

# *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 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(procedural), 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.  
“bid”= boats. “sid”: sailors
- ❖ 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

## ❖ Basic operations:

- Selection ( $\sigma$ ) Selects a subset of rows from relation.
- Projection ( $\pi$ ) 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 ( $\sqcup$ ) 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?!, what are the consequences?)
  - 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*.
- ❖ *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 |
| 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 |

$S1 \cap S2$

# 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. Filters tuples not satisfying the join condition.
- ❖ 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 |

- ❖ Result schema  $\pi_{sid, \dots, age, bid, \dots} (S1 \bowtie_{sid} R1)$  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.*
- ❖ Precondition: in  $A/B$ , the attributes in  $B$  must be included in the schema for  $A$ . Also, the result has attributes  $A-B$ .
  - $\text{SALES}(\text{supId}, \text{prodId})$ ;
  - $\text{PRODUCTS}(\text{prodId})$ ;
  - Relations  $\text{SALES}$  and  $\text{PRODUCTS}$  must be built using projections.
  - $\text{SALES}/\text{PRODUCTS}$ : the ids of the suppliers supplying ALL products.

# *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. Division is NOT implemented in SQL).
- ❖ *Idea:* For *SALES/PRODUCTS*, compute all products such that there exists at least one supplier not supplying it.
  - $x$  value is *disqualified* if by attaching  $y$  value from  $B$ , we obtain an  $xy$  tuple that is not in  $A$ .

$$A = \pi_{sid}((\pi_{sid}(Sales) \times Products) - Sales)$$

The answer is  $\pi_{sid}(Sales) - A$

*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):

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

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

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

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

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

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

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

# *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.