

# SQL: Queries, Constraints, Triggers

## Chapter 5

in the Course Textbook

SQL:1999

ramesh@cs.ubc.ca

<http://www.cs.ubc.ca/~ramesh/cpsc304>

# SQL: Structured Query Language

---

De Facto Standard

SQL:1999 - Current standard

Core SQL : Collection of features of SQL:1999 - A vendor **MUST** implement core SQL features to claim compliance with the SQL:1999 standard.

Also called as "SEQUEL"

# Various Aspects of SQL

## Data Manipulation Language (DML)

- Pose Queries, Insert, Delete, Modify rows

## Data Definition Language (DDL)

- Creation, Deletion and Modification of Table definitions and Views
- Integrity Constraints
- Commercial Implementations: Creation of Indexes

## Triggers and Advanced Integrity Constraints (ICs)

- Triggers: Actions executed by DBMS when changes to DB satisfy certain conditions

## Embedded and Dynamic SQL:

- SQL code called from a host language such as C, C++, COBOL or JAVA

## Other Aspects:

- Client-server execution and Remote DB access, Transaction Management
- Security, Advanced Features (OO, Recursive queries, XML, ...)

# Basic SQL Query

---

Syntax of a simple SQL query and its meaning through a *conceptual evaluation strategy* (based on ease of understanding rather than efficiency of evaluation)

```
SELECT [ DISTINCT ] select-list
FROM   from-list
WHERE  qualification
```

- Every query must have a SELECT clause.
  - ▷ Specified columns to be retained in the result
  - ▷ Corresponds to the PROJECT operator  $\pi$  in RA
- Every query must also have a FROM clause
  - ▷ Specifies a *cross product* of tables.
- WHERE clause is *optional*
  - ▷ Specifies *selection conditions* on the tables mentioned in the FROM clause.
- Corresponds to RA expressions with Selections, Projections and Cross Products

## Conceptual Evaluation Strategy

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

- Compute the cross product of the *from-list*
- Discard the resulting tuples if they fail *qualifications*
- Delete attributes that are not in the *select-list*
- Eliminate duplicates if DISTINCT is specified

This strategy may not be the most efficient way to compute the query. An optimizer will find more efficient strategies to compute the same answers

# Example

- Instance  $S_3$  of Sailors:

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

- Instance  $R_2$  of Reserves:

sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

- Instance  $B_1$  of Boats:

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

## Examples of SQL Queries

Query Q<sub>1</sub>:

```
SELECT DISTINCT S.sname, S.age
FROM Sailors S
```

- Answer to the query

sname	age
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Art	25.5
Bob	63.5

```
SELECT S.sname, S.age
FROM Sailors S
```

Answer WITHOUT DISTINCT

sname	age
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

## Examples of SQL Queries

If the `DISTINCT` keyword is used, the answer contains *distinct* tuples and duplicates are eliminated.

If the `DISTINCT` keyword is omitted, then for each sailor with name  $s$  and age  $a$ , the answer would be a multiset of rows.

- Consider the query: Find all sailors with a rating above 7

```
SELECT  S.sid, S.name, S.rating, S.age
FROM    Sailors AS S
WHERE   S.rating > 7
```

```
SELECT *
FROM    Sailors S
WHERE   S.rating > 7
```

- Uses the optional keyword `AS` to introduce a range variable
- `SELECT *` can be used as an alternative to select all columns in a relation
- `SELECT` Clause actually stands for *projection* in RA
- The conditions in `WHERE` clause do the *selection*

# Syntax of a Basic SQL Query

---

**from-list** - list of table names, followed by a *range variable*

**select-list** - List of expressions involving column names of tables named in the from-list (can be prefixed by a range variable)

**qualification** in the WHERE clause - boolean combination of conditions of the form `expression op expression` where `op` is  $\{<, >, =, \neq, \leq, \geq\}$ , `expression` is a column name, a constant or an arithmetic expression

DISTINCT keyword is optional (specifies result should not contain duplicates)

## Examples of more SQL Queries

- Find 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'
```

- Find the names of sailors who have reserved a red boat

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

- Join of three tables followed by a selection on the color of boats.

## More Examples

- Find the colors of boats reserved by Lubber

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

- In general, there may be more than one sailor called 'Lubber' (since sname is not a key)
- Boats reserved by *some* Lubber will be returned by the query
- Find the names of Sailors who have reserved at least one boat

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

- Join of Sailors and Reserves ensures that for each *sname*, the sailor has made some reservation.

# Expressions and Strings in SELECT

---

SQL: supports a more general version of the **select-list**

Each item in a **select-list** can be of the form *expression* **AS** *column\_name*

- *expression* is any arithmetic or string expression over column names and constants
- *column\_name* is a new name for this column in the output of the query
- Can also contain aggregates and expressions over date and time values

# EXAMPLES

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

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

- Each *qualification* can be as general as *expression1 = expression2* as in the following query

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

- Comparison operators like =, <, > etc. can be used for string comparison, ordering is alphabetical.

# String Operations

---

- For sorting strings other than alphabetical order (e.g. Month names in calendar order), SQL supports the general concept of **collation** or sort order for a character set.
- Allows users to specify which characters are less than which others
- SQL also supports *pattern matching* through the LIKE operator and wild card symbols
  - ▷ symbol % - stands for zero or more arbitrary characters
  - ▷ symbol \_ - stands for exactly one arbitrary character
  - ▷ Blanks can be significant for the LIKE operator
- Find the ages of sailors whose name begins and ends with B and has at least three characters

```
SELECT  S.age
FROM    Sailors S
WHERE   S.sname LIKE 'B_%B'
```

- SQL:1999 more powerful version of LIKE called SIMILAR - allows regular expressions in text search

# RA and SQL

---

Set operations of SQL are available in RA except that they are **multiset** operations in SQL.

Tables are multisets of tuples. Hence this model.

# UNION, INTERSECT and EXCEPT

---

SQL supports set operations  $\cup, \cap, -$  under the names UNION, INTERSECT and EXCEPT

The FINE PRINT: SQL includes these operations but many systems currently support only UNION. Also, many systems use MINUS for EXCEPT

Other set operations supported are

- IN - To check if an element is in a given set
- op ANY, op ALL - To compare a value with the elements in a given set using an operator op
- EXISTS - To check if a set is empty
- IN and EXISTS can also be prefixed with NOT

# EXAMPLES

- Find names of sailors who have reserved a red boat or a green boat

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid
        AND (B.color = 'Red' OR B.color = 'Green')
```

- Find the names of sailors who have reserved both a red boat and a green boat: Replacing OR in the previous query with AND will try to retrieve boats that are both red and green in color and will always return an empty answer.

```
SELECT  S.sname
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'
```

- Both the queries above can be expressed using UNION and INTERSECT

## EXAMPLES - UNION, INTERSECT

- The queries in the previous slide can be written as follows.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B,
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'Red'
UNION
SELECT  S2.sname
FROM    Sailors S2, Reserves R2, Boats B2,
WHERE   S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'Green'
```

- The query for "AND" can be written as follows.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B,
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'Red'
INTERSECT
SELECT  S2.sname
FROM    Sailors S2, Reserves R2, Boats B2,
WHERE   S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'Green'
```

## A Subtlety in the Intersect query

- If two sailors have the name *hook*, one has reserved a **red** boat and the other has reserved a **green** boat, then this name will be returned even though the *same* hook has not reserved both a red and a green boat.
- The CAUSE of this bug is because *sname* is not a key for Sailors and we are using it to identify sailors.

## Examples - Difference

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

```

SELECT  S.sid
FROM    Sailors S, Reserves R, Boats B,
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'Red'
EXCEPT
SELECT  S2.sid
FROM    Sailors S2, Reserves R2, Boats B2,
WHERE   S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'Green'

```

- We need sids - so we do not need the sailors relation. A simpler query is

```

SELECT  R.sid
FROM    Reserves R, Boats B,
WHERE   R.bid = B.bid AND B.color = 'Red'
EXCEPT
SELECT  R2.sid
FROM    Reserves R2, Boats B2,
WHERE   R2.bid = B2.bid AND B2.color = 'Green'

```

- The query assumes *referential integrity* - There are no reservations for nonexistent sailors

# Examples

- Find all sids of sailors who have a rating of 10 or reserved boat 104

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating = 10
UNION
SELECT  R.sid
FROM    Reserves R
WHERE   R.bid = 104
```

- IMPORTANT NOTE about DUPLICATE ELIMINATION: In contrast to the default that Duplicates are NOT eliminated unless the DISTINCT is specified,
  - ▷ The default for UNION queries is that duplicates are eliminated
  - ▷ To retain duplicates, UNION ALL must be used.
  - ▷ Similarly, INTERSECT ALL and EXCEPT ALL

# NESTED QUERIES

---

- A *nested query* is a query with another query embedded within it
- The embedded query is called a *subquery* (which can be a nested query too)
- Nesting of queries is not available in Relational Algebra but nested queries can be translated to Algebra queries
- Inspired more by calculus than algebra
- In conjunction with Aggregation, (multi) set operations, nesting is very expressive.

## Example

- Find the names of sailors who have reserved boat 103

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

- Find the names of sailors who have reserved a red boat

```
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
                                   WHERE   B.color = 'Red') )
```

## Example

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

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

- Note: In all our nested queries thus far, the inner subquery has been *completely* independent of the outer query
- In general, the inner subquery could depend on the row being examined in the outer query ⇒ **Correlated Nested Queries**

# Correlated Nested Queries

- Find the names of sailors who have reserved boat number 103

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

- EXISTS - another comparison operator such as IN. Allows us to test whether a set is nonempty (an implicit comparison with the null set)
- *Occurrence* of *S* in the subquery in the form of *S.sid* is called a *correlation* and this is an example of a correlated nested query
- Query also illustrates the use of *\** in situations where existence of rows is tested rather than retrieving some columns in a row (Other use is in aggregation)

## Correlated Nested Queries - Continued

- Using NOT EXISTS, we can compute the names of sailors who have not reserved a red boat
- Closely related to EXISTS is the UNIQUE predicate
- Applying UNIQUE to a subquery, the result is true if no row appears twice in the answer to the subquery (If there are no duplicates)
- Returns true if the answer is empty

## Set-Comparison Operators

- EXISTS, IN, UNIQUE and their *negated* versions
- op ANY and op ALL are two more operators where op is one of the arithmetic comparison operators {<, >, =, ≤, ≥, <>}
- SOME is available but is just a synonym for ANY
- Find sailors whose rating is better than some (every) sailor called Horatio

```

SELECT  S.sid
FROM    Sailors S
WHERE   S.rating > ANY(ALL) (SELECT  S2.rating
                             FROM    Sailors S2
                             WHERE   S2.name = 'Horatio' )

```

- Find the sailors with the highest rating

```

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

```

## More Examples of Nested Queries

- Find the names of sailors who have reserved a red boat and a green boat

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'Red'
        AND S.sid IN ( SELECT  S2.sid
                        FROM    Sailors S2, Reserves R2, Boats B2
                        WHERE   S2.sid = R2.sid AND R2.bid = B2.bid
                            AND B2.color = 'Green' )
```

- Illustrates how queries involving INTERSECT can be rewritten using IN
- Similarly queries using EXCEPT can be rewritten using NOT IN

## Example of DIVISION

- Find the names of sailors who have reserved all boats

```

SELECT  S.sname
FROM    Sailors S
WHERE   NOT EXISTS (( SELECT  B.bid
                       FROM    Boats B )
                   EXCEPT
                   (SELECT  R.bid
                    FROM    Reserves R
                    WHERE   R.sid = S.sid ))

```

- The query is correlated. Without using EXCEPT, the query can be written as

```

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 ))

```

# AGGREGATE OPERATORS

- Aggregate Operators - Help in performing additional computation/summarization on data
- Significant extension of Relational Algebra
- SQL supports *five* aggregate operators that can be applied on any column say  $A$  of a relation
  - ▷  $COUNT([DISTINCT]A)$  - The number of (unique) values in column  $A$
  - ▷  $SUM([DISTINCT]A)$  - The sum of all (unique) values in column  $A$
  - ▷  $AVG([DISTINCT]A)$  - The Average of all (unique) values in column  $A$
  - ▷  $MAX(A)$  - The maximum value in the  $A$  column
  - ▷  $MIN(A)$  - The minimum value in the  $A$  column
- SQL:1999 aggregate functions - greatly expanded including several statistical functions like standard deviation, covariance and percentiles - not supported by all vendors though

# EXAMPLES

- Find the average age of all sailors

```
SELECT  AVG (S.age)
FROM    Sailors S
```

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

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

- Find the name and age of the oldest sailor - **An illegal query is given below** - The intent is to return the name associated with the sailor with the maximum age

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

- If a SELECT clause uses *aggregate* operations, then it must use ONLY aggregate operations unless there is GROUP BY - This means that we cannot use S.sname and MAX(S.age) in the SELECT clause.

## Examples Continued

- The correct query is given below

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

- Observation: The following query is legal in the SQL standard but is unfortunately not supported in many systems

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

- Count the number of sailors

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

# Aggregate Operators Examples

- Count the number of different sailor names

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

- Aggregate operators provide an alternative to ANY and ALL - Find the names of sailors who are older than the oldest sailor with a rating of 10.

```
SELECT S.sname
FROM Sailors S
WHERE S.age > ( SELECT MAX ( S2.age )
                FROM Sailors S2
                WHERE S2.rating = 10 )
```

```
SELECT S.sname
FROM Sailors S
WHERE S.age > ALL ( SELECT S2.age
                    FROM Sailors S2
                    WHERE S2.rating = 10 )
```

## GROUP BY and HAVING Clauses

- Aggregate operators thus far have been used on *all* qualifying rows. Often these operators need to be applied on groups of rows
- Example: Find the age of the youngest sailor for each rating level - If we know the rating values are in the range 1 through 10, we write 10 queries of the form

```
SELECT  MIN  ( S.age )
FROM    Sailors S
WHERE   S.rating = i      (i ranging from 1 to 10)
```

- Major extension to SQL - GROUP BY and HAVING to specify qualification over groups

```
SELECT      [ DISTINCT ]  select-list
FROM        from-list
WHERE       qualification
GROUP BY   grouping-list
HAVING     group-qualification
```

# GROUP BY and HAVING Clauses - Points to Note

**select-list:** Consists of

- A list of column names
- A list of terms each having the form `aggop ( column-name ) AS new-name`

**grouping-list:** Every column that appears in the list of column names in select-list **must** also appear in the grouping-list

- A group is a collection of rows that agree on the values of columns in the grouping-list
- Each row in the result corresponds to one group

**group-qualification:** Expressions in the group-qualification in the HAVING clause must have a single value per group

- HAVING clause determines whether an answer row is to be generated for a given group
- SQL-92: A column appearing in group-qualification must appear as an argument to an aggregation operator or it must appear in grouping-list
- SQL:1999: Two new set of functions have been introduced that allows us to check whether *every* or *any* row in a group satisfies a condition

Entire table is a single group if GROUP BY is omitted

## GROUP BY and HAVING - Examples

- Find the age of the youngest sailor for each rating level

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

- Find the age of the youngest sailor who is at least 18 years for each rating level with at least two such sailors

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

- Let us evaluate this query step by step on an instance

## Evaluation Example

Instance S4 of Sailors:

First step is to construct the cross product of tables in the **from-list**

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

## Evaluation Steps

Step 2: Apply qualification in the WHERE clause: (S.age > 18)

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

The red row is removed.

## Evaluation Steps

Step 3: Eliminate unwanted columns - Retain only columns mentioned in the SELECT, GROUP BY or HAVING clause  $\Rightarrow$  eliminate sid and sname

Step 4: Sort the table according to the GROUP BY clause (S.rating)

After Step3 :

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

After Step4 :

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

## Evaluation Example

Step 5: Apply group-qualification in the HAVING clause -  $\text{COUNT} (*) > 1$

- Eliminates groups with rating 1, 9 and 10
- Order of considering WHERE and GROUP BY is important
- If WHERE is NOT applied first, then rating 10 would have met the group-qualification

Step 6: Generate one answer row for each remaining group

rating	minage
3	25.5
7	35.0
8	25.5

If query contains DISTINCT, duplicates are eliminated

## Examples

- Two new functions in SQL:1999 - EVERY and ANY
- Example:

```

SELECT      S.rating,  MIN (S.age) AS minage
FROM        Sailors S
WHERE       S.age >= 18
GROUP BY   S.rating
HAVING     COUNT (*) > 1 AND EVERY ( S.age <= 60 )

```

- In Step 5 of evaluation: Every row in a group must satisfy the condition  $S.age \leq 60$  to meet the group-qualification. Group 3 is hence dropped.
- The answer is

rating	minage
7	35.0
8	25.5

## A Slight Variation

- Consider a variation to the query in the previous slide

```
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
```

- Step 3 of the evaluation is altered now. Row with age 63.5 no longer exists.

rating	minage
3	25.5
7	35.0
8	25.5

## Aggregate Queries - Examples

- For each red boat, find the number of reservations for this boat

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

- The following is an *illegal* version of the above query

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

- The group-qualification `B.color = 'red'` is single-valued per group. `bid` is a key for `Boats` and therefore determines color. Therefore SQL disallows this query
- 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).
- In SQL:1999 this query can be rewritten easily using `EVERY` in the `HAVING` clause.

## More Examples

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

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

- An alternate way of writing the above query shows that the HAVING clause can have a nested subquery

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

## More Examples

- Find the average age of Sailors who are at least 18 years of age for each rating level that has at least two sailors

```
SELECT      S.rating,  AVG  (S.age)  AS Average
FROM        Sailors S
WHERE       S.age >= 18
GROUP BY   S.rating
HAVING      1 < (SELECT  COUNT (*)
                  FROM    Sailors S2
                  WHERE   S.rating = S2.rating )
```

- Find the average age of Sailors who are at least 18 years of age for each rating level that has at least two *such* sailors

```
SELECT      S.rating,  AVG  (S.age)  AS Average
FROM        Sailors S
WHERE       S.age >= 18
GROUP BY   S.rating
HAVING      1 < (SELECT  COUNT (*)
                  FROM    Sailors S2
                  WHERE   S.rating = S2.rating AND S2.age >= 18 )
```

## More Examples

- Find the average age of Sailors who are at least 18 years of age for each rating level that has at least two *such* sailors - Alternate Formulations
- WHERE clause is applied before GROUP BY - we can take advantage of this fact.

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

- Another formulation:

```
SELECT  Temp.rating, Temp.avgage
FROM    ( SELECT  S.rating, AVG ( S.age ) AS avgage,
              COUNT (*) AS ratingcount
          FROM    Sailors S
          WHERE   S.age >= 18
          GROUP BY S.rating ) AS Temp
WHERE   Temp.ratingcount > 1
```

# Discussion

## Several points of interest

- FROM clause can also contain nested subqueries
- HAVING clause is not needed at all - A query with HAVING can be rewritten into a query without one.
- Using a temporary name is *necessary* when a subquery appears in the FROM clause.

## More Examples

Aggregate operations cannot be nested

- Find those ratings for which the average age of sailors is the minimum over all ratings
- A first try - wrong one of course.. :)

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

- Will not work even if MIN(AVG(S2.age)) were allowed - MIN(AVG) will return the same value. Idea is to use temporary tables.

```
SELECT  Temp.rating, Temp.avgage
FROM    ( SELECT  S.rating, AVG ( S.age ) AS avgage,
                COUNT (*) AS ratingcount
          FROM    Sailors S
          GROUP BY  S.rating ) AS Temp
WHERE   Temp.avgage = ( SELECT  MIN (Temp.avgage) FROM Temp )
```

# NULL VALUES

---

- In practice, column values can be **unknown**
- SQL provides a special column value called *null* to use in such situations.
- We use *null* when the column value is either unknown or inapplicable.
- Presence of *null* complicates many issues - impact on SQL is considered here.
- Consider an example: Suppose **Dan** is a sailor who is new and hence does not have a **Rating** value. We can insert the tuple with a null value as  $\langle 98, \text{Dan}, \text{null}, 39 \rangle$

# NULL VALUES - COMPARISONS

---

- Consider a comparison `Rating = 8`. If this comparison is applied to the tuple where the `Rating` value is `null`, is it true or false?
- It is reasonable to say that the comparison should evaluate to `unknown`.
- Same is the case of `Rating > 8` and `Rating < 8` as well.
- How about comparing two `null` values using `<`, `>`, `=` etc.? The answer is `unknown`.
- SQL provides a special comparison operator `IS NULL` to test whether a column value is `null` (likewise `IS NOT NULL`)

# Logical Connectives

---

- Once we have null values, we must define logical operations AND, OR, NOT using a three-valued logic.
- The operations are defined as below:
- NOT unknown is unknown

AND	t	f	unknown
t	t	f	unknown
f	f	f	f

OR	t	f	unknown
t	t	f	t
f	f	f	unknown

# Impact on SQL Constructs

---

- WHERE clause: eliminates rows in which qualification does not evaluate to true
- In the presence of null values, any row evaluating to false or unknown is eliminated.
- Eliminating such rows has a subtle but significant impact on nested queries involving EXISTS or UNIQUE
- Duplicates: Two rows are duplicates if the corresponding columns are either equal or both contain null values
- Contrast this with comparison: Two null values when compared using = is unknown (In the case of duplicates, this comparison is implicitly treated as true which is an anomaly)
- Arithmetic operations +, -, \* and / all return null if one of their operators is null.
- COUNT(\*) treats null values just like other values (included in the count)
- ALL other operators COUNT, SUM, MIN, MAX, AVG and variations using DISTINCT - *discard null values*
- When applied to *ONLY* null values, then the result is null

# OUTER JOINS

- Interesting variant of the join operation that rely on *null values* - Supported in SQL
- Consider the join  $\text{Sailor} \bowtie_C \text{Reserves}$ .
  - ▷ Tuples of Sailors that do not match some row in Reserves according to the join condition  $C$  does not appear in the result
- In an outer join, sailor rows without a matching Reserves row appear exactly once in the result, with the columns from the Reserves relation assigned *null* values
- Variants:
  - ▷ **LEFT OUTER JOIN:** Sailor rows without matching Reserves row appear in the result, not vice versa.
  - ▷ **RIGHT OUTER JOIN:** Reserves rows without matching Sailor row appear in the result, not vice versa.
  - ▷ **FULL OUTER JOIN:** Both Sailor and Reserves rows without matching tuples, appear in the result.

# EXAMPLE

- In SQL, OUTER JOIN is specified in the FROM clause.

```
SELECT  S.sid, R.bid
FROM    Sailors S NATURAL LEFT OUTER JOIN Reserves R
```

- On the instances, the result is shown.

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

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

sid	bid
22	101
31	null
58	103

- We can disallow null values by specifying NOT NULL. There is an implicit NOT NULL constraint for every field listed in a PRIMARY KEY constraint.

# Complex Integrity Constraints

- Table Constraints over a single table using *CHECK conditional-expression* - To ensure that rating must be a value in the range 1 to 10

```
CREATE TABLE Sailors ( sid      INTEGER,  
                        sname    CHAR(10),  
                        rating   INTEGER,  
                        age      REAL,  
                        PRIMARY KEY (sid),  
                        CHECK ( rating >= 1 AND rating <= 10 ))
```

## Complex Integrity Constraints

- To enforce constraint that "Interlake" boats cannot be reserved, we use the following.

```
CREATE TABLE Reserves ( sid      INTEGER,
                        bid      INTEGER,
                        day      DATE,
                        PRIMARY KEY (sid,bid),
                        FOREIGN KEY (sid) REFERENCES Sailors
                        FOREIGN KEY (bid) REFERENCES Boats
                        CONSTRAINT noInterlakeRes
                        CHECK ( 'Interlake' <>
                                ( SELECT B.bname
                                  FROM Boats B
                                  WHERE B.bid = Reserves.bid )))
```

- When a row is inserted into Reserves, or an existing row is modified, the conditional-expression in the CHECK condition is evaluated and the command is rejected if the expression evaluates to false.

# Domain Constraints

- A new domain can be defined using the CREATE DOMAIN statement.

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1
        CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

- INTEGER is the *source* type for ratingval and every ratingval must be of this type.
- values are restricted by the CHECK constraint
- We can use the following in the schema declaration

```
rating ratingval
```

- Comparisons between types and the underlying source types will succeed.

## More Domain Constraints

- However, if BoatId and SailorId both have INTEGER as source types and we want comparisons between these types to fail, SQL:1999 has the notion of *distinct types*

```
CREATE TYPE ratingtype AS INTEGER
```

- Defines a new *distinct type* called ratingtype with INTEGER as source type.
- Comparisons between values of ratingtype and INTEGER not possible.
- Operations have to be defined explicitly on the new type.

## ICs over Several Tables

- Table constraints are required to hold only if the associated table is nonempty. May lead to actions that are not desired. To fortify this, SQL supports *assertions*
- As an example, suppose that number of boats plus the number of sailors should be less than 100.

```
CREATE TABLE Sailors (  sid      INTEGER,
                        sname   CHAR(10),
                        rating  INTEGER,
                        age     REAL,
                        PRIMARY KEY (sid),
                        CHECK ( rating >= 1 AND rating <= 10)
                        CHECK ( (SELECT COUNT (S.sid) FROM Sailors S )
                                + (SELECT COUNT (B.bid) FROM Boats B )
                                < 100 ) ) )
```

- Completely symmetric with respect to Boats (but associated with Sailors)
- If Sailors is empty, the constraint always holds even if Boats have over 100 rows.
- Checking Sailors to be nonempty is possible but cumbersome.

# Assertions

- Best approach is to create an assertion.

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

# Triggers and Active Databases

- A **trigger** is a procedure that is automatically invoked by the DB in response to specified changes to the DB
- Typically specified by the DBA
- A database that has a set of associated triggers is called an *active database*
- A trigger description contains three parts
  1. **Event:** A change to the database that activates the trigger
  2. **Condition:** A query or test that is run when a trigger is activated
  3. **Action:** A procedure that is executed when the trigger is activated and the condition is true

Here endeth the lesson on Triggers!!! Read up section 5.8 in text. :)

# EXAMPLES

- Consider the following schema

Flights(fno:integer, from:string, to:string, distance:integer, departs:time, arrives:time, price:integer)

Aircraft(aid:integer, aname:string, cruisingrange:integer)

Certified(eid:integer, aid:integer)

Employees(eid:integer, ename:string, salary:integer)

- Employees relation describes pilots and other kind of employees as well.
- EVERY pilot is certified for some aircraft and ONLY pilots are certified to fly.

## Examples of Queries

- Find the names of aircraft such that ALL pilots certified to operate them earn more than a 100,000 dollars

```

SELECT DISTINCT A.aname
FROM AIRCRAFT A
WHERE A.aid IN ( SELECT C.aid
                  FROM Certified C, Employee E
                  WHERE C.eid = E.eid AND
                        NOT EXISTS ( SELECT *
                                     FROM Employees E1
                                     WHERE E1.eid = E.eid AND E1.salary < 100000 ) )

```

- For each pilot who is certified for more than 5 aircrafts, find the eid and maximum cruisingrange of the aircraft for which he or she is certified.

```

SELECT C.eid, MAX(A.cruisingrange)
FROM Certified C, Aircraft A
WHERE C.aid = A.aid
GROUP BY C.eid
HAVING COUNT (*) > 3

```

## Examples of Queries

- Print the name and salary of every nonpilot whose salary is more than the average salary for pilots.

```
SELECT  E.ename, E.salary
FROM    Employees E
WHERE   E.eid IN ( SELECT  DISTINCT C.eid
                  FROM    Certified C )
AND     E.salary > ( SELECT  AVG (E1.salary)
                  FROM    Employees E1
                  WHERE   E1.eid IN
                  ( SELECT  DISTINCT C1.eid
                    FROM    Certified C1 ) )
```

## Examples of Queries

- Compute the difference between the average salary of a pilot and the average salary of all employees (including pilots)

```
SELECT      T1.avg - T2.avg
FROM        ( SELECT  AVG(E.salary) as avg
              FROM    Employees E
              WHERE   E.eid IN ( SELECT DISTINCT C.eid
                                FROM Certified C ) ) as T1,
            ( SELECT  AVG(E1.salary) as avg
              FROM    Employees E1 ) as T2
```

## Examples of Queries

- Print enames of pilots who can operate planes with cruising range greater than 3000 miles but who are not certified on any Boeing aircraft

```
SELECT  DISTINCT E.ename
FROM    Employees E
WHERE   E.eid IN ( ( SELECT  C.eid
                    FROM    Certified C
                    WHERE   EXISTS ( SELECT A.aid
                                    FROM Aircraft A
                                    WHERE A.aid = C.aid
                                    AND A.cruisingrange > 3000 )
                    AND
                    NOT EXISTS ( SELECT A1.aid
                                FROM Aircraft A1
                                WHERE A1.aid = C.aid
                                AND A1.aname LIKE 'Boeing%' ) ) )
```

# DB Application Development

## Parts of Chapter 6

in the Course Textbook

Embedded SQL, Cursors

ramesh@cs.ubc.ca

<http://www.cs.ubc.ca/~ramesh/cpsc304>

## DB access from Applications

### Relational DBMS - Supports an interactive SQL interface

- SQL executed from within a program in a host language like C or Java
- Called **Embedded SQL**

### Embedded SQL:

- Static SQL queries
- Can dynamically create queries at runtime and execute them - Dynamic SQL

# Embedded SQL

- SQL Statements within a host language - clearly marked so that a preprocessor can deal with them before compilation
- Host language variables used to pass arguments into an SQL command must be declared in SQL
- Some special host language variables must be declared in SQL (for communicating error conditions)
- Complications:
  - ▷ Data Types recognized by SQL and host language may not match - mismatch addressed by type casting
  - ▷ SQL - set oriented. This is addressed by means of **cursors**

## Declaring Variables and Exceptions

- SQL can refer to variables in the host program. Such variables are prefixed by a colon (:)

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

- SQL Standard defines correspondence between host language types and SQL types for a number of host languages
- Two special variables for reporting errors - in the SQL standard.
- `SQLCODE` - returns a negative value when an error condition occurs without specifying what a particular negative number denotes (type of this in C is long)
- `SQLSTATE` - Associates predefined values with some common error conditions introducing some uniformity to how errors are reported (type of this in C is `char[6]`)

# Embedding SQL Statements

- SQL statements must be prefixed by EXEC SQL
- Can appear anywhere where host language statements can appear

```
EXEC SQL
```

```
INSERT INTO Sailors VALUES (:c_sname, :c_sid, :c_rating, :c_age);
```

- SQLSTATE variable should be checked for errors and exceptions after each embedded SQL statement

```
EXEC SQL WHENEVER [ SQLERROR | NOT FOUND ] [ CONTINUE | GOTO stmt ]
```

# CURSORS

- **Impedence Mismatch:** SQL operates on sets of records, host languages do not support a clean set-of-records abstraction.
- **CURSOR** – Mechanism that allows us to retrieve rows one at a time from a relation
- Can declare a cursor on any relation or SQL query
- Once declared, we can *open* it, *fetch* the next row, *move* the cursor or *close* the cursor
- INSERT, DELETE and UPDATE statements typically require no cursor (some variants use cursors)
- SELECT usually requires a cursor though if the answer has a single row, it is not needed.

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating > :c_minrating;
```

## More Cursors

- Subsequent to the above definition, we can

```
OPEN sinfo
FETCH sinfo INTO :c_sname, :c_age;
CLOSE sinfo
```

- General form of a cursor declaration

```
DECLARE cursorname [ INSENSITIVE ] [SCROLL] CURSOR
    [ WITH HOLD ]
    FOR some query
    [ ORDER BY order-item-list ]
    [ FOR READ ONLY | FOR UPDATE ]
```

- A cursor can be a **read only cursor** or if on a base relation or an updatable view, to be an **updatable cursor**
- Cursor is Updatable by default unless it is a scrollable or insensitive cursor.

## More Cursors

- Scrollable cursor - Variants of the FETCH command can be used to position the cursor in very flexible ways. Otherwise, only the basic FETCH command can be used to get the next row
- Insensitive cursor - Cursor behaves as if it is ranging over a private copy of the collection of answer rows. Otherwise, other actions of transactions could modify these rows causing come unpredictable behaviour.
- Holdable cursor - Is not closed when the transaction is committed - Motivation from long transactions in which we access and possibly change a large number of rows of a table. Aborting a transaction entails redoing a transaction when restarted
- ORDER BY specifies the sort order in which the FETCH command has to retrieve the rows