SQL: Queries, Programming, Triggers

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

S1

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

S2

sid	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
```

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

Conceptual Evaluation Strategy

❖ Semantics of an SQL query based on R.A:

SELECT R.A,S.B

FROM R, S

WHERE R.C=S.C

==============>

$\Pi_{R.A,S.B}\sigma_{R.C=S.C}(R \times S)$

Example of Conceptual Evaluation

```
SELECT  S.sname
FROM   Sailors S, Reserves R ---->range variable
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've 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 <u>or</u> 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).
- ❖ If we replace **OR** by **AND** in the first version, what do we get?
- ❖ Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

```
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')
```

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

Find sid's of sailors who've reserved a red <u>and</u> a green boat

- ❖ INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- ❖ Included in the SQL/92 standard, **but some systems don't support it.**

```
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')
```

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

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, **not supported by all systems.)**
- ❖ To find sailors who've ***not*** reserved #103, use **NOT IN.**
- ❖ To understand semantics of nested queries, think of a <u>nested loops</u> 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 $>,<,=,\geq,\leq,\neq$
- 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 *sid*'s) 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 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

Aggregate Operators

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

single column

- Significant extension of relational algebra.

```
SELECT COUNT (*)
FROM  Sailors S
```

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

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

```
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*.)

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

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

rating	
7	35.0

Answer relation

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

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.

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!

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

- Aggregate operations cannot be nested!
- 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)
```

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
        ( sid  INTEGER,
        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

- ❖ 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 TABLE  Sailors
        ( sid  INTEGER,
        sname  CHAR(10),
        rating  INTEGER,
        age  REAL,
        PRIMARY KEY  (sid)
        )

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 (ECA rules):
  - 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
```

Summary

- SQL was an important factor in the early acceptance of the relational model; more natural than earlier, procedural query languages.
- Relationally complete; in fact, significantly more expressive power than relational algebra.
- Even queries that can be expressed in RA can often be expressed more naturally in SQL.
- Many alternative ways to write a query; optimizer should look for most efficient evaluation plan.
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.

Summary (Contd.)

- NULL for unknown field values brings many complications
- SQL allows specification of rich integrity constraints
- Triggers respond to changes in the database