

# *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**, very useful for representing execution plans.
- *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (**Non-operational**, *declarative*.)
- *Understanding Algebra & Calculus is key to*
- *understanding SQL, query processing!*



# 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 Relational Algebra and SQL

# Example Instances

*R1*

| <u>sid</u> | <u>bid</u> | <u>day</u> |
|------------|------------|------------|
| 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*

| <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 ( $\cup$ ) 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*.)

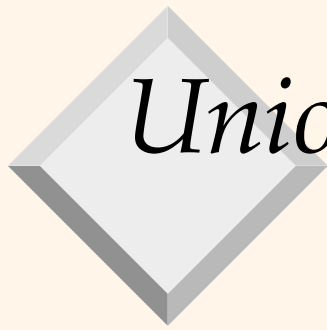
| sid | sname | rating | age  |
|-----|-------|--------|------|
| 28  | yuppy | 9      | 35.0 |
| 58  | rusty | 10     | 35.0 |

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

| 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 |
| 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, 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* and  $\wedge$ .

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

$$S1 \bowtie_{\text{sid}} R1$$

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



*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:

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

$\pi_{sname}(\textit{Tempboats} \bowtie \textit{Reserves} \bowtie \textit{Sailors})$

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

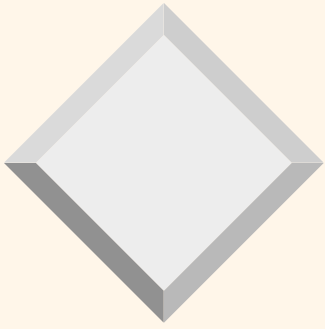


*Find sailors who've reserved a red and a green boat*

- 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*):

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



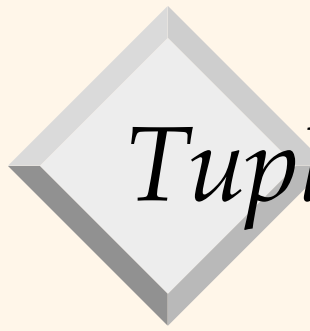


# *Relational Calculus*



# 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*.



# *Tuple Relational Calculus*

- *Query* has the form:  $\{ T \mid p(T) \}$
- *Answer* includes all tuples  $T$  that make the *formula*  $p(T)$  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*.



# TRC Formulas

## □ *Atomic formula:*

- $R \in \text{Rel}$ , or  $R.a$  *op*  $S.b$ , or  $R.a$  *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  $p(X)$



# Free and Bound Variables

- The use of **quantifiers**  $\forall X$  and  $\exists X$  in a formula is said to **bind**  $X$ .
  - A variable that is **not bound** is **free**.
- Let us revisit the definition of a **query**:  $\{T \mid p(T)\}$
- There is an important restriction: the variable **T** that appears to the left of  $\mid$  must be the **only** free variable in the formula  $p(\dots)$ .



*Find all sailors with a rating above 7*

- $\{S \mid S \in \text{Sailors} \wedge S.\text{rating} > 7\}$
- Query is evaluated on an instance of **Sailors**
- Tuple variable  $S$  is instantiated to each tuple of this instance in turn, and the condition “ **$S.\text{rating} > 7$** ” is applied to each such tuple.
- Answer contains all **instances of  $S$**  (which are tuples of Sailors) satisfying the condition.



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

- $\{S \mid (S \in \text{Sailors}) \wedge (S.\text{rating} > 7) \wedge (\exists R \in \text{Reserves} (R.\text{sid} = S.\text{sid} \wedge R.\text{bid} = 103))\}$
- Note the use of  $\exists$  to find a tuple in Reserves that 'joins with' the Sailors tuple under consideration.
- R is **bound**, S is not



# 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.,  $\{S \mid \neg (S \in Sailors)\}$
- It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC; the converse is also true.
- Relational Completeness: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.





## 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.
- Relational calculus is non-operational, and users define queries in terms of what they want, not in terms of how to compute it. (*Declarativeness.*)
- Several ways of expressing a given query; a *query optimizer* should choose the most efficient version.
- Algebra and safe calculus have same *expressive power*, leading to the notion of *relational completeness*.



# *Nested Relations*

- Attributes can be scalar (as before) or relation-valued
- Definition is recursive
- Example:  
create table Book (title: string, author:string,  
                  copies: (publ: string,  
                          pages: integer,  
                          date: integer))
- “copies” is a relation-valued field



# *Nested Relational Algebra*

- A spectrum of algebras can be defined
- At one end:
  - Relational algebra plus nest ( $\nu$ ) and unnest ( $\mu$ ):  
If  $B =$  \_\_\_\_\_

| title     | author   | copies        |       |      |
|-----------|----------|---------------|-------|------|
|           |          | publ          | pages | date |
|           |          | Prentice Hall | 613   | 1971 |
|           |          | McGraw Hill   | 542   | 1942 |
| Moby Dick | Melville |               |       |      |
| Marmion   | Scott    | { }           |       |      |

# Nesting and Unnesting

□ ... then  $\mu (B, \text{copies}) =$

| title     | author   | publ          | pages       | date        |
|-----------|----------|---------------|-------------|-------------|
| Moby Dick | Melville | Prentice Hall | 613         | 1971        |
| Moby Dick | Melville | McGraw Hill   | 542         | 1942        |
| Marmion   | Scott    | <i>null</i>   | <i>null</i> | <i>null</i> |

- Nulls must be supported *in algebra*
- $v (\mu (B, \text{copies}), \text{copies} (\text{publ}, \text{pages}, \text{date})) = B$
- $v, \mu$  inverses iff  $S \rightarrow N$ 
  - $S$  is set of scalar fields
  - $N$  is set of non-scalar fields
  - This is called PNF (partitioned normal form)



# *Extending Relational Operators*

- At other end of spectrum:
  - *Selection* allows set comparators and constants and use of select, project inside the formula
  - *Projection* allows arbitrary NF2 algebra expression in addition to simple field names
  - *Union, difference*: recursive definitions
  - *Cross product*: usually just relational.
- Example: retrieve title, number of pages of all books by Melville:
  - $\pi[\text{title}, \pi[\text{pages}](\text{copies})](\sigma[\text{author}=\text{'Melville'}](B))$



# *Nested Relations Summary*

- An early step on the way to OODBMS
- No products, only prototypes, but:
  - Many ideas from NF2 relations have survived
  - Collection types in SQL3 (nesting, unnesting)
  - Algebra ideas useful for Object Database QP
- Can provide a more natural model of data
- Are a straightforward extension of relations:
  - many solutions are thus also straightforward
  - formal foundation of relational model remains