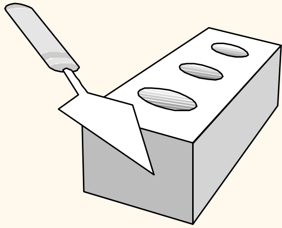
# *Relational Algebra*

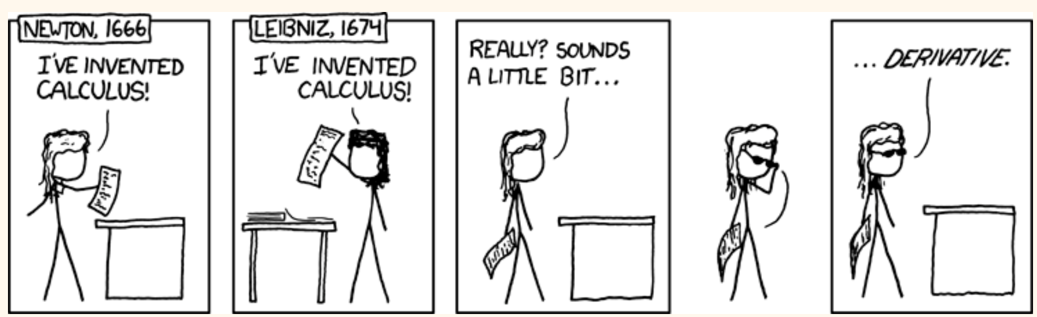


# *Relational Query Languages*

- *Query languages:* Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on algebra/logic.
  - Allows for much optimization.

# *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*.) Not covered in cours.



# *Motivation: Relational Algebra*

- Mathematically rigorous: the theory behind SQL.
- Relational algebra came first, SQL is an implementation.
- Under the hood: A query processing system translates SQL queries into relational algebra.
  - Supports optimization, efficient processing.

# *Overview*

- Notation
- Relational Algebra
- Relational Algebra basic operators.
- Relational Algebra derived operators.

# *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
  - The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.

# *Preliminaries*

- Positional vs. named-attribute notation:
  - Positional notation
    - Ex: Sailor(1,2,3,4)
    - easier for formal definitions
  - Named-attribute notation
    - Ex: Sailor(sid, sname, rating,age)
    - more readable
- Advantages/disadvantages of one over the other?

# *Example Instances*

***R1***

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

- "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.

***S1***

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

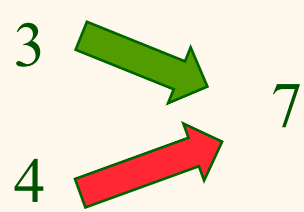
***S2***

| sid | 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*

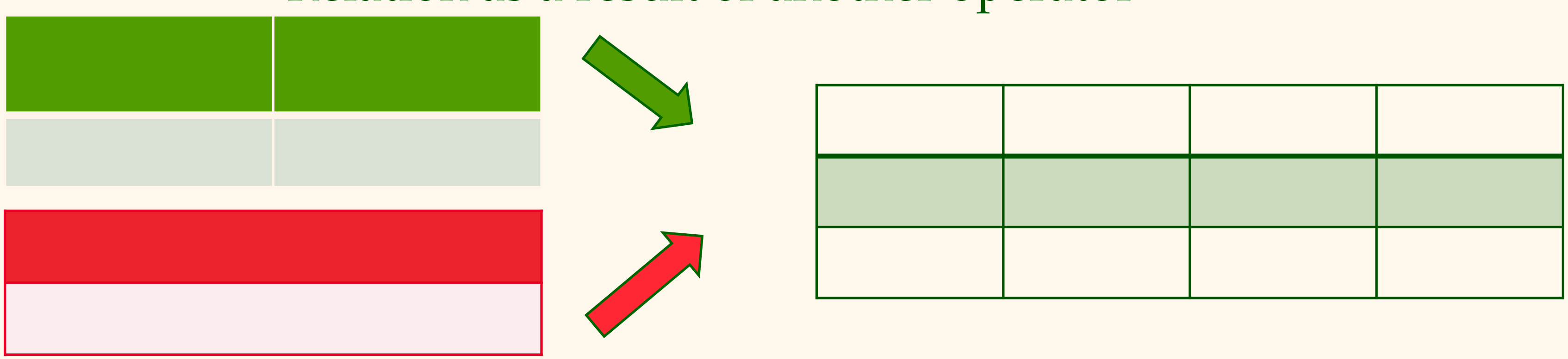
# *Algebra*

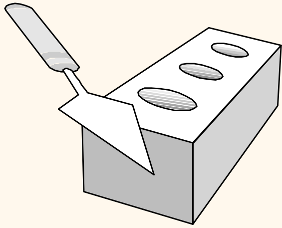
- In math, algebraic operations like +, -, x, /.
- Operate on numbers: input are numbers, output are numbers.
- Can also do Boolean algebra on sets, using union, intersect, difference.
- Focus on **algebraic identities**, e.g.
  - x (y+z) = xy + xz.
- (Relational algebra lies between propositional and 1st-order logic.)



# *Relational Algebra*

- Every operator takes <u>one or two</u> relation instances
- A relational algebra expression is recursively defined to be a relation
  - Result is also a relation
  - Can apply operator to
    - Relation from database
    - Relation as a result of another operator



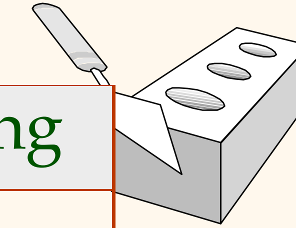


# *Relational Algebra Operations*

- Basic operations:
  - *Selection* ($\sigma$) Selects a subset of rows from relation.
  - *Projection* ($\pi$) Selects a subset of 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 derived operations:
  - Intersection, *join*, division, renaming.
    Not essential, but very useful.
- Since each operation returns a relation, operations can be *composed*!

# *Basic Relational Algebra Operations*

# *Projection*



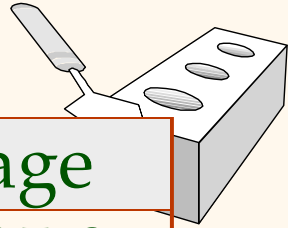
- Deletes attributes that are not in *projection list*.
- Like SELECT in SQL.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)

| sname | rating |
|---|---|
| yuppy | 9 |
| lubber | 8 |
| guppy | 5 |
| rusty | 10 |

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

| age |
|---|
| 35.0 |
| 55.5 |

$$\pi_{age}(S2)$$



# *Selection*

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

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

- Selects rows that satisfy *selection condition*.
- Like WHERE in SQL.
- No duplicates in result! (Why?)
- *Schema* of result identical to schema of (only) input relation.
- Selection conditions:
  - *simple conditions* comparing attribute values (variables) and / or constants or
  - *complex conditions* that combine simple conditions using logical connectives AND and OR.

| 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 |
| 31 | lubber | 8 | 55.5 |
| 58 | rusty | 10 | 35.0 |
| 44 | guppy | 5 | 35.0 |
| 28 | yuppy | 9 | 35.0 |

$S1 \cup S2$

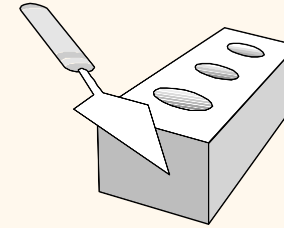
| sid | sname | rating | age |
|---|---|---|---|
| 22 | dustin | 7 | 45.0 |

$S1 - S2$

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

$S1 \cap S2$

# *Exercise on Union*



| Number | shape | holes |
|---|---|---|
| 1 | round | 2 |
| 2 | square | 4 |
| 3 | rectangle | 8 |

Blue blocks (BB)

| Number | shape | holes |
|---|---|---|
| 4 | round | 2 |
| 5 | square | 4 |
| 6 | rectangle | 8 |

Yellow blocks(YB)

Stacked(S)

| bottom | top |
|---|---|
| 4 | 2 |
| 4 | 6 |
| 6 | 2 |

1. Which tables are union-compatible?
2. What is the result of the possible unions?

# *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*: $\rho\,(C(1 \rightarrow sid1, 5 \rightarrow sid2),\ S1 \times R1)$

# *Exercise on Cross-Product*

| Number | shape | holes |
|---|---|---|
| 1 | round | 2 |
| 2 | square | 4 |
| 3 | rectangle | 8 |

| Number | shape | holes |
|---|---|---|
| 4 | round | 2 |
| 5 | square | 4 |
| 6 | rectangle | 8 |

Blue blocks (BB)

Stacked(S)

| bottom | top |
|---|---|
| 4 | 2 |
| 4 | 6 |
| 6 | 2 |

1. Write down 2 tuples in BB x S.
2. What is the cardinality of BB x S?

# *Derived Operators*
# *Join and Division*

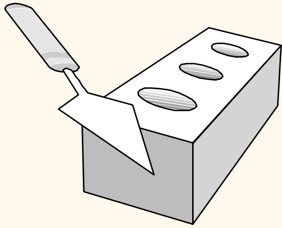
# Joins

- *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. How?
- Sometimes called a *theta-join*.
- $\Pi$ - $\sigma$ -x = SQL in a nutshell.



# *Exercise on Join*

| Number | shape | holes |
|---|---|---|
| 1 | round | 2 |
| 2 | square | 4 |
| 3 | rectangle | 8 |

Blue blocks (BB)

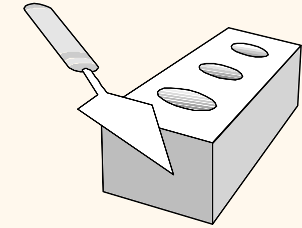
| Number | shape | holes |
|---|---|---|
| 4 | round | 2 |
| 5 | square | 4 |
| 6 | rectangle | 8 |

Yellow blocks(YB)

$$BB \bowtie_{BB.holes < YB.holes} YB$$

Write down 2 tuples in this 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 |

$$S1 \bowtie_{R.sid = S.sid} R1$$

- *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.
- *Natural Join*: Equijoin on *all* common fields. Without specified condition $A \bowtie B$ means the natural join of A and B.

# *Example for Natural Join*

| Number | shape | holes |
|---|---|---|
| 1 | round | 2 |
| 2 | square | 4 |
| 3 | rectangle | 8 |

Blue blocks (BB)

| shape | holes |
|---|---|
| round | 2 |
| square | 4 |
| rectangle | 8 |

Yellow blocks(YB)

What is the natural join of BB and YB?

# *Binary Choice Quiz*

- Consider two relations A and B that have exactly the same column headers.
- Is it true that $A \bowtie B = A \cap B$?

# *Join Examples*

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

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

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

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

  $\pi_{sname}(Temp2)$

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

# *Exercise: Find names of sailors who've reserved a red boat*

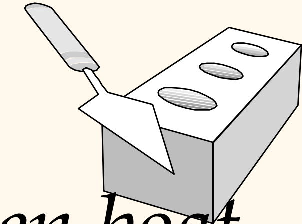
- Information about boat color only available in Boats; so need an extra join:

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

- 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 have reserved one of these boats:

$$\rho\ (Tempboats, (\sigma_{color='red' \vee color='green'}\ Boats))$$

$$\pi_{sname}(Tempboats \bowtie \mathrm{Re}serves \bowtie Sailors)$$

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

# *Exercise: Find sailors who've reserved a red <u>and</u> 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):

$$\rho\ (Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie \mathrm{Re}serves))$$

$$\rho\ (Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie \mathrm{Re}serves))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$

# *Division*

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

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

- Typical set-up: *A* has 2 fields *(x,y)* that are foreign key pointers, *B* has 1 matching field (*y*).
- Then *A/B* returns the set of *x*'s that match all *y* values in *B*.
- Example: *A* = Friend(x,y). *B* = set of 354 students. Then *A/B* returns the set of all x's that are friends with all 354 students.

# *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*

| pno |
|---|
| p2 |
| p4 |

*B2*

| pno |
|---|
| p1 |
| p2 |
| p4 |

*B3*

| sno |
|---|
| s1 |
| s2 |
| s3 |
| s4 |

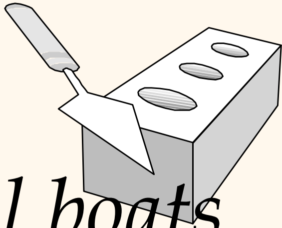
*A/B1*

| sno |
|---|
| s1 |
| s4 |

*A/B2*

| sno |
|---|
| s1 |

*A/B3*



# *Find the names of sailors who've reserved all boats*

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

$$\rho\ (Tempsids, (\pi_{sid,bid}\text{Re}serves) \,/\, (\pi_{bid}Boats))$$

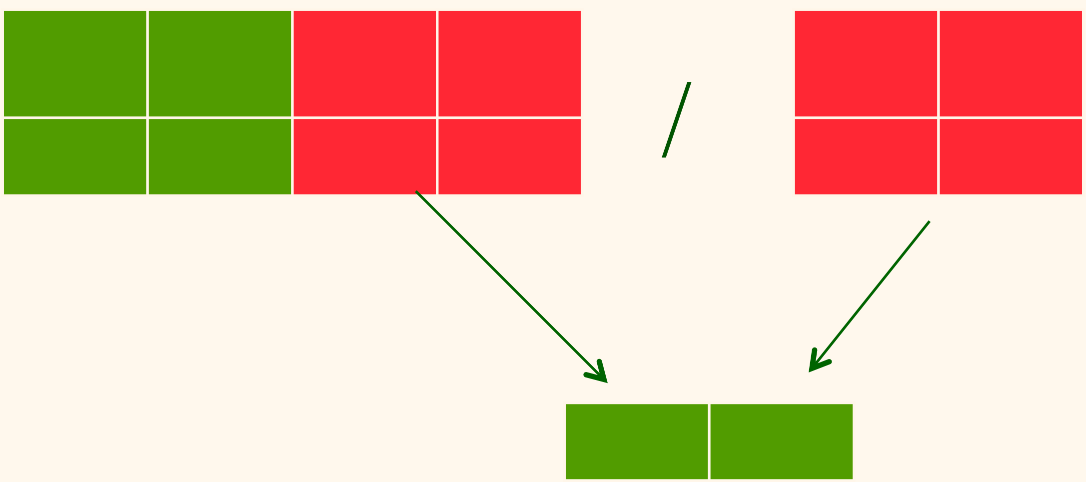
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

- To find sailors who have reserved all red boats:

$$\ldots\ldots \qquad / \ \pi_{bid}(\sigma_{color='red'}Boats)$$

# *Division in General*

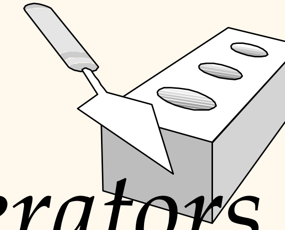
- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and *(x,y)* is the list of fields of *A*.
- Then *A/B* returns the set of all *x*-tuples such that for every *y*-tuple in B, the tuple *(x,y)* is in *A*.



# *Summary*

- The relational model supports 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.
- Book has lots of query examples.

# *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*.

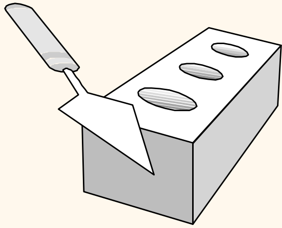
Disqualified *x* values: $\pi_x((\pi_x(A) \times B) - A)$

*A/B*: $\pi_x(A) -$ all disqualified tuples

# *Relational Calculus*

Chapter 4, Part B

# *Relational Calculus*



- Comes in two flavors: *Tuple relational calculus* (TRC) and *Domain relational calculus* (DRC).
- Calculus has *variables, constants, comparison ops, logical connectives* and *quantifiers*.
  - *TRC*: Variables range over (i.e., get bound to) *tuples*.
  - *DRC*: Variables range over *domain elements* (= field values).
  - Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called *formulas*. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to *true*.

# *Domain Relational Calculus*

- *Query* has the form:

$$\left\{ \left\langle x1, x2, \ldots, xn \right\rangle \mid p\left( \left\langle x1, x2, \ldots, xn \right\rangle \right) \right\}$$

- *Answer* includes all tuples $\left\langle x1, x2, \ldots, xn \right\rangle$ that make the *formula* $p\left( \left\langle x1, x2, \ldots, xn \right\rangle \right)$ be *true*.

- *Formula* is recursively defined, starting with simple *atomic formulas* (getting tuples from relations or making comparisons of values), and building bigger and better formulas using the *logical connectives*.

# DRC Formulas

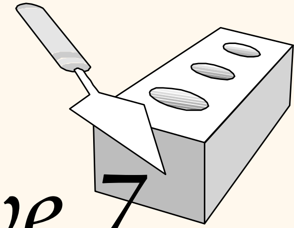
- *Atomic formula:*
  - $\langle x1, x2, ..., xn \rangle \in Rname$ , or X *op* Y, or X *op* constant
  - *op* is one of $<, >, =, \leq, \geq, \neq$
- *Formula:*
  - an atomic formula, or
  - $\neg p, p \wedge q, p \vee q$ , where p and q are formulas, or
  - $\exists X(p(X))$ , where variable X is *free* in p(X), or
  - $\forall X(p(X))$ , where variable X is *free* in p(X)
- The use of quantifiers $\exists X$ and $\forall X$ is said to *bind* X.
  - A variable that is not bound is free.

# *Free and Bound Variables*

- The use of quantifiers $\exists X$ and $\forall X$ in a formula is said to *bind* X.
  - A variable that is not bound is free.
- Let us revisit the definition of a query:

$$\left\{\left\langle x1,x2,\ldots,xn\right\rangle \mid p\left(\left\langle x1,x2,\ldots,xn\right\rangle\right)\right\}$$

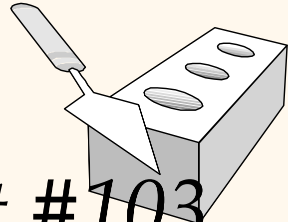
- There is an important restriction: the variables x1, ..., xn that appear to the left of `|' must be the *only* free variables in the formula p(...).



# *Find all sailors with a rating above 7*

$$\left\{ \langle I,N,T,A \rangle \mid \langle I,N,T,A \rangle \in Sailors \wedge T > 7 \right\}$$

- The condition $\langle I,N,T,A \rangle \in Sailors$ ensures that the domain variables *I, N, T* and *A* are bound to fields of the same Sailors tuple.
- The term $\langle I,N,T,A \rangle$ to the left of `|' (which should be read as *such that*) says that every tuple $\langle I,N,T,A \rangle$ that satisfies *T*>7 is in the answer.
- Modify this query to answer:
  - Find sailors who are older than 18 or have a rating under 9, and are called 'Joe'.



# *Find sailors rated > 7 who've reserved boat #103*

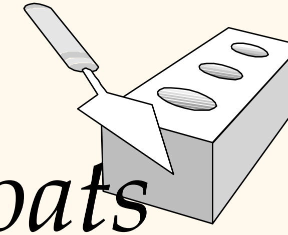
$$\left\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge T>7 \wedge \right.$$
$$\left. \exists\, Ir,Br,D\ \left(\langle Ir,Br,D\rangle \in \mathrm{Re}serves \wedge Ir=I \wedge Br=103\right)\right\}$$

- We have used $\exists\, Ir, Br, D\ (\ldots)$ as a shorthand for $\exists\, Ir\left(\exists\, Br\left(\exists\, D\,(\ldots)\right)\right)$
- Note the use of $\exists$ to find a tuple in Reserves that `joins with' the Sailors tuple under consideration.

## *Find sailors rated > 7 who've reserved a red boat*

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge T>7 \wedge$$
$$\exists\, Ir,Br,D\,(\langle Ir,Br,D\rangle \in \mathrm{Re}serves \wedge Ir=I \wedge$$
$$\exists\, B,BN,C\,(\langle B,BN,C\rangle \in Boats \wedge B=Br \wedge C='red'))\}$$

- Observe how the parentheses control the scope of each quantifier's binding.
- This may look cumbersome, but with a good user interface, it is very intuitive. (MS Access, QBE)

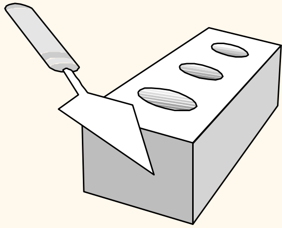


# *Find sailors who've reserved all boats*

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge$$
$$\forall\, B,BN,C \left( \neg \left(\langle B,BN,C\rangle \in Boats\right) \vee \right.$$
$$\left.\left( \exists\, Ir,Br,D \left(\langle Ir,Br,D\rangle \in \mathrm{Re}serves \wedge I = Ir \wedge Br = B\right)\right)\right)\}$$

- Find all sailors *I* such that for each 3-tuple $\langle B,BN,C\rangle$ either it is not a tuple in Boats or there is a tuple in Reserves showing that sailor *I* has reserved it.

# *Find sailors who've reserved all boats (again!)*



$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge$$
$$\forall \langle B,BN,C\rangle \in Boats$$
$$(\exists \langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))\}$$

- Simpler notation, same query. (Much clearer!)
- To find sailors who've reserved all red boats:

$$\ldots\ldots \quad (C \neq 'red' \vee \exists \langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))\}$$

# *Unsafe Queries, Expressive Power*

- It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called *unsafe*.
  - e.g.,

$$\left\{ S \mid \neg \left( S \in Sailors \right) \right\}$$

- It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC; the converse is also true.
- *Relational Completeness*: Query language (e.g., SQL) can express every query that is expressible in relational algebra/safe calculus.

# *Summary*

- Relational calculus is non-operational, and users define queries in terms of what they want, not in terms of how to compute it. (Declarativeness.)
- Algebra and safe calculus have same expressive power, leading to the notion of **relational completeness.**