1 Join Operations

1.1 Natural Joins

- The join condition for natural join is that all pairs of attributes from the two relations having a common name are equated
- Put simply, a natural join is an inner join on all attributes with common names, the `WHERE` clause (or `ON`) clause is not required (unless we want to specify additional non-join conditions)

Suppose we want the natural join of the relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

The result would be a relation whose schema includes `name` and `address` plus all attributes that appear in one or the other of the two relations. The resulting relation after the natural join would represent people who are both stars and executives, and will have all info pertinent to either: name, address, gender, birthdate, certificate number, and net worth:

```
MovieStar NATURAL JOIN MovieExec
```

1.2 Outer Joins

1.2.1 Full Outer Joins

- The *outer join* is formed by starting with an natural join on two relations, and then adding any dangling tuples from the two relations. The added tuples must have a *NULL* value, in all attributes that they do no posses but that appear in the join result.

Suppose we want the outer join of the relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

If MySQL of SQLite supported outer joins the syntax for the query above would look like:

```
MovieStar NATURAL FULL OUTER JOIN MovieExec;
```

Then the result of this operation would be a relation with the same six-attributes schema as the previous example for natural join. However, the tuples in this relation are of three kinds:

1. The first kind of tuple has values for all six attributes (no *NULL* values). This kind of tuple represents individuals who are both stars and executives.
2. The second kind of tuple is one for an individual who is a star but not an executive. These tuples have values for attributes `name, address, gender,` and `birthdate` taken from their tuple in `MovieStar`, while attributes belonging only to `MovieExec`, namely `cert#` and `netWorth`, have *NULL* values.
3. The third kind of tuple is for executives who are not stars. The tuples have values for attributes of `MovieExec` taken from their `MovieExec` tuple and *NULL* in the attributes `gender` and `birthdate` that come only from `MovieStar`.

1.2.2 LEFT Joins

A variation of outer join is left outer join

Consider the following query:

```
MovieStar NATURAL LEFT OUTER JOIN MovieExec;
```

This would yield the first and second types of tuples from the OUTER JOIN.

1.2.3 RIGHT Joins

Another variation of outer join is right outer join

Consider the following query:

```
MovieStar NATURAL RIGHT OUTER JOIN MovieExec;
```

This would yield the first and third types of tuples from the OUTER JOIN.

2 Set Operations

Unlike the `SELECT` statement, which preserves duplicates as a default and only eliminates them when instructed to by the `DISTINCT` keyword, the set union, set intersection, and set difference operations normally eliminate duplicates.

2.1 Set Intersection ∩

Consider the following two relations:

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

Suppose we wanted the names and addresses of all female movie stars who are also movie executives with a net worth over $10,000,000

```
(SELECT M.name, M.address
FROM MovieStar M
WHERE M.gender = 'F')
    INTERSECT
(SELECT ME.name, ME.address
FROM MovieExec ME
WHERE ME.netWorth > 100000000);
```

This query first yields two tables with columns `name` and `address`, and we can do an intersect since the two table's columns are the same.

2.2 Set Difference −

Suppose we wanted the names and addresses of stars who are not movie executives, regardless of gender or net worth

```
(SELECT name, address FROM MovieStar)
    EXCEPT
(SELECT name, address FROM MovieExec);
```

In the previous two examples conveniently the attributes of the relations that we were performing operations on are the same. However, we can also rename attributes if necessary to get a common set of attributes.

2.3 Set Union ∪

Consider the following two relations:

`Movies(title, year, length, genre, studioName, producerC#)`

`StarsIn(movieTitle, movieYear,starName)`

Ideally, these sets of movies would be the same, but in practice it is common for relations to diverge; for instance we might have movies with no listed stars or a `StarsIn` tuple that mentions a movie not found in the `Movies` relation. So we must write:

```
(SELECT title, year FROM Movie)
   UNION
(SELECT movieTitle AS title, movieYear as year FROM StarsIn);
```

The result would be all movies mentioned in both relations, with `title` and `year` as attributes of the resulting relation.

3 Questions

Using Join and Set operations, and the **SchoolScheduling Schema** answer the following 3 questions.

3.1 Question 1

Using a `LEFT NATURAL JOIN` on Staff and Faculty, print the names of staff members who are not faculty members.

```
SELECT StfFirstName, StfLastName
FROM Staff LEFT NATURAL JOIN Faculty
WHERE Title IS NULL;
```

3.2 Question 2

Find without duplicates, the first names of all students who have never received a grade under 80. *Hint: Use* `EXCEPT` *and* `NATURAL JOIN`

```
SELECT StudFirstName FROM Students
EXCEPT
SELECT StudFirstName
FROM Students NATURAL JOIN Student_Schedules
```

```
WHERE Grade>0 AND Grade<80;
```

3.3 Question 3

Print the names of all subjects and each subjects's faculty name (or names if more than 1) with the highest proficiency rating in that subject. *Hint: Use* `NATURAL LEFT JOIN`, `JOIN`, *and* `NATURAL JOIN`

```
SELECT SubjectName, GROUP_CONCAT(StfFirstName || " " || StfLastname),
   ProficiencyRating
FROM Faculty_Subjects FS JOIN
      (SELECT SubjectId, MAX(ProficiencyRating) AS MaxProficiency
      FROM Faculty_Subjects
      GROUP BY SubjectId) AS MP
      ON FS.SubjectID = MP.SubjectID AND FS.ProficiencyRating = MP.
         MaxProficiency
      NATURAL JOIN Staff
      NATURAL JOIN Subjects
GROUP BY SubjectName;
```

3.4 Question 4

Using the database schema below:

`Movies(title, year, length, genre, studioName, producerC#)`

`StarsIn(movieTitle, movieYear,starName)`

`MovieStar(name, address, gender, birthdate)`

`MovieExec(name, address, cert#, netWorth)`

`Studio(name, address, presC#)`

Write a query to find the male stars in *Titanic*.

```
SELECT name
FROM MovieStar
WHERE gender = 'M'

INTERSECT

SELECT starName as name
FROM StarsIn
WHERE movieTitle = 'Titanic';
```

3.5 Question 5

Using the database schema below:

`Product(maker, model, type)`

`PC(model, speed, ram, hd, price)`

`Laptop(model, speed, ram, hd, screen, price)`

`Printer(model, color, type, price)`

Write a query to find the model number and price of all products (of any type) made by manufacture B.

```
(SELECT Product.model, price
 FROM Product, PC
 WHERE Product.model = PC.model AND maker = 'B')

UNION

(SELECT Product.model, price
 FROM Product, Laptop
 WHERE Product.model = Laptop.model AND maker = 'B')

UNION

(SELECT Product.model, price
 FROM Product, Printer
 WHERE Product.model = Printer.model AND maker='B');
```

4 Regular Expressions

REGEX in SQL takes a regular expression and returns every tuple that matches the expression. Different SQL implementations may only recognize subsets of regular expressions, or treat some expressions slightly differently.

The simplest kind of regular expression is a sequence of simple characters. So in the `EntertainmentAgency.sqlite` table on Canvas, to search for all entertainers with the `EntStageName` equal to '*Topazz*' we would simply write:

```
SELECT *
FROM Entertainers
WHERE EntStageName REGEXP 'Topazz'
```

This query would return any tuple in which `EntStageName` contains the substring '*Topazz*'. So if there was a hypothetical `EntStageName` named '*Topazz the Topazz man*' this query would match it. It would also match '*429-rjeqpnfkrwbg_Topazz_lskad;fm*'. However, regular expressions are **case sensitive**; this means the previous query would not match '*topazz Sir topazz alot*'.

4.1 Disjunctions (Logical OR)

In regex to form a disjunction on characters we use square brackets:

RE	Match	Example Patterns
[wW]oodchucks	Woodchuck or woodchuck	'Woodchuck'
[abc]	'a', 'b', *or* 'c'	'In oumini, in soldati'
[1234567890]	any digit	'plenty of 7'
[A-Z]	an upper case letter	'we should call it Drenched Blossoms'
[a-z]	a lower case letter	'my beans were impatient to be hoed!'
[0-9]	a single digit	'Chapter 1: Down the Rabbit hole'
[^A-Z]	not an upper case letter	'Oyfn pripetchik'
[^Ss]	neither 'S' nor 's'	'SsSSSsisssSSsSs'

4.2 Special Characters ? * + .

? zero or one instance of the previous character or expression

RE	Match	Example Patterns
woodchucks?	woodchuck or woodchucks	'woodchuck'
colou?r	color or colour	'colour '

*** zero or more instances of the previous character or expression**
`a*` means "any string of zero or more a's". This will match *a* or *aaaaaa*, but it will also match *woodchuck* since the string *woodchuck* has zero a's. So the regex for matching one or more a is `aa*`, meaning one *a* followed by zero or more *a*'s.
At a more complex level `[ab]*` means "zero or more a's or b's" , matching strings like *aaaa* or *ababab* or *bbb*. To specify multiple digits (useful for finding prices) we cane extend `[0-9]` , the regular expression for a single digit. Therefore a sequence of digits is `[0-9][0-9]*` ?

+ one or more instances of the immediately previous character or expression
We can also use `[0-9]+` to specify a sequence of digits.

If for example we consider the language of sheep which consists of strings with *b*, followed by at least two a's, followed by an exclamation mark like so:

baa!
baaa!
baaaa!
baaaaa!
...
We can specify it with two alternative and equally valid ways `baaa*!` or `baa+!`

. matches any single character
The period '.' is called the **wildcard** expression,

RE	Match	Example Patterns
beg.n	any character between *beg* and *n*	'begin,beg'n,begun,beg5n'

Wildcard is often used together with * to mean *any string of characters*. For example, suppose we want to find any line in which a particular name, say, *Bert*, appears twice. We can specify this with the regex `Bert.*Bert`.

Enclosing a pattern in a parentheses makes it act like a single character
So to make disjunction operator apply only to a specific pattern we need to use the parenthesis operators (and). So the pattern `gupp(y|ies)` would match *guppy* or *guppies*, and we apply the disjunction only to suffixes *y* and *ies*. Or we have a line that has column labels of the form *Column 1 Column 2 Column 3* then we could use `(Column[[:space:]][0-9]+[[:space:]]*)*`, to match the world *Column*, followed by a number and optional spaces, the whole pattern repeated any number of times.

{n} means n occurrences of the previous char or expression
So `Pizza{3}` would match *Pizzaaa*.

{n,m} from n to m occurrences of the previous char or expression

{n,} at least n occurrences of the previous char or expression

{,m} at most m occurrences of the previous char or expression

4.3 Special Character Escapes

Special characters ? * + . $ all mean something. To use them in regex without invoking the function we escape them with the backslash so for example `[ ^e\^]` matches neither *e* nor the caret symbol ^ itself.

4.4 Anchors

Anchors are special characters that anchor regular expressions to particular places in a string. The most common are the caret ^ and the dollar sign $.

^ matches the start of a line

The pattern `^The` matches the word *The* only at the start of a line.

The caret ^ has three uses

1. to match the start of a line
2. to indicate a negation inside a square bracket
3. just mean a caret

$ matches the end of a line

The pattern `[[:space:]]$` is useful pattern for matching a space at the end of a line. `^The dog\.$` matches a line that contains only the phrase *The dog.* (using the backslash since we want the . to mean 'period' and not the wildcard)