

MySQL Workshop

Scott D. Anderson

Workshop Plan

- Part 1: Simple Queries
- Part 2: Creating a database
- Part 3: Joining tables
 - joining tables,
 - one-to-many and many-to-many relationships
 - foreign keys
- Part 4: complex queries with groups, subqueries and sorting
- Reference: <https://cs.wellesley.edu/~cs304/mysql-workshop/>



Files for this part

Do this with me. We'll use these files in this session

```
$ curl -O https://cs.wellesley.edu/~cs304/mysql-workshop/part3.tar  
$ tar xf part3.tar  
$ cd part3  
$ ls
```

Use the `c9-before-part3.sql` file if you need to start over.



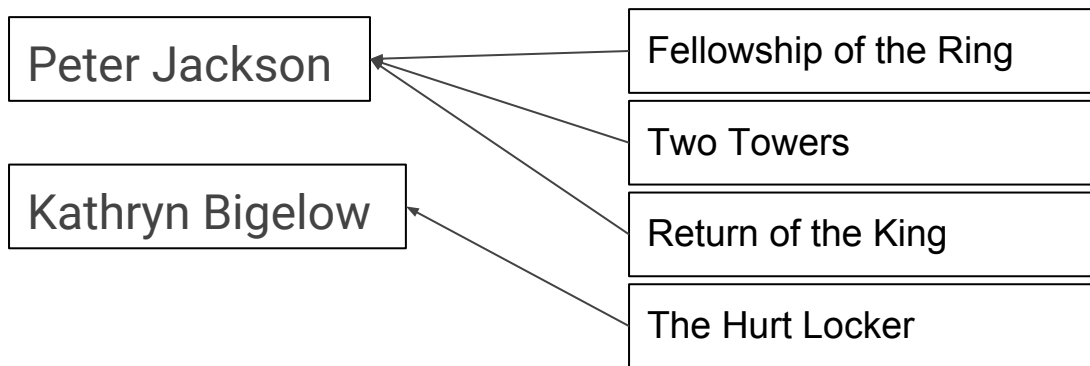
Joins

- A **join** connects one table with another, say **pet** with **owner**
- A **join** connects a row in one table with a row in another, say Crookshanks with Hermione Granger
- These connections can be
 - one-to-one (rare),
 - one-to-many (common) or
 - many-to-many (common)



Movie Directors

- In the Wellesley Movie Database (WMDB), a movie has only one director, but a person can direct more than one movie.
- This is a one-to-many (1:N) relationship.



Foreign Keys

- We can implement a 1:N relationship by storing the key for the person (director) in the row for that movie in the movie table.

nm	name
1392	Peter Jackson
941	Kathryn Bigelow

tt	director	title
120737	1392	Fellowship of the Ring
167261	1392	Two Towers
167262	1392	Return of the King
887912	941	The Hurt Locker

Backwards Representation!

- Compared to what we are used to in programming, this is **backwards**!
- We would usually represent a person and have a **list** of the movies they directed stored with the person.
- We don't have **lists** in relational databases.
- Here, the person has no list, but each movie as one director id
- The director id is a **foreign key**: a key in some **other** table
- Notice director can't be a key in the movie table, because there are **repeats**
- Director **can** be a key in the person table, since it's their ID (NM)



Queries with One to Many relationships

- Imagine we have a query that selects from two tables at the same time:

```
SELECT * FROM person, movie LIMIT 10;
```

- This pairs every person with every movie. (A **cross-product**.)
- But, if we use a WHERE clause to choose just the pairs we want, we get sensible results:

```
SELECT * FROM person, movie  
WHERE nm = director;
```



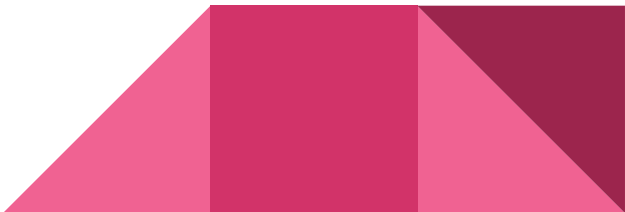
Looking up Movie Directors

- `director1.sql` lists all movie,director pairs:

```
select * from person,movie  
where nm = director;
```

- `director2.sql` gives our example with Peter Jackson and Kathryn Bigelow:

```
select * from person,movie  
where nm = director  
and nm in (941,1392);
```



Aside about Cross Products

A Cross Product is every possible pair. If set $A = \{a,b,c\}$ and set $B = \{1,2,3\}$, then

$$A \times B = \{(a,1), (a,2), (a,3), \\ (b,1), (b,2), (b,3), \\ (c,1), (c,2), (c,3)\}$$

That 9 pairs! (Why 9? 3×3).

Let's look at `dancers.sql`, load it, and then do

```
select * from leader, follower;
```



Aside continued: Self-Crosses

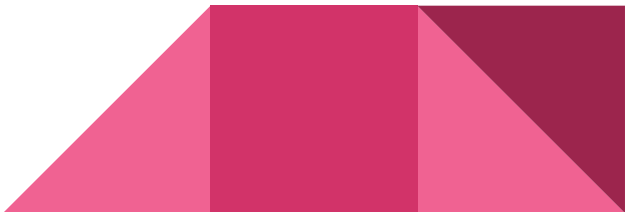
We can even cross a table with itself. See/run `flavor-table.sql` and `flavor-pairs1.sql`:

```
select * from flavor a, flavor b;
```

```
select a.name as 'scoop1', b.name as 'scoop2';
```

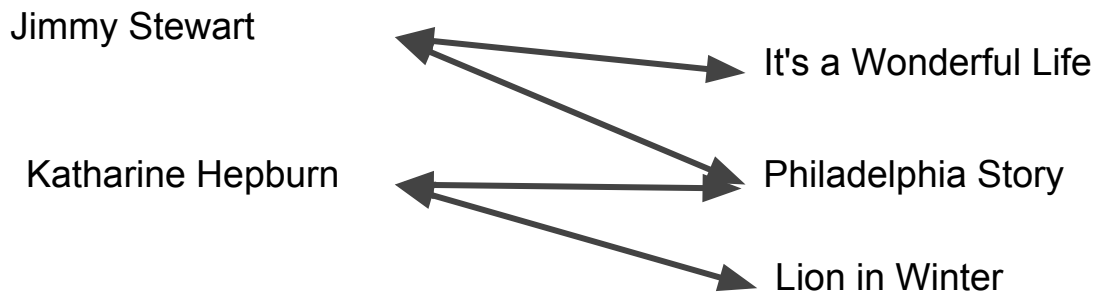
Most of the time, the join is perfectly intuitive:

```
select title,name from movie, person  
where movie.director=person.nm;
```



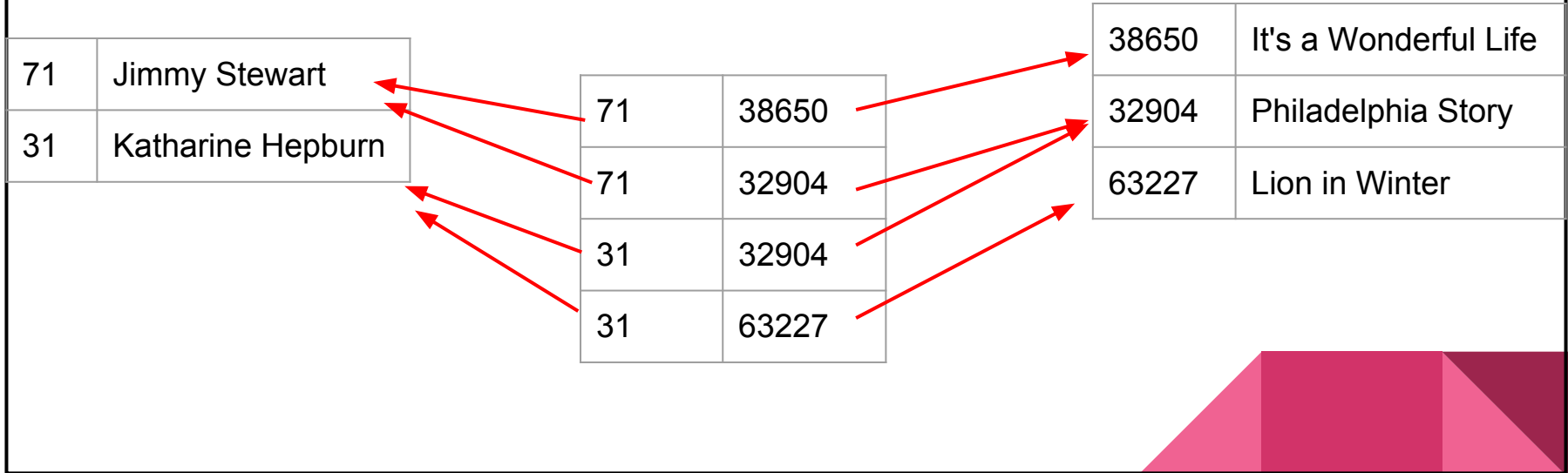
Many to Many Relationships

- These are harder. Let's start with an example:



Many to Many with an intermediate table

Create an intermediate table with **pairs of foreign keys**. A pair (P,M) means that **person P acted in movie M**.



Other attributes in intermediate table

A pair (P,M) means that **person P acted in movie M**. You can add extra information that belongs just to the **pair**, such as the name of the character.

71	Jimmy Stewart
31	Katharine Hepburn

71	38650	George Bailey
71	32904	Macaulay Connor
31	32904	Tracy Lord
31	63227	Eleanor of Aquitaine

38650	It's a Wonderful Life
32904	Philadelphia Story
63227	Lion in Winter



The WMDB credit table

An "acting credit" is just a pair, (P,M) saying that person P acted in movie M.

They get their name in the credits at the end of the movie.

They also get an entry in the credit table in WMDB.

Take a moment to look at it now:

```
mysql> describe credit;  
mysql> select * from credit limit 5;
```



Defining the CREDIT table

```
CREATE TABLE credit (  
    nm int,  
    tt int,  
    primary key (nm,tt)  
);
```

Note that the key has **both** columns, because neither will be unique, but the **pair** is.

See `credit-table.sql`



Looking up Acting Credits

To use the credit table, we employ the cross-product idea again, with both the movie and person tables.

This time, though, the NM and TT fields are ambiguous, because they appear in both tables. So we have to use the table name in the expression:

```
SELECT * from person, credit, movie  
WHERE person.nm = credit.nm and credit.tt = movie.tt;
```

try credits1.sql **and** credits2.sql **and** credits3.sql



Special Syntax for Joins

Joins are so common and important that there is special syntax for them.

Instead of just putting a comma between the joined tables, you can either specify a join condition (ON) or a shared column (USING):

```
select * from person inner join movie on (nm=director);
```

```
select * from person inner join credit using (nm);
```

An "inner join" just means to omit rows where there's no match.



Exercise: Vet office

- We have a draft of pet and owner tables.
- Add a foreign key to the pet table referring to the pet's owner. Note that it will *not* be auto_increment!
- Insert some data in both tables. You'll need to know the owner ID to put it in the pet's row
- Write some queries, say:
 - list all pets and owner names
 - list all pets and owners where the pet is a dog under 50 pounds



Solution

My solution is in `pet-and-owner.sql`

Obviously, your solution will be different.



Modeling Question

- In part 1, we said that the value in a column is not a list.
- So, there's no "list of courses" column for Hermione saying that she's taking ["arithmancy", "potions", "transfiguration", ...]
- How would we represent this information in a Relational Database?

Discuss this with an "elbow partner" and come up with an answer.



Modeling Solution

- Each person takes many courses
- Each course is taken by many people
- There's a many-to-many relationship between these entities
- Create an intermediate table, say called "**takes**" that has two foreign keys:
 - BID for the person
 - CRN for the course



Summary of Part 3

- **Joins** are essential in relational databases.
- A 1:N relationship can be implemented by storing the foreign key for the "one" in each row of the "many."
- Think of our "director" example.
- A N:N relationship can be implemented with an intermediate table that stores pairs of foreign keys.
- Think of our "acting credit" example, using the credit table.
- Special syntaxes:
 - A INNER JOIN B USING (shared-col)
 - A INNER JOIN B ON (expr)

