Course Notes on
The Logical Structure of Relational Query Languages

The Logical Structure of Relational Query Languages: Topics

- Overview
- First-order logic
- Tuple Relational Calculus (TRC)
- Domain Relational Calculus (DRC)
On the Way to SQL: Relational Calculi

- Historically, SQL was a major advance over older database languages (like DL/I of IMS or DDL, DML of CODASYL DBTG) because SQL is far easier to use
- To effectively master and use SQL up to relational completeness, first mastering first-order logic makes things significantly easier

Logic as a Basis for Database Languages

- First-order logic (predicate calculus) is simple at the level needed for relational languages
- Strong historical prejudice against logic (and theory) in the user world
- Formal definitions have many advantages
  - the ultimate reference document
  - test of language consistency during design
  - need not be shown to everybody
- Logic has become a basic formalism in informatics for e.g.,
  - assertions in programming
  - integrity formulation and maintenance in DBMS
  - data models of DBMS
  - semantics of programming languages
Relational Calculi

- More used than the algebra as a basis for user languages
- Directly based on first-order logic ⇒ regular, systematic structure
- Less procedural than the algebra: what versus how
- Relational completeness:
  - DRC, TRC, and algebra have same expressive power
  - SQL is slightly more powerful: some computation, ordering, etc.

TRC and DRC

- **Domain Relational Calculus (DRC)**
  - Most similar to logic as a modeling language
  - Typical modeling formalism in AI and natural-language studies: data is viewed as objects with properties
- **Tuple Relational Calculus (TRC)**
  - Reflects traditional pre-relational file structures
  - Closer to a view of relations implemented as files
A Simple Introduction to Logic

• General form of first-order logic is not necessary
• Logic is applied to a fixed domain of reference: the DB extension
• Formal system =

\[
\begin{align*}
\{ & \text{formal language (syntax + semantics)} \\
& \text{deductive mechanisms} \}
\end{align*}
\]

• Here we basically need the syntax of logic, and a simple “applied” semantics linked to the DB extension
• The language of logic is used to combine elementary DB facts

• Simple and intuitive introductions to logic:
  
  
The Structure of First-Order Logic

- The universe of reference is the current database
- **Elementary propositions**: express assertions that are true or false in the universe
- **Propositional connectives** ($\land$, $\lor$, $\rightarrow$, $\neg$, $\leftrightarrow$) combine propositions

<table>
<thead>
<tr>
<th></th>
<th>$P$</th>
<th>$Q$</th>
<th>$P \land Q$</th>
<th>$P \lor Q$</th>
<th>$P \rightarrow Q$</th>
<th>$\neg P$</th>
<th>$P \leftrightarrow Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$T$</td>
<td></td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$F$</td>
<td></td>
<td>$F$</td>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$T$</td>
<td></td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$F$</td>
<td></td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td></td>
</tr>
</tbody>
</table>

- Elementary propositions:
  - $P_1$: Smith was born on 09-Jan-55 is true in the current state of the world (i.e., of the database)
  - $P_2$: Smith is female is false
- Compound propositions:
  - $P_1 \land P_2 = $ Smith was born on 09-JAN-55 $\land$ Smith is female is false
  - $\neg P_2 = $ Smith is not female is true
- Much of the problem with the intuition of logic comes from implication, namely, with the fact that $P \rightarrow Q$ is true when $P$ is false
Relational Schema for the Company Example

<table>
<thead>
<tr>
<th>Employee</th>
<th>SSN</th>
<th>FName</th>
<th>LName</th>
<th>BDate</th>
<th>Address</th>
<th>Sex</th>
<th>Salary</th>
<th>SuperSSN</th>
<th>DNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department</td>
<td>DNumber</td>
<td>DName</td>
<td>MgrSSN</td>
<td>MgrStartDate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeptLocations</td>
<td>DNumber</td>
<td>DLocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>PNumber</td>
<td>PName</td>
<td>PLocation</td>
<td>DNumber</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WorksOn</td>
<td>PNo</td>
<td>ESSN</td>
<td>Hours</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependent</td>
<td>ESSN</td>
<td>DependentName</td>
<td>Sex</td>
<td>BDate</td>
<td>Relationship</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quantifiers

- Use variables to express more general assertions about the DB:
  - F1: there exists an employee who was born on 09-Jan-55 is true
  - F2: all employees were born on 09-Jan-85 is false, or
    - there is at least one employee who was not born on 09-Jan-85 is true
  - F3: all employees born after 1950 earn more than 40k is false, or
    - there is at least one employee born after 1950 who earns less than 40k is true
- More formally
  - F1: \( \exists e (e \text{ is an employee} \land e \text{ was born on 09-Jan-55}) \)
  - F2: \( \neg \forall e (e \text{ is an employee} \rightarrow e \text{ was born on 09-Jan-55}) \), or
  - \( \exists e (e \text{ is an employee} \land e \text{ was not born on 09-Jan-55}) \)
  - F3: \( \neg \forall e (e \text{ is an employee} \land e \text{ was born after 01-Jan-50} \rightarrow e \text{ earns more than 40k}) \), or
  - \( \exists e (e \text{ is an employee} \land e \text{ was born after 01-Jan-50} \land e \text{ earns less than 40k}) \)
• ∀ (for all) and ∃ (there exists)
• if you cannot do everything ...
  ◦ that does not mean that there is not anything that you can do ...
  ◦ nor that there is anything that you cannot do ...

Queries

• Free variables of logic are used as query variables
• List the employees who were born on 09-Jan-55
  \{e | e is an employee ∧ e was born on 09-Jan-55\}

• The \{e | P(e)\} syntax evokes set theory
• A more fancy syntax for the same expression (see later)
  SELECT ... FROM ... WHERE ...
Equivalence Rules

- Allow to replace a formula by another one

\[ P \rightarrow Q \] is equivalent to \[ \neg P \lor Q \]
\[ \neg(P \land Q) \] is equivalent to \[ \neg P \lor \neg Q \]
\[ \neg(P \lor Q) \] is equivalent to \[ \neg P \land \neg Q \]
\[ \forall x P(x) \] is equivalent to \[ \neg(\exists x \neg P(x)) \]
\[ \exists x P(x) \] is equivalent to \[ \neg(\forall x \neg P(x)) \]
\[ \exists x (\neg P(x)) \] is equivalent to \[ \neg(\forall x P(x)) \]

- Implication rules for quantifiers

\[ \forall x P(x) \] implies that \[ \exists x P(x) \]
\[ \neg(\exists x P(x)) \] implies \[ \neg(\forall x P(x)) \]

but not the converse

- This is about all the logic that is needed to master languages of traditional relational systems
### Tuple Relational Calculus (TRC)

- **Tuple variables:**
  - Range on (takes as values) tuples of a relation
  - Are explicitly linked to a relation

- List employees who make more than 50k
  \[
  \{ t \mid \text{Employee}(t) \land t.\text{Salary} > 50k \}
  \]
  - Employee(t) is a “relation predicate”, it links TRC with the DB
  - t.Salary is a term whose value is the value of attribute Salary of tuple t

- List birthdate and address of employees called John Smith
  \[
  \{ t.\text{BDate}, t.\text{Address} \mid \text{Employee}(t) \land t.\text{FName} = 'John' \land t.\text{LName} = 'Smith' \}
  \]

### General Structure of TRC Queries

\[
\{ t_1.A_1, t_2.A_2, \ldots, t_n.A_n \mid F(t_1, \ldots, t_n, t_{n+1}, \ldots, t_m) \}
\]

- \( t_1, t_2, \ldots, t_m \): tuple variables each associated in \( F \) with a relation through a relation predicate
- \( A_i \): attribute of the relation associated with \( t_i \)
- \( F \): logical formula containing variables \( t_1, t_2, \ldots, t_m \)
- \( t_1, t_2, \ldots, t_n \): free variables in \( F \) (“query variables”)
- \( t_{n+1}, \ldots, t_m \): variables quantified in \( F \)
TRC Semantics

- $F$ is evaluated for all possible values $t_1, t_2, \ldots, t_n$ (= Cartesian product)
- If $F$ is true for a tuple, then the projection $t_1.A_1, t_2.A_2, \ldots, t_n.A_n$ is included in the result
- Result = nameless relation with $n$ attributes; rules must be specified for deciding attribute names (e.g., $A_i$’s if they are all distinct)

Structure of TRC Formulas

- Formula $F$ is defined with the recursive structure of first-order logic
  - $R(t_i)$, where $R$ is a relation name
  - $t_i.A$ comparison $t_j.B$
  - $t_i.A$ comparison constant
  - $\neg F$
  - $F_1 \land F_2$
  - $F_1 \lor F_2$
  - $F_1 \rightarrow F_2$
  - $F_1 \leftrightarrow F_2$
  - $\exists t\ F(t)$
  - $\forall t\ F(t)$
- Comparison: $=, \neq, <, >, \leq, \geq$
Join

- List name and address of employees who work for the Research department

\[ \{ e.\text{LName}, e.\text{Address} \mid \text{Employee}(e) \land \exists d (\text{Department}(d) \land d.\text{DName} = \text{‘Research’} \land d.\text{DNumber} = e.\text{DNo}) \} \]

- “Join term” \( d.\text{DNumber} = e.\text{DNo} \) expresses a join between relation Department and relation Employee

Relative Procedurality of Languages

- Two different algebraic formulations for the previous example:
  \[ \pi_{\text{LName, Address}}(\sigma_{\text{DName} = \text{‘Research’}}(\text{Employee} \bowtie_{\text{DNo}=\text{DNumber}} \text{Department})) \]
  \[ \pi_{\text{LName, Address}}(\text{Employee} \bowtie_{\text{DNo}=\text{DNumber}} (\sigma_{\text{DName} = \text{‘Research’}}(\text{Department}))) \]

- Only one TRC formulation

\[ \{ e.\text{LName}, e.\text{Address} \mid \text{Employee}(e) \land \exists d (\text{Department}(d) \land d.\text{DName} = \text{‘Research’} \land d.\text{DNumber} = e.\text{DNo}) \} \]

- The algebra is more procedural than TRC: in TRC, the relative order of join and selection is not an issue
- For casual users, TRC style is simpler than algebra style (less to think about)
- Efficiency is another issue
• Efficiency:
  ◦ in most cases, the strategy that evaluates selection before joins is more efficient
  ◦ this is taken care of by the query optimizer of the DBMS

Two Joins

• For every project located in Brussels, list the project number, the controlling department number, and the name of the department manager

\[
\{ p.PNumber, p.DNum, m.LName \mid \text{Project}(p) \land \\
\text{Employee}(m) \land p.\text{Location} = 'Brussels' \land \\
\exists d (\text{Department}(d) \land d.\text{DNumber} = p.\text{DNum} \land d.\text{MgrSSN} = m.\text{SSN}) \}
\]

• Same conclusion about procedurality: algebra is more procedural

• In this example, if \( p.\text{DNum} \) is replaced by \( d.\text{DNumber} \) in the target of the query, then the quantifier \( \exists d \) disappears, yielding a more symmetric formulation

\[
\{ p.\text{PNumber}, d.\text{DNumber}, m.\text{LName} \mid \\
\text{Project}(p) \land \text{Employee}(m) \land \text{Department}(d) \land \\
p.\text{Location} = \text{Brussels} \land d.\text{DNumber} = p.\text{DNum} \land d.\text{MgrSSN} = m.\text{SSN} \}
\]
Other Example with two Joins

- List the name of employees who work on some project controlled by department number 5

\[
\{ e.\text{FName}, e.\text{LName} \mid \text{Employee}(e) \land \\
\exists p \exists w (\text{Project}(p) \land \text{WorksOn}(w) \land \\
p.\text{DNum} = 5 \land w.\text{ESSN} = e.\text{SSN} \land p.\text{PNumber} = w.\text{PNo}) \}
\]

- Same conclusion about procedurality: algebra is more procedural

A “Complex” Query

- List project names of projects for which an employee whose last name is Smith is a worker or a manager of the department that controls the project

\[
\{ p.\text{PName} \mid \text{Project}(p) \land \\
\exists e \exists w (\text{Employee}(e) \land \text{WorksOn}(w) \land \\
w.\text{PNo} = p.\text{PNumber} \land w.\text{ESSN} = e.\text{SSN} \land e.\text{LName} = \text{‘Smith’}) \lor \\
\exists m \exists d (\text{Employee}(m) \land \text{Department}(d) \land \\
p.\text{DNum} = d.\text{DNumber} \land d.\text{MgrSSN} = m.\text{SSN} \land m.\text{LName} = \text{‘Smith’}) \}
\]

- Union of two queries in the algebra is expressed in TRC with disjunction
• \( \{ x \mid P(x) \lor Q(x) \} \equiv \{ x \mid P(x) \} \cup \{ x \mid Q(x) \} \)

• Other version: factor out of the disjunction the repeated

\[ \exists e \left( \text{Employee}(e) \land e.\text{LName} = \text{Smith} \right) \]

---

**Join of a Relation with Itself**

• List the first and last name of each employee, and the first and last name of his/her immediate supervisor

\[
\{ e.\text{FName}, e.\text{LName}, s.\text{FName}, s.\text{LName} \mid \\
\text{Employee}(e) \land \text{Employee}(s) \land e.\text{SuperSSN} = s.\text{SSN} \}
\]

• The attributes of the result relation have to be specified explicitly (if the result is to be used elsewhere, i.e., not just displayed) through some kind of assignment

\[
F(\text{EmpFN}, \text{EmpLN}, \text{MgrFN}, \text{MgrLN}) \leftarrow \{ \ldots \}
\]

• Syntax is more difficult for the algebra, unless attributes are ordered
Other Example of Join of a Relation with Itself

- List the SSN of employees who have both a dependent son and a dependent daughter

\[
\{ e.\text{ESSN} \mid \text{Dependent}(e) \\
\land \exists d (\text{Dependent}(d) \\
\land e.\text{ESSN} = d.\text{ESSN} \\
\land d.\text{Relationship} = \text{‘Son’} \\
\land d.\text{Relationship} = \text{‘Daughter’}) \}
\]

Universal Quantifier

- List the name of employees who work on all projects

\[
\{ e.\text{FName}, e.\text{LName} \mid \text{Employee}(e) \\
\land \forall p \text{Project}(p) \rightarrow \\
\exists w (\text{WorksOn}(w) \land w.\text{PNumber} = p.\text{PNumber} \land w.\text{ESSN} = e.\text{SSN}) \}
\]

- “all projects” are those in relation Project
• **Various styles of universal quantification** (for List the employees who work on all projects):
  
  ◦ logical formulation:
    \[ \{ e \mid \text{Employee}(e) \land \forall p \text{ Project}(p) \rightarrow \text{Workson}(e,p) \} \]
  
  ◦ logic with range-coupled quantifiers:
    \[ \{ e \in \text{Employee} \mid \forall p \in \text{Project} \text{ Workson}(e,p) \} \]
  
  ◦ towards natural language (where quantification is “infix” rather than “prefix” as in logic, binary predicates are also infix rather than prefix, and variables are seldom used as such):
    * \( \{ e \in \text{Employee} \mid \text{for all } p \in \text{Project} \text{ Workson } p \} \)
    * \( \{ e \in \text{Employee} \mid e \text{ Workson(all } p \in \text{Project}) \} \)
    * \( \{ \text{Employee Workson (all Project)} \} \)

---

**Universal Quantifier**

• List the name of employees who have at least one dependent

\[
\{ e.\text{LName} \mid \text{Employee}(e) \land \\
\exists d \text{ (Dependent}(d) \land e.\text{SSN} = d.\text{ESSN}) \}
\]

• List the name of employees who have no dependent

\[
\{ e.\text{LName} \mid \text{Employee}(e) \land \\
\neg \exists d \text{ (Dependent}(d) \land e.\text{SSN} = d.\text{ESSN}) \}
\]

\[
\{ e.\text{LName} \mid \text{Employee}(e) \land \\
\forall d \text{ (Dependent}(d) \rightarrow e.\text{SSN} \neq d.\text{ESSN}) \}
\]

---

• Proof of equivalence of the formulations of *List the name of employees who have no dependent* by applying the equivalence rules of logic:
  
  ◦ \( \neg (\exists d P(d)) \equiv \forall d \ (\neg P(d)) \)
  
  ◦ \( \neg \exists d \text{ (Dependent}(d) \land e.\text{SSN} = d.\text{ESSN}) \)
  
  ◦ \( \forall d \ (\neg \text{(Dependent}(d) \land e.\text{SSN} = d.\text{ESSN}) \)
  
  ◦ \( \forall d \ (\neg \text{(Dependent}(d) \lor \neg (e.\text{SSN} = d.\text{ESSN}) \)
  
  ◦ \( \forall d \ (\neg \text{Dependent}(d) \lor e.\text{SSN} \neq d.\text{ESSN}) \)
  
  ◦ \( \forall d \ (\text{Dependent}(d) \rightarrow e.\text{SSN} \neq d.\text{ESSN}) \)
  
  ◦ \( \forall d \in \text{Dependent} \ (e.\text{SSN} \neq d.\text{ESSN}) \)
Safe Use of Universal Quantification

- Universal quantification must always be associated with implication.
- Given relations $\text{Prereq}(\text{Course}, \text{Pre})$ and $\text{Took}(\text{StudID}, \text{Course})$, give the names of students who took all prerequisites of the course Math210.
- Use of $\land$ instead of $\to$:

$$\{s.\text{Name} \mid \text{Student}(s) \land \forall p \,(\text{Prereq}(p) \land p.\text{Course} = \text{Math210}) \land \exists t \,(\text{Took}(t) \land t.\text{StudID} = s.\text{StudID} \land t.\text{Course} = p.\text{Pre})\}$$

- If Math210 has no prerequisites, the answer of the above query is always empty.
- Correct formulation:

$$\{s.\text{Name} \mid \text{Student}(s) \land \forall p \,(\neg \text{Prereq}(p) \land p.\text{Course} = \text{Math210}) \land \exists t \,(\text{Took}(t) \land t.\text{StudID} = s.\text{StudID} \land t.\text{Course} = p.\text{Pre})\}$$

- If Math210 has no prerequisites, the answer will be the names of all students.
Safe TRC

- Formulas with quantifiers, negation, some comparisons must be restricted so as to be meaningful

- Examples of ill-formed formulas with a comparison, a negation
  - \(\{n \mid n \geq 3\}\)
  - \(\{e \mid \neg \text{Employee}(e)\}\)

- Existential quantifiers
  - \(\exists t \ F(t)\) must have the form \(\exists t \ R(t) \land F'(t)\)
  - other notation: \((\exists t \in R) \ F'(t)\)

- Universal quantifiers must always be associated with implication
  - \(\forall t \ F(t)\) must have the form \(\forall t \ R(t) \rightarrow F'(t)\)
  - other notation: \((\forall t \in R) \ F'(t)\)

- \((\exists t \in R)\) and \((\forall t \in R)\) are called range-restricted or ranged-coupled quantifiers, where \(R\) is a relation predicate that defines and restricts the range of \(t\)

- General form of safe use of universal quantifier: \(\forall t \in (R(t) \land F'(t)) \ F''(t)\) (\(F'(t)\) and \(F''(t)\) are any TRC formulas)

- Intuition: \(\forall t \ F(t)\), where \(F(t)\) is a conjunction of database or comparison predicates, is meaningless (e.g., \(\forall t \ \text{Employee}(t)\))
Domain Relational Calculus (DRC)

• **Domain variables**\range on (i.e., take as values elements of) DB domains

• Relations are preferably viewed as predicates expressing properties of objects, represented as values

• **Relation predicates** (extensional predicates)
  - realize the link between DRC and the DB
  - $R(A_1: x_1, \ldots, A_n: x_n)$ is associated with relation $R(A_1: D_1, \ldots, A_n: D_n)$
  - $R(A_1: a_1, \ldots, A_n : a_n)$ is true if tuple $\langle A_1: a_1, \ldots, A_n : a_n \rangle$ belongs to relation $R$

- Predicate \texttt{WorksOn(ESSN:123456789, PNo:1, Hours:32.5)} is true because tuple $\langle \text{ESSN:123456789, PNo:1, Hours:32.5} \rangle$ belongs to relation \texttt{WorksOn}

- In \texttt{WorksOn(ESSN:123456789, PNo:1, Hours:32.5)}:
  - \texttt{WorksOn(E:SSN: , PNo: , Hours: )} is the predicate name
  - 123456789, 1 and 32.5 are the arguments
General Structure of DRC Queries

\{x_1, x_2, \ldots, x_n \mid F(x_1, \ldots, x_n, x_{n+1}, \ldots, x_m)\}

- where formula F has the structure of first-order logic
  - \( R(A_i : x_i, \ldots, A_j : x_j) \), where \( R \) is a relation name
  - \( x_i \) comparison \( x_j \)
  - \( x_i \) comparison constant
  - \( \neg F \)
  - \( F_1 \wedge F_2 \)
  - \( F_1 \vee F_2 \)
  - \( F_1 \rightarrow F_2 \)
  - \( F_1 \leftrightarrow F_2 \)
  - \( \exists x \ F(x) \)
  - \( \forall x \ F(x) \)

- As for TRC, the only things specific to DRC are the choice of domain variables and the definition of the relational predicates
- DRC has the structure of logic, applied as a DB query/assertion language
- Restrictions for safety similar to those of TRC for quantified formulas apply to DRC
Simplification of Notation

• List the birth date and address of employees named John Smith

\{dn, a \mid \exists fn, m, ln, ssn, sex, sal, ss, d \}

\text{Employee}(FName : fn, MInit : m, LName : ln, Address : a, BDate : dn,
ESSN : ssn, Sex : sex, Sal : sal, MgrSSN : ss, DNo : d)

\land fn = 'John' \land ln = 'Smith'\}

• Many variables! Suppress variables that only appear in a relational predicate under \exists

\{dn, a \mid \exists fn, ln

\text{Employee}(FName : fn, LName : ln, Address : a, BDate : dn) \land
fn = 'John' \land ln = 'Smith'\}

• \(2^n - 1\) predicates are associated with each relation with \(n\) attributes

Further Simplification

• Suppress variables that only appear in a relation predicate and in a test for equality with a constant in a conjunction (\land)

\{dn, a \mid \text{Employee}(FName : 'John', LName : 'Smith', Address : a, BDate : dn) \}

• Corresponds to projection + selection on equality in the algebra

• The rest of DRC has the structure of logic
- $P(x) \land x = 3 \equiv P(3)$
- TRC formulation of the same example:
  \[
  \{t.\text{BDate}, t.\text{Address} \mid \text{Employee}(t) \land t.\text{FName} = \text{John} \land t.\text{LName} = \text{Smith}\}
  \]

### Selection + Projection

List the name of employees with a salary greater than 50k

\[
\{fn, ln \mid \exists sal \ (\text{Employee}(\text{FName} : fn, \text{LName} : ln, \text{Salary} : sal) \land sal > 50k)\}
\]

Could also conceivably be written

\[
\{fn, ln \mid \text{Employee}(\text{FName} : fn, \text{LName} : ln, \text{Salary} : > 50k)\}
\]
Join

- List name and address of employees who work in the Research department

\{fn, ln, a \mid \exists d \ (\text{Employee}(\text{FName} : fn, \text{LName} : ln, \text{Address} : a, \text{DNo} : d) \land \text{Department}(\text{DName} : \text{‘Research’}, \text{DNumber} : d))\}

- A join is expressed through the occurrence of the same domain variable in two (or more) relation predicates in a conjunction (\&)

- In TRC, a join is signaled by an explicit “join condition”

\{e.\text{FName}, e.\text{LName}, e.\text{Address} \mid \text{Employee}(e) \land \exists d \ (\text{Department}(d) \land d.\text{DName} = \text{‘Research’} \land d.\text{DNumber} = e.\text{DNo})\}

Double Join

- For every project located in Brussels, list the project number, the controlling department number, and the name of the department manager

\{pn, d, mfn, mln \mid \exists e \ (\text{Project}(\text{PNumber} : pn, \text{PLocation} : \text{‘Brussels’}, \text{DNum} : d) \land \text{Department}(\text{MgrSSN} : e, \text{DNumber} : d) \land \text{Employee}(\text{SSN} : e, \text{FName} : mfn, \text{LName} : mln))\}
“Complex” Query

- List project number of projects for which an employee whose last name is Smith is a worker or a manager of the department that controls the project

\[ \{ p \mid \text{Project}(PNumber : p) \land \exists e \text{ Employee}(SSN : e, LName : ‘Smith’) \land \] \[ \quad \text{WorksOn}(ESSN : e, PNo : p) \lor \] \[ \quad \exists d \text{ Department}(MgrSSN : e, DNumber : d) \land \] \[ \quad \text{Project}(PNumber : p, DNum : d) \} \]

- Many variants

Join of a Relation with itself

- List first and last name of employees, and first and last name of their immediate supervisor

\[ \{ efn, eln, mfn, mln \mid \exists m \] \[ \quad \text{Employee}(FName : efn, LName : eln, SuperSSN : m) \land \] \[ \quad \text{Employee}(SSN : m, FName : mfn, LName : mln) \} \]

- Like for the algebra and TRC, attribute names for the result have to be explicitly specified through some kind of assertion

\[ \text{RES}(\text{EmpFN}, \text{EmpLN}, \text{SupFN}, \text{SupLN}) \leftarrow \{ efn, eln, mfn, mln \mid \ldots \} \]
Universal Quantifier

- List the name of employees who work on all projects

\[
\{ fn, ln \mid \exists e \text{ Employee}(FName : fn, LName : ln, SSN : e) \land \\
\forall p \text{ Project}(PNumber : p) \rightarrow \text{WorksOn}(PNo : p, ESSN : e) \}\]

\[
\{ fn, ln \mid \exists e \text{ Employee}(FName : fn, LName : ln, SSN : e) \land \\
\forall p \text{ WorksOn}(PNo : p) \rightarrow \text{WorksOn}(PNo : p, ESSN : e) \}\]

- List the name of employees who have no dependent

\[
\{ name \mid \exists s \left( \text{Employee}(LName : name, SSN : s) \land \\
\neg \text{Dependent}(ESSN : s) \right) \}
\]

\[
\{ name \mid \exists s \left( \text{Employee}(LName : name, SSN : s) \land \\
\neg \exists m \left( \text{Dependent}(ESSN : m) \land m = s \right) \right) \}
\]

\[
\{ name \mid \exists s \left( \text{Employee}(LName : name, SSN : s) \land \\
\forall m \left( \text{Dependent}(ESSN : m) \rightarrow m \neq s \right) \right) \}
\]