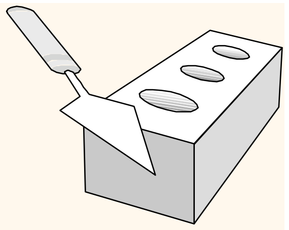


SQL: Queries, Programming, Triggers

Chapter 5

Introduction

- We now introduce SQL, the standard query language for relational DBS.
- Like relational algebra, an SQL query takes one or two input tables and returns one output table.
- Any RA query can also be formulated in SQL.
- In addition, SQL contains certain features of that go beyond the expressiveness of RA, e.g. sorting and aggregation functions.

Example Instances

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
28	yuppy	9	35.0
44	guppy	5	35.0
53	bob	null	18.0

Boats

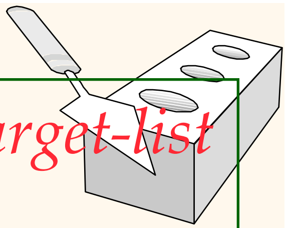
bid	colour
101	green
103	red

Reserves

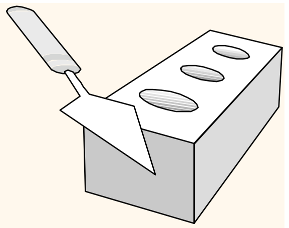
sid	bid	day
22	101	10/10/96
58	103	11/12/96

Basic SQL Query

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



- *relation-list* A list of relation names (possibly with a *range-variable* after each name).
- *target-list* A list of attributes of relations in *relation-list*
- *qualification* Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of $<, >, =, \leq, \geq, \neq$) combined using AND, OR and NOT.
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated!



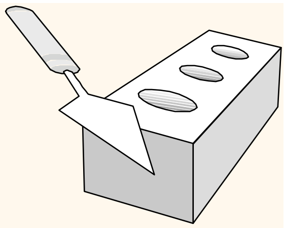
Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of *relation-list*.
  - Discard resulting tuples if they fail *qualifications*.
  - Delete attributes that are not in *target-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is typically the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Example of Conceptual Evaluation

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

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96



A Note on Range Variables

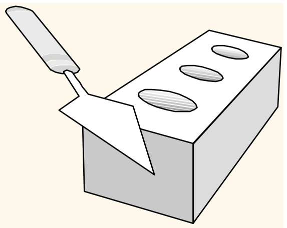
- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

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

OR

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

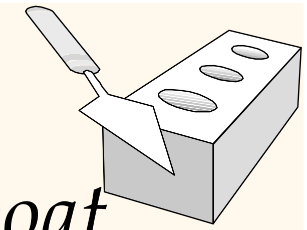
It is good style, however, to use range variables always!



π - σ - $\times$ Queries

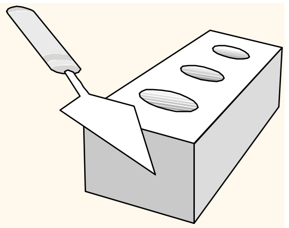
- SELECT [DISTINCT] *S.sname*
  - π_{sname}
- FROM *Sailors, Reserves*
  - Sailors × Reserves
- WHERE S.sid=R.sid AND R.bid=103
  - $\sigma_{Sailors.sid = Reserves.sid \text{ and } Reserves.bid=103}$
  - SELECT S.sname
    FROM Sailors S, Reserves R
    WHERE S.sid=R.sid AND R.bid=103
    =
    $\pi_{sname}(\sigma_{Sailors.sid = Reserves.sid \text{ and } Reserves.bid=103}(Sailors \times Reserves))$
  - It is often helpful to write an SQL query in the same order (FROM, WHERE, SELECT).

Find sailors who've reserved at least one boat



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

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

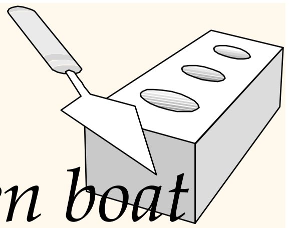


Expressions and Strings

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

- Illustrates use of arithmetic expressions and string pattern matching: *Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.*
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters. case sensitvity Oracle on Strings

Find sid's of sailors who've reserved a red or a green boat

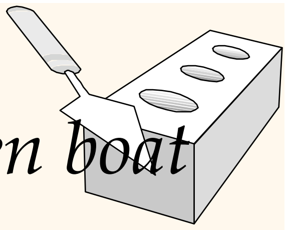


- UNION: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- If we replace OR by AND in the first version, what do we get?
- Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
  AND (B.color='red' OR 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'
UNION
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 and a green boat



- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- Included in the SQL/92 standard, but some systems don't support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

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

```
SELECT  S.sid        Key field!
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'
```

Exercise 5.2

Consider the following schema.

Suppliers(sid: integer, sname: string, address: string)

Parts(pid: integer, pname: string, color: string)

Catalog(sid: integer, pid: integer, cost: real)

The Catalog lists the prices charged for parts by Suppliers. Write the following queries in SQL:

1. Find the pnames of parts for which there is some supplier.
2. Find the sids of suppliers who supply a red part or a green part.
3. Find the sids of suppliers who supply a red part and a green part.

Nested Queries

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

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

- A powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- To find sailors who've *not* reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a <u>*nested loops*</u> evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

Nested Queries with Correlation

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

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

- EXISTS is another set comparison operator, like IN.
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

Exercise 5.2 ctd.

Consider the following schema.

Suppliers(sid: integer, sname: string, address: string)

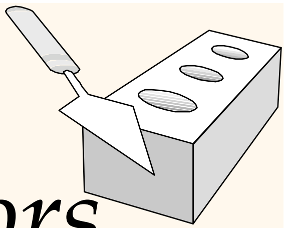
Parts(pid: integer, pname: string, color: string)

Catalog(sid: integer, pid: integer, cost: real)

The Catalog lists the prices charged for parts by Suppliers. Write the following queries in SQL. You can use NOT EXISTS.

1. Find the sids of suppliers who supply only red parts.
2. Find the snames of suppliers who supply every part. (difficult)

More on Set-Comparison Operators

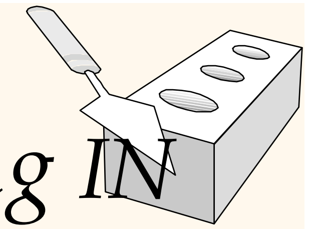


- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op* ANY, *op* ALL $>, <, =, \geq, \leq, \neq$
- Find sailors whose rating is greater than that of some sailor called Horatio:

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

Simple Examples for Any and All

❖ 1 = Any {1,3}	True
❖ 1 = All {1,3}	False
❖ 1 = Any {}	False
❖ 1 = All {}	True



Rewriting INTERSECT Queries Using IN

Find sid's of sailors who've reserved both a red and 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'
           AND S.sid IN  (SELECT  S2.sid
                          FROM  Sailors S2, Boats B2, Reserves R2
                          WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
                                 AND  B2.color='green')
```

- Similarly, EXCEPT queries re-written using NOT IN.
- To find *names* (not *sid*'s) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in SELECT clause.

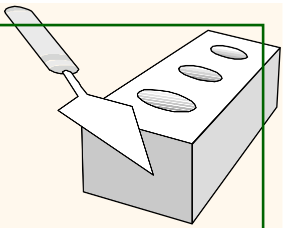
Division in SQL

Find sailors who've reserved all boats.

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

(1)

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



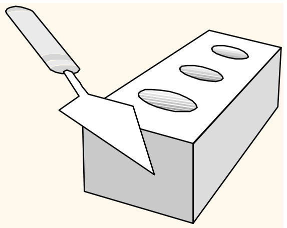
(2)

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

Sailors S such that ...

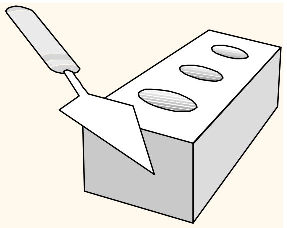
there is no boat B without ...

a Reserves tuple showing S reserved B



Summary: SQL Set Operators

- UNION, INTERSECT, EXCEPT behave like their relational algebra counterpart.
- *New Operator EXISTS tests if a relation is empty.*
- Can use ANY, ALL to compare a value against values in a set.

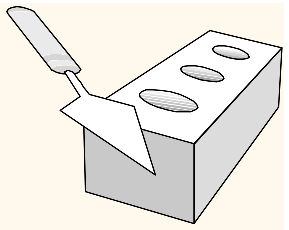


Follow-UP

- On "not equals".
- The SQL Standard operator ANSI is <>.
- Apparently many systems support != as well not equals discussion.
- We teach the standard but accept other common uses (unless explicitly ruled out).
- We'll start with group by and assertions, the come back to null values.

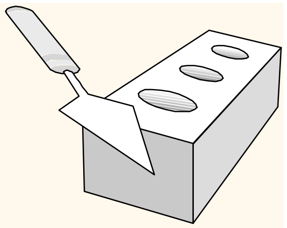
Null Values

Null Values



- Special attribute value *NULL* can be interpreted as
  - Value unknown (e.g., a rating has not yet been assigned),
  - Value inapplicable (e.g., no spouse's name),
  - Value withheld (e.g., the phone number).

- The presence of *NULL* complicates many issues:
  - Special operators needed to check if value is null.
  - Is rating>8 true or false when rating is equal to null?
  - What about AND, OR and NOT connectives?
  - Meaning of constructs must be defined carefully.
    - E.g., how to deal with tuples that evaluate neither to TRUE nor to FALSE in a selection?
- Mondial Example



Null Values

- *NULL* is not a constant that can be explicitly used as an argument of some expression.
- *NULL* values need to be taken into account when evaluating conditions in the *WHERE* clause.
- Rules for *NULL* values:
  - An arithmetic operator with (at least) one *NULL* argument always returns *NULL* .
  - The comparison of a *NULL* value to any second value returns a result of UNKNOWN.
- *A selection returns only those tuples that make the condition in the WHERE clause TRUE, those with UNKNOWN or FALSE result do not qualify.*

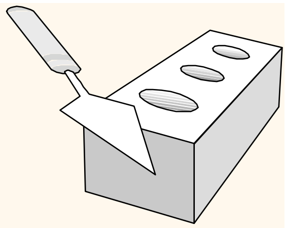
Truth Value Unknown

- Three-valued logic: TRUE, UNKNOWN, FALSE.
- Can think of TRUE = 1, UNKNOWN = ½, FALSE = 0
  - AND of two truth values: their minimum.
  - OR of two truth values: their maximum.
  - NOT of a truth value: 1 – the truth value.
- Examples:

  TRUE AND UNKNOWN = UNKNOWN
  FALSE AND UNKNOWN = FALSE

  FALSE OR UNKNOWN = UNKNOWN

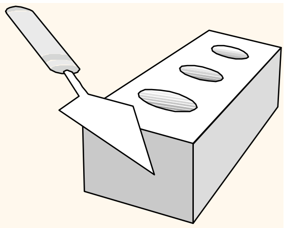
  NOT UNKNOWN = UNKNOWN



Truth Value Unknown

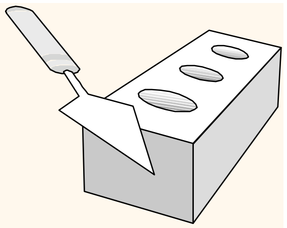
```
SELECT  *
FROM    Sailors
WHERE  rating < 5  OR  rating >= 5;
```

- What does this return?
- Does not return all sailors, but only those with non-NULL rating.



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

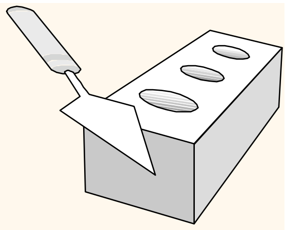
A SQL query walks into a bar and sees two tables. He walks up to them and says 'Can I join you?'

Cartesian Product

- Expressed in FROM clause.
- Forms the *Cartesian product* of all relations listed in the FROM clause, in the given order.

```
SELECT  *
FROM    Sailors, Reserves;
```

- So far, not very meaningful.



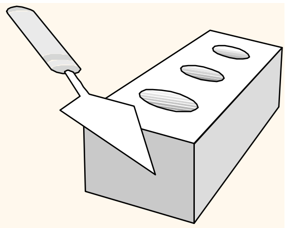
Join

- Expressed in FROM clause and WHERE clause.
- Forms the subset of the *Cartesian product* of all relations listed in the FROM clause that satisfies the WHERE condition:

```
SELECT  *
FROM    Sailors, Reserves
WHERE Sailors.sid = Reserves.sid;
```

- In case of ambiguity, prefix attribute names with relation name, using the dot-notation.

Join in SQL



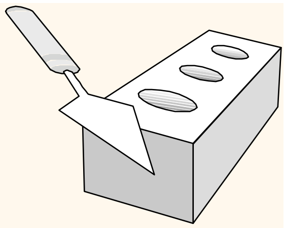
- Since joins are so common, SQL provides JOIN as a shorthand.

```
SELECT *
        FROM Sailors JOIN Reserves ON
                Sailors.sid = Reserves.sid;
```

- NATURAL JOIN produces the natural join of the two input tables, i.e. an equi-join on all attributes common to the input tables.

```
SELECT *
FROM Sailors NATURAL  JOIN Reserves;
```

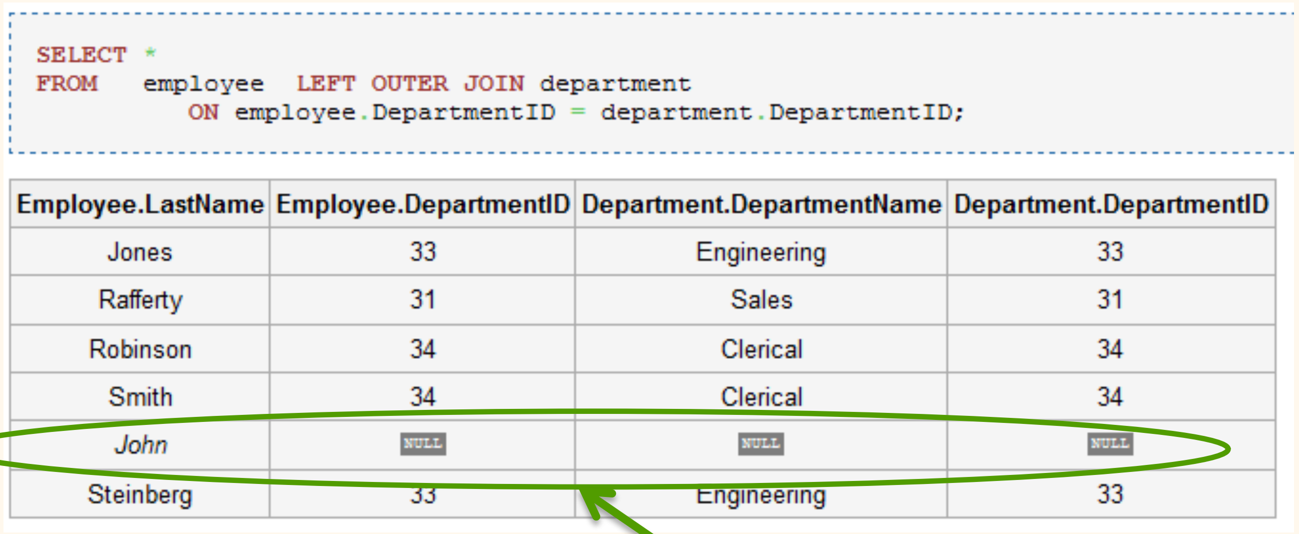
Outer Joins



- Typically, there are some *dangling* tuples in one of the input tables that have no matching tuple in the other table.
  - Dangling tuples are not contained in the output.
- Outer joins are join variants that do not lose any information from the input tables.

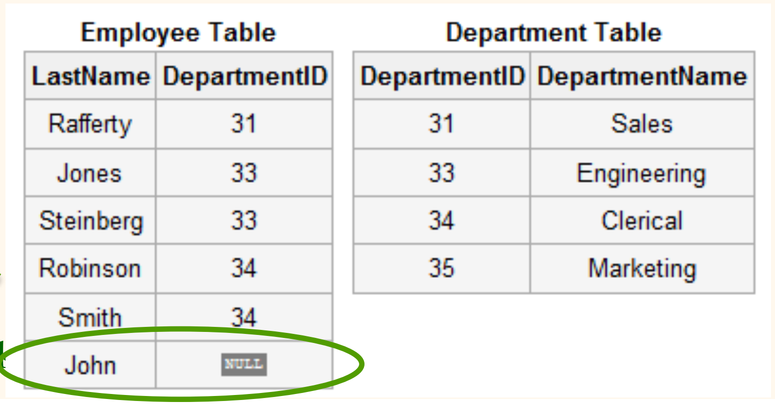
Left Outer Join

- includes all dangling tuples from the **left** input table
- *NULL* values filled in for all attributes of the



```
SELECT *
FROM   employee  LEFT OUTER JOIN department
          ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Jones	33	Engineering	33
Rafferty	31	Sales	31
Robinson	34	Clerical	34
Smith	34	Clerical	34
John	NULL	NULL	NULL
Steinberg	33	Engineering	33



Employee Table

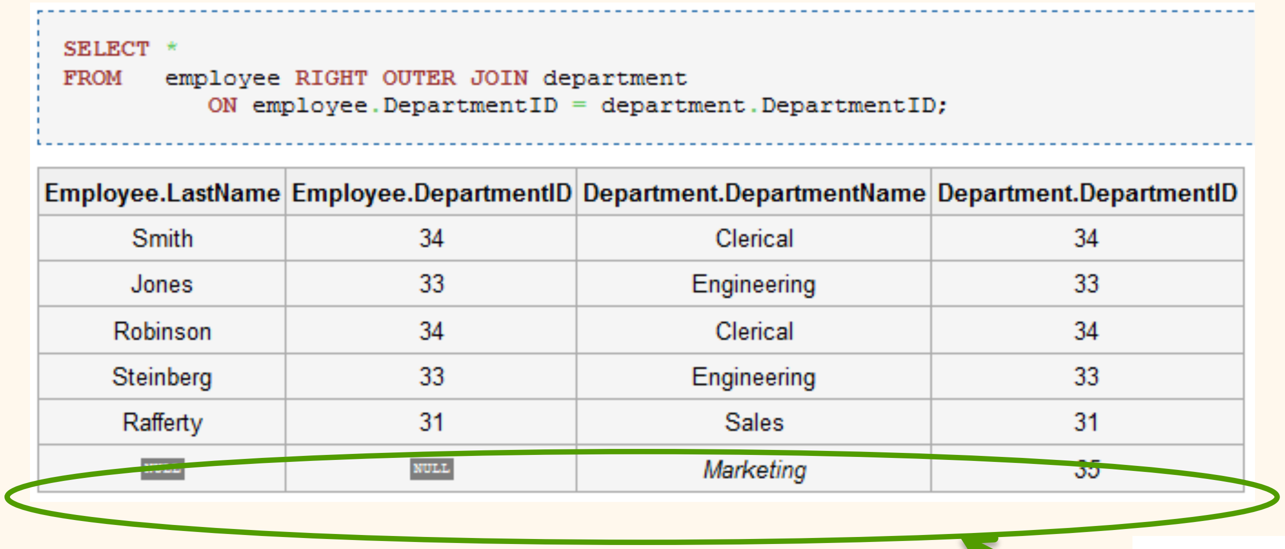
LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34
John	NULL

Department Table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

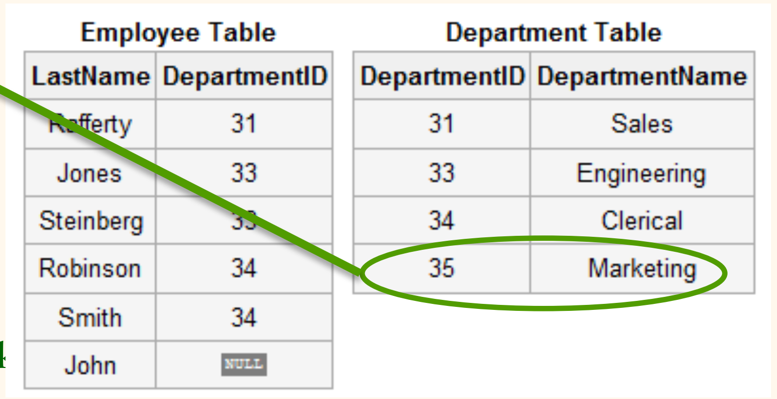
Right Outer Join

- includes all dangling tuples from the **right** input table
- *NULL* values filled in for all attributes of the right input table



```
SELECT *
FROM   employee RIGHT OUTER JOIN department
          ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Steinberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	35



Employee Table

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34
John	NULL

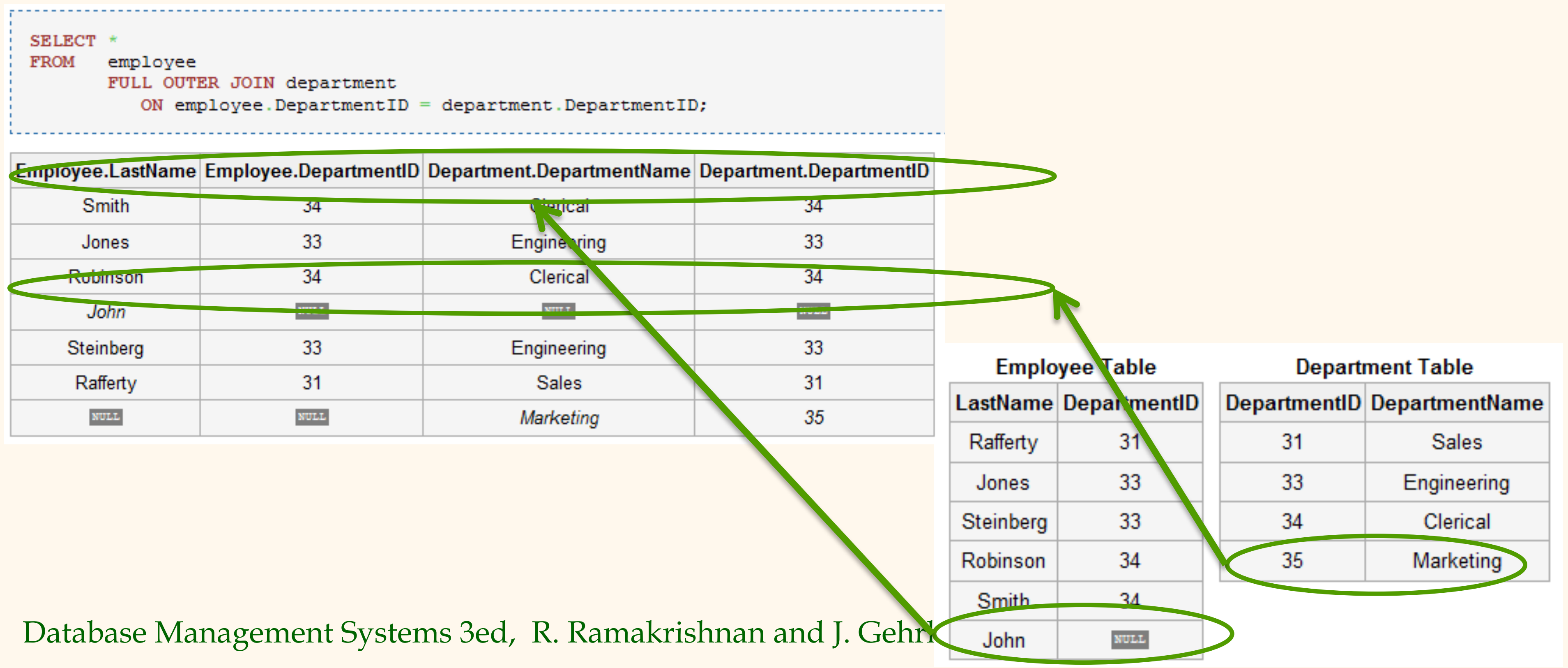
Department Table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

- What's the difference between LEFT and RIGHT joins?
- Can one replace the other?

Full Outer Join

- includes all dangling tuples from **both** input tables
- *NULL* values filled in for all attributes of any dangling tuples



```
SELECT *
FROM   employee
       FULL OUTER JOIN department
          ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
John	NULL	NULL	NULL
Steinberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	*35*

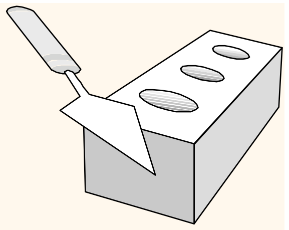
Employee Table

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34
John	NULL

Department Table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

Aggregation



Aggregate Operators

- Operates on tuple sets.
- 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  S.sname
FROM  Sailors S
WHERE  S.rating= (SELECT  MAX(S2.rating)
                  FROM  Sailors S2)
```

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

```
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 illegal! (We'll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

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

Exercise 5.2 ctd.

Consider the following schema.

Suppliers(sid: integer, sname: string, address: string)

Parts(pid: integer, pname: string, color: string)

Catalog(sid: integer, pid: integer, cost: real)

The Catalog lists the prices charged for parts by Suppliers. Write the following query in SQL:

1. Find the average cost of Part 70 (over all suppliers of Part 70).
2. Find the sids of suppliers who charge more for Part 70 than the average cost of Part 70.
3. Find the sids of suppliers who charge more for some part than the average cost of that part.

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 *and* HAVING

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

- 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 ≥ 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
```

- 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.)

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

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 ≥ 18, for each rating with at least 2 such sailors. Step 1.

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

Step 1: Apply Where clause.

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

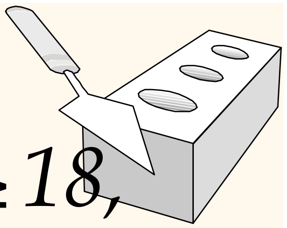
Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors. Step 2.

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

Step 2: keep only columns that appear in SELECT, GROUP BY, or HAVING

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

Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors. Step 3.



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

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

Step 3: sort tuples into groups.

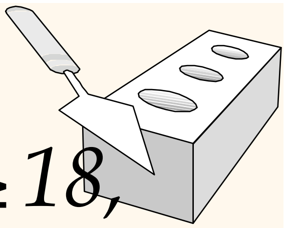
Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 <u>such</u> sailors. Step 4.

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

rating	age
7	45.0
7	35.0

Step 4: apply having clause to eliminate groups.

Find the age of the youngest sailor with age ≥18, for each rating with at least 2 <u>such</u> sailors. Step 5.



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

rating	age
7	35.0

Step 5: generate one answer tuple for each group.

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
```

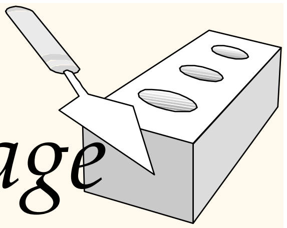
- Can we instead remove *B.color='red'* from the WHERE clause and add a HAVING clause with this condition?

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

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

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18.
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) >1

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



- Aggregate operations cannot be nested! WRONG:

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

- Correct solution (in SQL/92):

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

Exercise 5.2 ctd.

Consider the following schema.

Suppliers(sid: integer, sname: string, address: string)

Parts(pid: integer, pname: string, color: string)

Catalog(sid: integer, pid: integer, cost: real)

The Catalog lists the prices charged for parts by Suppliers. Write the following queries in SQL:

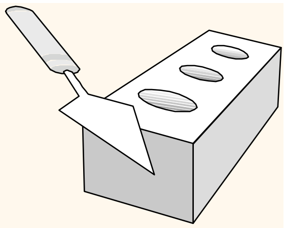
1. For every supplier that supplies only green parts, print the name of the supplier and the total number of parts that she supplies.
2. For every supplier that supplies a green part and a red part, print the name and price of the most expensive part that she supplies.

Group By

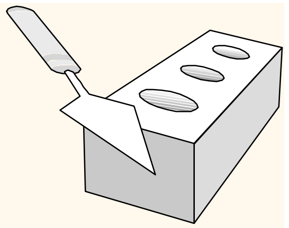
- In Assignment 2, the following question requires “group by”:
- “For each character and for each neutral planet, how much time total did the character spend on the planet”?

Integrity Constraints

Integrity Constraints

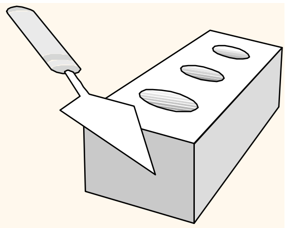


- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- *Types of IC's*: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  - *Domain constraints*: Field values must be of right type. Always enforced.



General Constraints

- Attribute-based CHECK
  - defined in the declaration of an attribute,
  - activated on insertion to the corresponding table or update of attribute.
- Tuple-based CHECK
  - defined in the declaration of a table,
  - activated on insertion to the corresponding table or update of tuple.
- Assertion
  - defined independently from any table,
  - activated on any modification of any table mentioned in the assertion.



Attribute-based CHECK

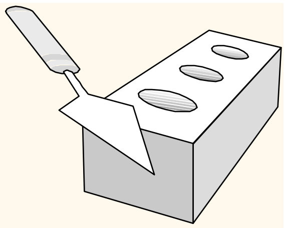
- Attribute-based CHECK constraint is part of an attribute definition.
- Is checked whenever a tuple gets a new value for that attribute (INSERT or UPDATE). Violating modifications are rejected.
- CHECK constraint can contain an SQL query referencing other attributes (of the same or other tables), if their relations are mentioned in the FROM clause.
- CHECK constraint is not activated if other attributes mentioned get new values.
- Most often used to check attribute values.

Attribute Check in SQL

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

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

```
CREATE TABLE  Reserves
        ( sname  CHAR(10),
        bid  INTEGER,
        day  DATE,
        PRIMARY KEY  (bid,day),
        CONSTRAINT  noInterlakeRes
        CHECK  (`Interlake' <>
                ( SELECT  B.bname
                FROM  Boats B
                WHERE  B.bid=bid)))
```



Tuple-based CHECK

- Tuple-based CHECK constraints can be used to constrain multiple attribute values within a table.
- Condition can be anything that can appear in a WHERE clause.
- Same activation and enforcement rules as for attribute-based CHECK.

```
CREATE TABLE   Sailors
   ( sid  INTEGER PRIMARY KEY,
   sname  CHAR(10),
   previousRating  INTEGER,
   currentRating  INTEGER,
   age  REAL,
   CHECK  (currentRating >= previousRating));
```

Tuple-based CHECK

- CHECK constraint that refers to other table:

```
CREATE TABLE  Reserves
   ( sname  CHAR(10),
   bid  INTEGER,
   day  DATE,
   PRIMARY KEY  (bid,day),
   CHECK  (‘Interlake’ <>
                    ( SELECT  B.bname
                    FROM  Boats B
                    WHERE  B.bid=bid)));
```

Interlake boats cannot be reserved

- But: these constraints are *invisible* to other tables, i.e. are not checked upon modification of other tables.

Constraints Over Multiple Relations

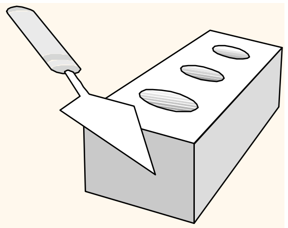
```
CREATE TABLE  Sailors
    ( sid  INTEGER,
    sname  CHAR(10),
    rating  INTEGER,
    age  REAL,
    PRIMARY KEY  (sid),
    CHECK
    ( (SELECT COUNT (S.sid) FROM Sailors S)
    + (SELECT COUNT (B.bid) FROM Boats B) < 100)
```

Number of boats plus number of sailors is < 100

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.

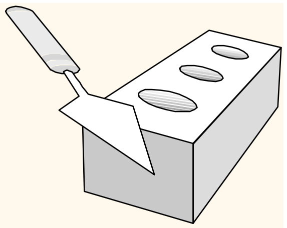
```
CREATE ASSERTION  smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT(B.bid) FROM Boats B) < 100)
```

Assertions



- Condition can be anything allowed in a WHERE clause.
- Constraint is tested whenever any (!) of the referenced tables is modified.
- Violating modifications are rejectced.
- CHECK constraints are more efficient to implement than ASSERTIONs.

Assertions



- *Number of boats plus number of sailors is < 100.*

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

- All relations are checked to comply with above.

- *Number of reservations per sailor is < 10.*

```
CREATE ASSERTION  notTooManyReservations
CHECK  ( 10 > ALL
          (SELECT COUNT (*)
          FROM Reserves
          GROUP BY sid));
```

Exercise 5.10

Consider the folllowing relational schema. An employee can work in more than one department; the pct_time field of the Works relation shows the percentage of time that a given employee works in a given department.

Emp(eid: integer, ename: string, age: integer, salary: real)

Works(eid: integer, did: integer, pct_time: integer)

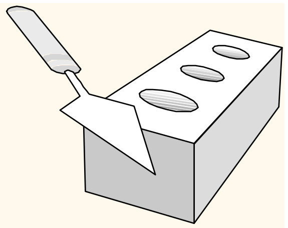
Dept(did: integer, budget: real, managerid: integer)

Write SQL integrity constraints (domain, key, foreign key or CHECK constraints or assertions) to ensure each of the following, independently.

1. Employees must make a minimum salary of $1000.
2. A manager must always have a higher salary than any employee that he or she manages.

Theory vs. Practice

- Unfortunately CHECK and ASSERTION are not well supported by SQL implementation.
- CHECK may not contain queries in SQL Server and other system.
  See http://consultingblogs.emc.com/davidportas/archive/2007/02/19/Trouble-with-CHECK-Constraints.aspx
- ASSERTION is not supported at all.
  http://www.sqlmonster.com/Uwe/Forum.aspx/sql-server-programming/8870/CREATE-ASSERTION-with-Microsoft-SQL-Server

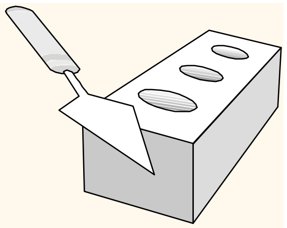


Triggers

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)
- Mainly related to transaction processing (Ch. 16, CMPT 454)

Triggers

- Synchronization of the Trigger with the activating statement (DB modification)
  - Before
  - After
  - Instead of
  - Deferred (at end of transaction).
- Number of Activations of the Trigger
  - Once per modified tuple (FOR EACH ROW)
  - Once per activating statement (default).

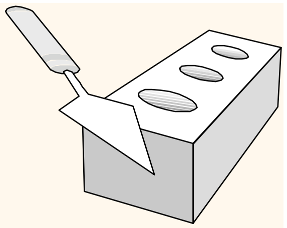


Triggers

```
CREATE TRIGGER youngSailorUpdate
   AFTER INSERT ON SAILORS                               /* Event */
   REFERENCING NEW TABLE NewSailors
   FOR EACH STATEMENT
       INSERT                                            /* Action */
             INTO YoungSailors(sid, name, age, rating)
             SELECT sid, name, age, rating
             FROM NewSailors N
             WHERE N.age <= 18;
```

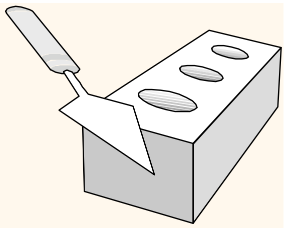
- This trigger inserts young sailors into a separate table.
- It has no (i.e., an empty, always true) condition.

Triggers: Example (SQL:1999)



```
CREATE TRIGGER youngSailorUpdate
   AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT
   INSERT
        INTO YoungSailors(sid, name, age, rating)
        SELECT sid, name, age, rating
        FROM NewSailors N
        WHERE N.age <= 18
```

Triggers



- Options for the REFERENCING clause:
  - NEW TABLE: the set (!) of tuples newly inserted (INSERT).
  - OLD TABLE: the set (!) of deleted or old versions of tuples (DELETE / UPDATE).
  - OLD ROW: the old version of the tuple (FOR EACH ROW UPDATE).
  - NEW ROW: the new version of the tuple (FOR EACH ROW UPDATE).
- The action of a trigger can consist of multiple SQL statements, surrounded by BEGIN . . . END.

Triggers

```
CREATE TRIGGER notTooManyReservations
  AFTER INSERT ON Reserves                              /* Event */
  REFERENCING NEW ROW NewReservation
  FOR EACH ROW
  WHEN (10 <= (SELECT COUNT(*) FROM Reserves
          WHERE sid =NewReservation.sid))               /* Condition */
     DELETE FROM Reserves R
     WHERE R.sid= NewReservation.sid                    /* Action */
          AND day=
        (SELECT MIN(day) FROM Reserves R2 WHERE R2.sid=R.sid);
```

- This trigger makes sure that a sailor has less than 10 reservations, deleting the oldest reservation of a given sailor, if neccesary.
- It has a non- empty condition (WHEN).

Trigger Syntax

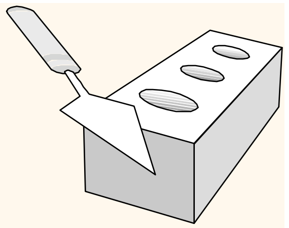
- Unfortunately trigger syntax varies widely among vendors.
- To make sure that no employee ID is negative:

SQL 99

```
CREATE TRIGGER checkrange
AFTER INSERT ON Employees
REFERENCING NEW TABLE NT
 WHEN
/* Condition */
(exists (Select *  FROM NT
Where NT.eid < 0))
/* Action */
ROLLBACK TRANSACTION
```

SQL SERVER

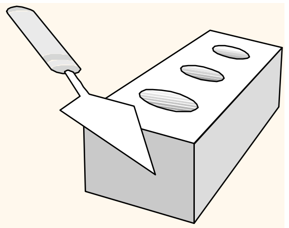
```
CREATE TRIGGER checkrange ON Emp
FOR INSERT
AS
IF
(exists (Select *  FROM inserted I
Where I.eid < 0))
BEGIN
 RAISERROR ('Employee ID out of range', 16, 1)
 ROLLBACK TRANSACTION
END
```



Triggers vs. General Constraints

- Triggers can be harder to understand.
  - Several triggers can be activated by one SQL statement (*arbitrary order*!).
  - A trigger may activate other triggers (*chain activation*).
- Triggers are procedural.
  - Assertions react on any database modification, trigger only only specified event.
  - Trigger execution cannot be optimized by DBMS.
- Triggers have more applications than constraints.
  - monitor integrity constraints,
  - construct a log,
  - gather database statistics, etc.

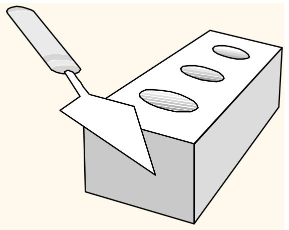
Summary



- SQL allows specification of rich integrity constraints (ICs): attribute-based, tuple-based CHECK and assertions (table-independent).
- CHECK constraints are activated only by modifications of the table they are based on, ASSERTIONs are activated by any modification that can possibly violate them.
- Choice of the most appropriate method for a particular IC is up to the DBA.
- Triggers respond to changes in the database. Can also be used to represent ICs.

Summary

- SQL was an important factor in the early acceptance of the relational model; more natural than earlier, procedural query languages.
- Relationally complete; in fact, significantly more expressive power than relational algebra.
- Even queries that can be expressed in RA can often be expressed more naturally in SQL.
- Many alternative ways to write a query; optimizer should look for most efficient evaluation plan.
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.



Summary (Contd.)

- NULL for unknown field values brings many complications
- SQL allows specification of rich integrity constraints
- Triggers respond to changes in the database