RELATIONAL ALGEBRA

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 2

Administrivia

- First assignment will be available today
  - Due next Thursday, October 9, 2:00 AM
- We have TAs:
  - Solomon Chang
  - Daniel Kong
  - Ryan Langman
  - Eric Pelz
  - Daniel Wang
- See Moodle for contact info and office hours
  - Can send questions to cs121tas@caltech.edu (Donnie + TAs)

Query Languages

- A query language specifies how to access the data in the database
- Different kinds of query languages:
  - Declarative languages specify what data to retrieve, but not how to retrieve it
  - Procedural languages specify what to retrieve, as well as the process for retrieving it
- Query languages often include updating and deleting data as well
- Also called data manipulation language (DML)

The Relational Algebra

- A procedural query language
- Comprised of relational algebra operations
- Relational operations:
  - Take one or two relations as input
  - Produce a relation as output
- Relational operations can be composed together
  - Each operation produces a relation
  - A query is simply a relational algebra expression
- Six "fundamental" relational operations
- Other useful operations can be composed from these fundamental operations

“Why is this useful?”

- SQL is only loosely based on relational algebra
- SQL is much more on the “declarative” end of the spectrum
- *Many* relational database implementations use relational algebra operations as basis for representing execution plans
  - Simple, clean, effective abstraction for representing how results will be generated
  - Relatively easy to manipulate for query optimization

Fundamental Relational Algebra Operations

- Six fundamental operations:

σ	select operation
Π	project operation
$\cup$	set-union operation
$-$	set-difference operation
$\times$	Cartesian product operation
ρ	rename operation

- Each operation takes one or two relations as input
- Produces another relation as output
- Important details:
  - What tuples are included in the result relation?
  - Any constraints on input schemas? What is schema of result?

Select Operation

- Written as: $\sigma_P(r)$
- P is the predicate for selection
  - P can refer to attributes in *r* (but no other relation!), as well as literal values
  - Can use comparison operators: $=, \neq, <, \leq, >, \geq$
  - Can combine multiple predicates using: $\wedge$ (and), $\vee$ (or), $\neg$ (not)
- *r* is the input relation
- Result relation contains all tuples in *r* for which P is true
- Result schema is identical to schema for *r*

Select Examples

Using the *account* relation:

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

"Retrieve all tuples for accounts in the Los Angeles branch."

$\sigma_{branch\_name=\text{"Los Angeles"}}(account)$

acct_id	branch_name	balance
A-318	Los Angeles	550
A-322	Los Angeles	275

"Retrieve all tuples for accounts in the Los Angeles branch, with a balance under $300."

$\sigma_{branch\_name=\text{"Los Angeles"} \wedge balance<300}(account)$

acct_id	branch_name	balance
A-322	Los Angeles	275

Project Operation

- Written as: $\Pi_{a,b,\ldots}(r)$
- Result relation contains only specified attributes of *r*
  - Specified attributes must actually be in schema of *r*
  - Result's schema only contains the specified attributes
  - Domains are same as source attributes' domains
- Important note:
  - Result relation may have fewer rows than input relation!
  - Why?
    - Relations are *sets* of tuples, not multisets

Project Example

Using the *account* relation:

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

"Retrieve all branch names that have at least one account."

$\Pi_{branch\_name}(account)$

branch_name
New York
Seattle
Los Angeles

- Result only has three tuples, even though input has five
- Result schema is just (*branch_name*)

Composing Operations

- Input can also be an expression that evaluates to a relation, instead of just a relation
- $\Pi_{acct\_id}(\sigma_{balance \geq 300}(account))$
  - Selects the account IDs of all accounts with a balance of \$300 or more
  - Input relation's schema is:

    $Account\_schema = (\underline{acct\_id}, branch\_name, balance)$
  - Final result relation's schema?
    - Just one attribute: ($acct\_id$)
- Distinguish between <u>base</u> and <u>derived</u> relations
  - *account* is a base relation
  - $\sigma_{balance \geq 300}(account)$ is a derived relation

Set-Union Operation

- Written as: $r \cup s$
- Result contains all tuples from *r* and *s*
  - Each tuple is unique, even if it's in both *r* and *s*
- Constraints on schemas for *r* and *s* ?
- *r* and *s* must have <u>compatible</u> schemas:
  - *r* and *s* must have same <u>arity</u>
    - (same number of attributes)
  - For each attribute i in *r* and *s*, *r*[i] must have the same domain as *s*[i]
  - (Our examples also generally have same attribute names, but not required! Arity and domains are what matter.)

Set-Union Example

- More complicated schema:

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

cust_name	acct_id
Johnson	A-318
Smith	A-322
Reynolds	A-319
Lewis	A-307
Reynolds	A-301

depositor

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

Set-Union Example (2)

- Find names of all customers that have either a bank account or a loan at the bank

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

cust_name	acct_id
Johnson	A-318
Smith	A-322
Reynolds	A-319
Lewis	A-307
Reynolds	A-301

depositor

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

Set-Union Example (3)

- Find names of all customers that have either a bank account or a loan at the bank
  - Easy to find the customers with an account:
    $\Pi_{cust\_name}(depositor)$
  - Also easy to find customers with a loan:
    $\Pi_{cust\_name}(borrower)$

cust_name
Johnson
Smith
Reynolds
Lewis

$\Pi_{cust\_name}(depositor)$

cust_name
Anderson
Jackson
Lewis
Smith

$\Pi_{cust\_name}(borrower)$

- Result is set-union of these expressions:
  $\Pi_{cust\_name}(depositor) \cup \Pi_{cust\_name}(borrower)$
  - Note that inputs have 8 tuples, but result has 6 tuples.

cust_name
Johnson
Smith
Reynolds
Lewis
Anderson
Jackson

Set-Difference Operation

- Written as: $r - s$
- Result contains tuples that are only in r, but not in s
  - Tuples in both r and s are excluded
  - Tuples only in s do not affect the result
- Constraints on schemas of r and s?
  - Schemas must be compatible
  - (Exactly like set-union.)

Set-Difference 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

cust_name	acct_id
Johnson	A-318
Smith	A-322
Reynolds	A-319
Lewis	A-307
Reynolds	A-301

depositor

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

"Find all customers that have an account but not a loan."

Set-Difference Example (2)

- Again, each component is easy
  - All customers that have an account:

    $\Pi_{cust\_name}(depositor)$
  - All customers that have a loan:

    $\Pi_{cust\_name}(borrower)$

cust_name
Johnson
Smith
Reynolds
Lewis

$\Pi_{cust\_name}(depositor)$

cust_name
Anderson
Jackson
Lewis
Smith

$\Pi_{cust\_name}(borrower)$

- Result is set-difference of these expressions

  $\Pi_{cust\_name}(depositor) - \Pi_{cust\_name}(borrower)$

cust_name
Johnson
Reynolds

Cartesian Product Operation

- Written as: $r \times s$
  - Read as "*r* cross *s*"
- No constraints on schemas of *r* and *s*
- Schema of result is *concatenation* of schemas for *r* and *s*
- If *r* and *s* have overlapping attribute names:
  - All overlapping attributes are included; none are eliminated
  - Distinguish overlapping attribute names by prepending the source relation's name
- Example:
  - Input relations: $r(a, b)$ and $s(b, c)$
  - Schema of $r \times s$ is $(a, r.b, s.b, c)$

Cartesian Product Operation (2)

- Result of $r \times s$
  - Contains *every* tuple in *r*, combined with *every* tuple in *s*
  - If *r* contains N_r tuples, and *s* contains N_s tuples, result contains $N_r \times N_s$ tuples
- Allows two relations to be compared and/or combined
  - If we want to correlate tuples in relation *r* with tuples in relation *s*…
  - Compute $r \times s$, then select out desired results with an appropriate predicate

Cartesian Product Example

- Compute result of *borrower* × *loan*

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

- Result will contain 4 × 4 = 16 tuples

Cartesian Product Example (2)

- Schema for borrower is:

  Borrower_schema = (*cust_name*, *loan_id*)

- Schema for loan is:

  Loan_schema = (*loan_id*, *branch_name*, *amount*)

- Schema for result of *borrower* × *loan* is:

  (*cust_name*, *borrower.loan_id*,
  loan.loan_id, *branch_name*, *amount*)

  - Overlapping attribute names are distinguished by including name of source relation

Cartesian Product Example (3)

Result:

cust_name	borrower. loan_id	loan. loan_id	branch_name	amount
Anderson	L-437	L-421	San Francisco	7500
Anderson	L-437	L-445	Los Angeles	2000
Anderson	L-437	L-437	Las Vegas	4300
Anderson	L-437	L-419	Seattle	2900
Jackson	L-419	L-421	San Francisco	7500
Jackson	L-419	L-445	Los Angeles	2000
Jackson	L-419	L-437	Las Vegas	4300
Jackson	L-419	L-419	Seattle	2900
Lewis	L-421	L-421	San Francisco	7500
Lewis	L-421	L-445	Los Angeles	2000
Lewis	L-421	L-437	Las Vegas	4300
Lewis	L-421	L-419	Seattle	2900
Smith	L-445	L-421	San Francisco	7500
Smith	L-445	L-445	Los Angeles	2000
Smith	L-445	L-437	Las Vegas	4300
Smith	L-445	L-419	Seattle	2900

Cartesian Product Example (4)

- Can use Cartesian product to associate related rows between two tables
  - …but, a lot of extra rows are included!

cust_name	borrower. loan_id	loan. loan_id	branch_name	amount
…	…	…	…	…
Jackson	L-419	L-437	Las Vegas	4300
Jackson	L-419	L-419	Seattle	2900
Lewis	L-421	L-421	San Francisco	7500
Lewis	L-421	L-445	Los Angeles	2000
…	…	…	…	…

- Combine Cartesian product with a select operation

$$\sigma_{borrower.loan\_id=loan.loan\_id}(borrower \times loan)$$

Cartesian Product Example (5)

- "Retrieve the names of all customers with loans at the Seattle branch."

cust_name	loan_id
Anderson	L-437
Jackson	L-419
Lewis	L-421
Smith	L-445

borrower

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

- Need both *borrower* and *loan* relations
- Correlate tuples in the relations using *loan_id*
- Then, computing result is easy.

Cartesian Product Example (6)

- Associate customer names with loan details, using Cartesian product and a select:

  $\sigma_{borrower.loan\_id=loan.loan\_id}(borrower \times loan)$

- Select out loans at Seattle branch:

  $\sigma_{branch\_name=\text{"Seattle"}}(\sigma_{borrower.loan\_id=loan.loan\_id}(borrower \times loan))$

  Simplify:

  $\sigma_{borrower.loan\_id=loan.loan\_id \wedge branch\_name=\text{"Seattle"}}(borrower \times loan)$

- Project results down to customer name:

  $\Pi_{cust\_name}(\sigma_{borrower.loan\_id=loan.loan\_id \wedge branch\_name=\text{"Seattle"}}(borrower \times loan))$

- Final result:

cust_name
Jackson

Rename Operation

- Results of relational operations are unnamed
  - Result has a schema, but the relation itself is unnamed
- Can give result a name using the rename operator
- Written as: $\rho_x(E)$
  - *E* is an expression that produces a relation
  - *E* can also be a named relation or a relation-variable
  - *x* is new name of relation
- More general form is: $\rho_{x(A_1,\, A_2,\, \ldots,\, A_n)}(E)$
  - Allows renaming of relation's attributes
  - Requirement: *E* has arity *n*

Scope of Renamed Relations

- Rename operation ρ only applies within a specific relational algebra expression
  - This does not create a new relation-variable!
  - The new name is only visible to enclosing relational-algebra expressions
- Rename operator is used for two main purposes:
  - Allow a derived relation and its attributes to be referred to by enclosing relational-algebra operations
  - Allow a base relation to be used multiple ways in one query
    - $r \times \rho_s(r)$
- In other words, rename operation ρ is used to resolve ambiguities within a specific relational algebra expression

Rename Example

- "Find the ID of the loan with the largest amount."

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan

  - Hard to find the loan with the largest amount!
    - (At least, with the tools we have so far…)
  - Much easier to find all loans that have an amount *smaller* than some other loan
  - Then, use set-difference to find the largest loan

Rename Example (2)

- How to find all loans with an amount smaller than some other loan?
  - Use Cartesian Product of *loan* with itself:

    loan × *loan*
  - Compare each loan's amount to all other loans
- Problem: Can't distinguish between attributes of left and right *loan* relations!
- Solution: Use rename operation

  $loan \times \rho_{test}(loan)$
  - Now, right relation is named *test*

Rename Example (3)

- Find IDs of all loans with an amount smaller than some other loan:

  $\Pi_{loan.loan\_id}(\sigma_{loan.amount<test.amount}(loan \times \rho_{test}(loan)))$

- Finally, we can get our result:

  $\Pi_{loan\_id}(loan)$ –

  $\Pi_{loan.loan\_id}(\sigma_{loan.amount<test.amount}(loan \times \rho_{test}(loan)))$

loan_id
L-421

- What if multiple loans have max value?
  - All loans with max value appear in result.

Additional Relational Operations

- The fundamental operations are sufficient to query a relational database…
- Can produce some large expressions for common operations!
- Several additional operations, defined in terms of fundamental operations:

$\cap$	set-intersection
$\bowtie$	natural join
$\div$	division
$\leftarrow$	assignment

Set-Intersection Operation

- Written as: $r \cap s$
- $r \cap s = r - (r - s)$

  $r - s$ = the rows in r, but not in s

  $r - (r - s)$ = the rows in both r and s
- Relations must have compatible schemas
- Example: find all customers with both a loan and a bank account

  $\Pi_{cust\_name}(borrower) \cap \Pi_{cust\_name}(depositor)$

Natural Join Operation

- Most common use of Cartesian product is to correlate tuples with same key-values
  - Called a <u>join</u> operation
- The <u>natural join</u> is a shorthand for this operation
- Written as: $r \bowtie s$
  - *r* and *s* must have common attributes
  - The common attributes are usually a key for *r* and/or *s*, but certainly don't have to be

Natural Join Definition

- For two relations $r(R)$ and $s(S)$
- Attributes used to perform natural join:

  $R \cap S = \{A_1, A_2, \ldots, A_n\}$
- Formal definition:

  $r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1=s.A_1 \wedge r.A_2=s.A_2 \wedge \ldots \wedge r.A_n=s.A_n}(r \times s))$
  - r and s are joined on their common attributes
  - Result is projected so that common attributes only appear once

Natural Join Example

- Simple example:

  "Find the names of all customers with loans."

- Result:

  $\Pi_{cust\_name}(\sigma_{borrower.loan\_id=loan.loan\_id}(borrower \times loan))$

- Rewritten with natural join:

  $\Pi_{cust\_name}(borrower \bowtie loan)$

Natural Join Characteristics

- Very common to compute joins across multiple tables
- Example: *customer* ⋈ *borrower* ⋈ *loan*
- Natural join operation is associative:
  - (*customer* ⋈ *borrower*) ⋈ *loan* is equivalent to *customer* ⋈ (*borrower* ⋈ *loan*)
- Note:
  - Even though these expressions are equivalent, order of join operations can dramatically affect query cost!
  - (Keep this in mind for later…)

Division Operation

- Binary operator: $r \div s$
- Implements a "for each" type of query
  - "Find all rows in *r* that have one row corresponding to each row in *s*."
  - Relation *r* divided by relation *s*
- Easiest to illustrate with an example:
- Puzzle Database

  puzzle_list(*puzzle_name*)
    - Simple list of puzzles by name

  completed(*person_name*, *puzzle_name*)
    - Records which puzzles have been completed by each person

Puzzle Database

"Who has solved every puzzle?"

- Need to find every person in *completed* that has an entry for every puzzle in *puzzle_list*
- Divide *completed* by *puzzle_list* to get answer:

  $completed \div puzzle\_list =$

person_name
Alex
Carl

- Only Alex and Carl have completed every puzzle in *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

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

Puzzle Database (2)

"Who has solved every puzzle?"

completed ÷ *puzzle_list* =

person_name
Alex
Carl

- Very reminiscent of integer division
  - Result relation contains tuples from *completed* that are evenly divided by *puzzle_name*
- Several other kinds of relational division operators
  - e.g. some can compute "remainder" of the division operation

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

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

Division Operation

For $r(R) \div s(S)$

- Required: $S \subseteq R$
  - All attributes in S must also be in R
- Result has schema $R - S$
  - Result has attributes that are in R but not also in S
  - (Probably best if $S \subset R$…)
- Every tuple t in result satisfies these conditions:

  $t \in \Pi_{R-S}(r)$

  $\langle \forall\ t_s \in s : \exists\ t_r \in r : t_r[S] = t_s[S] \cap t_r[R{-}S] = t \ \rangle$

  - Every tuple in the result has a row in r corresponding to every row in s

Puzzle Database

For *completed* ÷ *puzzle_list*

- Schemas are compatible
- Result has schema (*person_name*)
  - Attributes in *completed* schema, but not also in *puzzle_list* schema

person_name
Alex
Carl

completed ÷ *puzzle_list*

- Every tuple *t* in result satisfies these conditions:

$$t \in \Pi_{R-S}(r)$$

$$\langle\ \forall\ t_s \in s : \exists\ t_r \in r : t_r[S] = t_s[S] \cap t_r[R-S] = t\ \rangle$$

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 = *r*

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list = *s*

Division Operation

- Not provided natively in most SQL databases
  - Rarely needed!
  - Easy enough to implement in SQL, if needed
- Will see it in the homework assignments, and on the midterm… ☺
  - Often a very nice shortcut for more involved queries

Relation-Variables

- Recall: relation variables refer to a specific relation
  - A specific set of tuples, with a particular schema
- Example: *account* relation

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

  - *account* is actually technically a relation-variable, as are all our named relations so far

Assignment Operation

- Can assign a relation-value to a relation-variable
- Written as: *relvar* $\leftarrow$ *E*
  - *E* is an expression that evaluates to a relation
- Unlike ρ, the name *relvar* persists in the database
- Often used for temporary relation-variables:

$$temp1 \leftarrow \Pi_{R-S}(r)$$
$$temp2 \leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r))$$
$$result \leftarrow temp1 - temp2$$

  - Query evaluation becomes a sequence of steps
  - (This is an implementation of the $\div$ operator)
- Can also use to represent data updates
  - More about updates next time…