SQL: Queries, Constraints, Triggers

Chapter 5
### Example Instances

<table>
<thead>
<tr>
<th>R1</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S1</th>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S2</th>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>28</td>
<td>yuppie</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

- 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

- **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 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

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

<table>
<thead>
<tr>
<th>(sid)</th>
<th>name</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
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 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).

- 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**?)

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND (B.color='red' OR B.color='green')

SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
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-compatibility* sets of tuples.
- 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.

```sql
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
FROM Sailors S, Boats B, Reserves R
  AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND B.color='green'
```

Key field!
Nested Queries

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

```sql
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 recomputed 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:

\[
\text{SELECT } S.\text{sid} \\
\text{FROM } \text{Sailors } S, \text{ Boats } B, \text{ Reserves } R \\
\text{WHERE } S.\text{sid}=R.\text{sid} \text{ AND } R.\text{bid}=B.\text{bid} \text{ AND } B.\text{color}=\text{‘red’} \\
\text{AND } S.\text{sid} \text{ IN } (\text{SELECT } S2.\text{sid} \\
\text{FROM } \text{Sailors } S2, \text{ Boats } B2, \text{ Reserves } R2 \\
\text{WHERE } S2.\text{sid}=R2.\text{sid} \text{ AND } R2.\text{bid}=B2.\text{bid} \\
\text{AND } B2.\text{color}=\text{‘green’})
\]

- 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 \( S.\text{sid} \) by \( S.\text{snname} \) in SELECT clause. (What about INTERSECT query?)
Division in SQL

Find sailors who’ve reserved all boats.

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

(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))

(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

- Significant extension of relational algebra.

- \( \text{SELECT COUNT (*) FROM Sailors S} \)
- \( \text{SELECT AVG (S.age) FROM Sailors S WHERE S.rating=10} \)
- \( \text{SELECT COUNT (DISTINCT S.rating) FROM Sailors S WHERE S.sname='Bob'} \)
- \( \text{SELECT S.sname FROM Sailors S WHERE S.rating= (SELECT MAX(S2.rating) FROM Sailors S2)} \)
- \( \text{SELECT AVG (DISTINCT S.age) FROM Sailors S WHERE S.rating=10} \)

- \( \text{COUNT (*)} \)
- \( \text{COUNT ([DISTINCT] A)} \)
- \( \text{SUM ([DISTINCT] A)} \)
- \( \text{AVG ([DISTINCT] A)} \)
- \( \text{MAX (A)} \)
- \( \text{MIN (A)} \)
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.

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

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

```sql
SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
FROM Sailors S2) = S.age
```
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 \( i = 1, 2, \ldots, 10 \):  
    
    ```sql
    SELECT MIN (S.age)  
    FROM Sailors S  
    WHERE S.rating = \( i \)
    ```
Queries With GROUP BY and HAVING

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 age of the youngest sailor with age $\geq 18$, 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 (*) > 1
```

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Sailors instance:**

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors.

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors and with every sailor under 60.

**HAVING COUNT (*) > 1 AND EVERY (S.age $\leq$60)**

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

What is the result of changing EVERY to ANY?
Find age of the youngest sailor with age \( \geq 18 \), for each rating with at least 2 sailors between 18 and 60.

\[
\text{SELECT S.rating, MIN(S.age) AS minage}
\text{FROM Sailors S}
\text{WHERE S.age} \geq 18 \text{ AND S.age} \leq 60
\text{GROUP BY S.rating}
\text{HAVING COUNT(*)} > 1
\]

**Sailors instance:**

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
For each red boat, find the # of reservations for this boat (by an active sailor)

SELECT B.bid, COUNT(*) AS scount
FROM Boats B, Reserves R
WHERE 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  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:**

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

- Correct solution (in SQL/92):

  ```sql
  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)))
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
Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.

- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
Functions and Procedures

- SQL supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language
  - Functions are particularly useful with specialized data types such as images and geometric objects
    - Example: functions to check if polygons overlap, or to compare images for similarity
  - Some database systems support **table-valued functions**, which can return a relation as a result
- SQL also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL
SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

  ```sql
  create function account_count (customer_name varchar(20))
  returns integer
  begin
    declare a_count integer;
    select count (*) into a_count
    from depositor
    where depositor.customer_name = customer_name
    return a_count;
  end
  ```

- Find each customer that has more than one account.

  ```sql
  select customer_name, customer_street, customer_city
  from customer
  where account_count (customer_name) > 1
  ```

- How to do this w/o the function account_count defined?
Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```sql
create function accounts_of (customer_name char(20))
returns table ( account_number char(10),
                 branch_name char(15),
                 balance numeric(12,2))

return table
  (select account_number, branch_name, balance
   from account A
   where exists ( select *
                  from depositor D
                  where D.customer_name = accounts_of.customer_name
                  and D.account_number = A.account_number ))
```
Table Functions (cont’d)

- Usage
  
  `select *`  
  `from table (accounts_of ('Smith'))`

- Why do people want to do this?
The `author_count` function could instead be written as a procedure:

```sql
create procedure account_count_proc (in title varchar(20),
                                     out a_count integer)
begin
    select count(author) into a_count
    from depositor
    where depositor.customer_name = account_count_proc.customer_name
end
```

Procedures can be invoked either from an SQL procedure or from embedded SQL, using the `call` statement.

```sql
declare a_count integer;
call account_count_proc(‘Smith’, a_count);
```

Procedures and functions can be invoked also from dynamic SQL.
**Procedural Constructs**

- **Compound statement: begin ... end,**
  - May contain multiple SQL statements between `begin` and `end`.
  - Local variables can be declared within a compound statements

- **While and repeat statements:**
  
  ```
  declare n integer default 0;
  while n < 10 do
    set n = n + 1
  end while

  repeat
    set n = n - 1
  until n = 0
  end repeat
  ```
For loop

- Permits iteration over all results of a query
- Example: find total of all balances at the Perryridge branch

```sql
declare n integer default 0;
for r as
    select balance from account
    where branch_name = 'Perryridge'
do
    set n = n + r.balance
end for
```
Procedural Constructs (cont.)

- Conditional statements (if-then-else)
  E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)
    
    ```
    if r.balance < 1000
      then set l = l + r.balance
    elsif r.balance < 5000
      then set m = m + r.balance
    else set h = h + r.balance
    end if
    ```

- Signaling of exception conditions, and declaring handlers for exceptions
  ```
  declare out_of_stock condition
  declare exit handler for out_of_stock
  begin
    ...
    signal out-of-stock
  end
  ```
  - The handler here is `exit` -- causes enclosing `begin..end` to be exited
  - Other actions possible on exception
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
- Extension of SQL supports procedural programming