



Conceptual Query Languages

Relational Algebra

Basic Operations

Versions

initial: *2020 February 13*

last modified: *2020 September 30*

Acknowledgments

Thanks to

- *Jeffrey D. Ullman*
for initial slidedeck
- *Jarek Szlichta*
for the slidedeck with significant refinements on which this is derived
- *Wenxiao Fu*
for refinements

Printable version of talk

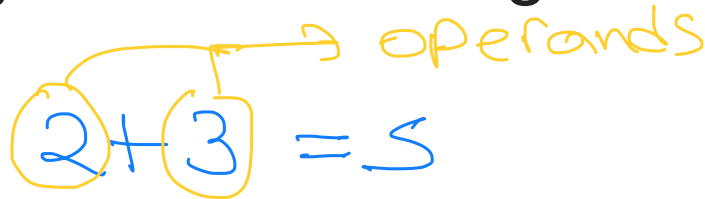
- Follow this link:
[*Relational Algebra* \[to pdf\]](#).
- Then *print* to get a **PDF**.

*(This only works correctly in **Chrome** or **Chromium**.)*

What is an *algebra*?

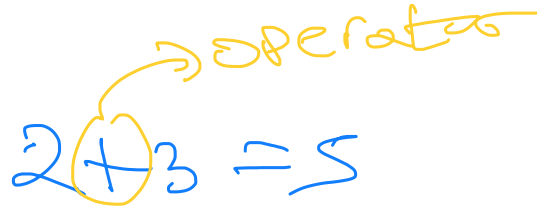
A mathematical system consisting of

- **operands**


$$\textcircled{2} + \textcircled{3} = 5$$

variables or values from which new values can be constructed; and

- **operators**


$$2 + \textcircled{+} 3 = 5$$

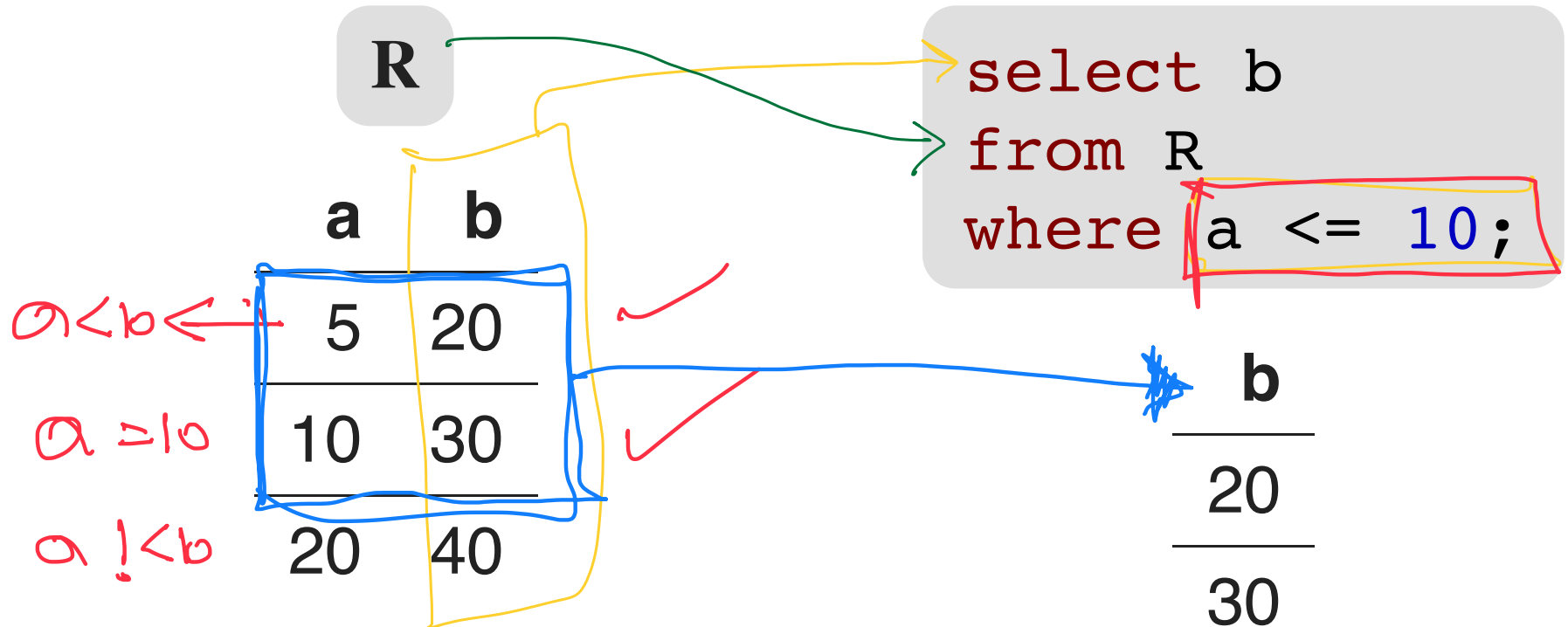
symbols denoting procedures that construct new values from given values.

What is the *relational algebra*?

- An algebra whose operands are *relations* (*tables*).
- Its operators are designed to do the most basic things over relations that we need in order to “query” over a database.

The result is an algebra that can be used as a *query language* over relations.

The SQL Language





The Core Relational Algebra

1. **Union** (“ \cup ”), **Intersection** (“ \cap ”), & **Difference** (“ $-$ ”):
 - the usual set operations;
 - but *both operands must have a matching schema*.
2. **Selection** (“ σ ”):
 - choosing (*selecting*) certain *rows*.
3. **Projection** (“ π ”):
 - choosing (*projecting*) certain *columns*.
4. **Product** (“ \times ”) & **Join** (“ \bowtie ”):
 - compositions of relations.
5. **Rename** (“ ρ ”):
 - renaming of relations and attributes.

Selection

$$\mathbf{R}_1 := \sigma_C(\mathbf{R}_2)$$

 *condition* (like if statement)

- *C* is a *condition* — as in “if” statements — written over the attributes of \mathbf{R}_2 that evaluates to *true* or *false* per tuple.
- \mathbf{R}_1 is the set of all the tuples of \mathbf{R}_2 that satisfy *C*; that is, those tuples from \mathbf{R}_2 for which *C* evaluates *true*.

Example of selection

Sells

pub	beer	price
Joe's	Bud	2.50
Joe's	Molsen	2.75
Fox	Bud	2.50
Fox	Molsen	3.50

$\sigma_{\text{pub}=\text{"Joe's"}}(\text{Sells})$

pub	beer	price
Joe's	Bud	2.50
Joe's	Molsen	2.75

Projection

$$\mathbf{R}_1 := \pi_L(\mathbf{R}_2)$$

 List of attributes from the schema

- L is a list of attr's from the schema of \mathbf{R}_2 .
- \mathbf{R}_1 is constructed by
 - taking *each* tuple from \mathbf{R}_2 ,
 - extracting the attr's from the tuple in list L , and
 - creating from those components a tuple for \mathbf{R}_1 .
- Eliminate duplicate tuples in \mathbf{R}_1 , if any.

Example of prjection

Sells

pub

beer

price

Joe's

Bud

2.50

Joe's

Molsen

2.75

Fox

Bud

2.50

Fox

Molsen

3.50

duplicates

$\pi_{\text{beer,price}}(\text{Sells})$

beer

price

Bud

2.50

Molsen

2.75

Molsen

3.50

Project

list of attr

Extended projection

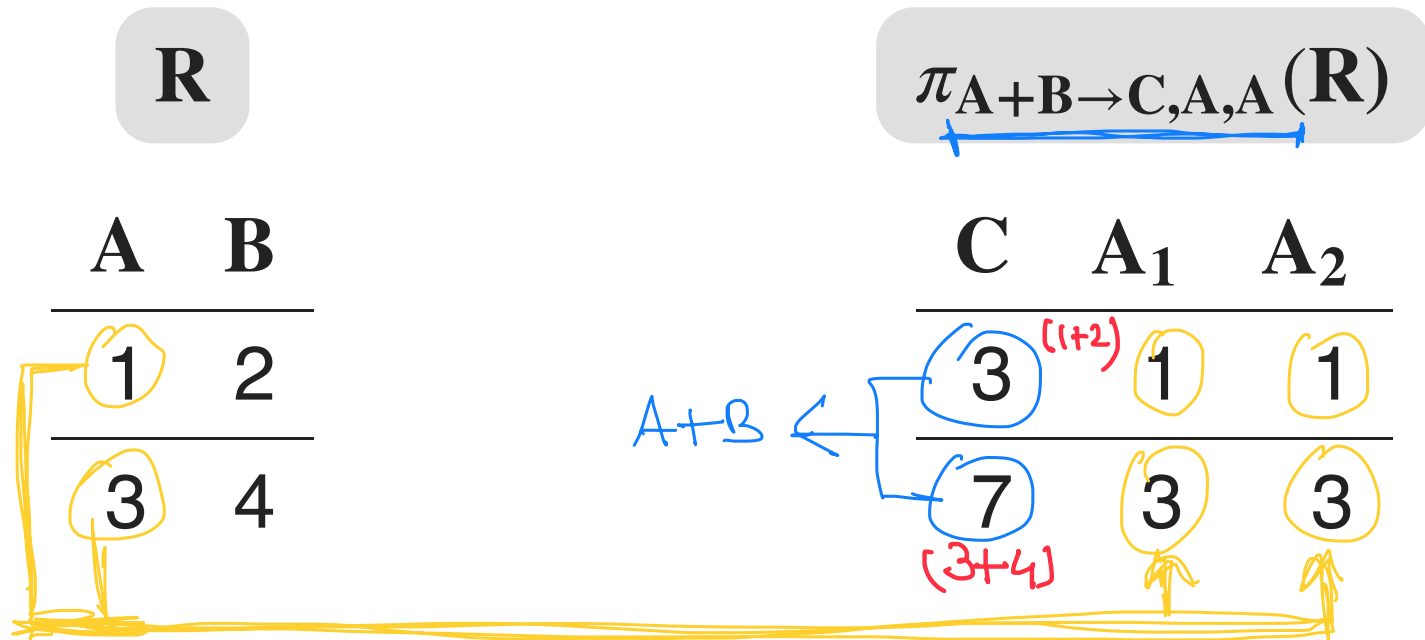
We extend what is allowed in list L for π .

We allow it to contain arbitrary expressions involving the attr's.

1. Arithmetic operations over the attr's; e.g., $A + B \rightarrow C$.
2. String manipulation operators over string-domain attr's, concatenate, etc.
3. Duplicate occurrences of the same attr.! E.g.,
 $\pi_{A,A,A}(\mathbf{R})$.

Note that an “identity” operator would let us “copy” an attr. *while* giving it a new name; i.e., $A \rightarrow B$. In fact, this would be preferable, as we want to insist that the columns (attr's) of a relation are uniquely named.

Example: extended projection



Product

$$\mathbf{R}_1 := \mathbf{R}_2 \times \mathbf{R}_3$$

- Pair each tuple $t_2 \in \mathbf{R}_2$ with each tuple $t_3 \in \mathbf{R}_3$.
- The concatenation $t_2 t_3$ is a tuple of \mathbf{R}_1 .
- The schema of \mathbf{R}_1 is the union of the attr's of \mathbf{R}_2 and \mathbf{R}_3 .

$$R_1 = \underset{\text{of } R_2}{\text{attribute}} \cup \underset{\text{of } R_3}{\text{attribute}}$$

Note. If there is an attr. named A in both \mathbf{R}_2 and \mathbf{R}_3 , we get *both* copies; by convention, we *rename* them $\mathbf{R}_2.A$ and $\mathbf{R}_3.A$, respectively.

Example: $R_1 := R_2 \times R_3$

R_2

A	B
1	2
3	4

R_3

B	C
5	6
7	8
9	10

R_1

A	$R_2.B$	$R_3.B$	C
1	2	5	6
1	2	7	8
1	2	9	10
3	4	5	6
3	4	7	8
3	4	9	10

Tuples = $2 \times 3 = 6$ (rows)
Columns = $2 \times 2 = 4$

Theta-Join *(more convenient than Product)*

$$\mathbf{R}_1 := \mathbf{R}_2 \bowtie_C \mathbf{R}_3$$

- Take the product $\mathbf{R}_2 \times \mathbf{R}_3$.
- Then apply σ_C to the results.

Thus, $\mathbf{R}_2 \bowtie_C \mathbf{R}_3 \equiv \sigma_C(\mathbf{R}_2 \times \mathbf{R}_3)$.

Condition to specify what we can join.

As with σ , C can be any boolean-valued condition.

Older versions of this allowed only $A \theta B$ where “ θ ” was limited to “=”, “<”, etc. Hence, the name.

Natural Join



A (very!) useful variant is called *natural join*.

- This assumes for “C” equalities between each pair of attr's from the two tables with the same name.
- And then only one copy of each such pair of equated attr's is kept; that is, one copy of each such pair is *projected out*.


Since the “C” is understood, this is denoted as

$$\mathbf{R_1} := \mathbf{R_2} \bowtie \mathbf{R_3}$$

Example: natural join

Sells

pub	beer	price
Joe's	Bud	2.50
Joe's	Molsen	2.75
Fox	Bud	2.50
Fox	Molsen	3.50





Pub

pub	addr
Joe's	Maple St
Fox	River Rd

Sells ⋈ Pub

pub	beer	price	addr
Joe's	Bud	2.50	Maple St
Joe's	Molsen	2.75	Maple St
Fox	Bud	2.50	River Rd
Fox	Molsen	3.50	River Rd



Renaming

The ρ operator gives a new schema to a relation. It is a way to “rename” a relation.

$\mathbf{R}_1 := \rho_{\mathbf{R}_1(A_1, \dots, A_n)}(\mathbf{R}_2)$ makes \mathbf{R}_1 as a relation with attr's A_1, \dots, A_n , and the same tuples as \mathbf{R}_2 .

Simplified notation. $\mathbf{R}_1(A_1, \dots, A_n) := \mathbf{R}_2$.

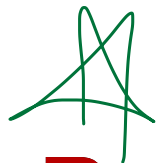
Example: Renaming

Pub

pub	addr
Joe's	Maple St
Fox	River Rd

R(bar, addr) := Pub

bar	addr
Joe's	Maple St
Fox	River Rd



Building complex expressions

Combine operations with parentheses and precedence rules.

Three notations, as in arithmetic.

1. Sequences of assignment statements.
2. Expressions with several operators.
3. Expression trees.

Sequences of assignments

Create temporary names.

Renaming can be done by giving relations lists of attr's.

Example: $R_3 := R_1 \bowtie_C R_2$ can be written as

- $R_4 := R_1 \times R_2$
- $R_5 := \sigma_C(R_4)$

Expressions in a single assignment

Example. The theta-join $\mathbf{R}_3 := \mathbf{R}_1 \bowtie_C \mathbf{R}_2$ can be written as $\mathbf{R}_3 := \sigma_C(\mathbf{R}_1 \times \mathbf{R}_2)$.

Precedence of relational operators.

1. $[\sigma, \pi, \rho]$ (*highest*)
2. $[\times, \bowtie]$
3. $[\cap]$
4. $[\cup, -]$

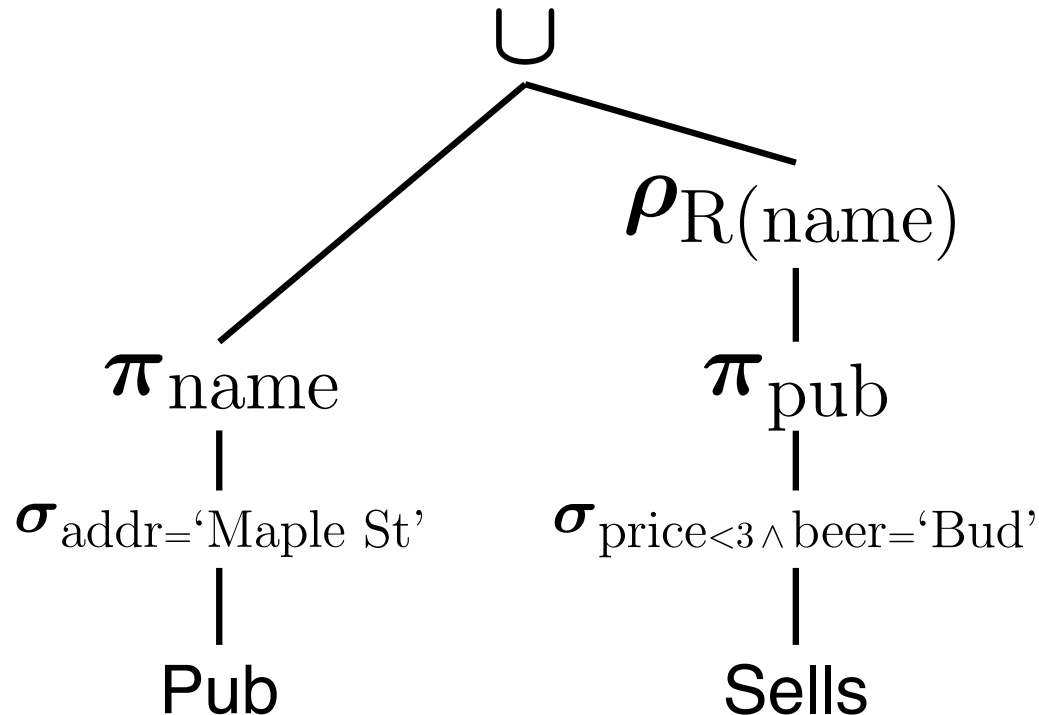


Expression trees

- Leaves are operands; standing for relations.
- Interior nodes are operators, applied to their child or children.

Example: tree for a query

Using the rel'ns **Pub**(name, addr) and **Sells**(pub, beer, price), find the names of all the pubs that are either on Maple St or that sell Bud for less than \$3.



Example: self-join

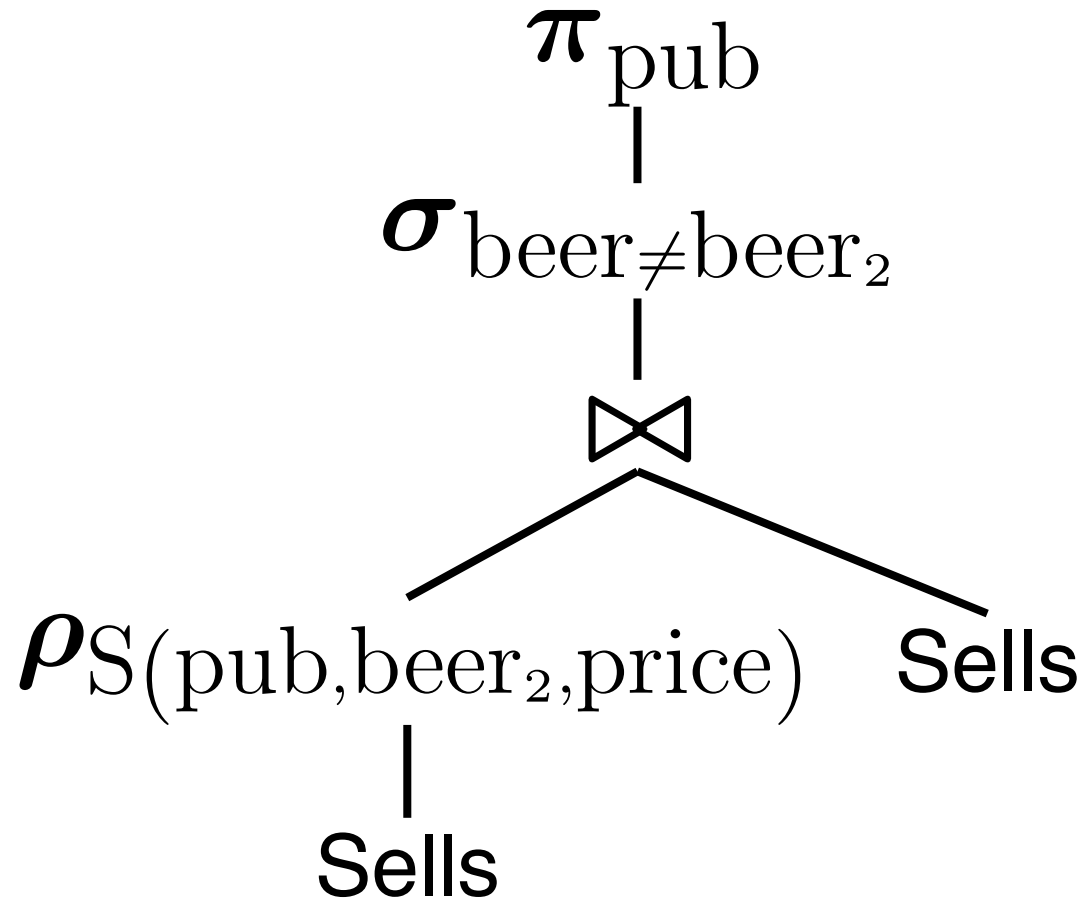
Using the rel'n **Sells**(pub, beer, price), find the pubs that sell two different beers at the *same* price.

Strategy. By renaming, define a *copy* of **Sells** — call it **S**(pub, beer₂, price). The natural join of **Sells** and **S** consists of the tuples of schema (pub, beer, beer₂, price) such that the pub sells both beers at the same price.

And *select* so that beer and beer₂ are *not* the *same* beer.

The tree

Beers the same price



Examples: Colour Schema

Customer

cust# **PK**
cname
fav_colour
phone#

Item

item# **PK**
prod# **FK to Product**
cust# **FK to Customer**
colour
date_sold

Product

prod# **PK**
pname
cost
maker **FK to Company**

Avail_Colour

prod# **PK, FK to Product**
colour **PK**

Q_1 . Products in customer's favorite colour

Show, for each customer (reporting the customer's name), the products by name that come in the customer's favourite colour.

$$\pi_{\text{cname,pname}} \left(\left(\mathbf{Customer} \bowtie_{\text{fav_colour=colour}} \mathbf{Avail_Colour} \right) \right. \\ \left. \bowtie \mathbf{Product} \right)$$

Q_2 . Products not in customer's favorite colour

Show, for each customer (reporting the customer's name), the products by name that *do not* come in the customer's favourite colour.

$$\pi_{\text{cname,pname}} \left(\pi_{\text{cust\#,cname,prod\#,pname}} (\text{Customer} \times \text{Product}) \right. \\ \left. - \right. \\ \left. \pi_{\text{cust\#,cname,prod\#,pname}} \left((\text{Customer} \bowtie_{\text{fav_colour=colour}} \text{Avail_Colour}) \bowtie \text{Product} \right) \right)$$

UPTD
here
or
22/10/20.

Q_3 . Two or more in common

List pairs of customers — with columns `first_cust#`, `first_cname`, `second_cust#`, `second_cname` — such that the two customers own *at least* two products in common.

Write your query so that a pair is *not* listed twice. For instance, if $\langle 5, \text{franck}, 7, \text{parke} \rangle$ is listed, then $\langle 7, \text{parke}, 5, \text{franck} \rangle$ should not be.

```
One :=  $\sigma_{c1\# < c2\#} ($   
     $\pi_{c1\#, cn1, prod\#} ( \pi_{cust\# \rightarrow c1\#, cname \rightarrow cn1} (Customer \bowtie Item) )$   
     $\bowtie$   
     $\pi_{c2\#, cn2, prod\#} ( \pi_{cust\# \rightarrow c2\#, cname \rightarrow cn2} (Customer \bowtie Item) )$   
)  
 $\pi_{c1\#, cn1, c2\#, cn2} ( \sigma_{p1\# \neq p2\#} ( \rho_{prod\# \rightarrow p1\#} (One) \bowtie \rho_{prod\# \rightarrow p2\#} (One) ) )$ 
```

Q_4 . Customers with all colours

List customers who own items in *all* the available colours. That is, for every available colour, the customer owns some item in that colour.

$$\begin{aligned} &\pi_{\text{cust\#,cname}}(\mathbf{Customer}) \bowtie \\ &\quad (\pi_{\text{cust\#}}(\mathbf{Customer}) - \\ &\quad \quad \pi_{\text{cust\#}}((\pi_{\text{cust\#}}(\mathbf{Customer}) \times \pi_{\text{colour}}(\mathbf{Avail_Colour})) - \pi_{\text{cust\#,colour}}(\mathbf{Item}))) \end{aligned}$$

Q_5 . Most expensive items

List each customer by name, paired with the product(s) by name that he or she has bought that was the most expensive (cost) of all the products he or she has bought.

Note that there actually may be ties. For instance, $\langle \text{bruce}, \text{ferrari} \rangle$ and $\langle \text{bruce}, \text{porsche} \rangle$ would both qualify if both were \$80,000, and for everything else he has bought, each item was less expensive than \$80,000.

Costs := $\pi_{\text{cust\#,cname,prod\#,pname,cost}}((\text{Customer} \bowtie \text{Item}) \bowtie \text{Product})$

$\pi_{\text{cust\#,cname,prod\#,pname}}(\text{Costs}$

Costs

—

$\pi_{\text{cust\#,cname,prod\#,pname,cost}}(\sigma_{\text{cost} < \text{cost2}}(\text{Costs} \bowtie \pi_{\text{cust\#,cname,cost} \rightarrow \text{cost2}}(\text{Costs}))))$

Bag vs set semantics

Relational algebra (RA) is usually considered with *set* semantics; that is, each operator returns a *set* of tuples. Thus, there are *no* duplicate tuples in the return.

But we can interpret RA with a *bag* (*multi-set*) semantics instead, if we wanted. Then duplicate tuples can be returned in the answer *bag*.

How would this change our different operators?

RA operators w/ bag semantics

- **select** (“ σ ”). Selects from the input, as before.
Can only return duplicates if the input table has duplicates.
- **project** (“ π ”). Now returns exactly the same number of tuples as the input table.
- **product** (“ \times ”). Each tuple of rel'n #1 is concatenated with each tuple of rel'n #2, as before.
Can only return duplicates if the input tables have duplicates.
- **join** (“ \bowtie ”). Defined via “ σ ” and “ \times ” as before.

RA “set” operators w/ bag semantics

- **intersection** (“ \cap ”). Given tuple t appears in rel'n #1 m times and in rel'n #2 n times, then t appears in the result $\min(m, n)$ times.
- **union** (“ \cup ”). Given tuple t appears in rel'n #1 m times and in rel'n #2 n times, then t appears in the result $m + n$ times.
- **minus** (“ $-$ ”). Say we have $\mathbf{R} - \mathbf{S}$. Given tuple t appears in \mathbf{R} m times and in \mathbf{S} n times, if $m > n$, then t appears in the result $m - n$ times; else, t does *not* appear in the result.

Declarative

But ... RA is *not* “declarative”. We are having to specify the *order* of our operations, to say *how* the query is to be evaluated.

In a *declarative* query language, we (ideally) just specify *what* we want, and not *how* to obtain it.

Would a *declarative* query language be possible? Yes.

- relational calculus
- datalog
- SQL

Relational calculus

- A query is stated in *predicate calculus* in *set definition* form.

I.e., {template | predicate statement}

- Rel'ns are just *predicates*.
- The *logical variables* range either over
 - attributes' *domains* (*domain relational calculus*) or
 - relations' *tuples* (*tuple relational calculus*).

Datalog

- Quite similar to the *domain relational calculus*, but a nicer syntax.
- Parallels the *programming language* **Prolog**.
- Used widely in academic research in databases and in AI.
- E.g. “Which students were enrolled in EECS-3421 and earned an ‘A’?”

← **student**(S#, Name, Addr),
 class(C#, ‘EECS’, ‘3421’, Term, Year, Sect),
 enrol(S#, C#, ‘A’).

SQL: “Intergalactic Data Speak”

- Declarative!
- RA principle of “tables in, tables out.”
- “Logical variables” that range over tuples.
(So, an implementation of *tuple relational calculus*.)
- Built-in choice of *set* or *bag* semantics.
- Handles *null* “values”.
- Meant to look so much like English that *anyone* can write queries.
- Plus lots, lots more! (Is a *standard*.)

Parke's thoughts. One of the ugliest languages ever!