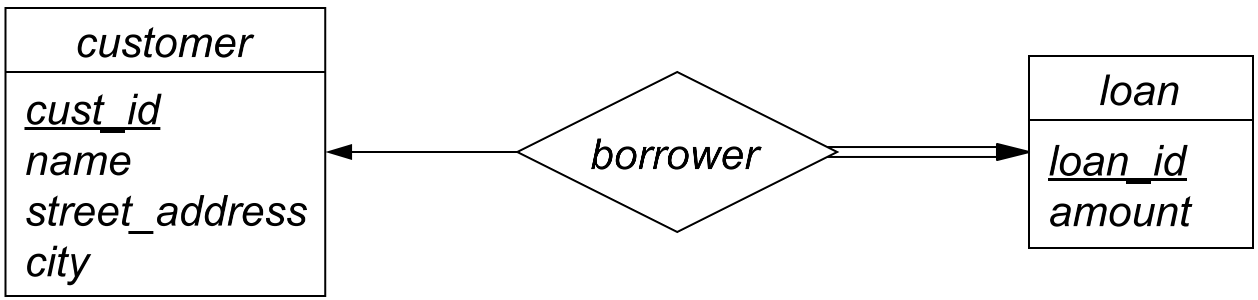


- Participation of *loan* in *borrower* will be total
  - Combining *borrower* into *customer* would require *null* values for customers without loans
- Better to combine *borrower* into *loan* schema

  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