

RELATIONAL ALGEBRA II

CS121: Relational Databases
Fall 2017 – Lecture 3

Last Lecture

2

- Query languages provide support for retrieving information from a database
- Introduced the relational algebra
 - ▣ A procedural query language
 - ▣ Six fundamental operations:
 - select, project, set-union, set-difference, Cartesian product, rename
 - ▣ Several additional operations, built upon the fundamental operations
 - set-intersection, natural join, division, assignment

Extended Operations

3

- Relational algebra operations have been extended in various ways
 - More generalized
 - More useful!
- Three major extensions:
 - Generalized projection
 - Aggregate functions
 - Additional join operations
- *All* of these appear in SQL standards

Generalized Projection Operation

4

- Would like to include computed results into relations
 - ▣ e.g. “Retrieve all credit accounts, computing the current ‘available credit’ for each account.”
 - ▣ Available credit = credit limit – current balance
- Project operation is generalized to include computed results
 - ▣ Can specify *functions* on attributes, as well as attributes themselves
 - ▣ Can also assign names to computed values
 - ▣ (Renaming attributes is also allowed, even though this is also provided by the ρ operator)

Generalized Projection

5

- Written as: $\Pi_{F_1, F_2, \dots, F_n}(E)$
 - ▣ F_i are arithmetic expressions
 - ▣ E is an expression that produces a relation
 - ▣ Can also name values: F_i **as** *name*
- Can use to provide derived attributes
 - ▣ Values are always computed from other attributes stored in database
- Also useful for updating values in database
 - ▣ (more on this later)

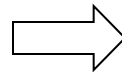
Generalized Projection Example

6

- “Compute available credit for every credit account.”

$\Pi_{cred_id, (limit - balance) \text{ as } available_credit}(credit_acct)$

cred_id	limit	balance
C-273	2500	150
C-291	750	600
C-304	15000	3500
C-313	300	25



cred_id	available_credit
C-273	2350
C-291	150
C-304	11500
C-313	275

credit_acct

Aggregate Functions

7

- Very useful to apply a function to a collection of values to generate a single result
- Most common aggregate functions:
 - sum** sums the values in the collection
 - avg** computes average of values in the collection
 - count** counts number of elements in the collection
 - min** returns minimum value in the collection
 - max** returns maximum value in the collection
- Aggregate functions work on multisets, not sets
 - ▣ A value can appear in the input multiple times

Aggregate Function Examples

8

“Find the total amount owed to the credit company.”

$G_{\text{sum}(\text{balance})}(\text{credit_acct})$

4275

cred_id	limit	balance
C-273	2500	150
C-291	750	600
C-304	15000	3500
C-313	300	25

credit_acct

“Find the maximum available credit of any account.”

$G_{\text{max}(\text{available_credit})}(\Pi_{(\text{limit} - \text{balance})} \text{ as available_credit}(\text{credit_acct}))$

11500

Grouping and Aggregation

9

- Sometimes need to compute aggregates on a *per-item* basis

- Back to the puzzle database:

puzzle_list(puzzle_name)

completed(person_name, puzzle_name)

puzzle_name
altekruise
soma cube
puzzle box

puzzle_list

- Examples:

- ▣ How many puzzles has *each person* completed?
- ▣ How many people have completed *each puzzle*?

person_name	puzzle_name
Alex	altekruise
Alex	soma cube
Bob	puzzle box
Carl	altekruise
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

Grouping and Aggregation (2)

10

puzzle_name
altekruise
soma cube
puzzle box

puzzle_list

“How many puzzles has each person completed?”

person_name	puzzle_name
Alex	altekruise
Alex	soma cube
Bob	puzzle box
Carl	altekruise
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

person_name $G_{\text{count}(\text{puzzle_name})}(\text{completed})$

- First, input relation *completed* is grouped by unique values of *person_name*
- Then, **count**(*puzzle_name*) is applied separately to each group

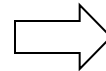
Grouping and Aggregation (3)

11

person_name **G****count**(*puzzle_name*)(*completed*)

Input relation is grouped by
person_name

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Alex	puzzle box
Bob	puzzle box
Bob	soma cube
Carl	altekruse
Carl	puzzle box
Carl	soma cube



Aggregate function is
applied to each group

person_name	
Alex	3
Bob	2
Carl	3

Distinct Values

12

- Sometimes want to compute aggregates over sets of values, instead of multisets

Example:

- Change puzzle database to include a *completed_times* relation, which records multiple solutions of a puzzle
- How many puzzles has each person completed?
 - Using *completed_times* relation this time

person_name	puzzle_name	seconds
Alex	altekruise	350
Alex	soma cube	45
Bob	puzzle box	240
Carl	altekruise	285
Bob	puzzle box	215
Alex	altekruise	290

completed_times

Distinct Values (2)

13

“How many puzzles has each person completed?”

- Each puzzle appears multiple times now.

person_name	puzzle_name	seconds
Alex	altekruise	350
Alex	soma cube	45
Bob	puzzle box	240
Carl	altekruise	285
Bob	puzzle box	215
Alex	altekruise	290

completed_times

- Need to count distinct occurrences of each puzzle's name

person_name $G_{\text{count-distinct}(\text{puzzle_name})}(\text{completed_times})$

Eliminating Duplicates

14

- Can append **-distinct** to any aggregate function to specify elimination of duplicates
 - ▣ Usually used with **count**: **count-distinct**
 - ▣ Makes no sense with **min**, **max**

General Form of Aggregates

15

- General form: $G_1, G_2, \dots, G_n \mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_m(A_m)}(E)$
 - ▣ E evaluates to a relation
 - ▣ Leading G_i are attributes of E to group on
 - ▣ Each F_i is aggregate function applied to attribute A_i of E
- First, input relation is divided into groups
 - ▣ If no attributes G_i specified, no grouping is performed (it's just one big group)
- Then, aggregate functions applied to each group

General Form of Aggregates (2)

16

- General form: $G_1, G_2, \dots, G_n \mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_m(A_m)}(E)$
- Tuples in E are grouped such that:
 - ▣ All tuples in a group have same values for attributes G_1, G_2, \dots, G_n
 - ▣ Tuples in different groups have different values for G_1, G_2, \dots, G_n
- Thus, the values $\{g_1, g_2, \dots, g_n\}$ in each group uniquely identify the group
 - ▣ $\{G_1, G_2, \dots, G_n\}$ are a superkey for the result relation

General Form of Aggregates (3)

17

- General form: $G_1, G_2, \dots, G_n \mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_m(A_m)}(E)$
- Tuples in result have the form:
 $\{g_1, g_2, \dots, g_n, a_1, a_2, \dots, a_m\}$
 - ▣ g_i are values for that particular group
 - ▣ a_i is result of applying F_i to the multiset of values of A_i in that group
- Important note: $F_i(A_i)$ attributes are unnamed!
 - ▣ Informally we refer to them as $F_i(A_i)$ in results, but they have no name.
 - ▣ Specify a name, same as before: $F_i(A_i)$ **as** *attr_name*

One More Aggregation Example

18

puzzle_name
altekruise
soma cube
puzzle box

puzzle_list

“How many people have completed each puzzle?”

$\text{puzzle_name } G_{\text{count}(\text{person_name})}(\text{completed})$


person_name	puzzle_name
Alex	altekruise
Alex	soma cube
Bob	puzzle box
Carl	altekruise
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

- What if nobody has tried a particular puzzle?
 - Won't appear in *completed* relation

One More Aggregation Example

19



puzzle_name
altekruise
soma cube
puzzle box
clutch box

puzzle_list

person_name	puzzle_name
Alex	altekruise
Alex	soma cube
Bob	puzzle box
Carl	altekruise
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

- New puzzle added to *puzzle_list* relation
 - ▣ Would like to see { “clutch box”, 0 } in result...
 - ▣ “clutch box” won’t appear in result!
- Joining the two tables doesn’t help either
 - ▣ Natural join won’t produce any rows with “clutch box”

Outer Joins

20

- Natural join requires that both left and right tables have a matching tuple

$$r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1=s.A_1 \wedge r.A_2=s.A_2 \wedge \dots \wedge r.A_n=s.A_n}(r \times s))$$

- Outer join is an extension of join operation
 - ▣ Designed to handle *missing information*
- Missing information is represented by *null* values in the result
 - ▣ *null* = unknown or unspecified value

Forms of Outer Join

21

- Left outer join: $r \bowtie\!\!\!\bowtie s$
 - ▣ If a tuple $t_r \in r$ doesn't match any tuple in s , result contains $\{ t_r, null, \dots, null \}$
 - ▣ If a tuple $t_s \in s$ doesn't match any tuple in r , it's excluded
- Right outer join: $r \bowtie\!\!\!\lrcorner s$
 - ▣ If a tuple $t_r \in r$ doesn't match any tuple in s , it's excluded
 - ▣ If a tuple $t_s \in s$ doesn't match any tuple in r , result contains $\{ null, \dots, null, t_s \}$

Forms of Outer Join (2)

22

□ Full outer join: $r \bowtie s$

- ▣ Includes tuples from r that don't match s , as well as tuples from s that don't match r

□ Summary:

$r =$

attr1	attr2
a	r1
b	r2
c	r3

$s =$

attr1	attr3
b	s2
c	s3
d	s4

$r \bowtie s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3

$r \Join s$

attr1	attr2	attr3
a	r1	null
b	r2	s2
c	r3	s3

$r \ltimes s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3
d	null	s4

$r \Join s$

attr1	attr2	attr3
a	r1	null
b	r2	s2
c	r3	s3
d	null	s4

Effects of *null* Values

23

- Introducing *null* values affects everything!
 - ▣ *null* means “unknown” or “nonexistent”
- Must specify effect on results when *null* is present
 - ▣ These choices are *somewhat* arbitrary...
 - ▣ (Read your database user’s manual! 😊)
- Arithmetic operations (+, −, *, /) involving *null* always evaluate to *null* (e.g. $5 + \text{null} = \text{null}$)
- Comparison operations involving *null* evaluate to unknown
 - ▣ *unknown* is a third truth-value
 - ▣ **Note:** Yes, even $\text{null} = \text{null}$ evaluates to *unknown*.

Boolean Operators and *unknown*

24

□ and

$\text{true} \wedge \text{unknown} = \text{unknown}$

$\text{false} \wedge \text{unknown} = \text{false}$

$\text{unknown} \wedge \text{unknown} = \text{unknown}$

□ or

$\text{true} \vee \text{unknown} = \text{true}$

$\text{false} \vee \text{unknown} = \text{unknown}$

$\text{unknown} \vee \text{unknown} = \text{unknown}$

□ not

$\neg \text{unknown} = \text{unknown}$

Relational Operations

25

- For each relational operation, need to specify behavior with respect to *null* and *unknown*
- Select: $\sigma_P(E)$
 - ▣ If P evaluates to *unknown* for a tuple, that tuple is excluded from result (i.e. definition of σ doesn't change)
- Natural join: $r \bowtie s$
 - ▣ Includes a Cartesian product, then a select
 - ▣ If a common attribute has a *null* value, tuples are excluded from join result
 - ▣ Why?
 - $null = (\text{anything})$ evaluates to *unknown*

Project and Set-Operations

26

- Project: $\Pi(E)$
 - ▣ Project operation must eliminate duplicates
 - ▣ *null* value is treated like any other value
 - ▣ Duplicate tuples containing *null* values are also eliminated
- Union, Intersection, and Difference
 - ▣ *null* values are treated like any other value
 - ▣ Set union, intersection, difference computed as expected
- These choices are somewhat arbitrary
 - ▣ *null* means “value is unknown or missing”...
 - ▣ ...but in these cases, two *null* values are considered equal.
 - ▣ Technically, two *null* values aren’t the same. (oh well)

Grouping and Aggregation

27

- In grouping phase:
 - ▣ *null* is treated like any other value
 - ▣ If two tuples have same values (including *null*) on the grouping attributes, they end up in same group
- In aggregation phase:
 - ▣ *null* values are removed from the input multiset before the aggregate function is applied!
 - Slightly different from arithmetic behavior; it keeps one *null* value from wiping out an aggregate computation.
 - ▣ If the aggregate function gets an empty multiset for input, the result is *null*...
 - ...except for **count**! In that case, **count** returns 0.

Generalized Projection, Outer Joins


28

- Generalized Projection operation:
 - ▣ A combination of simple projection and arithmetic operations
 - ▣ Easy to figure out from previous rules
- Outer joins:
 - ▣ Behave just like natural join operation, except for padding missing values with *null*

Back to Our Puzzle!

29

“How many people have completed each puzzle?”



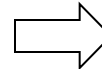
puzzle_name
altekruise
soma cube
puzzle box
clutch box

puzzle_list

person_name	puzzle_name
Alex	altekruise
Alex	soma cube
Bob	puzzle box
Carl	altekruise
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

- Use an outer join to include all puzzles, not just solved ones
puzzle_list ⋈ *completed*




puzzle_name	person_name
altekruise	Alex
soma cube	Alex
puzzle box	Bob
altekruise	Carl
soma cube	Bob
puzzle box	Carl
puzzle box	Alex
soma cube	Carl
clutch box	<i>null</i>

Counting the Solutions


30

- Now, use grouping and aggregation
 - ▣ Group on puzzle name
 - ▣ Count up the people!

puzzle_name $\mathcal{G}_{\text{count}(\text{person_name})(\text{puzzle_list} \bowtie \text{completed})}$



puzzle_name	person_name
altekruise	Alex
soma cube	Alex
puzzle box	Bob
altekruise	Carl
soma cube	Bob
puzzle box	Carl
puzzle box	Alex
soma cube	Carl
clutch box	<i>null</i>



puzzle_name	person_name
altekruise	Alex
altekruise	Carl
soma cube	Alex
soma cube	Bob
soma cube	Carl
puzzle box	Bob
puzzle box	Carl
puzzle box	Alex
clutch box	<i>null</i>

puzzle_name	
altekruise	2
soma cube	3
puzzle box	3
clutch box	0

Database Modification

31

- Often need to modify data in a database
- Can use assignment operator \leftarrow for this
- Operations:
 - ▣ $r \leftarrow r \cup E$ Insert new tuples into a relation
 - ▣ $r \leftarrow r - E$ Delete tuples from a relation
 - ▣ $r \leftarrow \Pi(r)$ Update tuples already in the relation
- Remember: r is a relation-variable
 - ▣ Assignment operator assigns a new relation-value to r
 - ▣ Hence, RHS expression may need to include existing version of r , to avoid losing unchanged tuples

Inserting New Tuples

32

- Inserting tuples simply involves a union:

$$r \leftarrow r \cup E$$

- E has to have correct arity

- Can specify actual tuples to insert:

$$completed \leftarrow completed \cup$$

$\{ ("Bob", "altekruise"), ("Carl", "clutch box") \}$

constant
relation

- Adds two new tuples to *completed* relation

- Can specify constant relations as a set of values

- Each tuple is enclosed with parentheses

- Entire set of tuples enclosed with curly-braces

Inserting New Tuples (2)

33

- Can also insert tuples generated from an expression
- Example:
 - “Dave is joining the puzzle club. He has done every puzzle that Bob has done.”
 - ▣ Find out puzzles that Bob has completed, then construct new tuples to add to *completed*

Inserting New Tuples (3)

34

- How to construct new tuples with name “Dave” and each of Bob’s puzzles?

- Could use a Cartesian product:

$$\{ ("Dave") \} \times \Pi_{puzzle_name}(\sigma_{person_name="Bob"}(completed))$$

- Or, use generalized projection with a constant:

$$\Pi_{\text{“Dave” as person_name, puzzle_name}}(\sigma_{person_name="Bob"}(completed))$$

- Add new tuples to *completed* relation:

$$completed \leftarrow completed \cup$$

$$\Pi_{\text{“Dave” as person_name, puzzle_name}}(\sigma_{person_name="Bob"}(completed))$$

Deleting Tuples

35

- Deleting tuples uses the $-$ operation:

$$r \leftarrow r - E$$

- Example:

Get rid of the “soma cube” puzzle.

puzzle_name
altekruise
soma cube
puzzle box

puzzle_list

Problem:

- *completed* relation references the *puzzle_list* relation
- To respect referential integrity constraints, should delete from *completed* first.

person_name	puzzle_name
Alex	altekruise
Alex	soma cube
Bob	puzzle box
Carl	altekruise
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

Deleting Tuples (2)

36

- *completed* references *puzzle_list*
 - *puzzle_name* is a key
 - *completed* shouldn't have any values for *puzzle_name* that don't appear in *puzzle_list*
 - Delete tuples from *completed* first.
 - Then delete tuples from *puzzle_list*.

$completed \leftarrow completed - \sigma_{puzzle_name = \text{"soma cube"}}(completed)$

$puzzle_list \leftarrow puzzle_list - \sigma_{puzzle_name = \text{"soma cube"}}(puzzle_list)$

Of course, could also write:

$completed \leftarrow \sigma_{puzzle_name \neq \text{"soma cube"}}(completed)$

Deleting Tuples (3)

37

- In the relational model, we have to think about foreign key constraints ourselves...
- Relational database systems take care of these things for us, automatically.
 - ▣ Will explore the various capabilities and options in a few weeks

Updating Tuples

38

- General form uses generalized projection:

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_n}(r)$$

- Updates all tuples in r

- Example:

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
A-319	New York	80
A-322	Los Angeles	275

account

“Add 5% interest to all bank account balances.”

$$account \leftarrow \Pi_{acct_id, branch_name, (balance*1.05)}(account)$$

- **Note:** Must include unchanged attributes too
- Otherwise you will change the schema of *account*

Updating *Some* Tuples

39

- Updating only *some* tuples is more verbose
 - ▣ Relation-variable is set to the *entire result* of the evaluation
 - ▣ Must include both updated tuples, and non-updated tuples, in result

- Example:

“Add 5% interest to accounts with a balance less than \$10,000.”

$$\text{account} \leftarrow \Pi_{\text{acct_id}, \text{branch_name}, (\text{balance} * 1.05)}(\sigma_{\text{balance} < 10000}(\text{account})) \cup \sigma_{\text{balance} \geq 10000}(\text{account})$$

Updating *Some* Tuples (2)

40

Another example:

“Add 5% interest to accounts with a balance less than \$10,000, and 6% interest to accounts with a balance of \$10,000 or more.”

$$\text{account} \leftarrow \Pi_{\text{acct_id}, \text{branch_name}, (\text{balance} * 1.05)}(\sigma_{\text{balance} < 10000}(\text{account})) \cup \\ \Pi_{\text{acct_id}, \text{branch_name}, (\text{balance} * 1.06)}(\sigma_{\text{balance} \geq 10000}(\text{account}))$$

- Don't forget to include any non-updated tuples in your update operations!

Relational Algebra Summary

41

- Very expressive query language for retrieving information from a relational database
 - ▣ Simple selection, projection
 - ▣ Computing correlations between relations using joins
 - ▣ Grouping and aggregation operations
- Can also specify changes to the contents of a relation-variable
 - ▣ Inserts, deletes, updates
- The relational algebra is a procedural query language
 - ▣ State a sequence of operations for computing a result

Relational Algebra Summary (2)

42

- Benefit of relational algebra is that it can be formally specified and reasoned about
- Drawback is that it is very verbose!
- Database systems usually provide much simpler query languages
 - ▣ Most popular *by far* is SQL, the Structured Query Language
- However, many databases use relational algebra-like operations internally!
 - ▣ Great for representing execution plans, due to its procedural nature

Next Time

43

- Transition from relational algebra to SQL
- Start working with “real” databases 😊