COMP353 Databases

More on SQL:

Nested Querie

Views

Scalar Values

- An SQL query is an expression that evaluates to a collection of tuples, i.e., it produces a relation/bag
- This "collection" may have only one attribute
- It is also possible that there will be only one single value produced for that attribute
- If all these hold, then we say that the query produces a **scalar** value
  - **Scalar values** – example include simple values such as integers, reals, strings, dates, etc.

Queries that Produce Scalar Values

- Relation schema:

  Movie(title, year, length, filmType, studioName, producerC#)

- Query:

  Find certificate number of the producer of "Star Wars"

- Query in SQL:

  SELECT producerC#

  FROM Movie

  WHERE title = 'Star Wars';

  Assuming that we have only one such movie.

Subqueries

- Conditions in the **WHERE** clause may have comparisons that involve scalar values
- A SQL query can produce a scalar value
- If so, we can use such SELECT-FROM-WHERE expression, surrounded by parentheses, **as if it were a constant**
- **Subquery** – a query within a query

  The result of a SQL subquery is a collection (relation/bag)

Example

- Relation schemas:

  Movie (title, year, length, filmType, studioName, producerC#)
  Exec (name, address, cert#, netWorth)

- Query:

  Find the name of the producer of "Star Wars"

- Query in SQL:

```
SELECT Exec.name
FROM Movie, Exec
WHERE Movie.title = 'Star Wars' AND
        Movie.producerC# = Exec.cert#;
```

Example

- Relation schemas:

  Movie (title, year, length, filmType, studioName, producerC#)
  Exec (name, address, cert#, netWorth)

- Query:

  Find the name of the producer of "Star Wars"

- Query with Subquery:

```
SELECT name
FROM Exec
WHERE cert# = ( SELECT producerC#
                FROM Movie
                WHERE title = 'Star Wars' );
```

Conditions Involving Relations

- There are a number of SQL checks/conditions that can be done on a relation **R** and produce a **boolean** value
- These conditions can be **negated** by putting a **NOT** before them
- Typically, **R** above is the result of an SQL subquery, shown as: **(R)**
- If such a condition involves a scalar value ***s*** or a tuple, we should make sure its type matches **R**.

Conditions Involving Relations

- "**EXISTS (*R*)**" is a condition that is true iff ***R*** is not empty
- "***s* IN (*R*)**" is true iff ***s*** is equal to **one** of the values in ***R***
  - "***s* NOT IN (*R*)**" is true iff ***s*** is not equal to any value in ***R***
- "***s* > ALL (*R*)**" is true iff ***s*** is greater than **every** value in ***R***
  - ">" could be replaced by other operators with the analogous meaning
  - **Note:** "***s* <> ALL (*R*)**" is the same as "***s* NOT IN *R***"
- "***s* > ANY (*R*)**" is true iff ***s*** is > **at least one** value in ***R***
  - ">" could be replaced by any of the other 5 comparison operators with the analogous meaning
  - **Note:** "***s* = ANY *(R)***" is the same as "***s* IN *R***"

Conditions Involving Tuples

- A tuple in SQL is represented by a parenthesized list of scalar values; the concept “tuple” can be viewed as an *extension* of the concept of scalar;
  - (123, ’foo’)
- Mixing of constants and attributes is also permitted in tuples
  - (123, Movie.title)
- If a tuple ***t*** has the same number of components as a relation ***R***, then it makes sense to compare ***t*** and ***R*** like:
  - ***t*** **IN (*R*)** -- this is true iff ***t*** *is in* ***R***
  - ***t*** **<> ANY (*R*)** -- *this is true* ***R*** *includes a tuple other than* ***t***

Example

- Relation schemas:

  Movie (title, year, length, filmType, studioName, producerC#)

  Exec (name, address, cert#, netWorth)

  StarsIn (title, year, starName)

- Query: Find the names of the producers of Harrison Ford's movies
- Query in SQL:

```
SELECT name
FROM Exec
WHERE cert#  IN (SELECT producerC#
                 FROM Movie
                 WHERE (title, year) IN (SELECT title, year
                                         FROM StarsIn
                                         WHERE starName = 'Harrison Ford'));
```

Example

- Relation schemas:

  Movie(title, year, length, filmType, studioName, producerC#)
  Exec(name, address, cert#, netWorth)
  StarsIn(title, year, starName)

- Query: Find names of the producers of Harrison Ford's movies

- Query in SQL:

```
SELECT Exec.name
FROM Exec, Movie, StarsIn
WHERE Exec.cert# = Movie.producerC# AND
                   Movie.title = StarsIn.title AND
                   Movie.year = StarsIn.year AND
                   starName = 'Harrison Ford';
```

Correlated Subqueries

- Simple subqueries can be evaluated once and the result be used in a higher level (calling) query
- A more complex use of nested subquery requires the subquery to be evaluated many times, once for each assignment of a value (to some term in the subquery) that comes from a tuple variable in the calling query
- A subquery of this type is called ***correlated subquery***

Correlated Subqueries

- Relation schema:

  Movie(title, year, length, filmType, studioName, producerC#)

- Query:

  Find movie titles that appear more than once

- Query in SQL:

```
SELECT title
FROM Movie Old
WHERE year < ANY (SELECT year
                  FROM Movie
                  WHERE title = Old.title);
```

Note the scopes of the variables in this query.

Correlated Subqueries

- Query in SQL

```
SELECT title
FROM Movie Old
WHERE year < ANY (SELECT year
                  FROM Movie
                  WHERE title = Old.title);
```

- The condition in the outer WHERE is true only if there is a movie with same title as *Old.title* that has a **later** year
  - ➔ The query will produce a title **one fewer times** than there are movies with that title
- What would be the result if we used "<>", instead of "<" ?
  - ➔ For a movie title appearing 3 times, we would get 3 copies of the title in the output

Views

- **View** is a table/relation defined in a database but has no tuples explicitly stored for it in the database but rather computed, when needed, from the **view definition**
- The view mechanism provides support for:
  - Logical data independence:
    - Views can be used to define relations in the external schema that mask, from the applications/users, changes in the *conceptual database schema*
    - If the schema of a relation is changed, we can define a view with the old schema so that applications that use the old schema can continue using it
  - Security:
    - Views can be used to restricts the users access only the information they are allowed to "see and operate on"

Views

- Relation schema:

  Movie(title, year, length, filmType, studioName, producerC#)

- View:

  Create the Paramount's movies (title and year)

- View in SQL:

```
CREATE VIEW ParamountMovie AS
SELECT title, year
FROM Movie
WHERE studioName = 'Paramount';
```

Views

- A view can be used in defining new queries/views in exactly the same way as an explicitly stored table may be used
- Example to query the (virtual) relation ParamountMovie

  SELECT title
  FROM ParamountMovie
  WHERE year = 1979;

- This query is translated, by the query processor, into:

  SELECT title
  FROM Movie
  WHERE studioName = 'Paramount' **AND** year = 1979;

Views

- Relation schema:

  ParamountMovie (title, year **)**

  StarsIn(title, year, starName)

- Query:

  List the stars of the movies made by Paramount.

- Query in SQL

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

Views

- Relation schema:

  Movie (title, year, length, filmType, studioName, producerC#)
  Exec (name, address, cert#, netWorth)

- View:

  Define a view of Movie (titles and executives/producers)

- View in SQL:

```
CREATE VIEW MovieProd AS
SELECT Movie.title, Exec.name
FROM Movie, Exec
WHERE Movie.producerC# = Exec.cert#;
```

Views

- Relation schema:

  MovieProd (title, name)

- Query:

  Find the name of the producer of ‘Gone With the Wind’?

- Query in SQL:

```
SELECT name
FROM MovieProd
WHERE title = ’Gone With the Wind’;
```

Views

- Renaming attributes used in view definitions
  - We can give new names to view attributes rather than using the names that come out of query defining the view
- Example:

```
CREATE VIEW MovieProd (MovieTitle, ProducerName) AS
SELECT Movie.title, Exec.name
FROM Movie, Exec
WHERE Movie.producerC# = Exec.cert#;
```

Views

- Relation schema:

  MovieProd (MovieTitle, ProducerName)

- Query:

  Find the name of the producer of 'Gone With the Wind'?

- Query in SQL:

```
SELECT ProducerName
FROM MovieProd
WHERE MovieTitle = 'Gone With the Wind';
```

Updating Views?

- We saw that a view can appear in a query in exactly the same way as a “base” table may appear.
- What about modifications/updates?
- What does it mean to update a view?
  - Translate modification of the view to the corresponding modification on the base tables used in the view definition
- Should we allow updates on views?
  - Yes, in principle, but some problems may arise
- Some “simple” views can be updated
  - Such views are called **updatable views**
- Many views cannot be updated
  - This is due to the so called **view-update anomaly**

Insertion into Views?

- Relation schema:

  Movie(title, year, length, filmType, studioName, producerC#)

- View: Recall the definition of ParamountMovie

  CREATE VIEW ParamountMovie **AS**

  SELECT title, year

  FROM Movie

  WHERE studioName = 'Paramount';

- Update statement:

  INSERT INTO ParamountMovie **(title,year) VALUES**('KK', 2002);

- Result: the following tuple being added to Movie

  ('KK', 2002, NULL, NULL, NULL, NULL) What's the problem?

Insertion into Views?

- Relation schema:

  Movie(title, year, length, filmType, studioName, producerC#)

- An updatable view:

  CREATE VIEW ParamountMovie **AS**

  SELECT title, year, studioName

  FROM Movie

  WHERE studioName = 'Paramount';

- Update statement:

  INSERT INTO ParamountMovie **VALUES**('KK',2002,'Paramount');

- Result: the following tuple is being added to Movie

  ('KK', 2002, NULL, NULL, 'Paramount', NULL) Problem solved!

Insertion into Views?

- Relation schemas:

  Movie(title, year, length, filmType, studioName, producerC#)

  Exec(name, address, cert# , netWorth)

- View in SQL:

  CREATE VIEW MovieProd **AS**
  SELECT Movie.title, Exec.name
  FROM Movie, Exec
  WHERE Movie.producerC# = Exec.cert#;

- Update statement

INSERT INTO MovieProd **(title,name) VALUES('The Movie', 'J. Smith');**

- Result: these tuples are added to the corresponding relations:

Movie('The Movie', NULL, NULL, NULL, NULL, NULL)

Exec('J. Smith', NULL, NULL, NULL) Problems? The insertion command will fail !

Deletion from Views?

- Relation schema:

  Movie(title, year, length, filmType, studioName, producerC#)

- View: Recall the definition :

  CREATE VIEW ParamountMovie **AS**

  SELECT title, year, studioName

  FROM Movie

  WHERE studioName = 'Paramount';

- Delete statement:

  DELETE FROM ParamountMovie **WHERE** title **LIKE** '%K%';

- Translated query:

  DELETE FROM Movie

  WHERE studioName = 'Paramount' **AND** title LIKE '%K%';

Updating Views?

- Relation schema:

  Movie(title, year, length, filmType, studioName, producerC#)

- View:

  CREATE VIEW ParamountMovie **AS**

  SELECT title, year,studioName

  FROM Movie

  WHERE studioName = 'Paramount';

- The view update statement:

  UPDATE ParamountMovie **SET** year = 1797 **WHERE** title ='KK';

  - We may drop a view: **DROP VIEW** ParamountMovie;

Updating Views?

- Recall: updating views includes insertion, deletion, and changing data
- SQL provides a formal definition of when modifications to a view are permitted
- Roughly, this is permitted when the view is defined by selecting some attributes from one relation **R**, which could be an "updatable" view itself
  - The list in the **SELECT** clause includes "enough" attributes that for every tuple inserted into the view, the tuple inserted into the base relation will "yield" the inserted tuple of the view
  - The **NOT NULL** constraints on the base table will not be violated
  - The view definition uses **SELECT** (but not **SELECT DISTINCT**)
  - The **WHERE** clause does not involve **R** in a subquery