Conceptual Query Languages

Relational Algebra

Basic Operations

Table of Contents

Parke Godfrey

2016-10-19 initial [v1]

2016-10-24 [v2]

Acknowledgments

Thanks

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

What is an *algebra*?

A mathematical system consisting of

- **operands**, variables or values from which new values can be constructed; and
- **operators**, symbols denoting procedures that construct new values from given values.

What is the *relational algebra*?

- An algebra whose operands are *relations*.
- 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* on relations.

The SQL Language

R

a	b
5	20
10	30
20	40

```
select b
from R
where a <= 10;
```

a	b
5	20
10	30

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

$\mathbf{R_1} := \sigma_C(\mathbf{R_2})$

- 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

$\sigma_{\text{pub}=\text{“Joe's”}}(\textbf{Sells})$

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

Projection

$$\mathbf{R_1} := \pi_{\mathrm{L}}(\mathbf{R_2})$$

- 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

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

beer	price
Bud	2.50
Molsen	2.75
Molsen	3.50

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

Note that an “identity” operator would let us “copy” an attr. *while* giving it a new name; i.e., $\mathrm{A} \rightarrow \mathrm{B}$. 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

$\mathbf{R}$

A	B
1	2
3	4

$\pi_{\mathbf{A+B \to C, A, A}}(\mathbf{R})$

C	A_1	A_2
3	1	1
7	3	3

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 $\mathbf{R_3}$.

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 $\mathbf{R_2}.\mathrm{A}$ and $\mathbf{R_3}.\mathrm{A}$, 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

Theta-Join

$$\mathbf{R_1} := \mathbf{R_2} \bowtie_{\mathrm{C}} \mathbf{R_3}$$

- Take the product $\mathbf{R_2} \times \mathbf{R_3}$.
- Then apply σ_C to the results.

Thus, $\mathbf{R_2} \bowtie_{\mathrm{C}} \mathbf{R_3} \equiv \sigma_{\mathrm{C}}(\mathbf{R_2} \times \mathbf{R_3})$.

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

Older versions of this allowed only $\mathrm{A}\,\theta\,\mathrm{B}$ 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

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

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 $\bowtie$ 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.

$\mathbf{R_1} := \rho_{\mathbf{R_1}(\mathrm{A}_1,\ldots,\mathrm{A}_n)}(\mathbf{R_2})$ makes $\mathbf{R_1}$ as a relation with attr's $\mathrm{A}_1, \ldots, \mathrm{A}_n$, and the same tuples as $\mathbf{R_2}$.

Simplified notation. $\mathbf{R_1}(\mathrm{A}_1, \ldots, \mathrm{A}_n) := \mathbf{R_2}$.

Example: Renaming

Pub

pub	addr
Joe's	Maple St
Fox	River Rd

$\mathbf{R}(\text{bar}, \text{addr}) := \mathbf{Pub}$

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: $\mathbf{R_3 := R_1 \bowtie_C R_2}$ can be written as

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

Expressions in a single assignment

Example. The theta-join $\mathbf{R_3 := R_1 \bowtie_C R_2}$ can be written as $\mathbf{R_3 := \sigma_C(R_1 \times R_2)}$.

Precedence of relational operators.

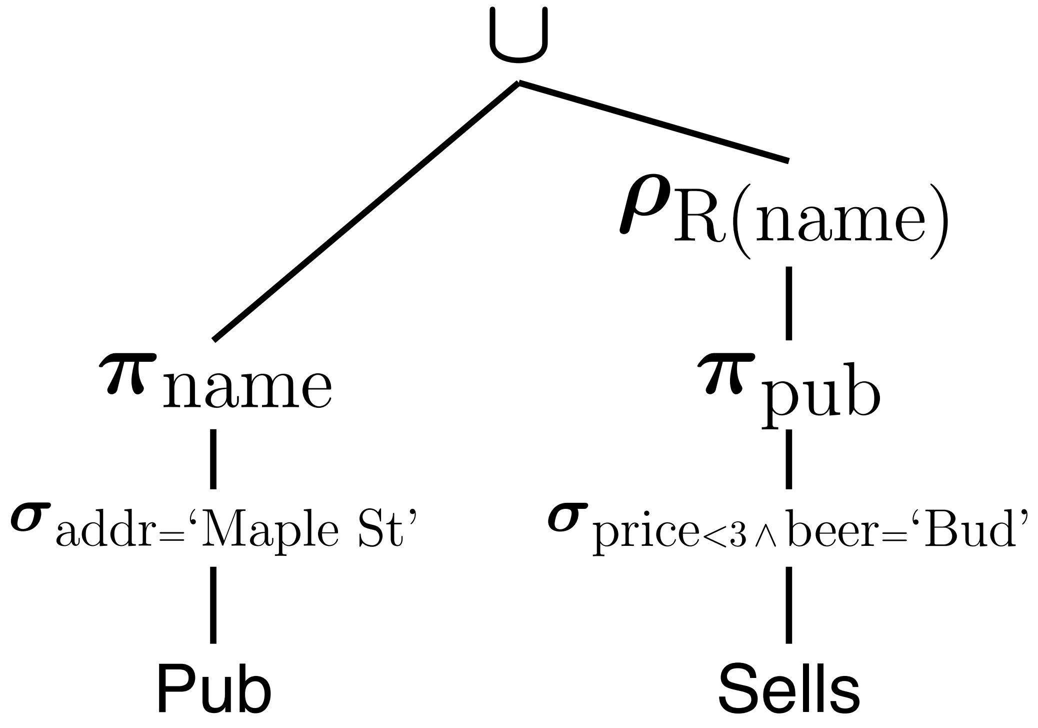
1. $[\sigma, \pi, \rho]$ (*highest*)
2. $[\times, \bowtie]$
3. $[\cap]$
4. $[\cup, -]$

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



$\cup$
$\rho_{R(\text{name})}$
π_{name}
π_{pub}
$\sigma_{\text{addr}=\text{'Maple St'}}$
$\sigma_{\text{price}<3 \wedge \text{beer}=\text{'Bud'}}$
Pub
Sells

Example: self-join

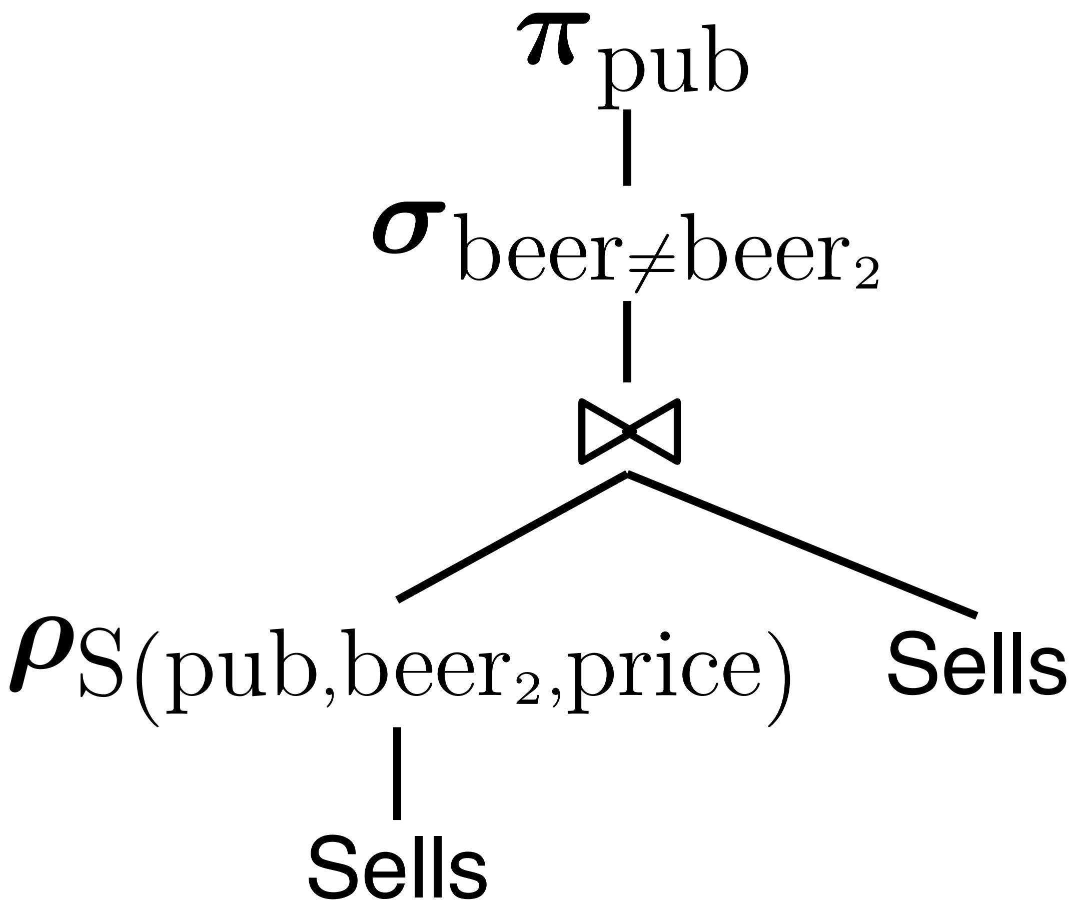
Using the rel'n **Sells**(pub, beer, price), find the pubs that sell two different beers at the *same* price.

Strategy. By renaming, define a *copy* of **Sells** — call it **S**(pub, beer_2, price). The natural join of **Sells** and **S** consists of the tuples of schema (pub, beer, beer_2, price) such that the pub sells both beers at the same price.

And *select* so that beer and beer_2 are *not* the *same* beer.

The tree

Beers the same price


π_{pub}
$\sigma_{\text{beer}\neq\text{beer}_2}$
$\bowtie$
$\rho_{S(\text{pub},\text{beer}_2,\text{price})}$
Sells
Sells

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**
colour **PK, FK** to **Company**

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.

$$\pi_{\text{cname,pname}}\left(\left(\textbf{Customer} \bowtie_{\text{fav\_colour=colour}} \textbf{Avail\_Colour}\right) \bowtie \textbf{Product}\right)$$

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.

$$\pi_{\text{cname,pname}}\left(\pi_{\text{cust\#,cname,prod\#,pname}}\left(\textbf{Customer} \times \textbf{Product}\right)\right.$$
$$-$$
$$\left.\pi_{\text{cust\#,cname,prod\#,pname}}\left(\left(\textbf{Customer} \bowtie_{\text{fav\_colour=colour}} \textbf{Avail\_Colour}\right) \bowtie \textbf{Product}\right)\right)$$

Q_3. Two or more in common

List pairs of customers — with columns $\text{first\_cust\#}, \text{first\_cname}, \text{second\_cust\#}, \text{second\_cname}$ — 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.

$$
\begin{aligned}
\textbf{One} := \sigma_{\text{c1\#<c2\#}}(\\
&\pi_{\text{c1\#,cn1,prod\#}}(\pi_{\text{cust\#}\to\text{c1\#,cname}\to\text{cn1}}(\textbf{Customer} \bowtie \textbf{Item}))\\
&\qquad\bowtie\\
&\pi_{\text{c2\#,cn2,prod\#}}(\pi_{\text{cust\#}\to\text{c2\#,cname}\to\text{cn2}}(\textbf{Customer} \bowtie \textbf{Item}))\\
)
\end{aligned}
$$

$$
\pi_{\text{c1\#,cn1,c2\#,cn2}}(\sigma_{\text{p1\#}\neq\text{p2\#}}(\rho_{\text{prod\#}\to\text{p1\#}}(\textbf{One}) \bowtie \rho_{\text{prod\#}\to\text{p2\#}}(\textbf{One})))
$$

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.

$$\begin{aligned}
&\pi_{\text{cust\#,cname}}(\textbf{Customer}) \bowtie \\
&\qquad (\pi_{\text{cust\#}}(\textbf{Customer}) - \\
&\qquad\qquad \pi_{\text{cust\#}}((\pi_{\text{cust\#}}(\textbf{Customer}) \times \pi_{\text{colour}}(\textbf{Avail\_Colour})) - \pi_{\text{cust\#,colour}}(\textbf{Item})))
\end{aligned}$$

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 $\$80{,}000$.

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

$$\begin{array}{l}
\pi_{\text{cust\#,cname,prod\#,pname}}(\\
\qquad \textbf{Costs} \\
\qquad\qquad - \\
\qquad \pi_{\text{cust\#,cname,prod\#,pname,cost}}(\sigma_{\text{cost<cost2}}(\\
\qquad\qquad \textbf{Costs} \bowtie \pi_{\text{cust\#,cname,cost}\rightarrow\text{cost2}}(\textbf{Costs}))))
\end{array}$$

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** (“∩”). 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** (“∪”). 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., $\{\text{template} \mid \text{predicate statement}\}$

- 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 enroled in EECS-3421 and earned an ‘A’?”

$$
\begin{aligned}
\leftarrow\ & \textbf{student}(\text{S\#, Name, Addr}),\\
& \textbf{class}(\text{C\#, `EECS', `3421', Term, Year, Sect}),\\
& \textbf{enrol}(\text{S\#, C\#, `A'}).
\end{aligned}
$$

SQL: “Intergalatic Data Speak”

- Declaractive!
- 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!