Relational Algebra

Relational Algebra

Relational algebra (RA) can be viewed as ...

- mathematical system for manipulating relations, or
- data manipulation language (DML) for the relational model

Relational algebra consists of:

- operands: relations, or variables representing relations
- operators that map relations to relations
- rules for combining operands/operators into expressions
- rules for evaluating such expressions

RA can be viewed as the "machine language" for RDBMSs

Relational Algebra (cont)

Core relational algebra operations:

- selection: choosing a subset of rows
- projection: choosing a subset of columns
- product, join: combining relations

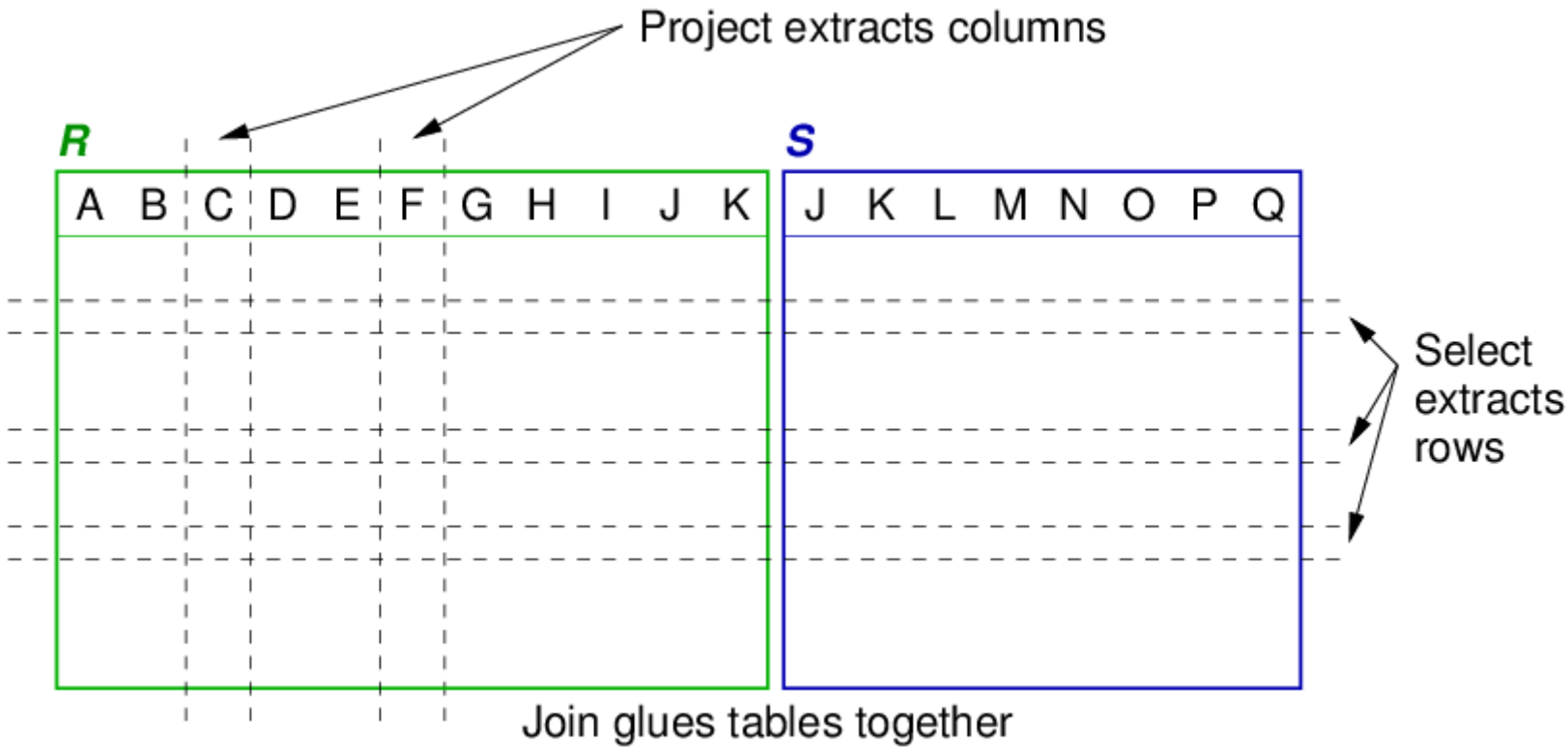
- union, intersection, difference: combining relations
- rename: change names of relations/attributes

Common extensions include:

- aggregation, projection++, division

Relational Algebra (cont)

Select, project, join provide a powerful set of operations for constructing relations and extracting relevant data from them.



Adding set operations and renaming makes RA complete.

Notation

Standard treatments of relational algebra use Greek symbols.

We use the following notation (because it is easier to reproduce):

Operation	Standard Notation	Our Notation
Selection	$\sigma_{expr}(Rel)$	*Sel[expr](Rel)*
Projection	$\pi_{A,B,C}(Rel)$	*Proj[A,B,C](Rel)*
Join	$Rel_1 \bowtie_{expr} Rel_2$	Rel_1 *Join[expr]* Rel_2
Rename	$\rho_{schema} Rel$	*Rename[schema](Rel)*

For other operations (e.g. set operations) we adopt the standard notation.

Notation (cont)

We define the semantics of RA operations using

- regular "conditional set" expressions e.g. *{ X | condition }*

  (tuple relational calculus ... cf. Haskell list comprehensions)

- tuple notations:
  - *t[AB]* (extracts attributes *A* and *B* from tuple *t*)
  - *(x,y,z)* (enumerated tuples; specify attribute values)
- quantifiers, set operations, boolean operators

Notation (cont)

All RA operators return a result relation (no DB updates).

For convenience, we can name a result and use it later.

E.g.

```
Temp = R op1 S op2 T
Res  = Temp op3 Z
-- which is equivalent to
Res  = (R op1 S op2 T) op3 Z
```

Each "intermediate result" has a well-defined schema.

Sample Relations

Example database #0 to demonstrate RA operators:

r1

A	B	C	D
a	1	x	4
b	2	y	5
c	4	z	4
d	8	x	5
e	1	y	4
f	2	x	5

r2

E	F	G
1	a	x
2	b	y
3	c	x
4	a	y
5	b	x

Sample Relations (cont)

Example database #1 to demonstrate RA operators:

R

A	B	C	D
a	1	x	4
b	2	y	5
c	4	z	4
d	8	x	5
e	1	y	4
f	2	x	5

S

D	E	F
1	a	x
2	b	y
3	c	x
4	a	y
5	b	x

Sample Relations (cont)

Example database #2 to demonstrate RA operators:

Beers(name,manf)
(VB, Carlton) (New, Tooheys) (Porter, Maltshovel) …

Beers(name,addr,licence)
(CBH, Coogee,433122) (Royal, Randwick, 632987) (Regent, Kingsford,112112 …

Frequents(drinker,bar)
(John, CBH) (Gernot, CBH) (Gernot, Regent) …

Likes(drinker, beer)
(Andrew, New) (Gernot, Porter) (John, Pale Ale) …

Beers(name,addr,phone)
(John, Alexandria, 93111139 (Gernot, Newtown, 92422429) (Andrew, Glebe, 90411049) …

Sells(beer,bar,price)
(CBH, New, 2.50) (CBH, VB, 1.99 Royal, Porter, 3.00 …

Selection

Selection returns a subset of the tuples in a relation *r* that satisfy a specified condition *C*.

$$\sigma_C(r) \quad = \quad Sel[C](r) \quad = \quad \{\ t \mid t \in r \wedge C(t)\ \}, \qquad \text{where } r(R)$$

C is a boolean expression on attributes in *R*.

Result size: $|\sigma_C(r)| \leqslant |r|$

Result schema: same as the schema of r (i.e. R)

Programmer's view:

```
result = {}
for each tuple t in relation r
    if (C(t)) { result = result ∪ {t} }
```

Selection (cont)

Example selections:

Sel [B = 1] (r1)

A	B	C	D
a	1	x	4
e	1	y	4

Sel [A=b ∨ A=c] (r1)

A	B	C	D
b	2	y	5
c	4	z	4

Sel [B ⩾ D] (r1)

A	B	C	D
c	4	z	4
d	8	x	5

Sel [A = C] (r1)

A	B	C	D

Selection (cont)

Example queries:

- Find details about the Perryridge branch?
  - *Sel [branchName=Perryridge] (Branch)*
- Which accounts are overdrawn?
  - *Sel [balance<0] (Account)*
- Which Round Hill accounts are overdrawn?
  - *Sel [branchName=Round Hill ∧ balance<0] (Account)*

Projection

Projection returns a set of tuples containing a subset of the attributes in the original relation.

$\pi_X(r) \quad = \quad Proj[X](r) \quad = \quad \{\ t[X] \mid t \in r\ \}, \qquad$ where $r(R)$

X specifies a subset of the attributes of R.

Note that removing key attributes can produce duplicates.

In RA, duplicates are removed from the result set.
(In many RDBMS's, duplicates are retained (i.e. they use bag, not set, semantics))

Result size: $|\pi_X(r)| \leqslant |r|$ Result schema: $R'(X)$

Programmer's view:

```
result = {}
for each tuple t in relation r
```

```
result = result ∪ {t[X]}
```

Projection (cont)

Example projections:

Proj [A,B,D] (r1)

A	B	D
a	1	4
b	2	5
c	4	4
d	8	5
e	1	4
f	2	5

Proj [B,D] (r1)

B	D
1	4
2	5
4	4
8	5

Proj [D] (r1)

D
4
5

Projection (cont)

Example queries:

- *Proj [branchName] (Branch)*
- Which branches actually hold accounts?
  - *Proj [branchName] (Account)*
- What are the names and addresses of all customers?
  - *Proj [name,address] (Customer)*
- Generate a list of all the account numbers
  - *Proj [accountNo] (Account)* or
  - *Proj [account] (Depositor)* (if we assume every account has a depositor)

Union

Union combines two compatible relations into a single relation via set union of sets of tuples.

$r_1 \cup r_2 \;=\; \{\, t \mid t \in r_1 \vee t \in r_2 \,\}$, where $r_1(R)$, $r_2(R)$

Compatibility = both relations have the same schema

Result size: $|r_1 \cup r_2| \leqslant |r_1| + |r_2|$ Result schema: R

Programmer's view:

```
result = r1
for each tuple t in relation r2
    result = result ∪ {t}
```

Union (cont)

Example queries:

- Which suburbs have either customers or branches?

- *Proj[address](Customer) ∪ Proj[address](Branch)*

- Which branches have either customers or accounts?
  - *Proj[homeBranch](Customer) ∪ Proj[branchName](Account)*

The union operator is symmetric i.e. $R \cup S = S \cup R$.

Intersection

Intersection combines two compatible relations into a single relation via set intersection of sets of tuples.

$r_1 \cap r_2 = \{ t \mid t \in r_1 \wedge t \in r_2 \}$, where $r_1(R)$, $r_2(R)$

Uses same notion of relation compatibility as union.

Result size: $|r_1 \cup r_2| \leqslant \min(|r_1|,|r_2|)$ Result schema: R

Programmer's view:

```
result = {}
for each tuple t in relation r1
    if (t ∈ r2) { result = result ∪ {t} }
```

Intersection (cont)

Example queries:

- Which suburbs have both customers and branches?
  - *Proj[address](Customer) ∩ Proj[address](Branch)*
- Which branches have both customers and accounts?
  - *Proj[homeBranch](Customer) ∩ Proj[branchName](Account)*

The intersection operator is symmetric i.e. $R \cap S = S \cap R$.

Difference

Difference finds the set of tuples that exist in one relation but do not occur in a second compatible relation.

$r_1 - r_2 \quad = \quad \{ t \mid t \in r_1 \wedge t \in r_2 \}, \qquad$ where $r_1(R), r_2(R)$

Uses same notion of relation compatibility as union.

Note: tuples in r_2 but not r_1 do not appear in the result

i.e. set difference != complement of set intersection Programmer's view:

```
result = {}
for each tuple t in relation r1
    if (!(t ∈ r2)) { result = result ∪ {t} }
```

Difference (cont)

Example difference:

s1 = Sel [B = 1] (r1) *s2 = Sel [C = x] (r1)*

A	B	C	D
a	1	x	4
e	1	y	4

A	B	C	D
a	1	x	4
d	8	x	5

s1 – s2

A	B	C	D
e	1	y	4

s2 – s1

A	B	C	D
d	8	x	5

Clearly, difference is not symmetric.

Difference (cont)

Example queries:

- Which customers have no accounts?
  - *AllCusts = Proj[customerNo](Customer)*
    CustsWithAccts = Proj[customer](Depositor)
    Result = AllCusts – CustsWithAccts
- Which branches have no customers?
  - *AllBranches = Proj[branchName](Branch)*
    BranchesWithCusts = Proj[homeBranch](Customer)
    Result = AllBranches – BranchesWithCusts

Product

Product (Cartesian product) combines information from two relations pairwise on tuples.

$r \times s \;=\; \{ (t_1 : t_2) \mid t_1 \in r \wedge t_2 \in s \}$, where $r(R)$, $s(S)$

Each tuple in the result contains all attributes from *r* and *s*, possibly with some fields renamed to avoid ambiguity.

If $t_1 = (A_1...A_n)$ and $t_2 = (B_1...B_n)$ then $(t_1:t_2) = (A_1...A_n,B_1...B_n)$

Note: relations do not have to be union–compatible.

Result size is large: $|r \times s| = |r|.|s|$ Schema: $R \cup S$

Programmer's view:

```
result = {}
for each tuple t1 in relation r
    for each tuple t2 in relation s
        result = result ∪ {(t1:t2)} }
```

Product (cont)

Example product #1:

r1 × r2

A	B	C	D	E	F	G
a	1	x	4	1	a	x

a	1	x	4	2	b	y
a	1	x	4	3	c	x
a	1	x	4	4	a	y
a	1	x	4	5	b	x
b	2	y	5	1	a	x
b	2	y	5	2	b	y
b	2	y	5	3	c	x
...						
f	2	z	5	3	c	x
f	2	z	5	4	a	y
f	2	z	5	5	b	x

Product (cont)

Example product #2:

Students × Marks

course	name	sid	stude	subj	mark
BE	Anne	21333	21333	1011	74
BE	Anne	21333	21333	1021	70
BE	Anne	21333	21333	2011	68
BE	Anne	21333	21531	1011	94
BE	Anne	21333	21531	1021	90
BE	Anne	21333	21623	1011	50
BE	Dave	21876	21333	1011	74
BE	Dave	21876	21333	1021	70
BE	Dave	21876	21333	2011	68
BE	Dave	21876	21531	1011	94
BE	Dave	21876	21531	1021	90
BE	Dave	21876	21623	1011	50
BSc	John	21531	21333	1011	74

BSc	John	21531	21333	1021	70
BSc	John	21531	21333	2011	68
BSc	John	21531	21531	1011	94
BSc	John	21531	21531	1021	90
BSc	John	21531	21623	1011	50
BSc	Tim	21623	21333	1011	74
BSc	Tim	21623	21333	1021	70
BSc	Tim	21623	21333	2011	68
BSc	Tim	21623	21531	1011	94
BSc	Tim	21623	21531	1021	90
BSc	Tim	21623	21623	1011	50

Product (cont)

By itself, product isn't a useful querying mechanism.

However, it gives a basis for combining information across relations.

A special (and very common) usage of product:

- form all possible pairs in *r* × *s*
- select just the "interesting" pairs (matching key values)

This usage is represented by a separate operation: join.

Note: join is not implemented using product (which is why RDBMSs work)

Join is critically important in implementing "navigation" in relational databases.

Product (cont)

Example query #1:

- Who are the owners of account A101?
  - combine information from *Customer* and *Depositor*
  - *tmp1 = Customer × Depositor*
    tmp2 = Sel [account=A101] (tmp1)

tmp3 = Sel [customer=customerNo] (tmp2)
res = Proj [name] (tmp3)

Product (cont)

Example query #2:

- Which accounts are held in branches in Horseneck or Brooklyn?
  - combine information from *Account* and *Branch*
  - *tmp1 = Account × Branch*
    tmp2 = Sel [B.address = Horseneck] (tmp1)
    tmp3 = Sel [B.address = Brooklyn] (tmp1)
    tmp4 = tmp2 ∪ tmp3
    tmp5 = Sel [A.branchName = B.branchName] (tmp4)
    res = Proj [A.accountNo] (tmp5)

Product (cont)

Example query #3:

- Which customers hold accounts at a Brooklyn branch?
  - need to combine information from all relations
  - *tmp1 = Account × Branch × Customer × Depositor*
    tmp2 = Sel [B.address = Brooklyn] (tmp1)
    tmp3 = Sel [B.branchName = A.branchName] (tmp2)
    tmp4 = Sel [A.accountNo = account] (tmp3)
    tmp5 = Sel [C.customerNo = A.customer] (tmp4)
    res = Proj [C.name] (tmp5)

Rename

Rename provides "schema mapping".

If expression *E* returns a relation $R(A_1, A_2, \ldots A_n)$, then

$Rename[S(B_1, B_2, \ldots B_n)](E)$

gives a relation called S

- containing the same set of tuples as E
- but with the name of each attribute changed from A_i to B_i

Rename is like the identity function on the *contents* of a relation; it changes only the schema.

Rename can be viewed as a "technical" apparatus of RA.

Rename (cont)

Examples:

Rename[s1(J,K,L,M)](r1)

J	K	L	M
a	1	x	4
b	2	y	5

c	4	z	4
d	8	x	5
e	1	y	4
f	2	z	5

s1 Rename[s2(X,Y,Z)](r2)

X	Y	Z
1	a	x
2	b	y
3	c	x
4	a	y
5	b	x

s2

Natural Join

Natural join is a specialised product:

- containing only pairs that match on their common attributes
- with one of each pair of common attributes eliminated

Consider relation schemas *R(ABC..JKLM)*, *S(KLMN..XYZ)*.

The natural join of relations *r(R)* and *s(S)* is defined as:

$r \bowtie s \;=\; r \; Join \; s \;=$
$\{ (t_1[ABC..J] : t_2[K..XYZ]) \mid t_1 \in r \wedge t_2 \in s \wedge match \}$

where $match \;=\; t_1[K] = t_2[K] \wedge t_1[L] = t_2[L] \wedge t_1[M] = t_2[M]$

Programmer's view:

```
result = {}
for each tuple t1 in relation r
   for each tuple t2 in relation s
      if (matches(t1,t2))
         result = result ∪ {combine(t1,t2)}
```

Natural Join (cont)

Natural join can also be defined in terms of other relational algebra operations:

r Join s = Proj[R ∪ S] (Sel[match] (r × s))

We assume that the union on attributes eliminates duplicates.

If we wish to join relations, where the common attributes have different names, we rename the attributes first.

E.g. *R(ABC)* and *S(DEF)* can be joined by

R Join Rename[S(DCF)](S)

Note: |*r* ▫ *s*| ▫ |*r* × *s*|, so *join* not implemented via *product*.

Natural Join (cont)

Example (assuming that *A* and *F* are the common attributes):

r1 Join r2

A	B	C	D	E	G
a	1	x	4	1	x
a	1	x	4	4	y
b	2	y	5	2	y
b	2	y	5	5	x
c	4	z	4	3	x

Strictly, the operation above was: *r1 Join Rename[r2(E,A,G)](r2)*

Natural Join (cont)

Natural join is not quite the same as:

Sel[sid=stude] (Students × Marks)

course	name	sid	stude	subj	mark
BE	Anne	21333	21333	1011	74
BE	Anne	21333	21333	1021	70
BE	Anne	21333	21333	2011	68
BSc	John	21531	21531	1011	94
BSc	John	21531	21531	1021	90
BSc	Tim	21623	21623	1011	50

Natural Join (cont)

Compare this to the previous result:

Students Join Marks

course	name	sid	subj	mark
BE	Anne	21333	1011	74

BE	Anne	21333	1021	70
BE	Anne	21333	2011	68
BSc	John	21531	1011	94
BSc	John	21531	1021	90
BSc	Tim	21623	1011	50

As above, we assume *Rename[Marks(sid,subj,mark)](Marks)*

Natural Join (cont)

Example queries:

- Who is the owner of account A101?
  - *Proj[name](Sel[account=A101](Customer ⋈ Depositor))*
- Which accounts are held in branches in Horseneck?
  - *tmp1 = Sel[address=Horseneck](Account ⋈ Branch)*
    res = Proj[accountNo](tmp1))

- Which customers hold accounts at a Brooklyn branch?
  - *tmp1 = Account ▷ Branch ▷ Customer ▷ Depositor*
    res = Proj[name](Sel[address=Brooklyn](tmp1))

Theta Join

The theta join is a specialised product containing only pairs that match on a supplied condition *C*.

$r \triangleright_C s = \{ (t_1 : t_2) \mid t_1 \in r \wedge t_2 \in s \wedge C(t_1 : t_2) \}$,
where *r(R),s(S)*

Examples: *(r1 Join[B>E] r2) ... (r1 Join[E<D∧C=G] r2)*

Can be defined in terms of other RA operations:

$r \triangleright_C s = r \text{ Join}[C] \ s = \text{Sel}[C] (r \times s)$

Unlike natural join, "duplicate" attributes are not removed.

Note that $r \bowtie_{true} s = r \times s$.

Theta Join (cont)

Example theta join:

r1 Join[D < E] r2

A	B	C	D	E	F	G
a	1	x	4	5	b	x
c	4	z	4	5	b	x
e	1	y	4	5	b	x

r1 Join[B > 1 ∧ D < E] r2

A	B	C	D	E	F	G

c	4	z	4	5	b	x

(Theta join is an important component of most SQL queries; many examples of its use to follow)

Theta Join (cont)

Comparison between join operations:

- theta join allows arbitrary tests in the condition
  (and leaves all attributes from the original relations in the result)
- equijoin has only equality tests in the condition
  (and leaves all attributes from the original relations in the result)
- natural join has only equality tests on common attributes
  (and removes one of each pair of matching attributes)

Equijoin is a specialised theta join; natural join is like theta join followed by projection.

Outer Join

r Join s eliminates all *s* tuples that do not match some *r* tuple.

Sometimes, we wish to keep this information, so outer join

- includes all tuples from each relation in the result
- for pairs of matching tuples, concatenate attributes as for standard join
- for tuples that have no match, assign null to "unmatched" attributes

---

Outer Join (cont)

Example (assuming that *A* and *F* are the common attributes):

r1 OuterJoin r2

A	B	C	D	E	G

a	1	x	4	1	x
a	1	x	4	4	y
b	2	y	5	2	y
b	2	y	5	5	x
c	4	z	4	3	x
d	8	x	5	null	null
e	1	y	4	null	null
f	2	z	5	null	null

Contrast this to the result for *R Join S* presented earlier.

Outer Join (cont)

Another outer join example (compare to earlier similar join):

Students *OuterJoin* *Marks*

course	name	sid	subj	mark
BE	Anne	21333	1011	74
BE	Anne	21333	1021	70
BE	Dave	21876	null	null
BE	Anne	21333	2011	68
BSc	John	21531	1011	94
BSc	John	21531	1021	90
BSc	Tim	21623	1011	50

Outer Join (cont)

There are three variations of outer join *R OuterJoin S*:

- left outer join (*LeftOuterJoin*) includes all tuples from *R*
- right outer join (*RightOuterJoin*) includes all tuples from *S*
- full outer join (*OuterJoin*) includes all tuples from *R* and *S*

Which one to use depends on the application e.g.

If we want to know about all Students, regardless of whether they're enrolled in anything

Students LeftOuterJoin Enrolment

Outer Join (cont)

Operational description of *r(R) LeftOuterJoin s(S)*:

```
result = {}
for each tuple t1 in relation r
    nmatches = 0
    for each tuple t2 in relation s
        if (matches(t1,t2))
            result = result ∪ {combine(t1,t2)}
            nmatches++
    if (nmatches == 0)
        result = result ∪
                 {combine(t1,Snull)}
```

where S_{null} is a tuple from S with all atributes set to NULL.

Division

Consider two relation schemas R and S where $S \subset R$.

The division operation is defined on instances $r(R)$, $s(S)$ as:

$$r / s \quad = \quad r \; Div \; s \quad = \quad \{ \; t \mid t \in r[R-S] \wedge satisfy \; \}$$

where $satisfy \quad = \quad \forall \; t_s \in S \; (\; \exists \; t_r \in R \; (\; t_r[S] = t_s \wedge t_r[R-S] = t \;) \;)$

Operationally:

- consider each subset of tuples in R that match on $t[R-S]$
- for this subset of tuples, take the $t[S]$ values from each
- if this covers all tuples in S, then include $t[R-S]$ in the result

---

Division (cont)

Example:

R=Proj[D,C](r1)

D	C
4	x
4	y
4	z
5	x
5	y
5	z

S=Proj[G](r2)

G
x
y

R / S

name
4
5

Strictly, *R/S* is *R / Rename[S(C)](S)*

Division (cont)

Division handles queries that include the notion "for all".

E.g. Which customers have accounts at all branches?

We can answer this as follows:

- generate a relation of customers and branches where they hold accounts
- generate a relation of all branch names
- find which customers appear in tuples with every branch name

---

Division (cont)

RA operations for answering the query:

- customers and branches where they hold accounts
  - *r1 = Account ▷ Branch ▷ Customer ▷ Depositor*
    r2 = Proj [name,branchName] (r1)
- branch names
  - *r3 = Proj[branchName](Branch)*

- customers who appear in tuples with every branch name
  - *res = r2 Div r3*

Division (cont)

Example of answering the query (in a different database instance):

r2

name	branchName
Davis	Downtown
Hayes	Round Hill
Jones	Brighton
Jones	Downtown
Jones	Round Hill
Smith	Downtown
Smith	Round Hill

r3

branchName
Brighton
Downtown
Round Hill

r2/r3

name
Jones

Division (cont)

Division can be implemented in terms of other RA operations.

Consider R/S where $R = X \cup Y$ and $S = Y$ ...

$T = Proj[X](R)$	generate all potential result tuples
$U = T \times S$	combine potential results with all tuples from S
$V = U - R$	find any combined tuples that are not in R; any potential results which occur with all S values in R will be removed; V thus contains potential result tuples which do not occur with all S values in R
$W = Proj[X](V)$	the X parts of these tuples are "disqualified"
$Res = T - W$	so remove them to produce the result

Aggregation

Two types of aggregation are common in database queries:

- accumulating summary values for data in tables
  - typical operations *Sum*, *Average*, *Count*
  - many operations work on a single column
    (e.g. *Sum[assets](Branch)*)
- grouping sets of tuples with common values
  - *GroupBy[A_1...A_n](R)*
  - typically we group using only a single attribute

Aggregation (cont)

Example aggregations:

GroupBy[C](r1)

A	B	C	D

Sum[B](r1)

Sum

GroupBy[C]$_{Sum[B]}$(r1)

C	Sum

a	1	x	4
d	8	x	5
f	2	x	5
b	2	y	5
e	1	y	4
c	4	z	4

18

x	11
y	3
z	4

Generalised Projection

In standard projection, we select values of specified attributes.

In generalised projection we perform some computation on the attribute value before placing it in the result tuple.

Examples:

- Display branch assets in Aus$ rather than US$.
  - *Proj [branchname,address,assets*0.75] (Branch)*
- Display employee records using age rather than birthday.
  - *Proj [id,name,(today-birthdate)/365,salary] (Employee)*

Produced: 13 Sep 2020