Relational Calculus, Visual Query Languages, and Deductive Databases

Chapter 13
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:

\[ \{ T \mid \text{Condition}(T) \} \]

– \( T \) is the target – a variable that ranges over tuples of values

– \( \text{Condition} \) 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.\text{Semester} = \text{‘F2000’} \}\}

- When a concrete tuple has been substituted for $T$:
  - $\text{Teaching}(T)$ is true if $T$ is in the relational instance of $\text{Teaching}$
  - $T.\text{Semester} = \text{‘F2000’}$ is true if the semester attribute of $T$ has value F2000
  - Equivalent to:

```
SELECT  *  
FROM    Teaching T  
WHERE   T.\text{Semester} = \text{‘F2000’}  
```
Relation Between SQL and TRC

\{ T \mid \text{Teaching}(T) \text{ AND } T.\text{Semester} = 'F2000' \}\n
SELECT *  
FROM Teaching T  
WHERE T.\text{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  
  – \( T.\text{Semester} = 'F2000' \) corresponds to WHERE clause
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
  – $T.A$ oper $S.B$ or $T.A$ oper $const$, where $T$ and $S$ are relation names, $A$ and $B$ are attributes and oper 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 $C_1(\text{X})$)
  – 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

$$\exists \text{X} \in S \ (C_2(\text{X}))$$

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)

\[
\{ S \mid \text{Student}(S) \land (\exists T \in \text{Transcript} \ (S.\text{Id} = T.\text{StudId} \land T.\text{CrsCode} = \text{‘CS305’})) \}
\]

– When a value is substituted for \( S \) the condition has value \text{true} or \text{false}

• There can be only one free variable in a condition (the one that appears in the target)
Example

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

- Returns the set of all course tuples corresponding to the courses that have been taken by every student
TRC Syntax Extension

• We add syntactic sugar to TRC, which simplifies queries and makes the syntax even closer to that of SQL

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

instead of

\{ R \mid \exists S \in \textbf{Student} (R.Name = S.Name) \\
\text{AND } \exists T \in \textbf{Transcript} (R.CrsCode = T.CrsCode) \\
\text{AND ... } \}

where \( R \) is a new tuple variable with attributes \textit{Name} and \textit{CrsCode}
Relation Between TRC and SQL (cont’d)

• List the names of all professors who have taught MGT123
  – In TRC:
    \[
    \{ \text{P.Name} \mid \text{Professor(P)} \land \exists T \in \text{Teaching} \quad (P.Id = T.ProfId \land T.CrsCode = ‘MGT123’) \} \]
  – In SQL:
    \[
    \text{SELECT P.Name } \\
    \text{FROM Professor P, Teaching T } \\
    \text{WHERE P.Id = T.ProfId \land T.CrsCode = ‘MGT123’}
    \]

The Core SQL is merely a syntactic sugar on top of TRC
What Happened to Quantifiers in SQL?

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

- Compare:

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

and

```
SELECT P.\text{Name}
FROM \text{Professor } P, \text{ Teaching } T

\ldots \ldots \ldots
```
Relation Between TRC and SQL (cont’d)

• SQL uses a subset of TRC with simplifying conventions for quantification
• Restrictions 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 to TRC
• 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:
  \[ \exists T \in \text{Transcript} \ (\exists T_1 \in \text{Teaching} \ (\ldots)) \text{ is same as } \exists T_1 \in \text{Teaching} \ (\exists T \in \text{Transcript} \ (\ldots)) \]

• Adjacent existential and universal quantifiers do not commute:
  \[ \exists T \in \text{Transcript} \ (\forall T_1 \in \text{Teaching} \ (\ldots)) \text{ is different from } \forall T_1 \in \text{Teaching} \ (\exists T \in \text{Transcript} \ (\ldots)) \]
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 \( \forall T \in U \ \text{and} \ \exists S \in U \) we simply write
  \[ \forall T \text{ and } \exists T \]
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
    \[
    \text{CSProf} = \{ P.ProfId | \text{Professor}(P) \text{ AND } P.DeptId = 'CS' \}
    \]
  - Then use it
    \[
    \{ S.Id | \text{Student}(S) \text{ AND } \forall P \in \text{CSProf} \exists T \in \text{Teaching} \exists R \in \text{Transcript} (\text{AND } P.Id = T.ProfId \text{ AND } S.Id = R.StudId \text{ AND } T.CrsCode = R.CrsCode \text{ AND } T.Semester = R.Semester) \}
    \]
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):

\[
\{ S.Id \mid \text{Student(S) AND} \\
\quad \forall P \in \text{Professor} \ ( \\
\quad \quad P.DeptId = \text{‘CS’} \rightarrow \\
\quad \exists T1 \in \text{Teaching} \exists R \in \text{Transcript} \ ( \\
\quad \quad \quad P.Id = T1.ProfId \ \text{AND} \ S.Id = R.Id \\
\quad \quad \quad \text{AND} \ T1.CrsCode = R.CrsCode \\
\quad \quad \quad \text{AND} \ T1.Semester = R.Semester \\
\quad \quad ) \\
\quad ) \\
\}
\]

• Why \( P.DeptId = \text{‘CS’} \rightarrow \ldots \) and not \( P.DeptId = \text{‘CS’} \ \text{AND} \ldots \)?
• Read those queries aloud (but slowly) in English and try to understand!
More complex SQL to TRC Conversion

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

```sql
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))
```

versus:

```sql
{R1.A, R2.C | Rel1(R1) AND Rel2(R2) AND condition1(R1, R2)
  AND R2.D = R3.D) }
```

\[ Temp = \{R3.E, R2.C, R2.D | Rel2(R2) AND Rel3(R3)
  AND \exists R4 \in Rel4 (condition2(R2, R3, R4)) \} \]
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:
  \[ \{ X_1, ..., X_n \mid \text{condition}(X_1, ..., X_n) \} \]

• \( X_1, ..., X_n \) is the target: a list of domain variables.

• \( \text{condition}(X_1, ..., X_n) \) is similar to a condition in TRC; uses free variables \( X_1, ..., X_n \).
  – However, quantification is over a domain
    • \( \exists X \in \text{Teaching.CrsCode} \ (\ldots \ldots \ldots) \)
      – i.e., there is \( X \) in Teaching.CrsCode, such that condition is true

• Example: \( \{ \text{Pid, Code} \mid \text{Teaching(Pid, Code, ‘F1997’)} \} \)
  – This is similar to the TRC query:
    \[ \{ T \mid \text{Teaching}(T) \ \text{AND} \ T.\text{ Semester} = ‘F1997’ \} \]
Query Result

• The result of the DRC query
  \[ \{ X_1, \ldots, X_n \mid \text{condition}(X_1, \ldots, X_n) \} \]
  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 \(\text{condition}(x_1, \ldots, x_n)\) 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:
  \[
  \{ \text{Name} \mid \exists \text{Id} \ \exists \text{Dept} \ \ (\text{Professor}(\text{Id}, \text{Name}, \text{Dept}) \ \text{AND} \ \exists \text{Sem} \ \ (\text{Teaching}(\text{Id}, \text{‘MGT123’}, \text{Sem})) \ \}) \}
  \]
  – The universal domain is used to abbreviate the query
  – Note the mixing of variables (\text{Id}, \text{Sem}) and constants (\text{MGT123})

• List names of all professors who ever taught Ann
  \[
  \{ \text{Name} \mid \exists \text{Pid} \ \exists \text{Dept} \ \ (\text{Professor}(\text{Pid}, \text{Name}, \text{Dept}) \ \text{AND} \ \exists \text{Crs} \ \exists \text{Sem} \ \exists \text{Grd} \ \exists \text{Sid} \ \exists \text{Add} \ \exists \text{Stat} \ \ (\text{Teaching}(\text{Pid}, \text{Crs}, \text{Sem}) \ \text{AND} \ \text{Transcript}((\text{Sid}, \text{Crs}, \text{Sem}, \text{Grd}) \ \text{AND} \ \text{Student}((\text{Sid}, \text{‘Ann’}, \text{Addr}, \text{Stat}) \ \})) \}) \}
  \]
  Lots of \exists — 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 | \text{NOT } Q(T)\}: returns the set of all tuples \textit{not} in relation \(Q\)
  – If the attribute domains change, the result set changes as well
  – This is referred to as a \textit{domain-dependent} query

• Another example: \{T| \forall S \ (R(S)) \lor Q(T)\}
  – Try to figure out why this is domain-dependent

• Only \textit{domain-independent} 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 (cont’d)

- Relational algebra (but not DRC or TRC) queries are always domain-independent (prove 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
Relationship between Algebra, TRC, DRC

• Algebra: \( \sigma_{\text{Condition}}(R) \)
• TRC: \( \{ T \mid R(T) \ \text{AND} \ \text{Condition}_1 \} \)
• DRC: \( \{ X_1, \ldots, X_n \mid R(X_1, \ldots, X_n) \ \text{AND} \ \text{Condition}_2 \} \)

• Let \( \text{Condition} \) be \( A=B \ \text{AND} \ C='Joe' \). Why \( \text{Condition}_1 \) and \( \text{Condition}_2 \)?
  – Because TRC, DRC, and the algebra have slightly different syntax:
    \( \text{Condition}_1 \) is \( T.A=T.B \ \text{AND} \ T.C='Joe' \)
    \( \text{Condition}_2 \) would be \( A=B \ \text{AND} \ C='Joe' \)
    (possibly with different variable names)
Relationship between Algebra, TRC, DRC

• Algebra: \( \pi_{A,B,C}(R) \)

• TRC: \( \{ T.A, T.B, T.C \mid R(T) \} \)

• DRC: \( \{ A,B,C \mid \exists D \exists E \ldots R(A,B,C,D,E,...) \} \)

• Algebra: \( R \times S \)

• TRC: \( \{ T.A, T.B, T.C, V.D, V,E \mid R(T) \text{ AND } S(V) \} \)

• DRC: \( \{ A,B,C,D,E \mid R(A,B,C) \text{ AND } S(D,E) \} \)
Relationship between Algebra, TRC, DRC

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

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

<table>
<thead>
<tr>
<th>Professor</th>
<th>Id</th>
<th>Name</th>
<th>DeptId</th>
</tr>
</thead>
<tbody>
<tr>
<td>P._John</td>
<td></td>
<td>MGT</td>
<td></td>
</tr>
</tbody>
</table>

Same, but print all attributes

<table>
<thead>
<tr>
<th>Professor</th>
<th>Id</th>
<th>Name</th>
<th>DeptId</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td></td>
<td>MGT</td>
</tr>
</tbody>
</table>

- Literals that start with “_” are variables.
Joins in QBE

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

<table>
<thead>
<tr>
<th>Professor</th>
<th>Id</th>
<th>Name</th>
<th>DeptId</th>
</tr>
</thead>
</table>
| _123      | P._John

<table>
<thead>
<tr>
<th>Teaching</th>
<th>ProfId</th>
<th>CrsCode</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>_123</td>
<td>MGT123</td>
<td>&lt;&gt; ‘F2002’</td>
<td></td>
</tr>
</tbody>
</table>

Simple conditions placed directly in columns
Condition Boxes

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

<table>
<thead>
<tr>
<th>Transcript</th>
<th>StudId</th>
<th>CrsCode</th>
<th>Semester</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td>CS532</td>
<td></td>
<td>_Gr</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>HasTaught</th>
<th>Professor</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>123456789</td>
<td>567891012</td>
</tr>
</tbody>
</table>
## A Complex Insert Using a Query

### Transcript

<table>
<thead>
<tr>
<th>StudId</th>
<th>CrsCode</th>
<th>Semester</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>_5678</td>
<td>_CS532</td>
<td>_S2002</td>
<td></td>
</tr>
</tbody>
</table>

### Teaching

<table>
<thead>
<tr>
<th>ProfId</th>
<th>CrsCode</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>_12345</td>
<td>_CS532</td>
<td>_S2002</td>
</tr>
</tbody>
</table>

### HasTaught

<table>
<thead>
<tr>
<th>Professor</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>_12345</td>
<td>_5678</td>
</tr>
</tbody>
</table>

### HasTaught

<table>
<thead>
<tr>
<th>Professor</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Connection to DRC

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

<table>
<thead>
<tr>
<th>Transcript</th>
<th>StudId</th>
<th>CrsCode</th>
<th>Semester</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>_123</td>
<td>_CS532</td>
<td>F2002</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

Transcript(X, Y, ‘F2002’, ‘A’)
Pitfalls: Negation

• List all professors who didn’t teach anything in S2002:

<table>
<thead>
<tr>
<th>Professor</th>
<th>Id</th>
<th>Name</th>
<th>DeptId</th>
</tr>
</thead>
<tbody>
<tr>
<td>_123</td>
<td>P.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Teaching</th>
<th>ProfId</th>
<th>CrsCode</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>_123</td>
<td></td>
<td>S2002</td>
<td></td>
</tr>
</tbody>
</table>

• Problem: What is the quantification of CrsCode?

\[
\{ \text{Name} \mid \exists \text{Id} \ \exists \text{DeptId} \ \exists \text{CrsCode} \ ( \text{Professor}(\text{Id},\text{Name},\text{DeptId}) \ \text{AND} \ \neg \text{Teaching}(\text{Id},\text{CrsCode},’S2002’)) \} \\
\]

• Not what was intended(!!), but what the convention about implicit quantification says

or

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

• The intended result!
Negation Pitfall: Resolution

- QBE changed its convention:
  - Variables that occur only in a negated table are *implicitly* quantified with $\forall$ instead of $\exists$
  - For instance: $\text{CrsCode}$ in our example. Note: _123 (which corresponds to $\text{Id}$ in DRC formulation) is quantified with $\exists$, because it also occurs in the non-negated table $\text{Professor}$

- Still, problems remain! Is it

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

  or

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

  Not the same query!
- QBE decrees that the $\exists$-prefix goes first
\[ \exists Id \ \exists DeptId \ \forall CrsCode \ \forall Id \ \exists DeptId \]

\{ Name | \exists Id \ \exists DeptId \ \forall CrsCode \}

\{ Name | \forall CrsCode \ \exists Id \ \exists DeptId \}

... exists a professor such that

Names such that

\[ \text{Professor}(Id, Name, DeptId) \ \\text{AND} \ \\text{NOT Teaching}(Id, CrsCode, 'S2002') \}\}

\[ \text{Professor}(Id, Name, DeptId) \ \\text{AND} \ \\text{NOT Teaching}(Id, CrsCode, 'S2002') \}\}

... for every course

Names such that

For every course

\[ \text{For every course that professor (Id) is not teaching that course(CrsCode)} \]

... exists a professor

... who (Id) is not teaching that course(CrsCode)
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:

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

  
  – In general, \( Prereq_i \) contains all prerequisites up to those that are \( i \)-1 removed for all courses:

    \[
    \pi_{Crs, PreCrs} ((Prereq \bowtie_{PreCrs=Crs} Prereq_{i-1}) [Crs, P1, C2, PreCrs] \\
    \cup Prereq_{i-1})
    \]
Limitations of SQL (con’t)

• **Question**: We can compute $\sigma_{Crs='CS632'}(Prereq_i)$ 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 $Prereq_i = Prereq_{i+1}$ 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:

```sql
CREATE RECURSIVE VIEW IndirectPrereq (Crs, PreCrs) AS
(a) 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
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
  - \( \text{IndirectPrereq}_{i+1} \) is obtained by taking the union of \( \text{IndirectPrereq}_i \) with the result of the query
    
    \[
    \begin{align*}
    \text{SELECT} & \quad P.Crs, I.PreCrs \\
    \text{FROM} & \quad \text{Prereq} \ P, \text{IndirectPrereq}_i \ I \\
    \text{WHERE} & \quad P.PreCrs = I.Crs
    \end{align*}
    \]

  - Successive values of \( \text{IndirectPrereq}_i \) are computed until a stable state is reached, i.e., when the result of the query \( (\text{IndirectPrereq}_{i+1}) \) is contained in \( \text{IndirectPrereq}_i \)
Recursion in SQL:1999

• Also provides the WITH construct, which does not require views.

• Can even define mutually recursive queries:

```sql
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/](http://www.w3.org/TR/rif-bld/)
Basic Syntax

• Rule:
  \( \text{head} : \text{body}. \)

• Query:
  \( ?- \text{body}. \)

• \textit{body}: any DRC expression without the quantifiers.
  
  • \textit{AND} is often written as ‘,’ (without the quotes)
  
  • \textit{OR} is often written as ‘;’

• \textit{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.

• \textit{body} in a rule and in a query has the same syntax.
Basic Syntax (cont’d)

NameSem(\(?Name,?Sem\)) : – Prof(?Id,?Name,?Dept), Teach(?Id,’MGT123’,?Sem).
? – NameSem(?Name,?Sem).

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:

\[
\text{NameSem} = \{ Name, Sem \mid \exists Id \exists Dept ( \text{Prof}(Id, Name, Dept) \ \text{AND} \\
\text{Teaching}(Id, ‘MGT123’, Sem) ) \} 
\]
Basic Semantics (cont’d)

• Another way to understand rules:

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

- If these tuples exist
- Then this one must also exist

As in DRC, join is indicated by sharing variables
Union Semantics of Multiple Rules

- Consider rules with the same head-predicate:
  
  NameSem(\(\?\text{Name}, \?\text{Sem}\)) : – \(\text{Prof}(\?\text{Id}, \?\text{Name}, \?\text{Dept})\), Teach(\?\text{Id}, ’MGT123’, \?\text{Sem}).

  NameSem(\(\?\text{Name}, \?\text{Sem}\)) : – \(\text{Prof}(\?\text{Id}, \?\text{Name}, \?\text{Dept})\), Teach(\?\text{Id}, ’CS532’, \?\text{Sem}).

- Semantics is the union:
  
  NameSem = \{ \text{Name}, \text{Sem} \mid \exists \text{Id} \exists \text{Dept} ( 
  \text{Prof}(\text{Id}, \text{Name}, \text{Dept}) \text{ AND } \text{Teaching}(\text{Id}, ‘MGT123’, \text{Sem}) 
  \text{ OR } \text{Prof}(\text{Id}, \text{Name}, \text{Dept}) \text{ AND } \text{Teaching}(\text{Id}, ‘CS532’, \text{Sem}) 
  ) \} 

  Equivalently:

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

- Above rules can also be written in one rule:

  NameSem(\(\?\text{Name}, \?\text{Sem}\)) : – \(\text{Prof}(\?\text{Id}, \?\text{Name}, \?\text{Dept})\),

  \(\text{( Teach}(\?\text{Id}, ‘MGT123’, \?\text{Sem}) \text{ ; Teach}(\?\text{Id}, ‘CS532’, \?\text{Sem}) \text{ )}.\)
Recursion

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

\[
\text{IndirectPrereq}(\text{?Crs}, \text{?Pre}) : \text{- } \text{Prereq}(\text{?Crs}, \text{?Pre}).
\]
\[
\text{IndirectPrereq}(\text{?Crs}, \text{?Pre}) : \text{- }
\]
\[
\text{Prereq}(\text{?Crs}, \text{?Intermediate}),
\]
\[
\text{IndirectPrereq}(\text{?Intermediate}, \text{?Pre}).
\]
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: $H : \neg (B;C)$ is equivalent to $H : \neg B$ and $H : \neg C$ because
      – $H : \neg B$ 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

\[ \text{HeadRelation} (\text{HeadVars}) : \neg \text{Body} \]

can be represented in DRC as

\[ \text{HeadRelation} = \{ \text{HeadVars} \mid \exists \text{BodyOnlyVars} \ \text{Body} \} \]

• We call this \textit{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
1. \textit{CurrentState} := \textit{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. \textit{NewState} := the database where instances of all derived relations have been replaced as in Step 2 above

4. \textbf{if} \textit{CurrentState} = \textit{NewState} \textbf{then} \textit{Stop: NewState} is the stable state that represents the meaning of that set of Datalog rules on the given DB
   \textbf{else} \textit{CurrentState} := \textit{NewState}; \textbf{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
      \[ \exists \text{Vars} (A \text{ and/or } B \text{ and/or } C \ldots) \]
      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 \( O(D^N \times D^N) \)
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*

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

versus

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

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).
```

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

```
HasTaught(?Pid, ?Sid) :- … … …
```

```
?- Answer(?Sid).
```

- What’s wrong?
- So, 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
    
    \[
    \{ \text{Name} \mid \forall \text{Crs}\exists \text{Grade}\exists \text{Sid} \\
    \text{(Student}(\text{Sid, Name}), \\
    \text{(Teaching}(1234567, \text{Crs,’S2006’}) \\
    \Rightarrow \text{Transcript}(\text{Sid, Crs,’S2006’,Grade}))) \}
    \]
  
  - Datalog
    
    \[
    \text{Answer(}\text{Name}) : - \text{Student(}\text{Sid, Name}), \\
    \text{not DidntTakeS2006CrsFrom1234567(}\text{Sid}). \\
    \text{DidntTakeS2006CrsFrom1234567(}\text{Sid}) : - \\
    \text{Teaching}(1234567, \text{Crs,’S2006’}), \text{not TookS2006Course(}\text{Sid,}\text{Crs)}. \\
    \text{TookS2006Course(}\text{Sid,}\text{Crs}) : - \text{Transcript(}\text{Sid,}\text{Crs,‘S2006’,}\text{Grade}). \\
    \]

  - **Pitfall:** Transcript(\text{Sid,}\text{Crs,’S2006’,}\text{Grade}) here won’t do because of \exists \text{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 won’t 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) \Rightarrow \text{Query result: } \{b, c\}

Add \ Q(b) \Rightarrow \text{Query result shrinks: just } \{c\}
“Well-behaved” Negation

- Negation is “well-behaved” if there is no recursion through it

\[
\begin{align*}
P(?X,?Y) & :\quad Q(?X,?Z), \quad \text{not } R(?X,?Y). \\
Q(?X,?Y) & :\quad P(?X,?Z), \quad R(?X,?Y). \\
R(?X,?Y) & :\quad S(?X,?Z), \quad R(?Z,?V), \quad \text{not } T(?V,?Y). \\
R(?X,?Y) & :\quad V(?X,?Z).
\end{align*}
\]

**Evaluation method for P:**

1. Compute \( T \), then its complement, \( \text{not } T \)
2. Compute \( R \) using the Negation-free Datalog algorithm. Treat \( \text{not } T \) as base relation
3. Compute \( \text{not } R \)
4. Compute \( Q \) and \( P \) using Negation-free Datalog algorithm. Treat \( \text{not } R \) 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).

Cycle through negation in dependency graph

Dependency graph
Dependency Graph for a Ruleset $\mathbf{R}$

- **Nodes**: relation names in $\mathbf{R}$
- **Arcs**:
  - if $\text{P}(\ldots) : - \ldots, \text{Q}(\ldots), \ldots$ is in $\mathbf{R}$ then the dependency graph has a *positive* arc $\text{Q} \rightarrow \mathbf{R}$
  - if $\text{P}(\ldots) : - \ldots, \text{not } \text{Q}(\ldots), \ldots$ is in $\mathbf{R}$ then the dependency graph has a *negative* arc $\text{Q} \rightarrow \mathbf{R}$ (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, $\pi$, to a stratum $\varphi$, 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:
$$\pi_3, \pi_5, \pi_4, \pi_2, \pi_1$$

Another stratification:
$$\pi_5, \pi_4, \pi_3, \pi_2, \pi_1$$
Stratifiable Rule sets

• This is what we meant earlier by “well-behaved” rule sets.
• 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 $\mathbf{R}$ be a ruleset and let $\pi_1, \pi_2, \ldots, \pi_n$ be a stratification.

• Then the rules of $\mathbf{R}$ 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 $\pi_i$. 
Evaluation of a Stratifiable Ruleset, $\mathbf{R}$

1. Partition the relations of $\mathbf{R}$ 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 $\textit{not } P$ occurs in $Q_2$, then treat it as a new base relation, $\textit{NotP}$, 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 evaluate Datalog rulesets. But this becomes a rather complicated stuff, which we won’t touch. (Refer to the bibliographic notes, if interested.)
The Flora-2 Datalog System

• We will use Flora-2 for Project 1
• Download: http://flora.sourceforge.net/ (take the latest release for your OS, currently 1.2)
  – Can also use Ergo Suite from coherentknowledge.com/free-trial – has IDE and other bells & whistles.
• Not just a Datalog system – it is a complete programming language, called Rulelog, which happens to support Datalog
• Has a number of extensions, some of which you need to know about for the project
Differences

- **Variables**: as in this lecture (start with a `?`)
- Each occurrence of a singleton symbol `?` or `?_` 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**:
  - Alphanumeric starting with a letter:
    - Example: `Abc, aBC123, abc_123, John`
  - or enclosed in single quotes
    - Example: `'abc &, % (, foobar1'`
    - Note: `abc and 'abc'` refer to the same thing
- **And**: comma `,` or `\and`
- **Or**: semicolon `;` or `\or`
Differences (cont’d)

• Negation: called \texttt{naf} (negation as failure)
  – Note: Flora-2 also has \texttt{neg}, but it’s a different thing – don’t use!
  – Use instead:
    \[
    \ldots : - \ldots, \texttt{naf} \texttt{foobar}(\texttt{?X}), \texttt{naf}(\texttt{abc}(\texttt{?X},\texttt{?Y}),\texttt{cde}(\texttt{?Y})).
    \]

• All variables under the scope of \texttt{naf} must also occur in the body of the rule in other non-negated relations:
  \[
  \textit{something} : - \texttt{p}(\texttt{?X}), \texttt{naf} \texttt{foobar}(\texttt{?X},\texttt{?Y}), \texttt{q}(\texttt{?Y}), \ldots
  \]
  – If not, that variable is implicitly existentially quantified and will likely have \textit{undefined} truth value:
    \[
    \textit{somethingelse} : - \texttt{p}(\texttt{?X},\texttt{?Z}), \texttt{naf} \texttt{foobar}(\texttt{?X},\texttt{?Y}), \ldots
    \]
Overview of Installation

- **Windows**: download the installer, double-click, follow the prompts
- **Linux/Mac**: Download the flora2.run file, put it where appropriate, then type
  ```sh
  sh flora2.run
  ```
  then follow the prompts.
- Consult [http://flora.sourceforge.net/installation.html](http://flora.sourceforge.net/installation.html) for the details, if necessary.
Use of Flora-2

• Put your ruleset and data in a file with extension .flr
  
  p(?X) :- q(?X,?). // a rule
  q(1,a). // a fact
  q(2,a).
  q(b,c).
  :- p(?X). // a query (starts with a ?-)

• Don’t forget: all rules, queries, and facts end with a period (.)
• Comments: /*…*/ or //.... (like in Java/C++)
• Type
  
  .../flora2/runflora (Linux/Mac)
  ...lora2\runflora (Windows)

  where … is the path to the download directory

  In Windows, you will also see a desktop icon, which you can double-click.

• You will see a prompt
  flora2 ?-

  and are now ready to type in queries
Use of Flora-2 (cont’d)

• Loading your program, myprog.f1r

  flora2 ?- [myprog]. // or
  flora2 ?- [‘H:/abc/cde/myprog’]. // note: / even in windows (or \)

  Flora-2 will compile myprog.f1r (if necessary) and load it. Now you can type further queries. E.g.:
  flora2 ?- p(?X).
  flora2 ?- p(1).
  etc.
Some Useful Built-ins

• write(?X)@\io – write whatever ?X is bound to
• writeln(?X)@\io – write then put newline
  • E.g., write(‘Hello World’)@\io.
  • ?X = ‘Hello World’, writeln(?X)@\io.
• nl@\io – output newline
• Equality, comparison: =, >, <, >=, <=
• Inequality: !=
• Lexicographic comparison: @>, @<
• You might need more, so take a look at the manual, if necessary:
  – You should need very little additional info from that manual, if at all.
Arithmetics

• If you need it: use the builtin \texttt{\textbackslash is}
  \[
  \texttt{p(1). p(2).}
  \]
  \[
  \texttt{q(?X) :- p(?Y), ?X \texttt{\textbackslash is} ?Y*2.}
  \]
  Now q(2), q(4) will become true.

• Note:
  \[
  \texttt{q(2*?X) :- p(?X).}
  \]
  will not do what you might think it would do.

It will make \( q(2*1) \) and \( q(2*2) \) true, where \( 2*1 \) and \( 2*2 \) are expressions, \textit{not} numbers.

\( 2*1 \neq 2 \) and \( 2*2 \neq 4 \) (no need to get into all that now)
Some Useful Tricks

• Flora-2 returns all answers to queries:
  
  ```prolog
  flora2 ?- q(?X).
  ?X = 2
  ?X = 4
  Yes
  flora2 ?-
  
  ?X = 1
  ?Y = 2
  ?Z = 3
  ?X = 2
  ?Y = 5
  ?Z = 7
  
  Vs.
  
  flora2 ?- q(?_Y, ?Z).
  ?X=1
  ?Z=3
  ?X=2
  ?Z=7
  ```

• Anonymous variables: start with a ?_. Used to avoid printing answers for some vars. Eg.,

  ```prolog
  p(1,2). q(2,3).
  p(2,5). q(5,7).
  p(a,b). q(c,d).
  vs.
  ```
Useful Tricks (cont’d)

• More on anonymous variables:

\[ p(?X,?Y) :- q(?Y,?Z,?W), r(?Z). \]

– Will issue 3 warnings:
  a) Head-only variable ?X  
  b) Singleton variable ?X  
  c) Singleton variable ?W  

– Don’t ignore these warnings!!

• Use anonymous vars to pacify the compiler:

\[ p(?_X,?Y) :- q(?Y,?Z,?_W), r(?Z). \]
Aggregate Functions

• \( \text{func}\{\text{ResultVar}[\text{GroupVar1},\ldots,\text{GroupVarN}] \mid \text{condition} \} \)
  
  – \( \text{func} \) can be \( \text{avg}, \text{min}, \text{max}, \text{sum}, \text{count}, \text{some others} \)

emp(John,CS,100). emp(Mary,CS,200).
emp(Bob,EE,75). emp(Hugh,EE,160). emp(Ugo,EE,300).
emp(Alice,Bio,200).
?- ?X = avg\{?Sal[?Dept] \mid \text{emp(?_Emp, ?Dept, ?Sal)}\}.

?X = 150.0000
?Dept = CS

?X = 178.3333
?Dept = EE

?X = 200.0000
?Dept = Bio

Anonymous – don’t want in answers
Quantifiers

• Supports explicit quantifiers: exist and forall. Also some, exists, all, each.

?- Student(?Stud,?_Name,?_Addr) \and
    forall(?Prof)^exist(?Crs,?Sem,?Grd)^(
        Teaching(?Prof,?Crs,?Sem) \implies
        Transcript(?Stud,?Crs,?Sem,?Grd)
    )
).

• Students (?Stud) who took a course from every teaching professor
Quantifiers (cont’d)

• Students (?Stu) who took a course from every CS prof:

?- Student(?Stu,?_Name,?_Addr) \and
   forall(?Prof)^exist(?Crs,?Sem,?Grd)^(
       Professor(?Prof,CS) \implies
       Teaching(?Prof,?Crs,?Sem),
       Transcript(?Stu,?Crs,?Sem,?Grd)
   ).

Slightly different from the previous query because this implies that every professor must have taught something. E.g., excludes some research or visiting professors.