SQL: Queries, Constraints, Triggers

Chapter 5

Overview

- The Data Manipulation Language (DML)
  - This subset of SQL allows users to pose queries and to insert, delete, and modify rows (Chapter 3).
- The Data Definition Language (DDL)
  - This subset of SQL supports the creation, deletion, and modification of definitions for tables and views (Chapter 3).
- Triggers and Advanced Integrity Constraints
  - Triggers are actions executed by the DBMS whenever changes to the database meet conditions specified in the trigger.

Overview (Cont.)

- Transaction Management
  - Various commands allow a user to explicitly control aspects of how a transaction is to be executed (Chapter 16).
- Security
  - SQL provides mechanisms to control users' access to data objects such as tables and views (Chapter 21).
- Advanced Features
  - Chapters 23 ~ 27 (object-oriented features, recursive queries, spatial data management, etc.)

Example Instances

Reserves

sid	bid	day
22	101	10/10/96
58	103	11/12/96

Sailors

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

Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

- 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?

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 $<, >, =, \leq, \geq, \neq$) 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 is 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*.

Find the names of sailors who have reserved boat number 103

```
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 a good style; however, to use range variables always!

Find the sids of sailors who have reserved a red boat

```
SELECT  R.sid
FROM    Boats B, Reserves R
WHERE   B.bid=R.bid AND B.color='red'
```

- Join of Boats and Reserves then a selection on the color of boats.

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


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.

Compute increments for the ratings of persons who have sailed two different boats on the same day

```
SELECT  S.name, S.rating+1 AS rating
FROM  Sailors S, Reserves R1, Reserves R2
WHERE  S.sid = R1.sid AND S.sid = R2.sid
        AND R1.day = R2.day AND R1.bid <> R2.bid
```

Union, Intersect, and Except

- Union, Intersection, Set-difference in Relational Algebra.
- UNION: Can be used to compute the union of any two *union-compatible* sets of tuples. Returns tuples which occur in either input relation (or both).
- INTERSECT: Returns all tuples that occur in both input relations.
- EXCEPT: Returns all tuples that occur in *relation 1* but not in *relation 2*.

Find the names of sailors who've reserved a red or a green boat

```
SELECT  S.name
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.name
FROM  Sailors S, Reserves R1, Boats B1, Reserves R2, Boats B2
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'
```

Find the names of sailors who've reserved a red or a green boat

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

UNION

SELECT  S2.sname
FROM  Sailors S2, Boats B2, Reserves R2
WHERE  S2.sid=R2sid AND R2.bid=B2.bid AND B2.color='green'
```


Find the names of sailors who've reserved a red and a green boat

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

INTERSECT

SELECT  S2.sname
FROM  Sailors S2, Boats B2, Reserves R2
WHERE  S2.sid=R2sid AND R2.bid=B2.bid AND B2.color='green'
```

- Any problem with this query sentence?

Find the sids of all sailors who have reserved red boats but not green boats

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

EXCEPT

SELECT  S2.sid
FROM  Sailors S2, Boats B2, Reserves R2
WHERE  S2.sid=R2.sid AND R2.bid=B2.bid AND B2.color='green'
```


Find the sids of all sailors who have reserved red boats but not green boats (Cont.)

```
SELECT  R.sid
FROM  Boats B, Reserves R
WHERE  R.bid=B.bid AND B.color='red'
EXCEPT
SELECT  R2.sid
FROM  Boats B2, Reserves R2
WHERE R2.bid=B2.bid AND B2.color='green'
```

- This query relies on *referential integrity*; that is there are no reservations for nonexisting sailors.

Duplicate Elimination

- The default for UNION, INTERSECT, and EXCEPT queries is that duplicates are eliminated.
- To retain duplicates, use the following:
- UNION ALL ➔ the number of copies of a row in the result is always $m + n$, where m and n are the numbers of times that the row appears in the two parts of the union.
- INTERSECT ALL ➔ the number of copies of a row: $min(m, n)$
- EXCEPT ALL ➔ the number of copies of a row: $max(0, m - n)$

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)
```

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

Nested Queries (Cont.)

❖ To find sailors who've *not* reserved #103, use NOT IN.

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid NOT IN  (SELECT  R.sid
                      FROM  Reserves R
                      WHERE  R.bid=103)
```

❖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.*

Find the names of sailors who have NOT reserved a red boat

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.sid IN  (SELECT  R.sid
                   FROM    Reserves R
       NOT IN      WHERE   R.bid IN  (SELECT B.bid
                                      FROM    Boats B
                                      WHERE   B.color = 'red')
```

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.sid IN  (SELECT  R.sid
                   FROM    Reserves R
                   WHERE   R.bid IN  (SELECT B.bid
                                      FROM    Boats B
                       NOT IN         WHERE   B.color = 'red')
```

Find the names of sailors who have reserved a boat that is not red, that is, if they have a reservation, it is not for a red boat.


```
SELECT  S.sname
FROM    Sailors S
WHERE   S.sid IN  (SELECT  R.sid
                   FROM    Reserves R
       NOT IN      WHERE   R.bid IN  (SELECT B.bid
                                      FROM    Boats B
                       NOT IN         WHERE   B.color = 'red')
```

Find the names of sailors who have not reserved a boat that is not red, that is, who have reserved only red boats if they have reserved any boat at all.

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
NOT              WHERE  R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN. Nonempty test.

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

correlation

Nested Queries with Correlation (Cont.)

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

- If UNIQUE is used, and * is replaced by *R.bid*, find 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?*)

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 $>, <, =, \geq, \leq, \neq$
- Find sailors whose rating is greater than that of some sailor called Andy:

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

- What if there were no sailors called Andy?

Find sailors whose rating is better than every sailor called Andy

```
SELECT S.name
FROM  Sailors S
WHERE  S.rating > ALL  (SELECT  S2.rating
                        FROM  Sailors S2
                        WHERE S2.sname='Andy')
```

- What if there were no sailor called Andy?
- Find the sailors with the highest rating

```
SELECT  S.sid
FROM    Sailors S
WHERE  S.rating >= ALL  (SELECT  S2.rating
                         FROM  Sailors S2)
```

Rewriting INTERSECT *Queries Using* IN

Find sids of sailors who've reserved both a red ~~and~~ a green boat:
but not

```
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
          NOT IN         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 red boats but not green boats, just replace *S.sid* by *S.sname* in SELECT clause.

Division in SQL

Find the names of 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))
```

For each sailor *S*, we check to see that the set of boats reserved by *S* includes every boat.

Division in SQL (Cont.)

- 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

For each sailor we check that there is no boat that has not been reserved by this sailor.

Aggregate Operators

- Significant extension of relational algebra.

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

single column

Count the number of sailors

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

Count the number of different sailor names.

```
SELECT  COUNT (DISTINCT S.sname)
FROM  Sailors S
```

Aggregate Operators (Cont.)

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

Find the name(s) of the sailor(s) who has the highest rating.

MIN or **MAX**

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

Find the average age of sailors with a rating of 10.

```
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.)

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

If the SELECT clause uses an *aggregate operation*, then it must use only aggregate operations unless the query contains a GROUP BY clause.

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

Motivation for Grouping

- 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 for each rating level

```
SELECT      S.rating, MIN (S.age)
FROM        Sailors S
GROUP BY    S.rating
```

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

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*!
- One answer tuple is generated per qualifying group.

Find age of the youngest sailor with age ≥ 18, for each rating level with at least 2 <u>such</u> sailors

Sailors instance:

<u>sid</u>	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

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


Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 <u>such</u> sailors.

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
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

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

HAVING COUNT (*) > 1 AND EVERY (S.age <=60)

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
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?

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 (*) > 1
```

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 reservationcount
FROM     Boats B, Reserves R
WHERE    R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```

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

For each red boat, find the number of reservations for this boat (Cont.)

```
SELECT    B.bid, COUNT (*) AS reservationcount
FROM      Boats B, Reserves R
WHERE     R.bid=B.bid
GROUP BY  B.bid
HAVING    B.color='red'
```

- Only columns that appear in the GROUP BY clause can appear in the HAVING clause, unless they appear as arguments to an aggregate operator in the HAVING clause.

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  1  <  (SELECT  COUNT (*)
               FROM  Sailors S2
               WHERE  S.rating=S2.rating)
```

- Shows HAVING clause can also contain a subquery.
- Compare this with the non-nested query.

Find the average age of sailors for each rating level that has at least two sailors

```
SELECT      S.rating,  AVG (S.age) AS avgage
FROM        Sailors S
GROUP BY  S.rating
HAVING     COUNT (*) > 1
```

```
SELECT      S.rating,  AVG (S.age) AS avgage
FROM        Sailors S
GROUP BY  S.rating
HAVING     1 < ( SELECT COUNT (*)
                 FROM Sailors S2
                 WHERE S.rating = S2.rating)
```

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

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

- WRONG => Aggregate operations cannot be nested!
- Correct solution:

```
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.)

Null Values (Cont.)

- Arithmetic operators: +, - , *, and / ➔ null if one of their arguments is null.
- COUNT(*) handles null values just like other values; that is, they get counted.
- New operators (in particular, *outer joins*) possible/needed ➔ left outer join, right outer join, and full outer join
- In a *left outer join*, Sailors rows without a matching Reserves row appear exactly once in the result, with the result columns inherited from Reserves assigned *null* values.

https://www.zentut.com/sql-tutorial/sql-outer-join/

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 < 20)
- *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 constraints.
- 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 (sid,bid,day),
    CONSTRAINT noInterlakeRes
    CHECK (`Interlake' <>
        ( SELECT B.bname
        FROM Boats B
        WHERE B.bid=bid)))
```


Constraints Over Multiple Relations

- Wrong!
- If Sailors is empty, this constraint is defined to always hold.
  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),
    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

```
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 (a change to the DB that activates the trigger)
  - Condition (tests whether the trigger should run)
  - Action (what happens if the trigger runs)
- A trigger can be thought of as a daemon that monitors a database.

Triggers: Example

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

CREATE TRIGGER incr_count AFTER INSERT ON Students
    WHEN (new.age < 18)
    FOR EACH ROW
    BEGIN
      count := count + 1;
    END
```

Triggers: Example

```
CREATE TRIGGER youngSailorUpdate
   AFTER INSERT ON Sailors
REFERENCING NEW TABLE AS InsertedTuples
FOR EACH STATEMENT
   INSERT
        INTO YoungSailors(sid, name, age, rating)
        SELECT sid, name, age, rating
        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 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