Chapter 5: Other Relational Languages

- Query-by-Example (QBE)
- Quel
- Datalog

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”

Skeleton Tables

branch	*branch-name*	*branch-city*	*assets*

customer	*customer-name*	*customer-street*	*customer-city*

loan	*branch-name*	*loan-number*	*amount*

Skeleton Tables (Cont.)

borrower	*customer-name*	*loan-number*

account	*branch-name*	*account-number*	*balance*

depositor	*customer-name*	*account-number*

Queries on One Relation

- Find all loan numbers at the Perryridge branch.

loan	*branch-name*	*loan-number*	*amount*
	Perryridge	P._*x*	

  - _*x* is a variable (optional)
  - P. means print (display)
  - duplicates are removed

loan	*branch-name*	*loan-number*	*amount*
	Perryridge	P.ALL.	

  - duplicates are not removed

Queries on One Relation (Cont.)

- Display full details of all loans
  - Method 1:

loan	*branch-name*	*loan-number*	*amount*
	P._x	P._y	P._z

  - Method 2: shorthand notation

loan	*branch-name*	*loan-number*	*amount*
P.			

- Find the loan number of all loans with a loan amount of more than $700.

loan	*branch-name*	*loan-number*	*amount*
		P.	$>$700

Queries on One Relation (Cont.)

- Find the loan numbers of all loans made jointly to Smith and Jones.

borrower	*customer-name*	*loan-number*
	"Smith"	P._*x*
	"Jones"	_*x*

- Find the loan numbers of all loans made to Smith, Jones or both.

borrower	*customer-name*	*loan-number*
	"Smith"	P._*x*
	"Jones"	P._*y*

Queries on Several Relations

- Find the names of all customers who have a loan from the Perryridge branch.

loan	*branch-name*	*loan-number*	*amount*
	Perryridge	_*x*	

borrower	*customer-name*	*loan-number*
	P._*y*	_*x*

Queries on Several Relations (Cont.)

- Find the names of all customers who have both an account and a loan at the bank.

depositor	*customer-name*	*account-number*
	P._*x*	

borrower	*customer-name*	*loan-number*
	_*x*	

Queries on Several Relations (Cont.)

- Find the names of all customers who have an account at the bank, but do not have a loan from the bank.

depositor	*customer-name*	*account-number*
	P._*x*	

borrower	*customer-name*	*loan-number*
$\neg$	_*x*	

$\neg$ means "there does not exist"

Queries on Several Relations

- Find all customers who have at least two accounts.

depositor	*customer-name*	*account-number*
	P._*x*	_*y*
	_*x*	¬ _*y*

¬ means "not equal to"

The Condition Box

- Allows the expression of constraints on domain variables that are either inconvenient or impossible to express within the skeleton tables.
- Find all account numbers with a balance between $1,300 and $2,000 but not exactly $1,500.

account	*branch-name*	*account-number*	*balance*
		P.	_*x*

conditions
$\_x = (\geq 1300$ **and** ≤ 2000 **and** $\neg$ $1500)$

The Result Relation

- Find the *customer-name*, *account-number*, and *balance* for all customers who have an account at the Perryridge branch.
  - We need to:
    - Join *depositor* and *account*.
    - Project *customer-name*, *account-number*, and *balance*.
  - To accomplish this we:
    - Create a skeleton table, called *result*, with attributes *customer-name*, *account-number*, and *balance*.
    - Write the query.

The Result Relation (Cont.)

- The resulting query is:

branch-name	*account-number*	*balance*
Perryridge	*_y*	*_z*

depositor	*customer-name*	*account-number*
	_x	*_y*

result	*customer-name*	*account-number*	*balance*
P.	*_x*	*_y*	*_z*

Ordering the Display of Tuples

- AO = ascending order; DO = descending order.
  When sorting on multiple attributes, the sorting order is specified by including with each sort operator (AO or DO) an integer surrounded by parentheses.

- List all account numbers at the Perryridge branch in ascending alphabetic order with their respective account balances in descending order.

account	*branch-name*	*account-number*	*balance*
	Perryridge	P.AO(1).	P.DO(2).

Aggregate Operations

- The aggregate operators are AVG, MAX, MIN, SUM,and CNT
- The above operators must always be postfixed with "ALL." (e.g., SUM.ALL.or AVG.ALL._*x*).
- Find the total balance of all the accounts maintained at the Perryridge branch.

account	*branch-name*	*account-number*	*balance*
	Perryridge		P.SUM.ALL.

Aggregate Operations (Cont.)

- Find the total number of customers having an account at the bank.

depositor	*customer-name*	*account-number*
	P.CNT.UNQ.ALL.	

Note: UNQ is used to specify that we want to eliminate duplicates.

Query Examples

- Find the average balance at each branch.

account	*branch-name*	*account-number*	*balance*
	P.G.		P.AVG.ALL._*x*

  Note:

  - The "G" in "P.G" is analogous to SQL's **group by** construct
  - The "ALL" in the "P.AVG.ALL" entry in the *balance* column ensures that all balances are considered

- Find the average account balance at only those branches where the average account balance is more than $1,200. Add the condition box:

conditions
AVG.ALL._$x > 1200$

Query Example

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

depositor	*customer-name*	*account-number*
	P.G._*x*	_*y*

account	*branch-name*	*account-number*	*balance*
	CNT.UNQ.ALL._*z*	_*y*	

branch	*branch-name*	*branch-city*	*assets*
	_*z*	Brooklyn	
	_*w*	Brooklyn	

Query Example (Cont.)

conditions
CNT.UNQ.ALL._z = CNT.UNQ.ALL._w

- CNT.UNQ.ALL._w specifies the number of distinct branches in Brooklyn.
- CNT.UNQ.ALL._z specifies the number of distinct branches in Brooklyn at which customer x has an account.

Modification of the Database – Deletion

- Deletion of tuples from a relation is expressed by use of a D. command. In the case where we delete information in only some of the columns, null values, specified by $-$, are inserted.
- Delete customer Smith

customer	*customer-name*	*customer-street*	*customer-city*
D.	Smith		

- Delete the *branch-city* value of the branch whose name is “Perryridge”.

branch	*branch-name*	*branch-city*	*assets*
	Perryridge	D.	

Deletion Query Examples

- Delete all loans with a loan amount between \$1300 and \$1500.

loan	*branch-name*	*loan-number*	*amount*
D.		$\_y$	$\_x$

borrower	*customer-name*	*loan-number*
D.		$\_y$

conditions
$\_x$ = ($\geq$ 1300 **and** $\leq$ 1500)

Deletion Query Examples (Cont.)

- Delete all accounts at branches located in Brooklyn.

account	*branch-name*	*account-number*	*balance*
D.	_*x*	_*y*	

depositor	*customer-name*	*account-number*
D.		_*y*

branch	*branch-name*	*branch-city*	*assets*
	_*x*	Brooklyn	

Modification of the Database – Insertion

- Insertion is done by placing the I. operator in the query expression.
- Insert the fact that account A-9732 at the Perryridge branch has a balance of $700.

account	*branch-name*	*account-number*	*balance*
I.	Perryridge	A-9732	700

- Provide as a gift for all loan customers of the Perryridge branch, a new $200 savings account for every loan account they have, with the loan number serving as the account number for the new savings account.

(next slide)

Modification of the Database – Insertion (Cont.)

account	*branch-name*	*account-number*	*balance*
I.	Perryridge	_*x*	200

depositor	*customer-name*	*account-number*
I.	_*y*	_*x*

loan	*branch-name*	*loan-number*	*amount*
	Perryridge	_*x*	

borrower	*customer-name*	*account-number*
	_*y*	_*x*

Modification of the Database – Updates

- Use the U. operator to change a value in a tuple without changing *all* values in the tuple. QBE does not allow users to update the primary key fields.

- Update the asset value of the of the Perryridge branch to $10,000,000.

branch	*branch-name*	*branch-city*	*assets*
	Perryridge		U.10000000

- Increase all balances by 5 percent.

account	*branch-name*	*account-number*	*balance*
U.			$\_x * 1.05$
			$\_x$

Quel

- Basic Structure
- Simple Queries
- Tuple Variables
- Aggregate Functions
- Modification of the Database
- Set Operations
- Quel and the Tuple Relational Calculus

Quel — Basic Structure

- Introduced as the query language for the Ingres database system, developed at the University of California, Berkeley.
- Basic structure parallels that of the tuple relational calculus.
- Most Quel queries are expressed using three types of clauses: **range of**, **retrieve**, and **where**.
  - Each tuple variable is declared in a **range of** clause.

    range of *t* **is** *r*

    declares *t* to be a tuple variable restricted to take on values of tuples in relation *r*.
  - The **retrieve** clause is similar in function to the **select** clause of SQL.
  - The **where** clause contains the selection predicate.

Quel Query Structure

- A typical Quel query is of the form:

  range of t_1 **is** r_1
  range of t_2 **is** r_2
  $\vdots$
  range of t_m **is** r_m
  retrieve $(t_{i_1}.A_{j_1},\ t_{i_2}.A_{j_2}, \ldots, t_{i_n}.A_{j_n})$
  where P

  – Each t_i is a tuple variable.
  – Each r_i is a relation.
  – Each A_{j_k} is an attribute.
  – The notation $t.A$ denotes the value of tuple variable t on attribute A.

Quel Query Structure (Cont.)

- Quel does not include relational algebra operations like **intersect**, **union**, and **minus**.
- Quel does not allow nested subqueries (unlike SQL).
  - Cannot have a nested **retrieve-where** clause inside a **where** clause.
- These limitations do not reduce the expressive power of Quel, but the user has fewer alternatives for expressing a query.

Simple Queries

- Find the names of all customers having a loan at the bank.

  range of *t* **is** *borrower*
  retrieve (*t.customer-name*)

- To remove duplicates, we must add the keyword **unique** to the **retrieve** clause:

  range of *t* **is** *borrower*
  retrieve unique (*t.customer-name*)

Query Over Several Relations

- Find the names of all customers who have both a loan and an account at the bank.

  range of *s* **is** *borrower*
  range of *t* **is** *depositor*
  retrieve unique (*s.customer-name*)
  where *t.customer-name* = *s.customer-name*

Tuple Variables

- Certain queries need to have two distinct tuple variables ranging over the same relation.
- Find the name of all customers who live in the same city as Jones does.

> **range of** *s* **is** *customer*
> **range of** *t* **is** *customer*
> **retrieve unique** (*s.customer-name*)
> **where** *t.customer-name* = "Jones" **and**
> *s.customer-city* = *t.customer-city*

Tuple Variables (Cont.)

- When a query requires only one tuple variable ranging over a relation, we can omit the **range of** statement and use the relation name itself as an implicitly declared tuple variable.
- Find the names of all customers who have both a loan and an account at the bank.

retrieve unique (*borrower.customer-name*)
where *depositor.customer-name* = *borrower.customer-name*

Aggregate Functions

- Aggregate functions in Quel compute functions on groups of tuples.
- Grouping is specified as part of each aggregate expression.
- Quel aggregate expressions may take the following forms:

  aggregate function ($t.A$)
  aggregate function ($t.A$ **where** P)
  aggregate function ($t.A$ **by** $s.B_1,\ s.B_2, ..., s.B_n$ **where** P)

  – *aggregate function* is one of **count**, **sum**, **avg**, **max**, **min**, **countu**, **sumu**, **avgu**, or **any**
  – t and s are tuple variables
  – A, B_1, $B_2, \ldots, B_n$ are attributes
  – P is a predicate similar to the **where** clause in a **retrieve**

Aggregate Functions (Cont.)

- The functions **countu**, **sumu**, and **avgu** are identical to **count**, **sum**, and **avg**, respectively, except that they remove duplicates from their operands.
- An aggregate expression may appear anywhere a constant may appear; for example, in a **where** clause.

Example Queries

- Find the average account balance for all accounts at the Perryridge branch.

  range of *t* **is** *account*
  retrieve avg (*t.balance* **where**
  t.branch-name = "Perryridge")

- Find all accounts whose balance is higher than the average balance of Perryridge-branch accounts.

  range of *u* **is** *account*
  range of *t* **is** *account*
  retrieve (*t.account-number*)
  where *t.balance* $>$ **avg** (*u.balance* **where**
  u.branch-name = "Perryridge")

Example Queries

- Find all accounts whose balance is higher than the average balance at the branch where the account is held.
  - Compute for each tuple *t* in *account* the average balance at branch *t.branch-name*.
  - In order to form these groups of tuples, use the **by** construct in the aggregate expression.

The query is:

range of *u* **is** *account*
range of *t* **is** *account*
retrieve (*t.account-number*)
where *t.balance* > **avg** (*u.balance* **by** *t.branch-name*
    where *u.branch-name* = *t.branch-name*)

Example Query

- Find the names of all customers who have an account at the bank, but do not have a loan from the bank.

range of *t* **is** *depositor*
range of *u* **is** *borrower*
retrieve unique (*t.customer-name*)
where any (*u.loan-number* **by** *t.customer-name*
where *u.customer-name* = *t.customer-name*) = 0

- The use of a comparison with **any** is analogous to the "there exists" quantifier of the relational calculus.

Example Query

- Find the names of all customers who have an account at all branches located in Brooklyn.
  - First determine the number of branches in Brooklyn.
  - Compare this number with the number of distinct branches in Brooklyn at which each customer has an account.

range of *t* **is** *depositor*
range of *u* **is** *account*
range of *s* **is** *branch*
range of *w* **is** *branch*
retrieve unique (*t.customer-name*)
where countu (*s.branch-name* **by** *t.customer-name*
        where *u.account-number* = *t.account-number*
        and *u.branch-name* = *s.branch-name*
        and *s.branch-city* = “Brooklyn”) =
    countu (*u.branch-name* **where** *u.branch-city* = “Brooklyn”)

Modification of the Database – Deletion

- The form of a Quel deletion is

  range of *t* **is** *r*
  delete *t*
  where *P*

- The tuple variable *t* can be implicitly defined.
- The predicate *P* can be any valid Quel predicate.
- If the **where** clause is omitted, all tuples in the relation are deleted.

Deletion Query Examples

- Delete all tuples in the *loan* relation.

  range of *t* **is** *loan*
  delete *t*

- Delete all Smith's account records.

  range of *t* **is** *depositor*
  delete *t*
  where *t.customer-name* = "Smith"

- Delete all account records for branches located in Needham.

  range of *t* **is** *account*
  range of *u* **is** *branch*
  delete *t*
  where *t.branch-name* = *u.branch-name* **and**
  u.branch-city = "Needham"

Modification of the Database – Insertion

- Insertions are expressed in Quel using the **append** command.
- Insert the fact that account A-9732 at the Perryridge branch has a balance of $700.

append to *account* (*branch-name* = “Perryridge”,
account-number = A-9732,
balance = 700)

Insertion Query Example

- Provide as a gift for all loan customers of the Perryridge branch, a new $200 savings account for every loan account that they have. Let the loan number serve as the account number for the new savings account.

range of *t* **is** *loan*
range of *s* **is** *borrower*
append to *account* (*branch-name* = *t.branch-name*,
account-number = *t.loan-number*,
balance = 200)
where *t.branch-name* = "Perryridge"
append to *depositor* (*customer-name* = *s.customer-name*,
account-number = *s.loan-number*)
where *t.branch-name* = "Perryridge" **and** *t.loan-number* =
s.loan-number

Modification of the Database – Updates

- Updates are expressed in Quel using the **replace** command.
- Increase all account balances by 5 percent.

  range of *t* **is** *account*
  replace *t* (*balance* = 1.05 * *t.balance*)

- Pay 6 percent interest on accounts with balances over $10,000, and 5 percent on all other accounts.

  range of *t* **is** *account*
  replace *t* (*balance* = 1.06 $*$ *balance*)
  where *t.balance* $>$ 10000
  replace *t* (*balance* = 1.05 $*$ *balance*)
  where *t.balance* $\leq$ 10000

Set Operations

- Quel does not include relational algebra operations like **intersect**, **union**, and **minus**.
- To construct queries that require the use of set operations, we must create temporary relations (via the use of regular Quel statements).
- Example: To create a temporary relation *temp* that holds the names of all depositors of the bank, we write

  range of *u* **is** *depositor*
  retrieve into *temp* **unique** (*u.customer-name*)

- The **into** *temp* clause causes a new relation, *temp*, to be created to hold the result of this query.

Example Queries

- Find the names of all customers who have an account, a loan, or both at the bank.
- Since Quel has no **union** operation, a new relation (called *temp*) must be created that holds the names of all borrowers of the bank.
- We find all borrowers of the bank and insert them in the newly created relation *temp* by using the **append** command.

range of *s* **is** *borrower*
append to *temp* **unique** (*s.customer-name*)

- Complete the query with:

range of *t* **is** *temp*
retrieve unique (*t.customer-name*)

Example Queries

- Find the names of all customers who have an account at the bank but do not have a loan from the bank.
- Create a temporary relation by writing:

  range of *u* **is** *depositor*
  retrieve into *temp* (*u.customer-name*)

- Delete from *temp* those customers who have a loan.

  range of *s* **is** *borrower*
  range of *t* **is** *temp*
  delete *t*
  where *t.customer-name* = *s.customer-name*

- *temp* now contains the desired list of customers. We write the following to complete our query.

  range of *t* **is** *temp*
  retrieve unique (*t.customer-name*)

Quel and the Tuple Relational Calculus

The following Quel query

range of t_1 **is** r_1
range of t_2 **is** r_2
.
.
.
range of t_m **is** r_m
retrieve unique $(t_{i_1}.A_{j_1},\ t_{i_2}.A_{j_2}, ..., t_{i_n}.A_{j_n})$
where P

would be expressed in the tuple relational calculus as:

$$\{t \mid \exists\, t_1 \in r_1,\ t_2 \in r_2, ..., t_m \in r_m\ ($$
$$t[r_{i_1}.A_{j_1}] = t_{i_1}[A_{j_1}] \wedge t[r_{i_2}.A_{j_2}] = t_{i_2}[A_{j_2}] \wedge ... \wedge$$
$$t[r_{i_n}.A_{j_n}] = t_{i_n}[A_{j_n}] \wedge P(t_1, t_2, ..., t_m))\}$$

Quel and TRC (Cont.)

- $t_1 \in r_1 \wedge t_2 \in r_2 \wedge \ldots \wedge t_m \in r_m$
  Constrains each tuple in $t_1, t_2, \ldots, t_m$ to take on values of tuples in the relation over which it ranges.

- $t[r_{i_1}.A_{j_1}] = t_{i_1}[A_{j_1}] \wedge t_{i_2}[A_{j_2}] = t[r_{i_2}.A_{j_2}] \wedge \ldots \wedge t[r_{i_n}.A_{j_n}] = t_{i_n}[A_{j_n}]$
  Corresponds to the **retrieve** clause of the Quel query.

- $P(t_1, t_2, \ldots, t_m)$
  The constraint on acceptable values for $t_1, t_2, \ldots, t_m$ imposed by the **where** clause in the Quel query.

- Quel achieves the power of the relational algebra by means of the **any** aggregate function and the use of insertion and deletion on temporary relations.

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

- 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.

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

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

$$?\ v1(\text{“A-217”}, B).$$

Example Queries

- Each rule defines a set of tuples that a view relation must contain.
- 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:

$$\begin{array}{l} \textit{interest-rate}(A, 0) \;\; :\!-\; \textit{account}(N, A, B),\; B < 2000 \\ \textit{interest-rate}(A, 5) \;\; :\!-\; \textit{account}(N, A, B),\; B >= 2000 \end{array}$$

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

$$\begin{array}{l} c(N) \;\; :\!-\; \textit{depositor}(N, A),\; \textbf{not}\; \textit{is-borrower}(N). \\ \textit{is-borrower}(N) \;\; :\!-\; \textit{borrower}(N, L). \end{array}$$

Syntax of Datalog Rules

- A *positive literal* has the form

$$p(t_1, t_2, \ldots, t_n)$$

  - p is the name of a relation with n attributes
  - each t_i is either a constant or variable

- A *negative literal* has the form

$$\textbf{not } p(t_1, t_2, \ldots, t_n)$$

Syntax of Datalog Rules (Cont.)

- *Rules* are built out of literals and have the form:

$$p(t_1, t_2, \ldots, t_n) \ :- \ L_1, L_2, \ldots, L_n.$$

  - each of the L_i's is a literal
  - head – the literal $p(t_1, t_2, \ldots, t_n)$
  - body – the rest of the literals

- A *fact* is a rule with an empty body, written in the form:

$$p(v_1, v_2, \ldots, v_n).$$

  - indicates tuple $(v_1, v_2, \ldots, v_n)$ is in relation p

Semantics of a Rule

- A *ground instantiation of a rule* (or simply *instantiation*) is the result of replacing each variable in the rule by some constant.
- Rule defining *v1*

  v1(*A*, *B*) :– *account*("Perryridge", *A*, *B*), $B > 700$.

- An instantiation of above rule:

  v1("A-217", 750) :– *account*("Perryridge", "A-217", 750),
  $750 > 700$.

- The body of rule instantiation R' is *satisfied* in a set of facts (database instance) *I* if

  1. For each positive literal $q_i(v_{i,1}, \ldots, v_{i,n_i})$ in the body of R', *I* contains the fact $q(v_{i,1}, \ldots, v_{i,n_i})$.
  2. For each negative literal **not** $q_j(v_{j,1}, \ldots, v_{j,n_j})$ in the body of R', *I* does not contain the fact $q_j(v_{j,1}, \ldots, v_{j,n_j})$.

Semantics of a Rule (Cont.)

- We define the set of facts that can be **inferred** from a given set of facts I using rule R as:

$$\begin{aligned} \mathit{infer}(R, I) = \{p(t_1, \ldots, t_{n_i}) \mid & \text{ there is an instantiation } R' \text{ of } R \\ & \text{where } p(t_1, \ldots, t_{n_i}) \text{ is the head of } R', \text{ and} \\ & \text{the body of } R' \text{ is satisfied in } I\ \} \end{aligned}$$

- Given a set of rules $\mathcal{R} = \{R_1,\ R_2,\ \ldots,\ R_n\}$, we define

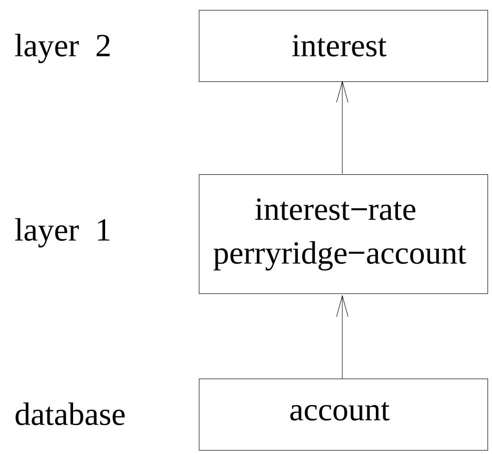
$$\mathit{infer}(\mathcal{R}, I) = \mathit{infer}(R_1, I) \cup \mathit{infer}(R_2, I) \cup \ldots \cup \mathit{infer}(R_n, I)$$

Layering of Rules

- Define the interest on each account in Perryridge.

  interest(*A*, *I*) :– *perryridge-account*(*A*, *B*),
  interest-rate(*A*, *R*), $I = B * R/100$.
  perryridge-account(*A*, *B*) :– *account*(“Perryridge”, *A*, *B*).
  interest-rate(*A*, 0) :– *account*(*N*, *A*, *B*), $B < 2000$.
  interest-rate(*A*, 5) :– *account*(*N*, *A*, *B*), $B >= 2000$.

- Layering of the view relations



Layering of Rules (Cont.)

Formally:

- A relation is in layer 1 if all relations used in the bodies of rules defining it are stored in the database.
- A relation is in 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 $1, 2, \ldots, i$
  - all relations used in the bodies of rules defining p are either stored in the database, or are in layers $1, 2, \ldots, i$

Semantics of a Program

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

- Define I_0 = set of facts stored in the database.
- Define $I_{i+1} = I_i \cup \mathit{infer}(\mathcal{R}_{i+1}, I_i)$
- 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.

$$gt(X, Y) \text{ :– } X > Y$$
$$not\text{-}in\text{-}loan(B, L) \text{ :– } \textbf{not } loan(B, L)$$

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

  – 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.

$$query(A) \text{ :– } account(N, A, B).$$

- Cartesian product of relations r_1 and r_2.

$$query(X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m) \text{ :– }$$
$$r_1(X_1, X_2, \ldots, X_n), \; r_2(Y_1, Y_2, \ldots, Y_m).$$

- Union of relations r_1 and r_2.

$$query\,(X_1, X_2, \ldots, X_n) \text{ :– } r_1(X_1, X_2, \ldots, X_n).$$
$$query\,(X_1, X_2, \ldots, X_n) \text{ :– } r_2(X_1, X_2, \ldots, X_n).$$

- Set difference of r_1 and r_2.

$$query\,(X_1, X_2, \ldots, X_n) \text{ :– } r_1(X_1, X_2, \ldots, X_n),$$
$$\textbf{not } r_2(X_1, X_2, \ldots, X_n).$$

Recursion in Datalog

- 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.

$$?\ empl(X, \text{“Jones”}).$$

manager

employee-name	*manager-name*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Semantics of Recursion in Datalog

- The view relations of a recursive program containing a set of rules $\mathcal{R}$ are defined to contain exactly the set of facts I computed by the iterative procedure *Datalog-Fixpoint*

```
procedure Datalog-Fixpoint
    I = set of facts in the database
    repeat
        Old_I = I
        I = I ∪ infer(R, I)
    until I = Old_I
```

- At the end of the procedure, $infer(\mathcal{R}, I) = I$
- I is called a *fixpoint* of the program.
- *Datalog-Fixpoint* computes only true facts so long as no rule in the program has a negative literal.

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.
- A view V is said to be **monotonic** if given any two sets of facts I_1 and I_2 such that $I_1 \subseteq I_2$, $E_V(I_1) \subseteq E_V(I_2)$, where E_V is the expression used to define V.
- Procedure *Datalog-Fixpoint* is sound provided the function *infer* is monotonic.
- Relational algebra views defined using only the operators: $\Pi, \sigma, \times, \bowtie, \cup, \cap$ and ρ are monotonic. Views using $-$ are not monotonic.