Fundamentals of Database Systems
COMPSCI 351

Instructor: Sebastian Link

The University of Auckland

SQL
As a Query Language

What are Query Languages?

Query languages (QLs) give us

access and retrieval of data from a database

Relational model supports simple, powerful QLs:

- an operational language that allows DBs to speed up query evaluation
- a declarative language with a strong formal foundation based on logic
- an industry standard language for specifying queries in real DBMSs

Query Languages are not programming languages

- QLs not expected to be “Turing complete”
- QLs not intended to be used for complex calculations
- QLs support easy, efficient access to large data sets

Formal Relational Query Languages

Two mathematical QLs form the basis for

- industry standards (e.g. SQL), and
- implementation

Relational Algebra

more operational, very useful for representing execution plans

Relational Calculus

- lets users describe what they want, rather than how to compute it
- i.e., non-operational, but declarative
- simple to write queries, and have them translated into SQL

Both languages can essentially express the same queries

Query Languages Defined

Definition (Query Language (QL))

A **query language** is

- a formal language $\mathcal{L}$ such that
- each query $Q \in \mathcal{L}$ is associated with
    - an *input-schema* $\mathit{in}(Q)$,
    - an *output-schema* $\mathit{out}(Q)$, and
    - a *query mapping* $q(Q) : \mathit{inst}(\mathit{in}(Q)) \rightarrow \mathit{inst}(\mathit{out}(Q))$

      $\hookrightarrow$ i.e., taking databases over $\mathit{in}(Q)$ to databases over $\mathit{out}(Q)$

Comparison of Query Languages

Two queries Q_1 and Q_2 are equivalent (denoted $Q_1 \equiv Q_2$), if

- they have the same input-schema $in(Q_1) = in(Q_2)$,
- the same output-schema $out(Q_1) = out(Q_2)$, and
- the same query mapping $q(Q_1) = q(Q_2)$, i.e.,

  $\hookrightarrow$ for each database $db \in inst(in(Q_1))$ we have $q(Q_1)(db) = q(Q_2)(db)$

Let $\mathcal{L}$ and $\mathcal{L}'$ be QLs

- $\mathcal{L}'$ **dominates** $\mathcal{L}$ (notation: $\mathcal{L} \sqsubseteq \mathcal{L}'$), if

  $\hookrightarrow$ for each query $Q \in \mathcal{L}$ there exists a query $Q' \in \mathcal{L}'$ with $Q' \equiv Q$
- $\mathcal{L}'$ and $\mathcal{L}$ are **equivalent** (notation: $\mathcal{L} \equiv \mathcal{L}'$), if

  $\hookrightarrow$ $\mathcal{L} \sqsubseteq \mathcal{L}'$ and $\mathcal{L}' \sqsubseteq \mathcal{L}$ both hold

Example: The Same Query in Different Languages

Consider our database schema from before

- MOVIE(title, year, country, run_time, genre), DIRECTOR(id, title, year)
- PERSON(id, first_name, last_name, year_born), ACTOR(id, title, year, role)

Query in English

What are the movies directed by 'Akira Kurosawa'?

SQL

```
SELECT   m.title, m.year
FROM     MOVIE m, DIRECTOR d, PERSON p
WHERE    m.title=d.title AND m.year=d.year AND
         d.id=p.id AND p.first_name='Akira' AND p.last_name='Kurosawa';
```

Example: The Same Query in Different Languages

Consider our database schema from before

- MOVIE(title, year, country, run_time, genre), DIRECTOR(id, title, year)
- PERSON(id, first_name, last_name, year_born), ACTOR(id, title, year, role)

Query in English

What are the movies directed by 'Akira Kurosawa'?

Relational algebra

$$\pi_{\text{title,year}}(\sigma_{\text{last\_name='Kurosawa'}}(\sigma_{\text{first\_name='Akira'}}(\textsc{Movie} \bowtie \textsc{Director} \bowtie \textsc{Person})))$$

Example: The Same Query in Different Languages

Consider our database schema from before

- MOVIE(title, year, country, run_time, genre), DIRECTOR(id, title, year)
- PERSON(id, first_name, last_name, year_born), ACTOR(id, title, year, role)

Query in English

What are the movies directed by 'Akira Kurosawa'?

Relational calculus

$$\{(x_{\text{title}}, x_{\text{year}}) \mid \exists x_{\text{country}}, x_{\text{run\_time}}, x_{\text{genre}}, x_{\text{id}}, x_{\text{year\_born}} (\textsc{Movie}(x_{\text{title}}, x_{\text{year}}, x_{\text{country}}, x_{\text{run\_time}}, x_{\text{genre}}) \wedge \textsc{Director}(x_{\text{id}}, x_{\text{title}}, x_{\text{year}}) \wedge \textsc{Person}(x_{\text{id}}, \text{'Akira'}, \text{'Kurosawa'}, x_{\text{year\_born}}))\}$$

SELECT... FROM... WHERE...

The basic form for all queries is given by the `SELECT` statement

`SELECT` ⟨attribute-list⟩ `FROM` ⟨table-list⟩ `WHERE` ⟨condition⟩ ;

Comments

- $*$ in the attribute-list means all attributes
- Attribute names may be qualified with the table name
  - required, if attribute-names are not unique
- Attribute and table names can be renamed: ⟨name⟩ `AS` ⟨short name⟩
  - in this case the short name is required for qualification

Duplicates and Order

SQL is based on multisets (instead of sets)

`SELECT DISTINCT` removes duplicates

The result may be ordered adding an `ORDER BY` clause

`ORDER BY` ⟨attribute-list⟩

Add `ASC` or `DESC` for ascending or descending order

Example

English Language Query

- List all people with their id, first name, last name and the year they were born
- Provided their id is between 221 and 300 or greater than 999.
- Order the list first by last name, then by first name.

Corresponding SQL Query

```
SELECT     *
FROM       PERSON
WHERE      (id >= 221 AND id <= 300) OR id >= 1000
ORDER BY   last_name, first_name;
```

Example

English Language Query

List the first and last name of all movie directors of non-US movies together with the titles and production years of these movies.

Corresponding SQL Query

```
SELECT  p.first_name, p.last_name, m.title, m.year
FROM    MOVIE m, PERSON p, DIRECTOR d
WHERE   d.title = m.title AND m.country <> 'USA' AND
        d.year = m.year AND d.id = p.id;
```

Simple `WHERE`-Conditions

Condition in `WHERE`-clause (in queries, deletes, updates)

- Can be a simple comparison: $\langle$expression$\rangle$ $\langle$operator$\rangle$ $\langle$expression$\rangle$
- The comparison operators can be $=$, $<>$, $<=$, $>=$, $>$ or $<$
- Expressions built from attributes, constant values and operators defined for the corresponding domains (see manuals for details)

More Complex `WHERE`-Conditions

Other conditions

- value lists:
  - ↪ ⟨attribute⟩ `BETWEEN` ⟨value⟩ `AND` ⟨value⟩, or
  - ↪ ⟨attribute⟩ `IN` ⟨value-list⟩
- `IS NULL` and `IS NOT NULL` test for null markers
- ⟨attribute⟩ `LIKE` ⟨string⟩ allows pattern matching
  - ↪ uses _ for a single symbol and % for any substring

`WHERE`-clause may be **complex condition** using `AND`, `OR` and `NOT`

Example

English Language Query

List all movies together with their directors, provided the last name of the director begins with 'H' and ends with an 'o' followed by two more letters.

Corresponding SQL Query

```
SELECT   m.title, p.first_name, p.last_name
FROM     MOVIE m, PERSON p, DIRECTOR d
WHERE    m.title = d.title AND m.year = d.year AND
         d.id = p.id AND p.last_name LIKE 'H%o__';
```

Example

English Language Query

List all movies produced between 1978 and 1995, ordered by year

Corresponding SQL Query

```
SELECT      title, year
FROM        MOVIE
WHERE       year BETWEEN 1978 and 1995
ORDER BY    year;
```

Aggregate functions

Aggregate functions applied to attributes or attribute lists result in single output value

Some functions

- `COUNT` returns the total number of input values
- `AVG` returns the average of input values, given some number domain
- `MIN` returns the minimum value of the inputs
- `MAX` returns the maximum value of the inputs
- `SUM` returns the sum of the input values, given some number domain

Several functions can be used in a query, but not in combination with simple attributes

The result is significantly different, if `DISTINCT` is used

Example

English language query

How many actors are stored in the database?

Corresponding SQL query

```
SELECT   COUNT (DISTINCT a.id) AS number_of_actors
FROM     ACTOR a
```

Example

English language query

List minimum, average and maximum run-time of all non-German movies

Corresponding SQL query

```
SELECT   MIN (run_time) AS min_run_time,
         AVG (run_time) AS avg_run_time,
         MAX (run_time) AS max_run_time
FROM     MOVIE
WHERE    country <> 'Germany';
```

Grouping

Apply aggregate function to selected groups of tuples, for instance the number of actors for each movie

`GROUP BY` ⟨attribute-list⟩

for building groups of tuples for each value combination in the attribute-list

The attributes in the attribute-list following the `GROUP BY` must be exactly the simple attributes in the list following the `SELECT` clause

Example

English language query

List each movie produced in 1999 together with its number of actors

Corresponding SQL query

```
SELECT     m.title, m.year COUNT(DISTINCT a.id) AS number_of_actors
FROM       MOVIE m, ACTOR a
WHERE      a.title = m.title AND a.year = 1999 AND m.year = 1999
GROUP BY   m.title, m.year;
```

The `HAVING` clause

The clause `HAVING` ⟨condition⟩ evaluates a condition on the groups

The `HAVING` condition must apply to the group, not to individual tuples in the group

Example

List for all countries except Australia the number of movies, if there are at least two.

```
SELECT     m.country, COUNT(m.title) AS number_of_movies
FROM       MOVIE m
WHERE      m.country <> 'Australia'
GROUP BY   m.country
HAVING     COUNT(*) >= 2;
```

Sub-queries

Sub-queries

are just queries that are used within `WHERE`-conditions

`IN` ⟨sub-query⟩

tests if a tuple occurs in the result of a sub-query

`EXISTS` ⟨sub-query⟩

tests whether sub-query results in non-empty relation

Sub-queries continued

UNIQUE ⟨sub-query⟩

tests if result of sub-query contains no duplicates

ALL, SOME or ANY (SOME=ANY)

before a sub-query makes sub-queries usable in comparison formulae

Negation

In all these cases the condition involving the sub-query can be negated using a preceding NOT

Example

English Language Query

List for all countries except Spain the movies produced in that country, provided there are at least two of them. Order the result first ascending by country, then descending by the production year of the movie.

Corresponding SQL Query

```
SELECT     m1.country, m1.year, m1.title
FROM       MOVIE m1
WHERE      m1.country <> 'Spain' AND m1.country IN (
                SELECT m2.country
                FROM MOVIE m2
                GROUP  BY m2.country
                HAVING COUNT(*) >= 2)
ORDER  BY  m1.country ASC, m1.year DESC;
```

Example

English Language Query

List the movies with exactly one actor, and return the first and last name of this actor.

Corresponding SQL Query

```
SELECT  m.title, m.year, p.first_name, p.last_name
FROM    MOVIE m, ACTOR a1, PERSON p
WHERE   m.title = a1.title AND m.year = a1.year AND a1.id = p.id AND NOT EXISTS(
            SELECT *
            FROM Actor a2
            WHERE a2.id <> a1.id AND a2.title = a1.title AND a2.year = a1.year);
```

Example

English Language Query

List the NZ movies together with their run time where the run time is longer than any Australian movie.

Corresponding SQL Query

```
SELECT   m1.title, m1.year, m1.run_time
FROM     MOVIE m1
WHERE    m1.country = 'New Zealand' AND m1.run_time > ANY (
                SELECT m2.run_time
                FROM MOVIE m2
                WHERE m2.country='Australia');
```

Example

English Language Query

List the actors who played in most movies

Corresponding SQL Query

```
SELECT  p.first_name, p.last_name, p.id, a2.num_of_movies
FROM    PERSON p1, (
          SELECT a1.id, COUNT(DISTINCT a1.title, a1.year) AS num_of_movies
          FROM ACTOR a1
          GROUP BY a1.id ) a2
WHERE   p.id = a2.id AND a2.num_of_movies = (
          SELECT MAX(a4.num_of_movies)
          FROM (SELECT a3.id, COUNT(DISTINCT a3.title, a3.year) as num_of_movies
                FROM ACTOR a3
                GROUP BY a3.id ) a4);
```

Limitations of SQL

FLIGHTS

Origin	Destination
Singapore	Tokyo
Singapore	Frankfurt
Singapore	Auckland
Tokyo	Joburg
⋮	

Reachability Query:
Find pairs of cities (O, D) such that one can fly from O to D with at most one stop

```
  SELECT   F1.Origin, F2.Destination
  FROM     FLIGHTS F1, FLIGHTS F2
  WHERE    F1.Destination = F2.Origin
UNION
  SELECT   *
  FROM     FLIGHTS;
```

Another Reachability Query

Find pairs of cities (O, D) such that one can fly from O to D with at most two stops

```
    SELECT   F1.Origin, F3.Destination
    FROM     FLIGHTS F1, FLIGHTS F2, FLIGHTS F3
    WHERE    F1.Destination = F2.Origin AND F2.Destination = F3.Origin
UNION
    SELECT   F1.Origin, F2.Destination
    FROM     FLIGHTS F1, FLIGHTS F2
    WHERE    F1.Destination = F2.Origin
UNION
    SELECT   *
    FROM     FLIGHTS;
```

Reachability Queries in General

For any fixed k, we can write in SQL

Find (O, D) so one can fly from O to D with at most k stops.

What about general reachability?

Find pairs of cities (O, D) such that one can fly from O to D.

SQL'99 solution

New construct to express reachability

SQL'92

is not able to express this query

This feature is not implemented in all products

A Different Point of View

Rewrite reachability queries as rules

$$\begin{aligned} \textsc{Reach}(x,y) & \;:-\textsc{Flights}(x,y) \\ \textsc{Reach}(x,y) & \;:-\textsc{Flights}(x,z), \textsc{Reach}(z,y) \end{aligned}$$

Latter rule is *recursive*: $\textsc{Reach}$ refers to itself

Evaluation

- Step 0: $\textsc{Reach}_0$ is initialised as the empty set
- Step $i+1$:

$$\begin{aligned} \textsc{Reach}_{i+1}(x,y) & \;:-\textsc{Flights}(x,y) \\ \textsc{Reach}_{i+1}(x,y) & \;:-\textsc{Flights}(x,z), \textsc{Reach}_i(z,y) \end{aligned}$$

- Stop condition:
  if $\textsc{Reach}_{i+1} = \textsc{Reach}_i$, then return it as answer of query

Example of Evaluation

FLIGHTS

Origin	Destination
Singapore	Tokyo
Tokyo	Joburg
Joburg	Rio

Evaluation steps

Stop condition: $\textsc{Reach}_4 = \textsc{Reach}_3$

REACH

{(Singapore,Tokyo), (Tokyo,Joburg), (Joburg,Rio), (Singapore, Joburg), (Tokyo, Rio), (Singapore, Rio)}

Recursion with SQL'99

SQL'99 syntax mimics that of recursive rules

```
WITH RECURSIVE REACH(Origin,Destination) AS
(
   SELECT *
   FROM FLIGHTS
UNION
   SELECT F.Origin, R.Destination
   FROM FLIGHTS F, REACH R
   WHERE F.Destination = R.Origin
)
SELECT *
FROM REACH
```

Recursion in SQL'99: Syntactic restrictions

A different way to describe reachability as a recursive rule-based query:

$$\textsc{Reach}(x, y) :- \textsc{Flights}(x, y)$$
$$\textsc{Reach}(x, y) :- \textsc{Reach}(x, z), \textsc{Reach}(z, y)$$

This translates into an SQL'99 query

```
WITH RECURSIVE REACH(Origin,Destination) AS (
   SELECT * FROM FLIGHTS
UNION
   SELECT R1.Origin, R2.Destination
   FROM REACH R1, REACH R2
   WHERE R1.Destination = R2.Origin)
SELECT * FROM REACH
```

Many implementations support *linear* recursion only

recursively defined relation mentioned just once in FROM line

Another Example of Recursion in SQL'99

Query: Find cities reachable from Singapore

Assume FLIGHTS contains the further attribute Aircraft

```
WITH CITIES AS SELECT Origin, Destination FROM FLIGHTS
   RECURSIVE REACH(Origin,Destination) AS
(
   SELECT * FROM CITIES
UNION
   SELECT C.Origin, R.Destination
   FROM CITIES C, REACH R
   WHERE C.Destination = R.Origin
)
SELECT R.Destination
FROM REACH R
WHERE R.Origin = "Singapore"
```

Problems with Negation in Recursion

Consider rule such as $R(X) :- S(X), \neg R(X)$ in SQL

```
WITH RECURSIVE R(A) AS
   (SELECT S.A
    FROM S
    WHERE S.A NOT IN (SELECT R.A FROM R))
SELECT * FROM R
```

For $S = \{1, 2\}$ evaluation leads to...

after step 0: $R_0 = \emptyset$;
after step 1: $R_1 = \{1, 2\}$;
$\dots$
after step $2n$: $R_{2n} = \emptyset$;
after step $2n+1$: $R_{2n+1} = \{1, 2\}$;

Problem: the evaluation does not terminate

negation (`NOT IN`) causes this problem

Summary for SQL

- SQL is the industry standard for defining and querying data
- DDL of SQL allows us to declare tables, views, and constraints
- DML of SQL allows us to specify and execute queries and updates
- SQL permits duplicate and partial information

  $\hookrightarrow$ uses list semantics rather than set semantics of relational model
- SQL has several additional features, including

  $\hookrightarrow$ Indices

  $\hookrightarrow$ Access rights

  $\hookrightarrow$ Aggregate functions

  $\hookrightarrow$ Transitive closures

  $\hookrightarrow$ Embedded SQL

  $\hookrightarrow$ Dynamic SQL

  $\hookrightarrow$ Transactions