



Lecture 9 of 42

Introduction to Relational Calculi Notes: MP2, Datalog Preview

Monday, 11 February 2008

William H. Hsu
Department of Computing and Information Sciences, KSU

KSOL course page: <http://snipurl.com/va60>
Course web site: <http://www.kddresearch.org/Courses/Spring-2008/CIS560>
Instructor home page: <http://www.cis.ksu.edu/~bhsu>

Reading for Next Class:
Rest of Chapter 5, Silberschatz *et al.*, 5th edition



Other Relational Languages: Review

- Tuple Relational Calculus
- Domain Relational Calculus
- Query-by-Example (QBE)
- Datalog





Tuple Relational Calculus

- A nonprocedural query language, where each query is of the form $\{t \mid P(t)\}$
- It is the set of all tuples t such that predicate P is true for t
- t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A
- $t \in r$ denotes that tuple t is in relation r
- P is a *formula* similar to that of the predicate calculus



Banking Example

- *branch* (branch_name, branch_city, assets)
- *customer* (customer_name, customer_street, customer_city)
- *account* (account_number, branch_name, balance)
- *loan* (loan_number, branch_name, amount)
- *depositor* (customer_name, account_number)
- *borrower* (customer_name, loan_number)



Example Queries

- Find the names of all customers who have an account at all branches located in Brooklyn:

$$\{t \mid \exists r \in \text{customer} (t[\text{customer_name}] = r[\text{customer_name}]) \wedge$$

$$(\forall u \in \text{branch} (u[\text{branch_city}] = \text{"Brooklyn"} \Rightarrow$$

$$\exists s \in \text{depositor} (t[\text{customer_name}] = s[\text{customer_name}]$$

$$\wedge \exists w \in \text{account} (w[\text{account_number}] = s[\text{account_number}]$$

$$\wedge (w[\text{branch_name}] = u[\text{branch_name}]))))\}$$

Handwritten notes: "the answer" points to the result set. "branch", "customer", "depositor", "account" are labeled next to their respective table names in the query. A circled '3' is also present.



Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example, $\{t \mid \neg t \in r\}$ results in an infinite relation if the domain of any attribute of relation r is infinite
- To guard against the problem, we restrict the set of allowable expressions to safe expressions.
- An expression $\{t \mid P(t)\}$ in the tuple relational calculus is *safe* if every component of t appears in one of the relations, tuples, or constants that appear in P
 - * NOTE: this is more than just a syntax condition.
 - ⇒ E.g. $\{t \mid t[A] = 5 \vee \text{true}\}$ is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in P .



Domain Relational Calculus

- A nonprocedural query language equivalent in power to the tuple relational calculus
- Each query is an expression of the form:

$$\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$$

- * x_1, x_2, \dots, x_n represent domain variables
- * P represents a formula similar to that of the predicate calculus



Example Queries

- Find the *loan_number*, *branch_name*, and *amount* for loans of over \$1200

$$\{ \langle l, b, a \rangle \mid \langle l, b, a \rangle \in \text{loan} \wedge a > 1200 \}$$

- Find the names of all customers who have a loan of over \$1200

$$\{ \langle c \rangle \mid \exists l, b, a (\langle c, l \rangle \in \text{borrower} \wedge \langle l, b, a \rangle \in \text{loan} \wedge a > 1200) \}$$

- Find the names of all customers who have a loan from the Perryridge branch and the loan amount:

$$\{ \langle c, a \rangle \mid \exists l (\langle c, l \rangle \in \text{borrower} \wedge \langle l, b, a \rangle \in \text{loan} \wedge b = \text{"Perryridge"}) \}$$

$$\{ \langle c, a \rangle \mid \exists l (\langle c, l \rangle \in \text{borrower} \wedge \langle l, \text{"Perryridge"}, a \rangle \in \text{loan}) \}$$



Example Queries

- Find the names of all customers having a loan, an account, or both at the Perryridge branch:

$$\{ \langle c \rangle \mid \exists l (\langle c, l \rangle \in \text{borrower} \wedge \exists b, a (\langle b, a \rangle \in \text{loan} \wedge b = \text{"Perryridge"}) \vee \exists a (\langle c, a \rangle \in \text{depositor} \wedge \exists b, n (\langle b, n \rangle \in \text{account} \wedge b = \text{"Perryridge"})) \}$$

- Find the names of all customers who have an account at all branches located in Brooklyn:

$$\{ \langle c \rangle \mid \exists s, n (\langle c, s, n \rangle \in \text{customer}) \wedge \forall x, y, z (\langle x, y, z \rangle \in \text{branch} \wedge y = \text{"Brooklyn"}) \Rightarrow \exists a, b (\langle a, y, b \rangle \in \text{account} \wedge \langle c, a \rangle \in \text{depositor}) \}$$



Safety of Expressions

The expression:

$$\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$$

is safe if all of the following hold:

- All values that appear in tuples of the expression are values from $\text{dom}(P)$ (that is, the values appear either in P or in a tuple of a relation mentioned in P).
- For every "there exists" subformula of the form $\exists x (P_1(x))$, the subformula is true if and only if there is a value of x in $\text{dom}(P_1)$ such that $P_1(x)$ is true.
- For every "for all" subformula of the form $\forall x (P_1(x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from $\text{dom}(P_1)$.



Datalog

- Basic Structure
- Syntax of Datalog Rules
- Semantics of Nonrecursive Datalog
- Safety
- Relational Operations in Datalog
- Recursion in Datalog
- The Power of Recursion



Basic Structure $h \leftarrow q_1, q_2, \dots, q_n$

- Prolog-like logic-based language that allows recursive queries; based on first-order logic.
- A Datalog program consists of a set of *rules* that define views.
- Example: define a view relation *v1* containing account numbers and balances for accounts at the Perryridge branch with a balance of over \$700.

$v1(A, B) :- account(A, \text{"Perryridge"}, B), B > 700.$

- Retrieve the balance of account number "A-217" in the view relation *v1*.

$? v1(\text{"A-217"}, B).$

- To find account number and balance of all accounts in *v1* that have a balance greater than 800

$? v1(A, B), B > 800$



Example Queries

- Each rule defines a set of tuples that a view relation must contain.
 - * E.g. $v1(A, B) :- account(A, \text{"Perryridge"}, B), B > 700$ is read as

for all A, B

if $(A, \text{"Perryridge"}, B) \in account$ and $B > 700$ *body*

then $(A, B) \in v1$ *head*

- The set of tuples in a view relation is then defined as the union of all the sets of tuples defined by the rules for the view relation.
- Example:

$interest_rate(A, 5) :- account(A, N, B), B < 10000$

$interest_rate(A, 6) :- account(A, N, B), B \geq 10000$



Negation in Datalog

- Define a view relation c that contains the names of all customers who have a deposit but no loan at the bank:

$c(N) :- depositor(N, A), \text{not } is_borrower(N).$

$is_borrower(N) :- borrower(N, L).$

- NOTE: using **not** $borrower(N, L)$ in the first rule results in a different meaning, namely there is some loan L for which N is not a borrower.

- * To prevent such confusion, we require all variables in negated "predicate" to also be present in non-negated predicates



Named Attribute Notation

- Datalog rules use a positional notation that is convenient for relations with a small number of attributes
- It is easy to extend Datalog to support named attributes.
 - * E.g., *v1* can be defined using named attributes as
v1 (*account_number* *A*, *balance* *B*) :-
account (*account_number* *A*, *branch_name* "Perryridge", *balance* *B*),
B > 700.



Formal Syntax and Semantics of Datalog

- We formally define the syntax and semantics (meaning) of Datalog programs, in the following steps
 1. We define the syntax of predicates, and then the syntax of rules
 2. We define the semantics of individual rules
 3. We define the semantics of non-recursive programs, based on a layering of rules
 4. It is possible to write rules that can generate an infinite number of tuples in the view relation. To prevent this, we define what rules are "safe". Non-recursive programs containing only safe rules can only generate a finite number of answers.
 5. It is possible to write recursive programs whose meaning is unclear. We define what recursive programs are acceptable, and define their meaning.





Layering of Rules

- Define the interest on each account in Perryridge

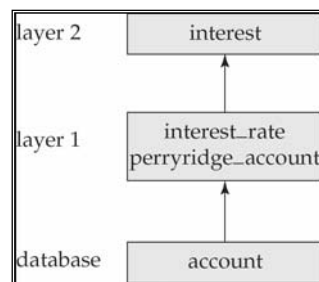
$$\text{interest}(A, I) :- \text{perryridge_account}(A, B),$$

$$\text{interest_rate}(A, R), I = B * R / 100.$$

$$\text{perryridge_account}(A, B) :- \text{account}(A, \text{"Perryridge"}, B).$$

$$\text{interest_rate}(A, 5) :- \text{account}(N, A, B), B < 10000.$$

$$\text{interest_rate}(A, 6) :- \text{account}(N, A, B), B \geq 10000.$$
- Layering of the view relations



Layering Rules (Cont.)

Formally:

- A relation is a layer 1 if all relations used in the bodies of rules defining it are stored in the database.
- A relation is a layer 2 if all relations used in the bodies of rules defining it are either stored in the database, or are in layer 1.
- A relation p is in layer $i + 1$ if
 - * it is not in layers 1, 2, ..., i
 - * all relations used in the bodies of rules defining a p are either stored in the database, or are in layers 1, 2, ..., i



Semantics of a Program

Let the layers in a given program be $1, 2, \dots, n$. Let \mathfrak{R}_i denote the set of all rules defining view relations in layer i .

- Define I_0 = set of facts stored in the database.
- Recursively define $I_{i+1} = I_i \cup \text{infer}(\mathfrak{R}_{i+1}, I_i)$
- The set of facts in the view relations defined by the program (also called the semantics of the program) is given by the set of facts I_n corresponding to the highest layer n .

Note: Can instead define semantics using view expansion like in relational algebra, but above definition is better for handling extensions such as recursion.



Safety

- It is possible to write rules that generate an infinite number of answers.

$$gt(X, Y) :- X > Y$$

$$not_in_loan(B, L) :- \text{not } loan(B, L)$$

To avoid this possibility Datalog rules must satisfy the following conditions.

- * Every variable that appears in the head of the rule also appears in a non-arithmetic positive literal in the body of the rule.
 - ⇒ This condition can be weakened in special cases based on the semantics of arithmetic predicates, for example to permit the rule
- * Every variable appearing in a negative literal in the body of the rule also appears in some positive literal in the body of the rule.





Relational Operations in Datalog

- Project out attribute *account_name* from *account*.
 $query(A) :- account(A, N, B).$
- Cartesian product of relations r_1 and r_2 .
 $query(X_1, X_2, \dots, X_n, Y_1, Y_2, \dots, Y_m) :-$
 $r_1(X_1, X_2, \dots, X_n), r_2(Y_1, Y_2, \dots, Y_m).$
- Union of relations r_1 and r_2 .
 $query(X_1, X_2, \dots, X_n) :- r_1(X_1, X_2, \dots, X_n),$
 $query(X_1, X_2, \dots, X_n) :- r_2(X_1, X_2, \dots, X_n),$
- Set difference of r_1 and r_2 .
 $query(X_1, X_2, \dots, X_n) :- r_1(X_1, X_2, \dots, X_n),$
 $not\ r_2(X_1, X_2, \dots, X_n),$



Recursion in Datalog

- Suppose we are given a relation
 $manager(X, Y)$
 containing pairs of names X, Y such that Y is a manager of X (or equivalently, X is a direct employee of Y).
- Each manager may have direct employees, as well as indirect employees
 - * Indirect employees of a manager, say Jones, are employees of people who are direct employees of Jones, or recursively, employees of people who are indirect employees of Jones
- Suppose we wish to find all (direct and indirect) employees of manager Jones. We can write a recursive Datalog program.
 $empl_jones(X) :- manager(X, Jones).$
 $empl_jones(X) :- manager(X, Y), empl_jones(Y).$



Example of Datalog-FixPoint Iteration

<i>employee_name</i>	<i>manager_name</i>
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Iteration number	Tuples in <i>empl_jones</i>
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)



A More General View

- Create a view relation *empl* that contains every tuple (X, Y) such that X is directly or indirectly managed by Y .

$empl(X, Y) :- manager(X, Y).$

$empl(X, Y) :- manager(X, Z), empl(Z, Y)$

- Find the direct and indirect employees of Jones.

$? empl(X, \text{"Jones"}).$

- Can define the view *empl* in another way too:

$empl(X, Y) :- manager(X, Y).$

$empl(X, Y) :- empl(X, Z), manager(Z, Y).$



The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
 - * Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of manager with itself
 - ⇒ This can give only a fixed number of levels of managers
 - ⇒ Given a program we can construct a database with a greater number of levels of managers on which the program will not work



Recursion in SQL

- Starting with SQL:1999, SQL permits recursive view definition
- E.g. query to find all employee-manager pairs

```
with recursive empl (emp, mgr) as (  
    select emp, mgr  
    from manager  
    union  
    select manager.emp, empl.mgr  
    from manager, empl  
    where manager.mgr = empl.emp )  
select *  
from empl
```





Query-by-Example (QBE)

- Basic Structure
- Queries on One Relation
- Queries on Several Relations
- The Condition Box
- The Result Relation
- Ordering the Display of Tuples
- Aggregate Operations
- Modification of the Database



QBE — Basic Structure

- A graphical query language which is based (roughly) on the domain relational calculus
- **Two dimensional syntax** – system creates templates of relations that are requested by users
- Queries are expressed “by example”





QBE — Basic Structure

- A graphical query language which is based (roughly) on the domain relational calculus
- **Two dimensional syntax** – system creates templates of relations that are requested by users
- Queries are expressed “by example”



QBE Skeleton Tables for the Bank Example

branch	branch-name	branch-city	assets

customer	customer-name	customer-street	customer-city

loan	loan-number	branch-name	amount





Microsoft Access QBE

- Microsoft Access supports a variant of QBE called Graphical Query By Example (GQBE)
- GQBE differs from QBE in the following ways
 - * Attributes of relations are listed vertically, one below the other, instead of horizontally
 - * Instead of using variables, lines (links) between attributes are used to specify that their values should be the same.
 - ⇒ Links are added automatically on the basis of attribute name, and the user can then add or delete links
 - ⇒ By default, a link specifies an inner join, but can be modified to specify outer joins.
 - * Conditions, values to be printed, as well as group by attributes are all specified together in a box called the **design grid**



Example Query in Microsoft Access QBE

- Example query: Find the *customer_name*, *account_number* and *balance* for all accounts at the Perryridge branch

Field:	customer_name	account_number	balance	branch_name
Table:	depositor	account	account	account
Sort:				
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Criteria:				"Perryridge"
or:				

