# DATABASE SCHEMA DESIGN
# ENTITY-RELATIONSHIP MODEL

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 14

# Designing Database Applications



- Database applications are large and complex
- A few of the many design areas:
  - Database schema (physical/logical/view)
  - Programs that access and update data
  - Security constraints for data access
- Also requires familiarity with the problem domain
  - Domain experts *must* help drive requirements

# General Approach



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

# Conceptual Schema



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

# Implementation Phases



- Once conceptual schema and functional requirements are verified:
  - Convert conceptual schema into an 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
- <u>Very important</u> to spend time up front designing the database schema

# Design Decisions



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

# More Design Decisions



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

# Simple Design Example



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

# The Entity-Relationship Model



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

# Entities and Entity-Sets



- An <u>entity</u> is any "thing" that can be uniquely represented
    - e.g. a product, an employee, a software defect
  - Each entity has a set of <u>attributes</u>
  - Entities are uniquely identified by some set of attributes
- An <u>entity-set</u> 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)



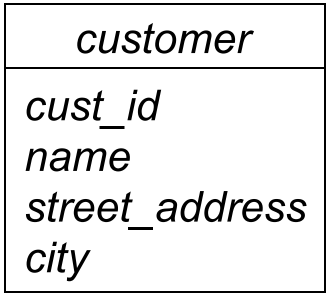
- An entity has a set of attributes
  - Each attribute has a name and domain
  - Each attribute also has a corresponding value
- Entity-sets also specify a set of attributes
  - Every entity in the entity-set has the same set of attributes
  - Every entity in the entity-set has its own value for each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

# Diagramming an Entity-Set



Example: a *customer* entity-set

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




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

# Relationships



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

# Relationships (2)



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

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

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

# Relationships and Roles



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

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

# Relationships and Roles (2)



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

Example: a relationship-set named *works_for*, specifying employee/manager assignments

  - Relationship involves two entities, and both are *employee* entities
- Roles are given names to distinguish entities
  - The relationship is a set of entities <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

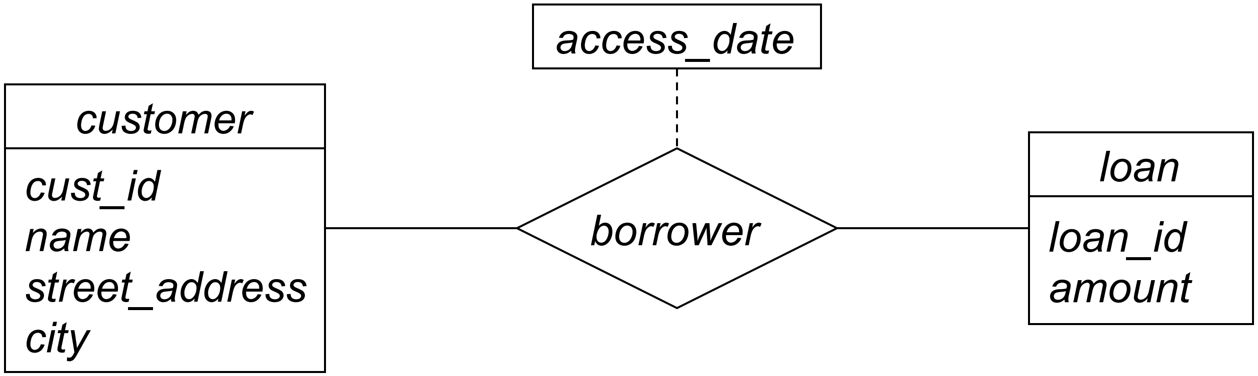


- Most relationships in a schema are <u>binary</u>
  - Two entities are involved in the relationship
- Sometimes there are ternary relationships
  - Three entities are involved
  - Far less common, but still useful at times
- The number of entity-sets that participate in a relationship-set is called 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*<br>*name*<br>    *first_name*<br>    *middle_initial*<br>    *last_name*<br>*address*<br>    *street*<br>    *city*<br>    *state*<br>    *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*<br>*name*<br>    *first_name*<br>    *middle_initial*<br>    *last_name*<br>*address*<br>    *street*<br>    *city*<br>    *state*<br>    *zip_code*<br>{ *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*<br>*name*<br>    *first_name*<br>    *middle_initial*<br>    *last_name*<br>*address*<br>    *street*<br>    *city*<br>    *state*<br>    *zip_code*<br>{ *phone_number* }<br>*birth_date*<br>*age* () |

# Representing Constraints



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

# Mapping Cardinalities



- Mapping cardinality represents:

  "How many other entities can be associated with an entity, via a particular relationship set?"

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

# Mapping Cardinalities (2)

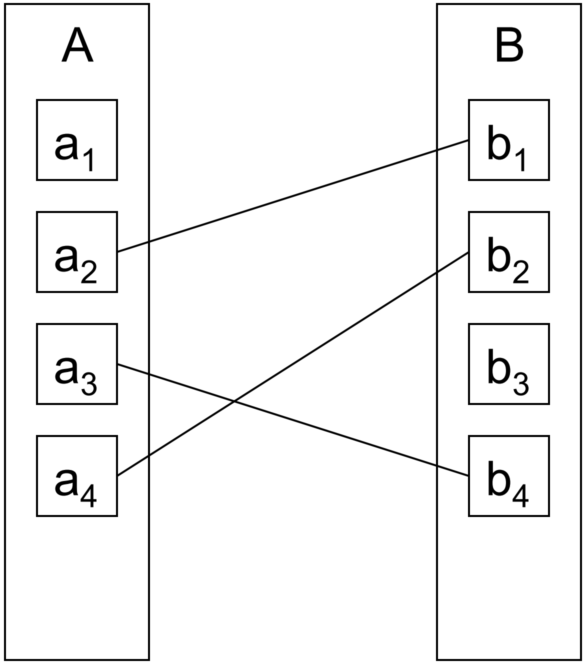


Given:

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

One-to-one mapping (1:1)

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



# Mapping Cardinalities (3)

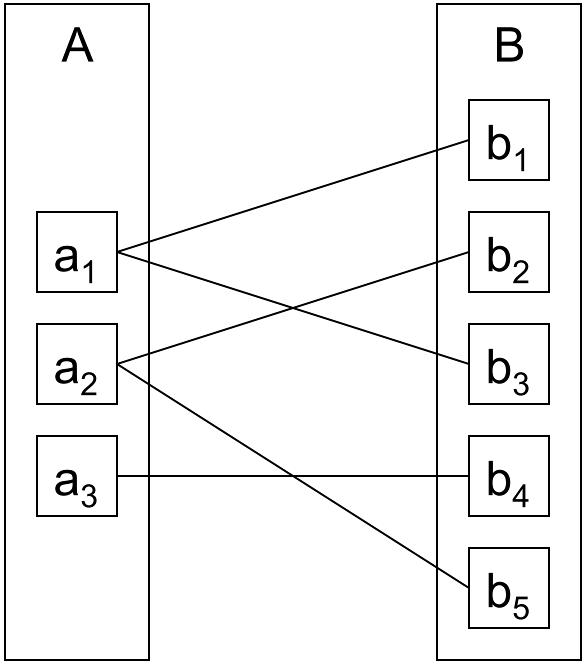


One-to-many mapping (1:*M*)

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

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

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

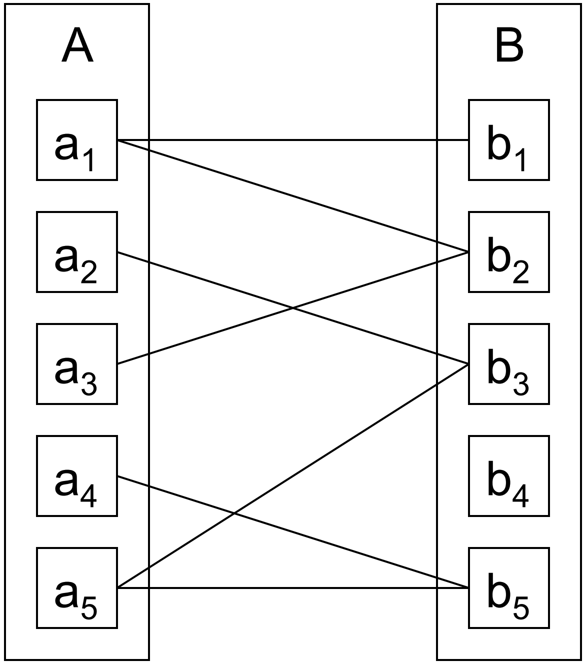


# Mapping Cardinalities (4)



Many-to-many mapping

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



# Mapping Cardinalities (5)



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

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



One-to-one mapping:

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

  (e.g. with spouse or business partner)

One-to-many mapping:

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

Many-to-one mapping:

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

Many-to-many mapping:

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

Best choice for *borrower* :

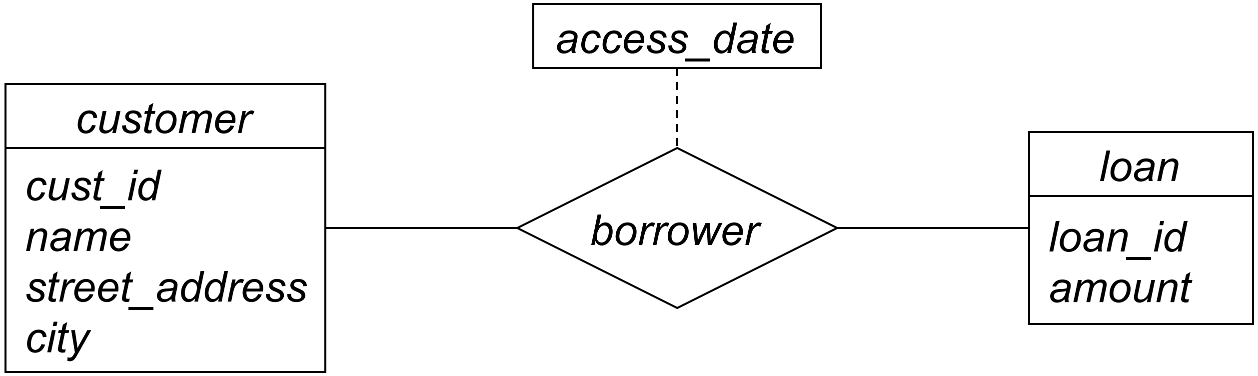
many-to-many mapping

***Handles real-world needs!***

# Diagramming Cardinalities



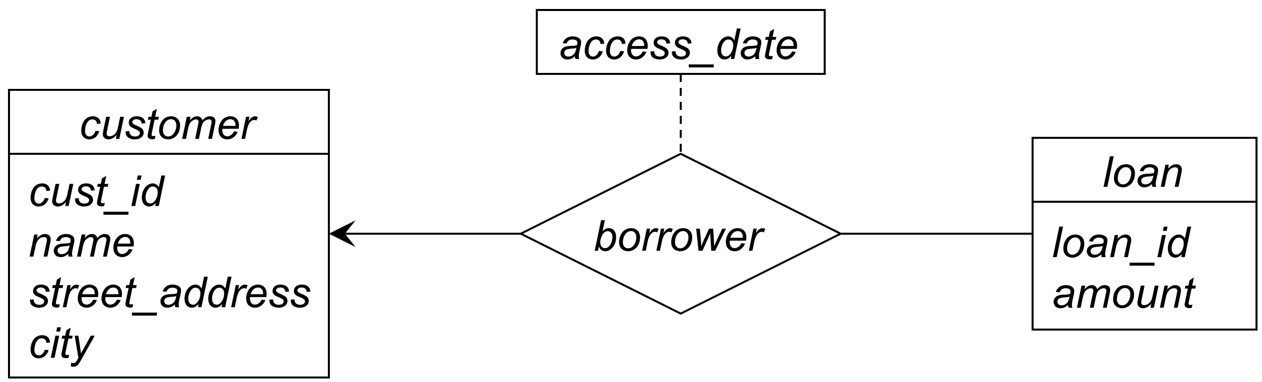
- In relationship-set diagrams:
  - an arrow towards an entity represents "one"
  - a simple line represents "many"
  - arrow is *always* towards the entity
- Many-to-many mapping between *customer* and *loan*:



# Diagramming Cardinalities (2)



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

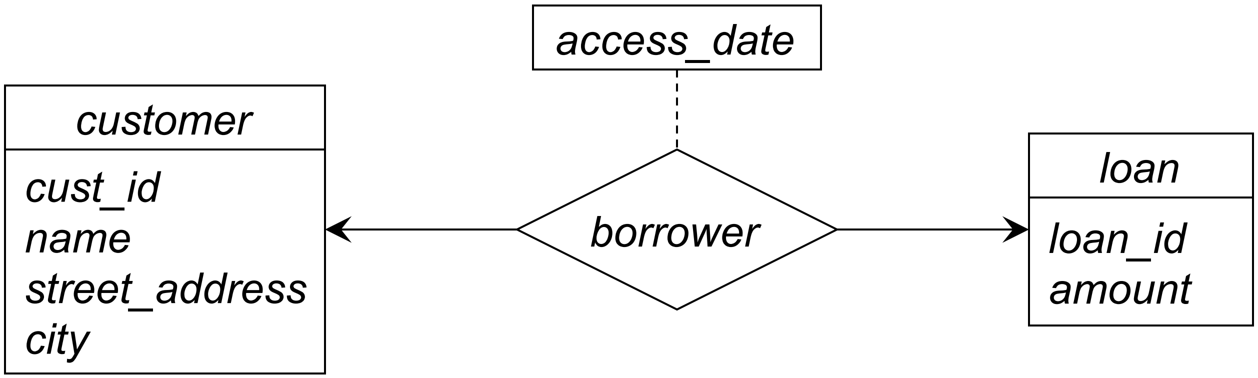



- Each customer can have multiple loans
- A loan is owned by <u>exactly</u> one customer
  - (Actually, this is technically "<u>at most</u> one". Participation constraints will allow us to say "exactly one.")

# Diagramming Cardinalities (3)



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




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