SQL: Queries, Constraints, Triggers

Chapter 5

Example Instances

R1

sid	bid	day
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

sid	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
```

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

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  1. Compute the cross-product of relation-list.
  2. Discard resulting tuples if they fail qualifications.
  3. Delete attributes that are not in target-list.
  4. If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query…
- Optimizer should 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'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, i.e., could a sailor returned by the original version disappear or could a new sailor appear?
- 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 (age of sailor 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 the result.
- LIKE is used for string matching
  - `_' stands for any one character
  - `%' stands for 0 or more arbitrary characters.

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

- UNION: Computes the union of any two union-compatible sets (which can themselves be 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
      AND B.color='red'
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: Computes intersection of any two union-compatible sets of tuples.
- Included in the SQL/92 standard, but some systems do not support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```
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')
```

Key field!

```
SELECT S.sid
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'
```


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)
```

- Very powerful feature of SQL: WHERE clause can itself contain an SQL query
  - And so can FROM and HAVING clauses.
- To find sailors who have 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 tests if the set is empty.
- If UNIQUE is used, and * is replaced by R.bid, finds sailors with at most one reservation for boat #103.
  - UNIQUE returns true if there are no duplicates in the result set.
  - Why do we have to replace * by R.bid for that query version?
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

More on Set-Comparison Operators

- Seen so far: IN, EXISTS, UNIQUE
- Can also use NOT IN, NOT EXISTS, NOT UNIQUE.
- Also available: op ANY, op ALL, where op is <, >, =, ≤, ≥, or ≠
  - Note: IN same as = ANY, NOT IN same as ≠ ALL
- 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?)

Review: Division Operator

A

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

Division in SQL

Find sailors who've reserved all boats.

(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))
```

- 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)

- First query is illegal. (Discussed in more depth later for GROUP BY.)
- Second query has implicit type cast (Which?)
- Third query is equivalent to second query
  - Allowed in the SQL/92 standard
  - But 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
```

Motivation for Grouping

- So far: Have applied aggregate operators to all (qualifying) tuples
- May want to apply them to each of several groups of tuples.
- E.g., 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
```

- target-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (S.age)).
  - Attributes used in target-list must be in grouping-list.
    - 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

1. Compute cross-product of relation-list.
2. Discard tuples that fail qualification.
3. Delete `unnecessary' fields.
4. Partition remaining tuples into groups by the value of attributes in grouping-list.
5. Apply group-qualification to eliminate some groups.
   - Expressions in group-qualification must have a single value per group.
   - 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 age of youngest sailor with age≥18 for each rating with at least 2 such sailors

```
SELECT S.rating, MIN (S.age)
                      AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) >= 2
```

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

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

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

→

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

→

rating	minage
3	25.5
7	35.0
8	25.5

Note: irrelevant attributes omitted on this and following slides.

Find age of the youngest sailor with age≥18, for each rating with at least 2 such sailors and with every sailor under 60.

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

→

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

→

rating	minage
7	35.0
8	25.5

What is the result of changing EVERY to ANY?

HAVING COUNT (*) >= 2 AND EVERY (S.age <=60)

Find age of the youngest sailor with age≥18, for each rating with at least 2 sailors between 18 and 60.

```
SELECT S.rating, MIN (S.age)
            AS minage
FROM Sailors S
WHERE S.age >= 18 AND S.age <= 60
GROUP BY S.rating
HAVING COUNT (*) >= 2
```

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

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 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 2 <= (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 of age≥18!
  - What if HAVING clause is replaced by HAVING COUNT(*) >= 2?

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

- 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.
- Presence of NULL complicates many issues:
  - Special operators needed to check if value is (not) NULL.
  - Is rating>8 true or false for rating=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 do not evaluate to true.)
    - New operators (in particular, outer joins) possible and needed.

Working with NULL

- NULL op constant evaluates to unknown
  - op is one of <, >, =, ≤, ≥, ≠
  - What about NULL = NULL?
- NOT unknown evaluates to unknown
- true OR unknown evaluates to true
  - What about false OR unknown?
- false AND unknown evaluates to false
- Definition of a duplicate: corresponding columns are either equal or both have value NULL
  - Implicitly evaluates (NULL = NULL) as true
- Arithmetic operators (+, -, *, /) return NULL if any input is NULL
- Aggregate operators affected differently
  - COUNT(*) not affected
  - All others, including COUNT(column), discard NULL values before computing the aggregate
    - Compare result of SUM(column) to using + on the same set of values
    - What if all values in the column are NULL?

Working with NULL

- NULL op constant evaluates to unknown
  - op is one of <, >, =, ≤, ≥, ≠
  - What about NULL = NULL? Unknown.
- NOT unknown evaluates to unknown
- true OR unknown evaluates to true
  - What about false OR unknown? Unknown.
- false AND unknown evaluates to false
- Definition of a duplicate: corresponding columns are either equal or both have value NULL
  - Implicitly evaluates (NULL = NULL) as true
- Arithmetic operators (+, -, *, /) return NULL if any input is NULL
- Aggregate operators affected differently
  - COUNT(*) not affected
  - All others, including COUNT(column), discard NULL values before computing the aggregate
    - Compare result of SUM(column) to using + on the same set of values
    - What if all values in the column are NULL? Result is NULL.

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

- First solution: 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.

Number of boats plus number of sailors is < 100

```
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

CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+(SELECT COUNT (B.bid) FROM Boats B) < 100 )
```


Triggers

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
  - Event
    - Change to the database that activates the trigger
  - Condition
    - Query or test that is run when the trigger is activated
  - Action
    - Procedure that is executed when the trigger is activated and its condition is true

Trigger Options

- Event can be insert, delete, or update on DB table
- Condition can be a true/false statement
  - All employee salaries are less than $100K
- Condition can be a query
  - Interpreted as true if and only if answer set is not empty
- Action can perform DB queries and updates that depend on
  - Answers to query in condition part
  - Old and new values of tuples modified by the statement that activated the trigger
  - Action can also contain data-definition commands, e.g., create new tables

Trigger Timing

- Should trigger action be executed before or after the statement that activated the trigger?
  - Consider triggers on insertions
  - Trigger that initializes a variable for counting how many new tuples are inserted: execute trigger before insertion
  - Trigger that updates this count variable for each inserted tuple: execute after each tuple is inserted (might need to examine values of tuple to determine action)
- Challenge: Trigger action can fire other triggers
  - Very difficult to reason about what exactly will happen
    - Trigger can fire “itself” again
  - Unintended effects possible

Trigger Example (Oracle Syntax)

```
CREATE TRIGGER init_count BEFORE INSERT ON Students /* Event */
    DECLARE
        count INTEGER;
    BEGIN /* Action */
        count := 0;
    END

CREATE TRIGGER incr_count AFTER INSERT ON Students /* Event */
    WHEN (new.age < 18) /* Condition, where new refers to inserted tuple */
    FOR EACH ROW
    BEGIN /* Action */
        count = count + 1;
    END
```


Trigger Example (SQL:1999)

```
CREATE TRIGGER set_count AFTER INSERT ON Students
REFERENCING NEW TABLE AS InsertedTuples /* Name for the
   set of newly inserted tuples */
FOR EACH STATEMENT /* Statement-level trigger */
   INSERT
        INTO StatisticsTable(ModifiedTable, ModificationType,
   Count)
        SELECT ‘Students’, ‘Insert’, COUNT(*)
        FROM InsertedTuples I
        WHERE I.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 than relational algebra.
- Even queries that can be expressed in relational algebra can often be expressed more naturally in SQL.
- Many alternative ways to write a query
  - Optimizer should find 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