Introduction to Data Management

Entity Relationship Diagrams

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Recap – Relational Model

- SQL is parsed by the DBMS and translated into an RA plan that is more directly executable.
- Both query types work on the assumption that you are using relational data.
- The relational model specifies mechanics of how data can be organized:
  - No prescription of how data should be organized.
Goals for Today

▪ With some application in mind, we can use an entity relationship (ER) diagram to conceptualize and communicate

▪ And with an ER diagram, we can use SQL to realize the model
Outline

- Introduce Database Design
- ER Diagrams
- ER-to-SQL conversion along the way
- Integrity constraints along the way
Database Design or Logical Design or Relational Schema Design is the process of organizing data into a database model. This is done by considering what data needs to be stored and the interrelationship of the data.
The Database Design Process

Conceptual Model

Relational Model
- + Schema
- + Constraints

Conceptual Schema
- + Normalization

Physical Schema
- + Partitioning
- + Indexing
The Database Design Process

Conceptual Model

Relational Model
- + Schema
- + Constraints

Conceptual Schema
- + Normalization

Physical Schema
- + Partitioning
- + Indexing
The Database Design Process

Conceptual Model

Relational Model
- + Schema
- + Constraints

Conceptual Schema
- + Normalization

Physical Schema
- + Partitioning
- + Indexing
The Database Design Process

Conceptual Model

Relational Model
- + Schema
- + Constraints

Conceptual Schema
- + Normalization

Physical Schema
- + Partitioning
- + Indexing
Communication is Key

- Other people are involved in the design process
- Non-computer scientists have to interact with the data too
- Future users will also need to understand your data
The Future

- Your database might be around for years
- Updating the schema in production is expensive
Humans tend to be visual creatures so a visual model serves us best
These are all the blocks we will learn about:

- **Entity set**
- **Attribute**
- **Relationship**
- **Subclass**
- **Weak Entity**
Entity Sets

- An “entity set” is like a class
- An attribute is like a field
- An “entity” is like a object
  - Corresponds to a row
Entity Sets

- An “entity set” is like a class
- An attribute is like a field
- An “entity” is like an object
  - Corresponds to a row

Underline indicates the attribute is part of the primary key
Entity Sets

- An “entity set” is like a class
- An attribute is like a field
- An “entity” is like an object
  - Corresponds to a row

- Underline indicates the attribute is part of the primary key
- Every entity set should have a primary key
Entity Sets

- An “entity set” is like a class
- An attribute is like a field
- An “entity” is like a object
  - Corresponds to a row

```
CREATE TABLE Person (  
  ssn INT PRIMARY KEY,  
  name TEXT,  
  address TEXT);  
```
Integrity Constraints

An **integrity constraint** is a condition specified on a database schema that restricts the data that can be stored in an instance of the database.

- **Why?**
  - Want our application data to be consistent with our design

- **How?**
  - The DBMS checks and enforces constraints during updates
Relations

Relationship

If $A$ and $B$ are sets, then a relation $R$ is a subset of $A \times B$
Relations

**Relationship**

*If $A$ and $B$ are sets, then a relation $R$ is a subset of $A \times B$*

- **Product**:
  - Beyblade, ...
  - Trolls, ...

- **Company**:
  - Hasbro, ...
  - Nyform, ...

**Product**

- **name**
- **price**

**Company**

- **name**
- **address**
- **ceo**
If $A$ and $B$ are sets, then a relation $R$ is a subset of $A \times B$.

**Relationship**

- **Product**
  - Beyblade, ...
  - Trolls, ...

- **Company**
  - Hasbro, ...
  - Nyform, ...

**Entities**

- **Product**
  - price
  - name

- **Company**
  - name
  - address
  - ceo

**Relationship**

- makes
Relation Multiplicity

- One-to-one
- Many-to-one
- Many-to-many

Product
- Beyblade, ...
- Trolls, ...

Company
- Hasbro, ...
- Nyform, ...
Relation Multiplicity

- One-to-one
- Many-to-one
- Many-to-many

Product makes Company

- Product: Beyblade, ..., Trolls, ...
- Company: Hasbro, ..., Nyform, ...
Relation Multiplicity

- **One-to-one**
- **Many-to-one**
- **Many-to-many**

CREATE TABLE Product (
    name VARCHAR(100) PRIMARY KEY,
    ...
);
CREATE TABLE Company (
    name VARCHAR(100) PRIMARY KEY,
    ...
);
CREATE TABLE Makes (
    cname VARCHAR(100) UNIQUE REFERENCES Company,
    pname VARCHAR(100) UNIQUE REFERENCES Product,
    ...
);
Relation Multiplicity

- One-to-one
- Many-to-one
- Many-to-many

Product
- Beyblade, ...
- Trolls, ...

Company
- Hasbro, ...
- Nyform, ...

Product makes Company
Relation Multiplicity

- One-to-one
- Many-to-one
- Many-to-many

CREATE TABLE Product (
    name VARCHAR(100) PRIMARY KEY,
    ...);
CREATE TABLE Company (
    name VARCHAR(100) PRIMARY KEY,
    ...);
CREATE TABLE Makes (
    cname VARCHAR(100) UNIQUE REFERENCES Company,
    pname VARCHAR(100) UNIQUE REFERENCES Product,
    PRIMARY KEY (cname, pname),
    ...);
Relation Multiplicity

- One-to-one
- Many-to-one
- Many-to-many

Product
- Beyblade, ...
- Trolls, ...

Company
- Hasbro, ...
- Nyform, ...

Product makes Company
Relation Multiplicity

- **One-to-one**
- **Many-to-one**
- **Many-to-many**

- **Product**
  - Beyblade, ...
  - Trolls, ...

- **Company**
  - Hasbro, ...
  - Nyform, ...

Product makes Company
Relation Multiplicity

- One-to-one
- Many-to-one
- Many-to-many

Do I need a Makes table?

Product
- Beyblade, ...
- Trolls, ...

Company
- Hasbro, ...
- Nyform, ...

Product makes Company
Relation Multiplicity

- One-to-one
- **Many-to-one**
- Many-to-many

Do I need a Makes table?

Key observation: In this many-to-one relationship, each company can make many products, but each product can only be made by one company.

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beyblade, ...</td>
<td>Hasbro,</td>
</tr>
<tr>
<td>Trolls, ...</td>
<td>Nyform,</td>
</tr>
</tbody>
</table>
Relation Multiplicity

- One-to-one
- Many-to-one
- Many-to-many

Do I need a Makes table? Key observation: In this many-to-one relationship, each company can make many products, but each product can only be made by one company.

If we allow products to be made by multiple companies, we would have a many-to-many relationship.
Relation Multiplicity

- One-to-one
- Many-to-one
- Many-to-many

CREATE TABLE Company (  
  name VARCHAR(100) PRIMARY KEY,  
  ...);
CREATE TABLE Product (  
  name VARCHAR(100) PRIMARY KEY,  
  cname VARCHAR(100) REFERENCES Company  
  ...);

Product
Beyblade, ...
Trolls, ...

Company
Hasbro, ...
Nyform, ...

Product makes Company
Relation Multiplicity

- One-to-one
- Many-to-one
- Many-to-many

CREATE TABLE Company (  
  name VARCHAR(100) PRIMARY KEY,  
  ...);  
CREATE TABLE Product (  
  name VARCHAR(100) PRIMARY KEY,  
  cname VARCHAR(100) REFERENCES Company  
  ...);  

Foreign key alone is able to encode the Makes relationship

```sql
CREATE TABLE Company (  
  name VARCHAR(100) PRIMARY KEY,  
  ...);  
CREATE TABLE Product (  
  name VARCHAR(100) PRIMARY KEY,  
  cname VARCHAR(100) REFERENCES Company  
  ...);  
```

Product
- Beyblade, ...
- Trolls, ...

Company
- Hasbro, ...
- Nyform, ...

Product makes Company
Relations can have attributes too!
Exactly-One Reference

▪ Rounded arrow means the relationship is not optional (exactly one vs. at most one)

CREATE TABLE Company (  
    name VARCHAR(100) PRIMARY KEY,  
    ...);  
CREATE TABLE Product (  
    name VARCHAR(100) PRIMARY KEY,  
    cname VARCHAR(100) NOT NULL REFERENCES Company  
    ...);
Multi-Way Relations

Product

purchase

Company

Person
Multi-Way Relations

Definition of a relation generalizes!

Relationship
If $A$ and $B$ are sets, then a relation $R$ is a subset of $A \times B$
Multi-Way Relations

If $A$, $B$, and $C$ are sets, then a relation $R$ is a subset of $A \times B \times C$
Multi-Way Relations

CREATE TABLE Product (  
    name VARCHAR(100) PRIMARY KEY,  
    ...);  
CREATE TABLE Company (  
    name VARCHAR(100) PRIMARY KEY,  
    ...);  
CREATE TABLE Person (  
    ssn INT PRIMARY KEY,  
    ...);  
CREATE TABLE Purchase (  
    cname VARCHAR(100) REFERENCES Company,  
    pname VARCHAR(100) REFERENCES Product,  
    ssn INT REFERENCES Person,  
    PRIMARY KEY (cname, pname, ssn),  
    ...);
It’s Your Turn!

I want purchases to be such that a person will only buy each product from a single company.

How would you draw it?
Remember that the arrows read like an implication/function

Discuss!
I want purchases to be such that a person will only buy each product from a single company.
Do I need a Purchase table?

- Product
- Company
- Person

purchase
Do I need a Purchase table?
Probably a good idea
Now do I need a Purchase table?

- Product
- Company
- Person
Now do I need a Purchase table?

Nope.

E/R Diagrams

Multi-Way Relations

Product

Company

purchase

Person
Design Principles (common sense):

▪ Pick the right entities
▪ Don’t overcomplicate things
▪ Follow the application spec
Weak Entity Set

- A weak entity set has a key that is from another entity set

University(size, name)
Team(sport, name, uname)
Subclassing

- Distinguish special entities in an entity set
- Mimics heuristics in object oriented programming

Product

Company

is a

Toy

Candy

price

name
ceo

name

address

isChocolate

age
Subclassing

- Distinguish special entities in an entity set
- Mimics heuristics in object oriented programming

Subclasses are mutually exclusive

Product

price
name

Company

ceo
name
address

Toy

age

Candy

isChocolate

is a
Subclassing

Think of it as if: Implicitly inherits superclass attributes and key

Product

- price
- name

makes

Company

- name
- address
- ceo

Toy

- price
- name
- age

Candy

- price
- name
- isChocolate
Subclassing

Company(…)
Makes(…)
Product(price, name)

Toy(name, age)
Candy(name, isChocolate)
Misc Constraints

- Normal arrows are shorthand versions of \((\leq 1)\)
- Rounded arrows are shorthand versions of \((= 1)\)

Each product can be made by, at most, 3 companies.
Other Constraints

- **CHECK** (condition)
  - Single attribute
  - Single tuples

```sql
CREATE TABLE User (
    uid INT PRIMARY KEY,
    firstName TEXT,
    lastName TEXT,
    age INT CHECK (age > 12 AND age < 120),
    email TEXT,
    phone TEXT,
    CHECK (email IS NOT NULL OR phone IS NOT NULL)
);
```
Referential Constraint Maintenance

ON UPDATE/ON DELETE

- **NO ACTION**  □ (default) error out
- **CASCADE**  □ update/delete referencers
- **SET NULL**  □ set referencers’ field to NULL
- **SET DEFAULT**  □ set referencers’ field to default
  - Assumes default was set, e.g.

```
CREATE TABLE Table (
    id INT DEFAULT 42 REFERENCES OtherTable,
    ...
);
```
CREATE TABLE Company (  
    name VARCHAR(100) PRIMARY KEY);
CREATE TABLE Product (  
    name VARCHAR(100) PRIMARY KEY,  
    cname VARCHAR(100)  
    REFERENCES Company  
    ON UPDATE CASCADE  
    ON DELETE SET NULL);

<table>
<thead>
<tr>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>Hasbro</td>
<td>Beyblade</td>
</tr>
<tr>
<td>Nyform</td>
<td>Troll</td>
</tr>
</tbody>
</table>
CREATE TABLE Company (  
    name VARCHAR(100) PRIMARY KEY);
CREATE TABLE Product (  
    name VARCHAR(100) PRIMARY KEY,  
    cname VARCHAR(100)  
        REFERENCES Company  
        ON UPDATE CASCADE  
        ON DELETE SET NULL);

UPDATE Company  
    SET name = 'foo'  
WHERE name = 'Hasbro';

<table>
<thead>
<tr>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>Hasbro</td>
<td>Beyblade</td>
</tr>
<tr>
<td>Nyform</td>
<td>Troll</td>
</tr>
</tbody>
</table>

CREATE TABLE Company (  
    name VARCHAR(100) PRIMARY KEY);
CREATE TABLE Product (  
    name VARCHAR(100) PRIMARY KEY,  
    cname VARCHAR(100)  
        REFERENCES Company  
        ON UPDATE CASCADE  
        ON DELETE SET NULL);

UPDATE Company  
    SET name = 'foo'  
WHERE name = 'Hasbro';

<table>
<thead>
<tr>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>Hasbro</td>
<td>Beyblade</td>
</tr>
<tr>
<td>Nyform</td>
<td>Troll</td>
</tr>
</tbody>
</table>
CREATE TABLE Company (  
    name VARCHAR(100) PRIMARY KEY);
CREATE TABLE Product (  
    name VARCHAR(100) PRIMARY KEY,
    cname VARCHAR(100)  
    REFERENCES Company  
    ON UPDATE CASCADE  
    ON DELETE SET NULL);

UPDATE Company  
SET name = 'foo'  
WHERE name = 'Hasbro';

<table>
<thead>
<tr>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>foo</td>
<td>Beyblade</td>
</tr>
<tr>
<td>Nyform</td>
<td>Troll</td>
</tr>
</tbody>
</table>
CREATE TABLE Company (  
    name VARCHAR(100) PRIMARY KEY);  
CREATE TABLE Product (  
    name VARCHAR(100) PRIMARY KEY,  
    cname VARCHAR(100)  
    REFERENCES Company  
    ON UPDATE CASCADE  
    ON DELETE SET NULL);  

DELETE FROM Company  
WHERE name = 'foo';
CREATE TABLE Company (  
  name VARCHAR(100) PRIMARY KEY);
CREATE TABLE Product (  
  name VARCHAR(100) PRIMARY KEY,  
  cname VARCHAR(100)  
  REFERENCES Company  
  ON UPDATE CASCADE  
  ON DELETE SET NULL);

DELETE FROM Company  
WHERE name = 'foo';

<table>
<thead>
<tr>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nyform</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>cname</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>Troll</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>Beyblade</td>
</tr>
<tr>
<td>NULL</td>
</tr>
</tbody>
</table>
Assertions

- Hard to support
- Usually impractical
- Usually not supported
  - Simulated with triggers

CREATE ASSERTION myAssert CHECK
  (NOT EXISTS (
    SELECT Product.name
    FROM Product, Purchase
    WHERE Product.name = Purchase.prodName
    GROUP BY Product.name
    HAVING count(*) > 200));

you don’t need to study this for the class
- Triggers activate on a specified event

```sql
CREATE TRIGGER LowCredit ON Purchasing.PurchaseOrderHeader
AFTER INSERT AS
  IF (ROWCOUNT_BIG() = 0) RETURN;
  IF EXISTS (SELECT *
    FROM Purchasing.PurchaseOrderHeader AS p
    JOIN inserted AS i
    ON p.PurchaseOrderID = i.PurchaseOrderID
    JOIN Purchasing.Vendor AS v
    ON v.BusinessEntityID = p.VendorID
    WHERE v.CreditRating = 5
  )
  BEGIN
    RAISERROR ('A vendor''s credit rating is too low to accept new purchase orders.', 16, 1);
    ROLLBACK TRANSACTION;
    RETURN
  END;
GO
```

You don't need to study this for the class.
Takeaways

- ER diagrams can sketch out **high-level designs**
- Certain rules of thumb for ER-to-SQL conversions help **preserve design semantics**
- SQL allows you to make **rules specific to your application**