ITS
SEMANGAT
BARU

ITS
Institut
Teknologi
Sepuluh Nopember

RELATIONAL ALGEBRA

- 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

- 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

- 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

Generalized Projection

- Written as: $\Pi_{F_1, F_2, \ldots, F_n}(E)$
  - 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

Generalized Projection Example

- "Compute available credit for every credit account."

$$\Pi_{cred\_id,\ (limit\ -\ balance)\ \textbf{as}\ available\_credit}(credit\_acct)$$

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

credit_acct

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

Aggregate Functions

- 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

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

credit_acct

"Find the total amount owed to the credit company."

$$\mathcal{G}_{\text{sum(balance)}}(\text{credit\_acct})$$

4275

"Find the maximum available credit of any account."

$$\mathcal{G}_{\mathbf{max}(\mathit{available\_credit})}(\Pi_{(\mathit{limit} - \mathit{balance})\ \mathbf{as}\ \mathit{available\_credit}}(\mathit{credit\_acct}))$$

1150

Grouping and Aggregation

- Sometimes need to compute aggregates on a *per-item* basis
- The puzzle database:
  - *puzzle_list*(*puzzle_name*)
  - *completed*(*person_name*, *puzzle_name*)
- Examples:
  - "How many puzzles has each person completed?"

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

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

completed

Grouping and Aggregation (2)

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

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

completed

"How many puzzles has each person completed?"

$$_{\text{person\_name}}\mathcal{G}_{\textbf{count}(\text{puzzle\_name})}(\mathit{completed})$$

- 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)

$_{\text{person\_name}}\mathcal{G}_{\textbf{count}(\text{puzzle\_name})}(\text{completed})$

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

- 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	altekruse	350
Alex	soma cube	45
Bob	puzzle box	240
Carl	altekruse	285
Bob	soma cube	215
Alex	altekruse	290

completed_times

Distinct Values (2)

"How many puzzles has each person completed?"

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

completed_times

- Each puzzle appears multiple times now.

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

$_{\text{person\_name}}\mathcal{G}_{\textbf{count-distinc}(\text{puzzle\_name})}(\mathit{completed\_times})$

Eliminating Duplicates

- 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

- General form: ${}_{G_1, G_2, \ldots, G_n}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots, F_m(A_m)}(E)$
  - E evalutes to a relation
  - Leading G_i are attributes of E to group on
  - Each F_j is aggregate function applied to attribute A_j 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)

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

General Form of Aggregates (3)

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

One More Aggregation Example

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

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

completed

"How many people have completed each puzzle?"

$$_{\text{puzzle\_name}}\mathcal{G}_{\textbf{count}(\text{person\_name})}(\text{completed})$$

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

One More Aggregation Example

puzzle_name
Altekruse
soma cube
puzzle box
clutch box

puzzle_list

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

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

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

- **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

- Left outer join: r⟕s
  - If a tuple $t_r \in r$ doesn't match any tuple in s, result contains $\{ t_r, null, \ldots, null \}$
  - If a tuple $t_s \in s$ doesn't match any tuple in r, it's excluded
- Right outer join: $r ⟖ s$
  - 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 $\{ null, \ldots, null, t_s \}$

Forms of Outer Join (2)

- Full outer join: $r ⟗ s$
  - 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

$r ⟕ s$

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

$r ⟖ s$

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

$r ⟗ s$

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

Effects of *null* Values

- Introducing *null* values affects everything!
  - *null* means "unknown" or "nonexistent"
- Must specify effect on results when *null* is present
  - These choices are somewhat arbitrary…
- Arithmetic operations (+, –, *, /) involving *null* always evaluate to *null* (e.g. 5 + null = null)
- Comparison operations involving *null* evaluate to *unknown*
  - *unknown* is a third truth-value
  - Note: Yes, even *null* = *null* evaluates to unknown.

Relational Algebra Summary

- 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)

- 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

Thank You!