## Relational Algebra

Module 3, Lecture 1

## 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. (Nonoperational, 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 SQL

#### R1 Example Instances

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

- "Sailors" and "Reserves" relations for 51 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.

<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* (  $\phi$  ) Selects a subset of rows from relation.
  - Projection ([]) Deletes unwanted columns from relation.
  - Cross-product (  $\times$  ) Allows us to combine two relations.
  - Set-difference ( ) Tuples in R1, but not in R2.
  - *Union* ( $\cup$ ) Tuples in R1 and in R2.
- 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 attributes 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

#### S2[sname,rating]

age			
35.0			
55.5			

S2[age]

#### 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

S2(rating > 8)

sname	rating
yuppy	9
rusty	10

(S2(rating > 8 )) [sname,rating]

### Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be union-compatible:
  - Same number of attributes
  - Corresponding' attributes 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

 $\square$  Renaming operator:  $\rho(C(1 \rightarrow sid\ 1, 5 \rightarrow sid\ 2), S1xR1)$ 

## Joins

 $\square$  Condition Join:  $R [\varphi] S =_{def} (R \times S) (\varphi)$ 

(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 [S1.sid < R1.sid] R1

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

## Joins

 $\square$  **Equi-Join**: A special case of condition join where the condition  $\varphi$  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

#### R[sid]S

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

R \* S

#### 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:
  - $-A + B = A[x] ((A[x] \times B) A)[x]$
  - i.e.,  $A \div B$  contains all x tuples (sailors) such that for <u>every</u> 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  $A \div B$ .
- In general, x,y can be any lists of attributes; y from B, and  $x \cup y$  from A.

## Examples of Division A + B

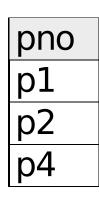
sno	pno
s1	p1
s1	p2
s1	р3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

pno	
p110	
μΖ	

**B1** 

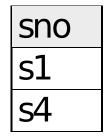
pno	
p2	
p4	

**B2** 

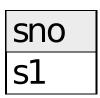


**B3** 

A + B1



A + B2



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.)
- Indea: For  $A \div B$ , 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[x] \times B) - A)[x]

A + B: A[x] - all disqualified tuples
```

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

Solution 1: ((Reserves(bid=103) \* Sailors)
[sname]

Solution 2: ρ(Temp1, Reserves(bid=103))ρ(Temp2, Temp1 \* Sailors)Temp2[sname]

Solution 3: (Reserves\* Sailors)(bid=103)
[sname]

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

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

(Boats(color='red')\*Reserves\* Sailors)[sname]

A more efficient solution:

((Boats(color='red')[bid]\*Reserves)[sid]\* Sailors) [sname]

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:

```
ρ (Tempboats, Boats(color='red' OR color='green'))
```

(Tempboats \* Reserves \* Sailors)[sname]

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 <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): ρ (Tempred, (Boats(color='red')\*Reserves)[sid] ρ (Tempgreen, (Boats(color='green')\*Reserves) [sid]

((Tempgreen ∩ Tempred)\*Sailors)[sname]

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

- Uses division; schemas of the input relations to ÷ must be carefully chosen:
  - ρ (Tempsids, Reserves[sid, bid] + Boats[bid])

(Tempsids \* Sailors)[sname]

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

... + Boats(bname='Interlake')[bid]