

## CS660: Intro to Database Systems

# Class 4: SQL, The Query Language – Part II

Instructor: Manos Athanassoulis

<https://bu-disc.github.io/CS660/>

# Recap: Basic SQL Query

<b>SELECT</b>	<b>[DISTINCT]</b>	<b>target-list</b>
<b>FROM</b>	<b>relation-list</b>	
<b>WHERE</b>	<b>qualification</b>	

*relation-list* : a list of relations

*target-list* : a list of attributes of tables in *relation-list*

*qualification* : comparisons using AND, OR and NOT

comparisons are: <attr> <op> <const> or <attr1> <op> <attr2>, where *op* is:

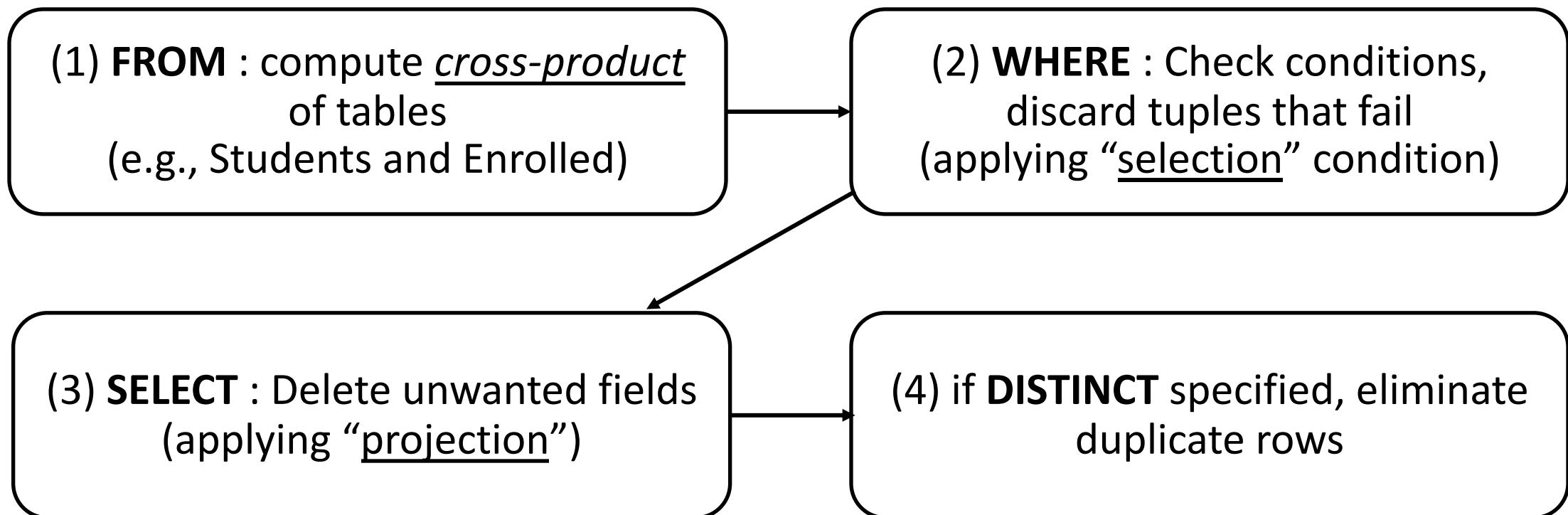
<,>,=,≤,≥,≠

*DISTINCT*: optional, removes duplicates

By default SQL SELECT does *not* eliminate duplicates! (“multiset”)

# Recap: Query Semantics

Conceptually, a SQL query can be computed:



probably the least efficient way to compute a query!

**Query Optimization** finds the *same answer* more efficiently

# Recap: Range Variables

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

can be  
rewritten using  
range variables as:

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

Can use Range Variables – do not need though. Why?

# Recap: Expressions

Use AS to provide column names

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

age2=2\*S.age

equivalent

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
```

# Recap: Expressions

Use **AS** to provide column names

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

age2=2\*S.age

equivalent

BUT only MS SQL Server supports it

MySQL, PostgreSQL, Oracle, SQLite do not!



# Recap: String operations

SQL also supports some string operations

“LIKE” is used for string matching.

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

'\_' stands for any one character

'%' stands for 0 or more arbitrary characters

>, < string comparison is supported by most systems

# Recap: Nested Queries

WHERE clause can itself contain an SQL query!

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

# Recap: Nested Queries with Correlation

Subquery must be recomputed for each Sailors tuple.

Think of subquery as a function call that runs a query!

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

# Let's revisit Query #3

3. Find all sailors who have not reserved a red boat

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

# Recap: Set-Difference using NOT IN

Find all sailors who have not reserved a red boat

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

Nested – NO correlation!

# Recap: Set-Difference using NOT EXISTS

Find all sailors who have not reserved a red boat

```
SELECT S.sid
FROM   Sailors S
WHERE  NOT EXISTS
       (SELECT *
        FROM Reserves R, Boats B
        WHERE R.sid = S.sid
              AND R.bid = B.bid
              AND B.color = 'red')
```

Nested – correlation!

# Recap: Set Operations

```
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'
```

```
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'
```

# Let's revisit UNION

we said they are equivalent

but do they always  
give the same result?



[example](#)

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

VS.

[example](#)

```
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'
```

# Recap: ANY and ALL Set-Comparison Operators

Find sailors with rating greater than the rating of at least one sailor called 'Horatio':

```
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT s2.rating
                        FROM Sailors S2
                        WHERE S2.sname='Horatio')
```

Find sailors with rating greater than the rating of all 20-year old sailors:

```
SELECT *
FROM Sailors S
WHERE S.rating > ALL (SELECT s2.rating
                        FROM Sailors S2
                        WHERE S2.age = 20)
```

# Division (“for all”) in SQL

**Find sailors who have reserved all boats.**

## *Sailors S for which ...*

# Division (“for all”) in SQL - alternative

Find sailors who have reserved all boats.

*Sailors S for which ...*

```
SELECT S.sname
FROM   Sailors S  there is no boat B without ...
WHERE  NOT EXISTS (SELECT B.bid
                   FROM   Boats B
                   EXCEPT
                           (SELECT R.bid
                            FROM   Reserves R
                            WHERE  R.bid=B.bid
                                   a Reserves tuple AND R.sid=S.sid))
                           showing S reserved B
```

# Aggregate Operators

Significant extension of relational algebra.

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

*single column*

```
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'
```

# Aggregate Operators

COUNT (\*)  
COUNT ( [DISTINCT] A)  
SUM ( [DISTINCT] A)  
AVG ( [DISTINCT] A)  
MAX (A)  
MIN (A)

*single column*

```
SELECT S.sname  
FROM Sailors S  
WHERE S.rating = (SELECT MAX(s2.rating)  
                  FROM Sailors s2)
```

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



# Find name and age of the oldest sailor(s)

The first query is incorrect!

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

Third query equivalent to second query  
allowed in SQL/92 standard, but not  
supported in some systems.

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 (SELECT MAX (S2.age)  
FROM Sailors S2)  
= S.age

# ARGMAX?

The Sailor with the highest rating

What about ties for highest?

```
SELECT *
FROM   Sailors S
WHERE  S.rating >= ALL
       (SELECT s2.rating
        FROM   Sailors s2)
```

```
SELECT *
FROM   Sailors S
WHERE  S.rating =
       (SELECT MAX(s2.rating)
        FROM   Sailors s2)
```

```
SELECT *
FROM   Sailors S
ORDER BY rating DESC
LIMIT 1;
```

# JOINS

# Joins

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

INNER is default

```
SELECT sname FROM sailors S JOIN reserves R ON S.sid=R.sid;
```

```
SELECT sname FROM sailors S NATURAL JOIN reserves R
WHERE R.bid = 102;
```

# Inner Joins

```
SELECT s.sid, s.sname, r.bid  
      FROM Sailors s, Reserves r  
     WHERE s.sid = r.sid
```

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

They are  
equivalent!

# 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.sname, r.bid  
  FROM Sailors s LEFT OUTER JOIN  
       Reserves r  
    ON s.sid = r.sid;
```

Returns all sailors & bid for boat in any of their reservations

Note: no match for s.sid? r.sid IS NULL!

```
SELECT s.sid, s.sname, r.bid  
FROM Sailors s LEFT OUTER JOIN Reserves r  
ON s.sid = r.sid;
```

<u>sid</u>	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
95	103	11/12/96

s.sid	s.name	r.bid
22	Dustin	101
95	Bob	103
31	Lubber	NULL

← NULL

# Right Outer Join

Returns all matched rows, plus all unmatched rows from the table on the **right** of the join clause

(use nulls in fields of non-matching tuples)

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

Returns all boats & information on which ones are reserved

Note: no match for b.bid? r.bid IS NULL!

# Full Outer Join

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

```
SELECT r.sid, b.bid, b.bname  
FROM Reserves2 r FULL OUTER JOIN  
Boats2 b  
ON r.bid = b.bid;
```

Returns all boats & all information on reservations

No match for r.bid?

- b.bid IS NULL AND b.bname is NULL

No match for b.bid?

- r.sid is NULL

# GROUP BY AND HAVING

# 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]  target-list
FROM        relation-list
WHERE       qualification
GROUP BY   grouping-list
[HAVING]   group-qualification
```

Group rows by columns in *grouping-list*

Every column from *target-list* must appear in the *grouping-list*

**HAVING** restricts through an *aggregate* which group-rows are part of the result

# Conceptual Evaluation

(1) Cross-product of  
*relation-list*

(2) Select only tuples that  
follow the where clause  
*qualification*)

(3) Partition rows by the value  
of attributes in *grouping-list*

(4) Select only groups that  
follow the *group-qualification*

(5) One answer tuple is  
generated per qualifying  
group, showing *target-list*

Attributes in *target-list* must also be in *grouping-list*.

Expressions in *group-qualification* must have a single value  
per group! That is, attributes in *group-qualification* must be  
part of an aggregate op / must appear in the *grouping-list*.

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 >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

2

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

3

rating	m-age	count
1	33.0	1
7	35.0	2
8	55.0	1
10	35.0	1

sid	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

4

rating	
7	35.0

```

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

```

s.sname	s.sid	r.sid	r.bid
Dustin	22	22	101
Lubber	31	22	101
Bob	95	22	101
Dustin	22	95	102
Lubber	31	95	102
Bob	95	95	102

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

Count (\*) from boats = 4

s.sname	s.sid	bcount
Dustin	22	1
Bob	95	1



s.sname	s.sid

Apply having clause to groups

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



can I ask...

~~SELECT S.rating, MIN (S.age)  
FROM Sailors S  
WHERE S.age >= 18  
GROUP BY S.rating  
HAVING S.name = 'Horatio'~~

what about ...

```
SELECT S.rating, MIN (S.age)  
FROM Sailors S  
WHERE S.age >= 18  
GROUP BY S.rating  
HAVING S.rating < 9
```

or...

...  
GROUP BY S.rating  
HAVING MIN (S.age) < 30

# Sorting the Results of a Query

ORDER BY *column* [ ASC | DESC ] [, ...]

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

Extra reporting power obtained by combining with aggregation.

```
SELECT      S.sid, COUNT (*) AS redrescnt
FROM        Sailors 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;
```

# Summary: The SQL Query

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

# Remember? Division (“for all”) in SQL

**Find sailors who have reserved all boats.**

## *Sailors S for which ...*

# Can you do this using Group By and Having?

Find sailors who have reserved all boats.

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



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

- (1) Attributes in *target-list* must also be in *grouping-list*.
- (2) Expressions in *group-qualification* must have a single value per group!
- (3) Without sid we are grouping together sailors with the same name!

# An Illustration

```

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)
  
```

sname	sid	bid
Frodo	1	102
Bilbo	2	101
Bilbo	2	102
Frodo	1	102
Bilbo	2	103

sname	sid	bid
Frodo	1	102,102
Bilbo	2	101, 102, 103

Boats

bid	bname	color
101	Nina	red
102	Pinta	blue
103	Santa Maria	red

Sailors

Reserves

sid	sname	rating	age
1	Frodo	7	22
2	Bilbo	2	39
3	Sam	8	27

Boats

count
3

sid	bid	day
1	102	9/12
2	102	9/12
2	101	9/14
1	102	9/10
2	103	9/13

# REVISITING DDL, NULL, AND MORE

```
INSERT [INTO] table_name [(column_list)]  
VALUES (value_list)
```

```
INSERT [INTO] table_name [(column_list)]  
<select statement>
```

```
INSERT INTO Boats VALUES ( 105, 'Clipper', 'purple')  
INSERT INTO Boats (bid, color) VALUES (99, 'yellow')
```

You can also do a “bulk insert” of values from one table into another:

```
INSERT INTO TEMP(bid)  
SELECT r.bid FROM Reserves R WHERE r.sid = 22;  
(must be type compatible)
```

```
DELETE [FROM] table_name  
[WHERE qualification]
```

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 NULL/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.

# NULLs

example

branch2 =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horse	1.7M
Mianus	Horse	.4M
Kenmore	Boston	NULL

Effect on Queries:

SELECT \* FROM branch2 WHERE assets = NULL

SELECT \* FROM branch2 WHERE assets IS NULL

What does this mean?

We don't know Kenmore's assets?  
Kenmore has no assets?

.....

bname	bcity	assets
Kenmore	Boston	NULL

bname	bcity	assets
Kenmore	Boston	NULL

# NULLs

Arithmetic with nulls:

- $n <\text{op}> \text{null} = \text{null}$
- $<\text{op}>$ : +, -, \*, /, mod, ...

“Booleans” with nulls: One can write:  
3-valued logic (true, false, unknown)

```
SELECT .....
FROM .....
WHERE boolexpr IS UNKNOWN
```

## What expressions evaluate to UNKNOWN?

1. Comparisons with NULL (e.g., assets = NULL)
2. FALSE OR UNKNOWN (but: TRUE OR UNKNOWN = TRUE)
3. TRUE AND UNKNOWN
4. UNKNOWN AND/OR UNKNOWN

# NULLs

Aggregate operations:

SELECT SUM(assets)  
FROM branch2

returns

SUM  
-----  
11.1M

branch2=

bname	bcity	assets
Downtown	Boston	9M
Perry	Horse	1.7M
Mianus	Horse	.4M
Kenmore	Boston	NULL

NULL is ignored  
Same for AVG, MIN, MAX

Let branch3 an empty relation

Then: SELECT SUM(assets)

FROM branch3      returns    NULL

but    SELECT COUNT(\*) FROM branch3      returns 0

# Views

Makes development **simpler**

Often used for **security**

**Not instantiated** - makes updates tricky

```
CREATE VIEW view_name
AS select_statement
```

```
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
```

# An illustration

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
```

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

b.bid	scount
102	1

**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
```

```
SELECT bname, scount  
      FROM Reds R, Boats B  
     WHERE R.bid=B.bid  
       AND scount < 10
```

b.bid	scount
102	1

**Reds**

# Views vs INTO

(1) `SELECT bname, bcity  
FROM branch  
INTO branch2`

vs

(2) `CREATE VIEW branch2 AS  
SELECT bname, bcity  
FROM branch`

(1) creates a new table that gets stored on disk

(2) creates a “virtual table” (materialized when needed)

Therefore: **changes in branch are seen in (2) but not in (1)**

# 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) – to create a foreign key references to <cols>
- All

Can later be **REVOKED**

**Users** can be **single users** or **groups**

See [Chapter 17](#) for more details.

Assertions and Triggers

# **CONSTRAINTS**

# Integrity Constraints

- predicates on the database
- must always be true (checked whenever db gets updated)

There are the following 4 types of IC's:

## **Key** constraints (1 table)

e.g., 2 accts can't share the same acct\_no

## **Attribute** constraints (1 table)

e.g., 2 accts must have nonnegative balance

## **Referential Integrity** constraints ( 2 tables)

E.g. bnames associated w/ loans must be names of real branches

## **Global** Constraints (n tables)

E.g., a loan must be carried by at least 1 customer with a svngs acct

# Global Constraints

Idea: two kinds

- 1) **single relation** (constraints spans multiple columns)

E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

- 2) **multiple relations**: CREATE ASSERTION

SQL examples:

- 1) **single relation**: All BOSTON branches must have assets > 5M

```
CREATE TABLE branch (
    .....
    bcity CHAR(15),
    assets INT,
    CHECK (NOT(bcity = 'BOS') OR assets > 5M))
```

**Affects:**

**insertions** into branch

**updates** of bcity or assets in branch

## Global Constraints

SQL example:

2) **Multiple relations:** every loan has a borrower with a savings account

```
CHECK (NOT EXISTS (
    SELECT *
    FROM loan AS L
    WHERE NOT EXISTS(
        SELECT *
        FROM borrower B, depositor D, account A
        WHERE B.cname = D.cname AND
              D.acct_no = A.acct_no AND L.lno = B.lno)))
```

Problem: Where to put this constraint? At depositor? Loan? ....

Ans: None of the above:

```
CREATE ASSERTION loan-constraint
    CHECK( ..... )
```

Checked with EVERY DB update!  
very expensive....

# Global Constraints

Issues:

- 1) How does one decide what global constraint to impose?
- 2) How does one minimize the cost of checking the global constraints?

Ans: Semantics of application and Functional dependencies.

## Summary: Integrity Constraints

Constraint Type	Where declared	Affects...	Expense
Key Constraints	CREATE TABLE (PRIMARY KEY, UNIQUE)	Insertions, Updates	Moderate
Attribute Constraints	CREATE TABLE  CREATE DOMAIN (Not NULL, CHECK)	Insertions, Updates	Cheap
Referential Integrity	Table Tag (FOREIGN KEY ....  REFERENCES ....)	1. Insertions into referencing rel'n 2. Updates of referencing rel'n of relevant attrs 3. Deletions from referenced rel'n 4. Update of referenced rel'n	1,2: like key constraints. Another reason to index/sort on the primary keys 3,4: depends on a. update/delete policy chosen b. existence of indexes on foreign key
Global Constraints	Table Tag (CHECK)  or  outside table (CREATE ASSERTION)	1. For single rel'n constraint, with insertion, deletion of relevant attrs 2. For assertions w/ every db modification	1. cheap  2. very expensive

# 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, 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  $\geq 70000$

# 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
```

```
    check for violation here ?
```

```
END;
```

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor  
FOR EACH ROW  
  
BEGIN  
  
IF (:new.salary < 70000)  
THEN RAISE_APPLICATION_ERROR (-20004,  
'Violation of Minimum Professor Salary');  
END IF;  
  
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 < 70000)

- then an application error is raised and hence the row is not inserted; otherwise the row is inserted.

## Use error code: -20004;

<sup>66</sup> – this is in the valid range

# Example Trigger Using Condition

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
    FOR EACH ROW
WHEN (new.salary < 70000)
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: REFERENCING

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor  
REFERENCING NEW as newTuple  
FOR EACH ROW  
WHEN (newTuple.salary < 70000)  
BEGIN  
    RAISE_APPLICATION_ERROR (-20004,  
    'Violation of Minimum Professor Salary');  
END;
```

# Example Trigger

```
CREATE TRIGGER updSalary
    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 (SQL:99)

```
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

# Row vs Statement Level Trigger

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**,

→ all the rows are inserted first,

→ if the condition fails → all inserts 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 < 70000) 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>;
```

# 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?)



# Links for Examples

Schema is available at:

<https://gist.github.com/manathan1984/35b189ae92fd996cce7816e2d7f9e40f>

Lightweight online SQL frontend:

<http://sqlfiddle.com/>