Database Systems

SQL

Feifei Li
University of Utah
The SQL Query Language

- SQL stands for Structured Query Language

- The most widely used relational query language. Current standard is SQL-2011
  - MySQL/PostgreSQL has some “unique” aspects
    - as do most systems.
DDL – Create Table

- CREATE TABLE table_name ( { column_name data_type [ DEFAULT default_expr ] [ column_constraint [, ... ] ] | table_constraint } [, ... ] )

- Data Types include:
  - character(n) – fixed-length character string
  - character varying(n) – variable-length character string
  - smallint, integer, bigint, numeric, real, double precision
  - date, time, timestamp, ...
  - serial - unique ID for indexing and cross reference
  ...

Create Table (w/ column constraints)

- CREATE TABLE table_name ( { column_name data_type [ DEFAULT default_expr ] [ column_constraint [, ... ] ] | table_constraint } [, ... ] )

Column Constraints:

- [ CONSTRAINT constraint_name ] { NOT NULL | NULL | UNIQUE | PRIMARY KEY | CHECK ( expression ) | REFERENCES reftable [ ( refcolumn ) ] [ ON DELETE action ] [ ON UPDATE action ] }

  action is one of:
  - NO ACTION, CASCADE, SET NULL, SET DEFAULT

expression for column constraint must produce a boolean result and reference the related column’s value only.
Create Table (w/table constraints)

- CREATE TABLE `table_name` ( { `column_name` `data_type` [ DEFAULT `default_expr` ] [ `column_constraint` [, ... ] ] | `table_constraint` } [, ... ] )

Table Constraints:
- [ CONSTRAINT `constraint_name` ]
  { UNIQUE ( `column_name` [, ... ] ) | PRIMARY KEY ( `column_name` [, ... ] ) | CHECK ( `expression` ) | FOREIGN KEY ( `column_name` [, ... ] ) REFERENCES `reftable` [ ( `refcolumn` [, ... ] ) ] [ ON DELETE `action` ] [ ON UPDATE `action` ] }

Here, expressions, keys, etc can include multiple columns
CREATE TABLE films (  
code CHAR(5) PRIMARY KEY,  
title VARCHAR(40),  
did DECIMAL(3),  
date_prod DATE,  
kind VARCHAR(10),  
CONSTRAINT production UNIQUE(date_prod)  
FOREIGN KEY did REFERENCES distributors ON DELETE NO ACTION );

CREATE TABLE distributors (  
did DECIMAL(3) PRIMARY KEY,  
name VARCHAR(40)  
CONSTRAINT con1 CHECK (did > 100 AND name <> ‘ ’ ) );
The SQL DML

- Single-table queries are straightforward.

- To find all 18 year old students, we can write:

```
SELECT *
FROM Students S
WHERE S.age = 18
```

• To find just names and logins, replace the first line:

```
SELECT S.name, S.login
```
Querying Multiple Relations

- Can specify a join over two tables as follows:
  
  ```sql
  SELECT S.name, E.cid
  FROM Students S, Enrolled E
  WHERE S.sid=E.sid AND E.grade='B'
  ```

```
<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
</tbody>
</table>
```

Note: obviously no referential integrity constraints have been used here.

result =

<table>
<thead>
<tr>
<th>S.name</th>
<th>E.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>History105</td>
</tr>
</tbody>
</table>
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 tables in *relation-list*
- **qualification**: Comparisons combined using AND, OR and NOT.
  - Comparisons are Attr *op* const or Attr1 *op* Attr2, where *op* is one of 
    $<$, $>$, $=$, $\leq$, $\geq$, $\neq$
- **DISTINCT**: optional keyword indicating that the answer should not contain duplicates.
  - In SQL SELECT, the default is that duplicates are *not* eliminated! (Result is called a “multiset”)

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

Semantics of an SQL query are defined in terms of the following conceptual evaluation strategy:

1. do FROM clause: compute cross-product of tables (e.g., Students and Enrolled).
2. do WHERE clause: Check conditions, discard tuples that fail. (called “selection”).
3. do SELECT clause: Delete unwanted fields. (called “projection”).
4. If DISTINCT specified, eliminate duplicate rows.

Probably the least efficient way to compute a query!
- An optimizer will find more efficient strategies to get the same answer.
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
Step 2 - Discard tuples that fail predicate

<table>
<thead>
<tr>
<th>S.sid</th>
<th>S.name</th>
<th>S.login</th>
<th>S.age</th>
<th>S.gpa</th>
<th>E.sid</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53832</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
## Step 3 - Discard Unwanted Columns

- **SELECT** S.name, E.cid
- **FROM** Students S, Enrolled E
- **WHERE** S.sid=E.sid AND E.grade='B'

### Table

<table>
<thead>
<tr>
<th>S.sid</th>
<th>S.name</th>
<th>S.login</th>
<th>S.age</th>
<th>S.gpa</th>
<th>E.sid</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53832</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>
Now the Details

We will use these instances of relations in our examples.

**Question:**
If the key for the Reserves relation contained only the attributes \textit{sid} and \textit{bid}, how would the semantics differ?
Example Schemas

CREATE TABLE Sailors (sid INTEGER PRIMARY KEY, sname CHAR(20), rating INTEGER, age REAL)

CREATE TABLE Boats (bid INTEGER PRIMARY KEY, bname CHAR(20), color CHAR(10))

CREATE TABLE Reserves (sid INTEGER REFERENCES Sailors, bid INTEGER, day DATE, PRIMARY KEY (sid, bid, day), FOREIGN KEY (bid) REFERENCES Boats)
Another Join Query

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

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</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>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Some Notes on Range Variables

- Can associate “range variables” with the tables in the FROM clause.
  - saves writing, makes queries easier to understand
- Needed when ambiguity could arise.
  - for example, if same table used multiple times in same FROM (called a “self-join”)

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

Can be rewritten using range variables as:

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

- Here's an example where range variables are required (self-join example):

```
SELECT x.sname, x.age, y.sname, y.age  
FROM Sailors x, Sailors y  
WHERE x.age > y.age
```

- Note that target list can be replaced by "*" if you don’t want to do a projection:

```
SELECT  *  
FROM Sailors x  
WHERE  x.age > 20
```
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 (DISTINCT forces the system to remove duplicates from the output)?
- 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

- Can use arithmetic expressions in SELECT clause (plus other operations we’ll discuss later)
- Use AS to provide column names (like a renaming operator)

```
SELECT S.age, S.age-5 AS age1, 2*S.age AS age2
    FROM Sailors S
WHERE S.sname = 'Dustin'
```

- Can also have expressions in WHERE clause:

```
SELECT S1.sname AS name1, S2.sname AS name2
    FROM Sailors S1, Sailors S2
WHERE 2*S1.rating = S2.rating - 1
```
String operations

SQL supports some basic string operations: “LIKE” is used for string matching

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

`_` 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).

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

```sql
SELECT R.sid 
  FROM Boats B, Reserves R 
WHERE R.bid = B.bid AND B.color = 'red' 
UNION
SELECT R.sid 
  FROM Boats B, Reserves R 
WHERE R.bid = B.bid AND B.color = 'green'
```
Find sid’s of sailors who’ve reserved a red and a green boat

- If we simply replace OR by AND in the previous query, we get the wrong answer. (Why?)
- Instead, could use a self-join:

```
SELECT R1.sid
FROM Boats B1, Reserves R1,
     Boats B2, Reserves R2
WHERE R1.sid=R2.sid
    AND R1.bid=B1.bid
    AND R2.bid=B2.bid
    AND (B1.color='red' AND B2.color='green')
```
Find sid’s of sailors who’ve reserved a red and a green boat

- Or you can use AS to “rename” the output of a SQL block:

```
SELECT R1.sid
    FROM Boats B1, Reserves R1,
        (SELECT R2.sid
            FROM Boats B2, Reserves R2
            WHERE B2.color = 'green'
                AND B2.bid=R2.bid) AS GR
    WHERE R1.sid=GR.sid
        AND R1.bid=B1.bid
        AND B1.color='red'
```

```
SELECT RR.sid
    FROM (SELECT R1.sid
            FROM Boats B1, Reserves R1,
                WHERE B1.color='red'
                    AND B1.bid=R1.bid) AS RR,
        (SELECT R2.sid
            FROM Boats B2, Reserves R2
            WHERE B2.color = 'green'
                AND B2.bid=R2.bid) AS GR
    WHERE RR.sid=GR.sid
```
AND Continued...

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

- **EXCEPT** (sometimes called MINUS)

- **many** systems don’t support them.

```
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'
```
Find sid’s of sailors who’ve reserved a red **but did not reserve** 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’
EXCEPT
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

- Powerful feature of SQL: WHERE clause can itself contain an SQL query!
  - Actually, so can FROM and HAVING clauses.

  **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)
  ```

- 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:

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

- **EXISTS** is another set comparison operator, like **IN**.
- Can also specify **NOT EXISTS**
- 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 in a subquery;
- Subquery must be recomputed for each Sailors tuple.
  - Think of subquery as a function call that runs a query!
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: \( \text{op ANY, op ALL} \)
- Find sailors whose rating is greater than that of some sailor called Horatio:

\[
\text{SELECT *}
\text{FROM Sailors S}
\text{WHERE S.rating > ANY (SELECT S2.rating FROM Sailors S2}
\text{WHERE S2.sname='Horatio')}
\]
Semantics of nested operators

- v is a value, A is a multi-set
- v IN A evaluates to true iff v ∈ A. v NOT IN A is the opposite.
- EXISTS A evaluates to true iff A ≠ Ø. NOT EXISTS A is the opposite.
- UNIQUE A evaluates to true iff A is a set. NOT UNIQUE A is the opposite.
- v OP ANY A evaluates to true iff ∃x ∈ A, such that v OP x evaluates to true.
- v OP ALL A evaluates to true iff ∀x ∈ A, v OP x always evaluates to true.
Rewriting INTERSECT Queries Using IN

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

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

- Similarly, EXCEPT queries re-written using NOT IN.
- How would you change this to find names (not sid’s) of Sailors who’ve reserved both red and green boats?
Division in SQL

Find sailors who’ve reserved all boats.

```
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
Basic SQL Queries - Summary

- An advantage of the relational model is its well-defined query semantics.
- SQL provides functionality close to that of the basic relational model.
  - some differences in duplicate handling, null values, set operators, etc.
- Typically, many ways to write a query
  - the system is responsible for figuring a fast way to actually execute a query regardless of how it is written.
- Lots more functionality beyond these basic features
Aggregate Operators

- Significant extension of relational algebra.

```
SELECT COUNT (*)
FROM  Sailors S

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

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

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

SELECT AVG (S.age)
FROM  Sailors S
WHERE S.rating=10
```
single column
Find name and age of the oldest sailor(s)

- The first query is incorrect!

- Third query equivalent to second query.

```sql
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 S.age >= ALL (SELECT S2.age
                      FROM Sailors S2)
```
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

- To generate values for a column based on groups of rows, use aggregate functions in SELECT statements with the GROUP BY clause

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

The `target-list` contains

(i) list of column names &
(ii) terms with aggregate operations (e.g., MIN (S.age)).

- column name list (i) can contain only attributes from the `grouping-list`, since
  the output for each group must represent a consistent value from that group.
Group By Examples

For each rating, find the average age of the sailors

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

For each rating find the age of the youngest sailor with age $\geq$ 18

SELECT  S.rating,  MIN (S.age)
FROM  Sailors S
WHERE  S.age $\geq$ 18
GROUP BY  S.rating
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`.

- One answer tuple is generated per qualifying group.
An illustration

SELECT S.rating, MIN (S.age) 
FROM  Sailors S 
WHERE  S.age >= 18 
GROUP BY  S.rating

<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>31</td>
<td>lubber</td>
<td>8</td>
<td>55.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Answer Table

3. Perform Aggregation

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</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>55.0</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

1. Form cross product

2. Delete unneeded columns, rows; form groups
Find the number of reservations for each red boat.

```
SELECT B.bid, COUNT(*) AS numres
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.
SELECT B.bid, COUNT(*) AS scount
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
GROUP BY B.bid
Use the HAVING clause with the GROUP BY clause to restrict which group-rows are returned in the result set.
Conceptual Evaluation

- Form groups as before.

- The *group-qualification* is then applied to eliminate some groups.
  - Expressions in *group-qualification* must have a *single value per group*!
  - That is, attributes in *group-qualification* must be arguments of an aggregate op or must also appear in the *grouping-list*.

- One answer tuple is generated per qualifying group.
Find the age of the youngest sailor with age \( \geq 18 \), for each rating with at least 2 such sailors

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

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
<th>rating</th>
<th>m-age</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
<td>1</td>
<td>33.0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
<td>7</td>
<td>35.0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
<td>8</td>
<td>55.0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
<td>10</td>
<td>35.0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Answer relation

<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>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>m-age</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>35.0</td>
<td>1</td>
</tr>
</tbody>
</table>
Find sailors who’ve reserved all boats.

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
Find sailors who’ve reserved all boats.

Can you do this using Group By and Having?

```
SELECT  S.name
FROM    Sailors S, reserves R
WHERE   S.sid = R.sid
GROUP BY S.name, S.sid
HAVING  COUNT(DISTINCT R.bid) =
         ( SELECT COUNT(*) FROM Boats)
```

Note: must have both sid and name in the GROUP BY clause. Why?
**An illustration**

```
SELECT S.name, S.sid
FROM Sailors S, reserves R
WHERE S.sid = r.sid
GROUP BY S.name, S.sid
HAVING COUNT(DISTINCT R.bid) =
Select COUNT (*) FROM Boats
```

<table>
<thead>
<tr>
<th>s.name</th>
<th>s.sid</th>
<th>r.sid</th>
<th>r.bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dustin</td>
<td>22</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>Lubber</td>
<td>31</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>Bob</td>
<td>95</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>Dustin</td>
<td>22</td>
<td>95</td>
<td>102</td>
</tr>
<tr>
<td>Lubber</td>
<td>31</td>
<td>95</td>
<td>102</td>
</tr>
<tr>
<td>Bob</td>
<td>95</td>
<td>95</td>
<td>102</td>
</tr>
</tbody>
</table>

**Count (*) from boats = 4**

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

Apply having clause to groups
Find the names of the sailors who’ve reserved most number of boats for each rating group

```
SELECT  S.name
FROM    Sailors S, reserves R
WHERE   S.sid = R.sid
GROUP BY S.name, S.sid
HAVING  COUNT(R.bid) =
    ( Select MAX(C) FROM
        (SELECT  S1.sid, COUNT(*) AS C FROM
            Sailors S1, reserves R1
            WHERE  S1.sid = R1.sid AND S1.rating = S.rating
            GROUP BY S1.sid) )
```
Find the names of the sailors who’ve reserved most number of boats for each rating group

```
SELECT  S.name
FROM    Sailors S, reserves R
WHERE   S.sid = R.sid
GROUP BY S.name, S.sid
HAVING  COUNT(R.bid) >= ALL
        (SELECT  COUNT(*) FROM
         Sailors S1, reserves R1
         WHERE  S1.sid = R1.sid AND S1.rating = S.rating
         GROUP BY S1.sid)
```
You can also do a “bulk insert” of values from one table into another:

\[
\text{INSERT INTO TEMP(bid)}
\]
\[
\text{SELECT r.bid FROM Reserves R WHERE r.sid = 22;}
\]

(must be type compatible)
DELETE & UPDATE

DELETE FROM Boats WHERE color = ‘red’

DELETE FROM Boats b
WHERE b. bid =
    (SELECT r.bid FROM Reserves R WHERE r.sid = 22)

Can also modify tuples using UPDATE statement.

UPDATE Boats
SET Color = “green”
WHERE bid = 103;
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.
Joins

SELECT (column_list)
FROM  table_name
[INNER | {LEFT | RIGHT | FULL } OUTER] JOIN
  table_name
 ON qualification_list
WHERE ...

Explicit join semantics needed unless it is an INNER join (INNER is default)
Inner Join

Only the rows that match the search conditions are returned.

```
SELECT s.sid, s.name, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
```

Returns only those sailors who have reserved boats

```
SELECT s.sid, s.name, r.bid
FROM Sailors s NATURAL JOIN Reserves r
```

“NATURAL” means equi-join for each pair of attributes with the same name
An illustration

```
SELECT s.sid, s.name, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</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>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s.sid</th>
<th>s.name</th>
<th>r.bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>101</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>103</td>
</tr>
</tbody>
</table>
Left Outer Join

Left Outer Join returns all matched rows, plus all unmatched rows from the table on the left of the join clause (use nulls in fields of non-matching tuples)

SELECT s.sid, s.name, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid

Returns all sailors & information on whether they have reserved boats
An illustration

SELECT s.sid, s.name, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid

<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>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s.sid</th>
<th>s.name</th>
<th>r.bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>101</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>103</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td></td>
</tr>
</tbody>
</table>
Right Outer Join

Right Outer Join returns all matched rows, plus all unmatched rows from the table on the right of the join clause

SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid

Returns all boats & information on which ones are reserved.
An illustration

```sql
SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid
```

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r.sid</th>
<th>b.bid</th>
<th>b.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>Interlake</td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>Interlake</td>
</tr>
<tr>
<td>95</td>
<td>103</td>
<td>Clipper</td>
</tr>
<tr>
<td>104</td>
<td></td>
<td>Marine</td>
</tr>
</tbody>
</table>
Full Outer Join

Full Outer Join returns all (matched or unmatched) rows from the tables on both sides of the join clause

```sql
SELECT r.sid, b.bid, b.name
FROM Reserves r FULL OUTER JOIN Boats b
ON r.bid = b.bid

Returns all boats & all information on reservations
SELECT r.sid, b.bid, b.name
FROM Reserves r FULL OUTER JOIN Boats b
ON r.bid = b.bid

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

Note: in this case it is the same as the ROJ because bid is a foreign key in reserves, so all reservations must have a corresponding tuple in boats.
CREATE VIEW view_name
AS select_statement

Makes development simpler
Often used for security
Not instantiated - makes updates tricky

CREATE VIEW Reds
AS SELECT B.bid, COUNT(*) AS scount
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
GROUP BY B.bid
CREATE VIEW Reds
AS SELECT B.bid, COUNT (*) AS scount
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
GROUP BY B.bid

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b.bid</th>
<th>scount</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>1</td>
</tr>
</tbody>
</table>

Reds
Views Instead of Relations in Queries

CREATE VIEW Reds
AS SELECT B.bid, COUNT (*) AS scount
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
GROUP BY B.bid

<table>
<thead>
<tr>
<th>bid</th>
<th>scount</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>1</td>
</tr>
</tbody>
</table>

SELECT bname, scount
FROM Reds R, Boats B
WHERE R.bid=B.bid
AND scount < 10
Sorting the Results of a Query

- ORDER BY column \[ ASC | DESC \] [, ...]

  ```sql
  SELECT S.rating, S.sname, S.age
  FROM sailed S, Boats B, Reserves R
  WHERE S.sid=R.sid
       AND R.bid=B.bid AND B.color='red'
  ORDER BY S.rating, S.sname;
  ```

- Can order by any column in SELECT list, including expressions or aggs, and select top-k:

  ```sql
  SELECT S.sid, COUNT(*) AS redrescnt
  FROM sailed S, Boats B, Reserves R
  WHERE S.sid=R.sid
       AND R.bid=B.bid AND B.color='red'
  GROUP BY S.sid
  ORDER BY redrescnt DESC
  LIMIT 10;
  ```
Discretionary Access Control

```
GRANT privileges ON object TO users [WITH GRANT OPTION]
```

- Object can be a **Table** or a **View**
- Privileges can be:
  - Select
  - Insert
  - Delete
  - References (cols) – allow to create a foreign key that references the specified column(s)
  - All
- Can later be **REVOKED**
- Users can be single users or groups
- See Chapter 17 for more details.
Two more important topics

- Constraints (such as triggers)
- SQL embedded in other languages
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.
- *Primary key and foreign key constraints*: you know them.
General Constraints

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

```sql
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

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 )

- Awkward and wrong!
- Only checks sailors!
- Only required to hold if the associated table is non-empty.

- ASSERTION is the right solution; not associated with either table.
- Unfortunately, not supported in some DBMS.
- Triggers are another solution.
Constraints Over Multiple Relations, Another Example

- No Sailors should be allowed to reserve more than 5 different Blue color boats.

```sql
CREATE ASSERTION noBlueMonopoly
CHECK Not Exists (Select sid
    From Reserve R, Boat B
    Where R.bid=B.bid and B.color = 'Red'
    Group By R.sid
    Having Count(distinct R.bid)>5)
```
Triggers (Active database)

- **Trigger**: A procedure that starts automatically if specified changes occur to the DBMS
- Analog to a "daemon" that monitors a database for certain events to occur
- Three parts:
  - **Event** (activates the trigger)
  - **Condition** (tests whether the triggers should run) [Optional]
  - **Action** (what happens if the trigger runs)

- **Semantics**:
  - When event occurs, and condition is satisfied, the action is performed.
An example of Trigger

CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW
WHEN (new.salary < 100,000)
BEGIN
    RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Professor Salary');
END;

- Conditions can refer to old/new values of tuples modified by the statement activating the trigger.
Triggers – Event, Condition, Action

- Events could be:
  
  BEFORE | AFTER INSERT | UPDATE | DELETE ON <tableName>

  e.g.: BEFORE INSERT ON Professor

- Condition is SQL expression or even an SQL query (query with non-empty result means TRUE)

- Action can be many different choices:
  
  - SQL statements, body of PSM (persistent stored modules), and even DDL and transaction-oriented statements like “commit”.
Example Trigger

Assume our DB has a relation schema:

Professor (pNum, pName, salary)

We want to write a trigger that:

Ensures that any new professor inserted has salary \( \geq 60000 \)
Example Trigger

CREATE TRIGGER minSalary BEFORE INSERT ON Professor

for what context  ?

BEGIN

check for violation here  ?

END;
Example Trigger

CREATE TRIGGER minSalary BEFORE INSERT ON Professor

FOR EACH ROW

BEGIN

Violation of Minimum Professor Salary?

END;
Example Trigger

CREATE TRIGGER minSalary BEFORE INSERT ON Professor

    FOR EACH ROW

BEGIN

    IF (:new.salary < 60000)
        THEN RAISE_APPLICATION_ERROR (-20004,
            'Violation of Minimum Professor Salary');
    END IF;

END;

CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW

DECLARE temp int; -- dummy variable not needed

BEGIN
    IF (:new.salary < 60000)
       THEN RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Professor Salary');
    END IF;

    temp := 10; -- to illustrate declared variables

END;
.
Details of Trigger Example

- BEFORE INSERT ON Professor
  - This trigger is checked before the tuple is inserted
- FOR EACH ROW
  - specifies that trigger is performed for each row inserted
- :new
  - refers to the new tuple inserted
- If (:new.salary < 60000)
  - then an application error is raised and hence the row is not inserted; otherwise the row is inserted.
- Use error code: -20004;
  - this is in the valid range
Example Trigger Using Condition

CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW
WHEN (new.salary < 60000)
BEGIN
    RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Professor Salary');
END;

- Conditions can refer to old/new values of tuples modified by the statement activating the trigger.
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

REFERENCING NEW as newTuple

FOR EACH ROW

WHEN (newTuple.salary < 60000)

BEGIN
   RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Professor Salary');
END;
Example Trigger

CREATE TRIGGER minSalary
    BEFORE UPDATE ON Professor
REFERENCING OLD AS oldTuple NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < oldTuple.salary)
BEGIN
    RAISE_APPLICATION_ERROR (-20004, ‘Salary Decreasing !!’);
END;

- Ensure that salary does not decrease
Another Trigger Example

CREATE TRIGGER youngSailorUpdate
   AFTER INSERT ON SAILORS
   REFERENCING NEW TABLE AS NewSailors
   FOR EACH STATEMENT
   INSERT
      INTO YoungSailors(sid, name, age, rating)
      SELECT sid, name, age, rating
      FROM NewSailors N
      WHERE N.age <= 18
**Row vs Statement Level Trigger**

- **Row** level: activated once per modified tuple
- **Statement** level: activate once per SQL statement

- **Row** level triggers can access new data, statement level triggers cannot always do that (depends on DBMS).

- **Statement** level triggers will be more efficient if we do not need to make row-specific decisions.
Example: Consider a relation schema

Account (num, amount)

where we will allow creation of new accounts only during normal business hours.
Example: Statement level trigger

CREATE TRIGGER MYTRIG1
BEFORE INSERT ON Account
FOR EACH STATEMENT --- is default
BEGIN
  IF (TO_CHAR(SYSDATE,'dy') IN ('sat','sun'))
  OR
    (TO_CHAR(SYSDATE,'hh24:mi') NOT BETWEEN '08:00' AND '17:00')
  THEN
    RAISE_APPLICATION_ERROR(-20500,'Cannot create new account now !!');
  END IF;
END;

When to use BEFORE/AFTER

- Based on efficiency considerations or semantics.

- Suppose we perform statement-level *after insert*, then all the rows are inserted first, then if the condition fails, and all the inserted rows must be “rolled back”

- Not very efficient !!
Combining multiple events into one trigger

CREATE TRIGGER salaryRestrictions
AFTER INSERT OR UPDATE ON Professor
FOR EACH ROW
BEGIN
IF (INSERTING AND :new.salary < 60000) THEN
    RAISE_APPLICATION_ERROR (-20004, 'below min salary'); END IF;
IF (UPDATING AND :new.salary < :old.salary) THEN
    RAISE_APPLICATION_ERROR (-20004, 'Salary Decreasing !!'); END IF;
END;
Summary: Trigger Syntax

CREATE TRIGGER <triggerName>
BEFORE|AFTER INSERT|DELETE|UPDATE
[OF <columnList>] ON <tableName>|<viewName>
[REFERENCING [OLD AS <oldName>] [NEW AS <newName>]]
[FOR EACH ROW] (default is “FOR EACH STATEMENT”)
[WHEN (<condition>)]
<PSM body>;
Some Points about Triggers

- Check the system tables:
  - user_triggers
  - user_trigger_cols
  - user_errors

- ORA-04091: mutating relation problem
  - In a row level trigger, you cannot have the body refer to the table specified in the event

- Also INSTEAD OF triggers can be specified on views
To Show Compilation Errors

SELECT line, position, text
FROM user_errors
WHERE name = 'MY_TRIGGER'
    AND TYPE = 'TRIGGER'

In SQL*Plus, you can also use the following shortcut:

SQL> SHOW ERRORS TRIGGER MY_TRIGGER
Constraints versus Triggers

- **Constraints** are useful for database consistency
  - Use IC when sufficient
  - More opportunity for optimization
  - Not restricted into insert/delete/update

- **Triggers** are flexible and powerful
  - Alerters
  - Event logging for auditing
  - Security enforcement
  - Analysis of table accesses (statistics)
  - Workflow and business intelligence ...

- But can be hard to understand ......
  - Several triggers (Arbitrary order → unpredictable !?)
  - Chain triggers (When to stop ?)
  - Recursive triggers (Termination?)
Writing Applications with SQL

- SQL is not a general purpose programming language.
  + Tailored for data retrieval and manipulation
  + Relatively easy to optimize and parallelize
  - Can’t write entire apps in SQL alone

Options:

Make the query language “turing complete”
  Avoids the “impedance mismatch”, but, loses advantages of relational lang simplicity
Allow SQL to be embedded in regular programming languages.

Q: What needs to be solved to make the latter approach work?
DBMS vendors usually provide “host language bindings”
   - E.g. for C or COBOL
   - Allow SQL statements to be called from within a program
   - Typically you preprocess your programs
   - Preprocessor generates calls to a proprietary DB connectivity library

General pattern
   - One call to connect to the right database (login, etc.)
   - SQL statements can refer to host variables from the language

Typically vendor-specific
   - We won’t look at any in detail, we’ll look at standard stuff

Problem
   - SQL relations are (multi-)sets, no a priori bound on the number of records. No such data structure in C.
   - SQL supports a mechanism called a cursor to handle this.
Just to give you a flavor

EXEC SQL SELECT S.sname, S.age
    INTO :c_sname,:c_age
FROM Sailors S
WHERE S.sid = :c_sid
Cursors

- Can declare a cursor on a relation or query
- Can *open* a cursor
- Can repeatedly *fetch* a tuple (moving the cursor)
- Special return value when all tuples have been retrieved.
- **ORDER BY** allows control over the order in which tuples are returned.
  - Fields in ORDER BY clause must also appear in SELECT clause.
- Can also modify/delete tuple pointed to by a cursor
  - A “non-relational” way to get a handle to a particular tuple
- There’s an Embedded SQL syntax for cursors
  - DECLARE `<cursorname>` CURSOR FOR `<select stmt>`
  - FETCH FROM `<cursorname>` INTO `<variable names>`
  - But we’ll use JDBC instead
Database APIs: alternative to embedding

- Rather than modify compiler, add a library with database calls (API)
  - special procedures/objects
  - passes SQL strings from language, presents result sets in a language-friendly way
  - **ODBC** a C/C++ standard started on Windows
  - **JDBC** a Java equivalent
  - Most scripting languages have similar things
    - E.g. For Perl there is DBI, “oraPerl”, other packages

- Mostly DBMS-neutral
  - at least try to hide distinctions across different DBMSs
A lookup service maps “data source names” (“DSNs”) to drivers
- Typically handled by OS
- Based on the DSN used, a “driver” is linked into the app at runtime
- The driver traps calls, translates them into DBMS-specific code
- Database can be across a network
- ODBC is standard, so the same program can be used (in theory) to access multiple database systems
- Data source may not even be an SQL database!
ODBC/JDBC

- Various vendors provide drivers
  - MS bundles a bunch into Windows

- Drivers for various data sources
  - Relational DBMSs (Oracle, DB2, SQL Server, etc.)
  - “Desktop” DBMSs (Access, etc.)
  - Spreadsheets (MS Excel, etc.)
  - Delimited text files (.CSV, .TXT, etc.)

- You can use JDBC/ODBC clients over many data sources
  - E.g. MS Query comes with many versions of MS office
  - Can write your own Java or C++ programs against xDBC
JDBC

- Part of Java, very easy to use
- Java comes with a JDBC-to-ODBC bridge
  - So JDBC code can talk to any ODBC data source
  - E.g. look in your Windows Control Panel for ODBC drivers!
- JDBC tutorial online
  - https://docs.oracle.com/javase/tutorial/jdbc/
JDBC Basics: Connections

- A Connection is an object representing a login to a database
  
  // GET CONNECTION
  Connection con;
  try {
      con = DriverManager.getConnection(
        "jdbc:odbc:sailorsDB",
        userName,password);
  } catch(Exception e){ System.out.println(e); }

- Eventually you close the connection
  
  // CLOSE CONNECTION
  try { con.close(); } catch (Exception e) { System.out.println(e); }
JDBC Basics: Statements

- You need a Statement object for each SQL statement
  
  ```java
  // CREATE STATEMENT
  Statement stmt;
  try {
      stmt = con.createStatement();
  } catch (Exception e){
      System.out.println(e);
  }
  ``

  Soon we’ll say stmt.executeQuery("select ...");
CreateStatement cursor behavior

- Two optional args to createStatement:
  - `createStatement(ResultSet.<TYPE>, ResultSet.<CONCUR>)`
  - Corresponds to SQL cursor features
- `<TYPE>` is one of
  - `TYPE_FORWARD_ONLY`: can’t move cursor backward
  - `TYPE_SCROLL_INSENSITIVE`: can move backward, but doesn’t show results of any updates
  - `TYPE_SCROLL_SENSITIVE`: can move backward, will show updates from this statement
- `<CONCUR>` is one of
  - `CONCUR_READ_ONLY`: this statement doesn’t allow updates
  - `CONCUR_UPDATABLE`: this statement allows updates
- Defaults:
  - `TYPE_FORWARD_ONLY` and `CONCUR_READ_ONLY`
A ResultSet object serves as a cursor for the statement’s results (stmt.executeQuery())

```
// EXECUTE QUERY
ResultSet results;
try {
    results = stmt.executeQuery("select * from Sailors")
} catch (Exception e){
    System.out.println(e);  }
```

Obvious handy methods:
- `results.next()` advances cursor to next tuple
  - Returns “false” when the cursor slides off the table (beginning or end)
- “scrollable” cursors:
  - `results.previous()`, `results.relative(int)`, `results.absolute(int)`, `results.first()`, `results.last()`, `results.beforeFirst()`, `results.afterLast()`
Can find out stuff about the ResultSet schema via `ResultSetMetaData`:

```java
ResultSetMetaData rsmd = results.getMetaData();
int numCols = rsmd.getColumnCount();
int i, rowcount = 0;

// get column header info
for (i=1; i <= numCols; i++){
    if (i > 1) buf.append(",");
    buf.append(rsmd.getColumnLabel(i));
}
buf.append("\n");
```

Other `ResultSetMetaData` methods:
- `getColumnType(i)`, `isNull(i)`, etc.
Getting Values in Current of Cursor

- **getString**
  ```java
  // break it off at 100 rows max
  while (results.next() && rowcount < 100){
    // Loop through each column, getting the
    // column data and displaying
    
    for (i=1; i <= numCols; i++) {
      if (i > 1) buf.append(",");
      buf.append(results.getString(i));
    }
    buf.append("\n");
    rowcount++;
  }
  ```

- Similarly, `getFloat`, `getInt`, etc.
Update fields in current of cursor:

```java
result.next();
result.updateInt("Rating", 10);
```

- Also updateString, updateFloat, etc.
- Or can always submit a full SQL UPDATE statement
  - Via executeQuery()

- The original statement must have been CONCUR_UPDATABLE in either case!
try {
    // CLOSE RESULT SET
    results.close();
    // CLOSE STATEMENT
    stmt.close();
    // CLOSE CONNECTION
    con.close();
} catch (Exception e) {
    System.out.println(e);
}
Putting it Together (w/o try/catch)

```
Connection con =
    DriverManager.getConnection("jdbc:odbc:weblog", userName, password);

Statement stmt = con.createStatement();

ResultSet results =
    stmt.executeQuery("select * from Sailors");

ResultSetMetaData rsmd = results.getMetaData();

int numCols = rsmd.getColumnCount(), i;

StringBuffer buf = new StringBuffer();

while (results.next() && rowcount < 100){
    for (i=1; i <= numCols; i++) {
        if (i > 1) buf.append(", ");
        buf.append(results.getString(i));
    }
    buf.append("\n");
}

results.close(); stmt.close(); con.close();
```
Similar deal for web scripting langs

- Common scenario today is to have a web client
  - A web form issues a query to the DB
  - Results formatted as HTML
- Many web scripting languages used
  - jsp, asp, PHP, etc.
  - we’ll use JSP in our class
  - most of these are similar, look a lot like jdbc with HTML mixed in
<?php  $conn = pg_pconnect("dbname=cowbook user=jmh\ 
password=secret");
    if (!$conn) {
        echo "An error occurred.\n";
        exit;
    }
$result = pg_query ($conn, "SELECT * FROM Sailors");
if (!$result) {
    echo "An error occurred.\n";  exit;
}
$num = pg_num_rows($result);
for ($i=0; $i < $num; $i++) {
    $r = pg_fetch_row($result, $i);
    for ($j=0; $j < count($r); $j++) {
        echo "$r[$j]&nbsp;";
    }
    echo "<BR>";
} ?>
API Summary

APIs are needed to interface DBMSs to programming languages

- Embedded SQL uses “native drivers” and is usually faster but less standard
- ODBC (used to be Microsoft-specific) for C/C++.
- JDBC the standard for Java
- Scripting languages (PHP, Perl, JSP) are becoming the preferred technique for web-based systems.
a connection pool is a cache of database connections maintained so that the connections can be reused when future requests to the database are required. Connection pools are used to enhance the performance of executing commands on a database.