CS 121: Relational Databases

Passwords, Final Project, MySQL with Python

Agenda

Account password management

Final Project Overview

Python with MySQL demo

Last year's Final Exam review slides included at the end of this slide deck for summary of relevant information

Account Password Management

Consider 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

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:
  - User enters password
  - Database application hashes the password
  - If hash matches value stored in DB, authentication succeeds

Hashed Passwords (2)

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 = 'dbadmin';
```

More encryption functions for MySQL 8.0

Hashed Passwords (3)

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 (4)

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

- Use hash-values to look up the corresponding password
- If attacker gets the hash values from the database, can crack some subset of accounts

Hashed, Salted Passwords

Solution: **salt** 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, <u>before</u> hashing
- Attacker would have to compute a dictionary of hashes for each salt value… Prohibitively expensive!

Password Management

Basically no 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

Final Project Overview

Final Project

El is grading the project proposals, hoping to have everyone's feedback by tomorrow (Wednesday)

You will submit the Final Project on **CodePost** with a minimum of the following:

- **`ra.pdf`** - Relational Algebra component
- **`setup.sql`** - Your DDL for your database, including indexes defined after your DDL
- **`setup-passwords.sql`** - A separate DDL file for implementing basic password management in your application (see appendix)
- **`load-data.sql`** - SQL with load statements to load your database
- **`setup-routines.sql`** - SQL for stored routines and triggers
- **`queries.sql`** - SQL queries for your database
- **`app.py`** - Your command-line Python program
- **`readme.txt|md`** - README for staff to follow steps on using your application
- **`reflection.pdf`** - Reflection component for your Final Project, including written portion requirements (e.g. your ER diagrams, justifications/workflow, and normal form responses).

Relational Algebra

This document will include relational algebra for at least 4 of your queries (schemas not required)

- At least one group by with aggregation
- At least one each of update, insert, delete
- At least 4 joins (minimum 2 queries)
  - If your application does not have 4 joins in your queries, you can add another query in your RA not used in your application
- Appropriate projection/extended projection use

DDL: `setup.sql`

In this file, you will include **documented** DDL for each of your tables in your database, including indexes defined after your DDL

- You should have a minimum of 4 tables in your DDL with at least 12 attributes total
- Use cascading deletes and updates where appropriate
- Use PKs and FKs where appropriate
- Use attribute types appropriately; e.g. don't use floating point types for something that should be fixed, use **`TINYINT`** for small ints, use **`CHAR`** vs. **`VARCHAR`** appropriately, etc.
  - You should have at least 5 different MySQL types across your tables
- Use **`NOT NULL`** where appropriate (don't forget to refer to your ER diagram, which should be in sync with your DDL where reasonable)
- Include brief comments for each of your tables, inline comments for less-obvious attributes (refer to [`setup-airbnb.sql`]() for an example)

Password Management: **`setup-passwords.sql`**

In this file, you will define your DDL for users with usernames, hashed passwords, and salts

- You may use peppers for an added challenge
- An appendix will be provided for you to walk you through implementing password management in MySQL
- If you have a schema you were planning on using for users in **`setup.sql`**, put it in this file, and assume we will run setup.sql followed by **`setup-passwords.sql`** (in MySQL 8.0)

Loading Data: `load-data.sql`

We will be grading your database and application based on data you provide

- You may either include local load statements in this file, similar to the Spotify assignment and midterm (preferred), or you may use INSERT statements for smaller datasets
- For the first option, don't forget to make sure your CSV files are uploaded!
  - If you have issues uploading very large CSV files on CodePost, please DM or email El

Procedural SQL: `setup-routines.sql`

This file will include the setup code for defining your procedural SQL, all of which should be used in your Python application:

- At least one UDF function
- At least one procedure
- At least one trigger

Partner projects must have at least one addition function, procedure, or trigger

SQL Queries: `queries.sql`

This file will include all of your SQL queries, which we will test independently of your Python application (but several will be used in your application)

Your Python Application: `app.py`

This will be the command-line program you implement to simulate your application

- Must be Python 3.0+, we will be running it!

Requirements:

- A start menu, listing options for users
- Functionality for a user to log in (at minimum, as admin or client)
  - Alternatively, may write two .py programs, one for admins, one for clients
- Functionality to use 3 select queries that make sense with your application
  - One of these may be related to logging in a user
- Functionality to call at least one procedure to modify your database
- No output should indicate MySQL usage; use mysql library appropriately!
- Separation of concerns between MySQL and Python

Your Python Application: `app.py` (2)

Requirements (continued)

- Part of your grade will be proper use of Python and program decomposition
  - E.g. use a main function, a show_options() function, and encapsulate your queries into functions based on your command-line support
- Use docstrings to document your functions
- You may use external Python libraries, but will have limited support from staff for material outside of CS 121
- You are also expected to follow standard testing and debugging of your program, don't forget to test edge cases!

`readme.txt|md`

For full credit, you must provide a README for staff to follow steps on using your application

Doesn't have to be fancy, but we should be able to know how to set up your database and load your data

Written Components (`reflection.pdf`)

Reflection component for your Final Project, including written portion requirements:

- ER diagrams (complete for your database)
- Normal Form portion
  - You will be asked to justify your choice of BCNF, 3NF, and/or 4NF for your schemas, as well as identifying functional dependencies
- Short written responses about your project design and implementation, such as:
  - Process of finding your dataset and citation of any external sources
  - Justification of your design
  - Challenges you ran into when designing, implementing, and possibly reiterating your application
  - Future work

More information will be laid out in the published specification by tomorrow

Python with MySQL

Requirements

- [Python 3]
- [mysql-connector-python] library

```
$ pip3 install mysql-connector-python
```

Creating a Database Connection

```
import mysql.connector;

conn = mysql.connector.connect(
  host='localhost", # would change to a database server in production
  user='yourusername',
  # Find port in MAMP or MySQL Workbench GUI or with
  # SHOW VARIABLES WHERE variable_name LIKE 'port';
  port='3306',
  password='yourpassword',
  database='databasename'
)

# For local dev, may be able to do:
conn = mysql.connector.connect(
  user='root', # default localhost, no pw
  database='shelterdb'
)
```

Wrapping into a Function...

```
DEBUG = True # top of file, don't want to leak sensitive information to users!
...
def get_conn():
    try:
        conn = mysql.connector.connect(
          # From before
        )
        if DEBUG:
            print('Successfully connected.')
        return conn
    except mysql.connector.Error as err:     # Error-handling
      if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        sys.stderr('Incorrect username or password.')
      elif err.errno == errorcode.ER_BAD_DB_ERROR and DEBUG:
        sys.stderr('Database does not exist.')
      elif DEBUG:
        sys.stderr(err)
      else:
        sys.stderr('An error occurred, please contact the administrator.')
        sys.exit(1)
```

Coding Demo!

Coding demo: `lecture-demo.py`

Template code for Final Project: `app-template.py`

Appendix: Final Review Slides

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 most or 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-set 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:

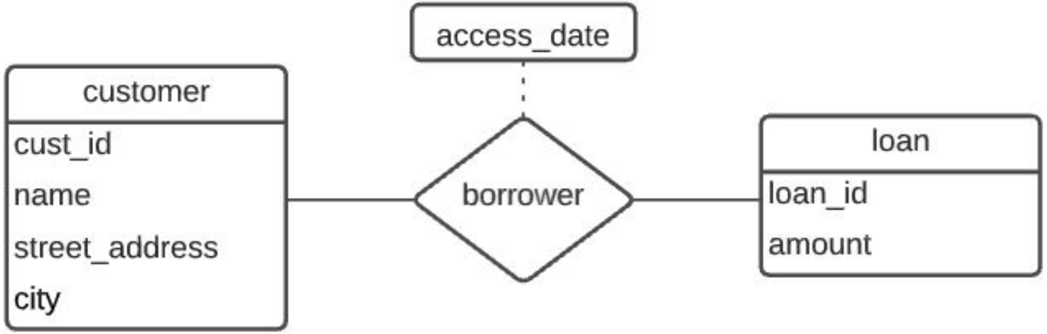
- *age*

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 <u>one</u> 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

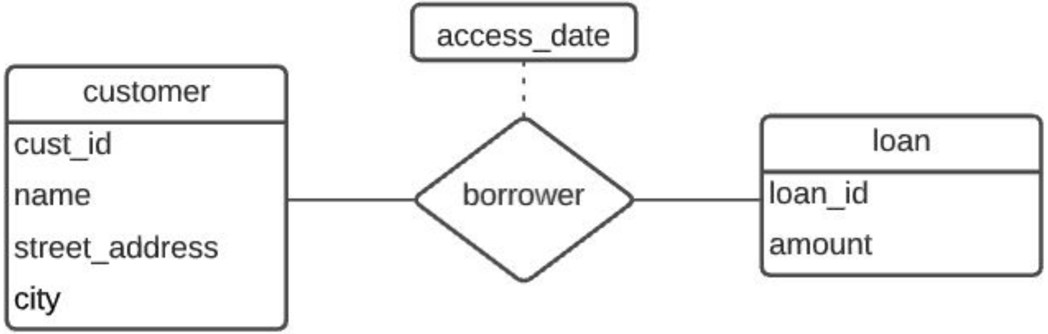
Mapping Cardinalities

In relationship-set diagrams:

- arrow towards entity-set represents “one”
- line with no arrow represents “many”
- arrow is *always* towards the entity-set

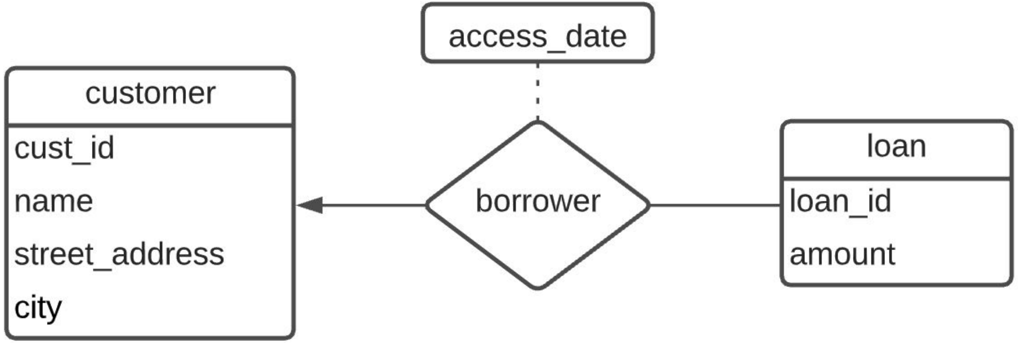
Example: many-to-many mapping

- The way that most banks work…

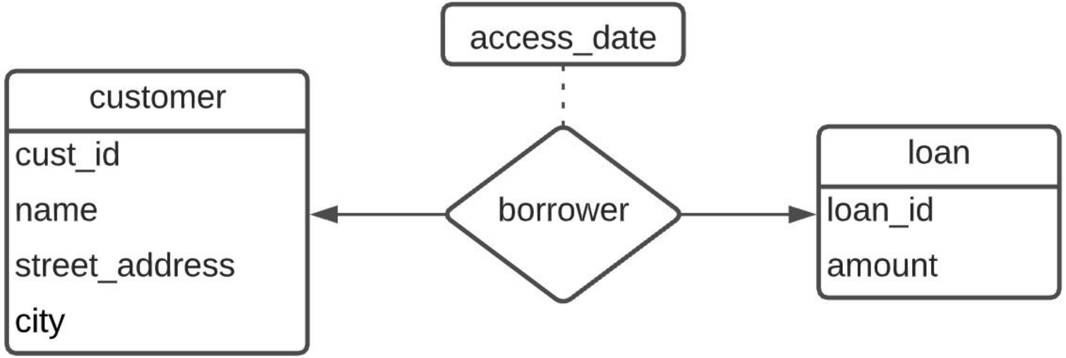


Mapping Cardinalities (2)

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*) ∪ *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 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:

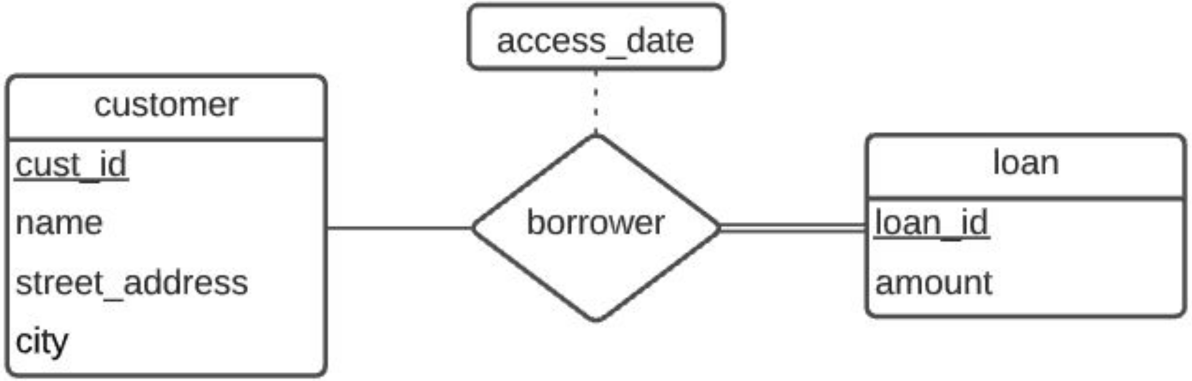
- *E*'s participation in *R* is **partial**

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

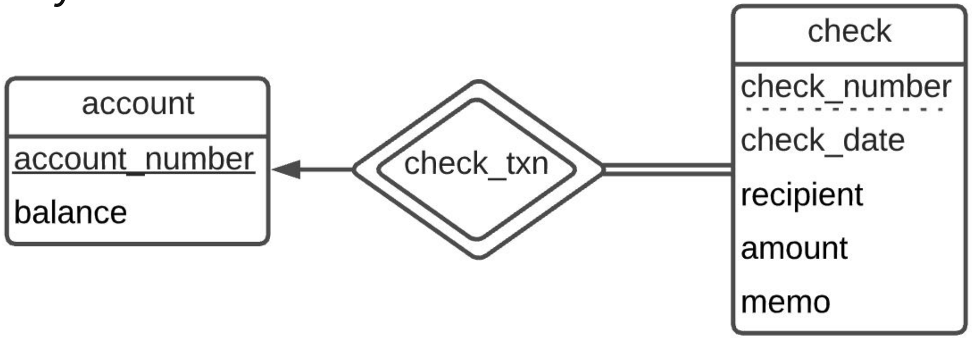
Diagramming Weak Entity-Sets

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

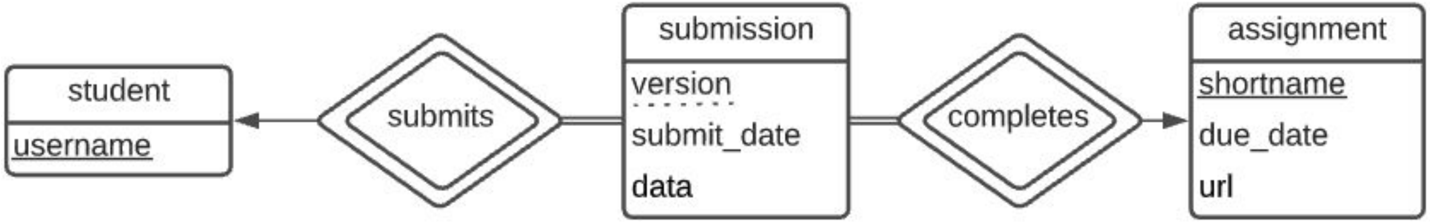


Weak Entity-Set Variations

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:

Maps to schema:

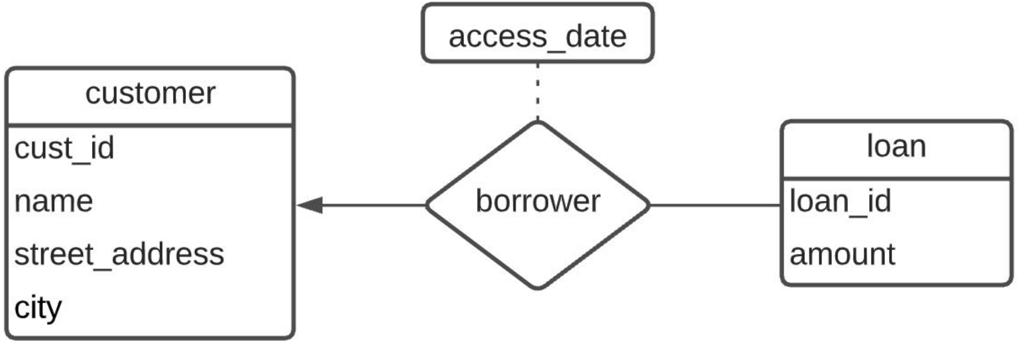
customer(*cust_id*, *name*, *street*, *city*, *state*, *zipcode*)

customer_emails(*cust_id*, *email*)

Primary-key attributes come first in attribute lists!

Schema Conversion Example (2)

Bank loans:



Maps to schema:

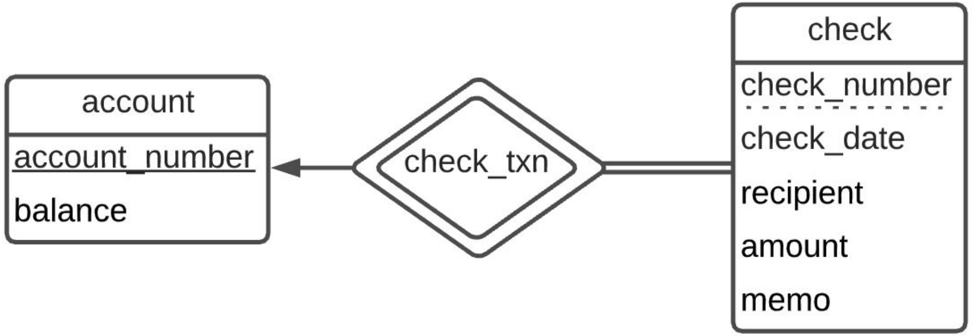
customer(<u>*cust_id*</u>, *name*, *street_address*, *city*)

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*)

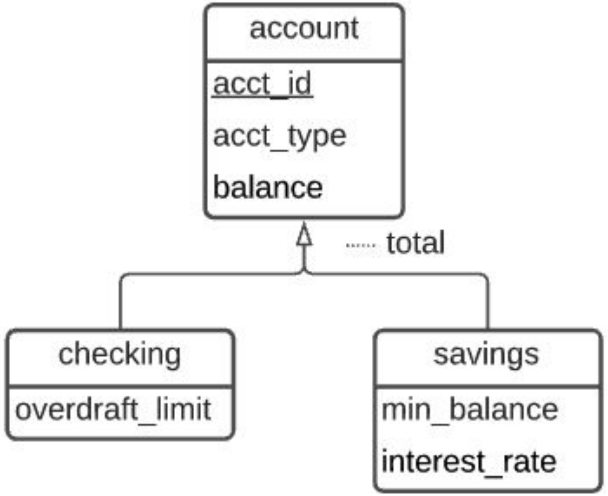
check(<u>*account_number*</u>, <u>*check_number*</u>, *check_date*, *recipient*, *amount*, *memo*)

- 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

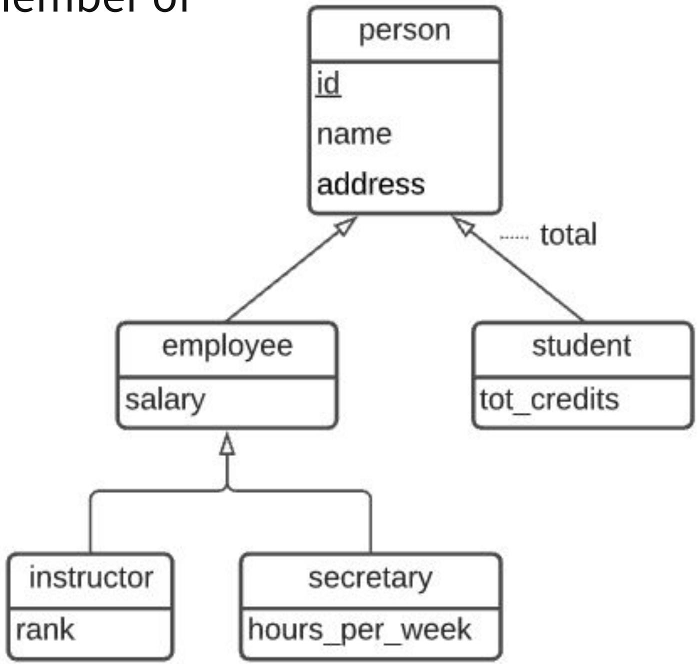
Specialization Constraints

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:

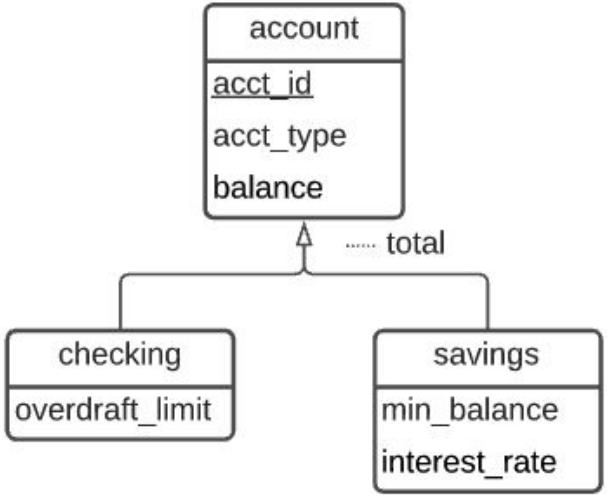
- An entity in the superclass entity-set can be a member of zero or more subclass entity-sets
- Multiple separate arrows show overlapping specialization



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

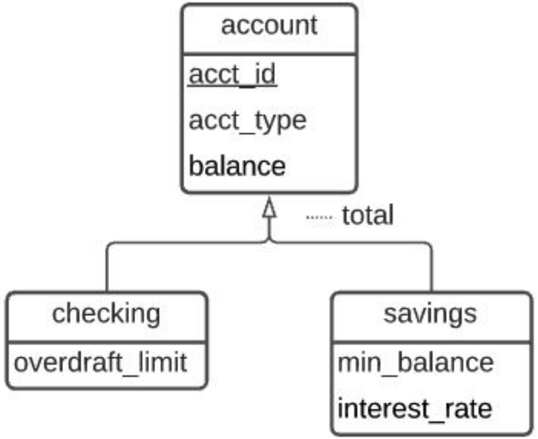


Membership constraint:

- 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*)

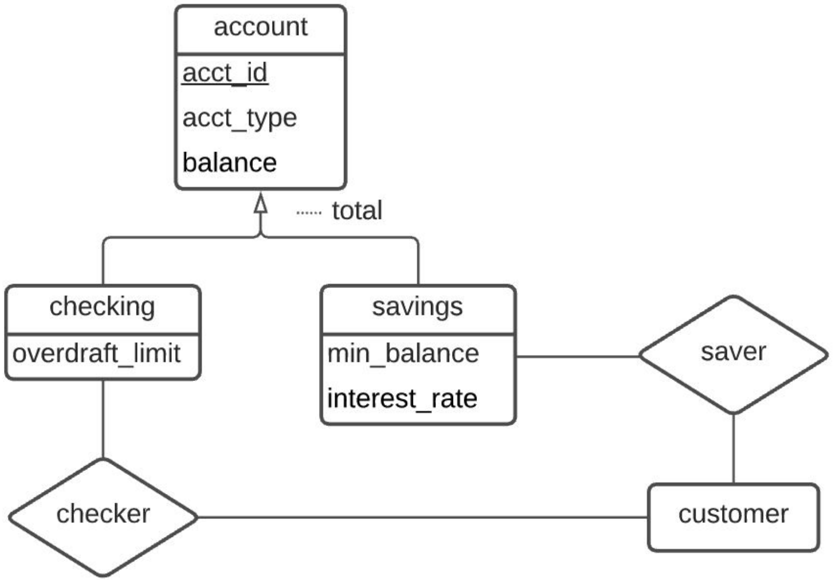
savings(<u>*acct_id*</u>, *min_balance*, *interest_rate*)

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!

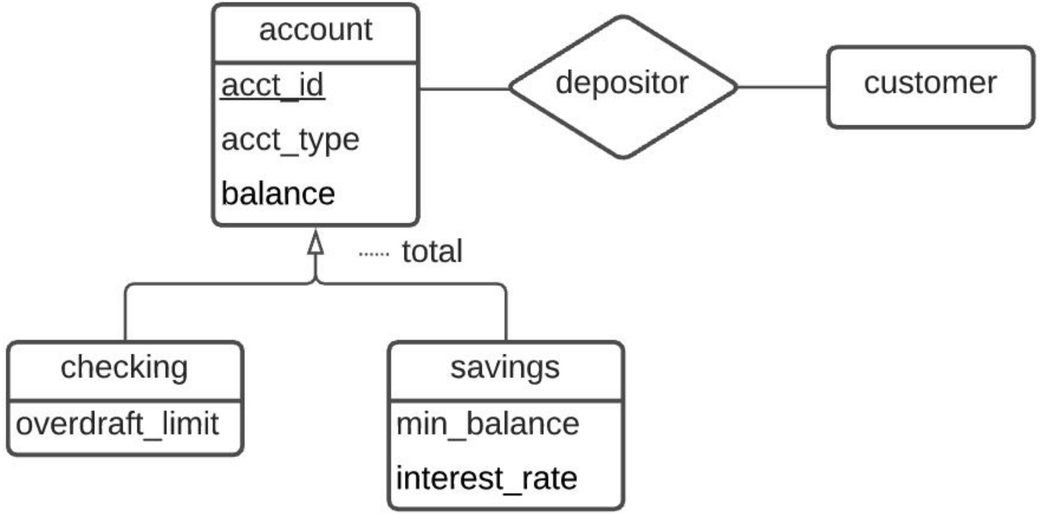


(Creates unnecessary complexity in the database schema)

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

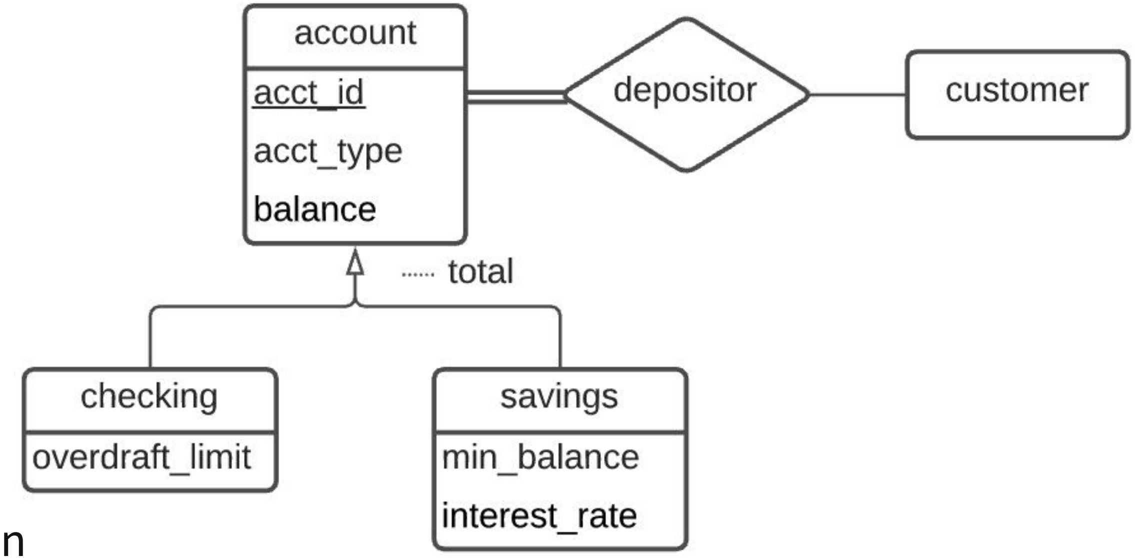


Both checking and savings accounts inherit relationships with customers.

Generalization, Relationships (3)

Finally, ask yourself:

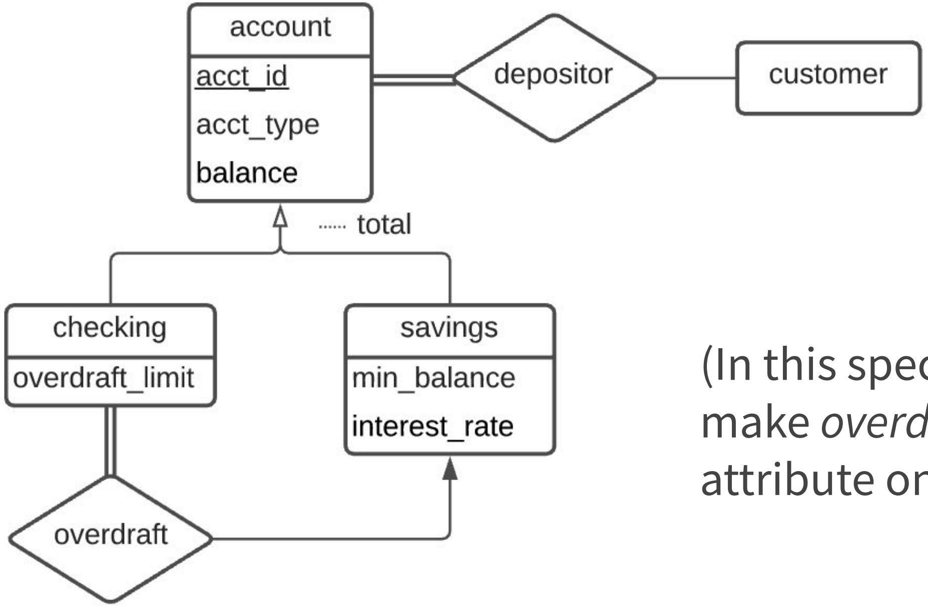
- "What constraints should I enforce on *depositor* ?"
- All accounts have to be associated with at least one customer
- A customer may have zero or more accounts
- *account* has total participation in *depositor*



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
- Don’t need to know about 5NF, join dependencies

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 relation values

- $\alpha \rightarrow \beta$ 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 $\alpha \rightarrow\rightarrow \beta$ 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 $\alpha \cap \beta = \varnothing$

- What is the key of R_1?
- $R_1(\underline{\alpha}, \beta)$

Given a relation $R_2(\alpha, \beta)$ with $\alpha \rightarrow\rightarrow \beta$ and $\alpha \cap \beta = \varnothing$

- What is the key of R_2?
- $R_2(\underline{\alpha, \beta})$ – 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 $\alpha \rightarrow\rightarrow \beta$ in D^+, where $\alpha \in R$ and $\beta \in R$, at least one of the following holds:
    - $\alpha \rightarrow\rightarrow \beta$ is a trivial multivalued dependency
    - α is a superkey for *R*
- Note: If $\alpha \rightarrow \beta$ then $\alpha \rightarrow\rightarrow \beta$

A database design is in 4NF if all schemas in the design are in 4NF