SQL Queries

Select-From-Where Statements

Principal form of a query is:

```
SELECT desired attributes
FROM one or more tables
WHERE condition about tuples of the tables
```

1. Begin with the relation in the **FROM** clause.
2. Apply the selection indicated by the **WHERE** clause.
3. Apply the projection indicated by the **SELECT** clause.

(Extended) Projection in SQL

```
SELECT title, length
FROM Movies
WHERE studioName = 'Disney';

SELECT title AS name, length AS duration
FROM Movies
WHERE studioName = 'Disney';

SELECT title AS name, length*0.016667 AS lenghtInHours
FROM Movies
WHERE studioName = 'Disney';

SELECT title AS name, length/60.0 AS length, 'hrs.' AS inHours
FROM Movies
WHERE studioName = 'Disney';
```

WHERE in SQL

- Build expressions by using the operators:
  =, <>, <, >, <=, >=
- String constants are surrounded by single quotes.
  studioName = 'Disney'
- Numeric constants are for e.g.: -12.34, 1.23E45
- Boolean operators are: **AND, OR, NOT**.

Example

Which movies are made by Disney and aren't rated ‘G’?

```
SELECT title
FROM Movies
WHERE (studioname = 'Disney') AND NOT (rating='G');
```

Selection in SQL (Cont.)

- Which Disney movies are after 1970 or have length greater than 90 mins?

```
SELECT title
FROM Movies
WHERE (year > 1970 OR length < 90) AND
  studioName='Disney';
```

- Parenthesis are needed because the precedence of OR is less than that of AND.

Patterns in WHERE

- General form:

  <Attribute> LIKE <pattern>

  <Attribute> NOT LIKE <pattern>

- **<pattern>** is a quoted string which may contain

  % = meaning "any string"

  _ = meaning "any character."

Example. Suppose we remember a movie "Princess *something*".

```
SELECT title
FROM Movies
WHERE title LIKE '%Princess%';
```

Or

```
SELECT title
FROM Movies
WHERE lower(title) LIKE '%princess%';
```

Ordering the Input

Example. Find the Disney movies and list them by length, shortest first.

```
SELECT *
FROM Movies
WHERE studioName = 'Disney'
ORDER BY length;
```

Example. Find the Disney movies and list them by length, shortest first, and among movies of equal length, sort alphabetically.

```
SELECT *
FROM Movies
WHERE studioName = 'Disney'
ORDER BY length, title;
```

Remarks

- Ordering is ascending, unless you specify the DESC keyword after **an attribute**.
- Ties are broken by the second attribute on the ORDER BY list, etc.

Products and Joins in SQL

- SQL has a simple way to couple tables: list them in the FROM clause.
  - All the tables in the FROM clause are coupled through Cartesian product
  - Then we can put conditions in the WHERE clause in order to get the desired kind of join.

Example. We want to know the website of the studio of Pretty Woman.

```
SELECT website
FROM Movies, Studios
WHERE title = 'Pretty Woman' AND studioName=name;
```

Products and Joins in SQL

Example. We want to know the stars of Paramount movies.

```
SELECT Stars.name, Stars.birthdate, Stars.birthplace
FROM Movies, Stars, StarsIn
WHERE studioname = 'Paramount' AND
  StarsIn.title = Movies.title AND
  StarsIn.year = Movies.year AND
  StarsIn.starName = Stars.name;
```

- When we involve **two or more** tables in a query, we can have attributes with the **same** name among these relations.
  - **Solution**: We disambiguate by putting the name of the relation followed by a dot and then the name of the attribute.

Natural Join

```
SELECT *
FROM Movies NATURAL JOIN StarsIn;
```

Possible because the join attributes have the same name
Almost the same as

```
SELECT *
FROM Movies, StarsIn
WHERE   Movies.title=StarsIn.title AND
        Movies.year=StarsIn.year;
```

Why not the same?

Natural Join with USING

Better than NATURAL JOIN:

```
SELECT *
FROM Movies JOIN StarsIn USING (title,year);
```

Because now it is explicit which attributes are used to join the tables.

Join with ON

A similar result can be obtained by:

```
SELECT *
FROM Movies JOIN StarsIn ON
                    Movies.title=StarsIn.title AND
                    Movies.year=StarsIn.year;
```

However, now we get two copies for title and year.
This is exactly the same as:

```
SELECT *
FROM Movies, StarsIn
WHERE     Movies.title=StarsIn.title AND
          Movies.year=StarsIn.year;
```

Outer Joins

```
SELECT *
FROM Movies NATURAL FULL OUTER JOIN StarsIn;
```

```
SELECT *
FROM Movies NATURAL LEFT OUTER JOIN StarsIn;
```

```
SELECT *
FROM Movies NATURAL RIGHT OUTER JOIN StarsIn;
```

One of LEFT, RIGHT, or FULL before OUTER (but not missing).

- LEFT = pad dangling tuples of Movies only.
- RIGHT = pad dangling tuples of StarsIn only.
- FULL = pad both.

Example

Compare:

```
SELECT *
FROM Movies JOIN Studios
        ON Movies.studioName = Studios.name;

SELECT *
FROM Movies LEFT OUTER JOIN Studios
        ON Movies.studioName = Studios.name;
```

Outerjoins: Students example (I)

```
CREATE TABLE Fexam (stdid INT PRIMARY KEY, mark INT);
CREATE TABLE Assig ( stdid INT PRIMARY KEY, mark INT);

INSERT INTO Fexam VALUES(1,60);
INSERT INTO Fexam VALUES(2,70);
INSERT INTO Fexam VALUES(3,80);
INSERT INTO Fexam VALUES(5,90);

INSERT INTO Assig VALUES(1,30);
INSERT INTO Assig VALUES(3,40);
INSERT INTO Assig VALUES(4,50);
INSERT INTO Assig VALUES(5,60);
```

Suppose we want to join Fexam and Assig to get both marks for each student.

Outerjoins: Students example (II)

Suppose we start with:

```
SELECT *
FROM Fexam NATURAL JOIN Assig;
```

Result?

Empty.

Why?

Attribute *mark* used too for the join. It shouldn’t .

Outerjoins: Students example (III)

```
SELECT *
FROM Fexam JOIN Assig USING(stdid);
```

Result?

stdid	mark	mark
1	60	30
3	80	40
5	90	60

Problem1?

Not clear which mark is fexam mark, which is assig mark.

Outerjoins: Students example (IV)

```
SELECT stdid, Fexam.mark AS fmark, Assig.mark AS amark
FROM Fexam JOIN Assig USING(stdid);
```

Result?

stdid	fmark	amark
1	60	30
3	80	40
5	90	60

Problem2?

Student 2 and 4 are lost.

Outerjoins: Students example (V)

```
SELECT stdid, Fexam.mark AS fmark, Assig.mark AS amark
FROM Fexam FULL OUTER JOIN Assig USING(stdid);
```

Result?

stdid	fmark	amark
1	60	30
2	70	NULL
3	80	40
4	NULL	60
5	90	60

Outerjoins: Students example (VI)

SELECT Fexam.stdid, Fexam.mark AS fmark, Assig.mark AS amark
FROM Fexam FULL OUTER JOIN Assig ON Fexam.stdid=Assig.stdid;

Result?

stdid	fmark	amark
1	60	30
3	80	40
	NULL	50
5	90	60
2	70	NULL

Why did we get NULL for stdid of this tuple here?

Because that tuple has stdid=4 which is not in Fexam.
Recall, we are printing Fexam.stdid
We should change the query to:

SELECT Fexam.stdid, Assig.stdid, Fexam.mark AS fmark, Assig.mark AS amark
FROM Fexam FULL OUTER JOIN Assig ON Fexam.stdid=Assig.stdid;

Union/Intersection/Difference

Find the movies where either Richard Gere or Julia Roberts star.

Find the movies where both Richard Gere and Julia Roberts star.

Find the movies where Richard Gere stars but Julia Roberts doesn't.

```
SELECT title, year
FROM StarsIn
WHERE starName='Richard Gere'
  UNION / INTERSECT / EXCEPT  (use one of them depending on request)
SELECT title, year
FROM StarsIn
WHERE starName='Julia Roberts' ;
```

Aliases

- Sometimes we need to ask a query that combines a table with itself.
  - We may list a table T as many times we want in the from clause but **we need a way to refer to each occurrence of T**.
  - SQL allows us to define, for each occurrence in the FROM clause, an alias (which is called "**tuple variable**").

Example. Find pairs of stars who have played together in the same movie.

```
SELECT S1.starname, S2.starname
FROM StarsIn S1, StarsIn S2
WHERE S1.title = S2.title AND S1.year = S2.year
        AND S1.starname < S2.starname;
```

Why not use <> instead?

Aggregations

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.

Example. Find the average length of movies from Disney.

```
SELECT AVG(length)
FROM Movies
WHERE studioName = 'Disney';
```

Eliminating Duplicates in an Aggregation

- **DISTINCT** inside an aggregation causes duplicates to be eliminated **before** the aggregation.

Example. Find the number of different studios in the Movies table.

```
SELECT COUNT(DISTINCT studioName)
FROM Movies;
```

This is not the same as:

```
SELECT DISTINCT COUNT(studioName)
FROM Movies;
```

DISTINCT here is useless!
Why?

Not only in COUNT…

```
SELECT AVG(DISTINCT length)
FROM Movies
WHERE studioname = 'Disney';
```

- This will produce the average of only the distinct values for length.

Grouping

- What if we want to find the average movie length for each studio?
- We may follow the query by **GROUP BY** and a list of attributes.
- The result
  - is grouped according to the values of all the listed attributes in GROUP BY, and
  - any aggregation is applied only within each group.

Example.

```
SELECT studioName, AVG(length)
FROM Movies
GROUP BY studioName;
```

Another Example

From **Movies** and **StarsIn**, find the star's total length of film played.

```
SELECT starName, SUM(length)
FROM Movies, StarsIn
WHERE Movies.title=StarsIn.title AND Movies.year=StarsIn.year
GROUP BY starName;
```

Compute those tuples first, then group by starName.

HAVING Clauses

HAVING <condition> may follow a GROUP BY clause.

- If so, the condition applies to each group, and groups not satisfying the condition are eliminated.

Example

```
SELECT starName, SUM(length)
FROM Movies, StarsIn
WHERE Movies.title=StarsIn.title AND Movies.year=StarsIn.year
GROUP BY starName;
```

Suppose we didn't wish to include all the stars in our table of aggregated lengths. We want those stars that have at least one movie before 2000.

Solution

```
SELECT starName, SUM(length)
FROM Movies, StarsIn
WHERE Movies.title=StarsIn.title AND Movies.year=StarsIn.year
GROUP BY starName
HAVING MIN(StarsIn.year) < 2000;
```

Requirements on HAVING Conditions

- These conditions may refer to any relation in the FROM clause.
- They may refer to attributes of those relations, as long as the attribute makes sense within a group; i.e., it is either:
    1. A grouping attribute, or
    2. Aggregated attribute.

Restriction on SELECT Lists With Aggregation

- If any aggregation is used, then each element of the SELECT list must be either:
  1. Aggregated, or
  2. An attribute on the GROUP BY list.

Illegal Query Example

- We might think we could find the shortest movie of Disney as:

```
SELECT title, MIN(length)
FROM Movies
WHERE studioName = 'Disney';
```

- But this query is illegal in SQL. Because **title** is neither aggregated nor on the GROUP BY list.

- We should do instead:

```
SELECT title, length
FROM Movies
WHERE studioName = 'Disney' AND length =
    (SELECT MIN(length)
    FROM Movies
    WHERE studioName = 'Disney');
```

Exercise

Using **Movies**, **StarsIn**, and **Stars**,

find the star's total length of film played.

We are interested only in Canadian stars and who first appeared in a movie before 2000.

```
SELECT starName, SUM(length)
FROM Movies, StarsIn, Stars
WHERE Movies.title=StarsIn.title AND Movies.year=StarsIn.year
      AND Stars.name=StarsIn.starName
      AND Stars.birthplace LIKE '%Canada%'
GROUP BY starName
HAVING MIN(StarsIn.year) < 2000;
```

Correlated Subqueries

- Suppose StarsIn table has an additional attribute “salary”

  StarsIn(movie, movie, starName, salary)

  Now, find the stars who were paid for some movie more than the average salary for that movie.

```
SELECT starName, title, year
FROM StarsIn X
WHERE salary >
        (SELECT AVG(salary)
        FROM StarsIn
        WHERE title = X.title AND year=X.year);
```

Remark

Semantically, the value of the X tuple changes in the outer query, so the database must rerun the subquery for each X tuple.

Another Solution (Nesting in FROM)

```
SELECT X.starName, X.title, X.year
    FROM StarsIn X, (SELECT title, year, AVG(salary) AS avgSalary
                     FROM StarsIn
                     GROUP BY title, year) Y
    WHERE X.salary>Y.avgSalary AND
                X.title=Y.title AND X.year=Y.year;
```

Views

- A view is a "*virtual table*", a relation that is defined in terms of the contents of other tables and views.
- In contrast, a relation whose value is really stored in the database is called a *base table*.

Example

```
CREATE VIEW DMovies AS
    SELECT title, year, length, rating
    FROM Movies
    WHERE studioName = 'Disney';
```

Accessing a View

Query a view as if it were a base table.

Examples

```
SELECT title
FROM DMovies
WHERE year = 2021;

SELECT DISTINCT starName
FROM DMovies, StarsIn
WHERE DMovies.title = StarsIn.title AND DMovies.year = StarsIn.year;
```

View on more than one relation

```
CREATE VIEW MovieStar AS
    SELECT title, year, studioName, starName
    FROM Movies JOIN StarsIn USING (title,year);
```

For each star that has more than two movies with Paramount, find how many movies he/she has with Fox.

```
CREATE VIEW ParamountStars2 AS
    SELECT starName
    FROM MovieStar
    WHERE studioName='Paramount'
    GROUP BY starName
    HAVING COUNT(title)>=2;

CREATE VIEW FoxStars AS
    SELECT *
    FROM MovieStar
    WHERE studioName='Fox';

SELECT starName, COUNT(title)
FROM ParamountStars2 NATURAL LEFT OUTER JOIN FoxStars
GROUP BY starName;
```

EXISTS / NOT EXISTS

Find the stars who have worked for every studio.

```
SELECT DISTINCT starName
FROM MovieStar X
WHERE NOT EXISTS (
    SELECT name
    FROM Studios
        EXCEPT
    SELECT studioName
    FROM MovieStar
    WHERE starName = X.starName);
```

Checks emptiness of the subquery.

Find the stars who have worked for Disney but no other studio.

```
SELECT starName
FROM MovieStar X
WHERE X.studioName='Disney' AND NOT EXISTS (
   SELECT *
   FROM MovieStar
   WHERE    starName=X.starName  AND
            studioName<>'Disney'
);
```

Find the stars who have worked for only one studio.

```
SELECT starName
FROM MovieStar X
WHERE NOT EXISTS (
   SELECT *
   FROM MovieStar
   WHERE    starName=X.starName  AND
            studioName<>X.studioName
);
```