

# More SQL

---

.....

## Reserves

## Sailors

<i>sid</i>	<i>sname</i>	<i>rating</i>	<i>age</i>
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

<i>sid</i>	<i>bid</i>	<i>day</i>
22	101	10/10/04
22	102	10/10/04
22	103	10/08/04
22	104	10/07/04
31	102	11/10/04
31	103	11/06/04
31	104	11/12/04
64	101	09/05/04
64	102	09/08/04
74	103	09/08/04

## Boats

<i>bid</i>	<i>bname</i>	<i>Color</i>
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

# Correlated Nested Queries (Revisit)

*Find names of sailors who have reserved boat 103*

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



Tests whether the set  
is nonempty

(For finding sailors who have **not** reserved boat 103, we would use **NOT EXISTS**)

# Correlated Nested Query

---

*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));
```



# ANY and ALL operators

---

*Find sailors whose rating is better than some sailor named Horatio*

```
SELECT S.sid  
FROM Sailors S  
WHERE S.rating > ANY (SELECT S2.rating  
                      FROM Sailors S2  
                      WHERE S2.sname='Horatio');
```

# Using ALL operator

---

*Find sailors whose rating is better than **every** sailor named Horatio*

```
SELECT S.sid  
FROM Sailors S  
WHERE S.rating > ALL(SELECT S2.rating  
                     FROM Sailors S2  
                     WHERE S2.sname='Horatio');
```

Note that IN is equivalent to = ANY  
NOT IN is equivalent to <> ALL

What if there were no sailor called Horatio?

# Last time we saw...

## Example of MAX operator

*Find the name and age of the oldest sailor*

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

But this is illegal in SQL!!

If the SELECT clause uses aggregate operation,  
it must **ONLY** use aggregate operations unless  
we use GROUP BY/HAVING



# Correct SQL Query for MAX

---

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

## Alternatively...

---

```
SELECT S.sname, S.age  
FROM Sailors S  
WHERE ROWNUM <= 1  
ORDER BY S.age DESC;
```

# Banking Examples

---

*branch (branch-id, branch-city, assets)*

*customer (customer-id, customer-name, customer-city)*

*account (account-number, branch-id, balance)*

*loan (loan-number, branch-id, amount)*

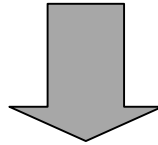
*depositor (customer-id, account-number)*

*borrower (customer-id, loan-number)*

# IN...Example I

---

*“Find the account numbers opened at branches of the bank in Fairfax”*



**SELECT A.account-number**

**FROM account A**

**WHERE A.branch-id IN (SELECT B.branch-id**

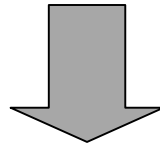
**FROM branch B**

**WHERE B.branch-city= 'Fairfax' )**

# IN...Example 2

---

*“Find the account numbers opened at branches 101 and 102 of the bank”*



```
SELECT A.account-number  
FROM A.account  
WHERE A.branch-id IN ( '101' , '102' )
```

# *EXISTS*

---

The *EXISTS* predicate is TRUE if and only if the Subquery returns a non-empty set.

The *NOT EXISTS* predicate is TRUE if and only if the Subquery returns an empty set.

The *NOT EXISTS* can be used to implement the SET DIFFERENCE operator from relational algebra.

# EXISTS...Example 1

*“Select all the account balances where the account has been opened in a branch in Fairfax”*

---

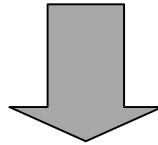


```
SELECT A.account-balance  
FROM account A  
WHERE EXISTS (SELECT *  
               FROM branch B  
               WHERE B.branch-city= 'Fairfax'  
               AND B.branch-id=A.branch-id)
```

# EXISTS...Example 2

---

*“Select all the account balances where the account has not been opened in a Fairfax branch ”*



**SELECT A.account-balance**

**FROM account A**

**WHERE NOT EXISTS (SELECT \***

**FROM branch B**

**WHERE B.branch-city= 'Fairfax'**

**AND B.branch-id=A.branch-id)**



---

\_\_\_\_\_

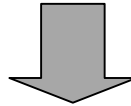


# *Quantified Comparison Predicate*

## **Example I**

---

*“Select account numbers of the accounts with the minimum balance”*



**SELECT A.account-number**

**FROM account A**

**WHERE A.balance <= ALL (SELECT A2.balance  
FROM account A2)**

# Aggregate Functions in SQL...

## revisited

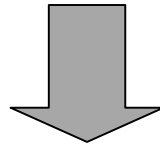
- SQL provides five built-in aggregate functions that operate on sets of column values in tables:
- *COUNT( ), MAX( ), MIN( ), SUM( ), AVG( )*.
- With the exception of *COUNT( )*, these set functions must operate on sets that consist of simple values-that is, sets of numbers or sets of character strings, rather than sets of rows with multiple values.

# Aggregate Functions in SQL

## Example I

---

*“Select the total amount of balance of the account in branches located in Fairfax”*



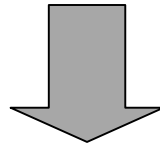
```
SELECT SUM(A.balance) AS total_amount  
FROM account A, branch B  
WHERE B.branch-city= 'Fairfax' AND  
B.branch-id= A.branch-id
```

# Aggregate Functions in SQL

## Example 2

---

*“Select the total number of opened accounts”*



```
SELECT COUNT(A.account-number)
```

```
FROM account A
```

***OR***

```
SELECT COUNT(*)
```

```
FROM account
```

# 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  $i = 1, 2, \dots, 10$ :

```
SELECT MIN (S.age)
FROM Sailors S
WHERE S.rating = i
```

# Queries With GROUP BY and HAVING

|          |                               |
|----------|-------------------------------|
| SELECT   | [DISTINCT] <i>target-list</i> |
| FROM     | <i>relation-list</i>          |
| WHERE    | <i>qualification</i>          |
| GROUP BY | <i>grouping-list</i>          |
| HAVING   | <i>group-qualification</i>    |

- 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 the age of the youngest sailor with age  $\geq 18$ , for each rating with at least 2 such sailors

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age  $\geq$  18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes 'unnecessary'.
- 2nd column of result is unnamed. (Use AS to name it.)

| <u>sid</u> | sname   | rating | age  |
|------------|---------|--------|------|
| 22         | dustin  | 7      | 45.0 |
| 31         | lubber  | 8      | 55.5 |
| 71         | zorba   | 10     | 16.0 |
| 64         | horatio | 7      | 35.0 |
| 29         | brutus  | 1      | 33.0 |
| 58         | rusty   | 10     | 35.0 |

| rating | age  |
|--------|------|
| 1      | 33.0 |
| 7      | 45.0 |
| 7      | 35.0 |
| 8      | 55.5 |
| 10     | 35.0 |

| rating |      |
|--------|------|
| 7      | 35.0 |

*Answer relation*

Find the age of the youngest sailor with age  $\geq 18$ , for  
each rating with at least 2 sailors (of any age)

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

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

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

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

Illegal!

*Can be rewritten using EVERY in HAVING:*

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

---

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

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

Can use nested aggregates with Group By

# Null Values

---

- We use *null* when the column value is either *unknown* or *inapplicable*.
- A comparison with at least one null value always returns *unknown*.
- SQL also provides a special comparison operator *IS NULL* to test whether a column value is *null*.
- To incorporate nulls in the definition of duplicates we define that two rows are duplicates if corresponding rows are equal or both contain *null*.

# Deal with the null value

---

- Special operators needed to check if value is/is not *null*.
  - “is null” always true or false (never unknown)
  - “is not null”
- Is *rating* > 8 true or false when *rating* is equal to *null*?
  - Actually, it's unknown.
  - Three-valued logic

# Three valued logic

| AND     | False | True    | Unknown |
|---------|-------|---------|---------|
| False   | False | False   | False   |
| True    | False | True    | Unknown |
| Unknown | False | Unknown | Unknown |

| OR      | False   | True | Unknown |
|---------|---------|------|---------|
| False   | False   | True | Unknown |
| True    | True    | True | True    |
| Unknown | Unknown | True | Unknown |

|         | NOT     |
|---------|---------|
| False   | True    |
| True    | False   |
| Unknown | Unknown |

# Other issues with the null value

---

- WHERE and HAVING clause eliminate rows that don't evaluate to true (i.e., rows evaluate to false or unknown).
- All of +, -, \*, / return null if any argument is null
- Can force "no nulls" while creating a table
  - sname char(20) NOT NULL
  - primary key is always not null
- Aggregate functions ignore nulls (except count (\*))
- DISTINCT treats all nulls as the same



# Aggregates with NULL

| sid | sname  | rating | age |
|-----|--------|--------|-----|
| 22  | dustin | 7      | 45  |
| 31  | lubber | 8      | 55  |
| 58  | rusty  | 10     | 35  |

R1

| sid | sname  | rating | age |
|-----|--------|--------|-----|
| 22  | dustin | 7      | 45  |
| 31  | lubber | null   | 55  |
| 58  | rusty  | 10     | 35  |

R2

What do you get for

- SELECT count(\*) from R1?
- SELECT count(rating) from R1?

What do you get for

- SELECT count(\*) from R2?
- SELECT count(rating) from R2?

- Aggregate functions ignore nulls (except count (\*))

# Aggregates with NULL

- COUNT, SUM, AVG, MIN, MAX (with or without DISTINCT)
  - Discards null values first
  - Then applies the aggregate
  - Except count(\*)
- If only applied to null values, the result is null

| sid | sname  | rating | age |
|-----|--------|--------|-----|
| 22  | dustin | 7      | 45  |
| 31  | lubber | null   | 55  |
| 58  | rusty  | 10     | 35  |

R2

SELECT sum(rating) from R2?

Ans: ...

| sid | sname  | rating | age |
|-----|--------|--------|-----|
| 22  | dustin | null   | 45  |
| 31  | lubber | null   | 55  |
| 58  | rusty  | null   | 35  |

R3

SELECT sum(rating) from R3?

Ans: ...

# General Constraints: CHECK

- Useful when more general ICs than keys are involved
- Constraints can be named
- Not checked if table is empty
- Standalone CHECK for single table only

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

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

# Complex Constraints: **Assertions**

---

- Table constraints apply to only one table
- Assertions are constraints that are separate from **CREATE TABLE** statements
  - Similar to domain constraints, they are separate statements in the DB schema
  - Assertions are tested **whenever the DB is updated**
    - Therefore they may introduce significant overhead

Note: Not supported in Oracle

# Assertions

## Example

---

- Number of boats plus number of sailors is  $< 100$ 
  - Not associated with a particular table.
  - Constraint may apply to multiple tables.

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

# Assertion Limitations

---

- There are some constraints that cannot be modeled with table constraints or assertions
  - What if there were participation constraints between customers and accounts?
    - Every customer must have at least one account and every account must be held by at least one customer
  - An assertion *could* be created to check this situation
    - But would prevent new customers or accounts being added!

# Views

---

- A **view** is just a relation, but we store a **definition**, rather than a set of tuples

```
CREATE VIEW YoungActiveStudents (name, grade)
AS SELECT S.name, E.grade
FROM Students S, Enrolled E
WHERE S.sid = E.sid and S.age < 21
```

- Views can be dropped using the **DROP VIEW** command
- **Views and Security:** Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s)
  - the above view hides courses “cid” from E

# Create a new table from a query on other tables

---

SELECT... INTO... FROM... WHERE

```
SELECT S.name, E.grade  
INTO YoungActiveStudents  
FROM Students S, Enrolled E  
WHERE S.sid = E.sid and S.age < 21
```



# Outer Joins

---

- Let R and S be two tables. The outer join preserves the rows of R and S that have no matching rows according to the join condition and outputs them with nulls at the non- applicable columns.
- There exist three different variants: *left outer join*, *right outer join* and *full outer join*.

# Outer joins

| <u>sid</u> | sname  | rating | age  |
|------------|--------|--------|------|
| 22         | dustin | 7      | 45.0 |
| 31         | lubber | 8      | 55.5 |
| 58         | rusty  | 10     | 35.0 |

(left outer-join)

| <u>sid</u> | <u>bid</u> | <u>day</u> |
|------------|------------|------------|
| 22         | 101        | 10/10/96   |
| 58         | 103        | 11/12/96   |

=

| <u>sid</u> | sname  | rating | age  | bid  | day      |
|------------|--------|--------|------|------|----------|
| 22         | dustin | 7      | 45.0 | 101  | 10/10/96 |
| 31         | lubber | 8      | 55.5 | Null | Null     |
| 58         | rusty  | 10     | 35.0 | 103  | 11/12/96 |

# Outer Joins in Oracle

---

```
Select *  
From Sailor S, Reserve R  
Where S.sid = R.sid(+);
```

*How about:*

```
Select S.sid, count(R.bid)  
From Sailor S, Reserve R  
Where S.sid = R.sid(+)  
Group by S.sid;
```

*OR*

```
Select S.sid, count(*)  
From Sailor S, Reserve R  
Where S.sid = R.sid(+)  
Group by S.sid;
```

# More Outer Joins

---

- Left outer join
  - + sign on the right in Oracle:  
Select \* from R, S where R.id=S.id(+)
- Right outer join
  - + sign on the left in Oracle:  
Select \* from R, S where R.id(+)=S.id
- Full outer join
  - Not implemented in Oracle 8
  - Added for Oracle 9 (or later)
    - Use full text instead of +’s: “full outer join”, “left outer join”, “right outer join”, “inner join”

# Conceptual order in query evaluation

---

- The cross products of the tables in the *FROM* clause are evaluated.
- Rows not satisfying the *WHERE* clause are eliminated.
- The remaining rows are grouped in accordance with the *GROUP BY* clause.
- Groups not satisfying the *HAVING* clause are then eliminated.
- The expressions in the *SELECT* list are evaluated.
- If the keyword *DISTINCT* is present, duplicate rows are now eliminated.
- Evaluate *UNION*, *INTERSECT* and *EXCEPT* for Subqueries up to this point.
- Finally, the set of all selected rows is sorted if the *ORDER BY* is present.

# Conclusion

---

- Nested queries are a very powerful feature in SQL; they help us write shorter and more efficient queries.
- Post processing on the result of queries is supported.
- Aggregation is the most complex “post processing”
  - “Group by” clause partition the results into groups
  - “Having” clause puts condition on groups (just like Where clause on tuples).