

Relational Algebra and SQL

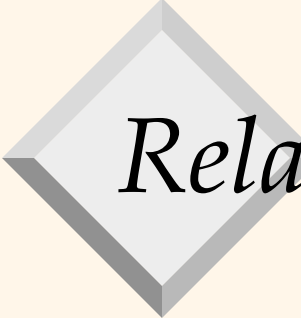
Johannes Gehrke

johannes@cs.cornell.edu

<http://www.cs.cornell.edu/johannes>

Slides from

*Database Management Systems, 3rd Edition,
Ramakrishnan and Gehrke.*



Relational Query Languages

- ◆ Query languages: Allow manipulation and **retrieval of data** from a database.
- ◆ Relational model supports simple, powerful QLS:
 - Strong formal foundation based on logic.
 - Allows for much optimization.
- ◆ Query Languages **!=** programming languages!
 - QLS not expected to be “Turing complete”.
 - QLS not intended to be used for complex calculations.
 - QLS support easy, efficient access to large data sets.



Formal Relational Query Languages

- ◆ Two mathematical Query Languages form the basis for “real” languages (e.g. SQL), and for implementation:
 - Relational Algebra: More **operational**, very useful for representing execution plans.
 - Relational Calculus: Lets users describe what they want, rather than how to compute it. (**Non-operational, declarative.**)

Preliminaries

- ◆ A query is applied to *relation instances*, and the result of a query is also a relation instance.
 - *Schemas of input* relations for a query are **fixed** (but query will run regardless of instance!)
 - The **schema for the result** of a given query is also **fixed!** Determined by definition of query language constructs.
- ◆ Positional vs. named-field notation:
 - Positional notation easier for formal definitions, named-field notation more readable.
 - Both used in SQL

Example Instances

- ◆ “Sailors” and “Reserves” relations for our examples.
- ◆ We’ll use positional or named field notation, assume that names of fields in query results are ‘inherited’ from names of fields in query input relations.

R1

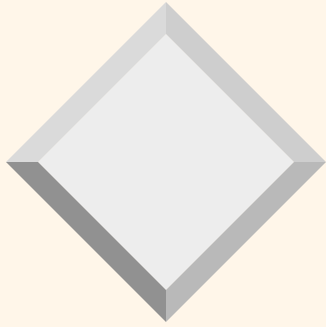
<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

S1

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

S2

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0



Relational Algebra

Relational Algebra

◆ Basic operations:

- Selection (σ) Selects a subset of rows from relation.
- Projection (π) Deletes unwanted columns from relation.
- Cross-product (\times) Allows us to combine two relations.
- Set-difference ($-$) Tuples in reln. 1, but not in reln. 2.
- Union (\cup) Tuples in reln. 1 and in reln. 2.

◆ Additional operations:

- Intersection, join, division, renaming: Not essential, but (very!) useful.

◆ Since each operation returns a relation, **operations can be composed!** (Algebra is “closed”.)

Projection

- ◆ Deletes attributes that are not in *projection list*.
- ◆ *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- ◆ Projection operator has to eliminate *duplicates!* (Why??)
 - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$\pi_{sname, rating}(S2)$

age
35.0
55.5

$\pi_{age}(S2)$

Selection

- ◆ Selects rows that satisfy *selection condition*.
- ◆ No duplicates in result! (Why?)
- ◆ *Schema* of result identical to schema of (only) input relation.
- ◆ *Result* relation can be the *input* for another relational algebra operation! (*Operator composition*.)

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$\sigma_{rating > 8}(S2)$$

sname	rating
yuppy	9
rusty	10

$$\pi_{sname, rating}(\sigma_{rating > 8}(S2))$$

Union, Intersection, Set-Difference

◆ All of these operations take two input relations, which must be union-compatible:

- Same number of fields.
- 'Corresponding' fields have the same type.

◆ What is the *schema* of result?

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

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

$S1 \cup S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

Cross-Product

- ◆ Each row of S1 is paired with each row of R1.
- ◆ *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
 - *Conflict*: Both S1 and R1 have a field called *sid*.

(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

- Renaming operator: $\rho (C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$

Joins

◆ Condition Join: $R \bowtie_c S = \sigma_c (R \times S)$

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

$S1 \bowtie_{S1.sid < R1.sid} R1$

- ◆ *Result schema* same as that of cross-product.
- ◆ Fewer tuples than cross-product, might be able to compute more efficiently
- ◆ Sometimes called a *theta-join*.

Joins

- ◆ Equi-Join: A special case of condition join where the condition c contains only *equalities*.

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

$$S1 \bowtie_{sid} R1$$

- ◆ Result schema similar to cross-product, but only one copy of fields for which equality is specified.
- ◆ Natural Join: Equijoin on *all* common fields.

Division

- ◆ Not supported as a primitive operator, but useful for expressing queries like:

Find sailors who have reserved all boats.

- ◆ Let A have 2 fields, x and y ; B have only field y :

- $A/B = \{ \langle x \rangle \mid \forall \langle y \rangle \in B \exists \langle x, y \rangle \in A \}$

- i.e., **A/B contains all x tuples (sailors) such that for every y tuple (boat) in B , there is an xy tuple in A .**

- Or: If the set of y values (boats) associated with an x value (sailor) in A contains all y values in B , the x value is in A/B .

- ◆ In general, x and y can be any lists of fields; y is the list of fields in B , and $x \cup y$ is the list of fields of A .

Examples of Division A/B

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

A

pno
p2

B1

sno
s1
s2
s3
s4

A/B1

pno
p2
p4

B2

sno
s1
s4

A/B2

pno
p1
p2
p4

B3

sno
s1


A/B3

Expressing A/B Using Basic Operators

- ◆ Division is not essential op; just a useful shorthand.
 - (Also true of joins, but joins are so common that systems implement joins specially.)
- ◆ *Idea*: For A/B , compute all x values that are not 'disqualified' by some y value in B .
 - x value is *disqualified* if by attaching y value from B , we obtain an xy tuple that is not in A .

Disqualified x values:

A/B :



Find names of sailors who've reserved boat #103


◆ **Solution 1:** $\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$

❖ **Solution 2:** $\rho(Temp1, \sigma_{bid=103} Reserves)$

$\rho(Temp2, Temp1 \bowtie Sailors)$

$\pi_{sname}(Temp2)$

❖ **Solution 3:** $\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$



Find names of sailors who've reserved a red boat


- ◆ Information about boat color only available in Boats; so need an extra join:

$$\pi_{sname}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$$

- ◆ A more efficient solution:

$$\pi_{sname}(\pi_{sid}((\pi_{bid} \sigma_{color='red'} Boats) \bowtie Res) \bowtie Sailors)$$

A query optimizer can find this, given the first solution!




Find sailors who've reserved a red or a green boat

- ◆ Can identify all red or green boats, then find sailors who've reserved one of these boats:

$$\rho (\text{Tempboats}, (\sigma_{color='red' \vee color='green'} \text{Boats}))$$
$$\pi_{sname}(\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors})$$

- ❖ Can also define Tempboats using union! (How?)
- ❖ What happens if \vee is replaced by \wedge in this query?



Find sailors who've reserved a red and a green boat

- ◆ Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for *Sailors*):

$$\rho (Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves))$$
$$\rho (Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie Reserves))$$
$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$



Find the names of sailors who've reserved all boats

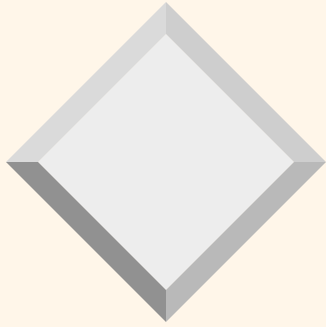
- ◆ Uses division; schemas of the input relations to / must be carefully chosen:

$$\rho (Tempkids, (\pi_{sid,bid} Reserves) / (\pi_{bid} Boats))$$

$$\pi_{sname} (Tempkids \bowtie Sailors)$$

- ◆ To find sailors who've reserved all 'Interlake' boats:

$$\dots / \pi_{bid} (\sigma_{bname='Interlake'} Boats)$$



SQL

Basic SQL Query

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

```
SELECT S.Name
FROM   Sailors S
WHERE  S.Age > 25
```

```
SELECT DISTINCT S.Name
FROM   Sailors S
WHERE  S.Age > 25
```

- Default is that duplicates are not eliminated!
 - Need to explicitly say “DISTINCT”

SQL Query

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

Sailors

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

Reserves

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



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 *condition*.
 - Delete attributes that are not in *target-list*
 - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably 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 Slightly Modified Query

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

- Would adding DISTINCT to this query make a difference?



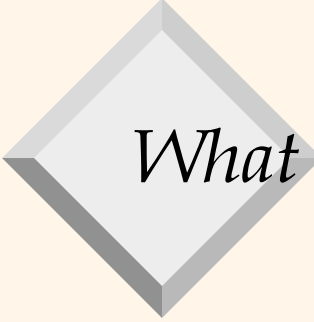
Find sid's of sailors who've reserved a red or a green boat

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



What does this query compute?

```
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'
```

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

Key field!

```
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'
```

- What if INTERSECT were replaced by EXCEPT?
 - EXCEPT is set difference



Expressions and Strings

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


- *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** is used to name fields in result.
- **LIKE** is used for string matching
 - **`_`** stands for any one character
 - **`%`** stands for 0 or more arbitrary characters.

Nested Queries (with Correlation)

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 S.sid=R.sid)
```

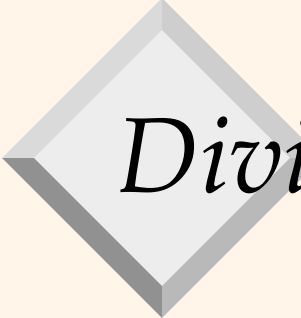




Nested Queries (with Correlation)

*Find names of sailors who have **not** reserved boat #103:*

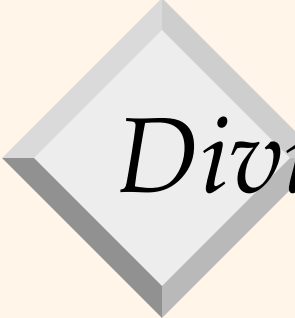
```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (SELECT *
                   FROM Reserves R
                   WHERE R.bid=103 AND S.sid=R.sid)
```



Division in SQL

Find sailors who've reserved all boats

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



Division in SQL (without Except!)

Find sailors who've reserved all boats.

```
SELECT S.sname  
FROM Sailors S
```

```
WHERE NOT EXISTS (SELECT B.bid  
                  FROM Boats B
```

Sailors S such that ...

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

there is no boat B without ...

a Reserves tuple showing S reserved B

More on Set-Comparison Operators

- *op* ANY, *op* ALL
 - *op* can be $>$, $<$, $=$, \geq , \leq , \neq
- Find sailors whose rating is greater than that of all sailors called Horatio:

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

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'
```



*Find name and age of the oldest sailor(s)
with rating > 7*

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.Rating > 7 AND
       S.age = (SELECT MAX (S2.age)
                FROM   Sailors S2
                WHERE  S2.Rating > 7)
```

Aggregate Operators

- ◆ 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.*
 - If 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
```



GROUP BY

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

Find the age of the youngest sailor for each rating level

```
SELECT    S.rating, MIN(S.Age)
FROM      Sailors S
GROUP BY  S.rating
```




Conceptual Evaluation Strategy

- Semantics of an SQL query defined as follows:
 - Compute the cross-product of *relation-list*
 - Discard resulting tuples if they fail *condition*.
 - Delete attributes that are not in *target-list*
 - Remaining tuples are partitioned into groups by the value of the attributes in *grouping-list*
 - One answer tuple is generated per group
- Note: Does not imply query will actually be evaluated this way!

*Find the age of the youngest sailor with age ≥ 18 ,
for each rating with at least one such sailor*

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

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	15.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

<u>sid</u>	sname	rating	age
29	brutus	1	33.0
22	dustin	7	45.0
64	horatio	7	35.0
58	rusty	10	35.0


rating	
1	33.0
7	35.0
10	35.0



Are These Queries Correct?


```
SELECT  MIN(S.Age)
FROM    Sailors S
GROUP BY S.rating
```

```
SELECT  S.name, S.rating, MIN(S.Age)
FROM    Sailors S
GROUP BY S.rating
```




What does this query compute?

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



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

```
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 (Temp2.avgage)
                    FROM (SELECT AVG(S.age) as avgage
                          FROM Sailors S
                          GROUP BY S.rating) AS Temp2
                    )
```



What does this query compute?

```
SELECT Temp.rating, Temp.minage
FROM   (SELECT S.rating, MIN (S.age) AS minage, COUNT(*) AS cnt
        FROM   Sailors S
        WHERE  S.age >= 18
        GROUP BY S.rating) AS Temp
WHERE  Temp.cnt >= 2
```

Queries With GROUP BY and HAVING

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

Find the age of the youngest sailor with age ≥ 18 for each rating level 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(*)  $\geq 2$ 
```



Conceptual Evaluation Strategy

- Semantics of an SQL query defined as follows:
 - Compute the cross-product of *relation-list*
 - Discard resulting tuples if they fail *condition*.
 - Delete attributes that are not in *target-list*
 - Remaining tuples are partitioned into groups by the value of the attributes in *grouping-list*
 - The *group-qualification* is applied to eliminate some groups
 - One answer tuple is generated per qualifying group
- Note: Does not imply query will actually be evaluated this way!

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
```

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

- ◆ 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.)

*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  $\geq$  18
GROUP BY S.rating
HAVING 1 < (SELECT COUNT (*)
            FROM Sailors S2
            WHERE S.rating=S2.rating)
```

Find the average age for each rating, and order results in ascending order on avg. age

```
SELECT S.rating, AVG (S.age) AS avgage  
FROM Sailors S  
GROUP BY S.rating  
ORDER BY avgage
```

- ❖ ORDER BY can only appear in top-most query
 - Otherwise results are unordered!



Null Values

- Field values in a tuple are sometimes *unknown*
 - e.g., a rating has not been assigned
- Field values are sometimes *inapplicable*
 - e.g., no spouse's name
- SQL provides a special value *null* for such situations.

Queries and Null Values

```
SELECT S.Name
FROM   Sailors S
WHERE  S.Age > 25
```

- What if S.Age is NULL?
 - S.Age > 25 returns NULL!
- Implies a predicate can return 3 values
 - True, false, NULL
 - Three valued logic!
- Where clause eliminates rows that do not return true (i.e., which are false or NULL)

Three-valued Logic

```
SELECT S.Name
FROM Sailors S
WHERE NOT(S.Age > 25) OR S.rating > 7
```

- What if one or both of S.age and S.rating are NULL?

NOT Truth Table

A	NOT(A)
True	False
False	True
NULL	NULL

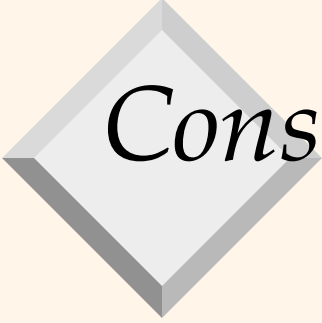
OR Truth Table

A/B	True	False	NULL
True	True	True	True
False	True	False	NULL
NULL	True	NULL	NULL

General Constraints

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

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



Constraints Over Multiple Relations

*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 )
```




Summary

- ◆ The relational model has rigorously defined query languages that are simple and powerful.
- ◆ Relational algebra is more operational; useful as internal representation for query evaluation plans.
- ◆ Several ways of expressing a given query; a query optimizer should choose the most efficient version.
- ◆ SQL is the lingua franca for accessing database systems today.