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".)

Slide No:L6-4

Basic operations:

– *Selection* ( ) Selects a subset of rows from relation.

– *Projection* ( ) Deletes unwanted columns from relation.

– *Cross-product* ( ) Allows us to combine two relations.

– *Set-difference* ( ) Tuples in reln. 1, but not in reln. 2.

– *Union* ( ) 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

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

Slide No:L6-5

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?)

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

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

Slide No:L6-6

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.*)

Set Operations:

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?

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

$S1 \cap S2$

Slide No:L6-7

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*.

<u>*Condition Join*</u>:

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*.

<u>*Equi-Join*</u>: A special case of condition join where the condition *c* contains only ***equalities.***

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* = $\{\langle x\rangle \mid \exists \langle x, y\rangle \in A \ \forall \langle y\rangle \in B\}$
  - 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* *y* is the list of fields of *A*.

Slide No:L6-11

Examples of Division A/B

A

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

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

Slide No:L6-12

Find names of sailors who’ve reserved boat #103

Solution 1:

Find names of sailors who’ve reserved a red boat

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

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:

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):

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:

TRC – a declarative query language

TRC Formulas

Atomic expressions are the following:

r (t) -- true if t is a tuple in the relation instance r

t1. Ai t2 .Aj compOp is one of {, ≥, =, ≠ }

t.Ai c c is a constant of appropriate type

Composite expressions:

Any atomic expression
F1 ∧ F2 ,, F1 ∨ F2 , ¬ F1 where F1 and F2 are expressio ns

(∀t) (F), (∃t) (F) where F is an expression and t is a tuple variable Free Variables

Bound Variables – quantified variables

Obtain the rollNo, name of all girl students in the Maths Dept

{s.rollNo,s.name | student(s) ^ s.sex=‘F’ ^ (∃ d)(department(d) ^ d.name=‘Maths’ ^ d.deptId = s.deptNo)}

s: free tuple variable

d: existentially bound tuple variable

Determine the departments that do not have any girl students

student (rollNo, name, degree, year, sex, deptNo, advisor) department (deptId, name, hod, phone)

{d.name| department(d) ^ ¬(∃ s)(student(s) ^ s.sex =‘F’ ^ s.deptNo = d.deptId)

Obtain the names of courses enrolled by student named Mahesh

{c.name | course(c) ^ (∃s) (∃e) (student(s) ^ enrollment(e) ^ s.name = “Mahesh” ^ s.rollNo = e.rollNo ^ c.courseId = e.courseId }

Get the names of students who have scored ‘S’ in all subjects they have enrolled. Assume that every student is enrolled in at least one course.

{s.name | student(s) ^ (∀e)((enrollment(e) ^ e.rollNo = s.rollNo) → e.grade =‘S’) }

Get the names of students who have taken at least one course taught by their advisor

{s.name | student(s) ^ (∃e)(∃t)(enrollment(e) ^ teaching(t) ^ e.courseId = t.courseId ^ e.rollNo = s.rollNo ^ t.empId = s.advisor}

Domain Relational Calculus:

Query has the form:

DRC Formulas

Atomic formula:

– , or X *op* Y, or X *op* constant

– *op* is one of

Formula:

– an atomic formula, or

– , where p and q are formulas, or

– , where variable X is *free* in p(X), or

– , where variable X is *free* in p(X)

• The use of quantifiers and is said to ***bind*** X.

– A variable that is not bound is free.

Free and Bound Variables

• The use of quantifiers and in a formula is said to ***bind*** X.

– A variable that is not bound is free.

Let us revisit the definition of a query:

Find all sailors with a rating above 7

The condition ensures that the domain variables *I*, *N*, *T* and *A* are bound to fields of the same Sailors tuple.

• The term to the left of `|' (which should be read as *such that*) says that every tuple 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 have reserved boat #103

We have used as a shorthand for

Note the use of to find a tuple in Reserves that `joins with' the Sailors tuple under consideration.

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

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

- Find all sailors *I* such that for each 3-tuple 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!) Simpler notation, same query.

(Much clearer!) To find sailors who've

reserved all red boats:

Expressive Power of Algebra and Calculus

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

– e.g.,

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.