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 specification of functional requirements
  - "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 implementation data model
  - 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
- Very important 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 attributes
  - 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*

customer
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 role 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 <u>ordered by</u> 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 <u>descriptive attributes</u>
  - 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)

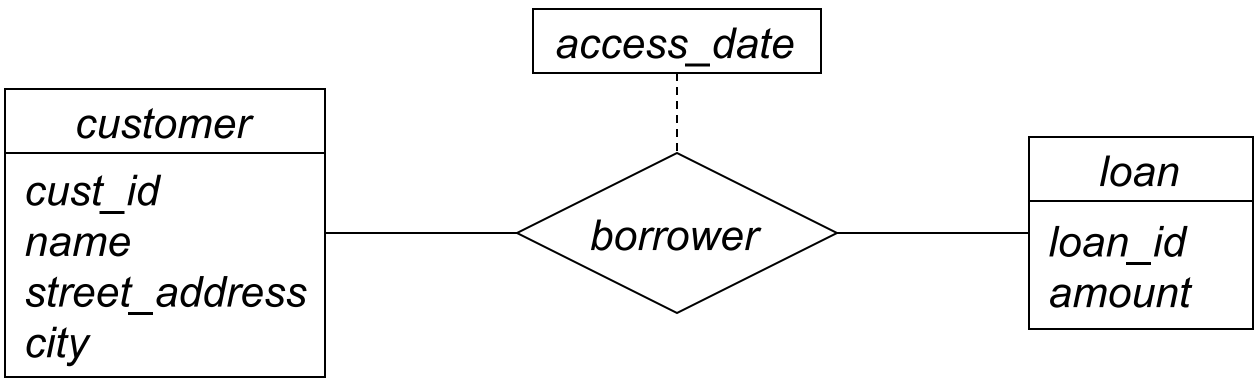
- Specific relationships are identified *only* by the participating entities
  - …not by any relationship attributes!
  - Different relationships are allowed to have the same value for a descriptive attribute
  - (This is why entities in an entity-set must be uniquely identifiable.)
- Given:
  - Entity-sets 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

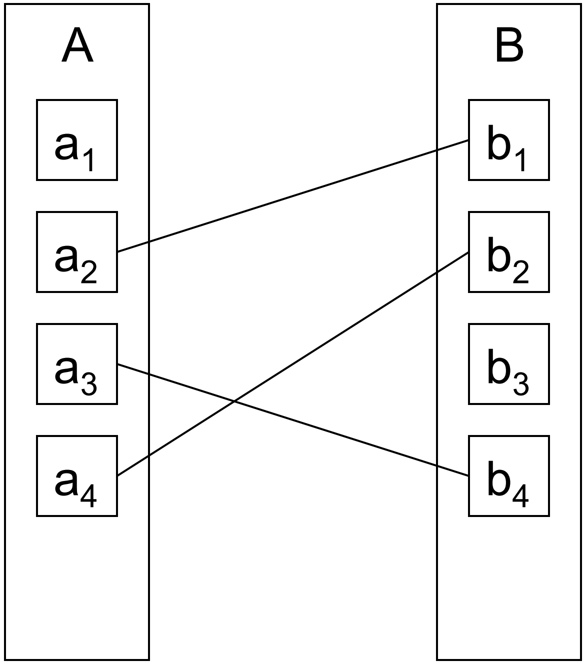
Mapping Cardinalities (2)

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*



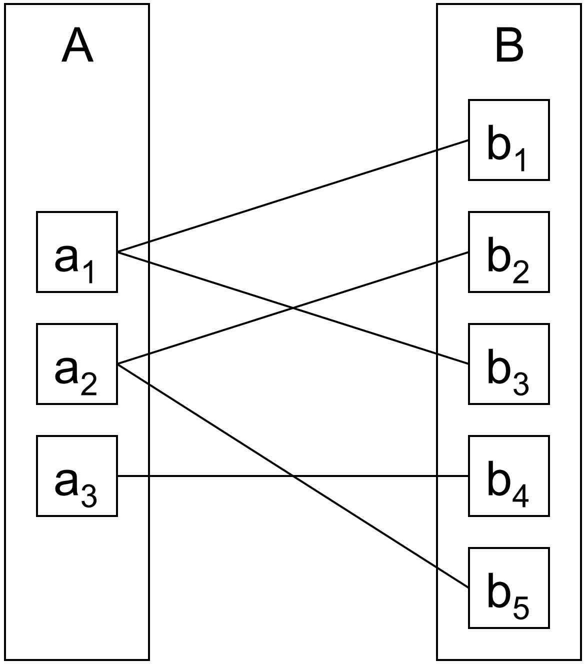
Mapping Cardinalities (3)

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)

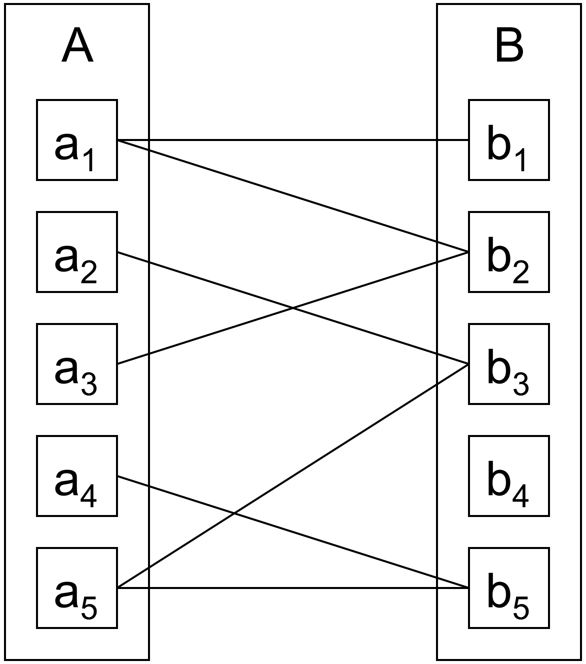
- Opposite of one-to-many
- An entity in *A* is associated with *at most* one entity in *B*
- An entity in *B* is associated with *zero or more* entities in *A*



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

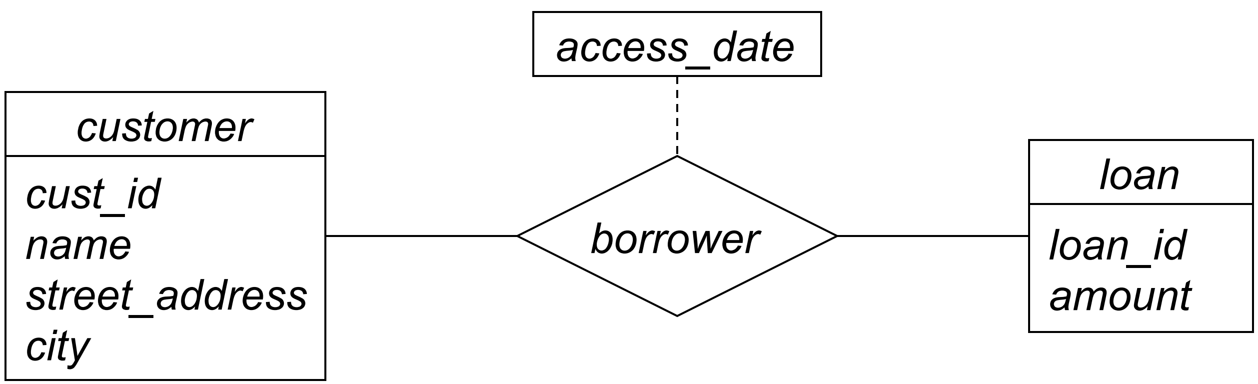
Best choice for *borrower* :

many-to-many mapping

Handles real-world needs!

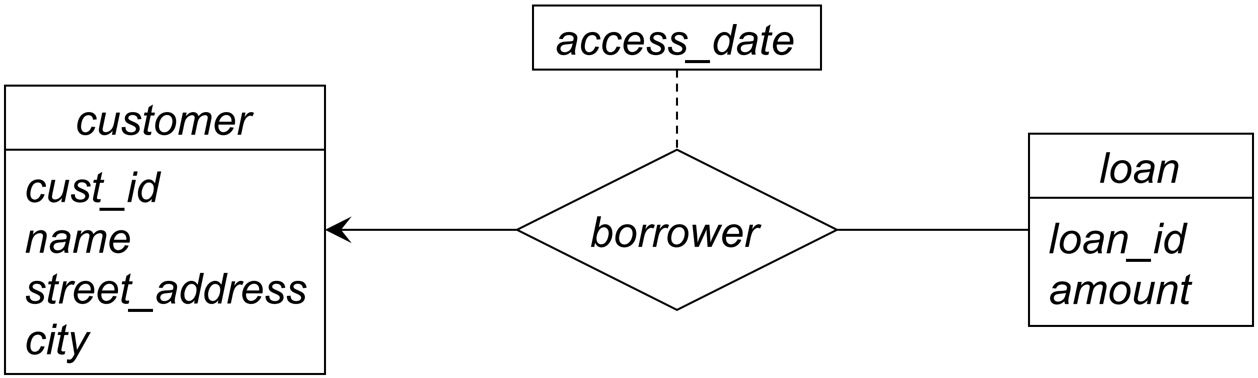
Diagramming Cardinalities

- 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)

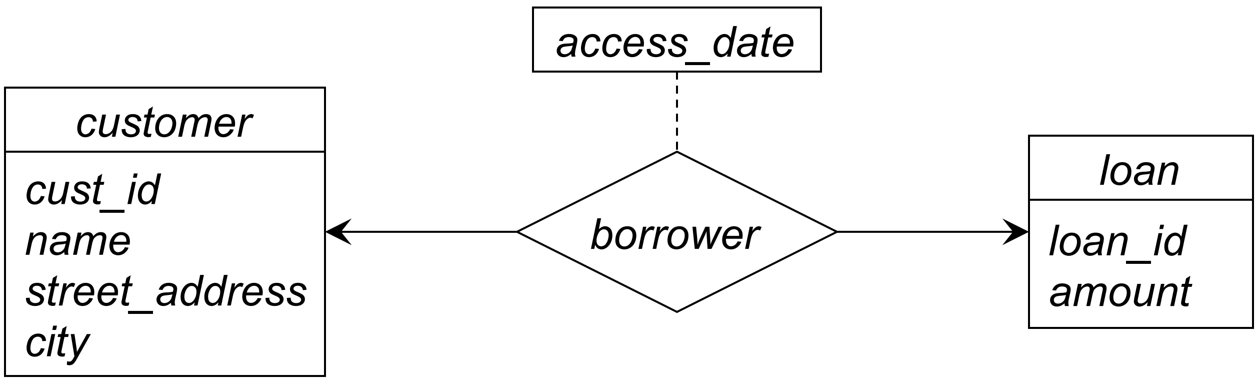
- One-to-many mapping between *customer* and *loan*:



  - Each customer can have multiple loans
  - A loan is owned by exactly one customer
    - (Actually, this is technically "at most 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-set: 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

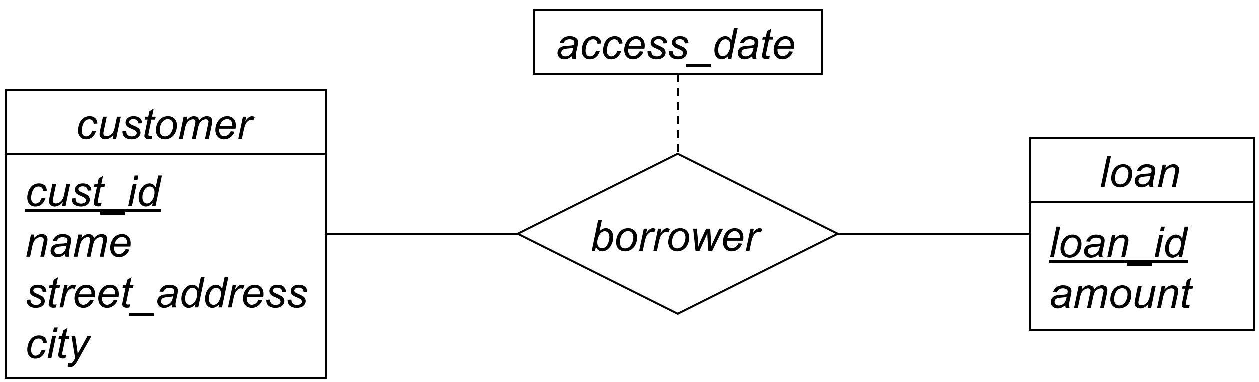
- An entity-set may have multiple candidate keys
- The primary key is the candidate key most often used to reference entities in the set
  - In logical/physical design, primary key values will be used to represent relationships
  - External systems may also use primary key values to reference entities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

Choosing Keys: Performance

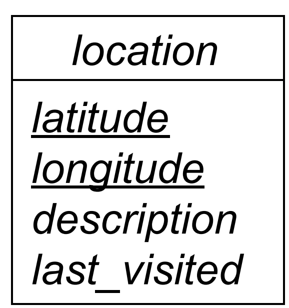
- Large, complicated, or multiple-attribute keys are generally slower
  - Use smaller, single-attribute keys
    - (You can always generate them…)
  - Use faster, fixed-size types
    - e.g. **`INT`** or **`BIGINT`**
- Especially true for primary keys!
  - Values used in both database and in access code
  - 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 $E_1, E_2, \ldots, E_n$ 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:

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

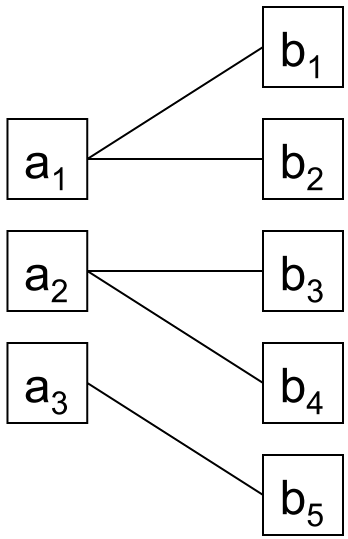
Relationship-Set Primary Keys

- 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)

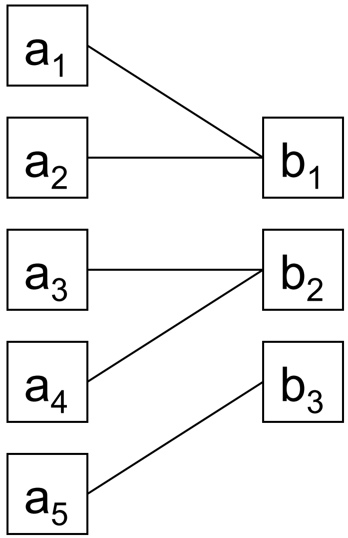
- Relationship-set *R*, involving entity-sets *A* and *B*
  - Individual relationships are described by *primary_key*(*A*) ∪ *primary_key*(*B*)
- If mapping is one-to-many:
  - Entities in *B* associated with *at most* one entity in *A*
  - A given *primary_key*(*A*) value can appear in multiple relationships
  - 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*

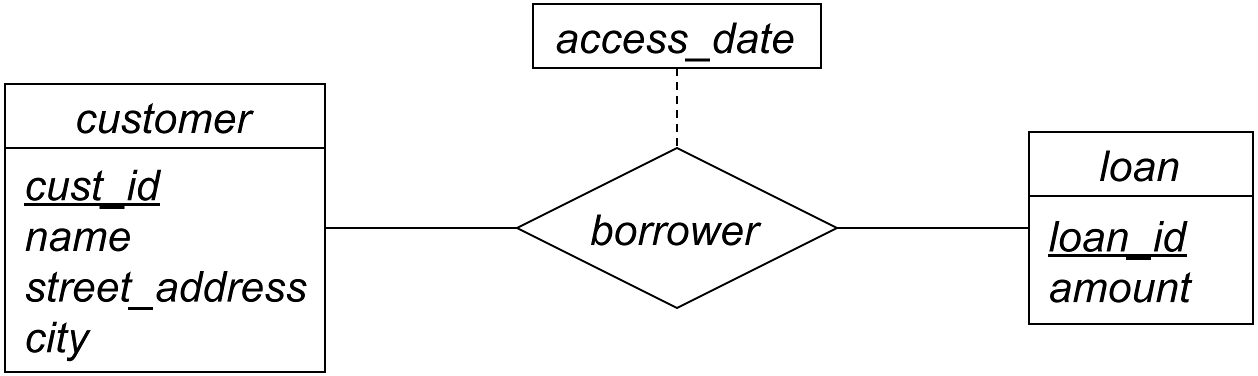


Relationship-Set Primary Keys (4)

- 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 <u>both</u> 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

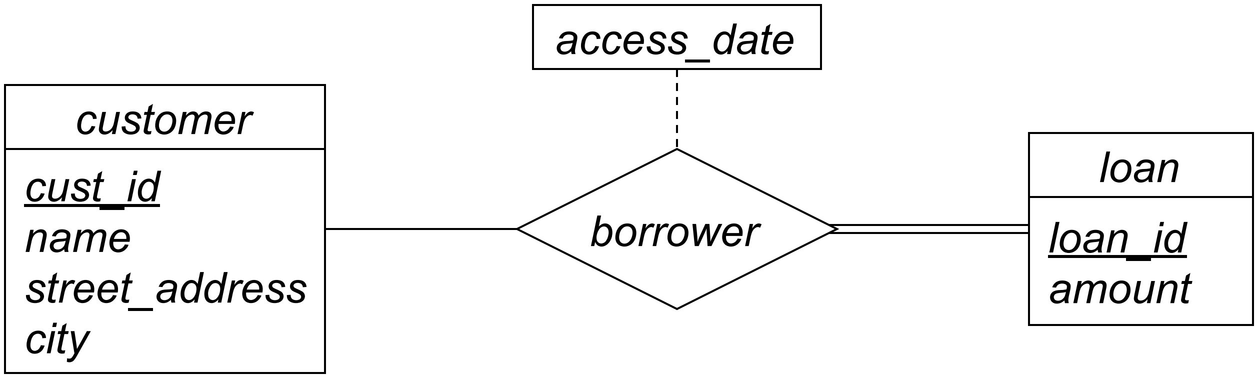
- Given entity-set *E*, relationship-set *R*
  - How many entities in *E* participate in *R* ?
  - In other words, what is minimum number of relationships that each entity in *E* *must* participate in?
- If every entity in *E* participates in at least one relationship in *R*, then:
  - *E*'s participation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

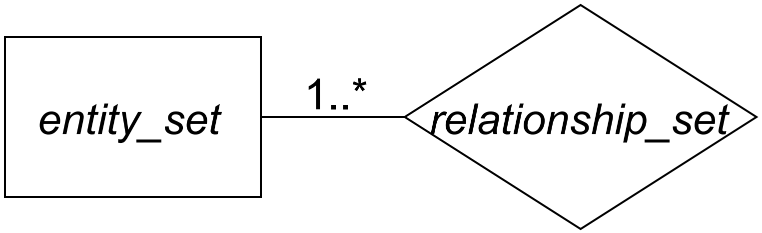
Diagramming Participation

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



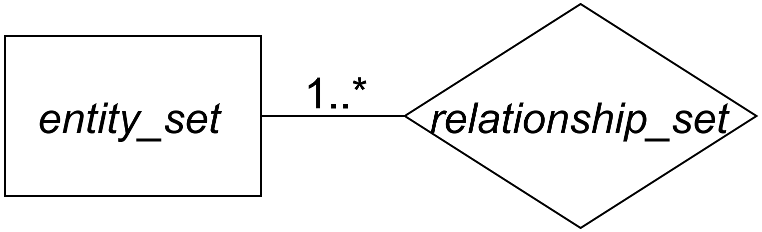
Numerical Constraints

- 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.



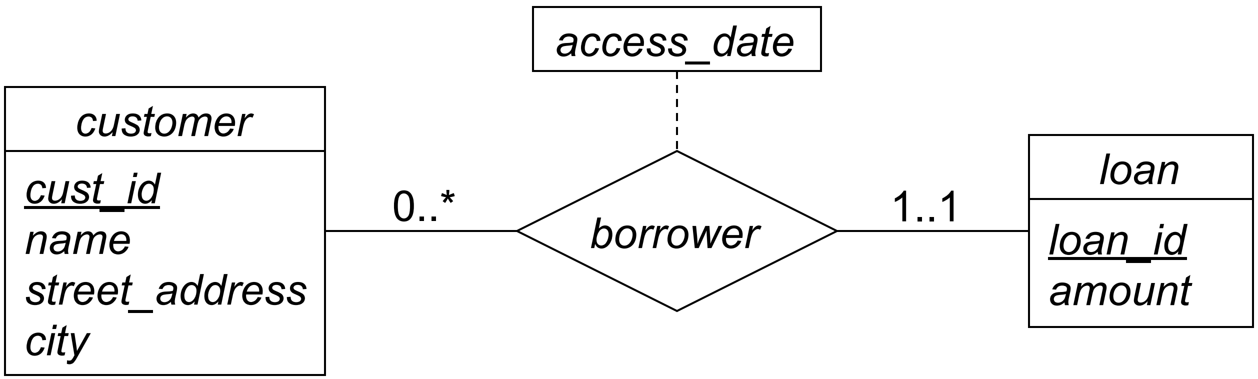
Numerical Constraints (2)

- 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

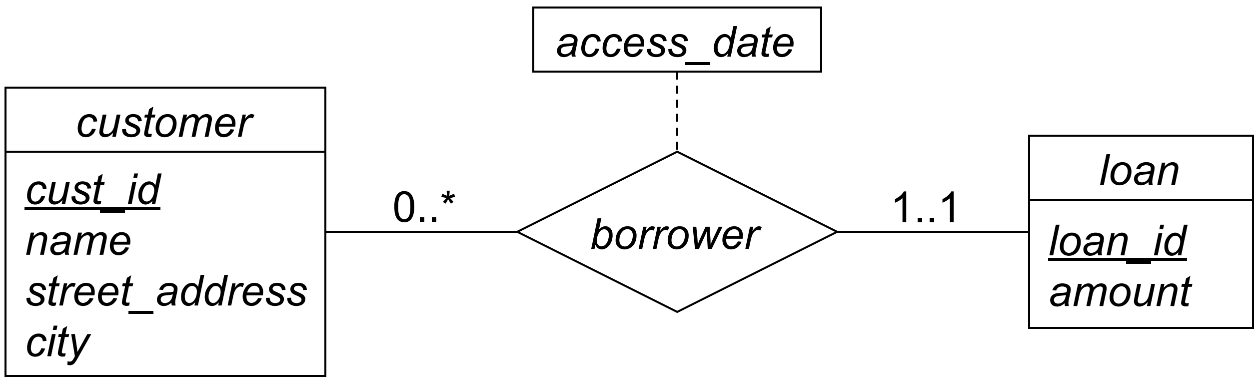
- What does this mean?



- 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* ?



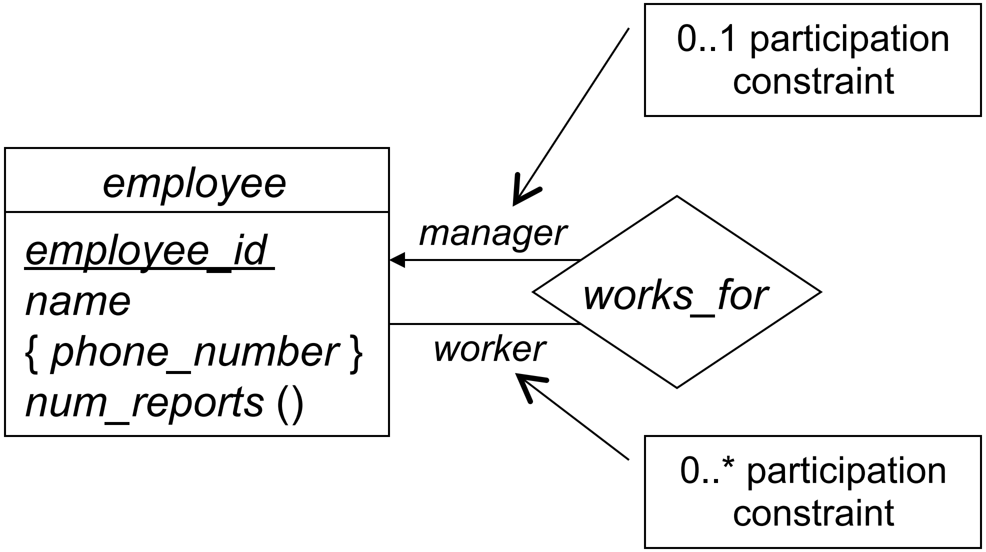
- From last slide:
  - *A customer can have zero or more loans*
  - *Each loan must be owned by exactly one customer.*
- This is a <u>one-to-many</u> mapping from *customer* to *loan*

Diagramming Roles

- Entities have <u>roles</u> 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

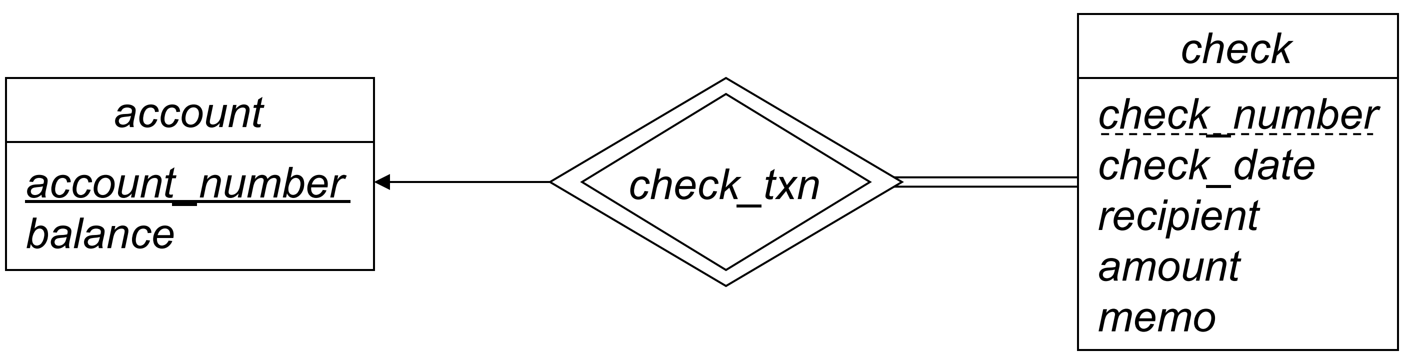
Weak Entity-Set Keys (2)

- 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:

  $primary\_key(S) \cup discriminator(W)$
- 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*)

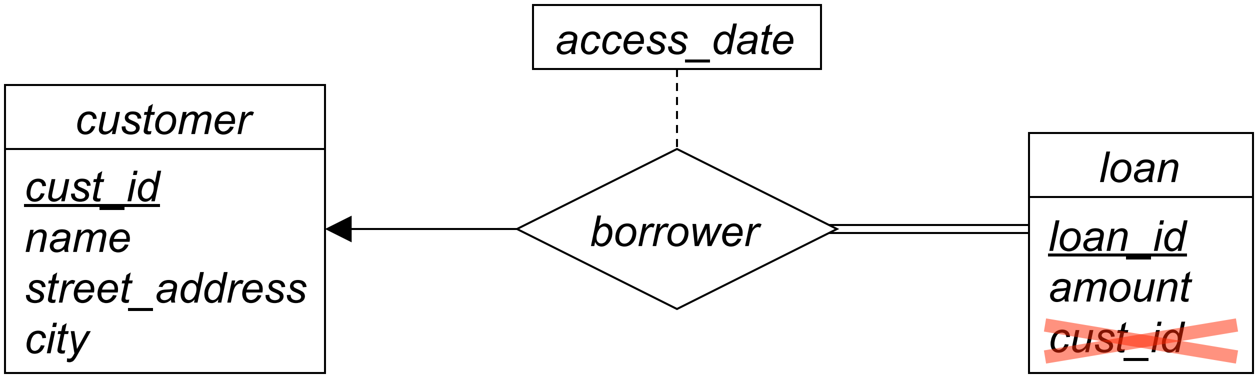
Diagramming Weak Entity-Sets

- 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

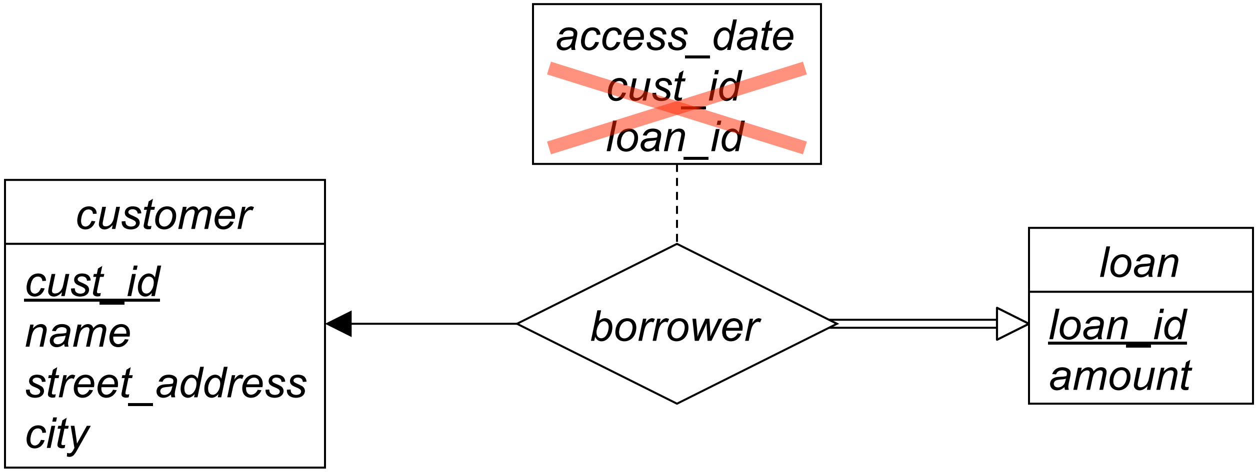
- Don't include entity-set primary key attributes on other entity-sets!
  - e.g. customers and loans, in a one-to-many mapping



- 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

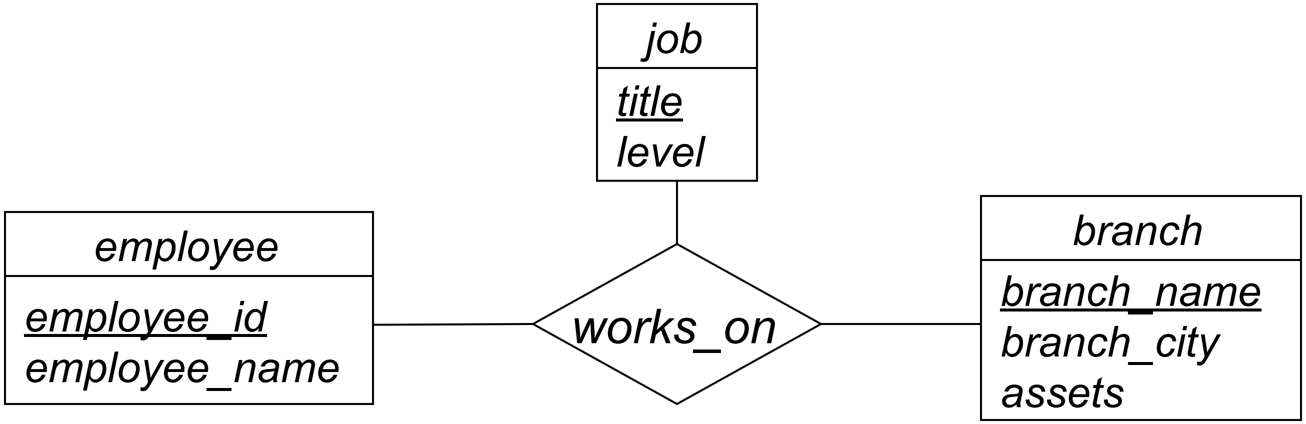
ENTITY-RELATIONSHIP MODEL III

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 16

N-ary Relationships

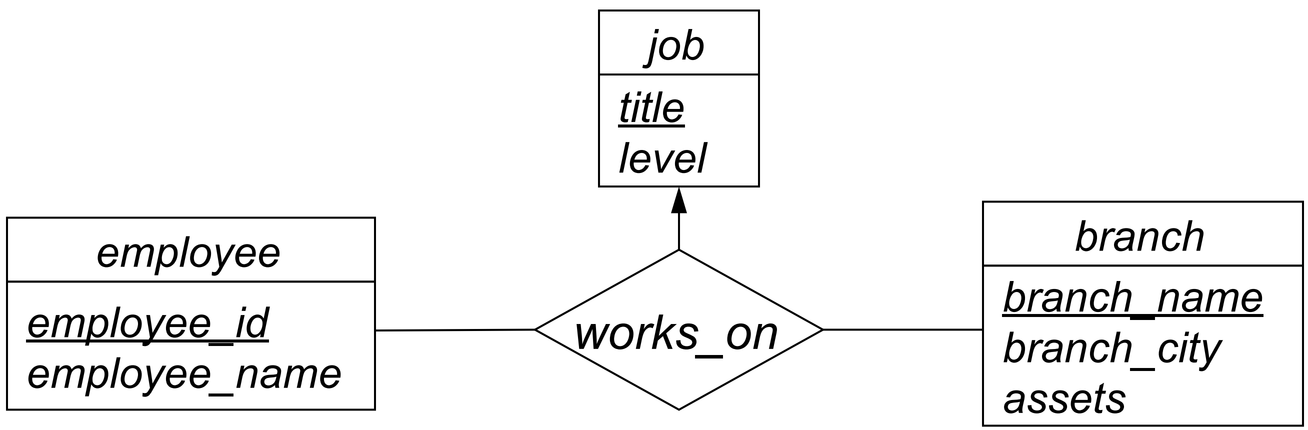
- Can specify relationships of degree > 2 in E-R model
- Example:



  - 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

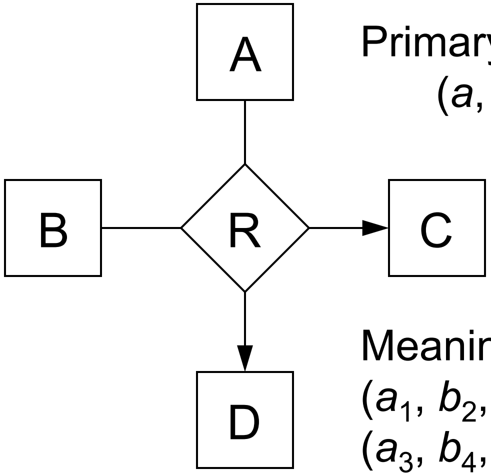
- Can specify *some* mapping cardinalities on relationships with degree > 2
- Each combination of employee and branch can only be associated with <u>one</u> job:



  - Each employee can have only one job at each branch

N-ary Mapping Cardinalities (2)

- For degree > 2 relationships, we only allow at most 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)
- 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 $\{ A_1, A_2, \ldots, A_i \}$

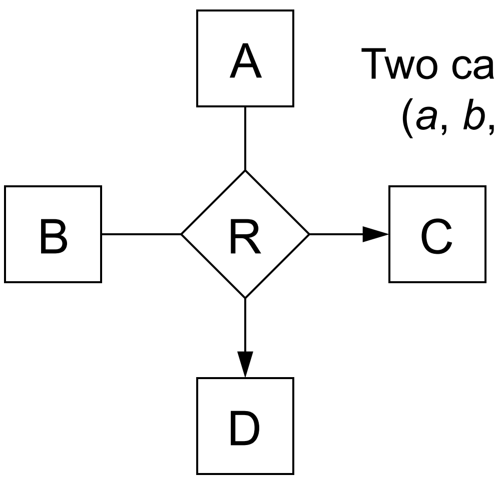


Primary key: (a, b)

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 { $A_1, A_2, \ldots, A_{k-1}, A_{k+1}, \ldots, A_n$ }



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)~~
…

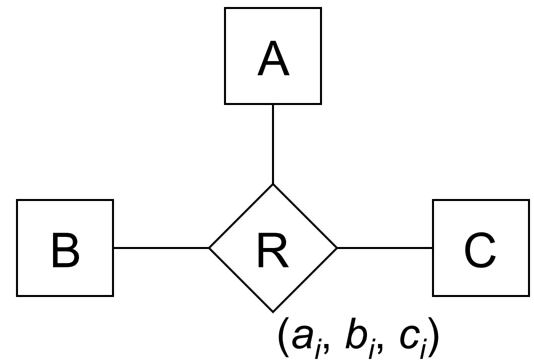
<u>All</u> disallowed by meaning 1!

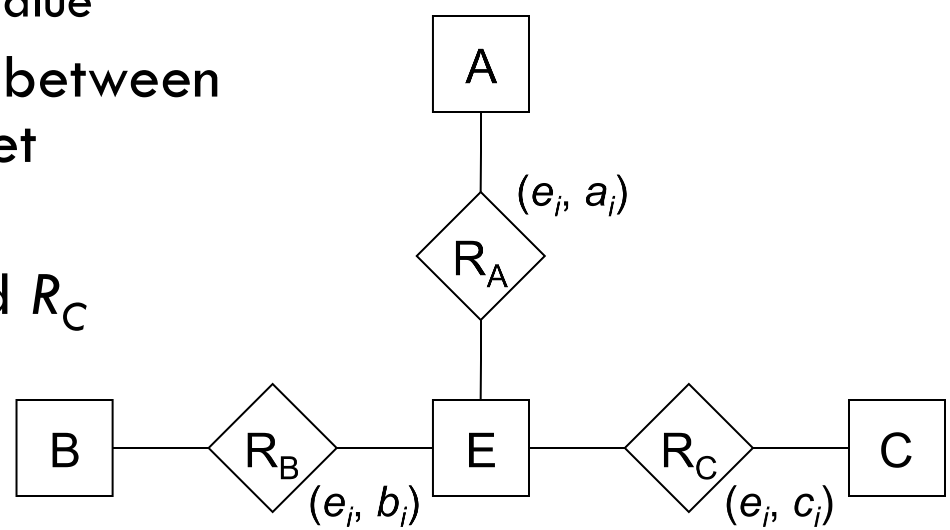
N-ary Mapping Cardinalities (4)

- 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

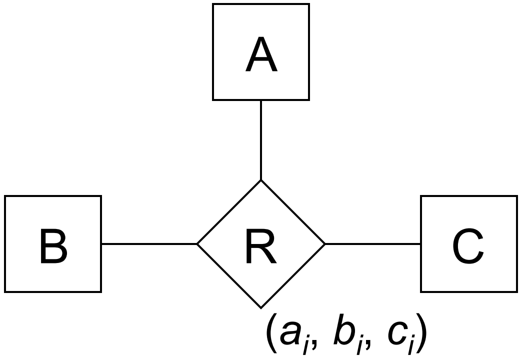
- Often have only binary relationships in DB schemas
- For degree > 2 relationships, *could* replace with binary relationships
  - Replace N-ary relationship-set with a new entity-set *E*
    - Create an identifying attribute for *E*
    - e.g. an auto-generated ID value
  - Create a relationship-set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

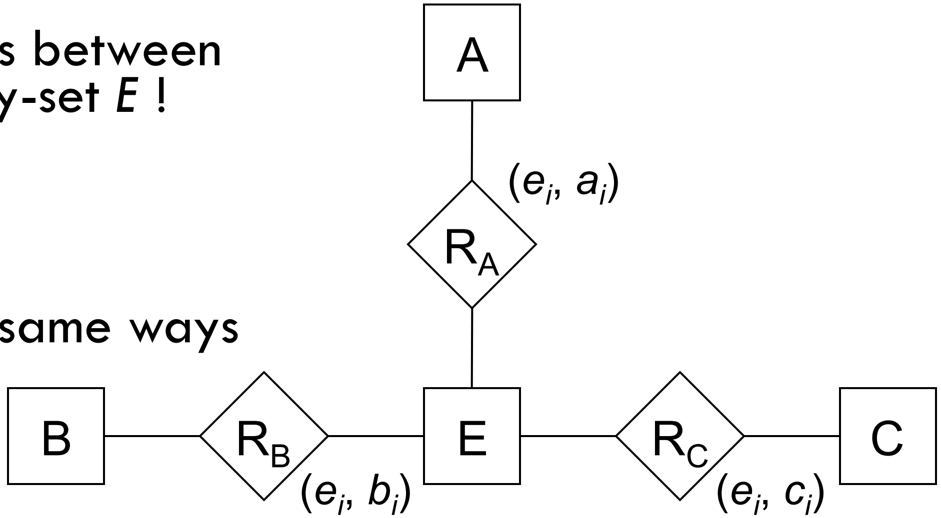




Binary vs. N-ary Relationships (2)

- **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



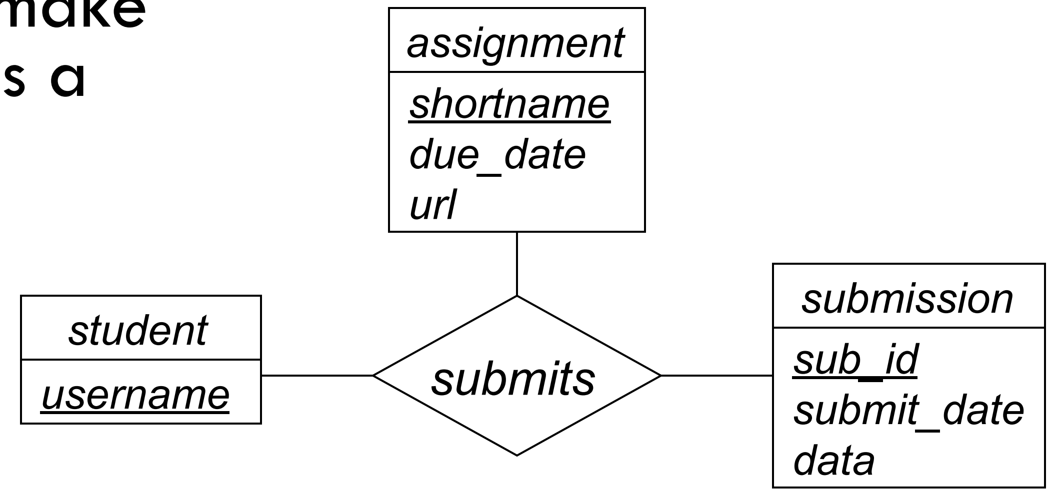


Binary vs. N-ary Relationships (3)

- 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!

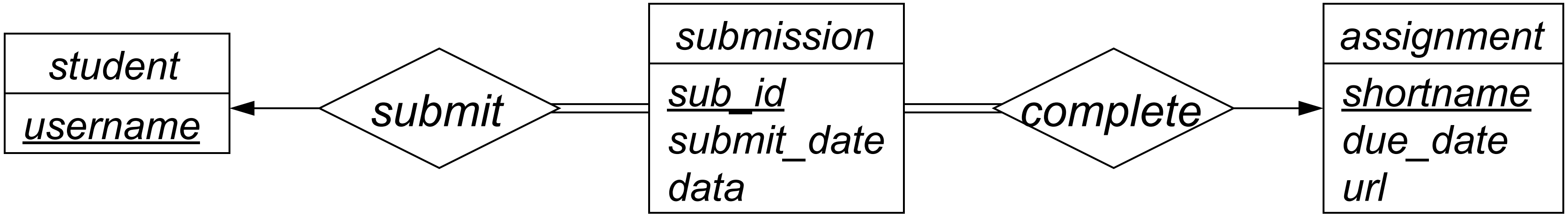
Course Database Example

- 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)

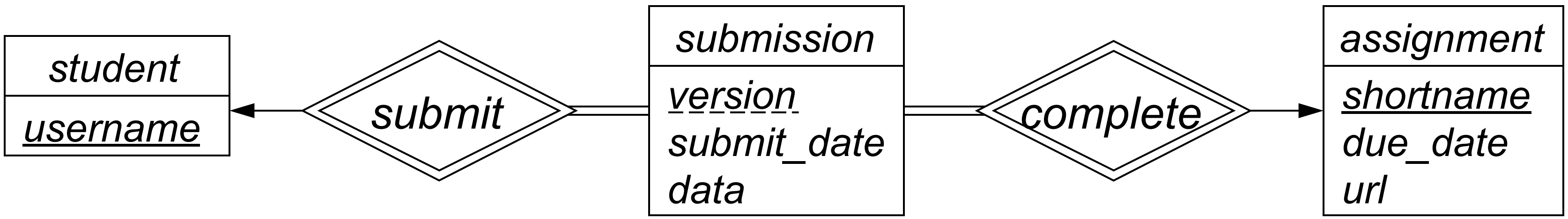
- Other ways to represent students, assignments and submissions?
- Can also represent as two binary relationships



- 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)

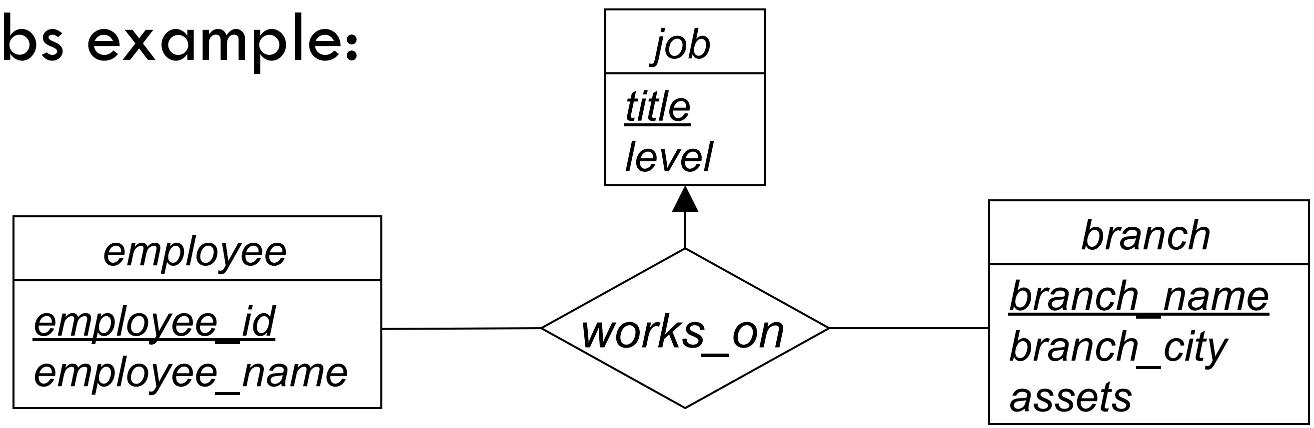
- Could even make *submission* a weak entity-set
  - Both *student* and *assignment* are identifying entities!



- 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

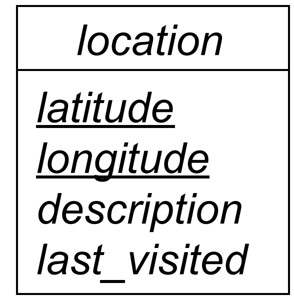
- For E-R model to be useful, need to be able to convert diagrams into an implementation schema
- Turns out to be very easy to do this!
  - Big overlaps between E-R model and relational model
  - Biggest difference is E-R composite/multivalued attributes, vs. relational model atomic attributes
- Three 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*

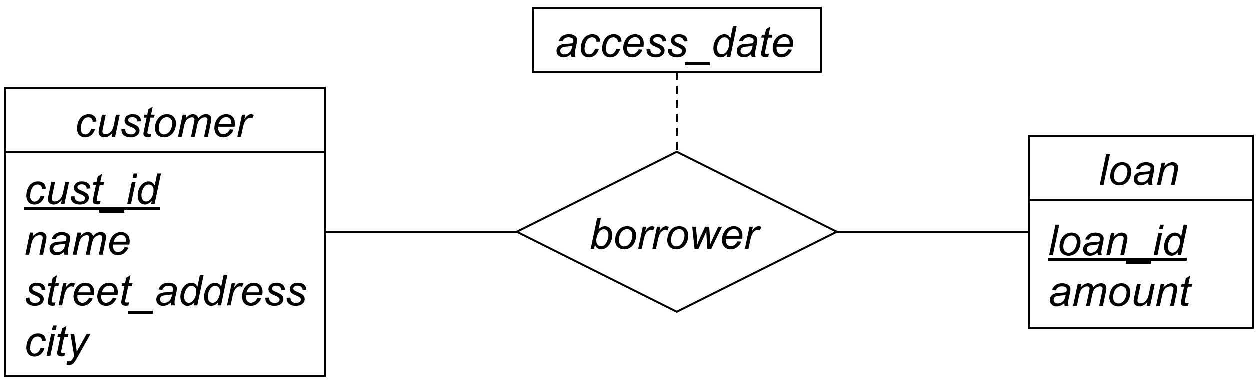


- Convert to relation schema:

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

Entity-Set Examples (2)

- E-R diagram for customers and loans:



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

  customer(<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!

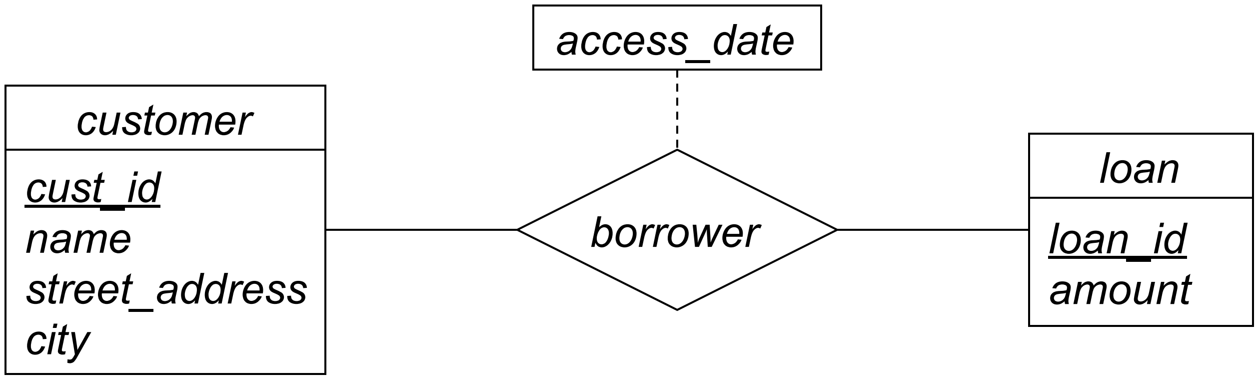
Relationship-Sets: Primary Keys (2)

- 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

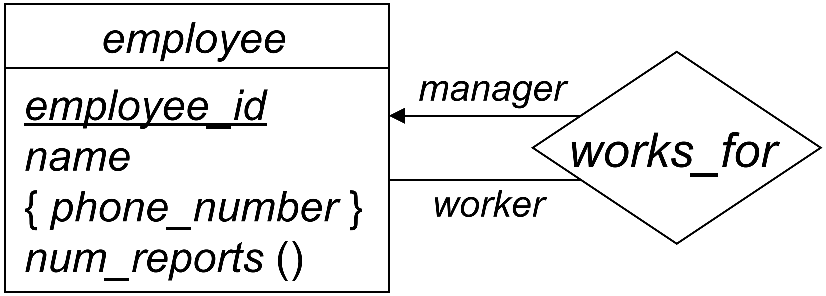
- Relationship-sets associate entities in entity-sets
  - We need foreign-key constraints on relation schema for *R* !
- For each entity-set E_i participating in *R* :
  - Relation schema for *R* has a foreign-key constraint on E_i relation, for *primary_key*(E_i) attributes
- Relation schema notation 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



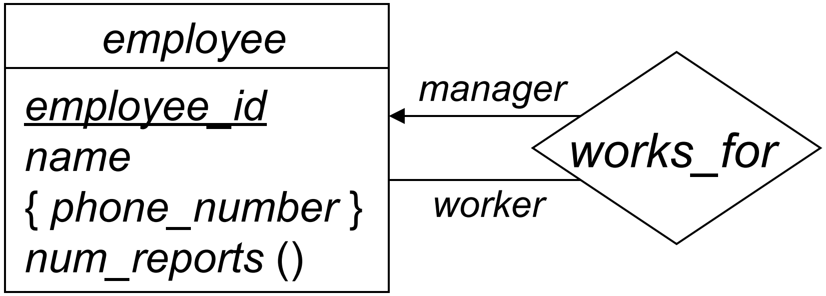
- Relation schema for *borrower*:
  - Primary key of *customer* is *cust_id*
  - Primary key of *loan* is *loan_id*
  - Descriptive attribute *access_date*
  - *borrower* mapping cardinality is many-to-many
  - Result: *borrower*(*cust_id*, *loan_id*, *access_date*)

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)

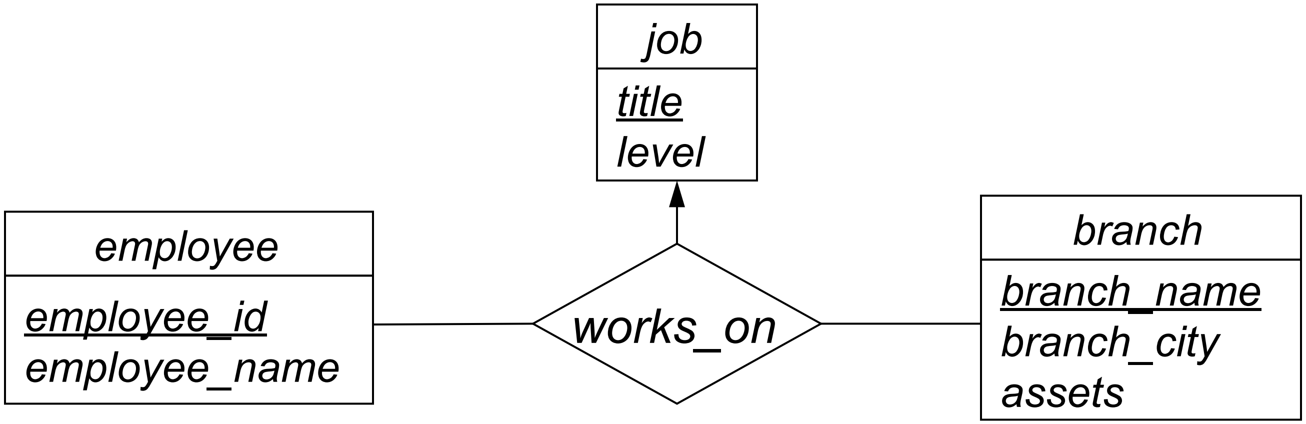


- Relation schema for *employee* entity-set:
  - (For now, ignore *phone_number* and *num_reports*…)
    employee(*employee_id*, *name*)
- Relation schema for *works_for*:
  - One-to-many mapping from *manager* to *worker*
  - "Many" side is used for primary key
  - Result: *works_for*(*employee_id*, *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 <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*)

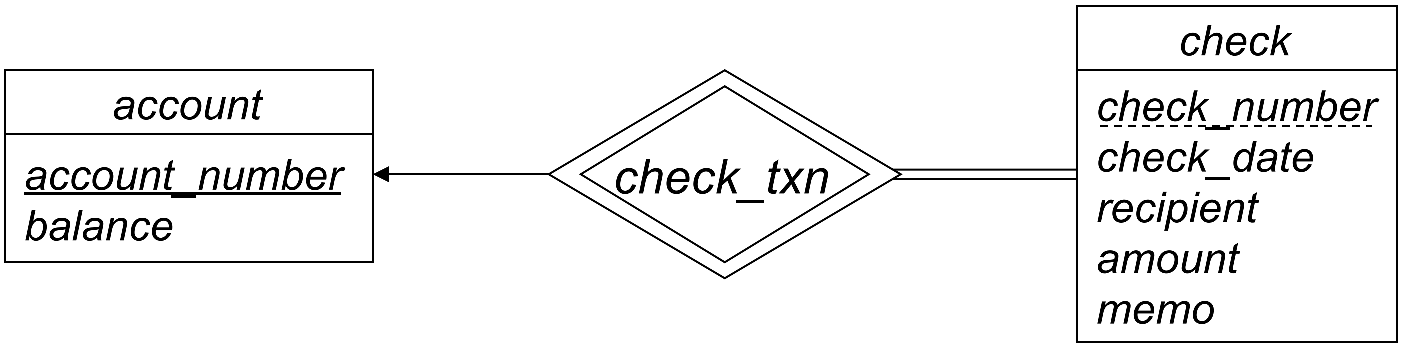
Weak Entity-Sets

- 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



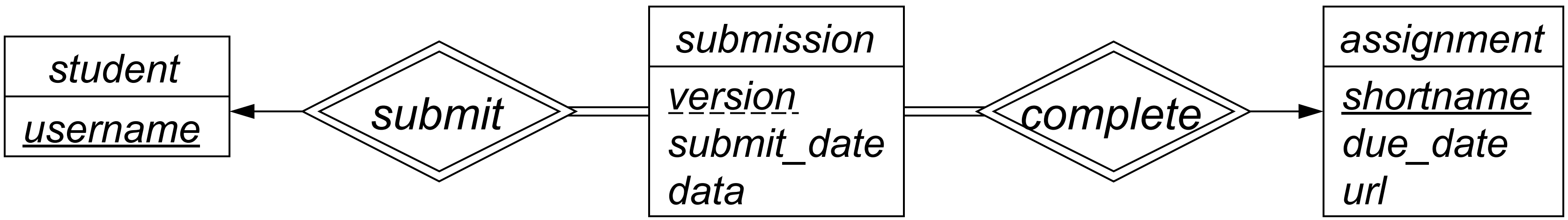
- *account* schema:

  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(<u>*username*</u>)
  assignment(<u>*shortname*</u>, *due_date*, *url*)
- Schema for *submission* weak entity-set:
  - Discriminator is *version*
  - Both *student* and *assignment* are owners!

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

Generalization and Specialization

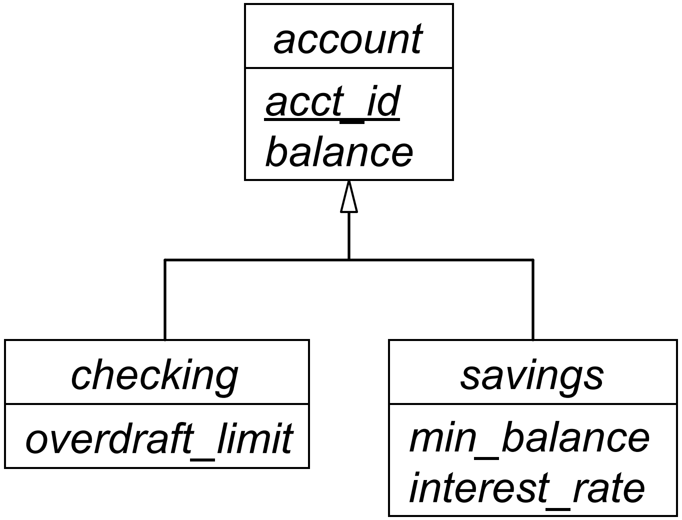
- Generalization: a “bottom up” approach
  - Taking similar entity-sets and unifying their common features
  - Start with specific entities, then create generalizations from them
- Specialization: a “top down” approach
  - Creating general purpose entity-sets, then providing specializations of the general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

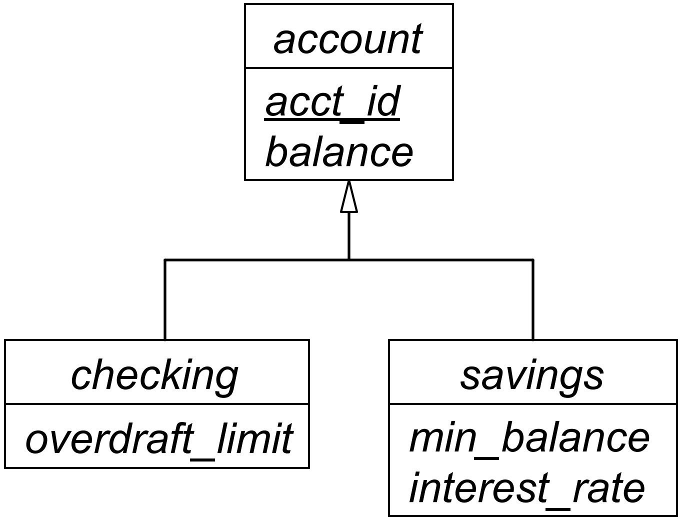
- Create entity-set to represent common attributes
  - Called the superclass, or higher-level entity-set
- Create entity-sets to represent specializations
  - Called subclasses, or lower-level entity-sets
- Join superclass to subclasses with hollow-head arrow(s)



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 <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

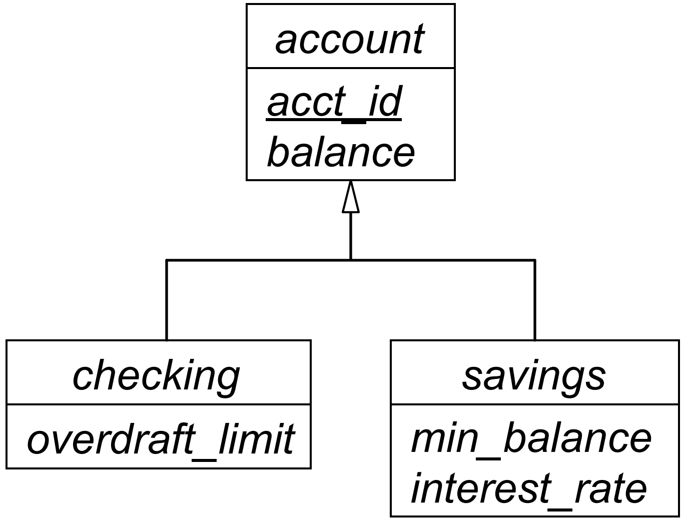
Disjointness Constraints

- "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)

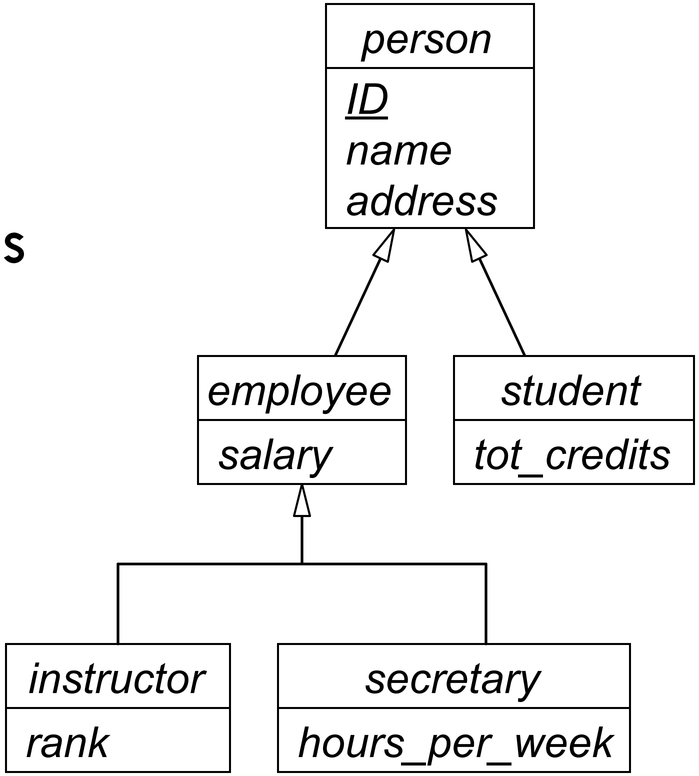
- How the arrows are drawn indicates whether the specialization is disjoint or overlapping

- 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

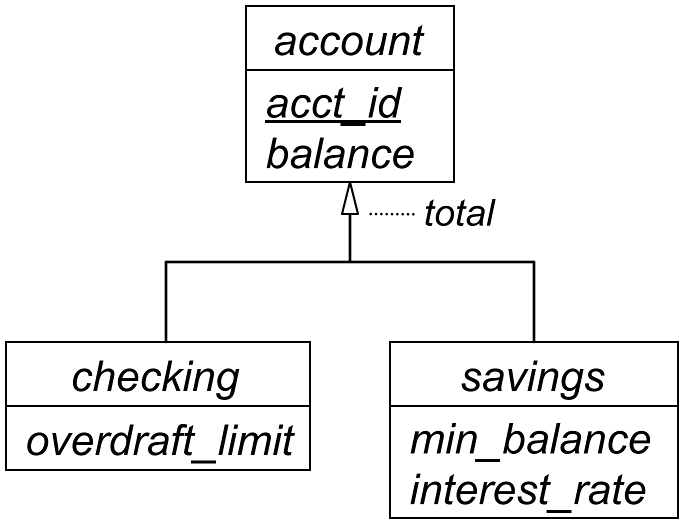


Completeness Constraints

- "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

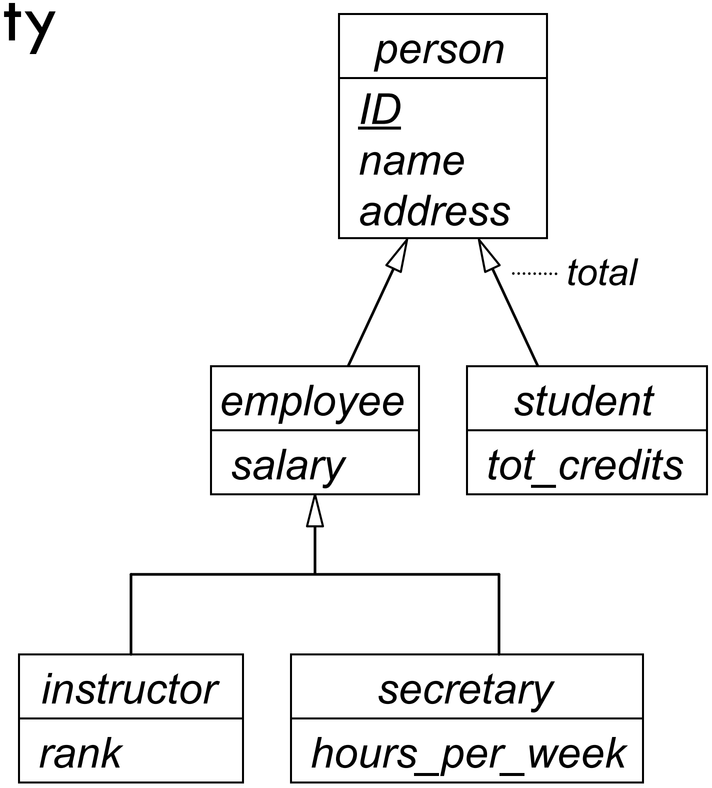
Completeness Constraints (2)

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



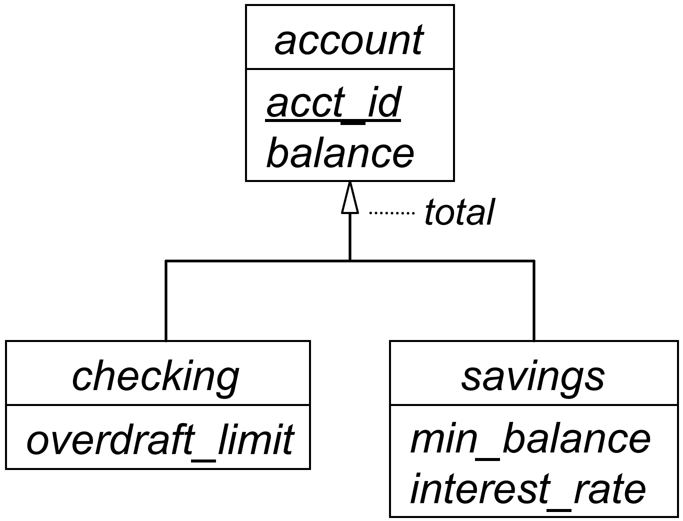
Completeness Constraints (3)

- 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

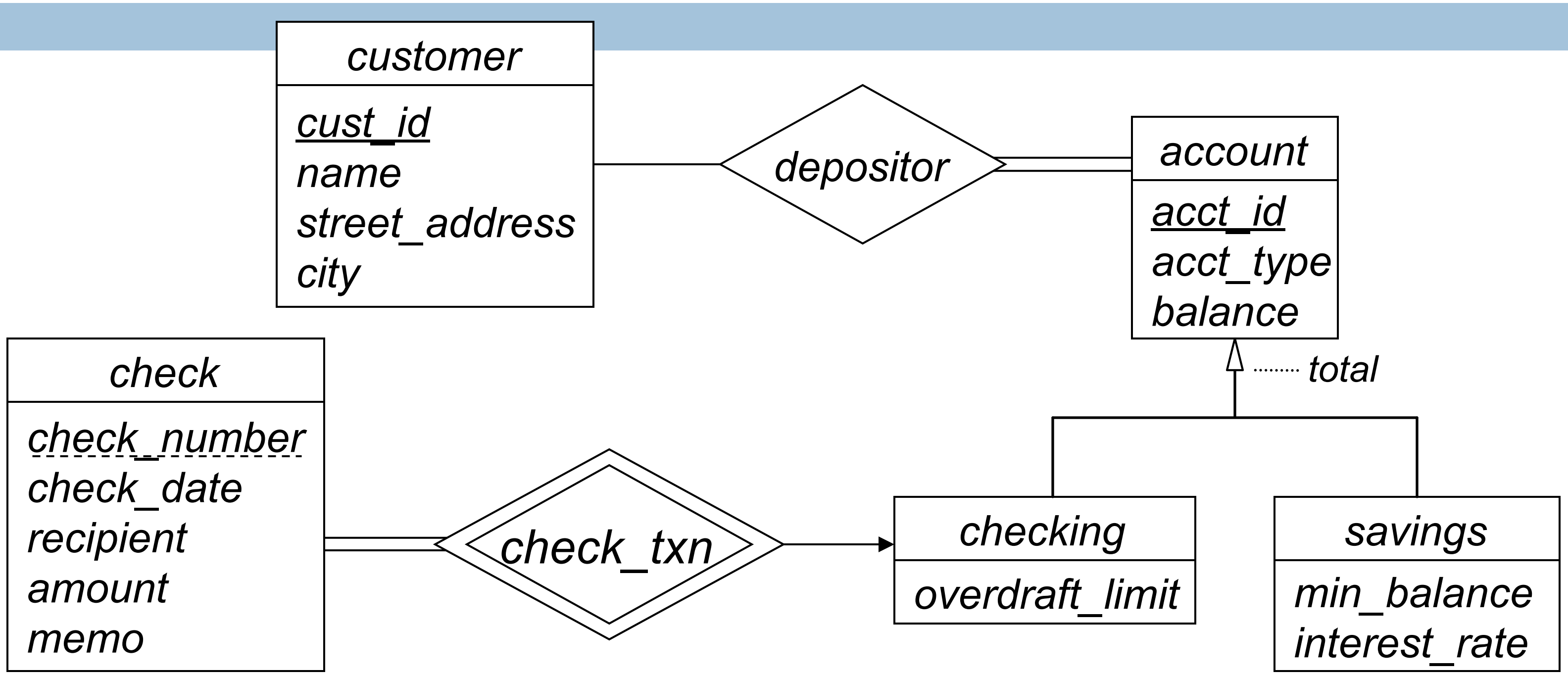
Membership Constraints

- 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

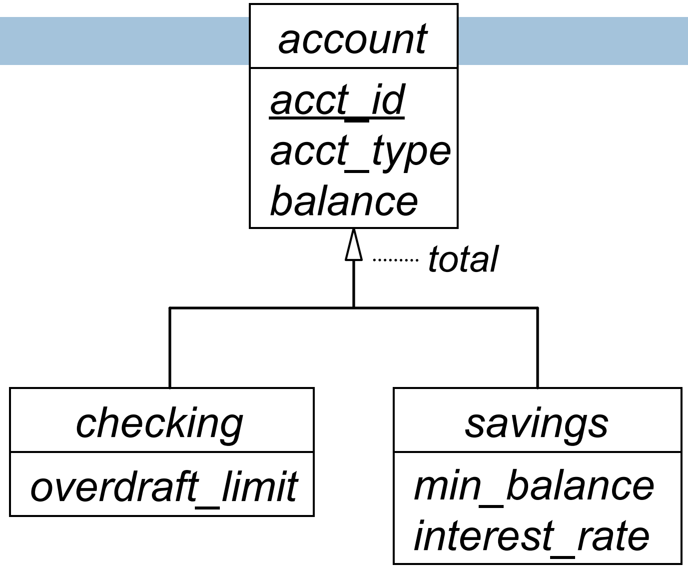


- Would also create relationship-sets against various entity-sets in hierarchy
  - associate *customer* with *account*
  - associate *check* weak entity-set with *checking*

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*)

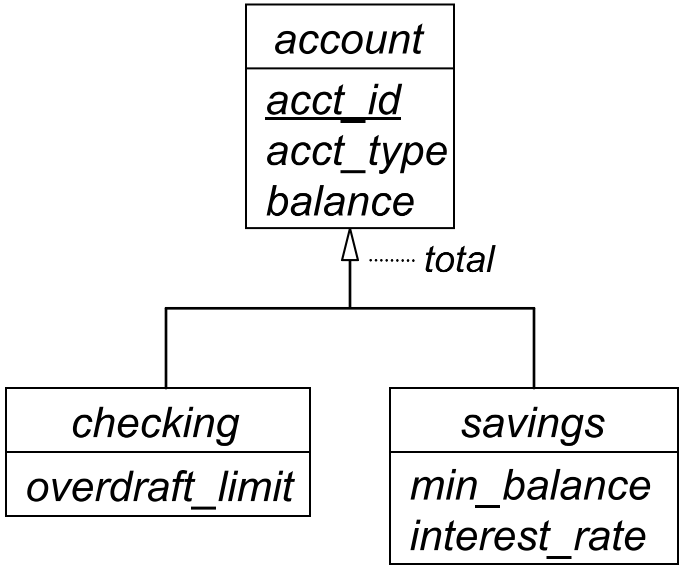
  savings(<u>*acct_id*</u>, *min_balance*, *interest_rate*)

  - 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

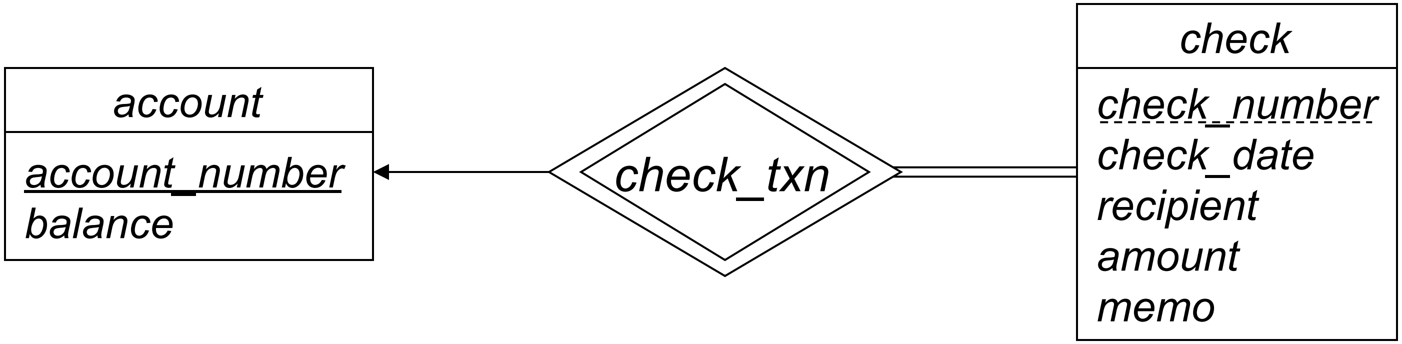
- *Generating* primary key values is actually the easy part
- Most databases provide <u>sequences</u>
  - A source of unique, increasing **INTEGER** or **BIGINT** values
  - Perfect for primary key values
  - Multiple tables can use the same sequence for their primary keys
- PostgreSQL example:

```
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:

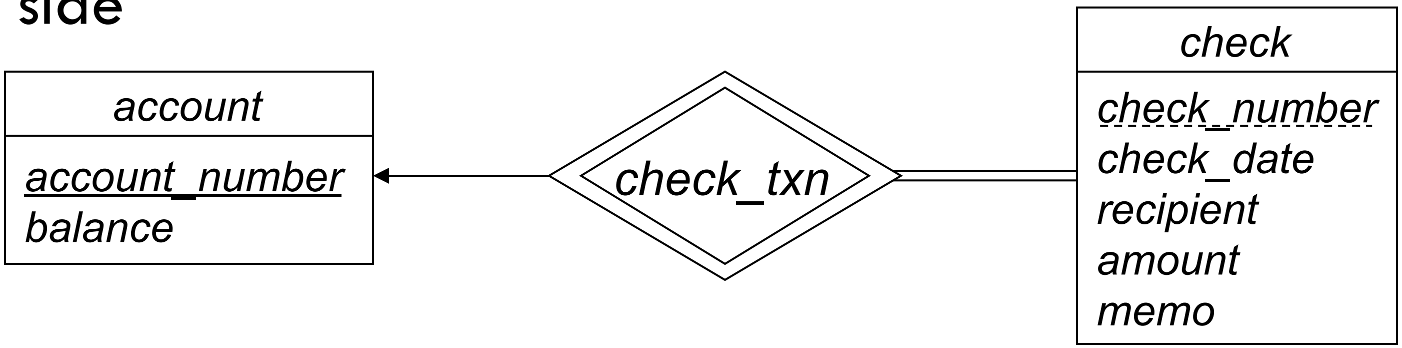
  account(<u>*account_number*</u>, *balance*)

- *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*)

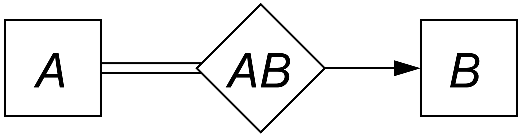
Schema Combination

- 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

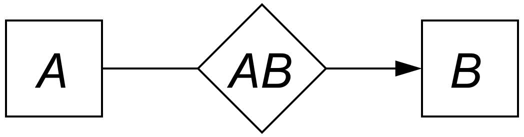


- Generates relation schemas *A*, *B*, *AB*
  - Primary key of *A* is *primary_key*(*A*)
  - Primary key of *AB* is also *primary_key*(*A*)
    - (*A* is on "many" side of mapping)
  - *AB* has foreign key constraints on both *A* and *B*
  - There is one relationship in *AB* for every entity in *A*
- 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)

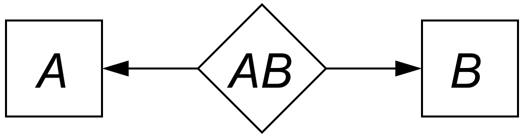
- In this case, when relationship-set is combined into the entity-set, the entity-set's primary key *doesn't change!*

- If *A*'s participation in *AB* is partial, can still combine schemas



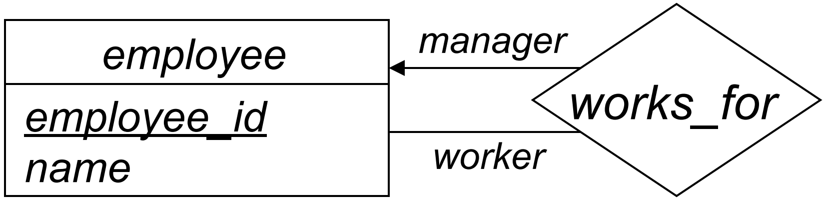
  - Must store *null* values for *primary_key*(*B*) attributes when an entity in *A* maps to no entity in *B*

- 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



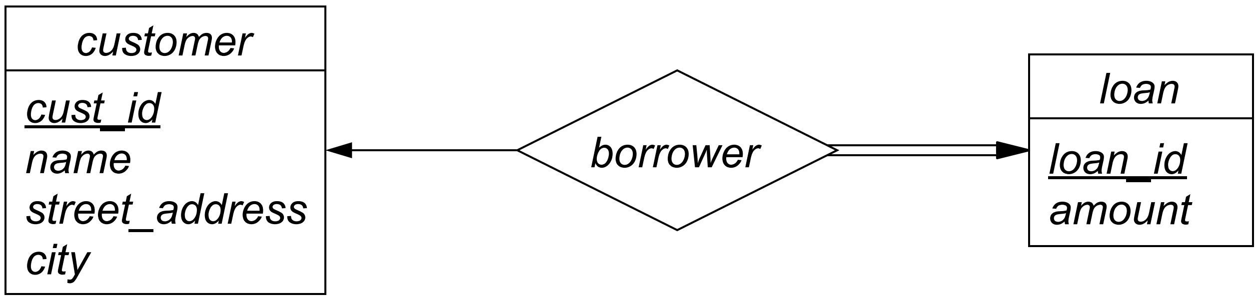
- Manager to worker mapping is one-to-many
- Relation schemas were:

  employee(<u>*employee_id*</u>, *name*)

  works_for(<u>*employee_id*</u>, *manager_id*)
- Could combine into:

  employee(<u>*employee_id*</u>, *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

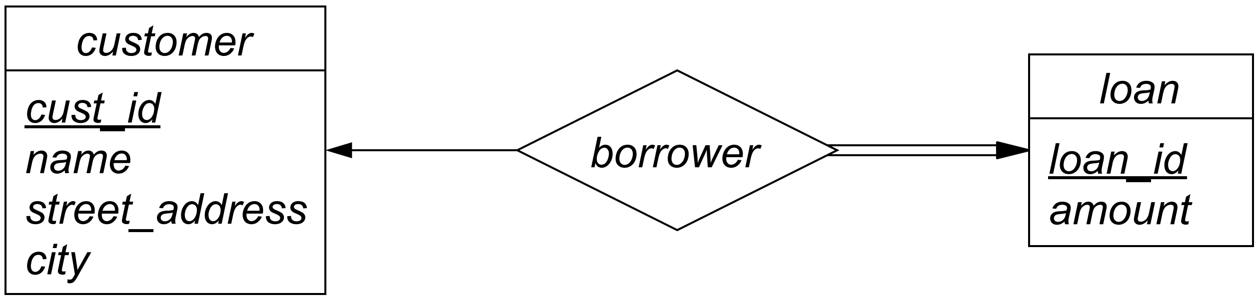
  customer(<u>*cust_id*</u>, *name*, *street_address*, *city*)

  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)

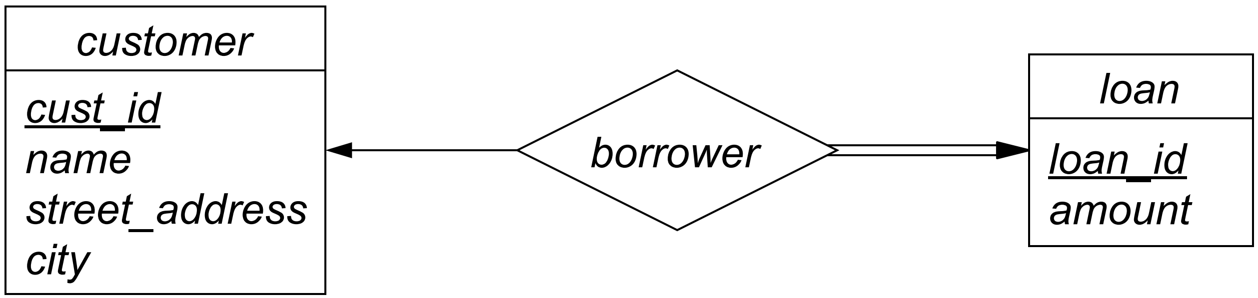


- Participation of *loan* in *borrower* will be total
  - Combining *borrower* into *customer* would require *null* values for customers without loans
- Better to combine *borrower* into *loan* schema

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

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

NORMAL FORMS

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 18

Equivalent Schemas

- Many different schemas can represent a set of data
  - Which one is best?
  - What does "best" even mean?
- Main goals:
  - Representation must be complete
  - Data should not be unnecessarily redundant
  - Should be easy to manipulate the information
  - Should be easy to enforce [most] constraints

Normal Forms

- A "good" pattern for database schemas to follow is called a normal form
- Several different normal forms, with different constraints
- Normal forms can be formally specified
  - Can test a schema against a normal form
  - Can transform a schema into a normal form
- Goal:
  - Design schemas that satisfy a particular normal form
  - If a schema isn't "good," transform it into an appropriate normal form

Example Schema Design

- Schema for representing loans and borrowers:
  - *customer* relation stores customer details, including a *cust_id* primary-key attribute
  - *loan*(*loan_id*, *amount*)
  - *borrower*(*cust_id*, *loan_id*)
- Many-to-many mapping
  - A customer can have multiple loans
  - A loan can be owned by multiple customers

loan_id	amount
…	…
L-100	10000
…	…

loan

cust_id	loan_id
…	…
23-652	L-100
15-202	L-100
23-521	L-100
…	…

borrower

Larger Schema?

- Could replace *loan* and *borrower* relations with a larger, combined relation

  bor_loan(*cust_id*, *loan_id*, *amount*)

cust_id	loan_id	amount
…	…	…
23-652	L-100	10000
15-202	L-100	10000
23-521	L-100	10000
…	…	…

bor_loan

- Rationale:
  - Eliminates a join when retrieving loan amounts
- Problem: mapping between customers and loans is many-to-many
  - Multiple redundant copies of *amount* to keep in sync!

Repeated Values

- How do we *know* that this is a problem?
  - "Because we see values that appear multiple times"
  - *This isn't a good enough reason!!!*
  - Could easily have different loans with the same amount
- A repeated value doesn't *automatically* indicate a problem…

cust_id	loan_id	amount
…	…	…
23-652	L-100	10000
19-065	L-205	10000
15-202	L-100	10000
23-521	L-100	10000
20-419	L-205	10000
…	…	…

bor_loan

Back to the Enterprise

- What are the rules of the enterprise that we are modeling?
  - "Every loan must have <u>only one</u> amount."
- In other words:
  - Every loan ID corresponds to exactly one amount.
  - If there were a schema (*loan_id*, *amount*) then *loan_id* can be a primary key.
- Specified as a <u>functional dependency</u>
  - $loan\_id \rightarrow amount$
  - *loan_id* functionally determines *amount*

Repeated Values v2.0

- *bor_loan* relation has both *loan_id* and *amount* attributes

  bor_loan(<u>*cust_id*</u>, <u>*loan_id*</u>, *amount*)

- But, *loan_id* → *amount*, and *loan_id* by itself <u>can't</u> be a primary key in *bor_loan*
  - Need to support many-to-many mappings between customers and loans
  - Combination of *cust_id* and *loan_id* must be a primary key, so a particular *loan_id* value can appear multiple times
- In rows with the same *loan_id* value, *amount* will have to be repeated.

Functional Dependencies

- Functional dependencies are *very* important in schema analysis
  - Have a *lot* to do with keys!
  - "Good" schema designs are guided by functional dependencies
  - Frequently helpful to identify them during schema design
- Can formally define functional dependencies, and reason about them
- Can also specify constraints on schemas using functional dependencies

Another Example Schema

- A "large" schema for employee information

  employee(<u>emp_id</u>, emp_name, phone, title, salary, start_date)

emp_id	emp_name	phone	title	salary	start_date
…	…	…	…	…	…
123-45-6789	Jeff	555-1234	CTO	120000	1996-03-15
314-15-9265	Mary	555-3141	CFO	120000	1997-08-02
987-65-4321	Helen	555-9876	Developer	90000	1996-05-23
101-01-0101	Marcus	555-1010	Tester	70000	1995-11-04
…	…	…	…		…

employee

- Employee ID is unique, but other attributes could have duplicate values

Smaller Schemas?

- Could represent this with two smaller schemas:
  emp_ids(*emp_id*, *emp_name*)
  emp_details(*emp_name*, *phone*, *title*, *salary*, *start_date*)

emp_id	emp_name
…	…
123-45-6789	Jeff
314-15-9265	Mary
987-65-4321	Helen
101-01-0101	Marcus
…	…

emp_ids

emp_name	phone	title	salary	start_date
…	…	…	…	…
Jeff	555-1234	CTO	120000	1996-03-15
Mary	555-3141	CFO	120000	1997-08-02
Helen	555-9876	Developer	90000	1996-05-23
Marcus	555-1010	Tester	70000	1995-11-04
…	…	…		…

emp_details

- Generate original employee data with a join:
  emp_ids ⋈ *emp_details*
- Any problems with this?

emp_name is not unique!

- Joins using *emp_name* can generate invalid tuples!

emp_id	emp_name
…	…
314-15-9265	Mary
161-80-3398	Mary
…	…

emp_ids

emp_name	phone	title	salary	start_date
…	…	…	…	…
Mary	555-3141	CFO	120000	1997-08-02
Mary	555-1618	Gofer	25000	1998-01-07
…	…	…		…

emp_details

emp_ids ⋈ *emp_details*

emp_id	emp_name	phone	title	salary	start_date
…	…	…	…	…	…
314-15-9265	Mary	555-1618	Gofer	25000	1998-01-07
314-15-9265	Mary	555-3141	CFO	120000	1997-08-02
161-80-3398	Mary	555-3141	CFO	120000	1997-08-02
161-80-3398	Mary	555-1618	Gofer	25000	1998-01-07
…	…	…	…		…

Bad Decompositions

- This decomposition is clearly broken
  - It can't represent the information correctly!
- Problem: enterprise needs to support different employees with the same name
- Lossy decompositions cannot accurately represent all facts about an enterprise
- Lossless decompositions can accurately represent all facts
- "Good" schema designs will avoid lossy decompositions

First Normal Form

- A schema is in first normal form (1NF) if all attribute domains are atomic
  - An atomic domain has values that are indivisible units
- E-R model supports non-atomic attributes
  - Multivalued attributes
  - Composite attributes
- Relational model specifies atomic domains for attributes
  - Schemas are automatically in 1NF
  - Mapping from E-R model to relational model changes composite/multivalued attributes into an atomic form

1NF Example

- E-R diagram for magazine subscribers
  - *address* is composite
  - *email_addr* is multivalued

subscriber

sub_id
{ *email_addr* }
address
    street
    city
    state
    zip_code

- Converts to a 1NF schema:

subscriber(*sub_id*, *street*, *city*, *state*, *zip_code*)

sub_emails(*sub_id*, *email_addr*)

  - The conversion rules we have discussed, *automatically* convert E-R schemas into 1NF

1NF and Non-Atomic Attributes

- Many, but not all, SQL DBs have non-atomic types
  - Some offer support for composite attributes
  - Some offer support for multivalued attributes
  - These are SQL extensions – not portable
- As long as you steer clear of using non-atomic attributes in primary/foreign keys, can sometimes be quite useful
  - Will likely encounter them very rarely in practice, though
  - Biggest reason: DB support for list/vector column-types isn't terribly widespread, or always very easy to use

1NF and Non-Atomic Attributes (2)

- Composite types:
  - e.g. defining an "address" composite type
  - Can definitely be useful for making a schema clearer, as long as they aren't used in a key!
- Multivalued types:
  - e.g. arrays, lists, sets, vectors
  - Can sometimes be useful for storing pre-computed values that aren't expected to change frequently
  - If you are regularly issuing queries that search through or change these values, you may need to revise your schema!
    - Should probably factor non-atomic data out into a separate table

Other Normal Forms

- Other normal forms relate to functional dependencies
- Analysis of functional dependencies shows if a schema needs decomposed
- Keys are functional dependencies too!
- Formally define functional dependencies, and reason about them
- Define normal forms in terms of functional dependencies

Schemas and Constraints

- Keys and functional dependencies are <u>constraints</u> that a database must satisfy
  - <u>Legal</u> relations satisfy the required constraints
  - Relation doesn't contain any tuples that violate the specified constraints
- More terminology:
  - Relation schema *R*, relation *r*(*R*)
  - A set of functional dependencies *F*
  - Relation *r* <u>satisfies</u> *F* if *r* is legal
  - When we say "*F* <u>holds</u> on *R*", specifies the set of relations with *R* as their schema, that are legal with respect to *F*

Functional Dependencies

- Formal definition of a functional dependency:
  - Given a relation schema R with attribute-sets α, $\beta \subseteq R$
  - The functional dependency $\alpha \rightarrow \beta$ holds on $r(R)$ if
    $\langle\ \forall\ t_1, t_2 \in r : t_1[\alpha] = t_2[\alpha] : t_1[\beta] = t_2[\beta]\ \rangle$
- In other words:
  - For all pairs of tuples t_1 and t_2 in r,
    if $t_1[\alpha] = t_2[\alpha]$ then $t_1[\beta] = t_2[\beta]$
  - α functionally determines β

Dependencies and Superkeys

- Given relation schema R, a subset K of R can be a superkey
  - In a relation $r(R)$, no two tuples can share the same values for attributes in K
- Can also say: K is a superkey if $K \rightarrow R$
  - The functional dependency $K \rightarrow R$ holds if
    $\langle\ \forall\ t_1, t_2 \in r(R) : t_1[K] = t_2[K] : t_1[R] = t_2[R]\ \rangle$
  - $t_1[R] = t_2[R]$ (or $t_1 = t_2$) means t_1 and t_2 are the same tuple
  - The superkey K functionally determines the whole relation R

- Functional dependencies are a more general form of constraint than superkeys are.

The *bor_loan* Relation

- *bor_loan*(*cust_id*, *loan_id*, *amount*)
  - Functional dependency: *loan_id* → *amount*
  - "Every loan has exactly one amount."
  - Every tuple in *bor_loan* with a given *loan_id* value must have the same *amount* value
- *bor_loan* also has a primary key
  - Specifies another functional dependency
  - *cust_id*, *loan_id* → *cust_id*, *loan_id*, *amount*
  - This is not a functional dependency *specifically required* by what the enterprise needs to model
    - Can be inferred from other functional dependencies in the schema

Trivial Dependencies

- A <u>trivial</u> functional dependency is satisfied by *all* relations!
  - For a relation R containing attributes A and B,

    $A \rightarrow A$ is a trivial dependency

    $\langle\ \forall\ t_1, t_2 \in r : t_1[A] = t_2[A] : t_1[A] = t_2[A]\ \rangle$
    - Well, duh!
  - $AB \rightarrow A$ is also a trivial dependency
    - If $t_1[AB] = t_2[AB]$, then of course $t_1[A] = t_2[A]$ too!
- In general: $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Closure

- Given a set of functional dependencies, we can infer other dependencies
  - Given relation schema $R(A, B, C)$
  - If $A \rightarrow B$ and $B \rightarrow C$, holds on R, then $A \rightarrow C$ also holds on R
- Given a set of functional dependencies F
  - F^+ denotes the closure of F
  - F^+ includes F, and all dependencies that can be inferred from F. ($F \subseteq F^+$)

Boyce-Codd Normal Form

- Eliminates all redundancy that can be discovered using functional dependencies
- Given:
  - Relation schema R
  - Set of functional dependencies F
- R is in BCNF with respect to F if:
  - For all functional dependencies $\alpha \rightarrow \beta$ in F^+, where $\alpha \in R$ and $\beta \in R$, at least one of the following holds:
    - $\alpha \rightarrow \beta$ is a trivial dependency
    - α is a superkey for R
- A database design is in BCNF if all schemas in the design are in BCNF

BCNF Examples

- The *bor_loan* schema isn't in BCNF

  bor_loan(<u>*cust_id*</u>, <u>*loan_id*</u>, *amount*)
  - *loan_id* → *amount* holds on *bor_loan*
  - This is not a trivial dependency, and *loan_id* isn't a superkey for *bor_loan*
- The *borrower* and *loan* schemas <u>are</u> in BCNF

  borrower(<u>*cust_id*</u>, <u>*loan_id*</u>)
  - No nontrivial dependencies hold

  loan(<u>*loan_id*</u>, *amount*)
  - *loan_id* → *amount* holds on *loan*
  - *loan_id* is the primary key of *loan*

BCNF Decomposition

- If *R* is a schema not in BCNF:
  - There is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ such that α is not a superkey for *R*
- Replace *R* with two schemas:

  $(\alpha \cup \beta)$

  $(R - (\beta - \alpha))$
    - (stated this way in case α and β overlap; usually they don't)
- *The new schemas might also not be in BCNF!*
  - Repeat this decomposition process until all schemas are in BCNF

Undoing the Damage

- For *bor_loan*, α = *loan_id*, β = *amount*

  $R = (cust\_id, loan\_id, amount)$

  $(\alpha \cup \beta) = (loan\_id, amount)$

  $(R - (\beta - \alpha)) = (cust\_id, loan\_id)$
- Rules successfully decompose *bor_loan* back into *loan* and *borrower* schemas

Review

- Normal forms are guidelines for what makes a database design “good”
  - Can formally specify them
  - Can transform schemas into normal forms
- Functional dependencies specify constraints between attributes in a schema
  - A more general kind of constraint than key constraints
- Covered 1NF and BCNF
  - 1NF requires all attributes to be atomic
  - BCNF uses functional dependencies to eliminate redundant data

Next Time!

- A big question to explore:
  - Given a set of functional dependencies F, we need to know what dependencies can be inferred from it!
    - i.e. given F, how to compute F^+
  - BCNF needs this information, as do other normal forms
- Does Boyce-Codd Normal Form have drawbacks?
  - (yes.)
  - Motivates the development of 3rd Normal Form

FUNCTIONAL DEPENDENCY THEORY

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 19

Last Lecture

- Normal forms specify "good schema" patterns
- First normal form (1NF):
  - All attributes must be atomic
  - Easy in relational model, harder/less desirable in SQL
- Boyce-Codd normal form (BCNF):
  - Eliminates redundancy using functional dependencies
  - Given a relation *R* and a set of dependencies *F*
  - For all functional dependencies $\alpha \rightarrow \beta$ in F^+, where $\alpha \cup \beta \subseteq R$, at least one of these conditions must hold:
    - $\alpha \rightarrow \beta$ is a trivial dependency
    - α is a superkey for *R*

Last Lecture (2)

- Can convert a schema into BCNF
- If *R* is a schema not in BCNF:
  - There is at least one nontrivial functional dependency $\alpha \rightarrow \beta \in F^+$ such that α is not a superkey for *R*
- Replace *R* with two schemas:

  $(\alpha \cup \beta)$

  $(R - (\beta - \alpha))$
- May need to repeat this decomposition process until all schemas are in BCNF

Functional Dependency Theory

- Important to be able to reason about functional dependencies!
- Main question:
  - What functional dependencies are implied by a set *F* of functional dependencies?
- Other useful questions:
  - Which attributes are functionally determined by a particular attribute-set?
  - What *minimal* set of functional dependencies must actually be enforced in a database?
  - Is a particular schema decomposition lossless?
  - Does a decomposition preserve dependencies?

Rules of Inference

- Given a set F of functional dependencies
  - Actual dependencies listed in F may be insufficient for normalizing a schema
  - Must consider all dependencies logically implied by F
- For a relation schema R
  - A functional dependency f on R is logically implied by F on R if every relation instance $r(R)$ that satisfies F also satisfies f
- Example:
  - Relation schema $R(A, B, C, G, H, I)$
  - Dependencies:
    $A \rightarrow B$, $A \rightarrow C$, $CG \rightarrow H$, $CG \rightarrow I$, $B \rightarrow H$
  - Logically implies: $A \rightarrow H$, $CG \rightarrow HI$, $AG \rightarrow I$

Rules of Inference (2)

- Axioms are rules of inference for dependencies
- This group is called Armstrong's axioms
- Greek letters α, β, γ, … represent attribute sets
- Reflexivity rule:

  If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- Augmentation rule:

  If $\alpha \rightarrow \beta$ holds, and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- Transitivity rule:

  If $\alpha \rightarrow \beta$ holds, and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Computing Closure of F

Can use Armstrong's axioms to compute F^+ from F

- F is a set of functional dependencies

$F^+ = F$

repeat

    for each functional dependency f in F^+

        apply reflexivity and augmentation rules to f

        add resulting functional dependencies to F^+

    for each pair of functional dependencies f_1 , f_2 in F^+

        if f_1 and f_2 can be combined using transitivity

            add resulting functional dependency to F^+

until F^+ stops changing

Armstrong's Axioms

- Axioms are <u>sound</u>
  - They don't generate any incorrect functional dependencies
- Axioms are <u>complete</u>
  - Given a set of functional dependencies F, repeated application generates all F^+
- F^+ could be <u>very</u> large
  - LHS and RHS of a dependency are subsets of R
  - A set of size n has 2^n subsets
  - $2^n \times 2^n = 2^{2n}$ possible functional dependencies in R !

More Rules of Inference

- Additional rules can be proven from Armstrong's axioms
  - These make it easier to generate F^+
- Union rule:

  If $\alpha \rightarrow \beta$ holds, and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- Decomposition rule:

  If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- Pseudotransitivity rule:

  If $\alpha \rightarrow \beta$ holds, and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Attribute-Set Closure

- How to tell if an attribute-set α is a superkey?
  - If $\alpha \rightarrow R$ then α is a superkey.
  - What attributes are functionally determined by an attribute-set α ?
- Given:
  - Attribute-set α
  - Set of functional dependencies F
  - The set of all attributes functionally determined by α under F is called the closure of α under F
  - Written as α^+

Attribute-Set Closure (2)

- It's easy to compute the closure of attribute-set α !
  - Algorithm is very simple
- Inputs:
  - attribute-set α
  - set of functional dependencies *F*

$\alpha^+ = \alpha$
repeat
  for each functional dependency $\beta \rightarrow \gamma$ in *F*
    if $\beta \subseteq \alpha^+$ **then** $\alpha^+ = \alpha^+ \cup \gamma$
until α^+ stops changing

Attribute-Set Closure (3)

- Can easily test if α is a superkey
  - Compute α^+
  - If $R \subseteq \alpha^+$ then α is a superkey of R
- Can also use with functional dependencies
  - $\alpha \rightarrow \beta$ holds if $\beta \subseteq \alpha^+$
    - Find closure of α under F; if it contains β then $\alpha \rightarrow \beta$ holds!
  - Can compute F^+ with attribute-set closure too:
    - For each $\gamma \subseteq R$, find closure γ^+ under F
      - We know that $\gamma \rightarrow \gamma^+$
    - For each subset $S \subseteq \gamma^+$, add functional dependency $\gamma \rightarrow S$

Attribute-Set Closure Example

- Relation schema $R(A, B, C, G, H, I)$
  - Dependencies:
    $A \rightarrow B$, $A \rightarrow C$, $CG \rightarrow H$, $CG \rightarrow I$, $B \rightarrow H$
- Is AG a superkey of R ?
- Compute $(AG)^+$
  - Start with $\alpha^+ = AG$
  - $\mathrm{A} \rightarrow B$, $A \rightarrow \mathrm{C}$ cause $\alpha^+ = ABCG$
  - $CG \rightarrow H$, $CG \rightarrow I$ cause $\alpha^+ = ABCGHI$
- AG is a superkey of R !

Attribute-Set Closure Example (2)

- Relation schema $R(A, B, C, G, H, I)$
  - Dependencies:
    $A \rightarrow B$, $A \rightarrow C$, $CG \rightarrow H$, $CG \rightarrow I$, $B \rightarrow H$
- Is AG a candidate key of R ?
  - A candidate key is a minimal superkey
  - Compute attribute-set closure of all proper subsets of superkey; if we get R then it's not a candidate key
- Compute the attribute-set closures under F
  - $A^+ = ABCH$
  - $G^+ = G$
- AG is indeed a candidate key!

BCNF Revisited

- BCNF algorithm states, if R_i is a schema not in BCNF:
  - There is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ such that α is not a superkey for R_i
- Two points:
  - $\alpha \rightarrow \beta \in F^+$, not just in F
  - For R_i, only care about func. deps. where $\alpha \cup \beta \in R_i$
- How do we tell if R_i is not in BCNF?
  - Can use attribute-set closure under F to find if there is a dependency in F^+ that affects R_i
  - For each proper subset $\alpha \subset R_i$, compute α^+ under F
  - If α^+ doesn't contain R_i, but α^+ <u>does</u> contain any attributes in $R_i - \alpha$, then R_i is <u>not</u> in BCNF

BCNF Revisited (2)

- If α^+ doesn't contain R_i , but α^+ does contain any attributes in $R_i - \alpha$, then R_i is not in BCNF

- If α^+ doesn't contain R_i , what do we know about α with respect to R_i ?
  - α is not a candidate key of R_i
- If α^+ contains attributes in $R_i - \alpha$:
  - Let $\beta = R_i \cap (\alpha^+ - \alpha)$
  - We know there is some non-trivial functional dependency $\alpha \rightarrow \beta$ that holds on R_i
- Since $\alpha \rightarrow \beta$ holds on R_i , but α is not a candidate key of R_i , we know that R_i cannot be in BCNF.

BCNF Example

- Start with schema $R(A, B, C, D, E)$, and $F = \{ A \rightarrow B, BC \rightarrow D \}$
- Is R in BCNF?
  - Obviously not.
  - Using $A \rightarrow B$, decompose into $R_1(\underline{A}, B)$ and $R_2(A, C, D, E)$
- Are we done?
  - Pseudotransitivity rule says that if $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$
  - $AC \rightarrow D$ also holds on R_2, so R_2 is not in BCNF!
  - Or, compute $\{AC\}^+ = ABCD$. Again, R_2 is not in BCNF.

Database Constraints

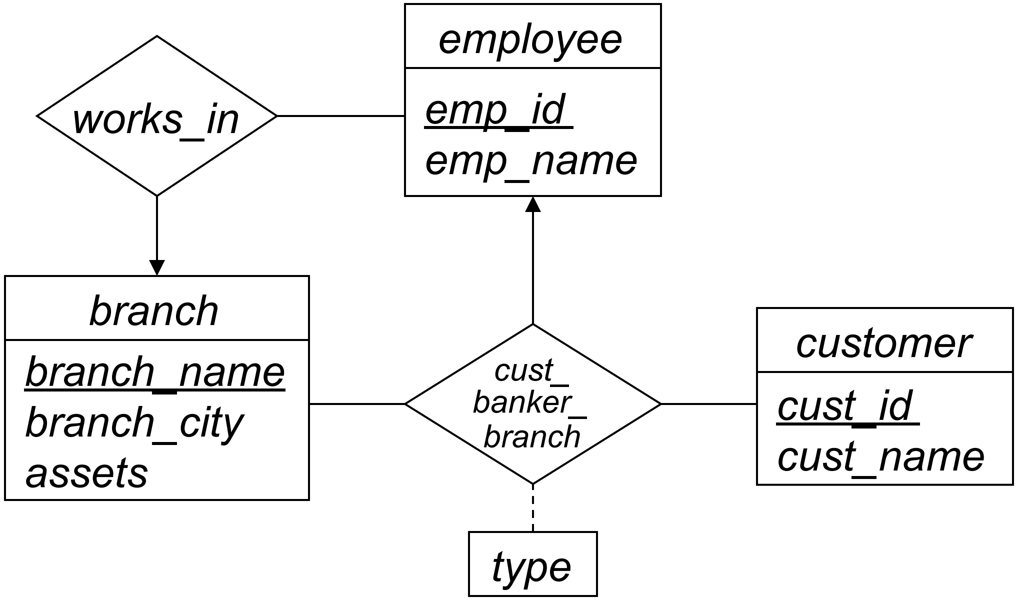
- Enforcing database constraints can easily become very expensive
  - Especially **`CHECK`** constraints!
- Best to define database schema such that constraint enforcement is <u>efficient</u>
- Ideally, enforcing a functional dependency involves only one relation
  - Then, can specify a key constraint instead of a multi-table **`CHECK`** constraint!

Example: Personal Bankers

- Bank sets a requirement on employees:
  - Each employee can work at only one branch
  - *emp_id* → *branch_name*
- Bank wants to give customers a personal banker at each branch
  - At each branch, a customer has only one personal banker
  - (A customer could have personal bankers at multiple branches.)
  - *cust_id*, *branch_name* → *emp_id*

Personal Bankers

- E-R diagram:



- Relationship-set schemas:

  works_in(*emp_id*, *branch_name*)

  cust_banker_branch(*cust_id*, *branch_name*, *emp_id*, *type*)

Personal Bankers (2)

- Schemas:

  works_in(<u>*emp_id*</u>, *branch_name*)

  cust_banker_branch(<u>*cust_id*</u>, <u>*branch_name*</u>, *emp_id*, *type*)

- Is this schema in BCNF?
  - *emp_id* → *branch_name*
  - *cust_banker_branch* isn't in BCNF
    - *emp_id* isn't a candidate key on *cust_banker_branch*
  - *cust_banker_branch* repeats *branch_name* unnecessarily, since *emp_id* → *branch_name*
- Decompose into two BCNF schemas:
  - *works_in* already has (<u>*emp_id*</u>, *branch_name*) $(\alpha \cup \beta)$
  - Create *cust_banker*(*cust_id*, *emp_id*, *type*) $(R - (\beta - \alpha))$

Personal Bankers (3)

- New BCNF schemas:

  works_in(*emp_id*, *branch_name*)

  cust_banker(*cust_id*, *emp_id*, *type*)

  - A customer can have one personal banker at each branch, so both *cust_id* and *emp_id* must be in the primary key
- Any problems with this new BCNF version?
  - Now we can't easily constrain that each customer has only one personal banker at each branch!
  - Could still create a complicated **`CHECK`** constraint involving multiple tables…

Preserving Dependencies

- The BCNF decomposition doesn't preserve this dependency:
  - *cust_id*, *branch_name* → *emp_id*
  - Can't enforce this dependency within a single table
- In general, BCNF decompositions are not <u>dependency-preserving</u>
  - Some functional dependencies are not enforceable within a single table
  - Can't enforce them with a simple key constraint, so they are more expensive
- Solution: Third Normal Form

Third Normal Form

- Slightly weaker than Boyce-Codd normal form
  - Preserves more functional dependencies
  - Also allows more repeated information!
- Given:
  - Relation schema *R*
  - Set of functional dependencies *F*
- *R* is in 3NF with respect to *F* if:
  - For all functional dependencies $\alpha \rightarrow \beta$ in F^+, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
    - $\alpha \rightarrow \beta$ is a trivial dependency
    - α is a superkey for *R*
    - Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R*

Third Normal Form (2)

- New condition:
  - Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R*
- A general constraint:
  - Doesn't require a single candidate key to contain all attributes in $\beta - \alpha$
  - Just requires that each attribute in $\beta - \alpha$ appears in *some* candidate key in *R*
  - …possibly even different candidate keys!

Personal Banker Example

- Our non-BCNF personal banker schemas again:
  - *works_in*(<u>*emp_id*</u>, *branch_name*)
  - *cust_banker_branch*(<u>*cust_id*</u>, <u>*branch_name*</u>, *emp_id*, *type*)
- Is this schema in 3NF?
  - *emp_id* → *branch_name*
  - *cust_id*, *branch_name* → *emp_id*
- *works_in* is in 3NF (*emp_id* is the primary key)
- What about *cust_banker_branch* ?
  - Both dependencies hold on *cust_banker_branch*
    - *emp_id* → *branch_name*, but *emp_id* isn't the primary key
    - *cust_id*, *branch_name* → *emp_id* ; is *emp_id* part of any candidate key on *cust_banker_branch* ?

Personal Banker Example (2)

- Look carefully at the functional dependencies:
  - Primary key of *cust_banker_branch* is (*cust_id*, *branch_name*)
    - { *cust_id*, *branch_name* } → *cust_banker_branch* (all attributes) (constraint arises from the E-R diagram & schema translation)
    - (Also specified this constraint: *cust_id*, *branch_name* → *emp_id*)
  - We also know that *emp_id* → *branch_name*
  - Pseudotransitivity rule: if $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$
    - { *emp_id* } → { *branch_name* }
    - { *cust_id*, *branch_name* } → *cust_banker_branch*
    - Therefore, { *emp_id*, *cust_id* } → *cust_banker_branch* also holds!
  - (*cust_id*, *emp_id*) is a candidate key of *cust_banker_branch*
- So *cust_banker_branch* is in fact in 3NF
  - *(And we need to enforce this second candidate key too…)*

Canonical Cover

- Given a relation schema, and a set of functional dependencies *F*
- Database needs to enforce *F* on all relations
  - Invalid changes should be rolled back
- *F* could contain a lot of functional dependencies
  - Dependencies might even logically imply each other
- Want a minimal version of *F*, that still represents all constraints imposed by *F*
  - Should be more efficient to enforce minimal version

Canonical Cover (2)

- A canonical cover F_c for F is a set of functional dependencies such that:
  - F logically implies all dependencies in F_c
  - F_c logically implies all dependencies in F
  - Can't infer any functional dependency in F_c from other dependencies in F_c
  - No functional dependency in F_c contains an extraneous attribute
  - Left side of all functional dependencies in F_c are unique
    - There are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in F_c such that $\alpha_1 = \alpha_2$

Extraneous Attributes

- Given a set F of functional dependencies
  - An attribute in a functional dependency is <u>extraneous</u> if it can be removed from F without affecting closure of F
- Formally: given F, and $\alpha \rightarrow \beta$
  - If $A \in \alpha$, and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$, then A is extraneous
  - If $A \in \beta$, and $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F, then A is extraneous
    - i.e. generate a new set of functional dependencies F' by replacing $\alpha \rightarrow \beta$ with $\alpha \rightarrow (\beta - A)$
    - See if F' logically implies F

Testing Extraneous Attributes

- Given relation schema *R*, and a set *F* of functional dependencies that hold on *R*
- Attribute *A* in $\alpha \rightarrow \beta$
- If $A \in \alpha$ (i.e. *A* is on left side of the dependency), then let $\gamma = \alpha - \{A\}$
  - See if $\gamma \rightarrow \beta$ can be inferred from *F*
  - Compute γ^+ under *F*
  - If $\beta \subseteq \gamma^+$, then *A* is extraneous in α

Testing Extraneous Attributes (2)

- Given relation schema R, and a set F of functional dependencies that hold on R
- Attribute A in $\alpha \rightarrow \beta$
- If $A \in \beta$ (on right side of the dependency), then try the altered set F'
  - $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$
  - See if $\alpha \rightarrow A$ can be inferred from F'
  - Compute α^+ under F'
  - If α^+ includes A, then A is extraneous in β

Computing Canonical Cover

- A simple way to compute the canonical cover of F

$F_c = F$
repeat
    apply union rule to replace dependencies in F_c of form
        $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\beta_2$
    find a functional dependency $\alpha \rightarrow \beta$ in F_c with an
        extraneous attribute
    /* Use F_c for the extraneous attribute test, not F !!! */
    if an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until F_c stops changing

Canonical Cover Example

- Functional dependencies F on schema (A, B, C)
  - $F = \{ A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C \}$
  - Find F_c
- Apply union rule to $A \rightarrow BC$ and $A \rightarrow B$
  - Left with: $\{ A \rightarrow BC, B \rightarrow C, AB \rightarrow C \}$
- A is extraneous in $AB \rightarrow C$
  - $B \rightarrow C$ is logically implied by F (obvious)
  - Left with: $\{ A \rightarrow BC, B \rightarrow C \}$
- C is extraneous in $A \rightarrow BC$
  - Logically implied by $A \rightarrow B, B \rightarrow C$
- $F_c = \{ A \rightarrow B, B \rightarrow C \}$

Another Canonical Cover Example

- Functional dependencies F on schema (A, B, C, D)
  - $F = \{ A \rightarrow B, BC \rightarrow D, AC \rightarrow D \}$
  - Find F_c
- Satisfies some of our constraints for F_c…
  - No functional dependency has extraneous attributes
  - All dependencies have a unique lefthand side
- Problem:
  - Can infer $AC \rightarrow D$ from the other two dependencies (pseudotransitivity)
  - Could argue that D is extraneous in $AC \rightarrow D$ (a bit weird)
  - Or, just argue that the entire dependency is extraneous

Canonical Covers

- A set of functional dependencies can have multiple canonical covers!
- Example:
  - $F = \{ A \rightarrow BC, B \rightarrow AC, C \rightarrow AB \}$
  - Has several canonical covers:
    - $F_c = \{ A \rightarrow B, B \rightarrow C, C \rightarrow A \}$
    - $F_c = \{ A \rightarrow B, B \rightarrow AC, C \rightarrow B \}$
    - $F_c = \{ A \rightarrow C, C \rightarrow B, B \rightarrow A \}$
    - $F_c = \{ A \rightarrow C, B \rightarrow C, C \rightarrow AB \}$
    - $F_c = \{ A \rightarrow BC, B \rightarrow A, C \rightarrow A \}$

FUNCTIONAL DEPENDENCY THEORY II

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 20

Last Time: Canonical Cover

- Last time, introduced concept of canonical cover
- A canonical cover F_c for F is a set of functional dependencies such that:
  - F logically implies all dependencies in F_c
  - F_c logically implies all dependencies in F
  - Can't infer any functional dependency in F_c from other dependencies in F_c
  - No functional dependency in F_c contains an extraneous attribute
  - Left side of all functional dependencies in F_c are unique
    - There are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in F_c such that $\alpha_1 = \alpha_2$

Extraneous Attributes

- Given a set *F* of functional dependencies
  - An attribute in a functional dependency is <u>extraneous</u> if it can be removed from *F* without affecting closure of *F*
- Formally: given *F*, and $\alpha \rightarrow \beta$
  - If $A \in \alpha$, and *F* logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$, then *A* is extraneous
  - If $A \in \beta$, and $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies *F*, then *A* is extraneous
    - i.e. generate a new set of functional dependencies F' by replacing $\alpha \rightarrow \beta$ with $\alpha \rightarrow (\beta - A)$
    - See if F' logically implies *F*

Testing Extraneous Attributes

- Given relation schema *R*, and a set *F* of functional dependencies that hold on *R*
- Attribute *A* in $\alpha \rightarrow \beta$
- If $A \in \alpha$ (i.e. *A* is on left side of the dependency), then let $\gamma = \alpha - \{A\}$
  - See if $\gamma \rightarrow \beta$ can be inferred from *F*
  - Compute γ^+ under *F*
  - If $\beta \subseteq \gamma^+$ then *A* is extraneous in α

Testing Extraneous Attributes (2)

- Given relation schema *R*, and a set *F* of functional dependencies that hold on *R*
- Attribute *A* in $\alpha \rightarrow \beta$
- If $A \in \beta$ (on right side of the dependency), then try the <u>altered</u> set F'
  - $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$
  - See if $\alpha \rightarrow A$ can be inferred from F'
  - Compute α^+ under F'
  - If α^+ includes *A* then *A* is extraneous in β

Computing Canonical Cover

- A simple way to compute the canonical cover of F

$F_c = F$
repeat
    apply union rule to replace dependencies in F_c of form
        $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\beta_2$
    find a functional dependency $\alpha \rightarrow \beta$ in F_c with an
        extraneous attribute
    /* Use F_c for the extraneous attribute test, not F !!! */
    if an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until F_c stops changing

Canonical Cover Example

- Functional dependencies F on schema (A, B, C)
  - $F = \{ A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C \}$
  - Find F_c
- Apply union rule to $A \rightarrow BC$ and $A \rightarrow B$
  - Left with: $\{ A \rightarrow BC, B \rightarrow C, AB \rightarrow C \}$
- A is extraneous in $AB \rightarrow C$
  - $B \rightarrow C$ is logically implied by F (obvious)
  - Left with: $\{ A \rightarrow BC, B \rightarrow C \}$
- C is extraneous in $A \rightarrow BC$
  - Logically implied by $A \rightarrow B, B \rightarrow C$
- $F_c = \{ A \rightarrow B, B \rightarrow C \}$

Canonical Covers

- A set of functional dependencies can have multiple canonical covers
- Example:
  - $F = \{ A \rightarrow BC, B \rightarrow AC, C \rightarrow AB \}$
  - Has several canonical covers:
    - $F_c = \{ A \rightarrow B, B \rightarrow C, C \rightarrow A \}$
    - $F_c = \{ A \rightarrow B, B \rightarrow AC, C \rightarrow B \}$
    - $F_c = \{ A \rightarrow C, C \rightarrow B, B \rightarrow A \}$
    - $F_c = \{ A \rightarrow C, B \rightarrow C, C \rightarrow AB \}$
    - $F_c = \{ A \rightarrow BC, B \rightarrow A, C \rightarrow A \}$

Another Example

- Functional dependencies F on schema (A, B, C, D)
  - $F = \{ A \rightarrow B, BC \rightarrow D, AC \rightarrow D \}$
  - Find F_c
- In this case, it may look like $F_c = F$…
- However, can infer $AC \rightarrow D$ from $A \rightarrow B$, $BC \rightarrow D$ (pseudotransitivity), so $AC \rightarrow D$ is extraneous in F
  - Therefore, $F_c = \{ A \rightarrow B, BC \rightarrow D \}$
- Alternately, can argue that D is extraneous in $AC \rightarrow D$
  - With $F' = \{ A \rightarrow B, BC \rightarrow D \}$, we see that $\{AC\}^+ = ACD$, so D is extraneous in $AC \rightarrow D$
  - (If you eliminate the entire RHS of a functional dependency, it goes away)

Lossy Decompositions

- Some schema decompositions lose information
- Example:

  employee(*emp_id*, *emp_name*, *phone*, *title*, *salary*, *start_date*)

  - Decomposed into:

  emp_ids(*emp_id*, *emp_name*)

  emp_details(*emp_name*, *phone*, *title*, *salary*, *start_date*)

- Problem:
  - *emp_name* doesn't uniquely identify employees
  - This is a lossy decomposition

Lossless Decompositions

- Given:
  - Relation schema *R*, relation *r*(*R*)
  - Set of functional dependencies *F*
- Let R_1 and R_2 be a decomposition of *R*
  - $R_1 \cup R_2 = R$
- The decomposition is lossless if, for <u>all</u> legal instances of *r* :

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

- A simple definition…

Lossless Decompositions (2)

- Can define with functional dependencies:
  - R_1 and R_2 form a lossless decomposition of R if at least one of these dependencies is in F^+:

    $$R_1 \cap R_2 \rightarrow R_1$$

    $$R_1 \cap R_2 \rightarrow R_2$$

- $R_1 \cap R_2$ forms a superkey of R_1 and/or R_2
  - Test for superkeys using attribute-set closure

Decomposition Examples (1)

- The *employee* example:

  employee(*emp_id*, *emp_name*, *phone*, *title*, *salary*, *start_date*)

- Decomposed into:

  emp_ids(*emp_id*, *emp_name*)

  emp_details(*emp_name*, *phone*, *title*, *salary*, *start_date*)

- *emp_name* is not a superkey of *emp_ids* or *emp_details*, so the decomposition is lossy

Decomposition Examples (2)

- The *bor_loan* example:

  bor_loan(*<u>cust_id</u>*, *<u>loan_id</u>*, *amount*)

- Decomposed into:

  borrower(*cust_id*, *loan_id*)

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

- *loan_id* is a superkey of *loan*, so the decomposition is lossless

BCNF Decompositions

- If R is a schema not in BCNF:
  - There is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ such that α is not a superkey for R
  - For simplicity, also require that $\alpha \cap \beta = \varnothing$
    - (if $\alpha \cap \beta \neq \varnothing$ then $(\alpha \cap \beta)$ is extraneous in β)
- Replace R with two schemas:

  $R_1 = (\alpha \cup \beta)$

  $R_2 = (R - \beta)$
    - (was $R - (\beta - \alpha)$, but $\beta - \alpha = \beta$, since $\alpha \cap \beta = \varnothing$)
- BCNF decomposition is lossless
  - $R_1 \cap R_2 = \alpha$
  - α is a superkey of R_1
  - α also appears in R_2

Dependency Preservation

- Some schema decompositions are not dependency-preserving
  - Functional dependencies that span multiple relation schemas are hard to enforce
  - e.g. BCNF may require decomposition of a schema for one dependency, and make it hard to enforce another dependency
- Can test for dependency preservation using functional dependency theory

Dependency Preservation (2)

- Given:
  - A set F of functional dependencies on a schema R
  - $R_1, R_2, \ldots, R_n$ are a decomposition of R
- The restriction of F to R_i is the set F_i of functional dependencies in F^+ that only has attributes in R_i
  - Each F_i contains functional dependencies that can be checked efficiently, using only R_i
- Find *all* functional dependencies that can be checked efficiently
  - $F' = F_1 \cup F_2 \cup \ldots \cup F_n$
  - If $F'^+ = F^+$ then the decomposition is dependency-preserving

Third Normal Form Schemas

- Can generate a 3NF schema from a set of functional dependencies F
- Called the <u>3NF synthesis algorithm</u>
  - Instead of decomposing an initial schema, generates schemas from a set of dependencies
- Given a set F of functional dependencies
  - Uses the canonical cover F_c
  - Ensures that resulting schemas are dependency-preserving

3NF Synthesis Algorithm

- Inputs: set of functional dependences F, on a schema R

let F_c be a canonical cover for F ;
i := 0;
for each functional dependency $\alpha \rightarrow \beta$ in F_c **do**
    if none of the schemas R_j, $j = 1, 2, \ldots, i$ contains $(\alpha \cup \beta)$ **then**
        $i := i + 1$;
        $R_i := (\alpha \cup \beta)$
    end if
done
if no schema R_j, $j = 1, 2, \ldots, i$ contains a candidate key for R **then**
    $i := i + 1$;
    R_i := any candidate key for R
end if
return $(R_1, R_2, \ldots, R_i)$

BCNF vs. 3NF

- Boyce-Codd Normal Form:
  - Eliminates more redundant information than 3NF
  - Some functional dependencies become expensive to enforce
    - The conditions to enforce involve multiple relations
  - Overall, a very desirable normal form!
- Third Normal Form:
  - All [more] dependencies are [probably] easy to enforce…
  - Allows more redundant information, which must be kept synchronized by the database application!
  - Personal banker example:

    works_in(<u>*emp_id*</u>, *branch_name*)

    cust_banker_branch(<u>*cust_id*</u>, <u>*branch_name*</u>, *emp_id*, *type*)
    - Branch names must be kept synchronized between these relations!

BCNF and 3NF vs. SQL

- SQL constraints:
  - Only key constraints are fast and easy to enforce!
  - Only easy to enforce functional dependencies $\alpha \rightarrow \beta$ if α is a key on some table!
  - Other functional dependencies (even "easy" ones in 3NF) may require more expensive constraints, e.g. **`CHECK`**
- For SQL databases with materialized views:
  - Can decompose a schema into BCNF
  - For dependencies $\alpha \rightarrow \beta$ not preserved in decomposition, create materialized view joining all relations in dependency
  - Enforce **unique**(α) constraint on materialized view
- Impacts both space and performance, but it works…

Multivalued Attributes

- E-R schemas can have multivalued attributes
- 1NF requires only atomic attributes
  - Not a problem; translating to relational model leaves everything atomic
- Employee example:

  employee(<u>*emp_id*</u>, *emp_name*)

  emp_deps(*emp_id*, *dependent*)

  emp_nums(*emp_id*, *phone_num*)

employee
<u>*emp_id*</u> *emp_name* { *phone_num* } { *dependent* }

- What are the requirements on these schemas for what tuples <u>must</u> appear?

Multivalued Attributes (2)

- Example data:

emp_id	*emp_name*
125623	Rick

employee

emp_id	*dependent*
125623	Jeff
125623	Alice

emp_deps

emp_id	*phone_num*
125623	555-8888
125623	555-2222

emp_nums

  - Every distinct value of multivalued attribute requires a separate tuple, including associated value of *emp_id*
- A consequence of 1NF, in fact!
  - If attributes could be nonatomic, could just store list of values in the appropriate column!
  - 1NF <u>requires</u> extra tuples to represent multivalues

Independent Multivalued Attributes

- Question is trickier when a schema stores several *independent* multivalued attributes
- Proposed combined schema:

  employee(*emp_id*, *emp_name*)

  emp_info(*emp_id*, *dependent*, *phone_num*)
- What tuples must appear in *emp_info* ?
  - *emp_info* is a relation
  - If an employee has M dependents and N phone numbers, *emp_info* must contain M × N tuples
    - Exactly what we get if we natural-join *emp_deps* and *emp_nums*
  - Every combination of the employee's dependents and their phone numbers

Independent Multivalued Attributes

- Example data:

emp_id	*emp_name*
125623	Rick

employee

emp_id	*dependent*	*phone_num*
125623	Jeff	555-8888
125623	Jeff	555-2222
125623	Alice	555-8888
125623	Alice	555-2222

emp_info

- Clearly has unnecessary redundancy
- Can't formulate functional dependencies to represent multivalued attributes
- Can't use BCNF or 3NF decompositions to eliminate redundancy in these cases

Multivalued Attributes Example

- Two employees: Rick and Bob
  - Both share a phone number at work
  - Both have two kids
  - Both have a kid named Alice
- Can't use functional dependencies to reason about this situation!
  - *emp_id* → *phone_num* doesn't hold since an employee can have several phone numbers
  - *phone_num* → *emp_id* doesn't hold either, since several employees can have the same phone number
  - Same with *emp_id* and *dependent*…

emp_id	*emp_name*
125623	Rick
127341	Bob

employee

emp_id	*phone_num*
125623	555-8888
125623	555-2222
127341	555-2222

emp_nums

emp_id	*dependent*
125623	Jeff
125623	Alice
127341	Alice
127341	Clara

emp_deps

Dependencies

- Functional dependencies rule out what tuples can appear in a relation
  - If $A \rightarrow B$ holds, then tuples cannot have same value for A but different values for B
  - Also called <u>equality-generating dependencies</u>
- <u>Multivalued dependencies</u> specify what tuples must be present
  - To represent a multivalued attribute's values properly, a certain set of tuples *must* be present
  - Also called <u>tuple-generating dependencies</u>

Multivalued Dependencies

- Given a relation schema R
  - Attribute-sets $\alpha \in R$, $\beta \in R$
  - $\alpha \twoheadrightarrow \beta$ is a multivalued dependency
  - "α multidetermines β"
- A multivalued dependency $\alpha \twoheadrightarrow \beta$ holds on R if, in any legal relation $r(R)$:

  For all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$,
  There also exists tuples t_3 and t_4 in r such that:
  - $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
  - $t_1[\beta] = t_3[\beta]$ and $t_2[\beta] = t_4[\beta]$
  - $t_1[R - \beta] = t_4[R - \beta]$ and $t_2[R - \beta] = t_3[R - \beta]$

Multivalued Dependencies (2)

- Multivalued dependency $\alpha \twoheadrightarrow \beta$ holds on R if, in any legal relation $r(R)$:

  For all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$,

  There also exists tuples t_3 and t_4 in r such that:

  - $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
  - $t_1[\beta] = t_3[\beta]$ and $t_2[\beta] = t_4[\beta]$
  - $t_1[R - \beta] = t_4[R - \beta]$ and $t_2[R - \beta] = t_3[R - \beta]$
- Pictorially:

	α	β	$R - (\alpha \cup \beta)$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

Multivalued Dependencies (3)

- Multivalued dependency:

	α	β	$R - (\alpha \cup \beta)$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

- If $\alpha \twoheadrightarrow \beta$ then $R - (\alpha \cup \beta)$ is independent of this fact
  - Every distinct value of β must be associated once with every distinct value of $R - (\alpha \cup \beta)$
- Let $\gamma = R - (\alpha \cup \beta)$
  - If $\alpha \twoheadrightarrow \beta$ then also $\alpha \twoheadrightarrow \gamma$
  - $\alpha \twoheadrightarrow \beta$ implies $\alpha \twoheadrightarrow \gamma$
  - Sometimes written $\alpha \twoheadrightarrow \beta \mid \gamma$

Trivial Multivalued Dependencies

- $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency on *R* if <u>all</u> relations *r*(*R*) satisfy the dependency
- Specifically, $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$, or if $\alpha \cup \beta = R$
- Employee examples:
  - For schema *emp_deps*(*emp_id*, *dependent*), *emp_id* $\twoheadrightarrow$ *dependent* is trivial
  - For *emp_info*(*emp_id*, *dependent*, *phone_num*), *emp_id* $\twoheadrightarrow$ *dependent* is <u>not</u> trivial

Inference Rules

- Can reason about multivalued dependencies, just like functional dependencies
  - There is a set of complete, sound inference rules for MVDs
- Example inference rules:
  - Complementation rule:
    - If $\alpha \twoheadrightarrow \beta$ holds on R, then $\alpha \twoheadrightarrow R - (\alpha \cup \beta)$ holds
  - Multivalued augmentation rule:
    - If $\alpha \twoheadrightarrow \beta$ holds, and $\gamma \subseteq R$, and $\delta \subseteq \gamma$, then $\gamma\alpha \twoheadrightarrow \delta\beta$ holds
  - Multivalued transitivity rule:
    - If $\alpha \twoheadrightarrow \beta$ and $\beta \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \gamma - \beta$ holds
  - Coalescence rule:
    - If $\alpha \twoheadrightarrow \beta$ holds, and $\gamma \subseteq \beta$, and there is a δ such that $\delta \subseteq R$, and $\delta \cap \beta = \varnothing$, and $\delta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ holds

Functional Dependencies

- Functional dependencies are also multivalued dependencies
- Replication rule:
  - If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$ too
  - Note there is an additional constraint from $\alpha \rightarrow \beta$: each value of α has *at most* one associated value for β
- Usually, functional dependencies are not stated as multivalued dependencies
  - The extra caveat is *important*, but not obvious in notation
  - Also, functional dependencies are easier to reason about!

Closures and Restrictions

- For a set D of functional and multivalued dependencies, can compute closure D^+
  - Use inference rules for both functional and multivalued dependencies to compute closure
- Sometimes need the restriction of D^+ to a relation schema R, too
- The restriction of D to a schema R_i includes:
  - All functional dependencies in D^+ that include only attributes in R_i
  - All multivalued dependencies of the form $\alpha \twoheadrightarrow \beta \cap R_i$, where $\alpha \subseteq R_i$, and $\alpha \twoheadrightarrow \beta$ is in D^+

Fourth Normal Form

- Given:
  - Relation schema R
  - Set of functional and multivalued dependencies D
- R is in 4NF with respect to D if:
  - For all multivalued dependencies $\alpha \twoheadrightarrow \beta$ in D^+, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
    - $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency
    - α is a superkey for R
  - Note: If $\alpha \rightarrow \beta$ then $\alpha \twoheadrightarrow \beta$
- A database design is in 4NF if all schemas in the design are in 4NF

4NF and BCNF

- Main difference between 4NF and BCNF is use of multivalued dependencies instead of functional dependencies
- Every schema in 4NF is also in BCNF
  - If a schema is not in BCNF then there is a nontrivial functional dependency $\alpha \rightarrow \beta$ such that α is not a superkey for R
  - If $\alpha \rightarrow \beta$ then $\alpha \twoheadrightarrow \beta$

4NF Decompositions

- Decomposition rule very similar to BCNF
- If schema R is not in 4NF with respect to a set of multivalued dependencies D :
  - There is some nontrivial dependency $\alpha \twoheadrightarrow \beta$ in D^+ where $\alpha \subseteq R$ and $\beta \subseteq R$, and α is not a superkey of R
    - Also constrain that $\alpha \cap \beta = \varnothing$
  - Replace R with two new schemas:
    - $R_1 = (\alpha \cup \beta)$
    - $R_2 = (R - \beta)$

Employee Information Example

- Combined schema:

  employee(*<u>emp_id</u>*, *emp_name*)

  emp_info(*emp_id*, *dependent*, *phone_num*)

  - Also have these dependencies:
    - *emp_id* $\rightarrow$ *emp_name*
    - *emp_id* $\twoheadrightarrow$ *dependent*
    - *emp_id* $\twoheadrightarrow$ *phone_num*
- *emp_info* is not in 4NF
- Following the rules for 4NF decomposition produces:

  (*emp_id*, *dependent*)

  (*emp_id*, *phone_num*)

  - Note: Each relation's candidate key is the entire relation. The multivalued dependencies are trivial.

Lossless Decompositions

- Can also define lossless decomposition with multivalued dependencies
  - R_1 and R_2 form a lossless decomposition of R if at least one of these dependencies is in D^+ :

$$R_1 \cap R_2 \twoheadrightarrow R_1$$

$$R_1 \cap R_2 \twoheadrightarrow R_2$$

Beyond Fourth Normal Form?

- Additional normal forms with various constraints
- Example: join dependencies
- Given R, and a decomposition R_1 and R_2 where $R_1 \cup R_2 = R$:
  - The decomposition is lossless if, for all legal instances of $r(R)$, $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$
- Can state this as a join dependency: $*(R_1, R_2)$
  - This is actually identical to a multivalued dependency!
  - $*(R_1, R_2)$ is equivalent to $R_1 \cap R_2 \twoheadrightarrow R_1 \mid R_2$

Join Dependencies and 5NF

- Join dependencies (JD) are a generalization of multivalued dependencies (MVD)
  - Can specify JDs involving N relation schemas, N ≥ 2
  - JDs are equivalent to MVDs when N = 2
  - Can easily construct JDs where N > 2, with no equivalent set of MVDs
- Project-Join Normal Form (a.k.a. PJNF or 5NF):
  - A relation schema R is in PJNF with respect to a set of join dependencies D if, for all JDs in D^+ of the form $*(R_1, R_2, \ldots, R_n)$ where $R_1 \cup R_2 \cup \ldots \cup R_n = R$, at least one of the following holds:
    - $*(R_1, R_2, \ldots, R_n)$ is a trivial join dependency
    - Every R_i is a superkey for R

Join Dependencies and 5NF (2)

- If a schema is in Project-Join Normal Form then it is also in 4NF (and thus, in BCNF)
  - Every multivalued dependency is also a join dependency
  - (Every functional dependency is also a multivalued dependency)
- One small problem:
  - There isn't a complete, sound set of inference rules for join dependencies!
  - Can't reason about our set of join dependencies *D*…
  - This limits PJNF's real-world usefulness

Domain-Key Normal Form

- Domain-key normal form (DKNF) is an even more general normal form, based on:
  - **Domain constraints:** what values may be assigned to attribute *A*
    - Usually inexpensive to test, even with `CHECK` constraints
  - **Key constraints:** all attribute-sets *K* that are a superkey for a schema *R* (i.e. $K \rightarrow R$)
    - Almost always inexpensive to test
  - **General constraints:** other predicates on valid relations in a schema
    - Could be very expensive to test!
- A schema *R* is in DKNF if the domain constraints and key constraints logically imply the general constraints
  - An "ideal" normal form difficult to achieve in practice…

OBJECTS AND DATABASES

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 21

Relational Model and 1NF

- Relational model specifies that all attribute domains must be atomic
  - A database schema is in 1NF if all attribute domains are atomic
- Not always preferred approach in real world use
- In relational model:

  employee(emp_id, emp_name)

  emp_phone(emp_id, phone_num)

  emp_deps(emp_id, dependent)

  - Need two joins just to get all data for an employee!

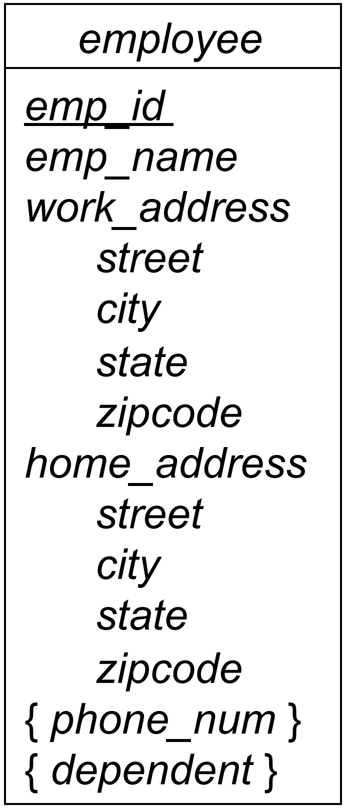
employee
emp_id *emp_name* { *phone_num* } { *dependent* }

Composite Types

- Also, frequently have composite types that are reused
- Example:
  - Add home/work addresses to design
- In relational model, composite types are decomposed

  employee(<u>emp_id</u>, emp_name,
  work_street, work_city, …)

- …but programming languages typically provide structures or classes for composite types!



Database Applications

- Programming languages have support for non-atomic types
  - Address class or structure:
    - street, city, state, zipcode
  - Arrays of phone numbers and dependents
- Application has to translate between relational model version and programming language representation
  - Annoying to deal with, at the least…
  - At worst, can have substantial application quality and performance impacts!

SQL User-Defined Types

- SQL:1999 includes User-Defined Types
  - Allows users to define non-atomic types where appropriate
  - (Make sure it's actually appropriate!)
  - Frequently abbreviated as UDT
- Multivalued types – arrays, sets, lists, etc.
  - Elements are all the same type
- Structured types – composite attributes
  - Elements may be different types

Non-Atomic Types for Employees

- Declare new UDT for addresses:

```
CREATE TYPE Address AS (
  street  VARCHAR(60),
  city    VARCHAR(40),
  state   CHAR(2),
  zipcode CHAR(9)
);
```

  - Only specify types, not constraints!
  - Defines a new structured type within the database schema
- For arrays, just add **`ARRAY[`*`n`*`]`** to column type
  - ***n*** is optional
  - Array elements have indexes 1 to ***n***

Using Non-Atomic Types

- Employee table:

```
CREATE TABLE employee (
  emp_id       INTEGER       PRIMARY KEY,
  emp_name     VARCHAR(100) NOT NULL,
  work_address Address      NOT NULL,
  home_address Address      NOT NULL,
  phone_nums   CHAR(12)     ARRAY[],
  dependents   VARCHAR(100) ARRAY[]
);
```

- Now all details of an employee are contained within a single table
  - E-R model maps directly into this design
- Retrieving all details of an employee will be fast

Structured Types in DML

- Accessing elements of a structured type:

```
SELECT emp_id, emp_name FROM employee
WHERE work_address.city = home_address.city;
```

- Specifying all values of a structured type:

```
UPDATE employee SET work_address =
  ('123 Main St.', 'Springfield', 'OH', '45505')
  WHERE emp_id = 5352;
```

- Specifying individual values of a structured type:

```
UPDATE employee SET work_address.city = 'Akron',
  work_address.zipcode = '44310'
  WHERE emp_id = 5352;
```

Array Types in DML

- Specifying all values of an array type:

```
UPDATE employee SET phone_nums =
  ARRAY['800-555-1234', '800-555-5678']
  WHERE emp_id = 5352;
```

- Specifying individual values of an array type:

```
UPDATE employee
  SET phone_nums[1] = '800-555-2345'
  WHERE emp_id = 5352;
```

- Order of elements in array is preserved!
  - Useful when order of values is meaningful
  - e.g. author-list in a database of research papers

Array Types in DML (2)

- Array columns are like nested relations
  - A nested relation is stored within a single column
- SQL:1999 provides nesting and unnesting operations for arrays
- To unnest an array value:

```
SELECT emp_id, emp_name, p.phone
  FROM employee AS e,
       UNNEST(e.phone_nums) AS p(phone)
  WHERE emp_id = 5352;
```

emp_id	*emp_name*	*phone*
5352	Bob Smith	800-555-2345
5352	Bob Smith	800-555-5678

Array Types in DML (3)

- Can also retrieve element ordering details

```
SELECT emp_id, emp_name, p.phone, p.p_index
  FROM employee AS e,
       UNNEST(e.phone_nums) WITH ORDINALITY
         AS p(phone, p_index);
```

emp_id	*emp_name*	*phone*	*p_index*
5352	Bob Smith	800-555-2345	1
5352	Bob Smith	800-555-5678	2

- Can use **COLLECT** to combine values into an array

```
SELECT emp_id, COLLECT(phone_num) AS phone_nums
FROM raw_employee_data GROUP BY emp_id;
```

  - Very similar to grouping and aggregation operation!
- Can also pass subquery to **ARRAY()** fn. to populate an array

SQL:1999 User Defined Types

- SQL:1999 user-defined types help with composite and multivalued attributes…
  - Can create schemas that don't incur join overheads for multivalued attributes
  - Can represent composite attributes more naturally within the SQL schema
- Still not quite the same as what programming languages can provide

Objects and Relations

- Many programming languages are object-oriented
  - Objects: encapsulation of state and behavior
  - Classes: specifications of objects' state and behavior
  - Also other features, such as class inheritance
    - A class can derive from a parent class
    - Child class has specialized capabilities and additional state
  - C++, Java, C#, Python, PHP, etc. All <u>widely</u> used.
- Typical approach for storing objects in a relational database:
  - Classes usually map to tables
  - Objects map to individual rows in a table

Objects and Relations (2)

- Relational databases aren't designed to store objects!
- "Object-relational impedance mismatch"
  - A number of serious issues arise when storing objects into a relational database
- Relational databases cannot enforce the same constraints that OOP languages can enforce!
  - Objects encapsulate and manage state very carefully
  - In a relational database, <u>all</u> values can be manipulated very easily
  - Storing object-data in a relational database increases potential for data corruption

Objects and Relations (3)

- Objects can reference each other
  - More akin to the network data model, which preceded the relational model
    - Objects are accessed by following specific object-references
    - Tuples are retrieved en masse via a query language
- Representing object identities and object references in a relational database can be tricky
  - In OOP languages, object-references are usually <u>opaque</u> to the user, and not manipulated directly
  - In relational model, identifying values are intentionally very visible and meaningful
    - Part of Codd's original intent with relational model

Objects and Relations (4)

- OOP languages also provide features not present in relational model
  - Ability to specify methods to be called on objects
  - Class inheritance and class hierarchies
  - Require careful modeling in a relational database, if it can be implemented at all!
    - Frequently have multiple choices for modeling, with different performance and space implications
- A number of other issues as well…
  - Some are more esoteric than others
- Can definitely live with most of these issues, but be aware of the mismatch in capabilities!

Addressing The Mismatch

- Two main approaches to object-relational mismatch
- Object-Oriented Databases (ODBMS/OODBMS)
  - Further extend SQL's type system to support basic object-oriented constructs
  - Database supports object-oriented abstractions directly and internally
- Persistent Programming Languages
  - Hide (R)DBMS storage operations from programmers
  - Automate the translation between programming-language objects and database storage

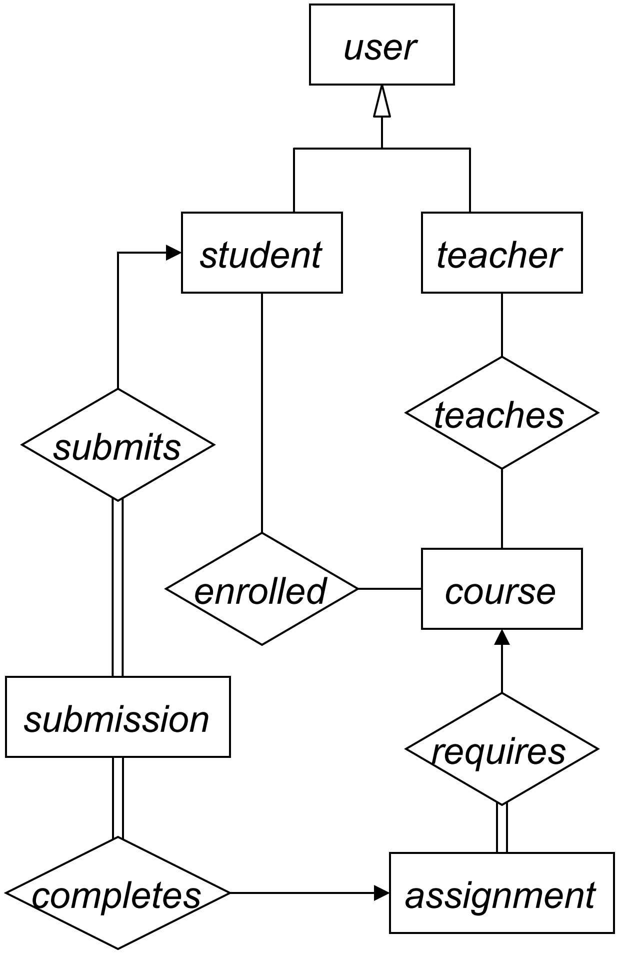
Object-Oriented Database Systems

- ODBMSes provide direct support for classes and objects
  - Define constructors and methods for classes
  - Define type hierarchies for classes
  - Provide object-reference support
- Inclusion of objects requires significant changes to the query language
  - Objects can refer to collections of objects
  - Must support path-based queries

ODBMS Example

- Course management system database schema
- Entities are objects in the ODBMS
  - Relationships specify object-references
- Retrieve names of students enrolled in CS121:

```
SELECT s.name FROM student s
WHERE s.enrolled.course.name = 'cs121';
```



ODBMS Operations

- ODBMSes must provide capabilities for:
  - Object-definition, similar to SQL DDL
  - Queries on objects, similar to SQL DML
- Object Data Management Group
  - Consortium founded in 1991, to create ODBMS specifications
- Standardization effort has had limited success
  - Several DB vendors offer ODBMS capabilities, but syntax and feature-sets vary pretty widely.
  - (not unlike SQL standard…)

ODBMS Operations (2)

- Object Description Language (ODL)
  - For specifying object-database schemas
  - Specify classes, class-members, and class inheritance hierarchies
- Object Query Language (OQL)
  - A SQL-like query language for querying object-databases
  - Most significant change is ability to specify "path expressions" that follow relationships between objects

Object-References

- All objects in an ODBMS have a unique identifier of some kind
  - Object Identifier (OID)
- Objects reference other objects using their OIDs
  - Akin to a pointer or reference to the object
- ODBMSes generally load referenced objects from disk, <u>as needed</u>.
  - Lazy loading, not eager loading
  - Objects tend to reference many other objects…
  - Object-loading must be done carefully, to avoid unnecessary resource usage!

ODBMS Summary

- Hasn't been widespread adoption of ODBMSes
  - Cost of switching is very high
    - A company's data is extremely valuable
    - Relational model is satisfactory for most needs, so why change?
- Many commercial databases provide a hybrid data model now
  - Object-Relational Database Management System (ORDBMS)
    - Blends object capabilities and relational database capabilities
  - Typically provide capabilities for simple type hierarchies, and simple class-method declarations

Persistent Programming Languages

- Most popular approach to object-relational impedance mismatch:
  - Create or enhance OOP languages to provide persistent objects directly in the language itself
- Normally, when a program terminates, all objects it created go away
  - These objects are <u>transient</u>
- <u>Persistent</u> objects are stored before termination
  - (in a database of some kind…)
  - Next time the program runs, persistent objects can be retrieved and used

Persistent Programming Languages (2)

- Persistent programming languages usually store objects in relational databases
  - PostgreSQL, MySQL, SQLite, Oracle, etc.
  - Also called "object-relational mapping layers" or simply "object-relational mappers" (abbrev. ORM)
- Type specification is <u>entirely</u> in the OO programming language itself
  - Able to leverage most types and OO capabilities of the programming language itself
  - Usually very few differences in capabilities between transient and persistent objects

Database Operations

- Database operations are usually entirely obscured from the programmer
  - Persistent object storage and retrieval is handled entirely by the framework itself
- Many ORM layers also provide automatic data-definition capabilities
  - Given a set of persistent objects, ORM layer can generate a SQL schema for those objects
  - Persistent objects are typically annotated to indicate "primary key" values, etc.

Automatic DDL Generation

- Persistence frameworks are becoming quite sophisticated with auto-DDL generation…
- Two main issues to consider!
- Database schema migration:
  - It's easy to change classes, or add new ones
  - Absolutely essential to have a migration path for existing data!
- Database performance:
  - ORM layers don't usually generate a schema tuned for high-performance and scalability

ORM Layers and Schema Migration

- Most ORM layers don't yet provide schema/data migration capabilities directly…
- Typically, external libraries/tools are available, to add schema-migration support to ORM layers
- General approach:
  - Take a snapshot of every stable version of data model
  - When the schema changes in *simple* ways, tool can generate the needed SQL DDL to migrate the schema
  - Tools also support manual data-migration steps for more involved changes
- <u>Always</u> back up data before using these tools! ☺

ORM Layers and Performance

- Most ORM layers also provide ability to run custom SQL on database before/after schema-generation
  - Add stored procedures and user-defined functions
  - Add indexes
  - Populate database with initial data
- To facilitate this, ORM layers frequently document exactly how table/column names are generated
- DDL that isn't specifically managed by the ORM can make data models significantly more fragile!
  - May need to change names/structure in multiple places

Manual DDL Creation

- Because of these issues, ORM layers also frequently support mapping objects to an existing schema
- You design the schema with specific needs in mind
  - Specify table indexes, partition large tables, etc.
- When a schema needs to change, it's easier to design a migration plan for your data
  - You have the "old" schema definition…
  - You provide the new schema definition yourself…
  - You can design the migration process for upgrading the database and preserving your data.

Persistence Framework Limitations

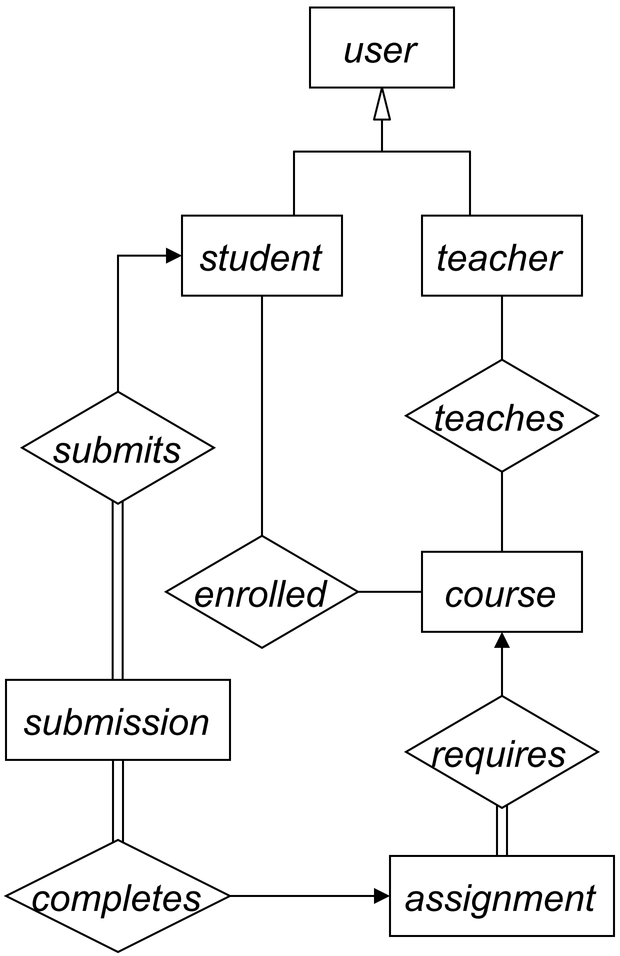
- Many persistence frameworks impose limitations on the kinds of schemas they support
  - Make sure to understand these limitations <u>before</u> designing schemas for these frameworks!
- Multiple-column primary keys
  - Supported by more advanced ORM layers…
  - …but they may not support multi-column foreign keys!
- Ternary relationships
  - Many ORM layers only support binary relationships
  - Need to model ternary relationships as a combination of binary relationships

Persistence Framework Challenges

- Many persistence frameworks also have limited support for database constraints
- Virtually all can handle referential integrity constraints
- Not all ORM layers can handle objects with multiple candidate keys
  - …but these days, most of them can.
- Be careful about general **`CHECK`** constraints!
- ORM layer must identify the cause of database errors generated by violating these constraints
  - Hard to do for a wide range of database vendors

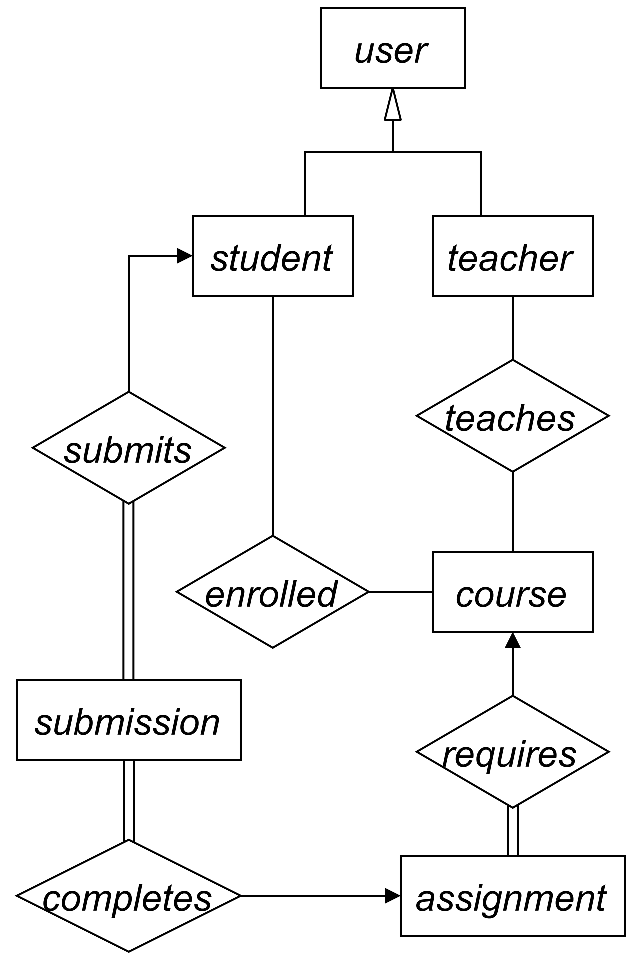
Loading Persistent Objects

- Like ODBMSes, persistent object frameworks must carefully manage object retrieval
- Example:
  - Retrieve all students in CS121
- Step 1: retrieve the **`Course`** object with name of "cs121"
  - The **`Course`** object will have a set of **`Student`**-references, a set of **`Assignment`**-refs, etc.



Loading Persistent Objects (2)

- Should the `Course` object eagerly or lazily load `Student` objects from the database?
  - We said we want all students…
  - Makes most sense to get all of them in one query.
- `Student` objects will have a collection of `Submission` objects, each of which has an `Assignment` object, …
  - ORM layer must get exactly what is needed, and no more!



Persistent-Object References

- OO programming language already has a way of referencing objects
  - e.g. pointers in C++, or references in Java/Python/…
- ORM layers must map between in-memory reference type and database reference type
  - A persistent object may not be loaded into memory yet, but other in-memory persistent objects refer to it
- Two kinds of persistent-object references:
  - A database-reference for when object isn't loaded yet
  - An in-memory reference for when the object is already in memory

Persistent-Object References (2)

- When a DB-reference is followed:
  - ORM layer loads object into memory from database
  - Then, ORM layer switches out the DB-reference for an in-memory reference
- In compiled languages, often implemented with <u>pointer-swizzling</u>
  - ORM layer uses special pointer-values for database-references to objects
  - When pointer is accessed, the ORM layer is notified (e.g. via a page-fault signal)
  - ORM layer loads the object, then directly changes the pointer value to point to the loaded object instead

Persistent-Object References (3)

- In interpreted (or VM-based) languages, often implemented with <u>hollow objects</u>
  - Before a persistent object is loaded, the reference actually points to a proxy
  - When the proxy is accessed, ORM layer retrieves the object from the DB
  - Proxy is replaced with the loaded object
- In-memory objects must also track state changes!
  - Writes to the object must flag the object as "dirty"
  - ORM layer ensures that dirty objects are saved to DB

Persistent Objects – Summary

- Much more popular solution to object-relational impedance mismatch
  - But, an admittedly incomplete solution to the mismatch
- Obscures much of the pain of moving objects to and from the database
- Also frequently provides ability to manage schema design directly (but watch out for limitations!)
- Some Java persistence frameworks:
  - Java Persistence API, Hibernate
- Some Python persistence frameworks:
  - Django models (+ South for migration), SQLAlchemy

DATA STREAMS AND DATABASES

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 22

Static and Dynamic Data Sets

- So far, have discussed relatively static databases
  - Data may change slowly over time…
  - Queries and updates operate against a static data-set
  - Especially true in context of transactions: within each txn, database appears to be unchanged by other txns
- Increasingly common to have data streams generated by various sources
  - An infinite, time-ordered sequence of tuples
  - Tuples have a particular schema, as before
  - One attribute in the tuple schema is a timestamp τ

Data Streams: Examples

- Many different examples of data streams

- Stock market data!
- Example: Volume-Weighted Average Price (VWAP)
  - Value computed over a time-window of stock trades
  - Window is fixed size, contains *n* stock trades
  - Trade *i* has price P_i, with Q_i shares changing hands
  - $P_{VWAP} = \Sigma P_i Q_i \div \Sigma Q_i$

Data Streams: Examples (2)

- Traffic-flow sensor networks generate data streams
  - Various estimates of traffic-flow characteristics, e.g. time/space mean speed, density, flow
- Seismographic networks generate many data streams!
  - Individual seismometers generate multiple data streams along different axes (vertical, N/S, E/W)
  - Other software consumes seismic data streams, identifies earthquakes, produces other data streams
- Computer network security monitoring systems
- Sensor networks in plants, factories, etc.

Data Streams in Relational Databases?

- How to implement Volume-Weighted Average Price (VWAP) computation with a relational database?
- Need to store the stream of stock prices into a table
  - Each stock price must have a timestamp
  - Example schema:
    - *stock_sales*(*sale_time*, *ticker_symbol*, *num_shares*, *price_per_share*)
- As new stock sales occur, store them in the table
- Need to eventually remove records from this table when we don't care anymore

Data Streams in Relational Databases?

- Periodically compute the VWAP against this table
  - ```
    SELECT ticker_symbol,
           SUM(price_per_share * num_shares) /
           SUM(num_shares)
    FROM stock_sales
    WHERE sale_time >= NOW() - INTERVAL 5 MINUTE
    GROUP BY ticker_symbol;
    ```
  - An index on (*sale_time*) will help select the rows in the window of interest
- Issues?
  - Performance: we throw away a significant amount of work every time the query is recomputed!
    ```

Data Streams in Relational Databases?

- Our query:
  - ```
    SELECT ticker_symbol,
           SUM(price_per_share * num_shares) /
           SUM(num_shares)
    FROM stock_sales
    WHERE sale_time >= NOW() - INTERVAL 5 MINUTE
    GROUP BY ticker_symbol;
    ```
  - *Can we do this incrementally instead?*
- Can easily apply same concepts as with materialized views to save and update intermediate state
  - As rows enter and leave the window of interest, we can update our rolling averages for each stock
  - Should make it very fast to generate the desired results!
    ```

Data Streams and Queries

- What if we want to update the output of our query every time the input data stream changes?
  - e.g. every time more stock values arrive, we update the corresponding VWAP immediately
  - (e.g. could implement this with triggers on *stock_sales*)
- What if we want to join the data stream against one or more relations?
- What if we want to generate a data stream based on the changes made to a relation?

Data Stream Management Systems

- Over the last few decades, many efforts to build Data Stream Management Systems (DSMS)
  - Like DataBase Management Systems (DBMS), but able to handle data streams as well as static tables
- Also called Complex Event Processing (CEP) systems
- Several approaches to building these systems…
- A popular approach:
  - Extend the relational model to add stream processing capabilities
  - (i.e. reuse the existing work of relational databases!)

Data Stream Management Systems (2)

- Many major research projects on relation-oriented stream databases:
  - Aurora/Borealis (Brown, Brandeis, MIT)
  - STREAM (Stanford)
  - TelegraphCQ (Berkeley)
- Commercial stream databases:
  - StreamBase (commercial version of Aurora/Borealis)
  - Truviso (commercial version of TelegraphCQ; acquired May 2012 by Cisco)
  - TIBCO Business Events, Oracle BAM

Stream Database Implementations

- Some stream databases were built by extending existing relational databases
  - e.g. TelegraphCQ and Truviso are extensions of PostgreSQL that incorporate data streams
  - Existing SQL syntax is extended to handle stream declarations, windowed queries on streams, etc.
- Many others are built from scratch
  - Custom stream-processing engines that are database-agnostic, or that don't use a relational database
  - Usually have a specific focus, e.g. high-volume data streams, low-latency results, specific kinds of queries, etc.
- Today: focus on relation-oriented stream databases

Data Stream Management Systems (3)

- In a relational database:
  - Data is static! (As long as no DML is issued…)
  - Issue queries against DB whenever we need results.
- In a stream database:
  - Stream data changes continually! Thus, results of queries against a stream also change continually.
  - Such queries are called continuous queries.
  - Register continuous queries with the database server.
  - As stream data changes, DB can incrementally update and output query results efficiently.

Stream Data Model

- A data stream S is an infinite, time-ordered multiset of tuples, one attribute being a timestamp τ
  - τ denotes the logical arrival time of the tuple into the system
- A relation R is an unordered multiset of tuples that varies over time
  - $R(\tau)$ is the version of the relation at a point in time τ
- A continuous query Q is constructed from a tree of operators against streams S_i and relations R_i

Continuous Query Operators

- Relation-to-relation operators take one or two relations and produce a relation
  - Exactly like standard relational algebra, except that relations in stream databases have a notion of time
  - Uses most recent versions of involved relations $R_i(\tau)$
- Relation-to-stream operators take a relation and produce a data stream
  - Typically defined to produce a stream containing changes made to a relation R_i
  - e.g. ISTREAM(R) produces tuples inserted into R at time τ
    - A stream of tuples $\langle s, \tau \rangle$ where $s \in R(\tau) - R(\tau - 1)$
  - e.g. RSTREAM(R) produces a stream of all tuples in R at τ

Continuous Query Operators (2)

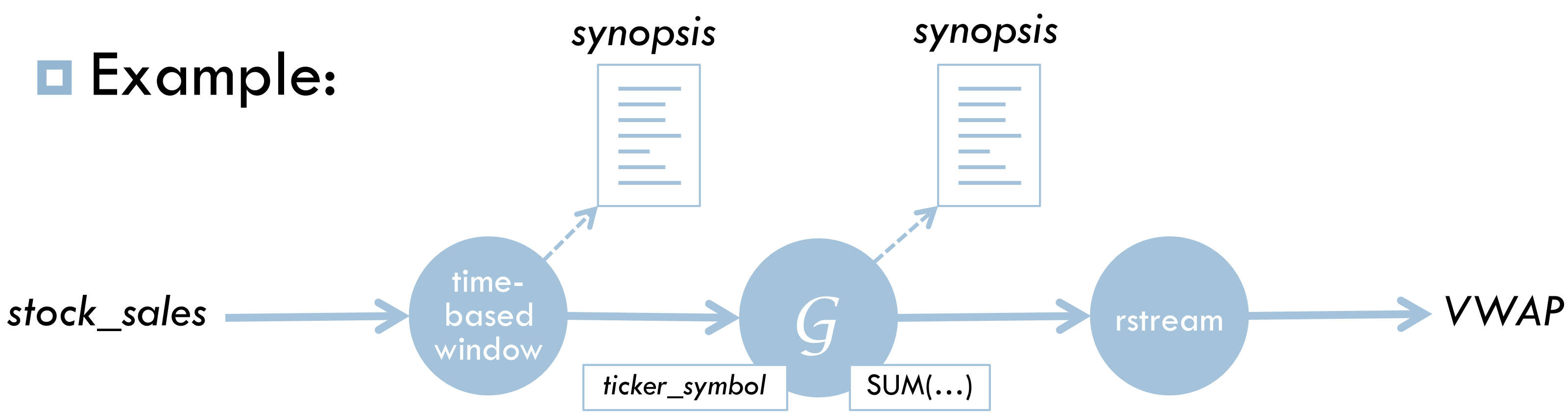
- Stream-to-relation operators take a data stream *S* and produce a relation *R*
  - Most continuous queries only care about the *recent* tuples in a data stream…
  - Define a <u>sliding window</u> on a stream that ends at time τ
- Tuple-based sliding windows contain the *N* most recent tuples from the data stream *S*
- Time-based sliding windows contain tuples from *S* that fall in a range of timestamps
  - $R(\tau)$ contains all tuples from *S* with timestamp in $[\tau - \omega, \tau]$
  - Also provide support for "now" tuples, where $\omega = 0$

Continuous Queries

- When a continuous query is added to the database, a plan is constructed from these operators
- Example: compute volume-weighted average stock prices from stream of stock sales
  - Convert stock-sale data stream into a relation using a time-based sliding-window operator over last 5 minutes of data
  - Compute a relation containing aggregates using standard grouping/aggregation operations
  - Convert entire relation into another stream containing results
- Problem: as stated, this is still expensive
  - Every time, results are computed entirely from scratch!

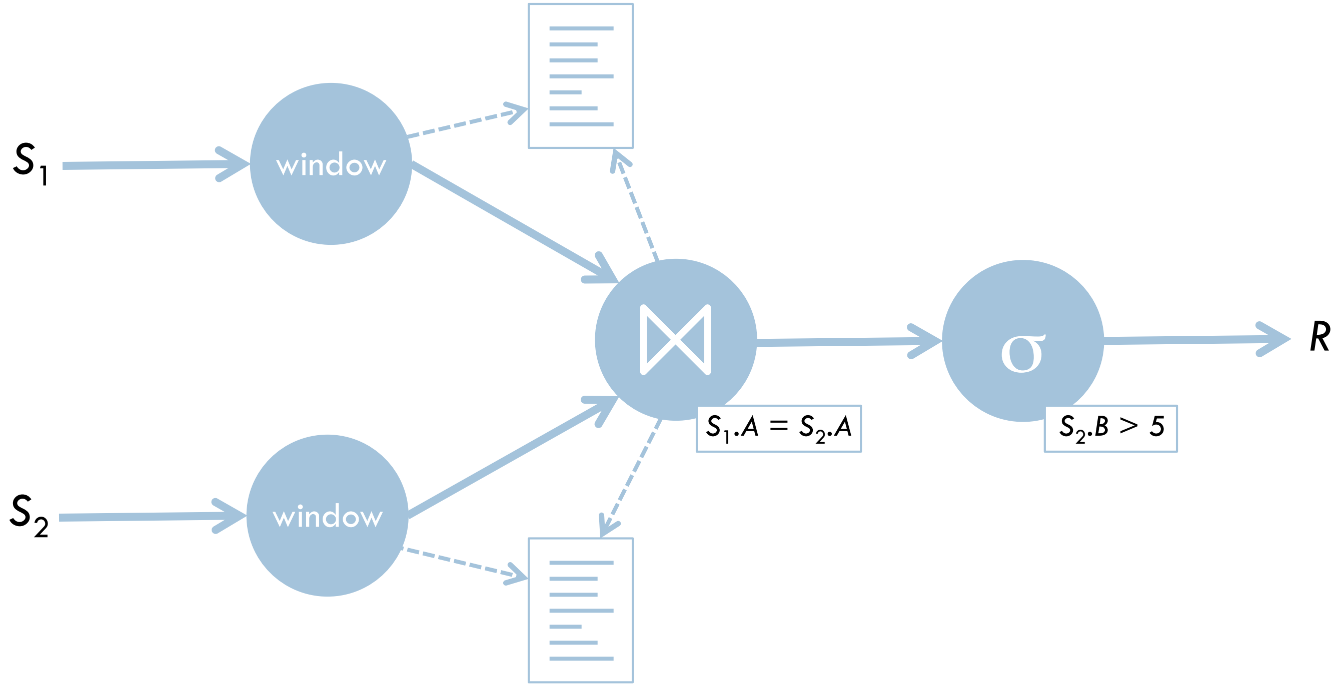
Continuous Queries (2)

- A continuous query operator can maintain a <u>synopsis</u> of its most recent results
  - A relation containing the rows used to generate results for most recent time τ
  - When rows enter or exit the sliding window, synopsis can be incrementally updated very efficiently
  - Example:



Continuous Queries (3)

- Can construct very complex queries, even with these [relatively] simple operators
- Example: windowed join of two data streams
  - Shared synopses across different nodes



Continuous Query Languages

- Some continuous query languages (CQL) are extensions of SQL
  - Particularly in implementations built on relational DBs
- Include ability to create and remove streams
  - ```
    CREATE STREAM stock_sales (
        sale_time TIMESTAMP,
        ticker_symbol VARCHAR(10),
        num_shares INTEGER,
        price_per_share NUMERIC(7, 2)
    );
    ```
  - `DROP STREAM stock_sales;`
- (Actual stream data usually arrives over network and must be in a specific format for the database to use it)
    ```

Continuous Query Languages (2)

- Need a way to indicate what column is timestamp τ
- Example: Truviso syntax:
  - ```
    CREATE STREAM stock_sales (
            sale_time TIMESTAMP CQTIME
                USER GENERATED,
            ticker_symbol VARCHAR(10),
            num_shares INTEGER,
            price_per_share NUMERIC(7, 2)
    );
    ```
  - Can also have system-generated timestamps
- Other stream databases use similar mechanisms, e.g. TelegraphCQ has **`TIMESTAMPCOLUMN`** modifier
    ```

Continuous Query Languages (3)

- Can also specify if a stream is archived or not
  - An archived stream can be queried for historical data; an unarchived stream cannot.
  - Most CQLs have **TYPE [ARCHIVED|UNARCHIVED]**
- To preserve historical data:
  - ```
    CREATE STREAM stock_sales (
        sale_time TIMESTAMP CQTIME
            USER GENERATED,
        ticker_symbol VARCHAR(10),
        num_shares INTEGER,
        price_per_share NUMERIC(7, 2)
    ) TYPE ARCHIVED;
    ```
    ```

Continuous Queries

- Can issue queries against both relations and streams
- If a stream is involved, need to specify the window!
- Stanford STREAM CQL annotates streams within the query:
  - ```
    SELECT ticker_symbol,
           SUM(num_shares * price_per_share) /
           SUM(num_shares)
    FROM stock_sales [RANGE 5 MINUTES]
    GROUP BY ticker_symbol;
    ```
- Truviso has a similar annotation:
  - ```
    SELECT ticker_symbol,
           SUM(num_shares * price_per_share) /
           SUM(num_shares)
    FROM stock_sales < VISIBLE '5 MINUTES' >
    GROUP BY ticker_symbol;
    ```

Continuous Queries (2)

- TelegraphCQ has a separate window specification:
  - ```
    SELECT ticker_symbol,
           SUM(num_shares * price_per_share) /
           SUM(num_shares)
    FROM stock_sales
    GROUP BY ticker_symbol
    WINDOW stock_sales ['5 MINUTES'];
    ```
    ```

Derived Streams

- Continuous queries run once and produce their results, just like standard queries
- Can create <u>derived streams</u> from continuous queries
  - Query is persistent, and produces a data stream of results
  - Results in output data stream have timestamps, as expected
- Truviso syntax:
  - ```
    CREATE STREAM vol_weighted_avg_price AS
        (SELECT ticker_symbol,
                SUM(num_shares * price_per_share) /
                SUM(num_shares)
         FROM stock_sales < VISIBLE '5 MINUTES'
                            ADVANCE '5 SECONDS' >
         GROUP BY ticker_symbol);
    ```
  - **ADVANCE** keyword specifies how often to generate results
    ```

Stream-Processing Challenges

- Many challenges in implementing stream databases
- Frequently, data streams are bursty and can have very high volumes
  - Peak message rate >> average message rate
  - Latency between input data and results skyrockets
- Most common approach: load-shedding
  - Simply drop some of the input tuples out of the stream!
  - Stream DBs use statistics gathered about stream data to drop/summarize tuples to maximize accuracy of results
  - May involve dropping tuples from multiple sources in a query to ensure accuracy is maximized

Stream-Processing Challenges (2)

- Other databases use windowed/aggregate operators to achieve a similar result:
  - Use windowed aggregate operations to generate multiple granularities of a given data stream
  - Different granularities will produce different volumes of tuples…
- As system load varies, can react very easily by changing the granularity of data being used
  - Choose the finest granularity of input data that the current system load will allow

Stream-Processing Challenges (3)

- Tuples in a stream don't always arrive in order!
  - A tuple's timestamp can be set by database when the tuple arrives, or it can be an externally specified field
  - e.g. stock-sale records might already include timestamp
- Generally a characteristic of a specific stream…
  - Can specify a stream's slack: the maximum "out-of-order"-ness that will be allowed
  - Tuples that arrive later than the specified slack are ignored!

Stream-Processing Challenges (4)

- Truviso example:
  - ```
    CREATE STREAM stock_sales (
         sale_time TIMESTAMP CQTIME
             USER GENERATED SLACK '1 MINUTE',
         ticker_symbol VARCHAR(10),
         num_shares INTEGER,
         price_per_share NUMERIC(7, 2)
    );
    ```
- Problem: if tuples can arrive out of order, a query may have already generated invalid results…
  - Could simply delay outputting results by the specified amount of slack. Then we know results won't change…
    ```

Stream-Processing Challenges (5)

- Another approach: output revisions to answers!
- Some input data streams can also include revisions to previous records
  - (commercial stock ticker feeds, for example)
  - e.g. correct invalid values, add extra rows, remove rows
- A few stream databases (e.g. Borealis) can handle revisions on data streams
- If all tuples affected by revision are still in memory:
  - Recompute affected results incrementally, using old and new versions of dataset for the affected timestamp(s)
  - Only output new records that actually changed
  - (Revised outputs may force other revisions to be made…)

Stream-Processing Challenges (6)

- If tuples affected by revision no longer in memory:
  - Must scan and replay archived data-stream tuples to find all relevant tuples
- Also likely that query nodes' synopses will no longer contain data for the affected timestamps ☹
  - Must recompute all work from scratch; can't use incremental updates
- As before, only issue revised output records for results that actually change
- If revisions are far enough in past, may cause *many* results to be recomputed and revised ☹

DSMS Summary

- Possible to use relational databases to continuously process data streams, but not very efficient!
- Relation-oriented data stream management systems (DSMS) blend together standard relational model databases with stream-processing capabilities
- Very powerful solution for problems involving data stream processing!

References

- STREAM: The Stanford Data Stream Management System [Arasu et al]
  - Introduction to how the relational model is extended to include data stream processing
- Models and Issues in Data Stream Systems [Babcock et al]
  - Discussion of data models and query languages used in relational data stream management systems
- The 8 Requirements of Real-Time Stream Processing [Stonebraker, Çetintemel, Zdonik]
  - A good summary of the major requirements of DSMSes
- Revision Processing in a Stream Processing Engine [Ryvkina, Maskey, Cherniack, Zdonik]
  - Description of Borealis revision processing

References (2)

- Processing Flows of Information: From Data Stream to Complex Event Processing [Cugola, Margara]
  - Excellent survey of many different data stream engines!
- Stanford STREAM research project:
  - http://infolab.stanford.edu/stream/
- Aurora and Borealis research projects:
  - http://www.cs.brown.edu/research/aurora/
  - http://www.cs.brown.edu/research/borealis/
- TelegraphCQ research project:
  - http://telegraph.cs.berkeley.edu/

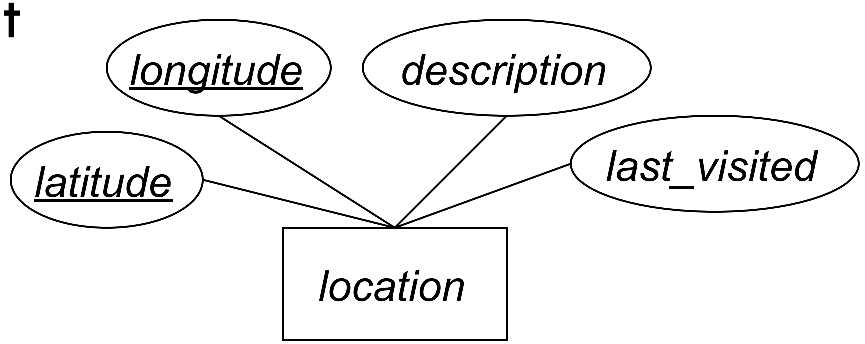
ALTERNATE SCHEMA DIAGRAMMING METHODS
DECISION SUPPORT SYSTEMS

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 23

E-R Diagramming

- E-R diagramming techniques used in book are similar to ones used in industry
  - Still, plenty of variation on how schemas are diagrammed
- Some books use a different diagramming technique
  - Attributes are represented as ovals attached to entity-set
  - Much harder to lay out!
  - Takes up a lot of room



- These methods don't include types or other constraints

Unified Modeling Language

- A standardized set of diagrams for specifying software systems
- Focuses on three major areas:
  - Functional requirements:
    - What is the system supposed to do?
    - Who may interact with the system, and what can they do?
  - Static structure:
    - What subsystems comprise the system?
    - What classes are needed, and what do they do?
  - Dynamic behavior:
    - What steps are taken to perform a given operation?
    - What is the flow of control through a system, and where are the decision points?

UML Class Diagrams

- UML class diagrams are typically used to diagram database schemas
  - Classes are similar to schemas
  - Objects are similar to tuples
- Two kinds of class diagrams for data modeling:
  - Logical data models (which are also called “E-R diagrams”)
    - Conceptual schema specification
    - Diagramming entity-sets and relationships, along the lines of the traditional E-R model, but not exactly like it
  - Physical data models
    - Implementation schema specification
    - Diagramming tables and foreign-key references
    - From a SQL perspective, is actually logical and view levels

UML Data Modeling

- Entity-sets and tables are represented as boxes
  - First line is entity-set name
  - Subsequent lines are attributes
  - First group of attributes usually the entity-set's primary key
    - Bolded, or marked with a *, +, or #

location
latitude
longitude
description
last_visited

- Table diagrams often also include type details

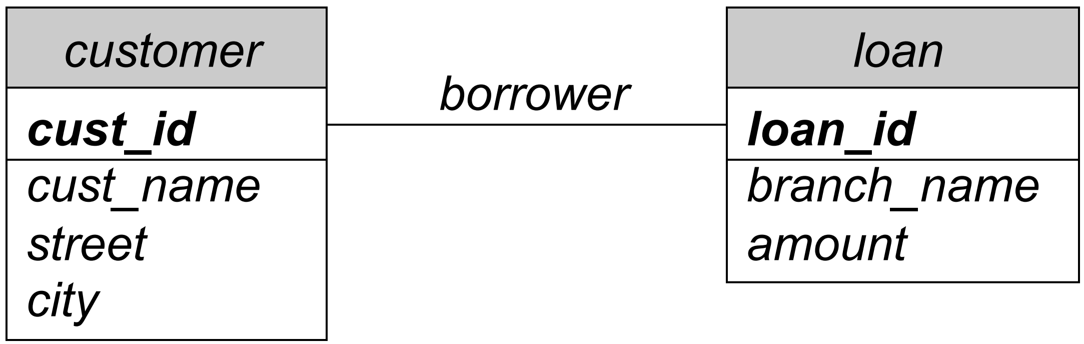
location	
latitude	NUMERIC(8, 5)
longitude	NUMERIC(8, 5)
description	VARCHAR(1000)
last_visited	TIMESTAMP

UML Relationships

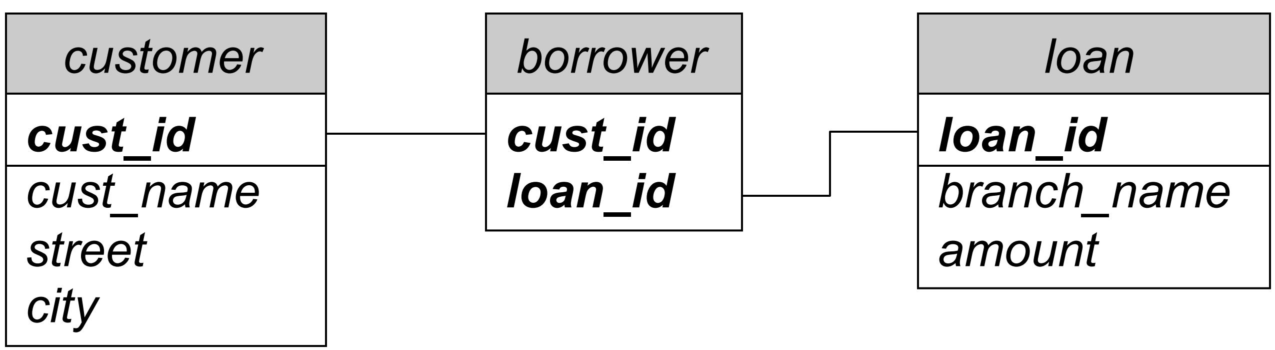
- Relationships are represented with a simple line
  - No diamond for the relationship
  - Relationship's name or role can be specified on line
- When modeling entity-sets (logical data model):
  - Don't include foreign-key columns
  - Foreign-key columns are implied by the relationship itself
- When modeling tables (physical data model):
  - Related tables actually include the foreign-key columns
  - Some relationships are modeled as separate tables
    - e.g. many-to-many relationships require a separate table

UML Relationship Examples

- Logical data model:

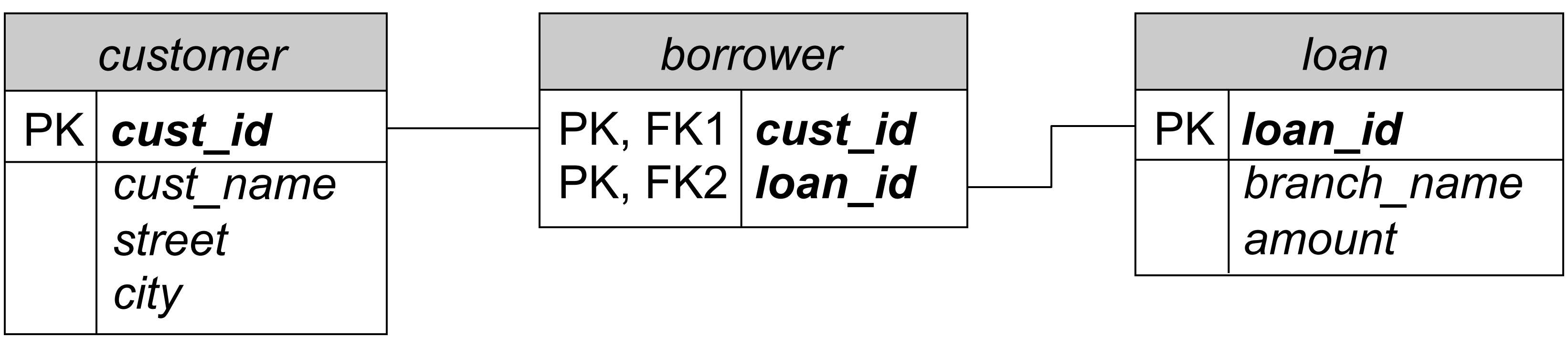


- Physical data model:
  - (would normally include type information too)



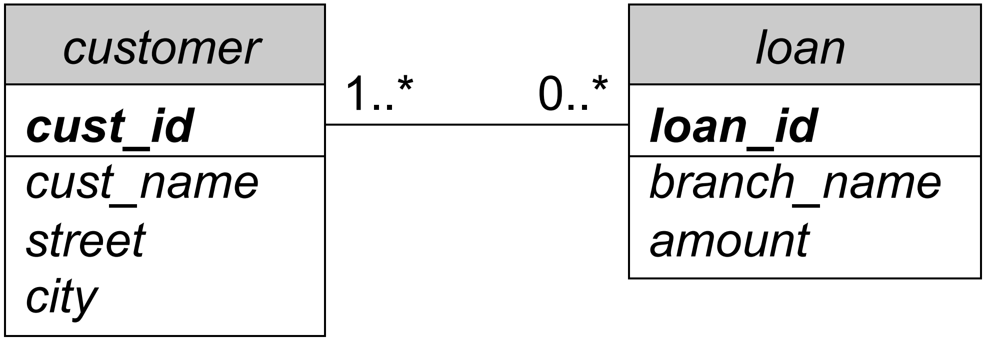
Annotating Keys

- Sometimes keys are indicated with two-character annotations
  - PK = primary key
  - FK = foreign key
- Candidate keys are specified with:
  - AK = alternate key
  - SK = surrogate key
    - (No difference between the two terms…)



Mapping Cardinalities

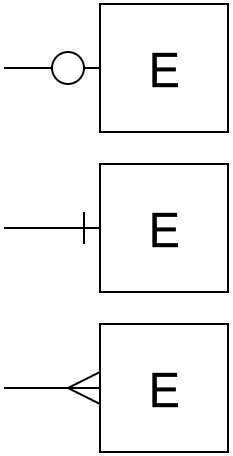
- Can specify numeric mapping constraints on relationships, just as in E-R diagrams
  - Can specify a single number for an exact quantity
  - lower..upper for lower and upper bounds
  - Use * for "many"



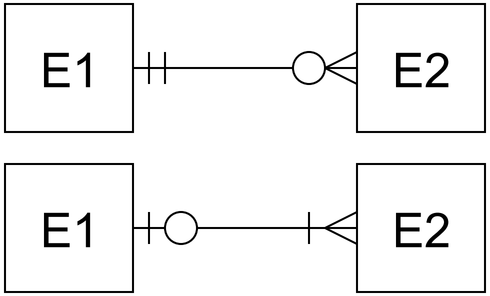
- Example:
  - Each customer is associated with zero or more loans
  - Each loan is associated with one or more customers

Information Engineering Notation

- Can also use Information Engineering Notation to indicate mapping cardinalities
  - Also called “crow’s foot notation”
- Symbols:
  - Circle means “zero”
  - Line means “one”
  - Crow’s foot means “many”



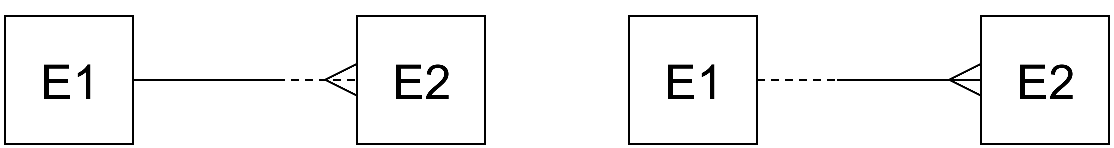
- Can combine symbols together
  - circle + line = “zero or one”
  - line + line = “exactly one”
  - line + crow’s foot = “one or more”



Barker's Notation

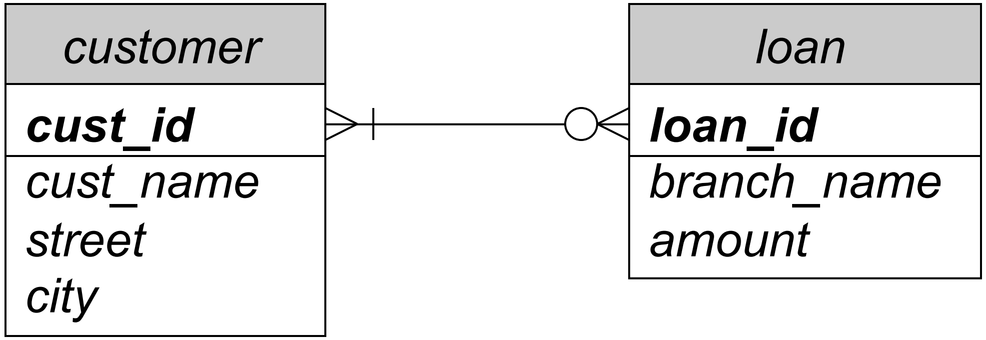
- A variant of Information Engineering Notation
- Symbols:
  - A solid line means "exactly one"
  - A dotted line means "zero or one"
  - Crow's foot + solid line means "one or more"
  - Crow's foot + dotted line means "zero or more"
- IE Notation:

- Barker's Notation:

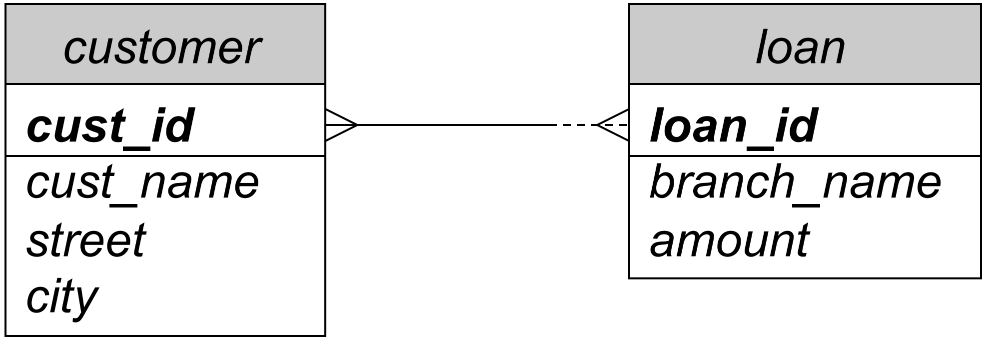


Examples

- Information Engineering notation:

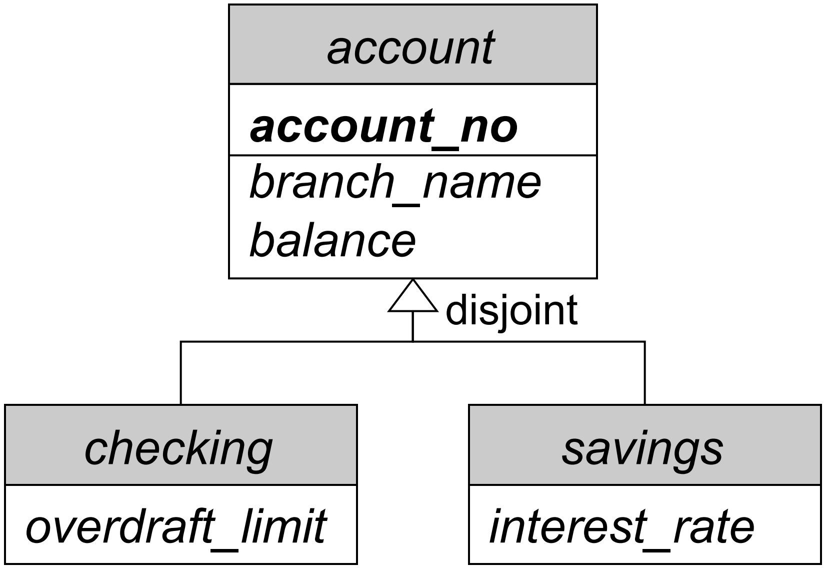


- Barker's notation:



Generalization and Specialization

- Can represent generalization in UML class diagrams
  - Open arrow, pointing from child to parent
- Can specify "disjoint" for disjoint specialization



UML Diagramming Summary

- <u>Very</u> good idea to learn UML diagramming!
  - Used extensively in the software industry
  - You can create visual diagrams of software, and other people will actually understand you! ☺
- Significant variation in details of how <u>data models</u> are diagrammed
  - Data modeling is *still* not yet a standard part of UML specification
  - Good to be familiar with all major techniques

OLTP and OLAP Databases

- OLTP: Online Transaction Processing
  - Focused on many short transactions, involving a small number of details
  - Database schemas are normalized to minimize redundancy
  - Most database applications are OLTP systems
- OLAP: Online Analytic Processing
  - Focused on analyzing patterns and trends in very large amounts of data
  - Database schemas are denormalized to facilitate better processing performance

Decision Support Systems

- Decision Support Systems (DSS) facilitate analyzing trends in large amounts of data
  - DSS don't actually identify the trends themselves
  - Are a tool for analysts familiar with what the data means
  - Analyze collected data to measure effectiveness of current strategies, and to predict future trends
  - Increasingly common for analysts to use data mining on a system to identify patterns and trends, too
- Decision support systems must provide:
  - Specific kinds of summary data generated from the raw input data
  - Ability to break down summary data along different dimensions, e.g. time interval, location, product, etc.

Decision Support Systems (2)

- OLAP databases are frequently part of decision support systems
  - Called data warehouses
  - Capable of storing, summarizing, and reporting on *huge* amounts of data
- Example data-sets presented via DSS:
  - Logs from web servers or streaming media servers
  - Sales records for a large retailer
  - Banner ad impressions and click-throughs
  - Very large data sets (frequently into petabyte range)
- Need to:
  - Generate summary information from these records
  - Facilitate queries against the summarized data

DSS Databases

- Example: sales records for a large retailer
  - Customer ID, time of sale, sale location
  - Product name, category, brand, quantity
  - Sale price, discounts or coupons applied
- Billions of sales records to process
  - Summary results may also include millions/billions of rows!
- Could fully normalize the database schema…
  - Information being analyzed and reported on would be spread through multiple tables
  - Analysis/reporting queries would require many joins
  - Often imposes a *heavy* performance penalty
- This approach is prohibitive for such systems!

Example Data Warehouses

- Starbucks figures from 2007:
  - 5TB data warehouse, growing by 2-3TB/year
- Wal-Mart figures from 2006:
  - 4PB data warehouse
- eBay figures from 2009:
  - Two data warehouses
  - Data warehouse 1: Teradata system
    - >2PB of user data
  - Data warehouse 2: Greenplum system
    - 6.5PB of user data
    - 17 trillion records – 150 billion new records each day
    - >50TB added each day

Measures and Dimensions

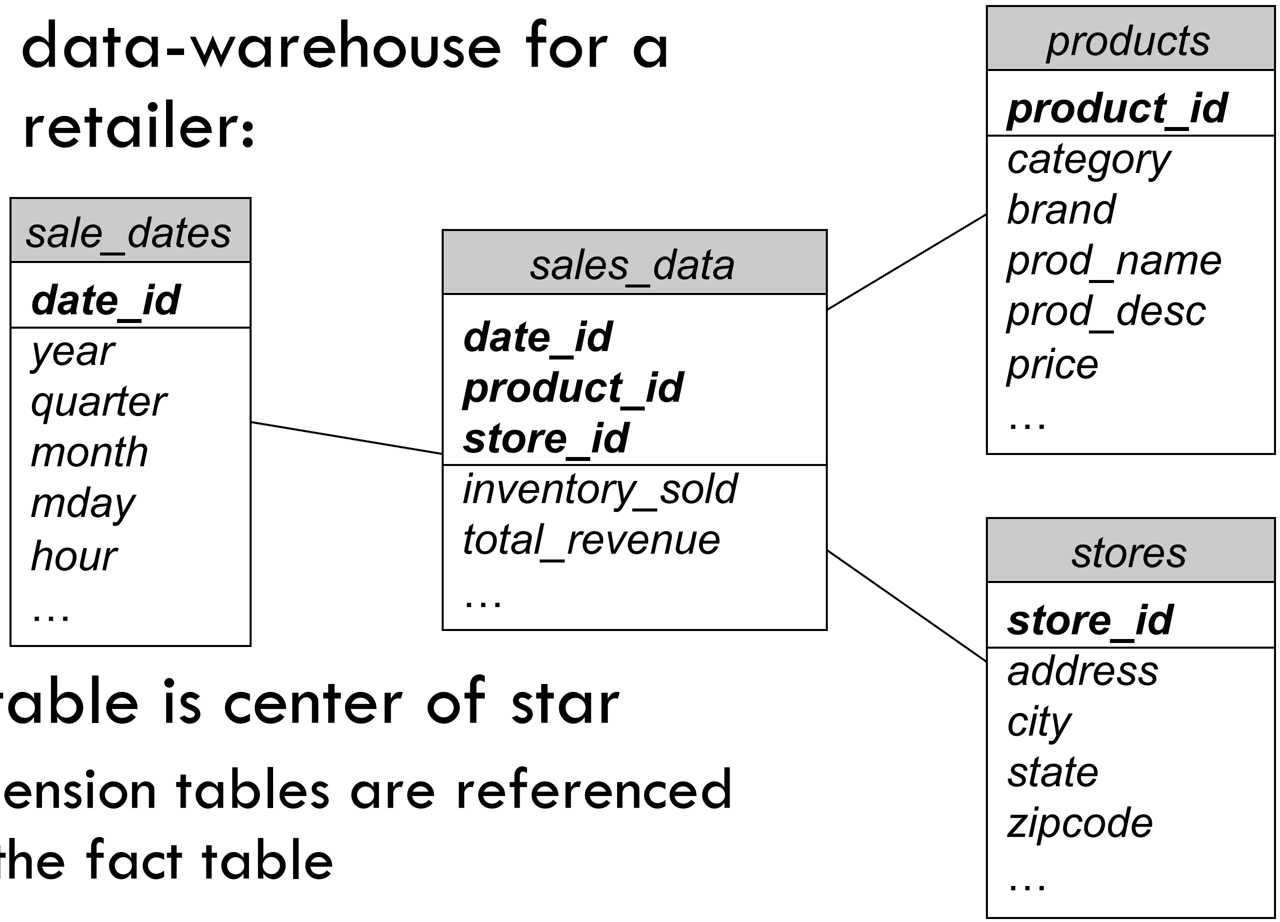
- Analysis queries often have two parts:
- A measure being computed:
  - "What are the total sales figures…"
  - "How many customers made a purchase…"
  - "What are the most popular products…"
- A dimension to compute the result over:
  - "…per month over the last year?"
  - "…at each sales location?"
  - "…per brand that we carry?"

Star Schemas

- Decision support systems often use a star schema to represent data
  - A *very* denormalized representation of data that is well suited to large-scale analytic processing
- One or more fact tables
  - Contain actual measures being analyzed and reported on
- Multiple dimension tables
  - Provide different ways to "slice" the data in the fact tables
- Fact tables have foreign-key references to the dimension tables

Example Star Schema

- Sales data-warehouse for a large retailer:



- Fact table is center of star
  - Dimension tables are referenced by the fact table

Dimensional Analysis

- This approach is called dimensional analysis
- Good example of denormalizing a schema to improve performance
  - Using a fully normalized schema will produce confusing and horrendously slow queries
- Decompose schema into a fact table and several dimension tables
  - Queries become very simple to write, or to generate
  - Database can execute these queries very quickly

Dimension Tables

- Dimension tables are used to select out specific rows from the fact table
  - Dimension tables should contain only attributes that we want to summarize over
  - Dimension tables can easily have many attributes
- Dimension tables are usually <u>very</u> denormalized
  - Specific values are repeated in many different rows
  - Only in 1NF
- Example: *sale_dates* dimension table
  - Year, quarter, month, day, and hour are stored as separate columns
  - Each row also has a unique ID column

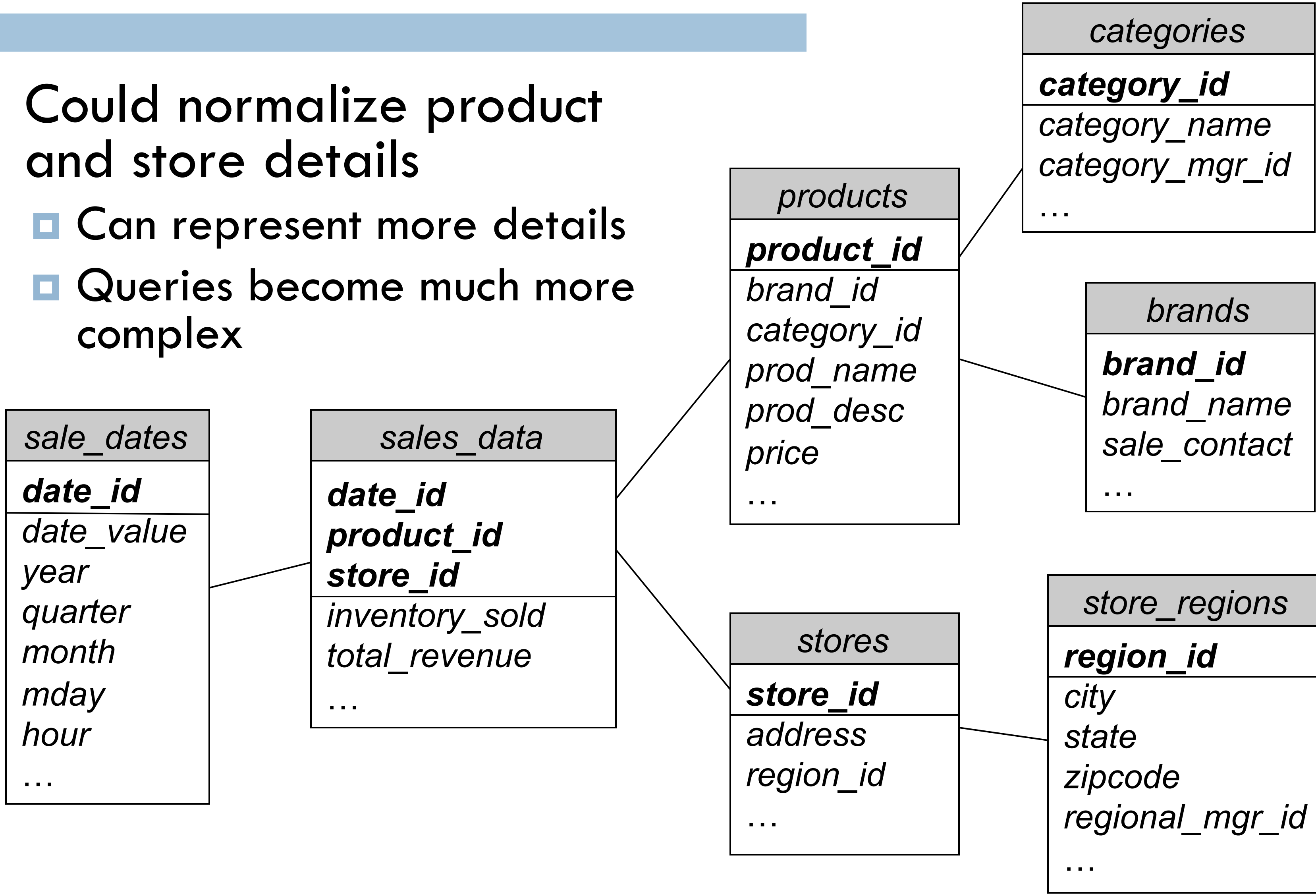
sale_dates
date_id
date_value *year* *quarter* *month* *mday* *hour* …

Dimension Tables (2)

- Dimension tables tend to be relatively small
  - At least, compared to the fact table!
  - Can be as small as a few dozen rows
  - All the way up to tens of thousands of rows, or more
    - Sometimes see dimension tables in 100Ks to millions of rows for very large data warehouses
- Sometimes need to normalize dimension tables
  - Eliminate redundancy to reduce size of dimension table
  - Increases complexity of query formulation and processing
  - Yields a <u>snowflake schema</u>
  - Star schemas *strongly* preferred over snowflake schemas, unless absolutely unavoidable!

Example Snowflake Schema

- Could normalize product and store details
  - Can represent more details
  - Queries become much more complex



Fact Tables

- Fact tables store aggregated values for the smallest required granularity of each dimension
  - Time dimension frequently drives this granularity
    - e.g. "daily measures" or "hourly measures"
- Fact tables tend to have fewer columns
  - Only contains the actual facts to be analyzed
  - Dimensional data is pushed into dimension tables
  - Each fact refers to its associated dimension values using foreign keys
  - All foreign keys in the fact table form its primary key
- Fact table contains the most rows, by far.
  - Well upwards of millions of rows (billions/trillions common)

Fact Tables (2)

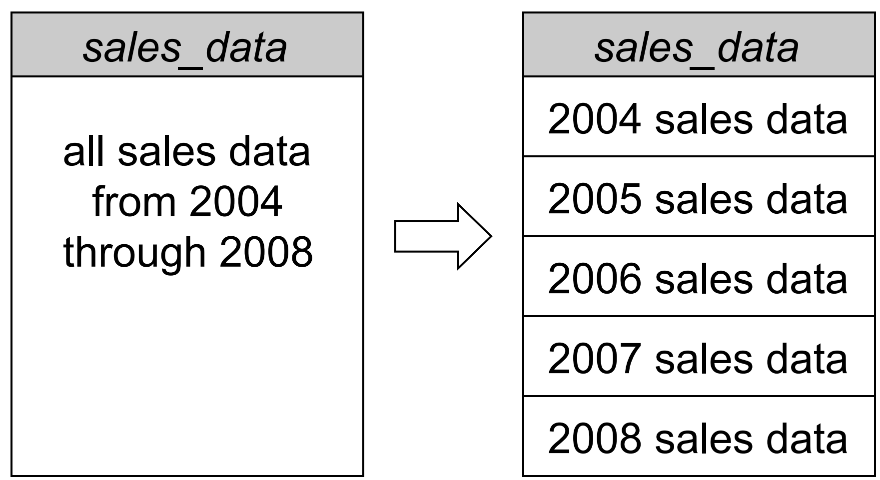
- Not uncommon to have multiple fact tables in a data warehouse
  - Facts relating to different aspects of the enterprise, where it doesn’t make sense to store in same table
  - Facts for a single aspect of the enterprise, but partitioned in different ways
    - Used in situations where combining into a single fact table would result in a huge, sparse fact table that is very slow to query
- Multiple fact tables frequently share dimension tables
  - e.g. date and/or time dimensions
  - May also have separate dimension tables only used by a particular fact-table

Analytic Queries

- Using a star schema, analytic queries follow a simple pattern
  - Query groups and filters rows from the fact table, using values in the dimension tables
  - Query performs simple aggregation of values contained within selected rows from fact table
- Queries contain only a few simple joins
  - Because dimension tables are (usually) small, joins can be performed very quickly
  - Fact table's primary key includes foreign keys to dimensions, so specific fact records can be located very quickly

Analytic Queries (2)

- Because only the fact tables are large, databases can provide optimized access
- Example: partitioned tables
  - Many databases can partition tables based on one or more attributes
  - Queries against the partitioned table are analyzed for which partitions are actually relevant to the query
- DSS schema design can partition the fact table to dramatically improve performance



Slowly-Changing Dimensions

- Frequently, data in dimension tables changes over time
  - e.g. a "user" dimension, where some user details change over time
    - e-mail address, rank/trust level within a community, last login time
- How do we represent slowly changing dimensions?
- Type 1 Slowly Changing Dimensions:
  - When a dimension value changes, overwrite the old values
  - Warehouse only maintains one row for each dimension value
  - Doesn't track any history of changes to dimension records
    - Can't analyze facts with respect to the change history!
    - e.g. "How do user behaviors change, with respect to how quickly their rank/trust level changes within their community?"

Slowly-Changing Dimensions (2)

- Type 2 Slowly Changing Dimensions:
  - Used to track change-history within a dimension
  - Rows in the dimension table are given additional attributes:
    - *start_date*, *end_date* – specifies the date/time interval when the values in this dimension record are valid
    - *version* – a count (e.g. starting from 0 or 1) indicating which version of the dimension record this row represents
    - *is_most_recent* – a flag indicating whether this is the most recent version of the dimension record
- Updating a dimension record is more complicated:
  - Find current version of the dimension record (if there is one)
  - Set the *end_date* to “now” to indicate the old row is finished
  - Create a new dimension record with a *start_date* of “now”
    - Fill in new dimension values; update *version*, *is_most_recent* values too

Good and Bad Measures

- Not all measures are suitable for star schemas!
- Fact table contains *partially* aggregated results
  - Analysis queries must complete aggregation, based on desired dimension and grouping aspects of query
- Example measures to track:
  - Quantities of each product sold
    - Easy to aggregate – just sum it up
  - Average per-customer sales totals
    - Fact table needs to store both the number of sales, and the total sale price, so that query can compute the average
  - Distinct customers over a particular time interval
    - Would need to store a list of actual customer IDs for each reporting interval! Much more complex.

Homework 7

- Includes a very simple data-warehouse exercise:
  - A simple OLAP database for analyzing web logs
    - Two months of access logs from NASA web server at Kennedy Space Center in Florida, from 1995
    - 3.6 million records, about 300MB storage size
    - Huge compared to what we have worked with so far!
    - Microscopic compared to most OLAP databases ☺
  - Create an OLAP database schema
    - Star schema diagram will be provided
  - Populate the schema from raw log data
  - Write some OLAP queries to do some simple analysis
- Please start this assignment early!
  - 100 students vs. 1 DB server… it could get messy… ☺

PASSWORDS
TREES AND HIERARCHIES

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 25

Account Password Management

- Mentioned a retailer with an online website…
- Need a database to store user account details
  - Username, password, other information
- How to store a user's password?
- What if the database application's security is compromised?
  - Can an attacker get a list of all user passwords?
  - Can the DB administrator be trusted?

- **Do we actually need to store the original password??**

A Naïve Approach

- A simple solution:
  - Store each password as plaintext

```
CREATE TABLE account (
    username VARCHAR(20) PRIMARY KEY,
    password VARCHAR(20) NOT NULL,
    ...
);
```

- Benefits:
  - If user forgets their password, we can email it to them
- Drawbacks:
  - Email is unencrypted – passwords can be acquired by eavesdropping
  - Users tend to use the same password for many different accounts
  - If database security is compromised, attacker gets <u>all</u> users' passwords
  - Of course, an unreliable administrator can also take advantage of this

Hashed Passwords

- A safer approach is to hash user passwords
  - Store hashed password, not the original
  - For authentication check:
    1. User enters password
    2. Database application hashes the password
    3. If hash matches value stored in DB, authentication succeeds
- Example using MD5 hash:

```
CREATE TABLE account (
    username VARCHAR(20) PRIMARY KEY,
    pw_hash  CHAR(32)    NOT NULL,
    ...
);
```

  - To store a password:

```
UPDATE account SET pw_hash = md5('new password')
    WHERE username = 'joebob';
```

Hashed Passwords (2)

- Want a cryptographically secure hash function:
  - Easy to compute a hash value from the input text
  - Even small changes in input text result in very large changes in the hash value
  - Hard to get a specific hash value by choosing input carefully
  - Should be collision resistant: hard to find two different messages that generate the same hash function
- MD5 is not collision resistant ☹
  - "[MD5] should be considered cryptographically broken and unsuitable for further use." – US-CERT
- SHA-1 was also discovered to not be very good
- Most people use SHA-2/3 hash algorithms now

Hashed Passwords (3)

- Benefits:
  - Passwords aren't stored in plaintext anymore
- Drawbacks:
  - Handling forgotten passwords is a bit trickier
    - Need alternate authentication mechanism for users
  - Isn't entirely secure! Still prone to dictionary attacks.
- Attacker computes a dictionary of common passwords, and each password's hash value
  - If attacker gets the hash values from the database, can crack some subset of accounts

Hashed, Salted Passwords

- Solution: <u>salt</u> passwords before hashing
- Example:

```
CREATE TABLE account (
    username VARCHAR(20) PRIMARY KEY,
    pw_hash CHAR(32) NOT NULL,
    pw_salt CHAR(6) NOT NULL,
    ...
);
```

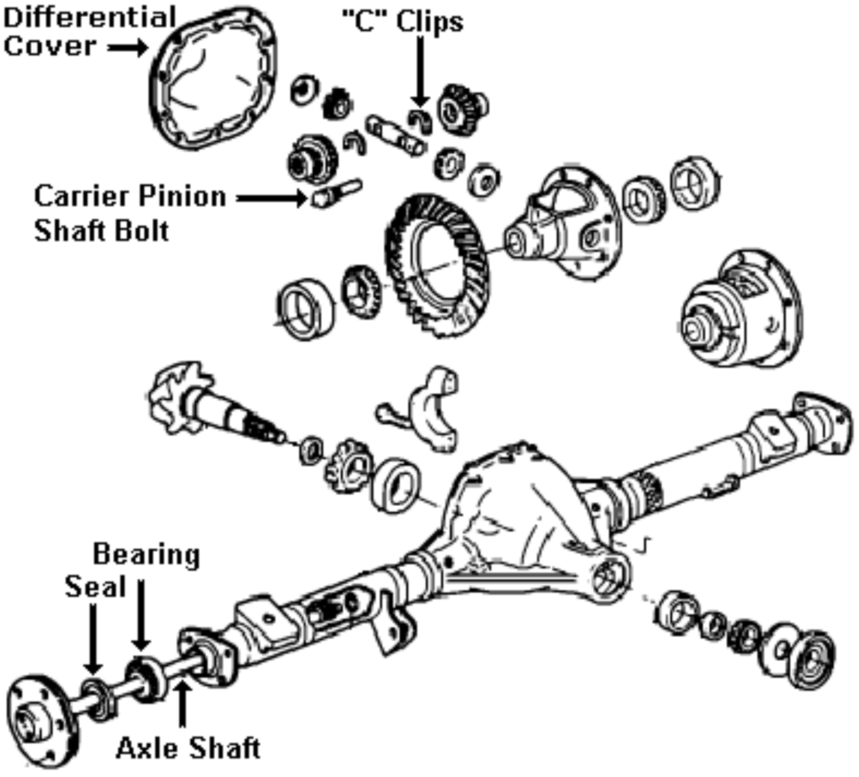
  - Each account is assigned a random salt value
    - Salt is always a specific length, e.g. 6 to 16 characters
  - Concatenate plaintext password with salt, before hashing
  - Attacker would have to compute a dictionary of hashes for each salt value… Prohibitively expensive!

Password Management

- Basically <u>no</u> reason to store passwords in plaintext!!
  - Users almost always use the same passwords in multiple places!
  - Only acceptable in the simplest circumstances
  - (You don't want to end up on the news because your system got hacked and millions of passwords leaked…)
- Almost always want to employ a secure password storage mechanism
  - Hashing is insufficient! Still need to protect against dictionary attacks by applying salt
  - Also need a good way to handle users that forget their passwords

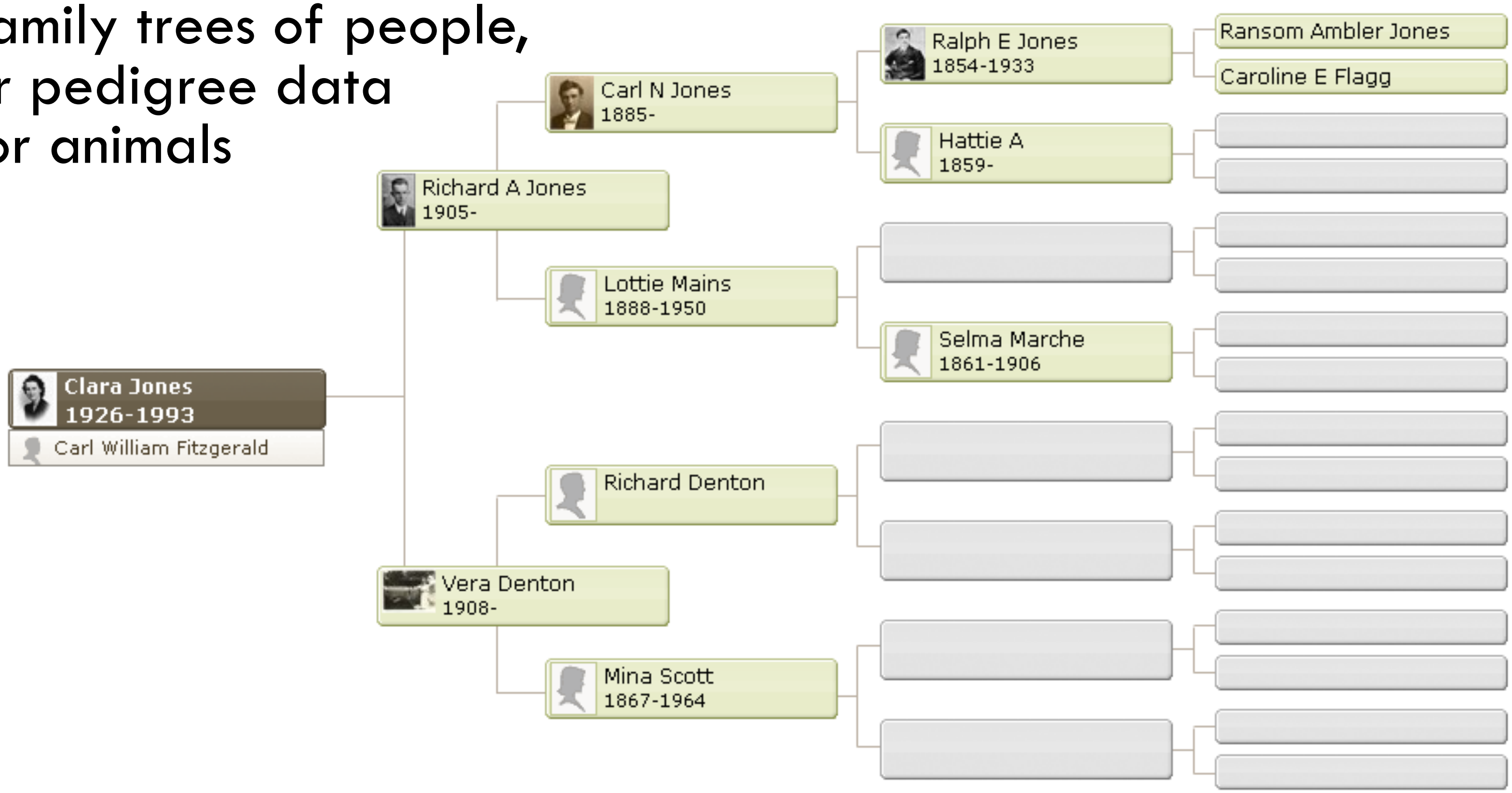
Trees and Hierarchies

- Many DB schemas need to represent trees or hierarchies of some sort
- Example: parts-explosion diagrams
  - "How are parts and subsystems assembled?"
  - "How much does a subsystem weigh?"
  - Other computations based on parts in subsystems



Trees and Hierarchies (2)

- Many kinds of relationships between people
  - Employee/manager relationships within an organization
  - Social graph data from a social network
  - Family trees of people, or pedigree data for animals



Trees and Hierarchies (3)

- Most common way of representing trees in the DB is an adjacency list model
  - Each node in the hierarchy specifies its parent node
  - Can represent arbitrary tree depths
- Example: employee database
  - *employee*(<u>*emp_name*</u>, *address*, *salary*)
  - *manages*(<u>*emp_name*</u>, *manager_name*)
  - Both attributes of *manages* are foreign keys referencing *employee* relation

Trees and Hierarchies (4)

- Adjacency list model is only one of <u>several</u> ways to represent trees and hierarchies
- Different approaches have different strengths and weaknesses
- Some approaches to consider:
  - Adjacency list models
  - Nested set models
  - Path enumeration models

General Design Questions

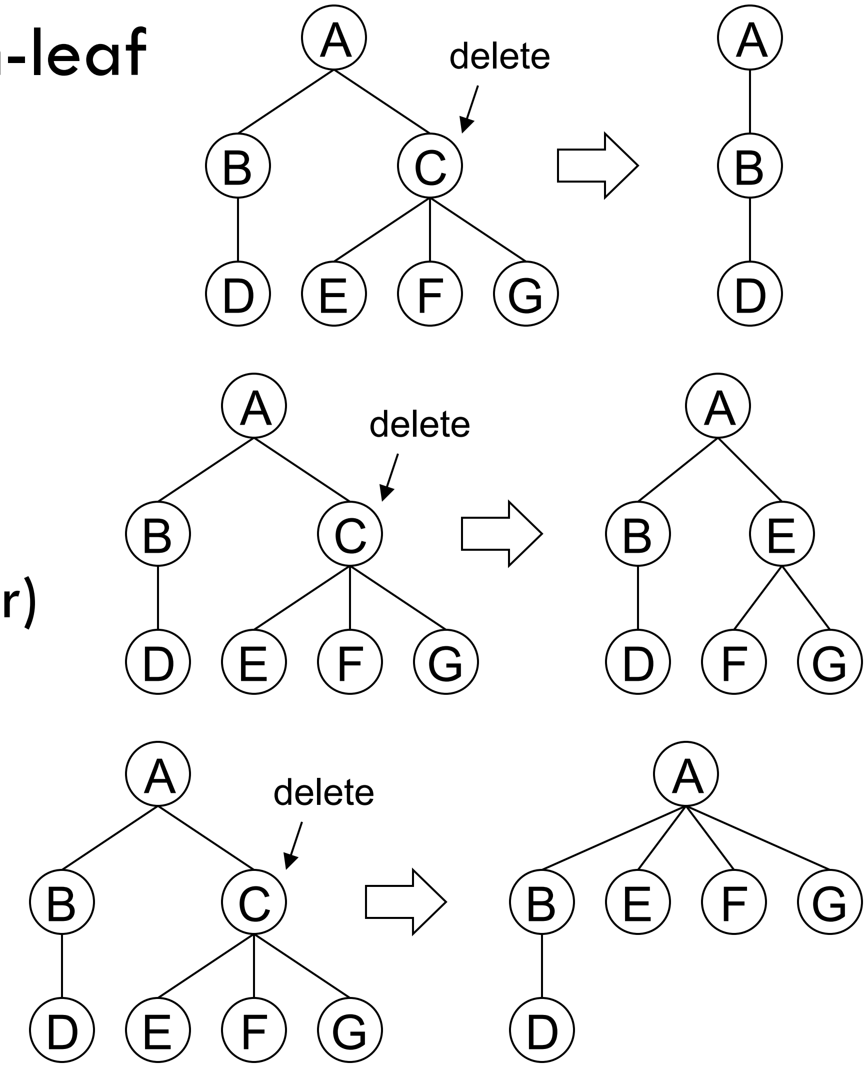
- How hard is it to access the tree?
  - Retrieve a specific node
  - Find the parent/children/siblings of a particular node
  - Retrieve all leaf nodes in the tree
  - Retrieve all nodes at a particular level in the tree
  - Retrieve a node and its entire subtree
- Also, path-based queries:
  - Retrieve a node corresponding to a particular path from the root
  - Retrieve nodes matching a path containing wildcards
  - Is a particular path in the hierarchy?

General Design Questions (2)

- How hard is it to modify the tree?
  - Add a single node
  - Delete a leaf node
  - Delete a non-leaf node
    - What to do with subtree of deleted node?
  - Move a subtree within the tree
- How to enforce constraints in the schema?
  - Enforce only one root
  - Disallow cycles in a tree
    - Simplest example: "my parent can't be myself"
  - Disallow multiple parents (tree vs. directed acyclic graph)
  - Enforce a maximum child-count, maximum depth, etc.

Deleting Nodes

- What happens when a non-leaf node is deleted?
- Option 1:
  - Delete entire subtree as well
- Option 2:
  - Promote one child node into removed node's position
  - (Or, replace with placeholder)
- Option 3:
  - Make all child nodes into children of deleted node's parent



Adjacency List Model

- Very common approach for modeling trees
- Each node specifies its parent node
- Relationship between nodes are frequently stored in a separate table
  - e.g. *employee* and *manages*
  - Can represent multiple trees without *null* values
  - Can have employees that are not part of the hierarchy

Adjacency List Model (2)

- Strengths:
  - A very flexible model for representing and manipulating trees
  - Easy to add a new node anywhere in the tree
  - Easy to move a whole subtree
- Weaknesses:
  - Deleting a node often requires extra steps (to relocate children)
  - Operations on entire subtrees are expensive
  - Operations applied to a particular level across a tree are expensive
  - Looking for a node at a specific path is expensive
  - (All of these operations require multiple queries to identify the nodes affected by the operation.)

Queries on Subtrees

- Example query using subtrees:
  - For every manager reporting to a given Vice President, find the total number of employees under that manager, and their total salaries
  - Computing an aggregate over individual subtrees
- Another example:
  - Database containing parts in a mechanical assembly
    - Parts combined into sub-assemblies
    - Sub-assemblies and parts combined into larger assemblies
    - Top level assembly is the entire system
  - Find number of parts, the total cost, and the total weight of each sub-assembly in the system

Finding Nodes in Subtrees

- To find all nodes in a specific subtree, must iterate a query using a temporary table
  - Example: Find all employees under manager "Jones"

```
CREATE TEMPORARY TABLE emps
  ( emp_name VARCHAR(20) NOT NULL );

INSERT INTO emps VALUES ('Jones');
INSERT INTO emps SELECT emp_name FROM manages
  WHERE manager_name IN (SELECT * FROM emps)
    AND emp_name NOT IN (SELECT * FROM emps);
```

  - Iterate last statement until no new rows added to temp table
  - Databases often report a count of how many rows are inserted/modified/deleted by each DML operation

Finding Nodes in Subtrees (2)

- Can also store a "depth" value in the result

```
CREATE TEMPORARY TABLE emps (
  emp_name VARCHAR(20) NOT NULL,
  depth    INTEGER     NOT NULL
);

INSERT INTO emps VALUES ('Jones', 1);

-- Some variable i, where i = 1 initially:
INSERT INTO emps SELECT emp_name, i + 1 FROM manages
  WHERE manager_name IN
        (SELECT * FROM emps WHERE depth = i);
```

  - Each level of the hierarchy has the same depth value
  - Slightly more efficient than the previous version
    - Each iteration has fewer rows to consider

Finding Nodes in Subtrees (3)

- Best to implement this as a stored procedure
  - Don't involve command/response round-trips with application code, if possible!
  - Perform processing entirely within the DB for best performance
- Still acceptable if application has to drive the iteration process
  - Most DB connectivity APIs let you create temporary tables, get number of rows changed, etc.

Other Adjacency List Notes

- Must manually order siblings under a node
  - Add another column to the table for ordering siblings
- Adjacency list model is also good for representing graphs
  - Actually easier than using for trees, because less constraints are required
  - Traversing the graph requires temporary tables and iterative stored procedures
  - (Other representations we will discuss today aren't well-suited for graphs at all!)

Aside: Recursive Queries

- Briefly looked at this Datalog query last time:
  - Manager relation:

    manager(*employee_name*, *manager_name*)
  - Define view relation "empl" for all employees directly or indirectly managed by a particular manager.

    empl(X, Y) :- manager(X, Y)

    empl(X, Y) :- manager(X, Z), empl(Z, Y)
  - To find all employees managed by Jones:

    ? empl(X, "Jones")
  - Datalog iterates on the rule definitions until the result doesn't change (fixpoint)
- Datalog supports traversing hierarchies very easily

Recursive Queries in SQL

- For a long time, SQL did not support recursive queries
  - Only way to traverse adjacency-list data model was to use a temporary table and repeated queries, as described
- Now, most databases also support some form of recursive SQL query
  - e.g. PostgreSQL 8.4+, SQLServer, Oracle (not MySQL ☹)
  - If available, makes it much easier to traverse adjacency-list datasets
- Still requires repeated queries against the database…
  - Even though query is easier to write, it's still slow to execute!

Recursive Queries in PostgreSQL

- Example using SQL99/Postgres 8.4 syntax
- Find all employees directly or indirectly managed by Jones:

Initial Subquery

Recursive Subquery

```
WITH RECURSIVE empl AS (
       SELECT employee_name, manager_name
       FROM manager
     UNION ALL
       SELECT e.employee_name, m.manager_name
       FROM empl AS e JOIN manager AS m
         ON e.manager_name = m.employee_name)
SELECT * FROM empl
WHERE manager_name = 'Jones';
```

Recursive Queries in PostgreSQL (2)

- Can compute the depth in the hierarchy, too:

```
WITH RECURSIVE empl AS (
       SELECT employee_name, manager_name,
              1 as depth
       FROM manager
     UNION ALL
       SELECT e.employee_name, m.manager_name,
              e.depth + 1 AS depth
       FROM empl AS e JOIN manager AS m
         ON e.manager_name = m.employee_name)
SELECT * FROM empl
WHERE manager_name = 'Jones';
```

SQLServer Recursive Queries

- Microsoft SQLServer 2005/2008 also uses **`WITH`** clause for recursive queries
  - Similar approach, using a non-recursive subquery and a recursive subquery, combined with **`UNION [ALL]`**
  - Doesn't use a **`RECURSIVE`** modifier
- Neither Postgres nor SQLServer recursive queries can handle cycles in the data…
  - Introducing a cycle into the data causes the query to infinite-loop

Oracle Recursive Queries

- Oracle has <u>much</u> more sophisticated recursive query support
- A simple example:

```
SELECT employee_name, manager_name, LEVEL
FROM manager
CONNECT BY PRIOR employee_name = manager_name;
```

  - **`PRIOR`** modifier specifies parent in parent/child relationship
  - **`LEVEL`** is a pseudocolumn specifying level in the hierarchy
- Can also do *many* other things, such as:
  - Specify the initial data-set to iterate on
  - Specify the order of siblings in each level of hierarchy
  - Detect and report which nodes are leaves in the hierarchy
  - Detect and report cycles in the dataset
  - Generate the full path from root to each node in the result

Final Notes on Recursive SQL Queries

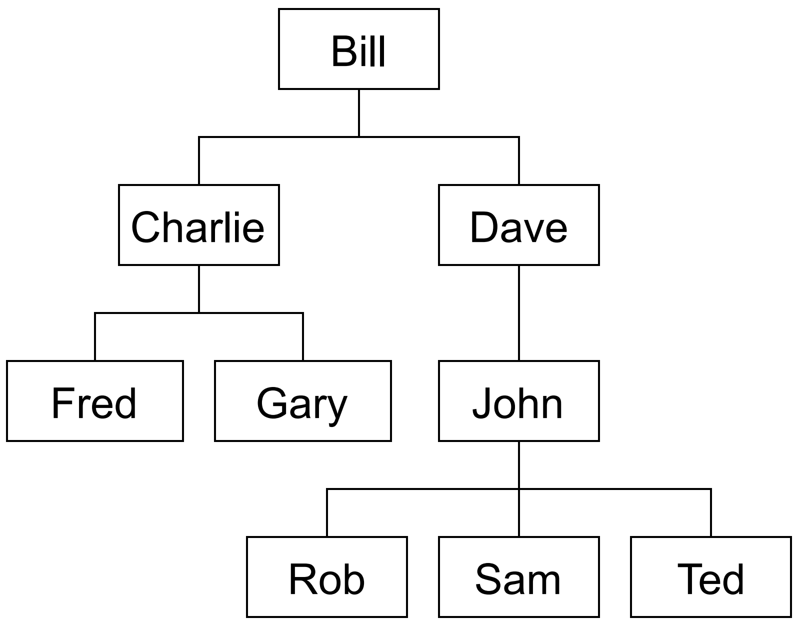
- As stated earlier:
  - Recursive SQL queries make it much easier to <u>write</u> the query that traverses adjacency-list data…
  - Still requires the DB engine to repeatedly issue queries until the entire hierarchy is traversed!
- If an application needs to store hierarchical or graph data, and must select entire subtrees <u>quickly</u>:
  - Adjacency-list model is *the most expensive* model to use!

Nested Set Model

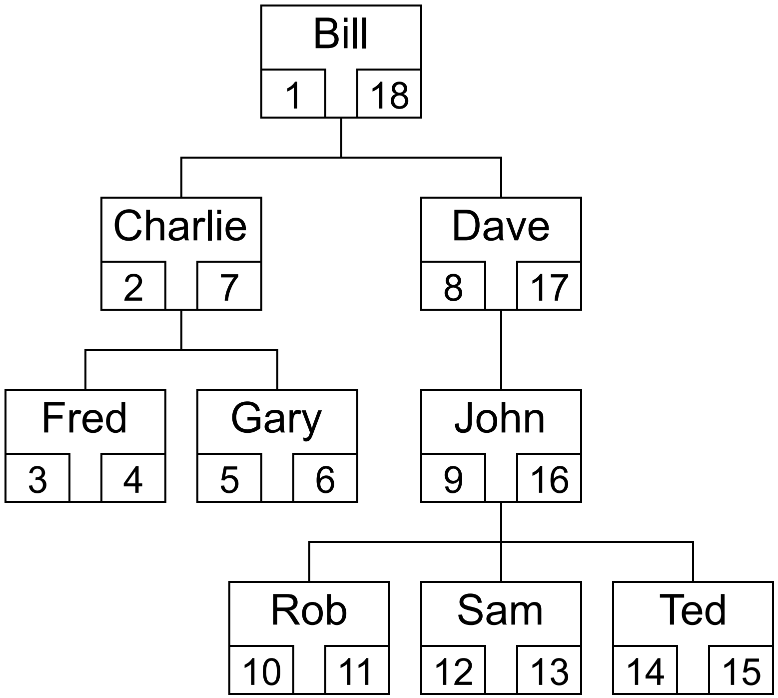
- A model optimized for selecting subtrees
- Represents hierarchy using containment
  - A node "contains" its children
- Give each node a "low" and "high" value
  - Specifies a range
  - Always: low < high
- Use this to represent trees:
  - A parent node's [low, high] range contains the ranges of all child nodes
  - Sibling nodes have non-overlapping ranges

Example using Nested Sets

- Manager hierarchy:

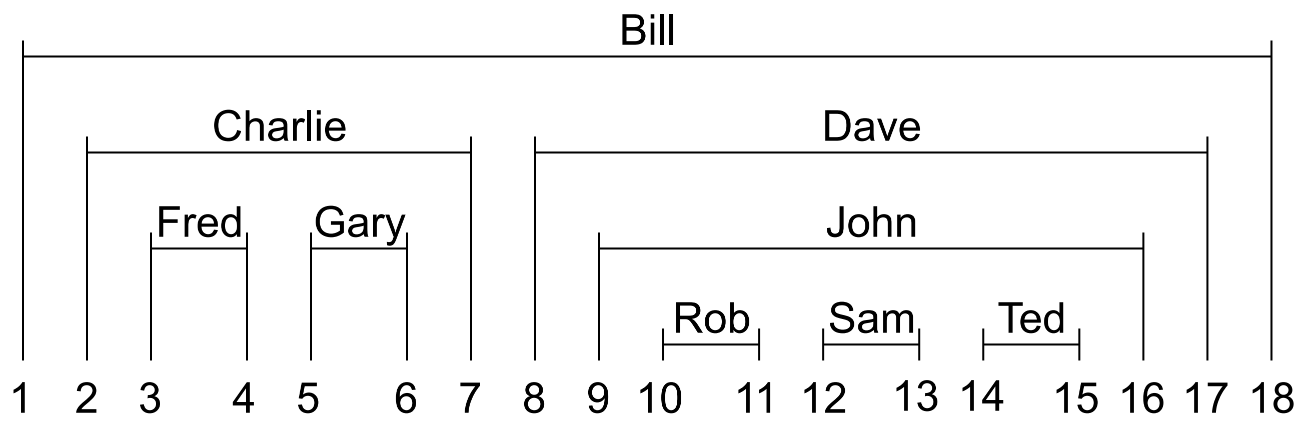


- Nested set hierarchy:



Selecting Subtrees

- Each parent node contains its children:



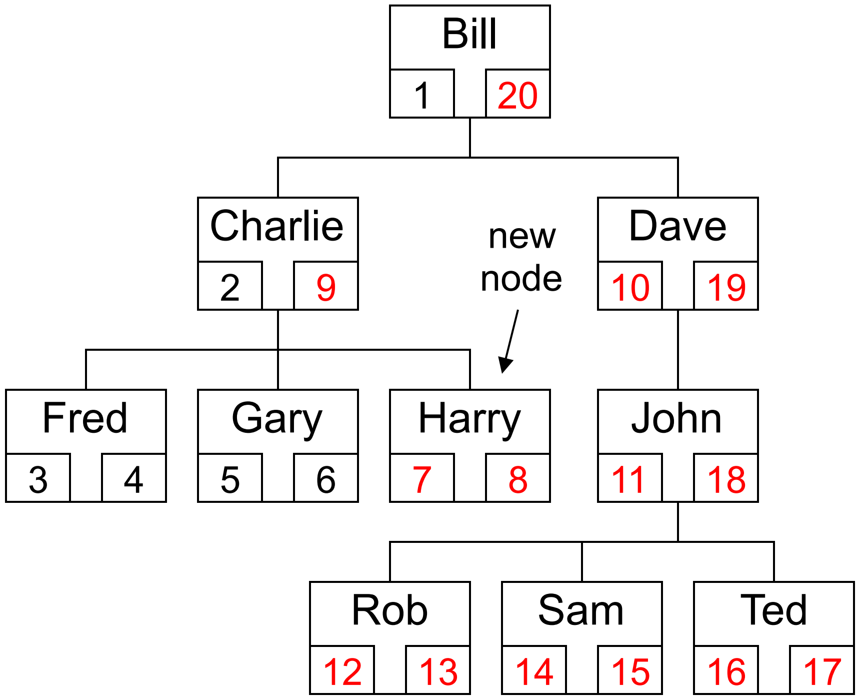
- Easy to select an entire subtree
  - Select all nodes with low (or high) value within node's range
- Can also select all leaf nodes [relatively] easily
  - If all range values separated by 1, find nodes with low + 1 = high
  - For arbitrary range sizes, find nodes that contain no other node's range values (requires self-join)

Nested Set Model

- Strengths:
  - Very easy to select a whole subtree
    - Very important for generating reports against subtrees
  - Tracking the order of siblings is built in
- Weaknesses:
  - Some selections are more expensive
    - e.g. finding all leaf nodes in tree requires self-join
  - Constraints on range values are expensive
    - **`CHECK`** constraint to ensure low < high is cheap…
    - **`CHECK`** constraint to verify other interval characteristics is expensive!
  - Pretty costly to insert nodes or restructure trees
    - Have to update node bounds properly

Adding Nodes In Nested Set Model

- If adding a node:
  - Must choose range values for new node correctly
  - Often need to update range values of many other nodes, even for simple updates
- Example:
  - Add new employee Harry under Charlie
  - Must update ranges of most nodes in the tree!



Supporting Faster Additions

- Can separate range values by larger amounts
  - e.g. spacing of 100 instead of 1
  - Situations requiring range-adjustments of many nodes will be far less frequent
- Can implement tree-manipulation operations as stored procedures
  - "Add a node," "move a subtree," etc.

Path Enumeration Models

- For each node in hierarchy:
  - Represent node's position in hierarchy as the path from the root to that node
  - Entire path is stored as a single string value
- Node enumeration:
  - Each node has some unique ID or name
  - A path contains the IDs of all nodes between root and a particular node
  - If ID values are fixed size, don't need a delimiter
  - If ID values are variable size, choose a delimiter that won't appear in node IDs or names

Path Enumeration Model (2)

- Path enumeration model is fastest when nodes are retrieved using full path
  - "Is a specified node in the hierarchy?"
  - "What are the details for a specified node?"
  - Adjacency list model and nested set model simply can't answer these queries quickly!
- Example:
  - A database-backed directory service (e.g. LDAP or Microsoft Active Directory)
  - Objects and properties form a hierarchy
  - Properties are accessed using full path names
    - "sales.printers.colorprint550.queue"

Strengths and Weaknesses

- Optimized for read performance
  - Retrieving a specific node using its path is *very* fast
  - Retrieving an entire subtree is also pretty fast
    - Requires text pattern-matching, but matching a prefix is fast, especially with a suitable index on the string
    - Example: Find all sales print servers
      - Use a condition: `path LIKE 'sales.printers.%'`
- Adding leaf nodes is fast
- Restructuring a tree can be very slow
  - Have to reconstruct many paths…
- Operations rely on string concatenation and string manipulation

Edge Enumeration

- Paths can enumerate edges instead of nodes
  - Each level of path specifies index of node to select
- Primary method used in books
  - Example:

Edge Enumeration	Section Name
3	SQL
3.1	Background
3.2	Data Definition
3.2.1	Basic Domain Types
3.2.2	Basic Schema Definition in SQL
…	…

- Like node enumeration, requires string manipulation for most operations

Summary: Trees and Hierarchies

- Can represent trees and hierarchies in several different ways
  - Adjacency list model
  - Nested set model
  - Path enumeration model
- Each approach has different strengths
  - Each is optimized for different kinds of usage
- When designing schemas that require hierarchy:
  - Consider functional and non-functional requirements
  - Choose a model that best suits the problem

DATABASE TRANSACTIONS

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 26

Database Transactions

- Many situations where a sequence of database operations must be treated as a single unit
  - A combination of reads and writes that must be performed "as a unit"
  - If any operation doesn't succeed, <u>all</u> operations in the unit of work should be rolled back
- An <u>essential</u> database feature for the correct implementation of tasks that could fail
- Also essential for databases with concurrent access!
  - An operation consisting of multiple reads may also need to see a single, consistent set of values
  - Reads should also be performed "as a unit"

Database Transactions (2)

- DBs provide <u>transactions</u> to demarcate units of work
- Issue **`BEGIN`** at start of unit of work
  - Can also say **`START TRANSACTION`**, etc.
- Issue one or more SQL DML statements within the txn
  - Database may or may not support DDL inside transactions
- When finished, issue a **`COMMIT`** command
  - Signals that the transaction is completed
- If transaction must be aborted, issue **`ROLLBACK`**
  - All changes made by the transaction are discarded
- If an error occurs along the way, DB will automatically roll back the transaction
  - An error could also occur at commit-time, causing rollback

Transaction Properties

- A transaction system should satisfy specific properties
  - Called the ACID properties
  - Specified by Jim Gray, who received a Turing Award for this work
- Atomicity
  - Either *all* the operations within the transaction are reflected properly in the database, or *none* are.
- Consistency
  - When a transaction completes, the database must be in a consistent state; i.e. all constraints *must* hold.

Transaction Properties (2)

- Isolation
  - When multiple transactions execute concurrently, they must appear to execute one after the other, in isolation of each other.
- Durability
  - After a transaction commits, all changes should persist, even when a system failure occurs.

Bank Account Example

- Transfer $400 from account A-201 to A-305
  - Clearly requires multiple steps
- If transaction isn't atomic:
  - Perhaps only one account shows the change!
- If transaction isn't consistent:
  - Perhaps a balance goes below zero, or the sum of the balances doesn't remain constant
- If transaction isn't isolated:
  - Other concurrent operations involving either account could result in inaccurate balances
- If transaction isn't durable:
  - If transaction commits and then the server crashes, database might not contain all (or any) of the transaction's changes!

Transaction Properties And You

- As a <u>user</u> of the database:
  - How atomicity, consistency, and durability are implemented is largely irrelevant
    - (Although, they are *very* cool subjects!)
  - Important point is <u>whether</u> they're provided
    - … and how completely they are provided!
- Isolation is another matter entirely
  - Turns out to affect implementation of database applications quite extensively
  - Database users are provided several different choices for how to handle transaction isolation

Transaction Isolation

- If database only has one client connection at a time, isolation is irrelevant
  - The client can only issue one transaction at a time
  - Two transactions can <u>never</u> be concurrent
- Most DB applications support *many* concurrent users
  - Concurrent transactions happen *all the time!*
- Without proper transaction isolation, the database would quickly become corrupt
  - Would frequently generate spurious results
- Five kinds of spurious results can occur in SQL, without proper transaction isolation

Concurrent Transaction Issues

- Dirty writes:
  - A transaction T_1 writes a value to X
  - Another txn T_2 also writes a value to X before T_1 commits or aborts
  - If T_1 or T_2 aborts, what should be the value of X?
- Dirty reads:
  - A transaction T_1 writes a value to X
  - T_2 reads X before T_1 commits
  - If T_1 aborts, T_2 has an invalid value for X
- Nonrepeatable reads:
  - T_1 reads X
  - T_2 writes to X, or deletes X, then commits
  - If T_1 reads X again, value is now different or gone

Concurrent Transaction Issues (2)

- Phantoms
  - Transaction T_1 reads rows that satisfy a predicate P
  - Transaction T_2 then writes rows, some of which satisfy P
  - If T_1 repeats its read, it gets a different set of results
  - If T_1 writes values based on original read, new rows aren't considered!
- Lost updates
  - Transaction T_1 reads the value of X
  - Transaction T_2 writes a new value to X
  - T_1 writes to X based on its previously read value
- **How can a database avoid these kinds of issues?**
  - A simple answer: <u>serialize</u> all transactions
  - No two transactions can overlap, *ever*.
  - A *very* slow approach, but it certainly works…

Serialized Transactions

- Serializing all transactions is prohibitively slow
- Definite benefits for allowing concurrent transactions:
  - Different transactions may use completely separate resources, and would run very efficiently in parallel
  - Long, slow transactions shouldn't hold up short, fast transactions that read the same resources
- Databases can execute transactions in a way that *appears* to be serialized

  Isolation: "When multiple transactions execute concurrently, they must appear to execute one after the other, in isolation of each other."
  - Transactions are sequences of read and write operations
  - Schedule these sequences of operations in a way that maintains serializability constraints
  - *(And if we can't successfully do this? Hmmm…)*

Transaction Isolation Constraints

- Serializable transaction constraint is one kind of isolation constraint
  - A very strict one, for critical operations
- Not all database applications require such strict constraints
  - Application may work fine with looser isolation constraints
  - Application might not achieve required throughput with serializable transactions
- SQL defines four transaction isolation levels for use in applications
  - Can set transactions to have a specific isolation level

Transaction Isolation Levels

- Serializable
  - Concurrent transactions produce the same result as if they were run in some serial order
  - NOTE: The serial order may not necessarily correspond to the exact order that transactions were issued
- Repeatable reads
  - During a transaction, multiple reads of X produce the same results, regardless of committed writes to X in other transactions
  - Other transactions' committed changes do not become visible in the middle of a transaction

Transaction Isolation Levels (2)

- Read committed
  - During a transaction, other transactions' committed changes become visible immediately
  - Value of X can change during a transaction, if other transactions write to X and then commit
- Read uncommitted
  - Uncommitted changes to X in other transactions become visible immediately

Transaction Isolation Levels (3)

- Back to the undesirable transaction phenomena:
  - What does each isolation level allow?

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
serializable	NO	NO	NO
repeatable reads	NO	NO	YES
read committed	NO	YES	YES
read uncommitted	YES	YES	YES

- To specify the transaction isolation level:

```
SET TRANSACTION ISOLATION LEVEL
  { SERIALIZABLE | REPEATABLE READ |
   READ COMMITTED | READ UNCOMMITTED }
```

  - Different databases support different isolation levels!

Databases and Isolation Levels

- Many databases implement isolation levels with locks
- At simplest level, locks are:
  - <u>Shared</u>, for read locks
  - <u>Exclusive</u>, for write locks
- Locks may have different levels of granularity
  - Row-level locks, page-level locks, table-level locks
  - Finer-grain locks allow more transaction concurrency, but demand greater system resources
    - A space overhead for representing locks at desired granularity
    - A time overhead for analyzing and manipulating the set of locks
  - Databases often provide multiple levels of granularity
    - Example: table-level locks and page-level locks
    - Becoming increasingly common for databases to provide row-level locks

Database Locks

- Rules for locking are carefully defined
  - What locks, or sequences of locks, satisfy the necessary isolation constraints?
  - Can a lock be upgraded from shared to exclusive? If so, when?
  - *(Take CS122 if you want to learn more!)*
- In general:
  - **`SELECT`** operations require shared locks
  - **`INSERT`**, **`UPDATE`**, **`DELETE`** require exclusive locks
  - In practice, implementation gets *much* more complicated, to prevent "phantom rows" phenomenon, etc.
  - Databases vary in locking implementations and behaviors! (Read your manual…)

Locking Issues

- Some transactions are incompatible with others
  - Each transaction requires some series of locks…
  - Can easily lead to deadlock between transactions
- This can't be avoided, because:
  - Database can't predict what sequence of SQL commands will be issued in each transaction!
  - Database can't predict where the needed rows will appear!
    - (e.g. in the same page that another transaction is writing to?)
- Solution:
  - Database lock managers are designed to detect deadlocks
  - If several transactions become deadlocked, one is aborted

Locking Issues (2)

- If a database application performs long or complex operations in a transaction:
  - It must be designed to handle situations where a transaction is aborted due to deadlock!
  - Solution is simple: just retry the operation
- Guidelines:
  - Keep transactions as short and simple as possible
  - If transactions are aborted frequently due to deadlock, the application needs to be reworked
  - Databases can usually report what commands caused the deadlock
  - Expect that deadlocks may still infrequently occur!

Concurrent Reads and Writes

- Example: two transactions using *account* table
  - Repeatable-read or read-committed isolation level
  - Only one copy of each tuple is kept for the *account* table
  - T_1 reads balance of account A-444, gets $850
  - T_2 reads balance of account A-444, gets $850
  - T_1 writes (balance + $300) back to balance of A-444
  - T_2 reads balance of A-444 again… ???
- If database stores each row in only one place, then T_2 must block until T_1 commits or aborts
  - (For all isolation levels besides read-uncommitted.)
- For repeatable-read/serializable isolation: If T_1 commits…
  - …then T_2 must be aborted!

Readers and Writers

- For certain database storage implementations and isolation levels, writers block readers
- Solution:
  - Keep multiple versions of each row in the database
  - If a writer updates a value:
    - *A new version of the entire row* is added to the database
    - Reader can continue with old version of value as long as the isolation level will allow it
  - In most cases, writers won't block readers anymore
  - Writers will only block other writers to the same row

Multiversion Concurrency Control

- Called multiversion concurrency control (MVCC)
  - Each row has some "version" indicator
    - Either a timestamp or a transaction ID
  - Transactions can see a specific range of versions
    - Depends on the isolation level, and the operations that the transaction is performing
  - If a transaction needs to read a row that another transaction has written, the reader can still proceed
    - (for read-committed and many repeatable-read scenarios)
- Yields <u>dramatic</u> performance improvements for concurrent transaction processing!
- Can also make transaction isolation much more confusing
  - Transactions proceed that would have blocked without MVCC

Read-Uncommitted Example

Operation: Deposit $100 into account A-333

T_1: `BEGIN;`

T_1: `SELECT balance FROM account WHERE account_number = 'A-333';`

```
+----------+
|  850.00  |
+----------+
```

T_2: `BEGIN;`

T_2: `SELECT balance FROM account WHERE account_number = 'A-333';`

```
+----------+
|  850.00  |
+----------+
```

T_1: `UPDATE account SET balance = balance + 100 WHERE account_number = 'A-333';`

T_2: `SELECT balance FROM account WHERE account_number = 'A-333';`

```
+----------+
|  950.00  |
+----------+
```

T_1: `ROLLBACK;`

T_2: `SELECT balance FROM account WHERE account_number = 'A-333';`

```
+----------+
|  850.00  |
+----------+
```

Read-Committed Example

Operation: Deposit $100 into account A-333

T_1: `BEGIN;`

T_1: `SELECT balance FROM account WHERE account_number = 'A-333';`

```
+---------+
|  850.00 |
+---------+
```

T_2: `BEGIN;`

T_2: `SELECT balance FROM account WHERE account_number = 'A-333';`

```
+---------+
|  850.00 |
+---------+
```

T_1: `UPDATE account SET balance = balance + 100 WHERE account_number = 'A-333';`

T_2: `SELECT balance FROM account WHERE account_number = 'A-333';`

```
+---------+
|  850.00 |
+---------+
```

T_1: `COMMIT;`

T_2: `SELECT balance FROM account WHERE account_number = 'A-333';`

```
+---------+
|  950.00 |
+---------+
```

Repeatable-Read Example

Operation: Deposit $100 into account A-333

T_1: **BEGIN;**

T_1: **SELECT balance FROM account**
    WHERE account_number = 'A-333';

```
+---------+
|  850.00 |
+---------+
```

T_2: **BEGIN;**

T_2: **SELECT balance FROM account**
    WHERE account_number = 'A-333';

```
+---------+
|  850.00 |
+---------+
```

T_1: **UPDATE account SET balance = balance + 100**
    WHERE account_number = 'A-333';

T_2: **SELECT balance FROM account**
    WHERE account_number = 'A-333';

```
+---------+
|  850.00 |
+---------+
```

T_1: **COMMIT;**

T_2: **SELECT balance FROM account**
    WHERE account_number = 'A-333';

```
+---------+
|  850.00 |
+---------+
```

Serializable Example

Operation: Deposit $100 into account A-333

```
T1: BEGIN;
T1: SELECT balance FROM account                +----------+
        WHERE account_number = 'A-333';        |  850.00  |
                                               +----------+
T2: BEGIN;
T2: SELECT balance FROM account                +----------+
        WHERE account_number = 'A-333';        |  850.00  |
                                               +----------+
T1: UPDATE account SET balance = balance + 100
        WHERE account_number = 'A-333';
```

- T_1 blocks on the update, because T_1 and T_2 must be completely isolated from each other!
- T_1 *must* wait for completion of T_2, since T_2 read balance of A-333 before T_1 tried to update it.

Read Issues

- Another simple example:
  - Bank account A-201 jointly owned by customers A and B, with balance of $900
  - Customer A requests a loan of $800 at the bank
    - This bank's policy is that the loan amount must be less than the current account balance.
  - At same time, Customer B withdraws $200 from the same account
- Customer A's transaction needs the latest value
  - But, value read from DB immediately goes out of date
  - Serializable transaction would prevent this, but read-committed and repeatable-read transactions allow it

Read Issues (2)

T_1: Customer A wants a loan of $800
- Customer A owns account A-201, balance $900
- Loan amount must be less than account balance

T_2: Customer B tries to withdraw $200 from A-201
- Customer B also owns account A-201

- T_1 reads account balance of $900
- T_2 subtracts $200 from account balance
- T_1 creates a new loan of amount $800
  - Bad assumption: Old value of $900 is still valid!
- Database is no longer in a consistent state
  - Bank's business rule is violated

Read-Only Values

- Transaction T_1 needs latest value, <u>and</u> it must not be allowed to change until T_1 is finished!
- **`SELECT ... LOCK IN SHARE MODE`** allows a transaction to mark selected values as read-only
  - Constraint is enforced until end of transaction
- Transaction T_1:

```
SELECT balance FROM account
  WHERE account_number = 'A-201'
  LOCK IN SHARE MODE;
```

  - T_2 cannot change balance of A-201 until T_1 is finished

Read/Write Issues

- Two banking transactions:
  - T_1 wants to withdraw all money in account A-102
  - T_2 wants to withdraw \$50 from A-102
  - T_1 needs to read current balance, before it can update
  - T_2 can simply update
- T_1 reads balance of A-102
- T_2 subtracts \$50 from A-102
- T_1 subtracts \$400 from A-102
  - Overdraft! T_1 must roll back.
- Again, prohibited by serializable transactions
  - Allowed by read-committed or repeatable-read levels

```
+----------+
|  400.00  |
+----------+
+----------+
|  350.00  |
+----------+
+----------+
|  -50.00  |
+----------+
```

Intention to Update

- Transaction T_1 must read before its update
  - …but a read lock is insufficient for T_1's needs
  - T_1 must state intention to update row, when it reads it
  - Otherwise, T_1 will be overruled frequently
- **SELECT … FOR UPDATE** command allows a transaction to state an intention to update

```
SELECT balance FROM account
  WHERE account_number = 'A-102'
  FOR UPDATE;
```

  - T_2 can't update A-102 until T_1 is finished

Serializable Transactions?

- Serializable transactions prevent a lot of issues
  - Serializable transactions are consistent
- Other isolation levels can cause some problems
  - Considered to be weak levels of consistency
- Why not serializable transactions for everything?
  - Serializable transactions are very slow for large database applications
  - Simply not scalable
  - Only certain operations run into trouble with other isolation levels
  - Can use features like `FOR UPDATE` as workarounds for these issues

Savepoints

- Transactions may involve a long sequence of steps
  - If one step fails, don't roll back entire transaction
  - Instead, roll back to last "good" point and try something else
- Some databases provide <u>savepoints</u>
  - Mark a savepoint in a transaction when it completes some tasks
  - Can roll back to savepoint, and continue transaction from there
- To mark a savepoint:

  `SAVEPOINT` ***`name;`***
  - Roll back to that savepoint:

    `ROLLBACK TO SAVEPOINT` ***`name;`***
  - Can release a savepoint when it becomes unnecessary:

    `RELEASE SAVEPOINT name;`
  - Commit and rollback commands work on whole-transaction level

Review

- Transaction processing is a very rich topic
  - Many powerful tools for applications to use
  - Optimizations that allow for faster throughput
- Subtle issues can arise with transactions!
  - Applications should expect that transactions might be aborted by the database
  - Sometimes operations require statements like **`SELECT … FOR UPDATE`** to work correctly
- Always read your database manual! ☺
  - What isolation levels are supported? Any variances?
  - Are **`FOR UPDATE`** / **`LOCK IN SHARE MODE`** supported?
  - Are savepoints supported?

FINAL EXAM REVIEW

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 27

Final Exam Overview

- 6 hours, multiple sittings
  - Open book, notes, MySQL database, etc. (the usual)
- Primary topics: everything in the last half of the term
  - DB schema design and Entity-Relationship Model
  - Functional/multivalued dependencies, normal forms
  - Also some SQL DDL, DML, stored routines, etc.
- Questions will generally take this form:
  - "Design a database to model such-and-such a system."
    - Create an E-R diagram for the database
    - Translate to relational model and DDL
    - Write some queries and/or stored routines against your schema
  - Functional/multivalued dependency problems as well

Final Exam Admin Notes

- Final exam will be available towards end of week
- **Due next Thursday, December 11 at 2am**
- Solution sets for all assignments will be available by the end of the week
- (Ideally, HW5 and HW6 will be graded before the exam, but no promises…)

Entity-Relationship Model

- Diagramming system for specifying DB schemas
  - Can map an E-R diagram to the relational model
- Entity-sets (a.k.a. strong entity-sets)
  - "Things" that can be uniquely represented
  - Can have a set of attributes; <u>must</u> have a primary key
- Relationship-sets
  - Associations between two or more entity-sets
  - Can have descriptive attributes
  - Relationships in a relationship-set are uniquely identified by the participating entities, *not* the descriptive attributes
  - Primary key of relationship depends on mapping cardinality of the relationship-set

Entity-Relationship Model (2)

- Weak entity-sets
  - Don't have a primary key; have a discriminator instead
  - Must be associated with a strong entity-set via an identifying relationship
  - Diagrams must indicate both weak entity-set and the identifying relationship(s)
- Generalization/specialization of entity-sets
  - Subclass entity-sets inherit attributes and relationships of superclass entity-sets

- Schema design problems will likely involve all of these things in one way or another

E-R Model Guidelines

- You should know:
  - How to properly diagram each of these things
  - Various constraints that can be applied, what they mean, and how to diagram them
  - How to map each E-R concept to the relational model
    - Including rules for primary keys, candidate keys, etc.
- Final exam problem will require familiarity with all of these points
- Make sure you are familiar with the various E-R design issues, so you don't make those mistakes!

E-R Model Attributes

- Attributes can be:
  - Simple or composite
  - Single-valued or multivalued
  - Base or derived
- Attributes are listed in the entity-set's rectangle
  - Components of composite attributes are indented
  - Multivalued attributes are enclosed with { }
  - Derived attributes have a trailing ()
- Entity-set primary key attributes are underlined
- Weak entity-set partial key has dashed underline
- Relationship descriptive attributes aren't a key!

Example Entity-Set

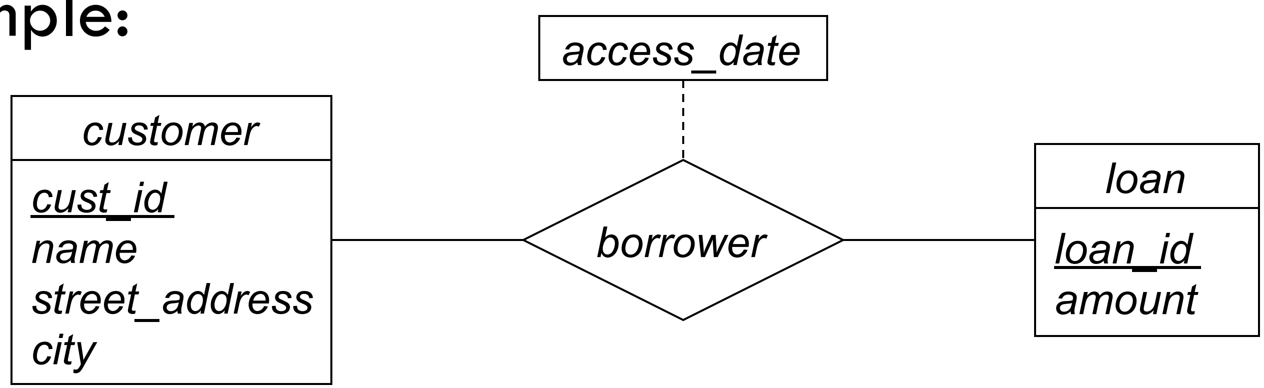
- *customer* entity-set
- Primary key:
  - *cust_id*
- Composite attributes:
  - *name, address*
- Multivalued attribute:
  - *phone_number*
- Derived attribute:
  - *age*

customer

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

Example Relationship-Set

- Relationships are identified *only* by participating entities
  - Different relationships can have same value for a descriptive attribute
- Example:



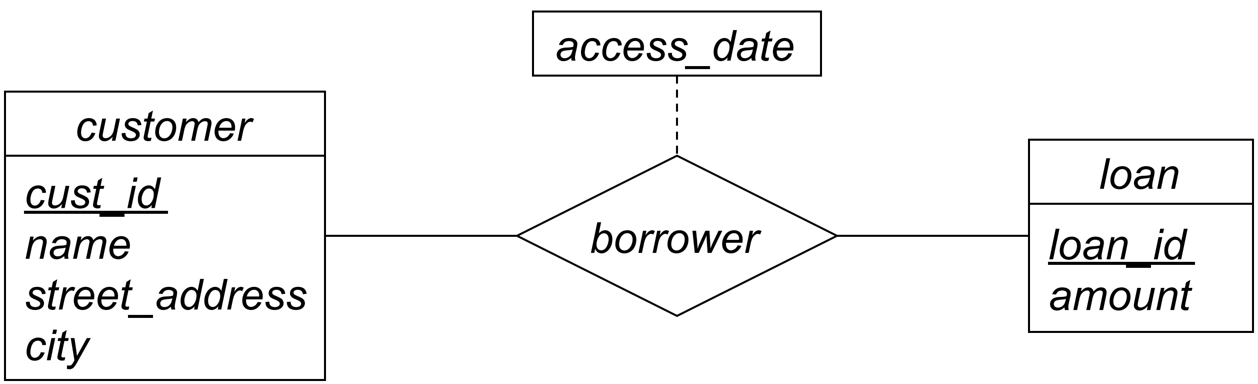
  - A given pair of *customer* and *loan* entities can only have one relationship between them via the *borrower* relationship-set

E-R Model Constraints

- E-R model can represent several constraints:
  - Mapping cardinalities
  - Key constraints in entity-sets
  - Participation constraints
- Make sure you know when and how to apply these constraints
- Mapping cardinalities:
  - "How many other entities can be associated with an entity, via a particular relationship set?"
  - Choose mapping cardinality based on the rules of the enterprise being modeled

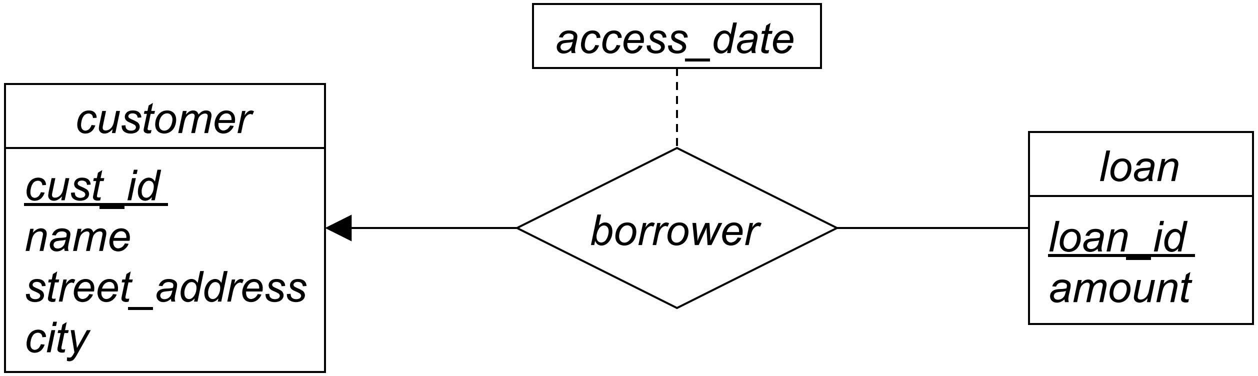
Mapping Cardinalities

- In relationship-set diagrams:
  - arrow towards entity-set represents "one"
  - line with no arrow represents "many"
  - arrow is *always* towards the entity-set
- Example: many-to-many mapping
  - The way that most banks work…

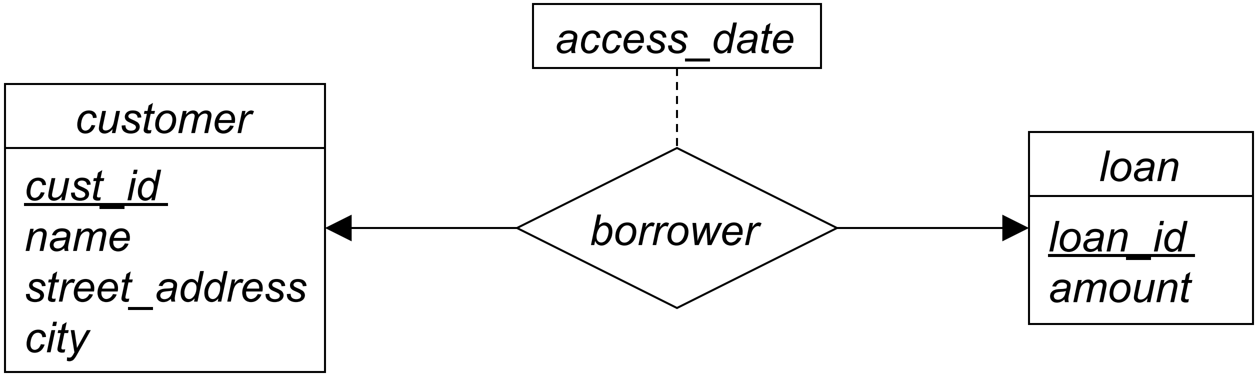


Mapping Cardinalities (2)

- One-to-many mapping:



- One-to-one mapping:



Relationship-Set Primary Keys

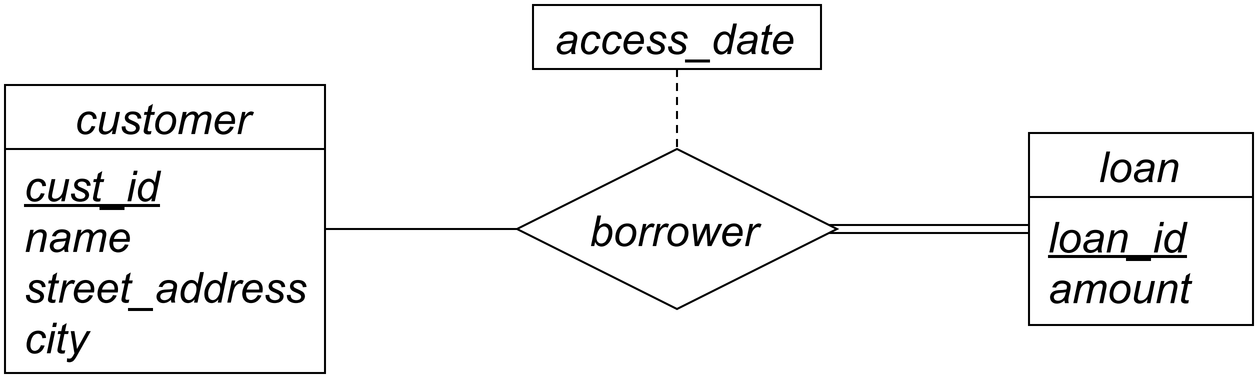
- Relationship-set *R*, involving entity-sets *A* and *B*
- If mapping is many-to-many, primary key is:
  $primary\_key(A) \cup primary\_key(B)$
- If mapping is one-to-many, $primary\_key(B)$ is primary key of relationship-set
- If mapping is many-to-one, $primary\_key(A)$ is primary key of relationship-set
- If mapping is one-to-one, use $primary\_key(A)$ or $primary\_key(B)$ for primary key
  - Enforce <u>both</u> as candidate keys in the implementation schema!

Participation Constraints

- Given entity-set *E*, relationship-set *R*
- If <u>every</u> entity in *E* participates in at least one relationship in *R*, then:
  - *E*'s participation in *R* is <u>total</u>
- If only some entities in *E* participate in relationships in *R*, then:
  - *E*'s participation in *R* is <u>partial</u>
- Use total participation when enterprise requires all entities to participate in at least one relationship

Diagramming Participation

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

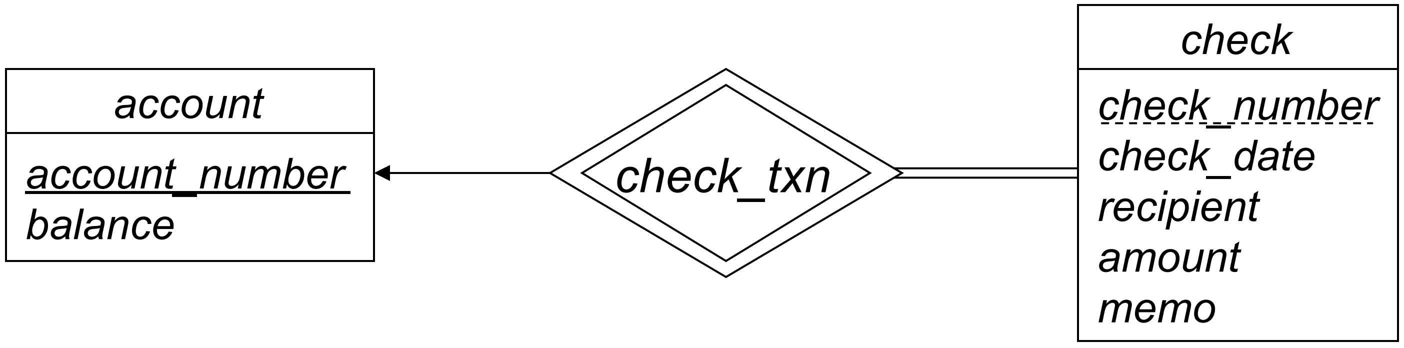


Weak Entity-Sets

- Weak entity-sets don't have a primary key
  - *Must* be associated with an identifying entity-set
  - Association called the identifying relationship
  - If you use weak entity-sets, make sure you also include both of these things!
- <u>Every</u> weak entity is associated with an identifying entity
  - Weak entity's participation in relationship-set is <u>total</u>
- Weak entities have a discriminator (partial key)
  - Need to distinguish between the weak entities
  - Weak entity-set's primary key is partial key combined with identifying entity-set's primary key

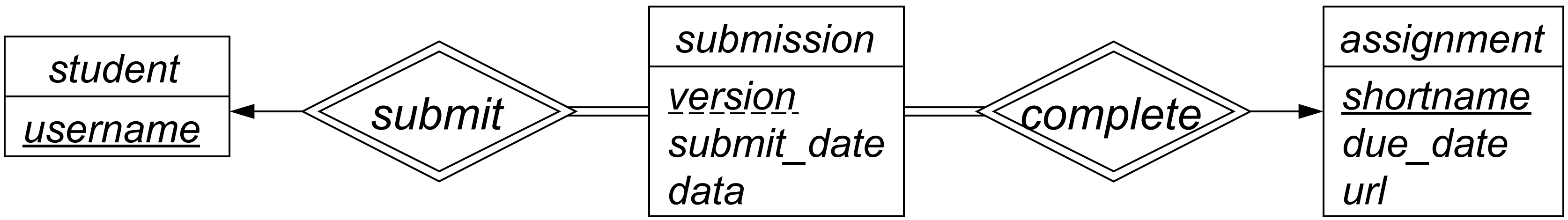
Diagramming Weak Entity-Sets

- In E-R model, can only tell that an entity-set is weak if it has a discriminator instead of a primary key
  - Discriminator attributes have a dashed underline
- Identifying relationship to owning entity-set indicated with a double diamond
  - One-to-many mapping
  - Total participation on weak entity side



Weak Entity-Set Variations

- Can run into interesting variations:
  - A strong entity-set that owns several weak entity-sets
  - A weak entity-set that has multiple identifying entity-sets
- Example:



  - Other (possibly better) ways of modeling this too, e.g. make submission a strong entity-set with its own ID
- Don't forget: weak entity-sets can also have their own non-identifying relationship-sets, etc.

Conversion to Relation Schemas

- Converting strong entity-sets is simple
  - Create a relation schema for each entity-set
  - Primary key of entity-set is primary key of relation schema
- Components of compound attributes are included directly in the schema
  - Relational model requires atomic attributes
- Multivalued attributes require a second relation
  - Includes primary key of entity-set, and "single-valued" version of attribute
- Derived attributes normally require a view
  - Must compute the attribute's value

Schema Conversion Example

- *customer* entity-set:

customer

cust_id
name
address
    street
    city
    state
    zip_code
{ *email* }

- Maps to schema:

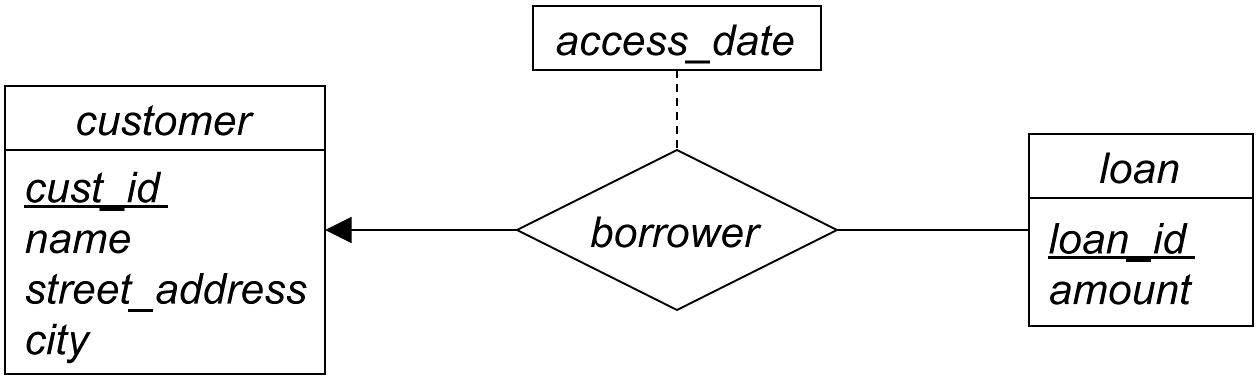
  customer(*cust_id*, *name*, *street*, *city*, *state*, *zipcode*)

  customer_emails(*cust_id*, *email*)

- Primary-key attributes come first in attribute lists!

Schema Conversion Example (2)

- Bank loans:



- Maps to schema:

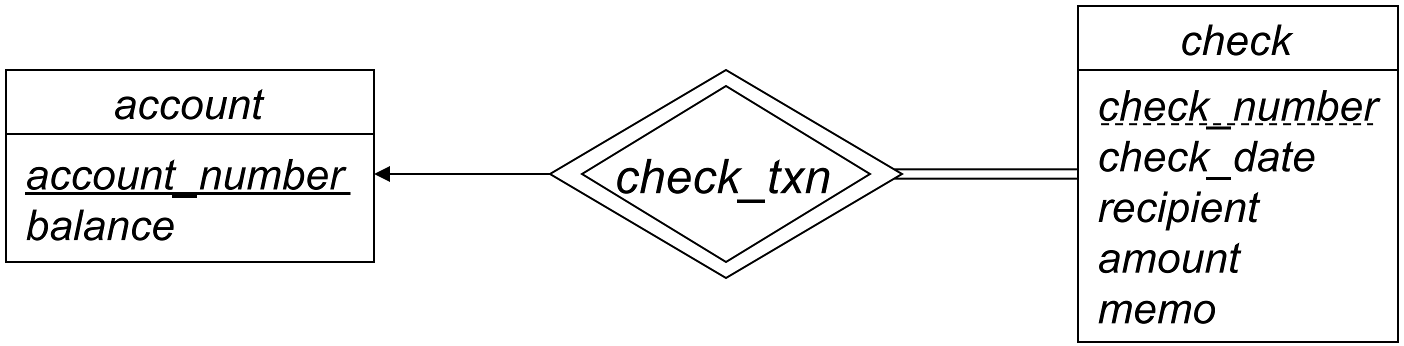
  customer(<u>*cust_id*</u>, *name*, *street_address*, *city*)

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

  borrower(<u>*loan_id*</u>, *cust_id*, *access_date*)

Schema Conversion Example (3)

- Checking accounts:



- Maps to schema:

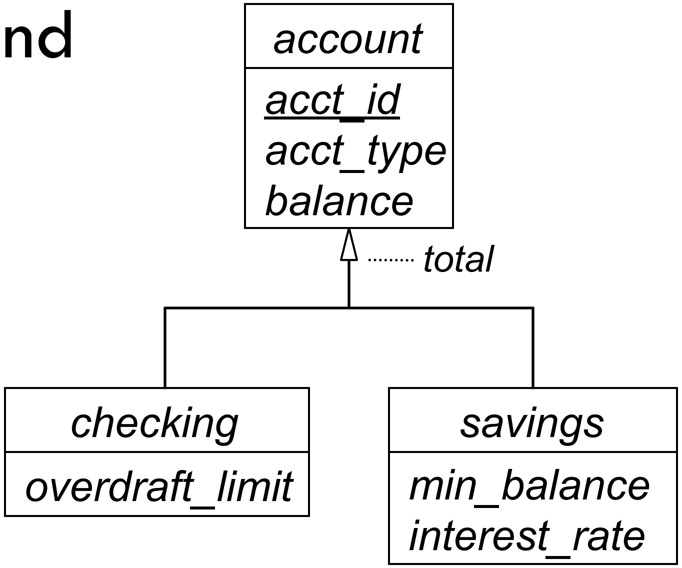
  account(<u>*account_number*</u>, *balance*)

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

  - No schema for identifying relationship!

Generalization and Specialization

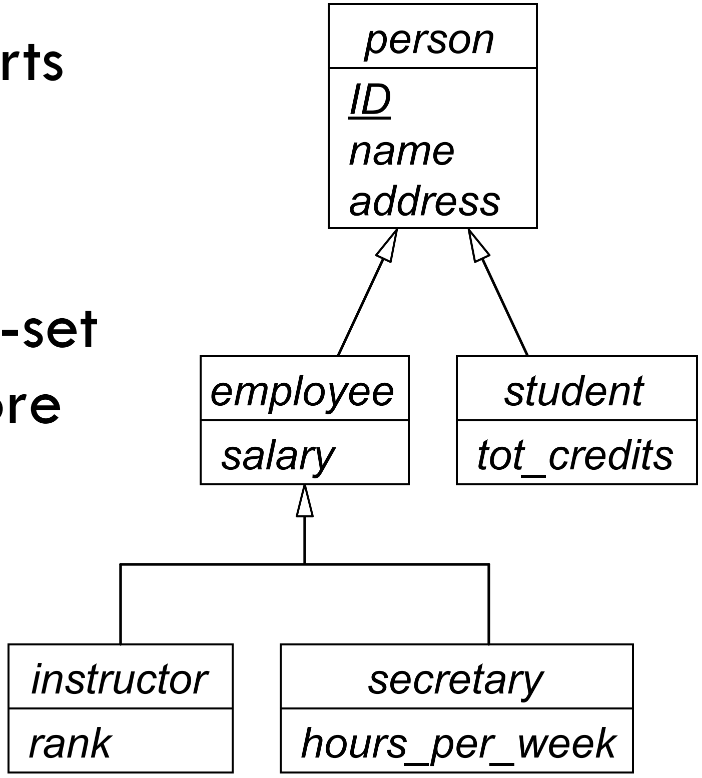
- Use generalization when multiple entity-sets represent similar concepts
- Example: checking and savings accounts



- Attributes <u>and relationships</u> are inherited
  - Subclass entity-sets can also have own relationships

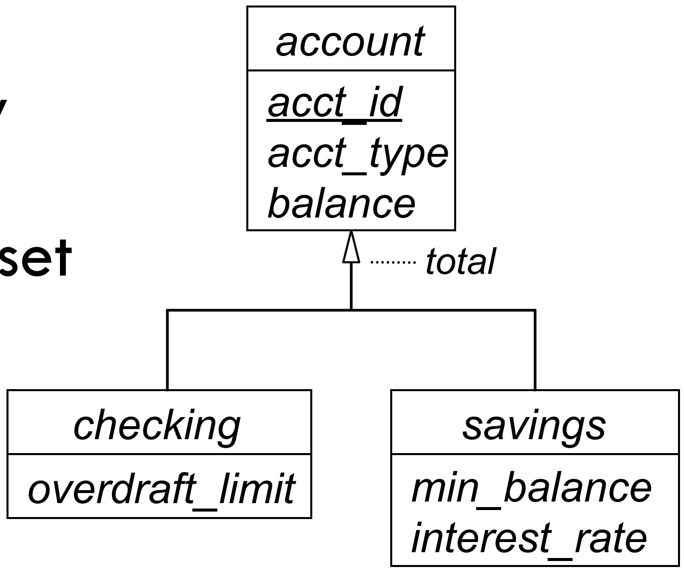
Specialization Constraints

- Disjointness constraint, a.k.a. disjoint specialization:
  - Every entity in superclass entity-set can be a member of at most one subclass entity-set
  - One arrow split into multiple parts shows disjoint specialization
- Overlapping specialization:
  - An entity in the superclass entity-set can be a member of zero or more subclass entity-sets
  - Multiple separate arrows show overlapping specialization



Specialization Constraints (2)

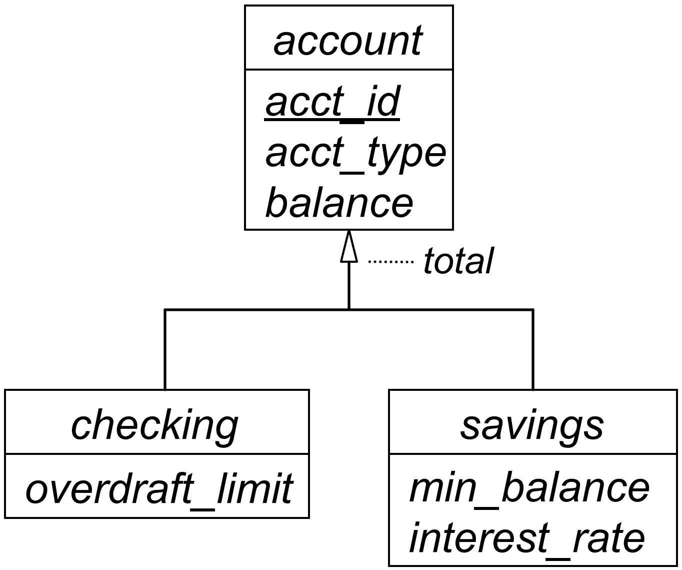
- Completeness constraint:
  - Total specialization: every entity in superclass entity-set must be a member of some subclass entity-set
  - Partial specialization is default
  - Show total specialization with "total" annotation on arrow



- Membership constraint:
  - What makes an entity a member of a subclass?
  - Attribute-defined vs. user-defined specialization

Generalization Example

- Checking and savings accounts:



- One possible mapping to relation schemas:

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

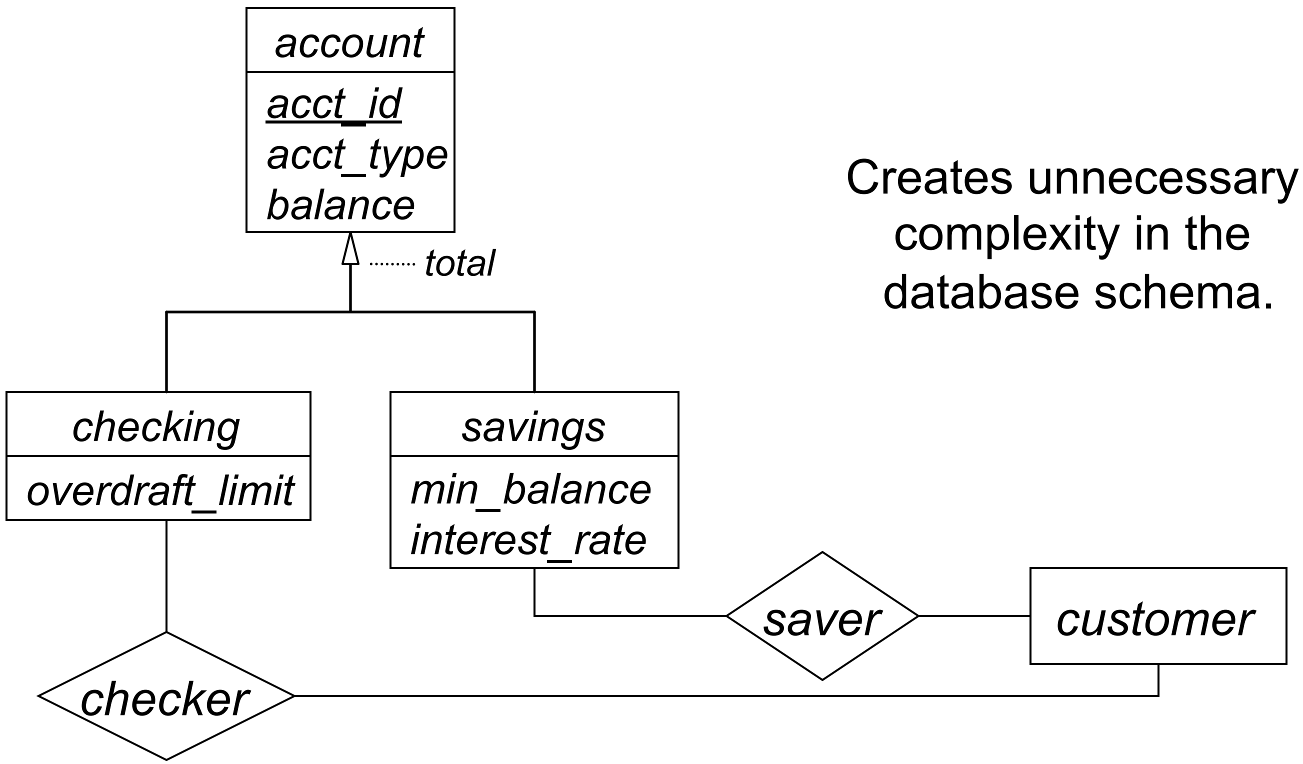
  checking(<u>*acct_id*</u>, *overdraft_limit*)

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

- Be familiar with other mappings, and their tradeoffs

Generalization and Relationships

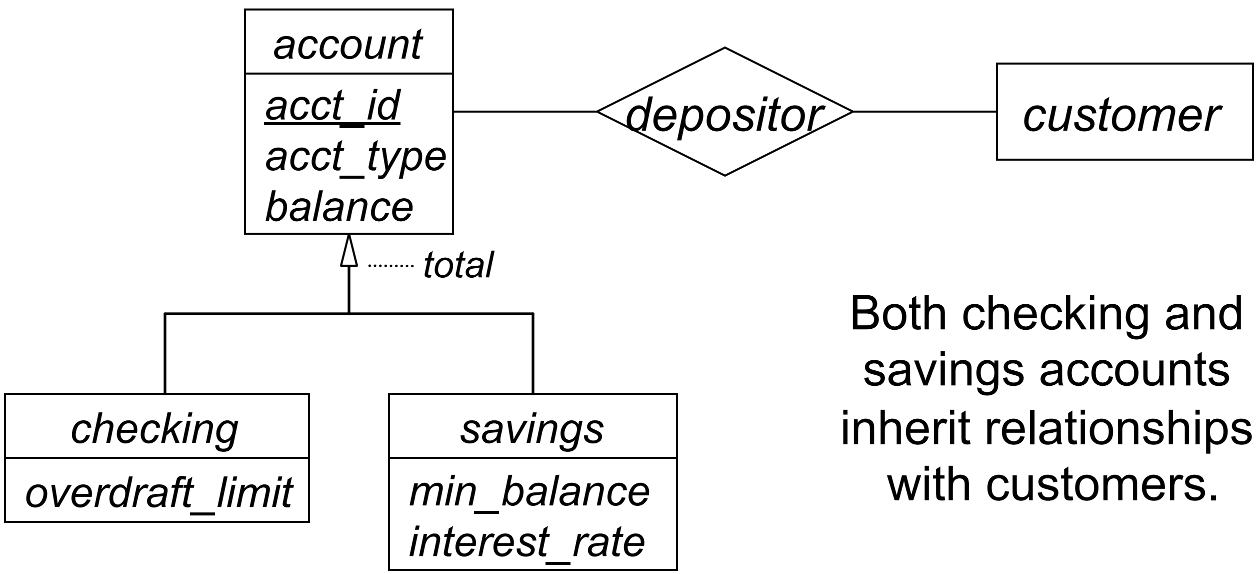
- If <u>all</u> subclass entity-sets have a relationship with a particular entity-set:
  - e.g. all accounts are associated with customers
  - <u>Don't</u> create a separate relationship for each subclass entity-set!



Creates unnecessary complexity in the database schema.

Generalization, Relationships (2)

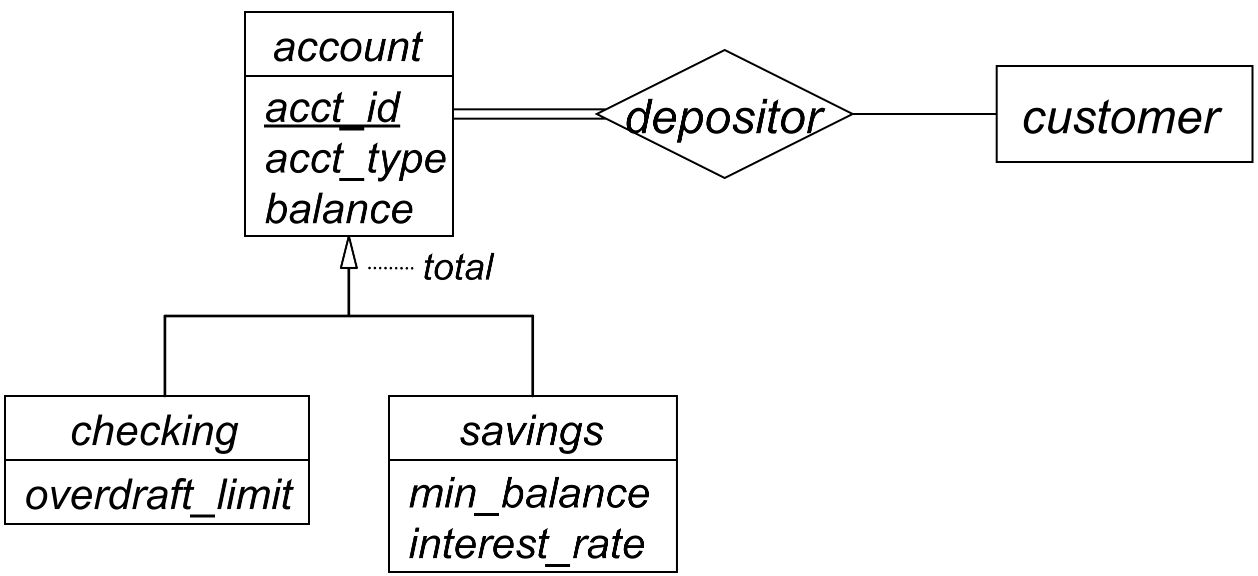
- If <u>all</u> subclass entity-sets have a relationship with a particular entity-set:
  - Create a relationship with superclass entity-set
  - Subclass entity-sets inherit this relationship



Both checking and savings accounts inherit relationships with customers.

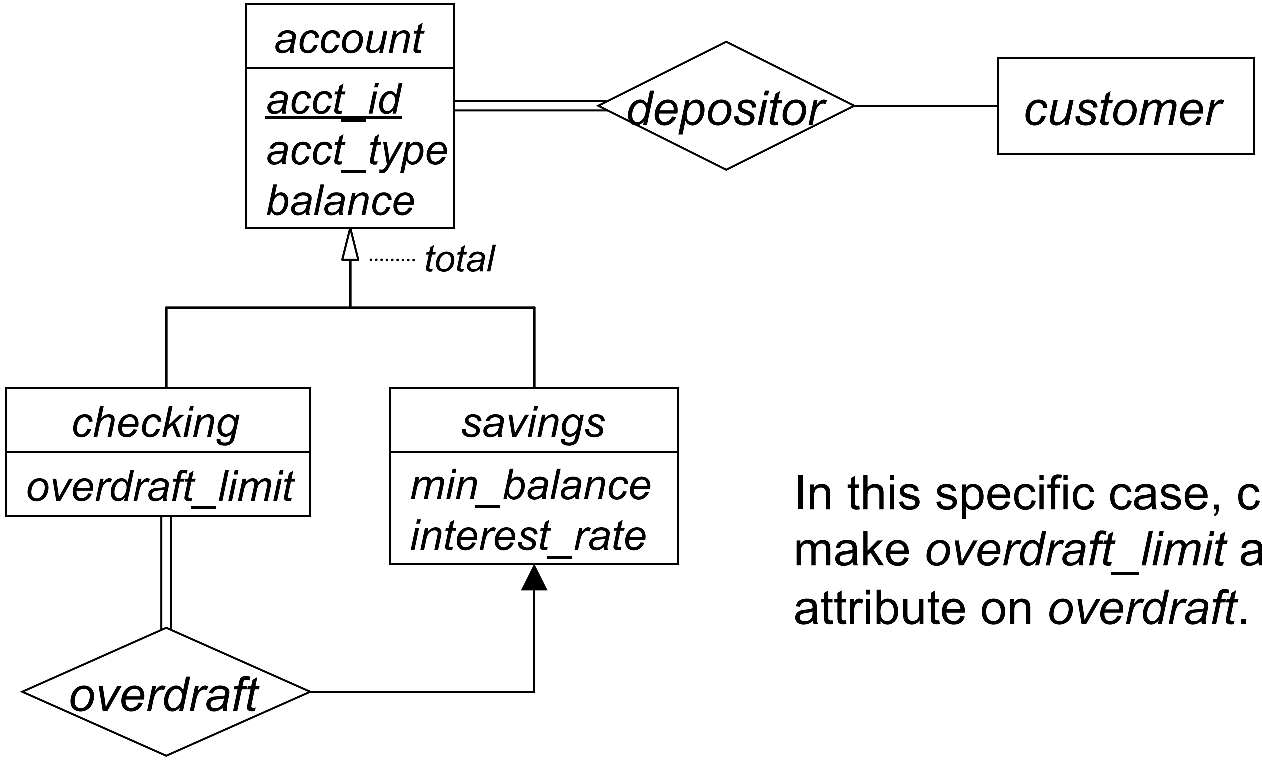
Generalization, Relationships (3)

- Finally, ask yourself:
  - "What constraints should I enforce on *depositor* ?"
  - All accounts have to be associated with at least one customer
  - A customer may have zero or more accounts
  - *account* has total participation in *depositor*



Generalization, Relationships (4)

- Subclass entity-sets can have their own relationships
  - e.g. associate every checking account with one specific "overdraft" savings account
  - What constraints on *overdraft* ?



In this specific case, could also make *overdraft_limit* a descriptive attribute on *overdraft*.

Normal Forms

- Normal forms specify “good” patterns for database schemas
- First Normal Form (1NF)
  - All attributes must have atomic domains
  - Happens automatically in E-R to relational model conversion
- Second Normal Form (2NF) of historical interest
  - Don’t need to know about it
- Higher normal forms use more formal concepts
  - Functional dependencies: BCNF, 3NF
  - Multivalued dependencies: 4NF

Normal Form Notes

- Make sure you can:
  - Identify and state functional dependencies and multivalued dependencies in a schema
  - Determine if a schema is in BCNF, 3NF, 4NF
  - Normalize a database schema
- Functional dependency requirements:
  - Apply rules of inference to functional dependencies
  - Compute the closure of an attribute-set
  - Compute F_c from F, without any programs this time ☺
  - Identify extraneous attributes

Functional Dependencies

- Given a relation schema R with attribute-sets $\alpha, \beta \subseteq R$
  - The functional dependency $\alpha \rightarrow \beta$ holds on $r(R)$ if $\langle \forall t_1, t_2 \in r : t_1[\alpha] = t_2[\alpha] : t_1[\beta] = t_2[\beta] \rangle$
  - If α is the same, then β must be the same too
- Trivial functional dependencies hold on all possible relations
  - $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$
- A superkey functionally determines the schema
  - K is a superkey if $K \rightarrow R$

Inference Rules

- Armstrong's axioms:
  - Reflexivity rule:
    If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
  - Augmentation rule:
    If $\alpha \rightarrow \beta$ holds, and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
  - Transitivity rule:
    If $\alpha \rightarrow \beta$ holds, and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.
- Additional rules:
  - Union rule:
    If $\alpha \rightarrow \beta$ holds, and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
  - Decomposition rule:
    If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
  - Pseudotransitivity rule:
    If $\alpha \rightarrow \beta$ holds, and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Sets of Functional Dependencies

- A set *F* of functional dependencies
- F^+ is closure of *F*
  - Contains all functional dependencies in *F*
  - Contains all functional dependencies that can be logically inferred from *F*, too
  - Use Armstrong's axioms to generate F^+ from *F*
- F_c is canonical cover of *F*
  - *F* logically implies F_c, and F_c logically implies *F*
  - No functional dependency has extraneous attributes
  - All dependencies have unique left-hand side
- **Review how to test if an attribute is extraneous!**

Boyce-Codd Normal Form

- Eliminates all redundancy that can be discovered using functional dependencies
- Given:
  - Relation schema R
  - Set of functional dependencies F
- R is in BCNF with respect to F if:
  - For all functional dependencies $\alpha \rightarrow \beta$ in F^+, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
    - $\alpha \rightarrow \beta$ is a trivial dependency
    - α is a superkey for R
- Is <u>not</u> dependency-preserving
  - Some dependencies in F may not be preserved

Third Normal Form

- A dependency-preserving normal form
  - Also allows more redundant information than BCNF
- Given:
  - Relation schema R, set of functional dependencies F
- R is in 3NF with respect to F if:
  - For all functional dependencies $\alpha \rightarrow \beta$ in F^+, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
    - $\alpha \rightarrow \beta$ is a trivial dependency
    - α is a superkey for R
    - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R
- Can generate a 3NF schema from F_c

Multivalued Dependencies

- Functional dependencies cannot represent multivalued attributes
  - Can't use functional dependencies to generate normalized schemas including multivalued attributes
- Multivalued dependencies are a generalization of functional dependencies
  - Represented as $\alpha \twoheadrightarrow \beta$
- More complex than functional dependencies!
  - Real-world usage is usually very simple
- Fourth Normal Form
  - Takes multivalued dependencies into account

Multivalued Dependencies (2)

- Multivalued dependency $\alpha \twoheadrightarrow \beta$ holds on R if, in any legal relation $r(R)$:
  - For all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$
  - There also exists tuples t_3 and t_4 in r such that:
    - $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
    - $t_1[\beta] = t_3[\beta]$ and $t_2[\beta] = t_4[\beta]$
    - $t_1[R - \beta] = t_4[R - \beta]$ and $t_2[R - \beta] = t_3[R - \beta]$
- Pictorially:

	α	β	$R - (\alpha \cup \beta)$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

Trivial Multivalued Dependencies

- $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency on R if all relations $r(R)$ satisfy the dependency
- Specifically, $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$, or if $\alpha \cup \beta = R$

- Note that a multivalued dependency's trivial-ness may depend on the schema!
  - $A \twoheadrightarrow B$ is trivial on $R_1(A, B)$, but it is not trivial on $R_2(A, B, C)$
  - A major difference between functional and multivalued dependencies!
  - For functional dependencies: $\alpha \rightarrow \beta$ is trivial only if $\beta \subseteq \alpha$

Functional & Multivalued Dependencies

- Functional dependencies are also multivalued dependencies
  - If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$ too
  - Additional caveat: each value of α has at most one associated value for β

- Don't state functional dependencies as multivalued dependencies!
  - Much easier to reason about functional dependencies!

Functional & Multivalued Dependencies (2)

- Given a relation $R_1(\alpha, \beta)$ with $\alpha \rightarrow \beta$ and $\alpha \cap \beta = \emptyset$
  - What is the key of R_1?
  - $R_1(\underline{\alpha}, \beta)$
- Given a relation $R_2(\alpha, \beta)$ with $\alpha \twoheadrightarrow \beta$ and $\alpha \cap \beta = \emptyset$
  - What is the key of R_2?
  - $R_2(\alpha, \beta)$ – i.e. all attributes $\alpha \cup \beta$ are part of the key of R_2
- This is why we don't state functional dependencies as multivalued dependencies

Fourth Normal Form

- Given:
  - Relation schema R
  - Set of functional and multivalued dependencies D
- R is in 4NF with respect to D if:
  - For all multivalued dependencies $\alpha \twoheadrightarrow \beta$ in D^+, where $\alpha \in R$ and $\beta \in R$, at least one of the following holds:
    - $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency
    - α is a superkey for R
  - Note: If $\alpha \rightarrow \beta$ then $\alpha \twoheadrightarrow \beta$
- A database design is in 4NF if all schemas in the design are in 4NF