CSE3026: Web Application Development
Relational Databases and SQL

Scott Uk-Jin Lee

13.1: Database Basics

- **13.1: Database Basics**
- 13.2: SQL
- 13.3: Multi-table Queries
- 13.4: Databases and PHP

Relational databases

- **relational database**: A method of structuring data as tables associated to each other by shared attributes.
- a table row corresponds to a unit of data called a **record**; a column corresponds to an attribute of that record
- relational databases typically use **Structured Query Language** (SQL) to define, manage, and search data

Why use a database?

- **powerful**: can search it, filter data, combine data from multiple sources
- **fast**: can search/filter a database very quickly compared to a file
- **big**: scale well up to very large data sizes
- **safe**: built-in mechanisms for failure recovery (e.g. **transactions**)
- **multi-user**: concurrency features let many users view/edit data at same time
- **abstract**: provides layer of abstraction between stored data and app(s)
    - many database programs understand the same SQL commands

Database software

- Oracle
- Microsoft SQL Server (powerful) and Microsoft Access (simple)
- PostgreSQL (powerful/complex free open-source database system)
- SQLite (transportable, lightweight free open-source database system)
- MySQL (simple free open-source database system)
  - many servers run "LAMP" (Linux, Apache, MySQL, and PHP)
  - Wikipedia is run on PHP and MySQL
  - we will use MySQL in this course

Example `simpsons` database

students

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

teachers

id	name
1234	Krabappel
5678	Hoover
5238	Lee

courses

id	name	teacher_id
10001	Computer Science 142	1234
10002	Computer Science 143	5678
10003	Computer Science and Engineering 326	5238
10004	Informatics 100	1234

grades

student_id	course_id	grade
123	10001	B-
123	10002	C
456	10001	B+
888	10002	A+
888	10003	A+
404	10004	D+

- to test queries on this database, download and import `simpsons.sql`
  1. `C:\MAMP\Library\bin\mysql>mysql -uroot -proot`
  2. `mysql>CREATE DATABASE simpsons;`
  3. `mysql>exit`
  4. `C:\MAMP\Library\bin\mysql>mysql -uroot -proot simpsons < simpsons.sql`

Example `world` database

countries (Other columns: **region**, **surface_area**, **life_expectancy**, **gnp_old**, **local_name**, **government_form**, **capital**, **code2**)

code	name	continent	independence_year	population	gnp	head_of_state	...
AFG	Afghanistan	Asia	1919	22720000	5976.0	Mohammad Omar	...
NLD	Netherlands	Europe	1581	15864000	371362.0	Beatrix	...
...	...	...	...	...	...	...	...

cities

id	name	country_code	district	population
3793	New York	USA	New York	8008278
1	Los Angeles	USA	California	3694820
...	...	...	...	...

languages

country_code	language	official	percentage
AFG	Pashto	T	52.4
NLD	Dutch	T	95.6
...	...	...	...

- to test queries on this database, download and import `world.sql`

Example `imdb` database

actors

id	first_name	last_name	gender
433259	William	Shatner	M
797926	Britney	Spears	F
831289	Sigourney	Weaver	F
…			

movies

id	name	year	rank
112290	Fight Club	1999	8.5
209658	Meet the Parents	2000	7
210511	Memento	2000	8.7
…			

roles

actor_id	movie_id	role
433259	313398	Capt. James T. Kirk
433259	407323	Sgt. T.J. Hooker
797926	342189	Herself
…		

movies_genres

movie_id	genre
209658	Comedy
313398	Action
313398	Sci-Fi
…	

directors

id	first_name	last_name
24758	David	Fincher
66965	Jay	Roach
72723	William	Shatner
…		

movies_directors

director_id	movie_id
24758	112290
66965	209658
72723	313398
…	

- also available, `imdb_small.sql` with fewer records (for testing queries)
- to test queries on this database, download and import `imbd_small.sql`

13.2: SQL

- 13.1: Database Basics
- **13.2: SQL**
- 13.3: Multi-table Queries
- 13.4: Databases and PHP

SQL basics

```sql
SELECT name FROM cities WHERE id = 17;
```

```sql
INSERT INTO countries VALUES ('SLD', 'ENG', 'T', 100.0);
```

- **Structured Query Language (SQL)**: a language for searching and updating a database
- a standard syntax that is used by all database software (with minor incompatiblities)
  - generally case-insensitive
- a **declarative** language: describes what data you are seeking, not exactly how to find it

Issuing SQL commands directly in MySQL

```sql
SHOW DATABASES;
USE database;
SHOW TABLES;
```

- open command promt and go to `MAMP\Library\bin\mysql`, then type:

```
C:\MAMP\Library\bin\mysql> mysql -u root -p
Password:
Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> USE world;
Database changed

mysql> SHOW TABLES;
+-----------+
| cities    |
| countries |
| languages |
+-----------+
3 rows in set (0.00 sec)
```

The SQL `SELECT` statement

```SQL
SELECT column(s) FROM table;
```

```SQL
SELECT name, code FROM countries;
```

name	code
China	CHN
United States	IND
Indonesia	USA
Brazil	BRA
Pakistan	PAK
…	…

- the `SELECT` statement searches a database and returns a set of results
  - the column name(s) written after `SELECT` filter which parts of the rows are returned
  - table and column names are case-sensitive (default on Linux MySQL & depends on setup)
  - `SELECT * FROM table;` keeps all columns

The `DISTINCT` modifier

```SQL
SELECT DISTINCT column(s) FROM table;
```

```SQL
SELECT language
FROM languages;
```

language
Dutch
English
English
Papiamento
Spanish
Spanish
Spanish
…

```SQL
SELECT DISTINCT language
FROM languages;
```

language
Dutch
English
Papiamento
Spanish
…

- eliminates duplicates from the result set

The `WHERE` clause

```
SELECT column(s) FROM table WHERE condition(s);
```

```
SELECT name, population FROM cities WHERE country_code = "FSM";
```

name	population
Weno	22000
Palikir	8600

- `WHERE` clause filters out rows based on their columns' data values
- in large databases, it's critical to use a `WHERE` clause to reduce the result set size
- suggestion: when trying to write a query, think of the `FROM` part first, then the `WHERE` part, and lastly the `SELECT` part

More about the `WHERE` clause

```
WHERE column operator value(s)
```
SQL

```
SELECT name, gnp FROM countries WHERE gnp > 2000000;
```
SQL

code	name	gnp
JPN	Japan	3787042.00
DEU	Germany	2133367.00
USA	United States	8510700.00
…	…	…

- the `WHERE` portion of a SELECT statement can use the following operators:
  - `=`, `>`, `>=`, `<`, `<=`
  - `<>` : not equal
  - `BETWEEN` *min* `AND` *max*
  - `LIKE` *pattern*
  - `IN` (*value*, *value*, …, *value*)

Multiple `WHERE` clauses: AND, OR

```
SELECT * FROM cities
WHERE country_code = 'USA' AND population >= 2000000;
```
SQL

id	name	country_code	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
…	…	…	…	…

- multiple `WHERE` conditions can be combined using `AND` and `OR`

Approximate matches: `LIKE`

```
WHERE column LIKE pattern
```

SQL

```
SELECT code, name, population FROM countries WHERE name LIKE 'United%';
```

SQL

code	name	population
ARE	United Arab Emirates	2441000
GBR	United Kingdom	59623400
USA	United States	278357000
UMI	United States Minor Outlying Islands	0

- `LIKE '`*text*`%'` searches for text that starts with a given prefix
- `LIKE '%`*text*`'` searches for text that ends with a given suffix
- `LIKE '%`*text*`%'` searches for text that contains a given substring

Sorting by a column: `ORDER BY`

```
ORDER BY column(s)
```

```
SELECT code, name, population FROM countries
WHERE name LIKE 'United%' ORDER BY population;
```

code	name	population
UMI	United States Minor Outlying Islands	0
ARE	United Arab Emirates	2441000
GBR	United Kingdom	59623400
USA	United States	278357000

- can write `ASC` or `DESC` to sort in ascending (default) or descending order:

```
SELECT * FROM countries ORDER BY population DESC;
```

- can specify multiple orderings in decreasing order of significance:

```
SELECT * FROM countries ORDER BY population DESC, gnp;
```

- see also: `GROUP BY`

Limiting rows: `LIMIT`

```
LIMIT number
```
SQL

```
SELECT name FROM cities WHERE name LIKE 'K%' LIMIT 5;
```
SQL

name
Kabul
Khulna
Kingston upon Hull
Koudougou
Kafr al-Dawwar

- can be used to get the top-N of a given category (`ORDER BY` and `LIMIT`)
- also useful as a sanity check to make sure your query doesn't return 10^7 rows

Learning about databases and tables

```
SHOW DATABASES;
SHOW TABLES;
DESCRIBE table;
```
SQL

```
SHOW TABLES;
+-----------+
| students  |
| courses   |
| grades    |
| teachers  |
+-----------+    4 rows in set
```

The SQL `INSERT` statement

```
INSERT INTO table
VALUES (value, value, ..., value);
```
SQL

```
INSERT INTO students
VALUES (789, "Nelson", "muntz@fox.com", "haha!");
```
SQL

- adds a new row to the given table
- columns' values should be listed in the same order as in the table
- How would we record that Nelson took CSE326 and got a D+ in it?

More about `INSERT`

```
INSERT INTO table (columnName, columnName, ..., columnName)
VALUES (value, value, ..., value);
```
SQL

```
INSERT INTO students (name, email)
VALUES ("Lewis", "lewis@fox.com");
```
SQL

- some columns have default or auto-assigned values (such as IDs)
- omitting them from the INSERT statement uses the defaults

The SQL REPLACE statement

```SQL
REPLACE INTO table (columnName, columnName, ..., columnName)
VALUES (value, value, ..., value);
```

```SQL
REPLACE INTO students
VALUES (789, "Martin", "prince@fox.com");
```

- just like INSERT, but if an existing row exists for that key (ID), it will be replaced
- can pass optional list of column names, like with INSERT

The SQL UPDATE statement

```SQL
UPDATE table
SET column = value,
    ...,
    column = value
WHERE column = value;
```

```SQL
UPDATE students
SET email = "lisasimpson@gmail.com"
WHERE id = 888;
```

- modifies an existing row(s) in a table
- BE CAREFUL! If you omit the WHERE, it modifies ALL rows

The SQL DELETE statement

```
DELETE FROM table
WHERE condition;
```
SQL

```
DELETE FROM students
WHERE id = 888;
```
SQL

- removes existing row(s) in a table
- can be used with other syntax like LIMIT, LIKE, ORDER BY, etc.
- BE CAREFUL! If you omit the WHERE, it deletes ALL rows

Creating and deleting an entire database

```
CREATE DATABASE name;
DROP DATABASE name;
```
SQL

```
CREATE DATABASE warcraft;
```
SQL

- adds/deletes an entire database from the server

Creating and deleting a table

```
CREATE TABLE name (
    columnName type constraints,
    ...
    columnName type constraints
);
DROP TABLE name;
```
SQL

```
CREATE TABLE students (
    id INTEGER,
    name VARCHAR(20),
    email VARCHAR(32),
    password VARCHAR(16)
);
```
SQL

- adds/deletes a table from this database
- all columns' names and types must be listed *(see next slide)*

SQL data types

- `BOOLEAN`: either `TRUE` or `FALSE`
- `INTEGER`
- `DOUBLE`
- `VARCHAR(`*length*`)` : a string
- `ENUM(`*value*`, ..., `*value*`)`: a fixed set of values
- `DATE`, `TIME`, `DATETIME`
- `BLOB` : binary data
- quick reference

Column constraints

```SQL
CREATE TABLE students (
    id INTEGER UNSIGNED NOT NULL PRIMARY KEY,
    name VARCHAR(20) NOT NULL,
    email VARCHAR(32),
    password VARCHAR(16) NOT NULL DEFAULT "12345"
);
```

- `NOT NULL`: not allowed to insert a null/empty value in any row for that column
- `PRIMARY KEY` / `UNIQUE`: no two rows can have the same value
- `DEFAULT` *value*: if no value is provided, use the given default
- `AUTO_INCREMENT`: default value is the last row's value plus 1 (useful for IDs)
- `UNSIGNED`: don't allow negative numbers (`INTEGER` only)

Rename a table

```SQL
ALTER TABLE name RENAME TO newName;
```

```SQL
ALTER TABLE students RENAME TO children;
```

- changes the name of an existing table

Add/remove/modify a column in a table

```SQL
ALTER TABLE name
    ADD COLUMN columnName type constraints;

ALTER TABLE name DROP COLUMN columnName;

ALTER TABLE name
    CHANGE COLUMN oldColumnName newColumnName type constraints;
```

- adds/deletes/respecifies a column in an existing table
- if a column is added, all existing rows are given a default value for that column

13.3: Multi-table Queries

- 13.1: Database Basics
- 13.2: SQL
- **13.3: Multi-table Queries**
- 13.4: Databases and PHP

Related tables and keys

students

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

courses

id	name	teacher_id
10001	Computer Science 142	1234
10002	Computer Science 143	5678
10003	Computer Science and Engineering 326	5238
10004	Informatics 100	1234

grades

student_id	course_id	grade
123	10001	B-
123	10002	C
456	10001	B+
888	10002	A+
888	10003	A+
404	10004	D+

teachers

id	name
1234	Krabappel
5678	Hoover
5238	Lee

- **primary key**: a column guaranteed to be unique for each record (e.g. Lisa Simpson's ID 888)
- **foreign key**: a column in table A storing a primary key value from table B
  - (e.g. records in `grades` with `student_id` of 888 are Lisa's grades)
- normalizing: splitting tables to improve structure / redundancy (linked by unique IDs)

Querying multi-table databases

When we have larger datasets spread across multiple tables, we need queries that can answer high-level questions such as:

- What courses has Bart taken and gotten a B- or better?
- What courses have been taken by both Bart and Lisa?
- Who are all the teachers Bart has had?
- How many total students has Ms. Krabappel taught, and what are their names?

To do this, we'll have to **join** data from several tables in our SQL queries.

Cross product with `JOIN`

```
SELECT column(s) FROM table1 JOIN table2;
```
SQL

```
SELECT * FROM students JOIN grades;
```
SQL

id	name	email	student_id	course_id	grade
123	Bart	bart@fox.com	123	10001	B-
404	Ralph	ralph@fox.com	123	10001	B-
456	Milhouse	milhouse@fox.com	123	10001	B-
888	Lisa	lisa@fox.com	123	10001	B-
123	Bart	bart@fox.com	123	10002	C
404	Ralph	ralph@fox.com	123	10002	C
... (24 rows returned)					

- **cross product** or **Cartesian product**: combines each row of first table with each row of second
  - produces $M * N$ rows, where table 1 has M rows and table 2 has N
  - problem: produces too much irrelevant/meaningless data

Joining with `ON` clauses

```
SELECT column(s)
FROM table1
JOIN table2 ON condition(s)
...
JOIN tableN ON condition(s);
```
SQL

```
SELECT *
FROM students
JOIN grades ON id = student_id;
```
SQL

- **join**: combines records from two or more tables if they satisfy certain conditions
- the `ON` clause specifies which records from each table are matched
- the rows are often linked by their **key** columns (id)

Join example

```SQL
SELECT *
FROM students
JOIN grades ON id = student_id;
```

id	name	email	student_id	course_id	grade
123	Bart	bart@fox.com	123	10001	B-
123	Bart	bart@fox.com	123	10002	C
404	Ralph	ralph@fox.com	404	10004	D+
456	Milhouse	milhouse@fox.com	456	10001	B+
888	Lisa	lisa@fox.com	888	10002	A+
888	Lisa	lisa@fox.com	888	10003	A+

- *table*.*column* can be used to disambiguate column names:

```SQL
SELECT *
FROM students
JOIN grades ON students.id = grades.student_id;
```

Filtering columns in a join

```SQL
SELECT name, course_id, grade
FROM students
JOIN grades ON id = student_id;
```

name	course_id	grade
Bart	10001	B-
Bart	10002	C
Ralph	10004	D+
Milhouse	10001	B+
Lisa	10002	A+
Lisa	10003	A+

Filtered join (`JOIN` with `WHERE`)

```SQL
SELECT name, course_id, grade
FROM students
JOIN grades ON id = student_id
WHERE name = 'Bart';
```

name	course_id	grade
Bart	10001	B-
Bart	10002	C

- `FROM` / `JOIN` glue the proper tables together, and `WHERE` filters the results
- what goes in the `ON` clause, and what goes in `WHERE`?
  - `ON` directly links columns of the joined tables
  - `WHERE` sets additional constraints such as particular values (`123`, `'Bart'`)

What's wrong with this?

```SQL
SELECT name, id, course_id, grade
FROM students
JOIN grades ON id = 123
WHERE id = student_id;
```

name	id	course_id	grade
Bart	123	10001	B-
Bart	123	10002	C

- The above query produces the same rows as the previous one, but it is poor style. Why?
- The `JOIN ON` clause is poorly chosen. It doesn't really say what connects a `grades` record to a `students` record.
  - They are related when they are for a student with the same `id`.
  - Filtering out by a specific ID or name should be done in the `WHERE` clause, not `JOIN ON`.

Giving names to tables

```SQL
SELECT s.name, g.*
FROM students s
JOIN grades g ON s.id = g.student_id
WHERE g.grade <= 'C';
```

name	student_id	course_id	grade
Bart	123	10001	B-
Bart	123	10002	C
Milhouse	456	10001	B+
Lisa	888	10002	A+
Lisa	888	10003	A+

- can give names to tables, like a variable name in Java
- to specify all columns from a table, write *table*`.*`
- (`grade` column sorts alphabetically, so grades C or better are ones <= it)

Multi-way join

```SQL
SELECT c.name
FROM courses c
JOIN grades g ON g.course_id = c.id
JOIN students bart ON g.student_id = bart.id
WHERE bart.name = 'Bart' AND g.grade <= 'B-';
```

name
Computer Science 142

- More than 2 tables can be joined, as shown above
- What does the above query represent?
- The names of all courses in which Bart has gotten a B- or better.

A suboptimal query

- Exercise: What courses have been taken by both Bart and Lisa?

```SQL
SELECT bart.course_id
FROM grades bart
JOIN grades lisa ON lisa.course_id = bart.course_id
WHERE bart.student_id = 123
AND lisa.student_id = 888;
```

- problem: requires us to know Bart/Lisa's Student IDs, and only spits back course IDs, not names.
- Write a version of this query that gets us the course *names*, and only requires us to know Bart/Lisa's names, not their IDs.

Improved query

- What courses have been taken by both Bart and Lisa?

```SQL
SELECT DISTINCT c.name
FROM courses c
JOIN grades g1 ON g1.course_id = c.id
JOIN students bart ON g1.student_id = bart.id
JOIN grades g2 ON g2.course_id = c.id
JOIN students lisa ON g2.student_id = lisa.id
WHERE bart.name = 'Bart'
AND lisa.name = 'Lisa';
```

Practice queries

- What are the names of all teachers Bart has had?

```SQL
SELECT DISTINCT t.name
FROM teachers t
JOIN courses c ON c.teacher_id = t.id
JOIN grades g ON g.course_id = c.id
JOIN students s ON s.id = g.student_id
WHERE s.name = 'Bart';
```

- How many total students has Ms. Krabappel taught, and what are their names?

```SQL
SELECT DISTINCT s.name
FROM students s
JOIN grades g ON s.id = g.student_id
JOIN courses c ON g.course_id = c.id
JOIN teachers t ON t.id = c.teacher_id
WHERE t.name = 'Krabappel';
```

Designing a query

- Figure out the proper SQL queries in the following way:
  - Which table(s) contain the critical data? (`FROM`)
  - Which columns do I need in the result set? (`SELECT`)
  - How are tables connected (`JOIN`) and values filtered (`WHERE`)?
- Test on a small data set (`imdb_small`).
- Confirm on the real data set (`imdb`).
- Try out the queries first in the MySQL console.
- Write the PHP code to run those same queries.
  - Make sure to check for SQL errors at every step!!

Example `imdb` database

actors

id	first_name	last_name	gender
433259	William	Shatner	M
797926	Britney	Spears	F
831289	Sigourney	Weaver	F
…			

movies

id	name	year	rank
112290	Fight Club	1999	8.5
209658	Meet the Parents	2000	7
210511	Memento	2000	8.7
…			

roles

actor_id	movie_id	role
433259	313398	Capt. James T. Kirk
433259	407323	Sgt. T.J. Hooker
797926	342189	Herself
…		

movies_genres

movie_id	genre
209658	Comedy
313398	Action
313398	Sci-Fi
…	

directors

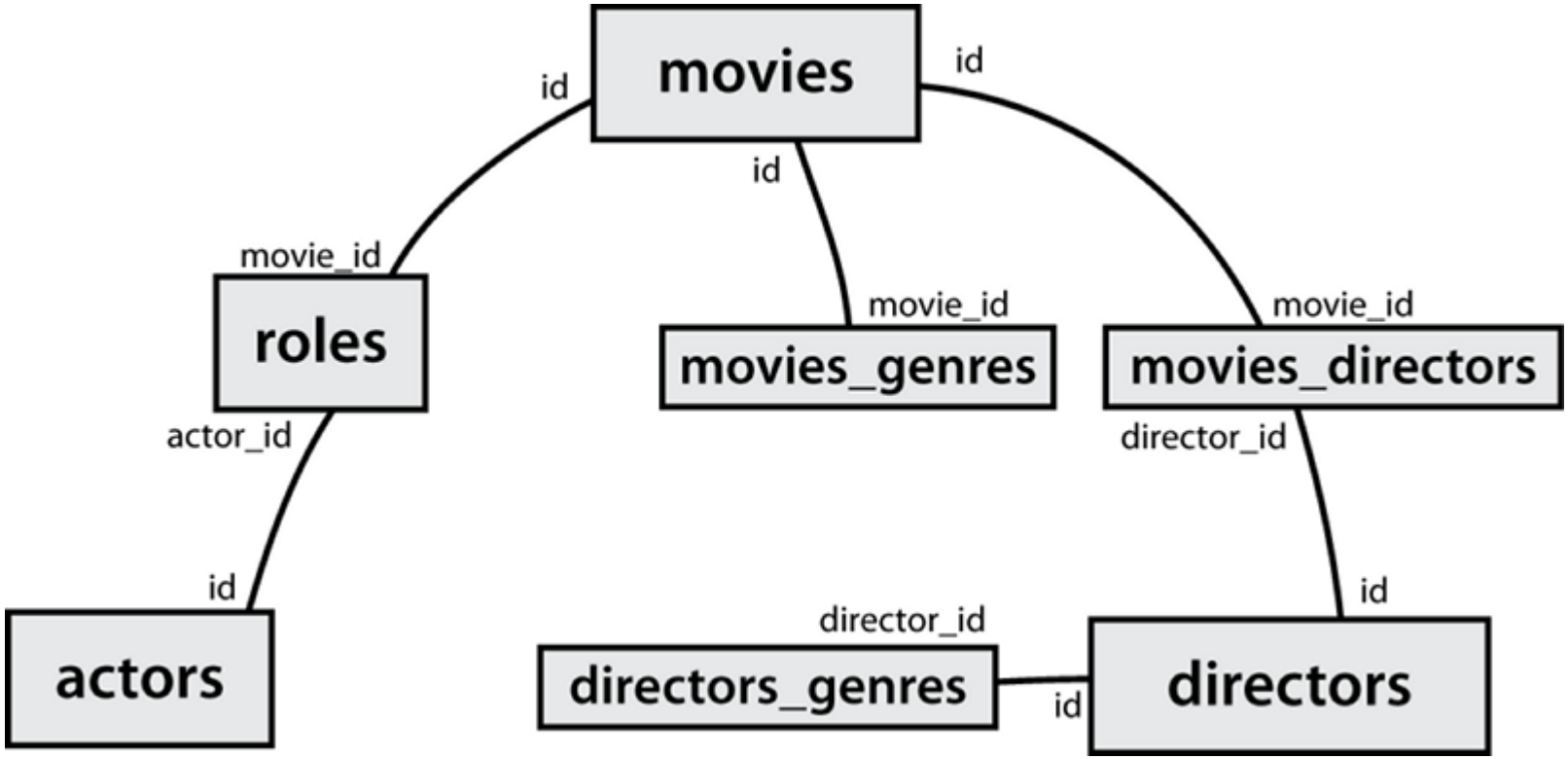
id	first_name	last_name
24758	David	Fincher
66965	Jay	Roach
72723	William	Shatner
…		

movies_directors

director_id	movie_id
24758	112290
66965	209658
72723	313398
…	

- also available, `imdb_small` with fewer records (for testing queries)

IMDb table relationships / ids



IMDb query example

```
Scotts-MacBook: ~scottlee$ mysql -u myusername -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> use imdb_small;
Database changed

mysql> select * from actors where first_name like '%mick%';
+--------+------------+-----------+--------+
| id     | first_name | last_name | gender |
+--------+------------+-----------+--------+
|  71699 | Mickey     | Cantwell  | M      |
| 115652 | Mickey     | Dee       | M      |
| 470693 | Mick       | Theo      | M      |
| 716748 | Mickie     | McGowan   | F      |
+--------+------------+-----------+--------+
4 rows in set (0.01 sec)
```

IMDb practice queries

- What are the names of all movies released in 1995?
- How many people played a part in the movie "Lost in Translation"?
- What are the *names* of all the people who played a part in the movie "Lost in Translation"?
- Who directed the movie "Fight Club"?
- How many movies has Clint Eastwood directed?
- What are the *names* of all movies Clint Eastwood has directed?
- What are the names of all directors who have directed at least one horror film?
- What are the names of every actor who has appeared in a movie directed by Christopher Nolan?

13.4: Databases and PHP

- 13.1: Database Basics
- 13.2: SQL
- 13.3: Multi-table Queries
- **13.4: Databases and PHP**

Querying a Database in PHP with PDO

```php
$name = new PDO("dbprogram:dbname=database;host=server", username, password);
$name->query("SQL query");
```

```php
# connect to world database on local server
$db = new PDO("mysql:dbname=world;host=localhost", "traveler", "packmybags");
$db->query("SELECT * FROM countries WHERE population > 100000000;");
```

- PDO database library allows you to connect to many different database programs
  - replaces older, less versatile functions like `mysql_connect`
- PDO object's `query` function returns rows that match a query

Result rows: `query`

```php
$db = new PDO("mysql:dbname=world;host=localhost", "traveler", "packmybags");
$rows = $db->query("SELECT * FROM countries WHERE population > 100000000;");
foreach ($rows as $row) {
    do something with $row;
}
```

- `query` returns all result rows
  - each row is an associative array of [column name -> value]
  - example: `$row["population"]` gives the value of the `population` column

A complete example

```php
$db = new PDO("mysql:dbname=imdb_small", "jessica", "guinness");
$rows = $db->query("SELECT * FROM actors WHERE last_name LIKE 'Del%'");
foreach ($rows as $row) {
    ?>
    <li> First name: <?= $row["first_name"] ?>,
         Last name:  <?= $row["last_name"]  ?> </li>
    <?php
}
```

- First name: Benicio, Last name: Del Toro
- First name: Michael, Last name: Delano
- …

output

PDO object methods

name	description
query	performs a SQL SELECT query on the database
exec	performs a SQL query that modifies the database (INSERT, DELETE, UPDATE, etc.)
getAttribute, setAttribute	get/set various DB connection properties
quote	encodes a value for use within a query

Including variables in a query

```php
# get query parameter for name of movie
$title = $_GET["movietitle"];
$rows = $db->query("SELECT year FROM movies WHERE name = '$title'");
```

- you should not directly include variables or query parameters in a query
- they might contain illegal characters or SQL syntax to mess up the query

Quoting variables

```php
# get query parameter for name of movie
$title = $_GET["movietitle"];
$title = $db->quote($title);
$rows = $db->query("SELECT year FROM movies WHERE name = $title");
```

- call PDO's `quote` method on any variable to be inserted
- `quote` escapes any illegal chars and surrounds the value with `'` quotes
- prevents bugs and security problems in queries containing user input

Database/query errors

```php
$db = new PDO("mysql:dbname=imdb_small", "jessica", "guinness");
$rows = $db->query("SEEELECT * FROM movies WHERE year = 2000");   # FALSE
```

- database commands can often fail (invalid query; server not responding; etc.)
- normally, PDO commands fail silently by returning `FALSE` or `NULL`
- but this makes it hard to notice and handle problems

Exceptions for errors

```php
$db = new PDO("mysql:dbname=imdb_small", "jessica", "guinness");
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$rows = $db->query("SEEELECT * FROM movies WHERE year = 2000");    # kaboom!
```

- using `setAttribute`, you can tell PDO to throw (generate) a `PDOException` when an error occurs
- the exceptions will appear as error messages on the page output
- you can **catch** the exception to gracefully handle the error

Catching an exception

```php
try {
    statement(s);
} catch (ExceptionType $name) {
    code to handle the error;
}
```

- a `try/catch` statement attempts to run some code, but if it throws a given kind of exception, the program jumps to the `catch` block and runs that code to handle the error

Example with error checking

```php
try {
    $db = new PDO("mysql:dbname=imdb_small", "jessica", "guinness");
    $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $rows = $db->query("SEEELECT * FROM movies WHERE year = 2000");
    foreach ($rows as row) { ... }
} catch (PDOException $ex) {
    ?>
    <p>Sorry, a database error occurred. Please try again later.</p>
    <p>(Error details: <?= $ex->getMessage() ?>)</p>
    <?php
}
```

PHP

PDOStatement methods

The `$rows` returned by PDO's `query` method is technically not an array but an object of type `PDOStatement`. Here are its methods:

`columnCount()`	number of columns in the results
`fetch()`	return the next row from the results
`fetchColumn(`*number*`)`	return the next column from the results
`rowCount()`	number of rows returned by the query

```sql
if ($db->rowCount() > 0) {
    $first_row = $db->fetch();
    ...
}
```

SQL

Database Design

- 13.1: Database Basics
- 13.2: SQL
- 13.3: Multi-table Queries
- 13.4: Databases and PHP

Database design principles

- **database design** : the act of deciding the schema for a database
- **database schema**: a description of what tables a database should have, what columns each table should contain, which columns' values must be unique, etc.
- some database design principles:
  - keep it simple, stupid (KISS)
  - provide an identifier by which any row can be uniquely fetched
  - eliminate redundancy, especially of lengthy data (strings)
    - integers are smaller than strings and better to repeat
  - favor integer data for comparisons and repeated values
    - integers are smaller than strings and better to repeat
    - integers can be compared/searched more quickly than strings, real numbers

First database design

student_grades

name	email	course	teacher	grade
Bart	bart@fox.com	Computer Science 142	Krabappel	B-
Bart	bart@fox.com	Computer Science 143	Hoover	C
Milhouse	milhouse@fox.com	Computer Science 142	Krabappel	B+
Lisa	lisa@fox.com	Computer Science 143	Hoover	A+
Lisa	lisa@fox.com	Computer Science and Engineering 326	Lee	A+
Ralph	ralph@fox.com	Informatics 100	Krabappel	D+

- what's good and bad about this design?
  - good: simple (one table), can see all data in one place
  - bad: redundancy (name, email, course repeated frequently)
  - bad: most searches (e.g. find a student's courses) will have to rely on string comparisons
  - bad: there is no single column whose value will be unique in each row

Improved database design

students

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

courses

id	name	teacher_id
10001	Computer Science 142	1234
10002	Computer Science 143	5678
10003	Computer Science and Engineering 326	5238
10004	Informatics 100	1234

grades

student_id	course_id	grade
123	10001	B-
123	10002	C
456	10001	B+
888	10002	A+
888	10003	A+
404	10004	D+

teachers

id	name
1234	Krabappel
5678	Hoover
5238	Lee

- normalizing: splitting tables to improve structure / redundancy (linked by unique IDs)
- **primary key**: a column guaranteed to be unique for each record (e.g. Lisa Simpson's ID 888)
- **foreign key**: a column in table A storing a primary key value from table B
  - (e.g. records in `grades` with `student_id` of 888 are Lisa's grades)