Advanced DB

CHAPTER 5
DATALOG

Datalog

- Basic Structure
- Syntax of Datalog Rules
- Semantics of Nonrecursive Datalog
- Safety
- Relational Operations in Datalog
- Recursion in Datalog
- The Power of Recursion
- *A More Theoretic View*

Basic Structure

- Prolog-like logic-based language 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 80.

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

Examples

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

- Each rule defines a set of tuples that a view relation must contain
- Above rule is read as

  for all A, B

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

  then $(A, B) \in v1$

- 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.

    $interest\_rate(A, 5) :- account(A, N, B), B < 10000$

    $interest\_rate(A, 6) :- account(A, N, B), B >= 10000$

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*), **not** *is_borrower*(*N*).

    is_borrower (*N*) :– *borrower*(*N*, *L*).

- NOTE:
  - The following definition results in a different meaning.

    c(*N*) :– *depositor*(*N*, *A*), **not** *borrower*(*N*, *L*).
  - 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*

Formal Syntax and Semantics of Datalog

- Steps in defining the syntax and semantics (meaning) of Datalog programs
  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. We define what rules are *safe*, because it is possible to write rules that can generate an infinite number of tuples in the view relation.
  5. It is possible to write recursive programs whose meaning is unclear. We define what recursive programs are acceptable, and define their meaning.

Syntax of Datalog Rules

- A *positive literal* has the form $p(t_1, t_2 ..., t_n)$
  - p is the name of a relation with n attributes
  - each t_i is a term (i.e., either a constant or variable)
- A *negative literal* has the form **not** $p(t_1, t_2 ..., t_n)$
- i.e., a *literal* is an atom or a negated atom
- Comparison operations are treated as positive predicates
  - E.g. $X > Y$ is treated as a predicate $>(X,Y)$
  - "$>$" is conceptually an (infinite) relation that contains all pairs of values such that the first value is greater than the second value
- Arithmetic operations are also treated as predicates
  - E.g. $A = B + C$ is treated as $+(B, C, A)$,

    where the relation "+" contains all triples such that the third value is the sum of the first two

Syntax of Datalog Rules (Cont.)

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

$$p(\underbrace{t_1, t_2, ..., t_n}_{head}) :- \underbrace{L_1, L_2, ..., L_m}_{body}.$$

  - each L_i is a literal
  - head: the literal $p(t_1, t_2, ..., t_n)$
  - body: the rest of the literals
- A fact
  - is a rule with an empty body, written in the form: $p(v_1, v_2, ..., v_n)$.
  - indicates that tuple $(v_1, v_2, ..., v_n)$ is in relation p
  - A base relation is actually a set of facts
- A *Datalog program* is a set of rules

Examples

- Example 1:

  parent(*X*, *Y*) :- *father*(*X*, *Y*)

  parent(*X*, *Y*) :- *mother*(*X*, *Y*)

  sibling(*X*, *Y*) :- *parent*(*X*, *Z*), *parent*(*Y*, *Z*), *X*<>*Y*

- Example 2:

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

  c(*N*) :– *depositor*(*N*, *A*), **not** *is_borrower*(*N*).

  is_borrower (*N*) :– *borrower*(*N*, *L*).

  interest(*A*, *I*) :– *perryridge_account*(*A*, *B*),
  interest_rate(*A*, R), *I* = *B**R/100.

  perryridge_account(*A*, *B*) :– *account* (*A*, "Perryridge", *B*).

  interest_rate(*A*, 5) :– *account* (*N*, *A*, *B*), *B*<10000.

  interest_rate(*A*, 6) :– *account* (*N*, A, B), *B*>=10000.

Semantics of a Rule

- *Ground instantiation* (or simply *instantiation*) of a rule
  - replacing each variable in the rule by some constant
  - E.g., an instantiation of *v1*(*A,B*) :– *account* (*A,*"Perryridge", *B*), *B*>700.

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

- For an instantiation *R'* (of rule *R*)

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

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

Semantics of a Rule (cont.)

- Set of facts that can be *inferred* from a given set of facts I using rule R (view definition)

  infer(R, I) = { $p(t_1, ..., t_n)$ | there is a ground instantiation R' of R
  where $p(t_1, ..., t_n)$ is the head of R', and
  the body of R' is satisfied in I }

- Example

  R1: *interest_rate*(A, 5) :– *account*(A, N, B), B<10000

  infer(R1, I) = { *interest_rate*(a, 5) | a is a constant and
  ∃ constants n, b such that
  <a, n, b> ∈ *account* and b<10000 in I }

Semantics of a Rule (cont.)

- Given a set of rules $\mathfrak{R} = \{R_1, ..., R_n\}$,

  $infer(\mathfrak{R}, I) = infer(R_1, I) \cup ... \cup infer(R_n, I)$

- Example

  R1: $interest\_rate(A, 5) :- account(A, N, B), B<10000$
  R2: $interest\_rate(A, 6) :- account(A, N, B), B >=10000$

  $infer(\mathfrak{R}, I) =$ { $interest\_rate(a, 5)$ | a is a constant and
  $\exists$ constants n, b such that
  $<a, n, b> \in account$ and $b<10000$ in I }

  $\cup$

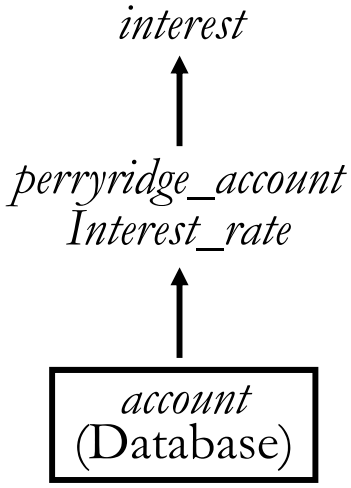
  { $interest\_rate(a, 6)$ | a is a constant and
  $\exists$ constants n, b such that
  $<a, n, b> \in account$ and $b>=10000$ in I }

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* (*A,* "Perryridge", *B).*
interest_rate(A, 5) :– *account* (*N, A, B), B*<10000.
interest_rate(*A,* 6) :– *account* (*N*, *A*, B), *B*>=10000.

- Layering of the view relations



Layering Rules (cont.)

- 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, ..., *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 $I_{i+1} = I_i \cup \mathit{infer}(\mathfrak{R}_{i+1}, I_i)$
- 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.

Examples

interest(A, lR) :– *perryridge_account(A, B),*
*interest_rate(A,R), lR = B*R/100.*

perryridge_account(A,B) :–
account (*A,* "Perryridge", *B).*

*interest_rate(A,*5) :–
account (*N, A, B), B*<10000.

interest_rate(*A,*6) :–
account (*N*, A, B), *B*>=10000.

I_0: *set of facts defined in the database;*
account

I_1 : $I_0 \cup$ set of facts inferred for *interest_rate* and *perryridge_account*

I_2 : $I_1 \cup$ set of facts inferred for *interest*

parent(*X*, *Y*) :- *father*(*X*, *Y*)

parent(*X*, *Y*) :- *mother*(*X*, *Y*)

sibling(*X*, *Y*) :- *parent*(*X*, *Z*), *parent*(*Y*, *Z*),
X<>*Y*

father(X, *Y*) :- *emp*(*X*, _, _, *Y*)

mother(X, *Y*) :- *emp*(*X*, _, *Y*, _)

I_0: set of facts defined in the database;
emp

I_1 : $I_0 \cup$ set of facts inferred for *father* and *mother*

I_2 : $I_1 \cup$ set of facts inferred for *parent*

I_3 : $I_2 \cup$ set of facts inferred for *sibling*

Safety

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

  gt(X, Y) :– $X > Y$

  not_in_loan (*B, L*) :– **not** *loan*(*B, L*)

  loves(*X, Y*) :- *lover*(*X*)

- To avoid this, Datalog rules must be *safe*
- A rule is *safe* if all its variables are *limited*.
  - Any variable that appears as an argument in a non-negated DB (view or base) predicate of the body is limited.
  - Any variable equated to a constant is limited ($X=a$).
  - Any variable equated to a limited variable is limited.
    - This condition can be weakened in special cases based on the semantics of arithmetic predicates, for example to permit the rule

      $p(A) :- q(B), A = B + 1$

Relational Operations in Datalog

- Select: $query(A, N, B) :- account(A, N, B), B>10000$
- Project: $query(A) :- account(A, N, B)$.
- Cartesian product:

  $query(X_1, ..., X_n, Y_1, ..., Y_m) :- r_1(X_1, ..., X_n), r_2(Y_1, ..., Y_m)$.
- Union: $query(X_1, ..., X_n) :- r_1(X_1, ..., X_n)$.

  $query(X_1, ..., X_n) :- r_2(X_1, ..., X_n)$.
- Set difference:

  $query(X_1, ..., X_n) :- r_1(X_1, ..., X_n)$, **not** $r_2(X_1, ..., X_n)$.
- Rename: $new\_name(X_1, ..., X_n) :- r(X_1, ..., X_n)$.
- Join? Intersection?

Recursion in Datalog

manager (*X*, *Y*) /* *X*'s direct manager is *Y* */

- Each manager may have direct employees, as well as indirect employees
- We wish to find all (direct and indirect) employees of manager 'Jones'.

  empl_jones(*X*) :- *manager* (*X*, 'Jones').

  empl_jones(*X*) :- *manager* (*X*, *Y*), *empl_jones* (*Y*).

Semantics of Recursion in Datalog

- Assumption (for now): program contains no negative literals
- The view relations of a recursive program $\mathfrak{R}$ are defined to contain exactly the set of facts I computed as follows

  procedure Datalog-Fixpoint
  I = set of facts in the database
  repeat
  $Old\_I = I$
  $I = I \cup infer(\mathfrak{R}, I)$
  until $I = Old\_I$

- At the end of the procedure, $infer(\mathfrak{R}, I) \subseteq I$
  - $Infer(\mathfrak{R}, I) = I$ if we consider the database to be a set of facts that are part of the program
- I is called a *fixed point* (*fixpoint*) of the program.

Example of Datalog-FixPoint Iteration

employee_name	*manager_name*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Iteration number	Tuples in *empl_jones*
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

Example: a more general definition of *empl*

- *empl* (*X*, *Y*): *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*, "Jones").

- Another way to define view *empl*

  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
  - 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
```

Another view of Fixpoint

- Fixpoint of a single view relation

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

Evaluation

$$empl(X, Y) = mn(X, Y) \cup (mn(X, W) \bowtie empl(W, Y))$$

$$empl^0 = \{\}$$
$$empl^1 = mn \cup (mn \bowtie empl^0) = mn$$
$$empl^2 = mn \cup (mn \bowtie empl^1) = mn \cup (mn \bowtie mn)$$
$$empl^3 = mn \cup (mn \bowtie empl^2) = mn \cup (mn \bowtie mn) \cup (mn \bowtie mn \bowtie mn)$$
$$\ldots$$

At some point, $empl^k = empl^{k+1}$ => Fixpoint!!!

Recursion mixed with Negation

- Example

  $p(X)$:- $r(X)$, **not** $q(X)$

  $q(X)$:- $r(X)$, **not** $p(X)$

  Let, $r = \{a, b, c\}$

- Fixpoints

  Case1: $p = r - q = \{a, b, c\} - \{\} = \{a, b, c\}$

  $q = r - p = \{a, b, c\} - \{a, b, c\} = \{\}$

  Case2: $q = r - p = \{a, b, c\} - \{\} = \{a, b, c\}$

  $p = r - q = \{a, b, c\} - \{a, b, c\} = \{\}$

- Two different meanings (fixed points) of the same database.

  => Bad!

Rules must be Stratified

- Layered
  - such that negation is not mixed with recursion within the same *stratum*
- Whenever there is a rule
  - *p* :- … **not** *q*,…
  - we should be able to compute *q* completely before having to compute *p*.
  - *q* should be in a lower stratum than *p*

- Example: the following set of rules is not stratified

  p(*X*) :- *r*(*X*), **not** *q*(*X*)

  q(*X*) :- *r*(*X*), **not** *p*(*X*)

Logic & Relational Algebra

- *Theorem* (from Relational Algebra to Logic)

  Every query expressible in Relational Algebra is expressible as a nonrecursive datalog program.

- *Theorem* (from Logic to Relational Algebra)

  Let R *be a collection of safe, nonrecursive datalog rules, possibly with negation. Then for each predicate p of* R*, there is an expression of relational algebra that computes the relation for p.*

A MORE THEORETIC VIEW

Language of First Order Logic

- Symbols
  - Variables, constants, function symbols, predicate symbols:
  - *X, Y, Z, …, a, b, c, …, f, g, h, …, p, q, r, …*
  - Logical connectives: $\wedge$, $\vee$, $\neg$, $\leftarrow$, $\rightarrow$
  - Quantifiers: $\forall$, $\exists$
  - Parentheses: ( )
- Terms
  - represent individual elements of a domain
  - constants, variables, functions with terms as arguments
  - *a, b, c, …, X, Y, Z, …, f*(*a*), *g*(*b, c*), …
- Atom
  - *Represents a statement (predicate)*
  - $p(t_1, \ldots, t_n)$, where p is an n-place predicate and t_i is a term, $1 \le i \le n$
  - note: $t_1 < t_2$ can be written as $<(t_1, t_2)$ or $lessThan(t_1, t_2)$

Language of First Order Logic (cont.)

- (Well formed) Formula
  1. An atom is a formula (atomic formula)
  2. If f_1 and f_2 are formulas then (f_1), $f_1 \wedge f_2$, $f_1 \vee f_2$, $\neg f_1$, $f_1 \rightarrow f_2$, and $f_1 \leftarrow f_2$ are formulas
  3. If X is a free variable in formula f, $\forall X f$, $\exists X f$ are formulas
  4. Formulas are generated only by a finite number of applications of rules 1 ~ 3.
- Scope of a quantifier
  - (sub)formula to which the quantifier applies
  - Examples
    - $\forall X p(X, Y) \wedge \exists Y q(X, Y)$
    - $\forall X(p(X,Y) \wedge \exists Y q(X,Y))$
    - $\forall X \exists Y(p(X,Y) \wedge q(X,Y))$
    - $\forall X \exists Y \; loves(X,Y)$
    - $\exists Y \forall X \; loves(X,Y)$

Language of First Order Logic (cont.)

- Free and bound variables
  - An occurrence of a variable in a formula is bound if it is within the scope of a quantifier employing the variable.
  - Otherwise the occurrence is free.
- Open formula
  - A formula is open if at least one occurrence of a variable is free.
  - Otherwise the formula is closed.
    - Closed formula: sentence
    - Theory: set of sentences
    - Open formula: query

Semantics (Meaning) of First Order Formula

- *Example: Are the following statements true?*
  - $\forall X(p(X) \rightarrow q(X))$
  - $\forall X(man(X) \rightarrow dies(X))$
  - $\forall X \exists Y(book(X) \rightarrow man(X) \wedge read(X,Y))$
  - $\forall X \exists Y(p(X) \rightarrow s(X) \wedge q(X,Y))$
- *Interpretation I*
  - *D*: domain
  - Map each constant to an element in *D*
  - each *n*-place function to a mapping: $D^n \rightarrow D$
  - each *n*-place predicate to a mapping: $D^n \rightarrow \{T, F\}$
  - $\wedge$, $\vee$, $\neg$, $\leftarrow$, $\rightarrow$: as usual
  - $\forall Xf$: T if *f* is T for every *d* in *D*, otherwise F
  - $\exists Xf$: T if *f* is T for any *d* in *D*, otherwise F
  - (open formula cannot be evaluated)

Semantics of First Order Formula (cont.)

- Example1: $\{ \forall X p(X), \exists Y \neg p(Y) \}$
  - I1: $D = \{1, 2\}$, $p(1)$: T, $p(2)$:T
    => $\forall X p(X)$:T, $\exists Y \neg p(Y)$:F
  - I2: $D = \{tom, sam, jim, NY\}$, p:person's name
    => $\forall X p(X)$:F, $\exists Y \neg p(Y)$:T

- Example 2: $\forall X(p(X) \rightarrow q(f(X), a))$
  - *I1*:
    - $D = \{$ *tom, sam, jim, thesis1, …, thesis4, pass, fail, …* $\}$
    - a=*pass* p: PhD: $p(tom)$, $p(sam)$
    - $f(X)$: X's thesis: $f(tom)=th1$, $f(sam)=th2$, $f(jim)=th3$
    - q: result of thesis exam

    => "every PhD must have passed her thesis exam"
  - *I2*:
    - $D = \{1, 2\}$, a=1, $f(1)$=2, $f(2)$=1,
    - $p(1)$:T, $p(2)$:T, $q(1,1)$:T, $q(1,2)$:T, $q(2,1)$:F, $q(2,2)$:T

Model

- *Model* of a formula
  - an interpretation in which the formula is true
- Partial ordering of interpretations
  - Let I_1 and I_2 be interpretations of a formula and R_1 and R_2 be the corresponding database instances.

    $I_1 \subseteq I_2$ if $R_1 \subseteq R_2$ (each relation-wise)
- Minimal model
  - M_0 is a *minimal model* of a formula F if there is no model M of F such that $M_0 \subseteq M$ (minimal number of truth assignments (tuples))
- Logical consequence ($F \models f$)

  iff f is true in every model of F.

  (i.e., for every interpretation I of F, f is true in I whenever F is true in I)

  (i.e., if we accept (interpret) that F is true, then we can conclude f is true also)
- F_1 and F_2 are equivalent ($\equiv$)

  if $F_1 \models F_2$ and $F_2 \models F_1$.

Semantics of a Datalog Program

- Model Theoretic View
  - Datalog program ($\mathfrak{R} \cup I$) is a theory
  - Facts in a minimal model of this theory are the only true facts
- Theory ($\mathfrak{R} \cup I$)
  - $\mathfrak{R} = \{R_1, ..., R_n\}$
  - For each base relation q with m tuples, we have m atomic clauses $q_i(v_{i1}, ..., v_{ini})$.
- A possible model for ($\mathfrak{R} \cup I$)
  - Predicates correspond to either DB predicates (table names) or built-in predicates (=, <, …).
  - D: active domain
  - Each constant to itself
  - No functions
  - Built-in predicates: as usual
  - Base DB predicates: T iff corresponding tuple is in respective table
  - View DB predicates: T for all combinations of values (not minimal)

Horn clauses

- CNF, $C_1 \wedge \ldots \wedge C_n$
  - each $C_i = p_1 \vee \ldots \vee p_m \vee \neg q_1 \vee \ldots \vee \neg q_k$ (clause)
  - Any formula can be represented in CNF
- C_i is a Horn clause
  - when m (number of positive literals) ≤ 1
  - $p_1 \leftarrow q_1, \ldots, q_k$
- Significance
  - Unique minimal model
  - Definite answers
    - Example: $p(a) \vee p(b)$ In a model of the formula, is $p(a)$ true?
- Datalog
  - A logic-based data model
  - Function-free: no function symbols
  - Horn clauses: $p_1 \leftarrow q_1, \ldots, q_k$

r

Three formal meanings

- Model-theoretic view (meaning)
  - Datalog program ($\mathfrak{R} \cup I$) is a theory
  - Facts in a minimal model are the only true facts
- Proof-theoretic view (meaning)
  - Facts that are derivable (can be proved) from the program ($\mathfrak{R} \cup I$) are the only true facts.
  - This is the view that we adopted so far (e.g., fixpoint)
  - For Horn DB (including Datalog), Proof-t-v ≡ Model-t-v
- Operational semantics (meaning)
  - Define an algorithm (rules of computation)
  - All the facts that the algorithm says true are true
  - Important for implementation considerations
  - Example: Prolog

END OF CHAPTER 5