

CIS 330: Applied Database Systems

Lecture 7: SQL

Johannes Gehrke
johannes@cs.cornell.edu
<http://www.cs.cornell.edu/johannes>

Logistics

- Office hours role call:
 - Mondays, 3-4pm
 - Tuesdays, 4:30-5:30
 - Wednesdays, 4:15-5:15pm
 - Fridays, 11am-noon
- Handout of CDs for next assignment

The SQL Query Language

- Developed by IBM (system R) in the 1970s
- Need for a standard since it is used by many vendors
- Standards:
 - SQL-86
 - SQL-89 (minor revision)
 - SQL-92 (major revision)
 - SQL-99 (major extensions, current standard)

Example Instances

| R1 | | |
|------------|------------|------------|
| <u>sid</u> | <u>bid</u> | <u>day</u> |
| 22 | 101 | 10/10/96 |
| 58 | 103 | 11/12/96 |

- We will use these instances of the Sailors and Reserves relations in our examples.

| S1 | | | |
|------------|--------|--------|------|
| <u>sid</u> | sname | rating | age |
| 22 | dustin | 7 | 45.0 |
| 31 | lubber | 8 | 55.5 |
| 58 | rusty | 10 | 35.0 |

- If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

| S2 | | | |
|------------|--------|--------|------|
| <u>sid</u> | sname | rating | age |
| 28 | yuppy | 9 | 35.0 |
| 31 | lubber | 8 | 55.5 |
| 44 | guppy | 5 | 35.0 |
| 58 | rusty | 10 | 35.0 |

Basic SQL Query

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
```

- relation-list**: A list of relation names (possibly with a range-variable after each name).
- target-list**: A list of attributes of relations in relation-list
- qualification**:
 - Comparisons: Attr op const or Attr1 op Attr2, where op is one of the following:
 $<$, $>$, $=$, \leq , \geq , \neq
 combined using AND, OR and NOT.
- DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
 - Compute the cross-product of *relation-list*.
 - Discard resulting tuples if they fail *qualifications*.
 - Delete attributes that are not in *target-list*.
 - If **DISTINCT** is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Example of Conceptual Evaluation

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
```

| (sid) | sname | rating | age | (sid) | bid | day |
|-------|--------|--------|------|-------|-----|----------|
| 22 | dustin | 7 | 45.0 | 22 | 101 | 10/10/96 |
| 22 | dustin | 7 | 45.0 | 58 | 103 | 11/12/96 |
| 31 | lubber | 8 | 55.5 | 22 | 101 | 10/10/96 |
| 31 | lubber | 8 | 55.5 | 58 | 103 | 11/12/96 |
| 58 | rusty | 10 | 35.0 | 22 | 101 | 10/10/96 |
| 58 | rusty | 10 | 35.0 | 58 | 103 | 11/12/96 |

A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND bid=103
```

OR

```
SELECT sname
FROM Sailors, Reserves
WHERE Sailors.sid=Reserves.sid
AND bid=103
```

*It is good style,
however, to use
range variables
always!*

Find sailors who have reserved at least one boat

```
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
```

- Would adding **DISTINCT** to this query make a difference?
- What is the effect of replacing S.sid by S.sname in the SELECT clause? Would adding **DISTINCT** to this variant of the query make a difference?

Expressions and Strings

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

- Illustrates use of arithmetic expressions and string pattern matching: Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. `_' stands for any one character and `%` stands for 0 or more arbitrary characters.

Find sid's of sailors who've reserved a red or a green boat

- **UNION:** Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
AND (B.color='red' OR B.color='green')
```
- If we replace **OR** by **AND** in the first version, what do we get?

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
AND B.color='red'
```
- Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

```
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
AND B.color='green'
```

Find sid's of sailors who've reserved a red and a green boat

- **INTERSECT:** Can be used to compute the intersection of any two *union-compatible* sets of tuples.

```
SELECT S.sid
FROM Sailors S, Boats B1, Reserves R1,
Boats B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
AND S.sid=R2.sid AND R2.bid=B2.bid
AND (B1.color='red' AND B2.color='green')
```
- Included in the SQL/92 standard, but some systems don't support it.
- Contrast symmetry of the **UNION** and **INTERSECT** queries with how much the other versions differ.

```
SELECT S.sid      Key field!
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
AND B.color='green'
```

In-Class Exercise

Suppliers(sid:integer, sname:string,
address:string)

Parts(pid:integer, pname:string, color:string)

Catalog(sid:integer, pid:integer, cost: real)

- Find the *pnames* of parts for which there is some supplier.
- Find the *sids* of suppliers who supply a red part and a green part.
- Find the *sids* of suppliers who supply a red part or a green

Back to SQL: Nested Queries

Find names of sailors who've reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid=103)
```

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- To find sailors who've not reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a nested loops evaluation: For each Sailors tuple, check the qualification by computing the subquery.

Nested Queries with Correlation

Find names of sailors who've reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
              FROM Reserves R
              WHERE R.bid=103 AND S.sid=R.sid)
```

- **EXISTS** is another set comparison operator, like **IN**.
- If **UNIQUE** is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103. (**UNIQUE** checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by *R.bid*?)
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op ANY, op ALL, op IN* >, <, =, ≥, ≤, ≠
- Find sailors whose rating is greater than that of some sailor called Horatio:

```
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
                     FROM Sailors S2
                     WHERE S2.sname='Horatio')
```

Rewriting INTERSECT Queries Using IN

Find sid's of sailors who've reserved both a red and a green boat:

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
AND S.sid IN (SELECT S2.sid
             FROM Sailors S2, Boats B2, Reserves R2
             WHERE S2.sid=R2.sid AND R2.bid=B2.bid
             AND B2.color='green')
```

- Similarly, EXCEPT queries re-written using NOT IN.
- To find *names* (not *sids*) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in SELECT clause. (What about INTERSECT query?)

Division

- Not supported as a primitive operator, but useful for expressing queries like:
Find sailors who have reserved all boats.
- Let *A* have 2 fields, *x* and *y*, *B* have only field *y*:
 - $A/B = \{ \langle x \rangle \mid \exists \langle x, y \rangle \in A \ \forall \langle y \rangle \in B \}$
 - i.e., *A/B* contains all *x* tuples (sailors) such that for every *y* tuple (boat) in *B*, there is an *xy* tuple in *A*.
 - Or: If the set of *y* values (boats) associated with an *x* value (sailor) in *A* contains all *y* values in *B*, the *x* value is in *A/B*.
- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and $x \cup y$ is the list of fields of *A*.

Examples of Division A/B

| sno | pno | pno | pno | pno |
|-----|------|------|------|-----|
| s1 | p1 | p2 | p2 | p1 |
| s1 | p2 | B1 | p4 | p2 |
| s1 | p3 | | B2 | p4 |
| s1 | p4 | | | B3 |
| s2 | p1 | sno | | |
| s2 | p2 | s1 | | |
| s3 | p2 | s2 | sno | |
| s4 | p2 | s3 | s1 | sno |
| s4 | p4 | s4 | s4 | s1 |
| A | A/B1 | A/B2 | A/B3 | |

Expressing A/B Using Basic Operators

- Division is not essential op; just a useful shorthand.
 - (Also true of joins, but joins are so common that systems implement joins specially.)
- *Idea:* For A/B , compute all x values that are not 'disqualified' by some y value in B .
 - x value is *disqualified* if by attaching y value from B , we obtain an xy tuple that is not in A .

Disqualified x values: $\pi_x((\pi_x(A) \times B) - A)$

A/B : $\pi_x(A) - \text{all disqualified tuples}$

Find the names of sailors who've reserved all boats

- Uses division; schemas of the input relations to $/$ must be carefully chosen:

$$\rho (Temp\ sid, (\pi_{sid, bid} Reserves) / (\pi_{bid} Boats))$$

$$\pi_{sname}(Temp\ sid \bowtie Sailors)$$
- To find sailors who've reserved all 'Interlake' boats:

$$\dots / \pi_{bid}(\sigma_{bname = 'Interlake'} Boats)$$

Division in SQL

```
(1) SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS
  ((SELECT B.bid
   FROM Boats B)
  EXCEPT
  (SELECT R.bid
   FROM Reserves R
   WHERE R.sid=S.sid))
```

Find sailors who've reserved all boats.

- Let's do it the hard way, without EXCEPT:

```
(2) SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (SELECT B.bid
                  FROM Boats B
                  WHERE NOT EXISTS (SELECT R.bid
                                    FROM Reserves R
                                    WHERE R.bid=B.bid
                                    AND R.sid=S.sid))
```

Sailors S such that ...
there is no boat B without ...
a Reserves tuple showing S reserved B

In-Class Exercise

Suppliers(sid:integer, sname:string, address:string)
Parts(pid: integer, pname:string, color:string)
Catalog(sid: integer, pid: integer, cost: real)

- Find the *snames* of suppliers who supply every part.
- Find the *snames* of suppliers who supply every red part.
- Find the *pnames* of parts supplied by Acme Widget Suppliers and no one else.

Aggregate Operators

- Significant extension of relational algebra.

```
COUNT (*)
COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)
```

single column

```
SELECT COUNT (*)
FROM Sailors S
SELECT S.sname
FROM Sailors S
WHERE S.rating= (SELECT MAX(S2.rating)
                 FROM Sailors S2)

SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating=10

SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'

SELECT AVG (DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10
```

Find name and age of the oldest sailor(s)

- The first query is illegal! (We'll look into the reason a bit later, when we discuss **GROUP BY**.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
SELECT S.sname, MAX (S.age)
FROM Sailors S

SELECT S.sname, S.age
FROM Sailors S
WHERE S.age =
      (SELECT MAX (S2.age)
       FROM Sailors S2)

SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
       FROM Sailors S2)
      = S.age
```

GROUP BY and HAVING

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the age of the youngest sailor for each rating level.*
 - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
 - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

```
For i = 1, 2, ..., 10:
SELECT MIN (S.age)
FROM Sailors S
WHERE S.rating = i
```

Queries With GROUP BY and HAVING

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

- The *target-list* contains (i) **attribute names** (ii) terms with aggregate operations (e.g., `MIN (S.age)`).
- The **attribute list (i)** must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation

- The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, 'unnecessary' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a *single value per group!*
 - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Find the age of the youngest sailor with age ≥ 18 , for each rating with at least 2 such sailors

```
SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```

| sid | sname | rating | age |
|-----|---------|--------|------|
| 22 | dustin | 7 | 45.0 |
| 31 | lubber | 8 | 55.5 |
| 71 | zorba | 10 | 16.0 |
| 64 | horatio | 7 | 35.0 |
| 29 | brutus | 1 | 33.0 |
| 58 | rusty | 10 | 35.0 |

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes 'unnecessary'.
- 2nd column of result is unnamed. (Use AS to name it.)

| rating | age |
|--------|------|
| 1 | 33.0 |
| 7 | 45.0 |
| 7 | 35.0 |
| 8 | 55.5 |
| 10 | 35.0 |

Answer relation

For each red boat, find the number of reservations for this boat

```
SELECT B.bid, COUNT(*) AS scount
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```

- Grouping over a join of three relations.
- What do we get if we remove *B.color='red'* from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?

Find the age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age > 18
GROUP BY S.rating
HAVING 1 < (SELECT COUNT (*)
            FROM Sailors S2
            WHERE S.rating=S2.rating)
```

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18!
- What if HAVING clause is replaced by:
 - HAVING COUNT(*) >1

Find those ratings for which the average age is the minimum over all ratings

- Aggregate operations cannot be nested! **WRONG:**

```
SELECT S.rating
FROM Sailors S
WHERE S.age = (SELECT MIN (AVG (S2.age)) FROM Sailors S2)
```

- ❖ Correct solution (in SQL/92):

```
SELECT Temp.rating, Temp.avgage
FROM (SELECT S.rating, AVG (S.age) AS avgage
      FROM Sailors S
      GROUP BY S.rating) AS Temp
WHERE Temp.avgage = (SELECT MIN (Temp.avgage)
                    FROM Temp)
```

In-Class Exercise

Suppliers(sid:integer, sname:string, address:string)

Parts(pid: integer, pname:string, color:string)

Catalog(sid: integer, pid: integer, cost: real)

- Find the *sids* of suppliers who charge more for some part than the average cost of that part (averaged over all the suppliers who supply that part).
- For each part, find the *sname* of the supplier who charges the most for that part.
- Find the *sids* of suppliers who supply only red parts.
- For every supplier that only supplies green parts, print the name of the supplier and the total number of parts that she supplies.
- For every supplier that supplies a green part and a red part, print the name and price of the most expensive part that she supplies.

Null Values

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).
 - SQL provides a special value *null* for such situations.
- The presence of *null* complicates many issues. E.g.:
 - Special operators needed to check if value is/is not *null*.
 - Is *rating > 8* true or false when *rating* is equal to *null*? What about **AND**, **OR** and **NOT** connectives?
 - We need a **3-valued logic** (true, false and *unknown*).
 - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
 - New operators (in particular, *outer joins*) possible/needed.

Integrity Constraints (Review)

- An IC describes conditions that every *legal instance* of a relation must satisfy.
 - Inserts/deletes/updates that violate IC's are disallowed.
 - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- **Types of IC's:** Domain constraints, primary key constraints, foreign key constraints, general constraints.
 - **Domain constraints:** Field values must be of right type. Always enforced.

General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

```
CREATE TABLE Sailors
(sname CHAR(10),
rating INTEGER,
age REAL,
PRIMARY KEY (sid),
CHECK (rating >= 1
AND rating <= 10))

CREATE TABLE Reserves
(sname CHAR(10),
bid INTEGER,
day DATE,
PRIMARY KEY (bid,day),
CONSTRAINT noInterlakeRes
CHECK ('Interlake' <>
(SELECT B.bname
FROM Boats B
WHERE B.bid=bid)))
```

Constraints Over Multiple Relations

- ```
CREATE TABLE Sailors
 (sid INTEGER,
 sname CHAR(10),
 rating INTEGER,
 age REAL,
 PRIMARY KEY (sid),
 CHECK
 ((SELECT COUNT (S.sid) FROM Sailors S)
 + (SELECT COUNT (B.bid) FROM Boats B) < 100))
```
- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.
- ```
CREATE ASSERTION smallClub
  CHECK
    ((SELECT COUNT (S.sid) FROM Sailors S)
     + (SELECT COUNT (B.bid) FROM Boats B) < 100))
```

*Number of boats
plus number of
sailors is < 100*

Triggers

- Trigger: Procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
 - Event (activates the trigger)
 - Condition (tests whether the triggers should run)
 - Action (what happens if the trigger runs)

Triggers: Example (SQL:1999)

```
CREATE TRIGGER youngSailorUpdate
  AFTER INSERT ON SAILORS
  REFERENCING NEW TABLE NewSailors
  FOR EACH STATEMENT
  INSERT
    INTO YoungSailors(sid, name, age, rating)
  SELECT sid, name, age, rating
  FROM NewSailors N
  WHERE N.age <= 18
```

Lecture Summary

- SQL

- Next lecture: Normalization
