1 The Everything Store

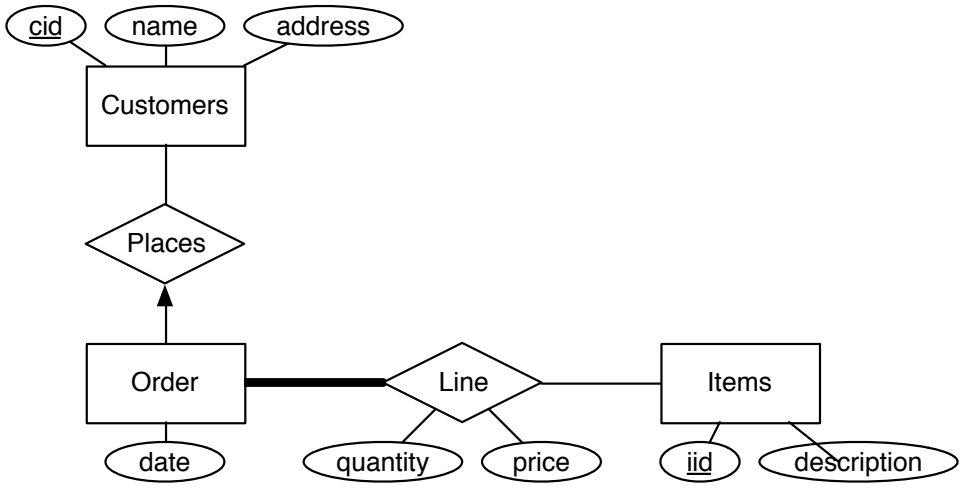
Columbia Consulting Corporation has been hired by a former hedge fund analyst to create The Everything Store, an online store that will sell everything. The analyst provides Columbia Consulting the following requirements:

- Items have a description and a price.
- Customers have a name and an address.
- When customers create an order, it may contain multiple items.

From this, they create the following SQL schema:

```
CREATE TABLE Items(
  iid int PRIMARY KEY,
  description text NOT NULL,
  price DECIMAL(8, 2) NOT NULL
);

CREATE TABLE Customers(
  cid int PRIMARY KEY,
  name text NOT NULL,
  address text NOT NULL
);

CREATE TABLE Orders(
  oid int PRIMARY KEY,
  cid INT NOT NULL REFERENCES Customers,
  date date NOT NULL
);

CREATE TABLE OrderLines(
  oid int REFERENCES Orders,
  iid int REFERENCES Items,
  quantity int NOT NULL,
  PRIMARY KEY(oid, iid)
);
```

They created this schema from the following Entity-Relationship Diagram:

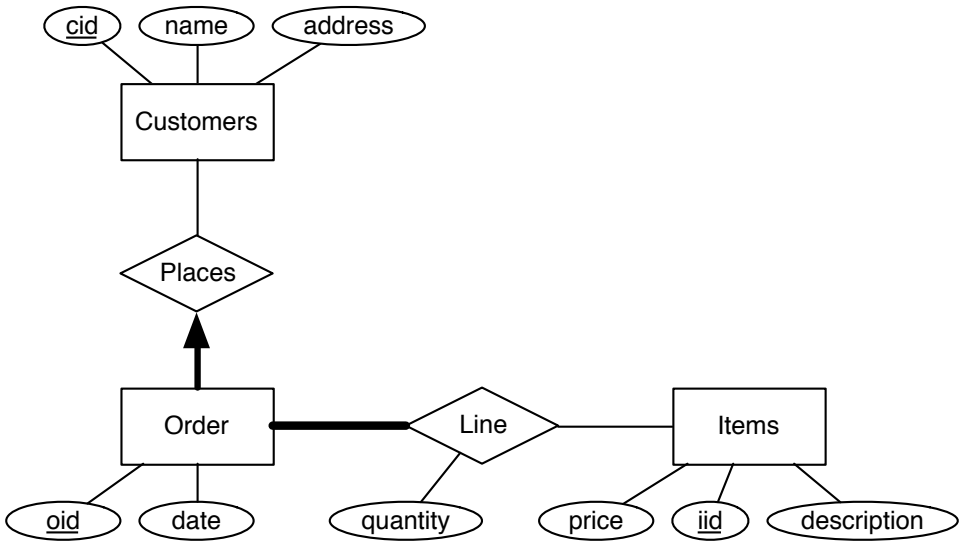


Q: What parts of the ER diagram do not match the SQL schema? For each error, provide an example that is permitted in one model but not in the other. (I found three errors, there may be more!)

- Orders: Missing a primary key.
  Example: SQL can have two orders on the same date with different ids. The ER diagram can't distinguish between these two!
- Orders must have *exactly one* customer (instead of at least one in the diagram).
  Example: Order without a customer. Not permitted in SQL.
- Line does not have a price: The price is on item.
  Example: ER allows two iPhone 6s to be sold for different prices. SQL does not permit this.

Note: The at least one constraint between Line and Order is correct. There is no easy way to represent this in SQL.

The following is a corrected diagram:



Q: List three additional CHECK constraints might make sense for this schema. Describe in one sentence why they might be useful. (I can think of at least one for each table.)

- Customers: `CHECK(length(name) > 0)` and `CHECK(length(address) > 0)`: Verify that names and addresses are not empty.
- Orders: `CHECK(date > '2015-01-01')`: Ensure that the date is within a "sane" range, and isn't too far in the past (e.g. before the store existed?). Add a constraint relative to the current time might also make sense, e.g. `CHECK(date >= now())`. However, that won't let you create orders from yesterday that were missed, and it might have issues when you update old orders.
- Items: `CHECK(price > 0)`: Prevent free or negative price items.
  `CHECK(length(description) > 0)`: Prevent empty descriptions.
- OrderLines: `CHECK(quantity > 0)`: Zero or negative number of items does not make sense.

Note: The at least one constraint between Line and Order is correct. There is no easy way to represent this in SQL.

Write SQL queries to answer the following questions. You can go to http://w4111db.eastus.cloudapp.azure.com:8111/orders to run queries on some sample data, or use the psql command with your project account: `psql -U (uni) -h w4111db.eastus.cloudapp.azure.com orders`. I *strongly* recommend trying to write the queries on paper first, then checking on the computer. This is better practice for exams.

Q: Compute the total number of items and the total dollar value of all items sold.

```
SELECT SUM(ol.quantity), SUM(ol.quantity*i.price)
FROM OrderLines ol, Items i
WHERE ol.iid = i.iid;
```

Q: What are the names of customers who purchased a Nexus 5X?

```
SELECT DISTINCT c.name
FROM Customers c, Orders o, OrderLines ol, Items i
WHERE i.description = 'Nexus 5X' AND
  c.cid = o.cid AND o.oid = ol.oid AND ol.iid = i.iid;
```

Note: You need `DISTINCT` to eliminate duplicates. Otherwise, if a customer has purchased a Nexus 5X multiple times, their name will be listed multiple times. You can also use `GROUP BY c.name` instead of DISTINCT. With both queries: if there are two customers with the same name who both purchased a Nexus 5X, the name will only be listed once. I think this is the best interpretation of this specific question, which asks explicitly for the *names*. If you want to know the *customers* who purchased a Nexus 5X, it would be best to display `DISTINCT cid, name`.

Q: Compute the products that are purchased together the most often. Rank the items by the number of times the two items appear together in the same order, ignoring the quantity. For example, if an order contains 100 "iPhone 6" and 1000 "MacBook Pro", that counts the same as an order that has 1 iPhone 6 and 1 MacBook Pro.

```
SELECT ol1.iid, ol2.iid, count(*)
FROM OrderLines ol1, OrderLines ol2
WHERE ol1.oid = ol2.oid AND ol1.iid < ol2.iid
GROUP BY ol1.iid, ol2.iid
ORDER BY count(*) DESC
LIMIT 1;
```

Q: For each item, compute the number of separate customers who have purchased each one.

```
SELECT i.iid, count(DISTINCT o.cid)
FROM Items i
  LEFT OUTER JOIN OrderLines ol ON i.iid = ol.iid
  LEFT OUTER JOIN Orders o ON ol.oid = o.oid
GROUP BY i.iid;
```

Q: For each item, compute the number of orders that include that item, the total quantity purchased, and the total amount charged for them.

Close but omits items that are never ordered:

```
SELECT i.iid, i.description, count(*),
  sum(ol.quantity), sum(ol.quantity * i.price)
FROM Items i, OrderLines ol
WHERE i.iid = ol.iid
GROUP BY i.iid;
```

Use `LEFT OUTER JOIN` to get closer, but this has 1 for the count of unordered items instead of 0:

```
SELECT i.iid, i.description, count(*),
  sum(ol.quantity), sum(ol.quantity * i.price)
FROM Items i LEFT OUTER JOIN OrderLines ol ON i.iid = ol.iid
GROUP BY i.iid;
```

To fix that, count a specific column (e.g. `ol.quantity`). This would get full credit on a homework or exam, but shows null instead of 0 for the sums. Optional: to fix that, you can use the `COALESCE` function (see the Postgres documentation):

```
SELECT i.iid, i.description, count(ol.quantity),
  COALESCE(sum(ol.quantity), 0), COALESCE(sum(ol.quantity * i.price), 0)
FROM Items i LEFT OUTER JOIN OrderLines ol ON i.iid = ol.iid
GROUP BY i.iid;
```

Q: Compute the average number of orders per customer.

```
SELECT COUNT(DISTINCT o.oid) / COUNT(DISTINCT c.cid)::real
FROM Customers c, Orders o
```

Note: The type conversion from integer to real (`::real`) is required to compute the fractional component (rather than using integer division). I would accept the version without the type conversion for full credit on a homework or exam. An alternative version:

```
SELECT b.y / a.x::real
FROM
  (SELECT COUNT(*) x FROM Customers) a,
  (SELECT COUNT(*) y FROM Orders) b;
```

You can also use `AVG` to compute the answer, but I find this to be a bit more complicated. You must use `COUNT(o.oid)` to ensure you get zero for customers without orders. If you use `COUNT(*)`, it will count the number of rows, which with a `LEFT OUTER JOIN` is 1 for a customer without orders, even though the values contain NULL. The correct alternative version:

```
SELECT AVG(orderCount.count) FROM (
  SELECT COUNT(o.oid) count
  FROM Customers c LEFT OUTER JOIN Orders o ON o.cid = c.cid
  GROUP BY c.cid
) orderCount;
```

Q: Which of these questions can be answered in the basic relational algebra model taught in class? Write a relational algebra expression to compute it.

There is only one that does not include an aggregate: Customers who purchased a Nexus 5X:
$\pi_{name}(Customers \bowtie_{cid} Orders \bowtie_{oid} OrderLines \bowtie_{iid} (\sigma_{description='Nexus5X'} Items))$

Optional Q: This question is (in my opinion) harder than questions on the midterm. I was not able to write this without multiple attempts. For each customer, display the customer id and name, the total number of items ordered by that customer, and the total cost of those items for each customer.

Hint: What intermediate queries could you write to get you part of the way towards this? (Everyone should be able to write a query to get you part of the way towards this answer.)

Intermediate Q: For each customer, display all order lines

We need to use LEFT OUTER JOIN to include customers without orders, and to include the "weird" order that doesn't have any lines/items in it:

```
SELECT *
FROM Customers c
  LEFT OUTER JOIN Orders o ON c.cid = o.cid
  LEFT OUTER JOIN OrderLines ol ON o.oid = ol.oid;
```

Now we can work on aggregating some of these columns:

Intermediate Q: For each customer, display the customer name, id and the total number of items ordered.

This produces null instead of zero for the customer without orders, which I would consider correct, but we can use `COALESCE` to fix this (see above for details about this function).

```
SELECT c.cid, c.name, SUM(ol.quantity)
FROM Customers c
  LEFT OUTER JOIN Orders o ON c.cid = o.cid
  LEFT OUTER JOIN OrderLines ol ON o.oid = ol.oid
GROUP BY c.cid;
```

Final answer: Add items and sum quantity * price. Use `COALESCE` to produce 0 instead of NULL.

```
SELECT c.cid, c.name, COALESCE(SUM(ol.quantity), 0) as "total items",
  COALESCE(SUM(ol.quantity * i.price), 0) as "total value"
FROM Customers c
  LEFT OUTER JOIN Orders o ON c.cid = o.cid
  LEFT OUTER JOIN OrderLines ol ON o.oid = ol.oid
  LEFT OUTER JOIN Items i ON ol.iid = i.iid
GROUP BY c.cid;
```

2 The Oscars

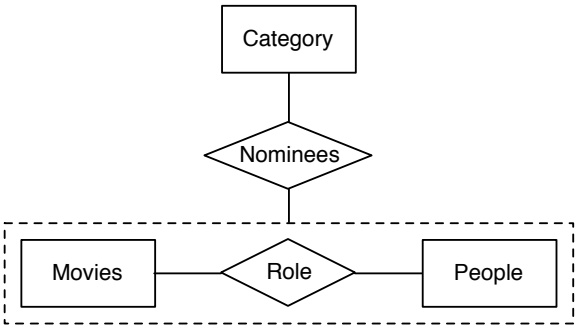
Columbia Consulting Corporation has created a database to power the official Oscars web site (e.g. http://oscar.go.com/nominees). The want to model the acting awards, which have the following facts:

- Movies have a title. It is possible for multiple movies to have the same title (e.g. they get remade).
- People have names. It is possible for multiple people to have the same name.
- The Oscars has award categories with names like "Actress in a Leading Role" and "Actor in a Supporting Role".
- Each actor in a movie plays a role with a name, such as "Steve Jobs".
- For an award category in a year, there a set of people are nominated. These people are nominated for playing a specific role.
- For each award category in a year, one of the nominees can be marked as a winner.

To model this database, Columbia Consulting created the following database schema:

```
CREATE TABLE Movies(
  mid int PRIMARY KEY,
  title text NOT NULL
);

CREATE TABLE Category(
  cid int PRIMARY KEY,
  name text NOT NULL
);

CREATE TABLE People(
  pid int PRIMARY KEY,
  name text NOT NULL
);

CREATE TABLE Roles(
  pid int REFERENCES People,
  mid int REFERENCES Movies,
  name text NOT NULL,
  PRIMARY KEY(pid, mid)
);

CREATE TABLE Nominees(
  pid int,
  mid int,
  cid int REFERENCES Category,
  year int NOT NULL,
  won bool NOT NULL,
  PRIMARY KEY(pid, mid, cid),
  FOREIGN KEY (pid, mid) REFERENCES Roles
);
```

Q: Draw an entity-relationship diagram for this database.



Note: At least one constraints cannot be expressed in SQL. For this application, there are many cases where that would be acceptable:

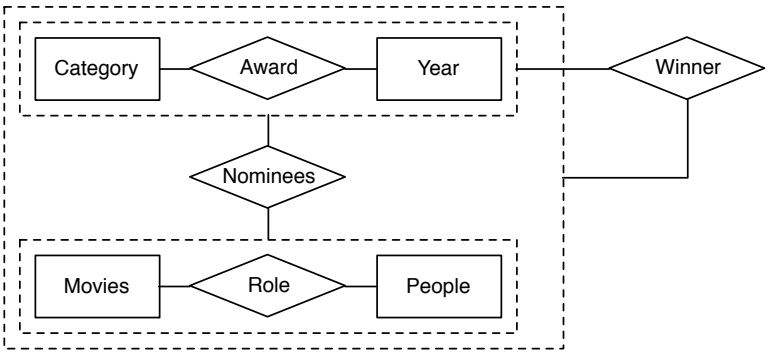
- Category and Nominees (e.g. every category must have at least one nominee).
- People and Role (e.g. every person must be in at least one movie).
- Movies and Role (e.g. every movie must have at least one role).

It is *incorrect* to have an at least one constraint between Nominees and Role. That would mean every Role must be nominated for at least one category, which is not true. It is also incorrect to have an at most one constraint between Role and Nominees (see the answer to the next question below).

Q: Describe one real world constraint for the acting awards that is not modeled by this SQL schema. Provide an example of how this is violated. There are at least two answers for this, there are probably others! **Optional**: Describe how you could correct the SQL schema to fix this problem. How does that change the ER diagram? *Note*: Finding the problems should be relatively easy. Fixing these problems is very challenging, meaning probably harder than the midterm.

- A role can only be nominated for a single award. E.g. this database allows an actor in a movie to be nominated for both best supporting actor and best leading actor. To fix this: combine Nominees into the Roles table. This means Roles will have `nominated int REFERENCES Category, year int, won bool` columns added to it. The Entity-Relationship diagram would have an at most one constraint.
- No restriction on the number of winners for an award. E.g. this database allows everyone to be marked a winner. There are two ways to fix this:
  1. Add a trigger to forbid updates on the `won` column where the following query would be true: `SELECT COUNT(won)>1 FROM Nominees WHERE year=x AND category=y AND won`
  2. Create a separate Winners table, with `PRIMARY KEY(cid, year)`, and `FOREIGN KEY (pid, mid, cid, year) REFERENCES Nominees`. The primary key constraint enforce one winner per (category, year).

An E-R diagram that enforces one award per (category, year) and that you can only give the award to one person who was nominated for that specific award might look like the following.



Write SQL queries to answer the following questions. You can go to http://w4111db.eastus.cloudapp.azure.com:8111/oscars to run queries on some sample data, or use the psql command with your project account: `psql -U (uni) -h w4111db.eastus.cloudapp.azure.com oscars`. I *strongly* recommend trying to write the queries on paper first, then checking on the computer. This is better practice for exams.

Q: Find the names of all movies nominated in 2016 in the database.

```
SELECT DISTINCT m.title
FROM Movies m, Nominees n
WHERE m.mid = n.mid;
```

Q: Find the people nominated for "Actress in a Leading Role" in 2016 with the movie name, the name of their role, and if they won. (Also: try querying for all combinations of Actor/Actress and Leading/Supporting roles)

```
SELECT p.name, m.title, r.name, n.won
FROM People p, Roles r, Movies m, Nominees n, Category c
WHERE c.name = 'Actress in a Leading Role' AND n.year = 2016
  AND p.pid = r.pid AND r.mid = m.mid AND m.mid = n.mid
  AND n.cid = c.cid AND n.pid = p.pid;
```

The results for the other categories are generated by changing the category name.

Q: Find the year, award category, movie, and winning status of all of Jennifer Lawrence's nominations in order of oldest to most recent.

```
SELECT n.year, c.name, m.title, n.won
FROM People p, Nominees n, Category c, Movies m
WHERE p.name = 'Jennifer Lawrence'
  AND p.pid = n.PID AND n.cid = c.cid AND n.mid = m.mid
ORDER BY n.year;
```

Q: Find the name and number of nominations for each movie in the database.

```
SELECT m.title, count(n.mid)
FROM Movies m LEFT OUTER JOIN Nominees n
  ON n.mid = m.mid
GROUP BY m.mid, m.title;
```

Q: Find the name and number of wins for each movie in the database.

```
SELECT m.title, count(wins.mid)
FROM Movies m LEFT OUTER JOIN (
  SELECT n.mid
  FROM Nominees n
  WHERE n.won
) as wins
  ON m.mid = wins.mid
GROUP BY m.mid, m.title;
```

A simpler version uses the fact that you can convert the boolean `won` to an integer and sum it:

```
SELECT m.title, SUM(n.won::int)
FROM Movies m, Nominees n
WHERE m.mid = n.mid
GROUP BY m.mid, m.title
```

Q: What is the maximum number of nominations for any movie?

```
SELECT max(count) FROM (
  SELECT m.mid, count(*) as count
  FROM Movies m, Nominees n
  WHERE n.mid = m.mid
  GROUP BY m.mid
) counts;
```

The following questions are (in my opinion) harder than questions on the midterm. I was not able to write these without multiple attempts.

Optional Q: Find the name, number of nominations and number of wins for each movie.

This combines the previous answers for nominations and wins into a single query, by nesting those answers in the FROM clause, and joining them in the WHERE clause.

```
SELECT m.title, nominees.count, wins.count
FROM Movies m, (
  SELECT m.mid, count(n.mid) as count
```

```
  FROM Movies m LEFT OUTER JOIN Nominees n
    ON n.mid = m.mid
  GROUP BY m.mid
) as nominees, (
  SELECT m.mid, count(wins.mid) as count
  FROM Movies m LEFT OUTER JOIN (
    SELECT n.mid
    FROM Nominees n
    WHERE n.won
  ) as wins
    ON m.mid = wins.mid
  GROUP BY m.mid
) as wins
WHERE m.mid = nominees.mid AND m.mid = wins.mid;
```

Optional Q: Find the name(s) of the movie(s) with the most nominations in the database.

This uses the nominations count query from a previous answer as a nested query. I used `WITH` to make it easier to read. The ideas is as follows:

1. Compute the maximum nomination count value in the database.
2. Find the id for all movies with a count equal to the maximum nomination count.
3. Join with the movies table to get the titles.

```
WITH NominationCounts(mid, count) AS (
  SELECT m.mid, count(*) as count
  FROM Movies m, Nominees n
  WHERE n.mid = m.mid
  GROUP BY m.mid
)
SELECT m.title
FROM Movies m, NominationCounts n1, (
  SELECT max(count) max FROM NominationCounts
) max
WHERE m.mid = n1.mid AND n1.count = max.max;
```

Optional Q: There are additional facts for the Oscars that are not modelled in this database:

- Some awards can be given to groups of people. For example, the Best Picture awards are given to the producers of the film, which is frequently a group.
- A category can include single people as well as groups. For example, in 2016 the nominees for the category of Documentary (Short Subject) included Adam Benzine for a film called "Claude Lanzmann: Spectres of the Shoah", as well as groups, such as Courtney Marsh and Jerry Franck for "Chau, Beyond the Lines".
- People can be nominated for multiple films in one category. For example, in 2016 Sandy Powell was nominated for both Carol and Cinderella in the category of Costume Design.
- Some categories credit people slightly differently. For example, in the category for Music (Original Song), some people are credited separately for the Music and others for the Lyrics. The Writing categories do something similar.

Modify the the entity-relationship diagram and the SQL schema to be able to model these facts.

Note: I don't have solutions to this because I didn't actually finish this. I had intended to do it, but ran out of time.