RELATIONAL ALGEBRA

Revisit: Data Model

cust_id	cust_name	ssn
23-652	Joe Smith	330-25-8822
15-202	Ellen Jones	221-30-6551
23-521	Dave Johnson	005-81-2568
. . .	. . .	. . .

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

cust_id	acct_id
15-202	A-301
23-521	A-307
23-652	A-318
. . .	. . .

cust_id	branch_name	balance
23-652	Joe Smith	330-25-8822
15-202	Ellen Jones	221-30-6551
23-521	Dave Johnson	005-81-2568
15-202	Albert Stevens	450-22-5869
. . .	. . .	. . .

n

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 databases use relational algebra operations 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
  - ∪ set-union operation
  - – set-difference operation
  - × 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: $\boldsymbol{\sigma_P(r)}$

- r is the input relation
- 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)
- 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."

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

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."

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

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 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 branch names that have at least one account."

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

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 = (*acct_id*, *branch_name*, *balance*)
  - Final result relation's schema?
    - Just one attribute: (*acct_id*)
- Distinguish between base and derived relations
  - *account* is a base relation
  - $\sigma_{balance \geq 300}(account)$ is a derived relation

Set-Union Operation

Written as: **r U 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 compatible schemas:
  - *r* and *s* must have same **arity** (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

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

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

- 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:

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

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

cust_name
Anderson
Jackson
Lewis
Smith

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

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* are also excluded
- Constraints on schemas of *r* and *s*?
  - Schemas must be compatible
  - (Exactly like set-union)

Set-Difference Example

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

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-Difference Example

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

- Easy to find the customers with an account:

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

- Also easy to find customers with a loan:

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

- Result is set-difference of these expressions:

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

cust_name
Johnson
Smith
Reynolds
Lewis

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

cust_name
Anderson
Jackson
Lewis
Smith

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

cust_name
Johnson
Reynolds

Cartesian Product Operation

Written as: $\boldsymbol{r \times s}$

- 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

- 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

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

- Compute result of *borrower* × *loan*
  - Result will contain 4 × 4 = 16 tuples

Cartesian Product Example

- Schema for borrower is:

  Borrower_schema = (*cust_name*, *loan_id*)

- Schema for loan is:

  Loan_schema = (<u>*loan_id*</u>, *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

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

- 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

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

Cartesian Product Example

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

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

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

Cartesian Product Example

- 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:

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

- Final result:

cust_name
Jackon

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: $\rho_x(E)$ (Greek rho, not lowercase "P")
  - 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

    $r \times \rho_s(\mathrm{r})$
- 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

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

    $loan \times loan$

  - 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

- 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:

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

- 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 the same key-values
  - Called a join operation
- The natural join 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:
- $r \bowtie s = \Pi_{R \cup S}\left(\sigma_{r.A_1=s.A_1 \land r.A_2=s.A_2 \land \ldots \land r.A_n=s.A_n}(r \times s)\right)$
  - r and s are joined using an equality condition based 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:

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

- Rewritten with natural join:

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

Natural Join Characteristics

- Very common to compute joins across multiple tables
- Example: $customer \bowtie borrower \bowtie loan$
- Natural join operation is associative:

  $(customer \bowtie borrower) \bowtie loan$ is equivalent to $customer \bowtie (borrower \bowtie loan)$

- 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*
- Not provided natively in most SQL databases
  - Rarely needed!
  - Easy enough to implement in SQL, if needed

Division Operation

- 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

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

"Who has solved every puzzle?"

$completed \div puzzle\_list =$

person_name
Alex
Carl

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 a relation
- Unlike ρ, the name *relvar* persists in the database
- Often used for temporary relation-variables:

$$temp_1 \longleftarrow \Pi_{R-S}(r)$$
$$temp_2 \longleftarrow \Pi_{R-S}((temp_1 \times s) - \Pi_{R-S,S}(r))$$
$$result \longleftarrow temp_1 - temp_2$$

  - Query evaluation becomes a sequence of steps
  - (This is an implementation of the ÷ operator)
- Can also use assignment operation to modify data
  - More about updates next time…

Guest (GuestID, GuestName, GuestAddress)

Hotel (HotelID, HotelName, HotelAddress)

Room (RoomID, HotelID, Type, Price)

Booking (HotelID, GuestID, RoomID, DateFrom, DateTo)

For each question, write the relational algebra that will fulfill the request.

1. List full details of all hotels.
2. List full details of all hotels in Surabaya.
3. List the number of rooms in each hotel in Surabaya.
4. List the names and addresses of all guests in Surabaya.
5. List all double or family rooms with a price below Rp 1,500,000 per night.

Store(<u>StoreID</u>, StoreName, address)

Product(<u>ProductID</u>, ProductName, colour)

Catalog(<u>StoreID</u>, <u>ProductID</u>, price)

For each question, write the relational algebra that will fulfill the request.

1. Find the names of all black products.
2. Find all prices for products that are black or white.
3. Find the StoreID of all stores who sell a product that is black or white.
4. Find the StoreID of all stores who sell a product that is black and white.
5. Find the names of all stores who supply a product that is black or white.

Thank You!