Relational Calculus,Visual Query Languages, and Deductive Databases

CSE 532, Theory of Database Systems

Stony Brook University

http://www.cs.stonybrook.edu/~cse532

SQL and Relational Calculus

- Although relational *algebra* is useful in the analysis of query evaluation, SQL is actually based on a different query language: *relational calculus*
- There are two relational calculi:
  - *Tuple relational calculus* (TRC)
  - *Domain relational calculus* (DRC)

Tuple Relational Calculus

- Form of query:

$$\{T \mid Condition(T)\}$$

  - T is the *target* – a variable that ranges over tuples of values
  - *Condition* is the *body* of the query
    - Involves T (and possibly other variables)
    - Evaluates to *true* or *false* if a specific tuple is substituted for T

Tuple Relational Calculus: Example

$$\{T \mid \text{Teaching}(T) \text{ AND } T.\mathit{Semester} = \text{'F2000'}\}$$

- When a concrete tuple has been substituted for T:
  - Teaching(T) is true if T is in the relational instance of Teaching
  - T.*Semester* = 'F2000' is true if the semester attribute of T has value F2000
  - Equivalent to:

```
SELECT   *
FROM     Teaching T
WHERE    T.Semester = 'F2000'
```

Relation Between SQL and TRC

{T | Teaching(T) AND T.*Semester* = 'F2000'}

```
SELECT  *
FROM    Teaching T
WHERE   T.Semester = 'F2000'
```

- Target T corresponds to SELECT list: the query result contains the entire tuple
- Body split between two clauses:
  - Teaching(T) corresponds to FROM clause
  - T.*Semester* = 'F2000' corresponds to WHERE clause

Query Result

- The result of a TRC query with respect to a given database is the set of all choices of tuples for the variable T that make the query condition a true statement about the database

Query Condition

- *Atomic condition*:
  - $P(T)$, where P is a relation name
  - $T.A$ *operator* $S.B$ or $T.A$ *operator constant*, where T and S are relation names, A and B are attributes and *operator* is a comparison operator (*e.g.*, $=, \neq, <, >, \in$, etc)
- (*General*) *condition*:
  - atomic condition
  - If C_1 and C_2 are conditions then C_1 AND C_2, C_1 OR C_2, and NOT C_1 are conditions
  - If R is a relation name, T a tuple variable, and $C(T)$ is a condition that uses T, then $\forall T \in R\ (C(T))$ and $\exists T \in R\ (C(T))$ are conditions

Bound and Free Variables

- X is a *free* variable in the statement C_1: "*X is in* CS305" (this might be represented more formally as $C_1(X)$)
  - The statement is neither true nor false in a particular state of the database until we assign a value to X
- X is a *bound* (or *quantified*) variable in the statement C_2: "*there exists a student X such that X is in* CS305" (this might be represented more formally as

$$\exists \ X \in S \ (C_2(X))$$

where S is the set of all students)
  - This statement can be assigned a truth value for any particular state of the database

Bound and Free Variables in TRC Queries

- Bound variables are used to make assertions about tuples in database (used in conditions)
- Free variables designate the tuples to be returned by the query (used in targets)

$$\{S \mid \text{Student}(S) \;\text{AND}\; (\exists T \in \text{Transcript} \; (S.Id = T.StudId \;\text{AND}\; T.CrsCode = \text{'CS305'}))\}$$

  - When a value is substituted for S the condition has value *true* or *false*
- There can be only one free variable in a condition (the one that appears in the target)

Example 2

$$
\{\, E \mid Course(E)\ \text{AND}\ \forall S \in Student\ (\ \exists\, T \in Transcript\ (\ T.\mathit{StudId} = S.\mathit{Id}\ \ \text{AND}\ \ T.\mathit{CrsCode} = E.\mathit{CrsCode}\ )\ )\ \}
$$

- Returns the set of all course tuples corresponding to all courses that have been taken by all students

TRC Syntax Extension

- We add syntactic sugar to TRC, which simplifies queries and make the syntax even closer to that of SQL:

$$\{S.\mathit{Name}, T.\mathit{CrsCode} \mid \text{Student}(S) \text{ AND } \text{Transcript}(T) \text{ AND } \ldots\}$$

instead of

$$\{R \mid \exists S \in \text{Student}\,(R.\mathit{Name} = S.\mathit{Name}) \text{ AND } \exists T \in \text{Transcript}(R.\mathit{CrsCode} = T.\mathit{CrsCode}) \text{ AND } \ldots\}$$

where R is a new tuple variable with attributes *Name* and *CrsCode*

Relation Between TRC and SQL (cont'd)

- List the names of all professors who have taught MGT123
  - In TRC:

$$\{P.Name \mid Professor(P) \text{ AND } \exists T \in Teaching \; (P.Id = T.ProfId \text{ AND } T.CrsCode = \text{'MGT123'})\}$$

  - In SQL:

```
SELECT   P.Name
FROM     Professor P, Teaching T
WHERE    P.Id = T.ProfId  AND  T.CrsCode = 'MGT123'
```

Core of SQL is merely a syntactic sugar on top of TRC

What Happened to Quantifiers in SQL?

- SQL has no quantifiers: how come? It uses conventions:
  - *Convention* 1. Universal quantifiers are not allowed (but SQL:1999 introduced a limited form of explicit $\forall$)
  - *Convention* 2. Make existential quantifiers *implicit*: Any tuple variable that does not occur in SELECT is assumed to be implicitly quantified with $\exists$
- Compare:

$$\{P.Name \mid Professor(P) \text{ AND } \exists T \in Teaching \ \ldots\}$$

and

```
SELECT   P.Name
FROM     Professor P, Teaching T
… … …
```

Implicit $\exists$

Relation Between TRC and SQL (cont'd)

- SQL uses a subset of TRC with simplifying conventions for quantification
- Restricts the use of quantification and negation (so TRC is more general in this respect)
- SQL uses aggregates, which are absent in TRC (and relational algebra, for that matter). But aggregates can be added
- SQL is extended with relational algebra operators (MINUS, UNION, JOIN, etc.)
  - This is just more syntactic sugar, but it makes queries easier to write

More on Quantification

- Adjacent existential quantifiers and adjacent universal quantifiers commute:
  - $\exists T \in Transcript\ (\exists T1 \in Teaching\ (\ldots))$ is *same* as $\exists T1 \in Teaching\ (\exists T \in Transcript\ (\ldots))$
- Adjacent existential and universal quantifiers *do not* commute:
  - $\exists T \in Transcript\ (\forall T1 \in Teaching\ (\ldots))$ is *different* from $\forall T1 \in Teaching\ (\exists T \in Transcript\ (\ldots))$

More on Quantification (con't)

- A quantifier defines the scope of the quantified variable (analogously to a begin/end block):

  $\forall T \in R1\ (U(T)\ \text{AND}\ \exists T \in R2(V(T)))$

  is the same as:

  $\forall T \in R1\ (U(T)\ \text{AND}\ \exists S \in R2(V(S)))$

- *Universal domain*: Assume a domain, U, which is a union of all other domains in the database. Then, instead of $\forall T \in U$ and $\exists S \in U$ we simply write $\forall T$ and $\exists T$

Views in TRC

- **Problem**: List students who took a course from every professor in the Computer Science Department
- **Solution:**
  - First create view: All CS professors

$$\text{CSProf} = \{\text{P}.\textit{ProfId} \mid \text{Professor(P) AND P}.\textit{DeptId} = \text{'CS'}\}$$

  - Then use it

$$\{\text{S}.\ \textit{Id} \mid \text{Student(S) AND}$$
$$\forall \text{P} \in \text{CSProf}\ \exists \text{T} \in \text{Teaching}\ \exists \text{R} \in \text{Transcript} \ ($$
$$\text{AND P}.\ \textit{Id} = \text{T}.\textit{ProfId} \text{ AND S}.\textit{Id} = \text{R}.\textit{StudId} \text{ AND}$$
$$\text{T}.\textit{CrsCode} = \text{R}.\textit{CrsCode} \text{ AND T}.\textit{Semester} = \text{R}.\textit{Semester}$$
$$)\ \}$$

Queries with Implication

- Did not need views in the previous query, but doing it without a view has its pitfalls: need the implication → (if-then):

$$
\begin{aligned}
&\{S.\,Id \mid \text{Student}(S) \text{ AND} \\
&\quad \forall P \in \text{Professor} \; (\\
&\qquad P.DeptId = \text{'CS'} \rightarrow \\
&\qquad \exists T1 \in \text{Teaching} \; \exists R \in \text{Transcript} \; (\\
&\qquad\quad P.Id = T1.ProfId \text{ AND } S.Id = R.Id \\
&\qquad\quad \text{AND } T1.CrsCode = R.CrsCode \\
&\qquad\quad \text{AND } T1.Semester = R.Semester \\
&\qquad\quad) \\
&\qquad) \\
&\quad \}
\end{aligned}
$$

- Why $P.DeptId = \text{'CS'} \rightarrow \ldots$ and **not** $P.DeptId = \text{'CS'}$ AND … ?
  - List students who took a course from every professor in the Computer Science Department!!!

More complex SQL to TRC Conversion

- Using views, translation between complex SQL queries and TRC is direct:

SELECT R1.*A*, R2.*C*
FROM Rel1 R1, Rel2 R2
WHERE *condition1*(R1, R2) AND
R1.*B* IN (SELECT R3.*E*
FROM Rel3 R3, Rel4 R4
WHERE *condition2*(R2, R3, R4))

TRC view corresponds to subquery

versus:

$\{R1.A, R2.C \mid Rel1(R1) \text{ AND } Rel2(R2) \text{ AND } condition1(R1, R2)$
$\text{AND } \exists R3 \in Temp\ (R1.B = R3.E \text{ AND } R2.C = R3.C$
$\text{AND } R2.D = R3.D) \}$

$Temp = \{R3.E, R2.C, R2.D \mid Rel2(R2) \text{ AND } Rel3(R3)$
$\text{AND } \exists R4 \in Rel4\ (condition2(R2, R3, R4))\}$

Domain Relational Calculus (DRC)

- A *domain variable* is a variable whose value is drawn from the domain of an attribute
  - Contrast this with a tuple variable, whose value is an entire tuple
  - *Example*: The domain of a domain variable *Crs* might be the set of all possible values of the *CrsCode* attribute in the relation Teaching

Queries in DRC

- Form of DRC query:

  $$\{X_1, \ldots, X_n \mid condition(X_1, \ldots, X_n)\}$$

- $X_1, \ldots, X_n$ is the *target*: a list of domain variables.
- $condition(X_1, \ldots, X_n)$ is similar to a condition in TRC; uses free variables $X_1, \ldots, X_n$.
  - However, quantification is over a domain
    - $\exists X \in$ Teaching.*CrsCode* (… … …)
      - i.e., there is X in Teaching.*CrsCode*, such that condition is true
- Example: {*Pid*, *Code* | Teaching(*Pid*, *Code*, 'F1997')}
  - This is similar to the TRC query:

    {T | Teaching(T) AND T.*Semester* = 'F1997'}

Query Result

- The result of the DRC query

  $$\{X_1 , \ldots, X_n \mid condition(X_1 , \ldots, X_n) \}$$

  with respect to a given database is the set of all tuples $(x_1,\ldots,x_n)$ such that, for $i = 1,\ldots,n$, if x_i is substituted for the free variable X_i , then $condition(x_1 , \ldots, x_n)$ is a true statement about the database
  - X_i can be a constant, c, in which case $x_i = c$

Examples

- List names of all professors who taught MGT123:

  $\{Name \mid \exists Id\, \exists Dept\, (\text{Professor}(Id, Name, Dept) \text{ AND}$
  $\exists Sem\, (\text{Teaching}(Id, \text{'MGT123'}, Sem))\,)\}$

  - The universal domain is used to abbreviate the query
  - Note the mixing of variables (*Id*, *Sem*) and constants (MGT123)

- List names of all professors who ever taught Ann

$\{Name \mid \exists Pid\, \exists Dept\, ($
$\quad \text{Professor}(Pid, Name, Dept) \text{ AND}$
$\quad \exists Crs\, \exists Sem\, \exists Grd\, \exists Sid\, \exists Add\, \exists Stat\, ($
$\quad\quad \text{Teaching}(Pid, Crs, Sem) \text{ AND}$
$\quad\quad \text{Transcript}(Sid, Crs, Sem, Grd) \text{ AND}$
$\quad\quad \text{Student}(\text{Sid}, \text{'Ann'}, Addr, Stat)$
$))\,\}$

Lots of ∃ – a hallmark of DRC. Conventions like in SQL can be used to shorten queries

Relation Between Relational Algebra, TRC, and DRC

- Consider the query $\{T \mid \text{NOT } Q(T)\}$: returns the set of all tuples ***not*** in relation Q
  - If the attribute domains change, the result set changes as well
  - This is referred to as a *domain-dependent* query
- Another example: $\{T \mid \forall S(R(S)) \lor Q(T)\}$
  - It is domain-dependent
- Only *domain-**in**dependent* queries make sense, but checking domain-independence is undecidable
  - But there are syntactic restrictions that guarantee domain-independence

Relation Between Relational Algebra, TRC, and DRC

- Relational algebra (but not DRC or TRC) queries are always domain-<u>in</u>dependent (proved by induction!)
- TRC, DRC, and relational algebra are equally expressive for domain-independent queries
  - Proving that every domain-independent TRC/DRC query can be written in the algebra is somewhat hard
  - We will show the other direction: that algebraic queries are expressible in TRC/DRC

Relation Between Relational Algebra, TRC, and DRC

- Algebra: $\sigma_{Condition}(\mathbf{R})$
- TRC: $\{T \mid R(T)$ AND $Condition_1\}$
- DRC: $\{X_1,\ldots,X_n \mid R(X_1,\ldots,X_n)$ AND $Condition_2\}$

- Let *Condition* be $A=B$ AND $C=$'Joe'. Why $Condition_1$ and $Condition_2$?
  - Because TRC, DRC, and the algebra have slightly different syntax:

    $Condition_1$ is T.A=T.B AND T.C='Joe'

    $Condition_2$ would be $A=B$ AND $C=$'Joe'

    (possibly with different variable names)

Relation Between Relational Algebra, TRC, and DRC

- Algebra: $\pi_{A,B,C}(\mathbf{R})$
- TRC: $\{T.A, T.B, T.C \mid \mathbf{R}(T)\}$
- DRC: $\{A,B,C \mid \exists D\ \exists E \ldots\ \mathbf{R}(A,B,C,D,E,\ldots)\ \}$

- Algebra: $\mathbf{R} \times \mathbf{S}$
- TRC: $\{T.A.T.B, T.C, V.D, V,E \mid \mathbf{R}(T)\ \text{AND}\ \mathbf{S}(V)\ \}$
- DRC: $\{A,B,C,D,E \mid \mathbf{R}(A,B,C)\ \text{AND}\ \mathbf{S}(D,E)\ \}$

Relation Between Relational Algebra, TRC, and DRC

- Algebra: $\mathbf{R} \cup \mathbf{S}$
- TRC: $\{T \mid \mathbf{R}(T) \text{ OR } \mathbf{S}(T)\}$
- DRC: $\{A,B,C \mid \mathbf{R}(A,B,C) \text{ OR } \mathbf{S}(A,B,C)\}$

- Algebra: $\mathbf{R} - \mathbf{S}$
- TRC: $\{T \mid \mathbf{R}(T) \text{ AND NOT } \mathbf{S}(T)\}$
- DRC: $\{A,B,C \mid \mathbf{R}(A,B,C) \text{ AND NOT } \mathbf{S}(A,B,C)\}$

QBE: Query by Example

- Declarative query language, like SQL
- Based on DRC (rather than TRC)
- Visual
- Other visual query languages (MS Access, Paradox) are just incremental improvements

QBE Examples

Print all professors' names in the MGT department

Professor	*Id*	*Name*	*DeptId*
		P. _John	MGT

Operator "Print"

Targetlist "example" variable

Same, but print all attributes

Professor	*Id*	*Name*	*DeptId*
P.			MGT

- Literals that start with "_" are variables.

Joins in QBE

- Names of professors who taught MGT123 in any semester except Fall 2002

Professor	*Id*	*Name*	*DeptId*
	_123	**P.**_John	

Teaching	*ProfId*	*CrsCode*	*Semester*
	_123	MGT123	<> 'F2002'

Simple conditions placed directly in columns

Condition Boxes

- Some conditions are too complex to be placed directly in table columns

Transcript	*StudId*	*CrsCode*	*Semester*	*Grade*
	P.	CS532		_Gr

Conditions
_Gr = 'A' OR _Gr = 'B'

- Students who took CS532 & got A or B

Aggregates, Updates, etc.

- Has aggregates (operators like AVG, COUNT), grouping operator, etc.
- Has update operators
- To create a new table (like SQL's CREATE TABLE), simply construct a new template:

HasTaught	*Professor*	*Student*
I.	123456789	567891012

A Complex Insert Using a Query

q
u
e
r
y

Transcript	*StudId*	*CrsCode*	*Semester*	*Grade*
	_5678	_CS532	_S2002	

Teaching	*ProfId*	*CrsCode*	*Semester*
	_12345	_CS532	_S2002

u
p
d
a
t
e

HasTaught	*Professor*	*Student*
I.	_12345	_5678

query
target

HasTaught	*Professor*	*Student*
P.		

Connection to DRC

- Obvious: just a graphical representation of DRC
- Uses the same convention as SQL: existential quantifiers (∃) are omitted

Transcript	*StudId*	*CrsCode*	*Semester*	*Grade*
	_123	_CS532	F2002	A

Transcript(*X*, *Y*, 'F2002', 'A')

Pitfalls: Negation

- List all professors who didn't teach anything in S2002:

Professor	*Id*	*Name*	*DeptId*
	_123	**P.**	

Teaching	*ProfId*	*CrsCode*	*Semester*
¬	_123		S2002

- *Problem*: What is the quantification of *CrsCode*?

  $\{Name \mid \exists Id\ \exists DeptId\ \exists CrsCode\ (\ \text{Professor}(Id, Name, DeptId)\ \ \text{AND}$

  $\text{NOT}\ \ \text{Teaching}(Id, CrsCode, \text{'S2002'})\ )\ \}$

  - <u>Not</u> what was intended(!!), but what the convention about implicit quantification says

or

  $\{Name \mid \exists Id\ \exists DeptId\ \forall CrsCode\ (\ \text{Professor}(Id, Name, DeptId)\ \ \text{AND}\ \ \ldots\ldots\}$

  - The <u>intended</u> result!

Negation Pitfall: Resolution

- QBE changed its convention:
  - Variables that occur only in a negated table are *implicitly* quantified with ∀ instead of ∃
  - For instance: *CrsCode* in our example. Note: _123 (which corresponds to *Id* in DRC formulation) is quantified with ∃, because it also occurs in the non-negated table Professor
- Still, problems remain! Is it

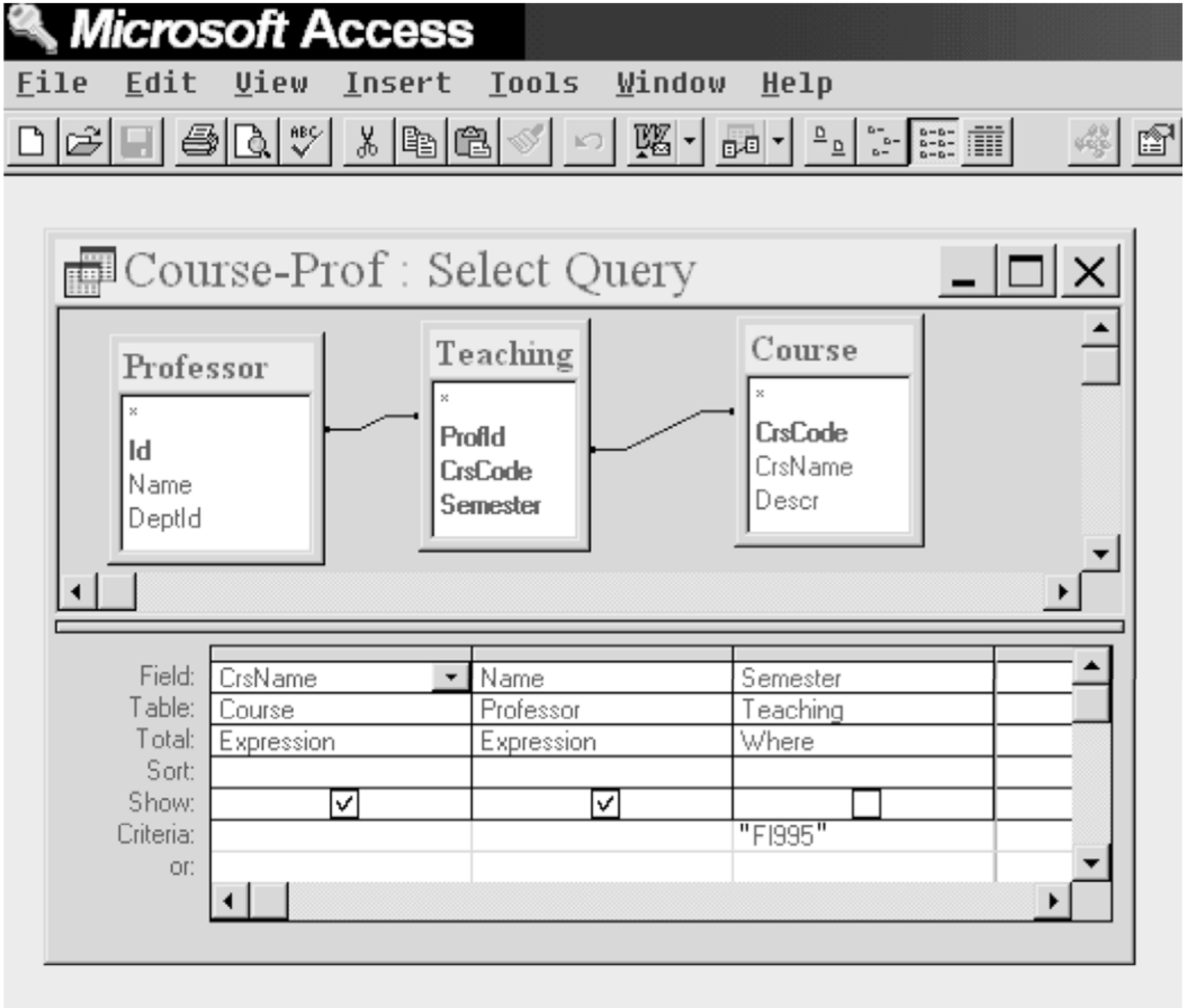
  $\{Name \mid \exists Id\ \exists DeptId\ \forall CrsCode\ (\ \text{Professor}(Id, Name, DeptId)\ \text{ AND } \ldots\}$

  or

  $\{Name \mid \forall CrsCode\ \exists Id\ \exists DeptId\ (\ \text{Professor}(Id, Name, DeptId)\ \text{ AND } \ldots\}$

  Not the same query!
  - QBE decrees that the ∃-prefix goes first

Microsoft Access



PC Databases

- A spruced up version of QBE (better interface)
- Be aware of implicit quantification
- Beware of negation pitfalls

Deductive Databases

- Motivation: Limitations of SQL
- Recursion in SQL:1999
- Datalog – a better language for complex queries

Limitations of SQL

- Given a relation Prereq with attributes *Crs* and *PreCrs*, list the set of all courses that must be completed prior to enrolling in CS632
  - The set Prereq $_2$, computed by the following expression, contains the immediate and once removed (i.e. 2-step prerequisites) prerequisites for all courses:

$$\pi_{Crs,\ PreCrs}\ ((\text{Prereq} \quad \bowtie_{PreCrs=Crs} \text{Prereq})[Crs,\ P1,\ C2,\ PreCrs]$$
$$\cup\ \text{Prereq}$$

  - In general, Prereq$_i$ contains all prerequisites up to those that are *i*-1 removed for all courses:

$$\pi_{Crs,\ PreCrs}\ ((\text{Prereq} \quad \bowtie_{PreCrs=Crs} \text{Prereq}_{i\text{-}1})[Crs,\ P1,\ C2,\ PreCrs]$$
$$\cup\ \text{Prereq}_{i\text{-}1}$$

Limitations of SQL (con't)

- **Question**: We can compute $\sigma_{Crs=\text{'CS632'}}(\text{Prereq}_i)$ to get all prerequisites up to those that are *i*-1 removed, but how can we be sure that there are not additional prerequisites that are *i* removed?
- **Answer**: When you reach a value of *i* such that $\text{Prereq}_i = \text{Prereq}_{i+1}$ you've got them all. This is referred to as a *stable state*
- **Problem**: There's no way of doing this within relational algebra, DRC, TRC, or SQL (this is *not* obvious and *not* easy to prove)

Recursion in SQL:1999

- Recursive queries can be formulated using a recursive view:

(a) CREATE RECURSIVE VIEW IndirectPrereq (*Crs*, *PreCrs*) AS
SELECT * FROM Prereq
UNION
(b) SELECT P.*Crs*, I.*PreCrs*
FROM Prereq P, IndirectPrereq I
WHERE P.*PreCrs* = I.*Crs*

- (a) is a *non*-recursive subquery – it cannot refer to the view being defined
  - Starts recursion off by introducing the *base case* – the set of direct prerequisites

Recursion in SQL:1999 (cont'd)

```
    CREATE RECURSIVE VIEW  IndirectPrereq (Crs, PreCrs)  AS
    SELECT  *  FROM  Prereq
    UNION
    SELECT   P.Crs, I.PreCrs
(b) FROM     Prereq  P,  IndirectPrereq  I
    WHERE    P.PreCrs = I.Crs
```

- (b) contains *recursion* – this subquery refers to the view being defined.
  - This is a declarative way of specifying the iterative process of calculating successive levels of indirect prerequisites until a stable point is reached

Recursion in SQL:1999

- The recursive view can be evaluated by computing successive approximations
  - IndirectPrereq$_{i+1}$ is obtained by taking the union of IndirectPrereq$_i$ with the result of the query

```
SELECT   P.Crs, I.PreCrs
FROM     Prereq P, IndirectPrereq_i I
WHERE    P.PreCrs = I.Crs
```

  - Successive values of IndirectPrereq$_i$ are computed until a stable point is reached, i.e., when the result of the query (IndirectPrereq$_{i+1}$) is contained in IndirectPrereq$_i$

Recursion in SQL:1999

- Also provides the WITH construct, which does not require views.
- Can even define mutually recursive queries:

```
WITH
        RECURSIVE OddPrereq(Crs, PreCrs) AS
          (SELECT * FROM Prereq)
          UNION
          (SELECT P.Crs, E.PreCrs
           FROM   Prereq P, EvenPrereq E
           WHERE  P.PreCrs=E.Crs ) ),
        RECURSIVE EvenPrereq(Crs, PreCrs) AS
          (SELECT P.Crs, O.PreCrs
           FROM   Prereq P, OddPrereq O
           WHERE  P.PreCrs = O.Crs )
SELECT * FROM OddPrereq
```

Datalog

- Rule-based query language
- Easier to use, more modular than SQL
- *Much* easier to use for recursive queries
- Extensively used in research
- Partial implementations of Datalog are used commercially
- W3C is standardizing a version of Datalog for the Semantic Web
  - RIF-BLD: Basic Logic Dialect of the Rule Interchange Format http://www.w3.org/TR/rif-bld/

Basic Syntax

- Rule:

  head `:-` *body*.

- Query:

  `?-` *body*.

- *body*: any DRC expression without the quantifiers.
  - *AND* is often written as ',' (without the quotes)
  - *OR* is often written as ';'
- *head*: a DRC expression of the form $R(t_1,\ldots,t_n)$, where t_i is either a constant or a variable; *R* is a relation name.
- *body* in a rule and in a query has the same syntax.

Basic Syntax (cont'd)

Derived relation;
Like a database view

NameSem(?*Name*,?*Sem*) :- Prof(?*Id*,?*Name*,?*Dept*), Teach(?*Id*,'MGT123',?*Sem*).

?- NameSem(?Name,?Sem).

Base relation, if never
occurs in a rule head

Answers:

?Name = kifer

?Sem = F2005

?Name = lewis

?Sem = F2004

… … …

Basic Syntax (cont'd)

- Datalog's quantification of variables
  - Like in SQL and QBE: *implicit*
  - Variables that occur in the rule body, *but not in the head* are viewed as being quantified with $\exists$
  - Variables that occur in the head are like target variables in SQL, QBE, and DRC

Basic Semantics

NameSem(?*Name*,?*Sem*) :- Prof(?*Id*,?*Name*,?*Dept*), Teach(?*Id*,'MGT123',?*Sem*).

?- NameSem(?Name, ?Sem).

The easiest way to explain the semantics is to use DRC:

$$\text{NameSem} = \{Name, Sem \mid \exists Id\, \exists Dept\ (\ \text{Prof}(Id, Name, Dept)\ \text{ AND } \ \text{Teaching}(Id, \text{`MGT123'}, Sem)\ )\ \}$$

Basic Semantics (cont'd)

- Another way to understand rules:

As in DRC, join is indicated by sharing variables

NameSem(?*Name*,?*Sem*) :- Prof(?*Id*,?*Name*,?*Dept*), Teach(?*Id*,'MGT123',?*Sem*).

$\Uparrow$ $\Uparrow$ $\Uparrow$

(bob, F2002) (1111, bob, CS) and (1111, MGT123, F2002)

If these tuples exist

Then this one must also exist

Union Semantics of Multiple Rules

- Consider rules with the same head-predicate:

  NameSem(?*Name*,?*Sem*) :- Prof(?*Id*,?*Name*,?*Dept*), Teach(?*Id*,'MGT123',?*Sem*).

  NameSem(?*Name*,?*Sem*) :- Prof(?*Id*,?*Name*,?*Dept*), Teach(?*Id*,'CS532',?*Sem*).

- Semantics is the *union*:

$$\text{NameSem} = \{Name, Sem \mid \exists Id\, \exists Dept\, ((\text{Prof}(Id, Name, Dept) \text{ AND } \text{Teaching}(Id, \text{'MGT123'}, Sem)) \text{ OR } (\text{Prof}(Id, Name, Dept) \text{ AND } \text{Teaching}(Id, \text{'CS532'}, Sem)))\}$$

Equivalently:

$$\text{NameSem} = \{Name, Sem \mid \exists Id\, \exists Dept\, (\text{Prof}(Id, Name, Dept) \text{ AND } (\text{Teaching}(Id, \text{'MGT123'}, Sem) \text{ OR } \text{Teaching}(Id, \text{'CS532'}, Sem)))\}$$

by distributivity

- Above rules can also be written in one rule:

  NameSem(?*Name*,?*Sem*) :- Prof(?*Id*,?*Name*,?*Dept*),
  (Teach(?*Id*,'MGT123',?*Sem*) ; Teach(?*Id*,'CS532',?*Sem*)).

Recursion

- Recall: DRC cannot express transitive closure
- SQL was specifically extended with recursion to capture this (in fact, but mimicking Datalog)
- Example of recursion in Datalog:

IndirectPrereq(?*Crs*,?*Pre*) :- Prereq(?*Crs*,?*Pre*).

IndirectPrereq(?*Crs*,?*Pre*) :-

Prereq(?*Crs*,?*Intermediate*),

IndirectPrereq(?*Intermediate*,?*Pre*).

Semantics of Recursive Datalog Without Negation

- ***Positive*** rules
  - No negation (not) in the rule body
  - No disjunction in the rule body
    - The last restriction does not limit the expressive power: *H* **:-** *(B;C)* is equivalent to *H* **:-** *B* and *H* **:-** *C* because
      - *H* **:-** *B* is *H or* not *B*
      - Hence
        - *H or* ***not*** (*B or C*) is equivalent to the pair of formulas

          H or ***not*** *B*

          and

          H or ***not*** *C*.

Semantics of Negation-free Datalog (cont'd)

- A Datalog rule

$$HeadRelation(HeadVars) \text{ :- } Body$$

can be represented in DRC as

$$HeadRelation = \{HeadVars \mid \exists BodyOnlyVars \;\; Body\}$$

- We call this ***the DRC query corresponding to the above Datalog rule***

Semantics of Negation-free Datalog - An Algorithm

- Semantics can be defined completely declaratively, but we will define it using an algorithm
- *Input*: A set of Datalog rules without

  negation + a database
- The *initial state* of the computation:
  - *Base relations* – have the content assigned to them by the database
  - *Derived relations* – initially empty

Semantics of Negation-free Datalog - An Algorithm (cont'd)

1. *CurrentState* := *InitialDBState*
2. For each derived relation **R**, let $r_1,\ldots,r_k$ be all the rules that have **R** in the head
   - Evaluate the DRC queries that correspond to each r_i
   - Assign the union of the results from these queries to **R**
3. NewState := the database where instances of all derived relations have been replaced as in Step 2 above
4. **if** *CurrentState* = *NewState*

   then *Stop*: *NewState* is the stable state that represents the meaning of that set of Datalog rules on the given DB

   else *CurrentState* **:=** *NewState*; Goto Step 2.

Semantics of Negation-free Datalog - An Algorithm (cont'd)

- The algorithm always **terminates**:
  - *CurrentState* constantly grows (at least, never shrinks)
    - Because DRC expressions of the form

      $\exists$Vars (A and/or B and/or C ...)

      which have no negation, are ***monotonic***: if tuples are added to the database, the result of such a DRC query grows monotonically
  - It cannot grow indefinitely (Why?)
- **Complexity**: number of steps is polynomial in the size of the DB (if the ruleset is fixed)
  - *D* – number of constants in DB;

    N – sum of all arities
  - Can't take more than D^N iterations
  - Each iteration can produce at most D^N tuples

  ➢ Hence, the number of steps is $O(D^N * D^N)$

Expressivity

- Recursive Datalog can express queries that cannot be done in DRC (e.g., transitive closure) – recall recursive SQL
- DRC can express queries that cannot be expressed in Datalog without negation (e.g., complement of a relation or set-difference of relations)
- Datalog with negation is strictly more expressive than DRC

Negation in Datalog

- Uses of negation in the rule body:
  - *Simple uses*: For set difference
  - *Complex cases*: When the (relational algebra) division operator is needed
- Expressing division is hard, as in SQL, since no explicit universal quantification

Negation (cont'd)

- *Find all students who took a course from every professor*

```
Answer(?Sid) :- Student(?Sid, ?Name, ?Addr),
                not DidNotTakeAnyCourseFromSomeProf(?Sid).

DidNotTakeAnyCourseFromSomeProf(?Sid) :-
                Professor(?Pid,?Pname,?Dept),
                Student(?Sid,?Name,?Addr),
                not HasTaught(?Pid,?Sid).
HasTaught(?Pid,?Sid) :- Teaching(?Pid,?Crs,?Sem),
                        Transcript(?Sid,?Crs,?Sem,?Grd).

?- Answer(?Sid).
```

- Not as straightforward as in DRC, but still quite logical!

Negation Pitfalls: Watch Your Variables

- Has problem similar to the wrong choice of operands in relational division
- Consider: *Find all students who have passed all courses that were taught in spring 2006*

$$\pi_{StudId,\ CrsCode,\ Grade}(\sigma_{Grade\neq 'F'}(\text{Transcript})) \ / \ \pi_{CrsCode}(\sigma_{Semester='S2006'}(\text{Teaching}))$$

versus

$$\pi_{StudId,\ CrsCode}(\sigma_{Grade\neq 'F'}(\text{Transcript})) \ / \ \pi_{CrsCode}(\sigma_{Semester='S2006'}(\text{Teaching}))$$

Which is correct? Why?

Negation Pitfalls (cont'd)

- Consider a reformulation of: *Find all students who took a course from every professor*

Answer(?*Sid*) :- ∃?*Pid* ∃?*Name*
    Student(?*Sid*, ?*Name*, ?*Addr*),
    Professor(?*Pid*,?*Pname*,?*Dept*),
    not ProfWhoDidNotTeachStud(?*Sid*,?*Pid*).

Implied quantification is wrong!

ProfWhoDidNotTeachStud(?Sid,?*Pid*) :-
    Professor(?*Pid*,?*Pname*,?*Dept*),
    Student(?*Sid*,?*Name*,?*Addr*),
    not HasTaught(?*Pid*,?*Sid*).
HasTaught(?*Pid*,?*Sid*) :- … … …

The only real differences compared to
DidNotTakeAnyCourseFromSomeProf

?- Answer(?*Sid*).

- What's wrong?
- The answer will consist of students *who were taught by* ***some*** *professor*

Negation and a Pitfall: Another Example

- Negation can be used to express containment: *Students who took every course taught by professor with Id 1234567 in spring 2006.*
  - DRC

$$\{Name \mid \forall Crs \exists Grade \exists Sid$$
$$(\text{Student}(Sid, Name),$$
$$(Teaching(1234567, Crs, \text{'S2006'})$$
$$\Rightarrow Transcript(Sid, Crs, \text{'S2006'}, Grade)))\}$$

  - Datalog

Answer(?*Name*) :- Student(?*Sid*,?*Name*),
    not DidntTakeS2006CrsFrom1234567(?*Sid*).

DidntTakeS2006CrsFrom1234567(?*Sid*) :-
    Teaching(1234567,?*Crs*,'S2006'), ***not*** TookS2006Course(?Sid,?Crs).

TookS2006Course(?Sid,?Crs) :- Transcript(?Sid,?Crs,'S2006',?Grade).

  - **Pitfall**: Transcript(?*Sid*,?*Crs*,'S2006',?*Grade*) here won't do because of ∃?*Grade* !

Negation and Recursion

- What is the meaning of a ruleset that has recursion through ***not***?
- Already saw this in recursive SQL – same issue

OddPrereq(?*X*,?*Y*) :- Prereq(?*X*,?*Y*).
OddPrereq(?*X*,?*Y*) :- Prereq(?*X*,?*Z*), EvenPrereq(?*Z*,?*Y*),
not EvenPrereq(?X,?Y).
EvenPrereq(?*X*,?*Y*) :- Prereq(?*X*,?*Z*), OddPrereq(?*Z*,?*Y*).

?- OddPrereq(?*X*,?*Y*).

- *Problem*:
  - Computing OddPrereq depends on knowing the complement of EvenPrereq
  - To know the complement of EvenPrereq, need to know EvenPrereq
  - To know EvenPrereq, need to compute OddPrereq first!

Negation Through Recursion (cont'd)

- The algorithm for positive Datalog wont work with negation in the rules:
  - For convergence of the computation, it relied on the ***monotonicity*** of the DRC queries involved
  - But with negation in DRC, these queries are no longer monotonic:

    Query = $\{X \mid P(X) \text{ and not } Q(X)\}$

    P(a), P(b), P(c); Q(a) => Query result: {b,c}

    Add Q(b) => Query result shrinks: just {c}

"Well-behaved" Negation

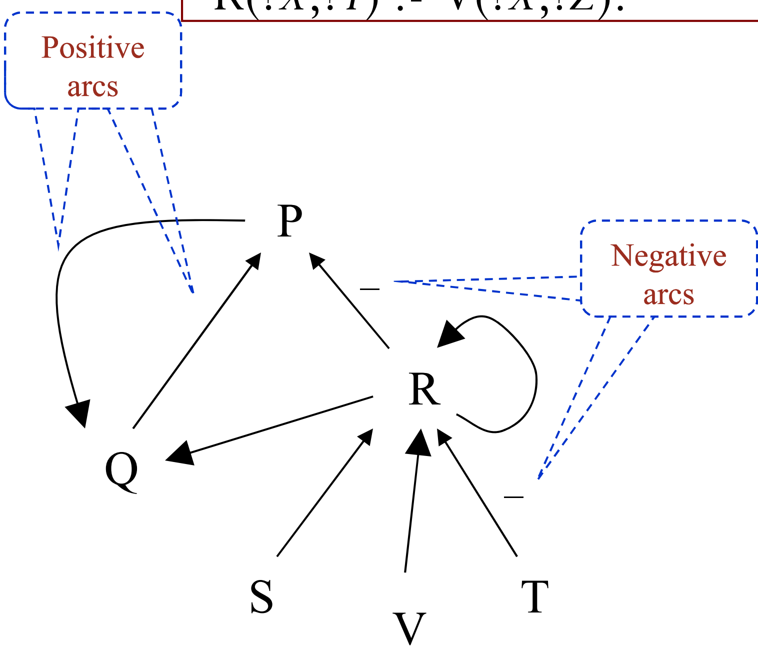
- Negation is "well-behaved" if there is no recursion through it

P(?*X*,?*Y*) :- Q(?*X*,?*Z*), ***not*** R(?*X*,?*Y*).

Q(?*X*,?*Y*) :- P(?*X*,?*Z*), R(?*X*,?*Y*).

R(?*X*,?*Y*) :- S(?*X*,?*Z*), R(?*Z*,?*V*), ***not*** T(?*V*,?*Y*).

R(?*X*,?*Y*) :- V(?*X*,?*Z*).



Dependency graph

Evaluation method for P:

1. Compute T , then its complement, **not** T
2. Compute R using the Negation-free Datalog algorithm. Treat **not** T as base relation
3. Compute **not** R
4. Compute Q and P using Negation-free Datalog algorithm. Treat **not** R as base

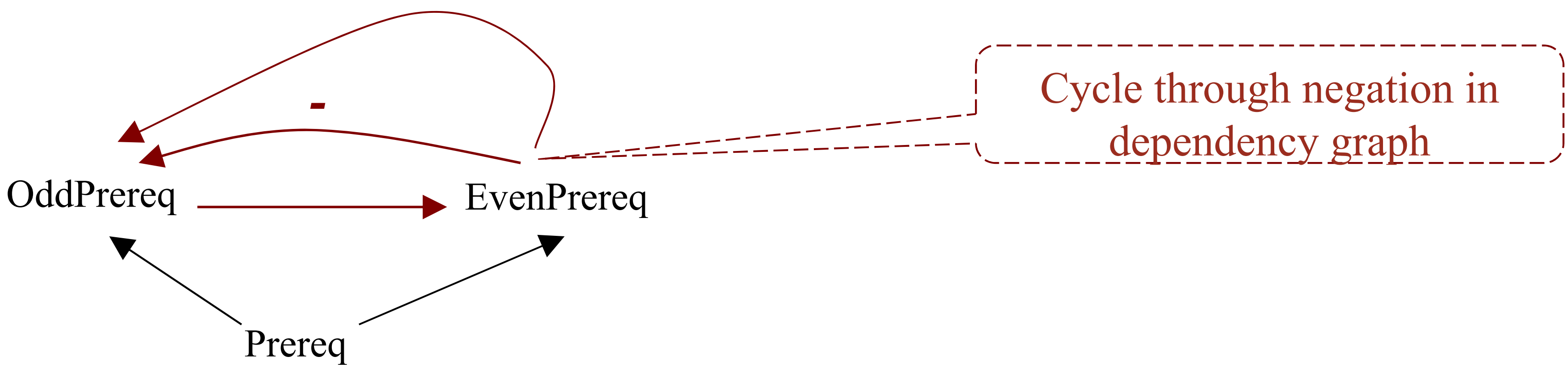
"Ill-behaved" Negation

- What was wrong with the even/odd prerequisites example?

OddPrereq(?X,?Y) :- Prereq(?X,?Y).

OddPrereq(?X,?Y) :- Prereq(?X,?Z), EvenPrereq(?Z,?Y),
not EvenPrereq(?X,?Y).

EvenPrereq(?X,?Y) :- Prereq(?X,?Z), OddPrereq(?Z,?Y).



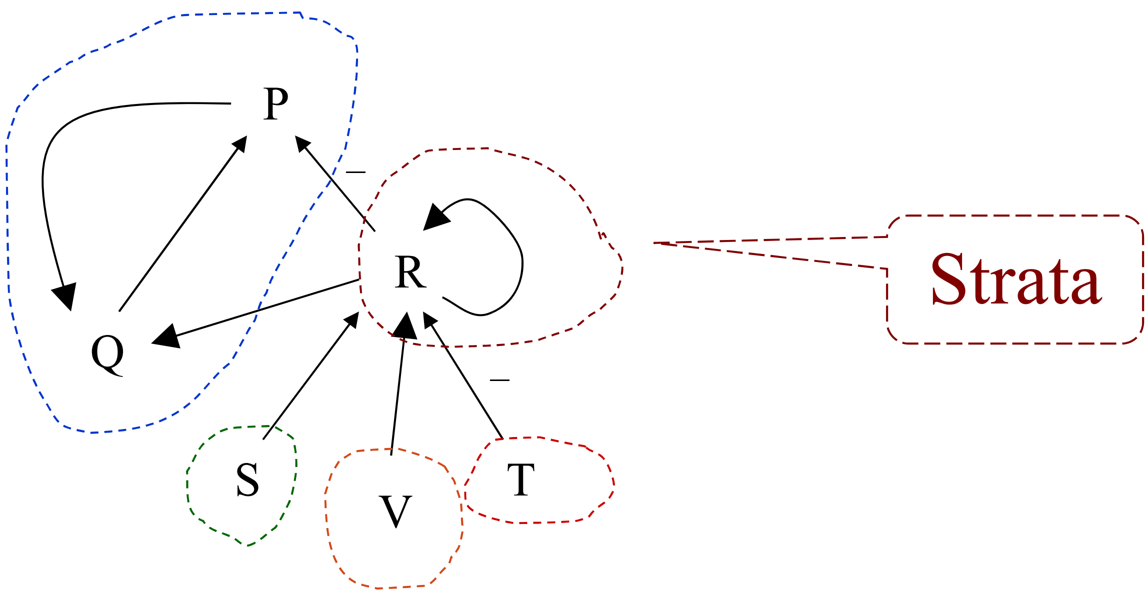
Dependency graph

Dependency Graph for a Ruleset R

- *Nodes*: relation names in **R**
- *Arcs*:
  - if P(…) `:-` …, Q(…), … is in **R** then the dependency graph has a *positive* arc Q `------>` R
  - if P(…) `:-` …, ***not*** Q(…), … is in **R** then the dependency graph has a *negative* arc Q `------>` R (marked with the minus sign)

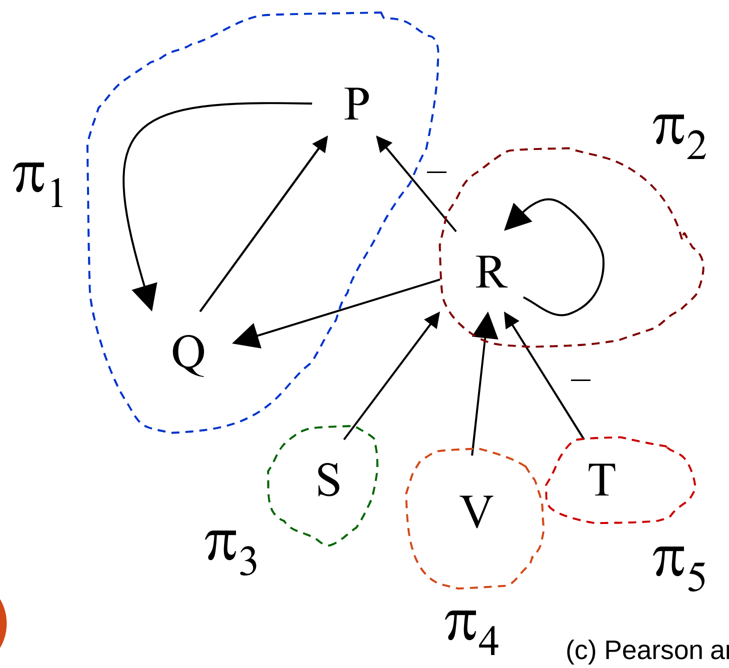
Strata in a Dependency Graph

- A ***stratum*** is a positively strongly connected component, i.e., a subset of nodes such that:
  - No *negative paths* among any pair of nodes in the set
  - Every pair of nodes has a *positive path* connecting them (i.e., a----> b and b----> a)



Stratification

- *Partial order on the strata*: if there is a path from a node in a stratum, π, to a stratum φ, then $\pi < \varphi$.
  (Are $\pi < \varphi$ and $\varphi < \pi$ possible together?)
- ***Stratification***: any total order of the strata that is consistent with the above partial order.



A possible stratification:

π_3 , π_5 , π_4 , π_2 , π_1

Another stratification:

π_5 , π_4 , π_3 , π_2 , π_1

Stratifiable Rulesets

- This is what we meant earlier by "well-behaved" rulesets
- A ruleset is ***stratifiable*** if it has a stratification
- Easy to prove (see the book):
  - *A ruleset is stratifiable* iff *its dependency graph has no negative cycles (or if there are no cycles, positive or negative, among the strata of the graph)*

Partitioning of a Ruleset According to Strata

- Let **R** be a ruleset and let $\pi_1 , \pi_2 , \ldots , \pi_n$ be a stratification
- Then the rules of **R** can be partitioned into subsets $Q_1 , Q_2 , \ldots, Q_n$, where each Q_i includes exactly those rules whose head relations belong to π_i

Evaluation of a Stratifiable Ruleset, R

1. Partition the relations of **R** into strata
2. Stratify (order)
3. Partition the ruleset according to the strata into the subsets Q_1 , Q_2 , Q_3 , …, Q_n
4. Evaluate
   a. Evaluate the lowest stratum, Q_1, using the negation-free algorithm
   b. Evaluate the next stratum, Q_2, using the results for Q_1 and the algorithm for negation-free Datalog
      - If relation P is defined in Q_1 and used in Q_2, then treat P as a base relation in Q_2
      - If ***not*** P occurs in Q_2, then treat it as a new *base* relation, NotP, whose extension is the complement of P (which can be computed, since P was computed earlier, during the evaluation of Q_1)
   c. Do the same for Q_3 using the results from the evaluation of Q_2, etc.

Unstratified Programs

- Truth be told, stratification is *not* needed to to evaluate Datalog rulesets. But this becomes a rather complicated stuff, which we won't touch. (Refer to the bibliographic notes, if interested.)

The XSB Datalog System

- http://xsb.sourceforge.net
- Developed at Stony Brook by Prof. Warren and many contributors
- Not just a Datalog system – it is a complete programming language, called Prolog, which happens to support Datalog
- Has a number of syntactic differences with the version we have just seen

Differences

- *Variables*: Any alphanumeric symbol that starts with a capital letter or a _ (underscore)
  - Examples: Abc, ABC2, _abc34
  - Non-examples: 123, abc, aBC
- Each occurrence of a singleton symbol _ is treated as a *new* variable, which was never seen before:
  - Example: p(_,abc), q(cde,_) – the two _'s are treated as completely different variables
  - But the two occurrences of _xyz in p(_xyz,abc), q(cde,_xyz) refer to the same variable
- Relation names and constants:
  - must either start with a lowercase letter (and include only alphanumerics and _)
    - Example: abc, aBC123, abc_123
  - or be enclosed in single quotes
    - Example: 'abc &% (, foobar1'
    - Note: abc *and* 'abc' refer to the same thing

Differences (cont'd)

- Negation: called ***tnot***
  - Note: XSB also has ***not***, but it is a different thing!
  - Use: … :- …, ***tnot***(foobar(X)).
- All variables under the scope of ***tnot*** must also occur to the left of that scope in the body of the rule in other <u>positive</u> relations:
  - Ok: … :- p(X,Y), ***tnot***(foobar(X,Y)), …
  - Not ok: … :- p(X,Z), ***tnot***(foobar(X,Y)), …
- XSB does not support Datalog by default – must tell it to do so with this instruction:

  :- auto_table.

  at the top of the program file

Overview of Installation

- Unzip/untar; this will create a subdirectory XSB
- Windows: you are done
- Linux:

  cd XSB/build

  ./configure

  ./makexsb

  That's it!
- Cygwin under Windows: same as in Linux

Use of XSB

- Put your ruleset *and* data in a file with extension .P (or .pl)

```
p(X) :- q(X,_).
q(1,a).
q(2,a).
q(b,c).
?- p(X).
```

- Don’t forget: all rules and facts end with a period (.)
- Comments: /*…*/ or %…. (% acts like // in Java/C++)
- Type

  …/XSB/bin/xsb (Linux/Cygwin)

  …\XSB\config\x86-pc-windows\bin\xsb (Windows)

  where … is the path to the directory where you downloaded XSB
- You will see a prompt

```
| ?-
```

  and are now ready to type queries

Use of XSB (cont'd)

- Loading your program, myprog.P

  | ?- [myprog].

  XSB will compile myprog.P (if necessary) and load it. Now you can type further queries, e.g.

  | ?- p(X).

  | ?- p(1).

  Etc.

Some Useful Built-ins

- write(X) – write whatever X is bound to
- writeln(X) – write then put newline
- nl – output newline
- Equality: =
- Inequality: \=

http://xsb.sourceforge.net/manual1/index.html (Volume 1)
http://xsb.sourceforge.net/manual2/index.html (Volume 2)

Arithmetics

- If you need it: use the builtin *is*

```
p(1). p(2).
q(X) :- p(Y), X is Y*2.
```

Now q(2), q(4) will become true.

- Note:

```
q(2*X) :- p(X).
```

will not do what you might think it will do.

It will make q(2*1) and q(2*2) true, but 2*1 and 2*2 are treated completely differently from 2 and 4 (no need to get into all that for now)

Some Useful Tricks

- XSB returns only the first answer to the query. To get the next, type ; <Return>. For instance:

```
| ?- q(X).    <Return>
X = 2    ; <Return>
X = 4      <Return>
yes
| ?-
```

- Usually, typing the **;**'s is tedious. To do this programmatically, use this idiom:

```
| ?- (q(_X), write('X='), writeln(_X), fail ; true).
```

_X here tells XSB to not print its own answers, since we are printing them by ourselves. (XSB won't print answers for variables that are prefixed with a _.)

Aggregates in XSB

- setof(?Template, +Goal, ?Set) : ?Set is the set of all instances of Template such that Goal is provable.
- bagof(?Template, +Goal, ?Bag) has the same semantics as setof/3 except that the third argument returns an unsorted list that may contain duplicates.
- findall(?Template, +Goal, ?List) is similar to predicate bagof/3, except that variables in Goal that do not occur in Template are treated as existential, and alternative lists are not returned for different bindings of such variables.
- tfindall(?Template, +Goal, ?List) is similar to predicate findall/3, but the Goal must be a call to a single tabled predicate.

XSB Prolog basics

- An **atom** is a general-purpose name with no inherent meaning.
- **Numbers** can be floats or integers.
- A **compound term** is composed of an atom called a "functor" and a number of "arguments", which are again terms: tree(node(a),tree(node(b),node(c)))
- Special cases of compound terms:
  - *Lists:* ordered collections of terms: [], [1,2,3], [a,1,X|T]
  - *Strings*: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes: “abc”, “to be, or not to be”

XSB Prolog

- Variables begin with a capital letter or underscore:

  `X, Socrates, _result`

- Atoms do *not* begin with a capital letter:

  `socrates, paul`

  - Atoms containing special characters, or beginning with a capital letter, must be enclosed in single quotes: `'Socrates'`

Representation of Lists

- List is handled as binary tree in Prolog

  [Head | Tail] OR

  .(Head,Tail)

  - Where Head is an atom and Tail is a list
  - We can write [a,b,c] or .(a,.(b,.(c,[]))).

Matching

- Given two terms, they are identical or the variables in both terms can have same objects after being instantiated

  date(D,M,2006) unification date(D1,feb,Y1)

  D=D1, M=feb, Y1=2006

- General Rule to decide whether two terms, S and T match are as follows:
  - If S and T are constants, S=T if both are same object
  - If S is a variable and T is anything, T=S
  - If T is variable and S is anything, S=T
  - If S and T are structures, S=T if
    - S and T have same functor
    - All their corresponding arguments components have to match

Declarative and Procedural Way

- Prolog programs can be understood two ways: declaratively and procedurally.
- P:- Q,R
- Declarative Way
  - P is true if Q and R are true
- Procedural Way
  - To solve problem P, first solve Q and then R (or) To satisfy P, first satisfy Q and then R
  - Procedural way does not only define logical relation between the head of the clause and the goals in the body, but also the order in which the goal are processed.

Evaluation

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).

sibling(X, Y):- parent_child(Z, X), parent_child(Z, Y).

?- sibling(sally, erica).
Yes
```

(by chronological backtracking)

Evaluation

- `?- father_child(Father, Child).`

  enumerates all valid answers on backtracking.

- `?- sibling(S1, S2).`

  enumerates all valid answers on backtracking.

Append example

```
append([],L,L).
append([X|L], M, [X|N]) :- append(L,M,N).
```

```
append([1,2],[3,4],X)?
```

Append example

```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```

```
append([1,2],[3,4],X)?    X=1,L=[2],M=[3,4],A=[X|N]
```

Append example

```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```

append([2],[3,4],N)?	
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[X|N]

Append example

```
append([],L,L).
append([X|L],M,[X|N’]) :- append(L,M,N’).
```

append([2],[3,4],N)?	X=2,L=[],M=[3,4],N=[2|N’]
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[1|N]

Append example

```
append([],L,L).
append([X|L],M,[X|N’]) :- append(L,M,N’).
```

append([],[3,4],N’)?	
append([2],[3,4],N)?	X=2,L=[],M=[3,4],N=[2|N’]
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[1|N]

Append example

```
append([],L,L).
append([X|L],M,[X|N’]) :- append(L,M,N’).
```

append([],[3,4],N’)?	L = [3,4], N’ = L
append([2],[3,4],N)?	X=2,L=[],M=[3,4],N=[2|N’]
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[1|N]

Append example

```
append([],L,L).
append([X|L],M,[X|N']) :- append(L,M,N').
```

```
A = [1|N]
 N = [2|N']
 N'= L
 L = [3,4]
```

Answer: **A** = **[1,2,3,4]**

append([],[3,4],N')?	L = [3,4], N' = L
append([2],[3,4],N)?	X=2,L=[],M=[3,4],N=[2|N']
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[1|N]

More Examples

```
member(X,[X|R]).
member(X,[Y|R]) :- member(X,R)
```

- *X is a member of a list whose first element is X.*
- *X is a member of a list whose tail is R if X is a member of R.*

```
?- member(2,[1,2,3]).
Yes
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

More Examples

```
select(X,[X|R],R).
select(X,[F|R],[F|S]) :- select(X,R,S).
```

- *When X is selected from [X|R], R results.*
- *When X is selected from the tail of [X|R], [X|S] results, where S is the result of taking X out of R.*

```
?- select(X,[1,2,3],L).
X=1  L=[2,3] ;
X=2  L=[1,3] ;
X=3  L=[1,2] ;
No
```

More Examples

```
reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).
reverse([],X,X).

?- reverse([1,2,3],[],X).
X = [3,2,1]
Yes
```

More Examples

```
perm([],[]).
perm([X|Y],Z) :- perm(Y,W), select(X,Z,W).

?- perm([1,2,3],P).
P = [1,2,3] ;
P = [2,1,3] ;
P = [2,3,1] ;
P = [1,3,2] ;
P = [3,1,2] ;
P = [3,2,1]
```

Recursion

- Transitive closure:

```
edge(1,2).
edge(2,3).
edge(2,4).
reachable(X,Y) :- edge(X,Y).
reachable(X,Y) :- edge(X,Z),
                  reachable(Z, Y).
```

```
| ?- reachable(X,Y).

X = 1
Y = 2;
```

Type a semi-colon repeatedly

```
X = 2
Y = 3;

X = 2
Y = 4;

X = 1
Y = 3;

X = 1
Y = 4;

no

| ?- halt.
```

Command to Exit XSB

Cut (logic programming)

- Cut (! in Prolog) is a goal which always succeeds, but cannot be backtracked past
- **Green cut**

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X), \+ gotmoney(X).
```

  - **cut says "stop looking for alternatives"**
  - by explicitly writing \+ gotmoney(X), it guarantees that the second rule will always work even if the first one is removed by accident or changed
- **Red cut**

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X).
```