Deductive Databases

Fernando Sáenz Pérez
Grupo de Programación Declarativa (GPD)
Universidad Complutense de Madrid
Facultad de Informática
Contents

1. Introduction
   DES
2. Data Models
   Relational
   Logic
3. Datalog
   Fixpoint Semantics
   Completeness of Datalog w.r.t. RA
   Stratified Negation
4. Deductive Systems
   Expressions
   Constraints
   Duplicates
   Aggregates
   Outer Joins
   Current DBMS's Limitations
5. Debugging
   Declarative Debugging
   Tracers
   Test Case Generator
6. Other Features
7. Conclusions
Introduction

- **Databases (as early as 1965):**
  - Data banks
  - Network (Codasyl model) – Mid 60s BF Goodrich IDMS
  - Hierarchic – 1969 IBM IMS-DB
  - Inverted files – 1970 Software AG ADABAS
  - Relational – 1970 Codd - 1974 IBM System R
  - Deductive – Mid 70s IBM DEDUCE 2
  - Object oriented – 1988 Objectivity/DB
  - Constraint – 1990 CQL
  - Geodatabases – 1992 ESRI Arc/Info
  - Active – 1996 ACT-Net Consortium
  - Relational object-oriented – 2000 ODMG 3.0 SUN Oracle
  - (Native) XML – 2003 MonetDB/XQuery
Introduction

- **Some concepts:**
  - Database (DB)
  - Database Management System (DBMS)
  - Data model
    - (Abstract) data structures
    - Operations
  - Query language
Introduction

- Query languages:
  - Older Navigational: IMS - Cobol
  - RA
  - TRC
  - TRD
  - SQL (SEQUEL IBM System R)
  - Datalog
  - Newer Navigational: XPath/XQuery
Introduction

- *De-facto* standard technologies in databases:
  - “Relational” model
  - SQL

- But, a current trend towards deductive databases:
  - Datalog 2.0 Conference
    - *The resurgence of Datalog in academia and industry*
  - Ontologies
  - Semantic Web
  - Social networks
  - Policy languages
Introduction. Systems

- Classic academic deductive systems:
  - LDL++ (UCLA)
  - CORAL (Univ. of Wisconsin)
  - NAIL! (Stanford University)

- Ongoing developments
  - Recent commercial deductive systems:
    - DLV (Italy, University of Calabria)
    - LogicBlox (USA)
    - Intellidimension (USA)
    - Semmle (UK)
  - Recent academic deductive systems:
    - 4QL (Warsaw University)
    - bdd (Stanford University)
    - ConceptBase (Passau, Aachen, Tilburg Universities, since 1987)
    - XSB (Stony Brook University, Universidade Nova de Lisboa, XSB, Inc., Katholieke Universiteit Leuven, and Uppsala Universitet)
    - DES (Complutense University)
Introduction.

Datalog Educational System (DES)

- Yet another system, Why?
- We needed an interactive system targeted at teaching Datalog in classrooms
- So, what a whole set of features we would ask for such a system?
  - A system oriented at teaching
  - User-friendly:
    - Installation
    - Usability
  - Multiplatform (Windows, Linux, Mac, …)
  - Interactive
  - Query languages
  - …
DES Concrete Features (1/2)

- Free, Open-source, Multiplatform, Portable
- Query languages sharing EDB/IDB:
  - Datalog following ISO Prolog standard
  - (Recursive) SQL following ANSI/ISO standard
- Database updates:
  - SQL DML
  - Commands
- Temporary Datalog views
- Duplicates
- Declarative debugging of Datalog programs
- Test case generation for SQL views
- Datalog and SQL tracers
DES Concrete Features (2/2)

- Null value support à la SQL
- Outer joins for both SQL and Datalog
- Aggregates
- Stratified Negation
- Integrity constraints:
  - Domain
  - Referential integrity
  - Functional dependencies
  - Uniqueness
- Full-fledged arithmetic
- Type system for SQL tables and views, and Datalog programs
- Source-to-source program transformations:
  - Safety
  - Performance (simplifications)
- Tabling-based implementation
SQL

- Follows ISO Standard
- **DQL:**
  - `SELECT ... FROM ... WHERE`
  - `WITH RECURSIVE ...`
- **DML:**
  - `INSERT ...`
  - `UPDATE ...`
  - `DELETE ...`
- **DDL:**
  - `CREATE [OR REPLACE] TABLE ...`
  - `CREATE [OR REPLACE] VIEW ...`
  - `DROP ...`
Datalog and SQL in DES

- Deductive engine (DE):
  - Tabling implementation
- Datalog programs are solved by DE
- Compilation of SQL views and queries to Datalog programs
- SQL queries are also solved by DE
- Interoperability is allowed: SQL and Datalog do share the deductive database!
  - Datalog queries ↔ SQL queries
  - Datalog typed relations ↔ SQL tables and views
**ODBC Connections**

- New feature since version 2.0
- Access to Relational DBMS
  - MySQL
  - MS Access
  - Oracle
  - ...
- SQL statements injected to the DBMS engine
- Query results are cached by the Datalog engine
DES as a Test-Bed for Research

- Test case generation for SQL views
- Datalog declarative (algorithmic) debugging
- Datalog and SQL tracers
- Novel proposal for outer joins in Datalog
- Theses, Papers, Academia... See DES Facts at its web page
"Impact factor"

- Up to more than 1,500 downloads a month
- More than 36,000 downloads since 2004
- Month downloads greater than XSB, GNU Prolog, YAP, …
- More than 10,000 entries in Google
- First positions in search engines when looking for "Datalog"
- Moved at the very first position in Datalog entry at Wikipedia

- It was not me! 😊
Installing DES

- **Distro under GPL in Sourceforge:**
  - Sources
  - Portable Executables (Windows, Linux, Mac OS X)
  - Portable Bundle including Java IDE (Windows)

- **Starting the system. Either:**
  - From a Prolog interpreter
    - Ciao 1.14.2
    - GNU Prolog 1.4.0
    - SICStus Prolog 4.2.0 (Complete support for all features)
    - SWI-Prolog 5.10.5 (Complete support for all features)
  - Simply execute the binary
  - Start the Java application
DES running under ACIDE

```plaintext
Datalog aggregates

1. employee (Name, Department, Salary)
2. employee (anderson, accounting, 1200).
3. employee (andrews, accounting, 1200).
4. employee (erling, accounting, 1000).
5. employee (nolan, null, null).
6. employee (norton, null, null).
7. employee (randall, resources, 800).
8. employee (sanders, sales, null).
```

DES: Datalog Educational System v.2.5

Type "/help" for help about commands
Type "des." to continue if you get out of DES
from a Prolog interpreter

Fernando Sáenz-Pérez (C) 2004-2011
GPD DISIA UCM

Please send comments, questions, etc. to:
fernan@sip.ucm.es
Web site:
http://des.sourceforge.net/
DES running under Emacs
DES running as a Windows application

DES: Datalog Educational System v.2.0

Type "/help" for help about commands
Type "des." to continue if you get out of DES from a Prolog interpreter

Fernando Sáenz-Pérez (c) 2004-2010
DISIA UCM
Please send comments, questions, etc. to:
fernan@sip.ucm.es
Web site:
http://des.sourceforge.net/

DES-Datalog>
DES running under Mac OS X Snow Leopard

DES: Datalog Educational System v.2.5

Type "/help" for help about commands
Type "des." to continue if you get out of DES from a Prolog interpreter

Fernando Sánchez-Pérez (c) 2004-2011
GPD DISIA UCM
Please send comments, questions, etc. to:
fernan@sip.ucm.es
Web site:
http://des.sourceforge.net/

DES-Datalog>
DES running in a Linux terminal

```
fernan@fernan-ubuntu:~/.Escritorio/des$ ./des

DES: Datalog Educational System v.2.0

* Type "/help" for help about commands
* Type "des." to continue if you get out of DES
  from a Prolog interpreter

Fernando Sánchez-Pérez (c) 2004-2010
DISIA UCM
Please send comments, questions, etc. to:
  fernan@sip.ucm.es
  Web site: http://des.sourceforge.net/

DES-Datalog>
```
Implementation

- **DES command-line interpreter:** Prolog
  - Tabling (Bottom-up Top-down driven)
  - Computation by strata saturations (negation and aggregates)
- **Type system:** CLP(FD)
- **Datalog Debugger:** Prolog + Java
  - \[CGS07\] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez, A new proposal for debugging datalog programs. WFLP’07
- **SQL Debugger:** CLP(FD)
- **Test Case Generator:** CLP(FD)
  - \[CGS10a\] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez, Applying Constraint Logic Programming to SQL Test Case Generation, FLOPS 2010
- **ACIDE:** Java
  - A Configurable IDE (LaTeX, SQL, Prolog, Datalog, …)
Introduction

Goals of this talk:

- Databases:
  - From relational to deductive
- (Declarative) Query Languages:
  - From SQL to Datalog
    - SQL: Many functionalities
    - Datalog: Neater formulations

How:

- Data models
- Concrete system: DES
Relational Model

- Simple and elegant model with a mathematical basis.
- Language: Relational algebra operations
  - Query optimization and execution
- Database standard ANSI/ISO SQL
- Current DBMSs support relational model
  - Warning: They are flawed extensions.
Relational Model. Concepts

- Relation schema
  - Relation name
  - (Typed) Attributes
    - Domains
- Relation instance
  - Actual values
- Relation constraints
  - Primary key
  - Foreign key
Relational Model. Languages

- **Calculus-oriented Languages**
  - Tuple Relational Calculus
  - Domain Relational Calculus

- **Algebra-oriented Language**
  - Relational Algebra (RA)
Relational Algebra

- **Basic Operations:**
  - Selection $\sigma_{\text{Condition}}(\text{Relation})$
  - Projection $\pi_{\text{Attributes}}(\text{Relation})$
  - Renaming $\rho_{\text{Schema}}(\text{Relation})$
  - Cartesian product $LR \times RR$
  - Union $LR \cup RR$
  - Difference $LR - RR$
Structured Query Language (SQL)

- Based on Relational Algebra:
  - SELECT \( \sigma_{\text{Condition}}(\text{Relation}) \)
  - Projection list \( \pi_{\text{Attributes}}(\text{Relation}) \)
  - AS, WITH \( \rho_{\text{Schema}}(\text{Relation}) \)
  - FROM \( LR \times RR \)
  - UNION \( LR \cup RR \)
  - EXCEPT \( LR - RR \)
What relational languages can do

- Row filtering
- Tuple filtering
- Build new relations by combinations
- Expression evaluation (Generalized projection)
What relational languages *cannot* do

- Are we running low on any parts needed to build a ZX600 sports car?
- What is the total component and assembly cost to build a ZX600 at today's part prices?

- What do we simply need is *recursion*
Example

Find components of a bike

Assembly instance

<table>
<thead>
<tr>
<th>part</th>
<th>subpart</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>bike</td>
<td>wheel</td>
<td>2</td>
</tr>
<tr>
<td>bike</td>
<td>frame</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>seat</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>pedal</td>
<td>1</td>
</tr>
<tr>
<td>wheel</td>
<td>spoke</td>
<td>16</td>
</tr>
<tr>
<td>wheel</td>
<td>tire</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>tube</td>
<td>1</td>
</tr>
</tbody>
</table>
What relational languages cannot do

- Intuitively, we must join Assembly with itself to deduce that bike contains spoke and tire.
  - Takes us one level down Assembly hierarchy.
  - To find components that are one level deeper (e.g., rim), need another join.
  - To find all components, need as many joins as there are levels in the given instance!

- For any relational algebra expression, we can create an Assembly instance for which some answers are not computed by including more levels than the number of joins in the expression!
Logic as a Data Model

- Relational databases distinguish between DDL and DML
  - DDL is for creating schemas or views
  - DML is for maintaining data and queries

- In deductive databases both data and queries are specified by formulas
A deductive database consists of facts and rules
- The set of facts is called extensional database (EDB)
  - If no functions are used in the facts, it can be represented as a simple relational database table
- The set of rules is called intensional database (IDB)
  - Analogous to views in relational databases
A logical language $\mathcal{L}$ is given by the signature $\mathcal{L} = (\Gamma, \Omega, \Pi, X)$:
- $\Gamma$: All constant symbols
- $\Omega$: All functional symbols
- $\Pi$: All predicate symbols
- $X$: All variable symbols

An interpretation captures the semantics of $\mathcal{L}$:
- An interpretation assigns each term to an element of some universe of discourse
- An interpretation assigns a truth value to each formula, i.e., decides which statements are true and which are false
FOL Recall

- Model-Theoretic Interpretation of rules
  - Rules define possible worlds or models
  - A model of a set of rules is an interpretation that makes true all rules
  - There are many models for a set of rules
  - Without negation, a unique minimal model can be found
FOL Recall

Example:

(1) p(X) ← q(X)
(2) q(X) ← r(X)
M1:

- r(1), q(1), p(1), q(2), p(2), p(3) are true
- false, otherwise

Substitution ρ={X/1}:

- (1) p(1) ← q(1) : true ← true : true
- (2) q(1) ← r(1) : true ← true : true

Substitution ρ={X/2}:

- (1) p(2) ← q(2) : true ← true : true
- (2) q(2) ← r(2) : true ← false : true

Substitution ρ={X/3}:

- (1) p(3) ← q(3) : true ← false : true
- (2) q(3) ← r(3) : true ← false : true

Substitution ρ={X/i} for all other i:

- (1) p(i) ← q(i) : false ← false : true
- (2) q(i) ← r(i) : false ← false : true
FOL Recall

Example (ctd.):

- (1) $p(X) \leftarrow q(X)$
- (2) $q(X) \leftarrow r(X)$
- (3) $r(1)$

M2:

- $r(1)$,
  - $q(1), p(1)$ are true
  - false, otherwise

M2 is a minimal model
No true fact can be extracted from M2 and resulting a model consistent with \{r(1)\}
FOL and Databases

What a typical database user would expect from a query language?

- Finite data, finite computations (terminating queries)
  - No terms or bound depth
  - Still, be aware of built-in infinite relations!
- All answer tuples at once
  - Prolog returns several answers upon backtracking
FOL and Databases

- **Restrict FOL allowing only Horn clauses and non-recursive typed functions**
  - Ground instances are thus Horn clauses
  - Check for unsatisfiability of finite subsets of Herbrand expansion is in P
  - Herbrand expansion is finite as the Herbrand universe is finite
So, every predicate is written by **Horn clauses**:

- $\forall (L_1 \lor L_2 \lor \ldots \lor L_n), L_i \in L_L$
- With atomic formulae $L_i$ and at most one positive literal $L_j$

Logic programming introduced a slightly different notation of Horn clauses for simplicity

- $L_j \leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_n$

Where:

- $\leftarrow$ is understood as implication
- , as conjunction
- . denotes the end of a clause
Datalog

- Datalog is a query and rule language specifically defined for deductive databases
  - Syntactically similar to Prolog
  - Introduced around 1978 for academic database research by Hervé Gallaire and Jack Minker
  - Used as the main foundation for expert systems theory during the 1980ies
A database query language stemming from Prolog

<table>
<thead>
<tr>
<th>Prolog</th>
<th>Datalog</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td>Relation</td>
<td>Relation</td>
</tr>
<tr>
<td>Goal</td>
<td>Query</td>
<td>Expression</td>
</tr>
</tbody>
</table>

Prolog and Datalog are based of First Order Logic (FOL)
- AKA Predicate Logic
Datalog

- **Datalog differs from Prolog:**
  - Datalog does not allow function symbols in arguments
  - Facts are ground
  - Meaning of a Datalog relation follows the model-theoretic point-of-view
    - Intensionally (Rules or Clauses)
    - Extensionally (Facts)
  - **Datalog is truly declarative:**
    - Clause order is irrelevant
    - Order of literals in a body is irrelevant
    - No extra-logical constructors as the feared cut
Datalog Syntax

- **Program**: Set of rules.
- **Rule**:
  - head :- body.
  - ground_head.
- **Head**: Positive atom.
- **Body**: Conjunctions (,) and disjunctions (;) of literals
- **Literal**: Atom, Built-in (> , < , …).
- **Query**:
  - Literal with variables or constants in arguments
  - Body (Conjunctive queries, …)
Datalog Example

- **Facts:**
  father(tom, amy).
  father(jack, fred).
  father(tony, carolIII).
  father(fred, carolIII).

  mother(graceI, amy).
  mother(amy, fred).
  mother(carolI, carolIII).
  mother(carolII, carolIII).

- **Rules:**

  parent(X, Y) :- father(X, Y).
  parent(X, Y) :- mother(X, Y).

- **Query:**

  parent(X, Y)

- **Minimal model for parent:**

  \{
  (tom, amy), (grace, amy), (jack, fred),
  (amy, fred), ...
  \}
Recursion

father(tom, amy).
father(jack, fred).
father(tony, carolII).
father(fred, carolIII).

mother(graceI, amy).
mother(amy, fred).
mother(carolI, carolII).
mother(carolII, carolIII).

parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

ancestor(X, Y) :- 
    parent(X, Y).
ancestor(X, Y) :- 
    parent(X, Z),
    ancestor(Z, Y).

ancestor(tom, X)
{
    ancestor(tom, amy),
    ancestor(tom, carolIII),
    ancestor(tom, fred)
}
Fixpoint Semantics of Recursive Datalog

- Let $f$ be a function that takes values from domain $D$ and returns values from $D$.
- A value $v$ in $D$ is a fixpoint of $f$ if $f(v) = v$.
- Consider the fn $double+$, which is applied to a set of integers and returns a set of integers (i.e., $D$ is the set of all sets of integers).
  - E.g., $double+({1,2,5}) = \{2,4,10\}$ Union $\{1,2,5\}$
  - The set of all integers is a fixpoint of $double+$.
  - The set of all even integers is another fixpoint of $double+$; it is smaller than the first fixpoint.
The least fixpoint \( \text{lfp} \) of a function \( f \) is a fixpoint \( v \) of \( f \) such that every other fixpoint of \( f \) is smaller than or equal to \( v \).

In general, there may be no least fixpoint (we could have two minimal fixpoints, neither of which is smaller than the other).

If we think of a Datalog program as a function that is applied to a set of tuples and returns another set of tuples, this function (fortunately!) always has a least fixpoint.
Unrestricted Variables

- **Problem** of variables in heads of rules
  - Consider a rule \( p(X) :- r(Y) \).
  - What does this mean?
    - If there is a substitution for \( Y \) making \( r(Y) \) true, then \( p(X) \) is true for all possible substitutions for \( X \)?
    - If \( r(Y) \) is true for all possible substitutions of \( Y \), then \( p(X) \) is true??
    - Or only for \( Y=X??\)

- **Problem** with (built-in) comparison operators:
  - \( \text{less}(X,Y) :- X < Y \)
  - There are infinite pairs that commit to \( X < Y \)
A rule is safe if all its variables are limited (range restricted)

A *limited* variable in a rule is defined as:
- Any variable in a user-defined predicate in a subgoal is limited
- Any variable $X$ that occurs in a subgoal $X=a$ or $a=X$ is limited
- A variable $X$ is limited if it occurs in a subgoal $X=Y$ or $Y=X$, where $Y$ is already known to be limited

**Bottomline:**
- Any head variable and variables in built-ins become ground
Completeness of Datalog w.r.t. RA

- Recursion is appealing but,
  Is Datalog as expressive as RA and SQL?
- Let’s see…
Selection

s(sno INTEGER, name STRING)

- RA:
  \( \sigma_{sno=1}(s) \)

- SQL:
  SELECT name FROM s WHERE sno=1;

- Datalog:
  \( s(1, Y) \).
Projection

\[ s(\text{sno INTEGER, name STRING}) \]

- RA:
  \[ \pi_{\text{name}}(s) \]

- SQL:
  SELECT name FROM s;

- Datalog:
  \[ s(_X, Y) . \]
Renaming

\[ s(sno \text{ INTEGER}, \text{ name STRING}) \]

- **RA:**
  \[ \rho_{projection}(\pi_{\text{name}}(s)) \]

- **SQL:**
  
  ```sql
  WITH projection(name) AS (SELECT name FROM s)
  SELECT * FROM projection;
  ```

- **Datalog:**
  
  ```datalog
  projection(Y) :- s(X, Y).
  ```
Cartesian Product

\[ s(\text{sn} \ \text{INT}, \ \text{name} \ \text{STRING}) \]
\[ sp(\text{sn} \ \text{INT}, \ \text{pno} \ \text{INT}) \]

- **RA:**
  \[ s \times sp \]

- **SQL:**
  \[ SELECT * FROM s, sp; \]

- **Datalog:**
  \[ s(X, Y), \ sp(U, V). \]
Set Union

\[ s(sno \text{ INT}, \text{name STRING}) \]
\[ q(sno \text{ INT}, \text{name STRING}) \]

- RA:
  \[ s \cup q \]

- SQL:
  \[
  \text{SELECT} \ast \text{ FROM } s \\
  \text{UNION} \\
  \text{SELECT} \ast \text{ FROM } q; 
  \]

- Datalog:
  \[
  s(X,Y) ; q(U,V). 
  \]
Set Difference

\[ s(sno \text{ INT}, \text{name STRING}) \]
\[ q(sno \text{ INT}, \text{name STRING}) \]

- **RA:**
  \[ s - q \]

- **SQL:**
  
  ```sql
  SELECT * FROM s
  EXCEPT
  SELECT * FROM q;
  ```

- **Datalog:**
  
  \[ s(X,Y), \text{not}(q(X,Y)) . \]
Expressive Power

- So, for Datalog to be complete w.r.t. RA, a form of (dreaded) negation must be faced.
- There are several Datalog proposals.
Datalog Language Classes

- Depending on the use of functions and negation several Datalog language classes can be distinguished:
  - $\text{Datalog}^{\text{neg}}$ programs do not contain function symbols
  - $\text{Datalog}^{f}$ programs (or definite programs) do not contain negative literals
  - $\text{Datalog}$ programs contain neither negative literals nor function symbols
Equivalent DB Languages

- Relational Algebra
- Relational Calculus
- Non-recursive Datalog with Negation
The negation of a relation stands for its \textit{complement}, but this is rather not a well-defined term.

Finding a complement must be w.r.t. to a domain.

RA does the trick with the difference operator (both instance relations are ground).

Even knowing the actual domain, the complement may be infinite.

- $X$: integer, not(p(X))
Negation and Safety

- Also, unlimited variables are a problem
  - bachelor(X) :- male(X), not(married(X,Y))
  - Joining male(X) with the complement of married yields all tuples (X,Y) such that X is a male who is not married to absolutely everybody in the universe

DES> /assert bachelor(X) :- male(X), not(married(X,Y))
Error: not(married(X,Y)) might not be correctly computed because of the unrestricted variable(s): [Y]

- So, variables in negation must be limited
Negation in Rule Bodies

- Limiting variables is a need but does not completely solve the problem with negation
- If negation is allowed, there might not be a single fixed point, but several ones
  - If so, what does a logic program mean?
- Example:
  \[
  p(X) : - r(X), \not q(X).
  \]
  \[
  q(X) : - r(X), \not p(X).
  \]
  \[
  r(1).
  \]
  - Possible solutions (both are minimal fixed points):
    - S1: P=\emptyset, Q={1}
    - S2: P={1}, Q= \emptyset
- We need a way to choose the intended fixpoint for safe programs.
Stratified Negation

- Stratified negation is a property that some programs enjoy, which help to deal with the problem of many minimal fixpoints.

- Predicate dependency:
  - T depends on S if some rule with T in the head contains S or (recursively) some predicate that depends on S, in the body.
  - A predicate dependency graph (PDG) is this way defined

- Stratified program: If T depends on not S, then S cannot depend on T (or not T).

- If a program is stratified, predicates in the program can be partitioned into strata:
  - Stratum 0: All EDB predicates.
  - Stratum i: Predicates defined in terms of tables in Stratum i and lower strata.
  - If T depends on not S, S is in a lower stratum than T.
Stratified Negation. Safety

- Stratified negation still requires safety.
- Negated goals are not allowed to be range restrictors, they rather need limited variables.
The semantics of a stratified program is given by one of the minimal fixpoints, which can be identified by the following operational definition:

- First, compute the least fixpoint of all predicates in Stratum 1. (Stratum 0 predicates are fixed.)
- Then, compute the least fixpoint of predicates in Stratum 2; then the \( \text{lfp} \) of predicates in Stratum 3, and so on, stratum-by-stratum.
Typically, a database user does not need all the intensional data. Rather, he queries views of interest.

So, DES implements tabling (cf. M. Carro talk).

Top-down driven, bottom-up computation.

Restrict PDG to the query, find a stratification and compute only the involved predicates.

It might be the case of loading a non-stratifiable program but finding a stratification for a given query.

Tabling, as already introduced, provides:

- Termination
- Performance
Non-stratifiable Example: Russell Paradox

- The barber in a town shaves every person who does not shave himself

```
DES-Datalog> /c russell
Info: Consulting russell...
  shaves(barber,M) :- man(M), not(shaves(M,M)).
  man(barber).
  man(mayor).
  shaved(M) :- shaves(barber,M).
  end_of_file.
Info: 4 rules consulted.
Warning: Non stratifiable program.

DES-Datalog> shaves(X,Y)
Warning: Unable to ensure correctness for this query.
{  shaves(barber,mayor)
} Info: 1 tuple computed.
Undefined:
{  shaves(barber,barber)
} Info: 1 tuple undefined.
```

- But, warning: DES does not offer complete support for a 3VL
Is That All about Negation, Folks?

- Negation is a huge source of problems and fixes
- Further proposals:
  - Stable Model Semantics (Gelfond)
    - Based on Moore’s autoepistemic logic
    - 2VL and 3VL
    - DLV, ASP (Answer Set Programming)
  - Well-founded Semantics (WFS)
    - It relies on unfounded sets to deduce negative information
    - XSB
  - 4-Valued Logic (4VL)
    - ‘true’, ‘false’, ‘inconsistent’ and ‘unknown’
    - 4QL
Practical Database Systems

- What we do need are systems that solve everyday problems, not toy examples.
- First, expressive power does not only depend on recursion, but on other needs, as:
  - Evaluation of expressions (arithmetic, strings, …)
  - (Strong) Constraints
  - Duplicates
  - Aggregates
  - NULL-related operators
  - …
- Second, efficient systems
  - Inferring method
  - Indexing
  - Scalability
  - …
Evaluation of Expressions

- Arithmetic expressions are usually evaluated by a couple of methods in DDB:
  - Built-in predicate: X is $Expression$
    \[
    \text{DES-Datalog} > X \text{ is } \text{sqrt}(2)
    \]
    \[
    \{ \quad 1.4142135623730951 \text{ is } \text{sqrt}(2) \}
    \]
    Info: 1 tuple computed.
  - Equality: $LeftExpr = RightExpr$ (DLV provides limited support)
    \[
    \text{DES-Datalog} > 2+1 = Y
    \]
    \[
    \{ 2 + 1 = 3 \}
    \]
    Info: 1 tuple computed.

- As expected, they are source or unsafety…
Arithmetic Unsafety

\[ p(0). \]
\[ p(X) : - \\
    p(Y), \\
    X \text{ is } Y + 1. \]

- Unlimited tuples \((N, N+1)\) become computed (although not stored)
- Unbounded arithmetic
- Try the query, press Ctrl-C, and type: des.
  /list_et.

DES-Datalog> p(X)
Prolog interruption (h for help)? a
| ?- des.
DES-Datalog> /list_et

Answers:
{ 
  p(0),
  p(1),
  p(2),
  p(3),
  p(4),
  p(5),
  p(6),
  p(7),
  p(8),
  p(9),
  ...}
Integrity Constraints

- Integrity constraints (IC):
  - Strong constraints as known in databases
  - Do not mix up with constraints as in CLP(D)!

- Usual IC:
  - Type (domain)
  - Primary key
  - Foreign key
  - Existency constraints
  - Check constraints

- Not so-usual IC:
  - Candidate key
  - Functional dependencies
  - User-defined integrity constraints

- Expressed in DES as either Datalog assertions or SQL statements
Types

- Types stemmed from SQL

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>String of unbounded length</td>
</tr>
<tr>
<td>string</td>
<td></td>
</tr>
<tr>
<td>char(N)</td>
<td>String with length up to N</td>
</tr>
<tr>
<td>varchar(N)</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>String with length 1</td>
</tr>
<tr>
<td>integer</td>
<td>Integer number</td>
</tr>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>Real number</td>
</tr>
<tr>
<td>real</td>
<td></td>
</tr>
</tbody>
</table>

- Alternative Datalog syntax:

```
DES-Datalog> :- type(p, [int, string])
DES-Datalog> :- type(p(int, string))
```
Types. Intensional Database

- Predicates defined by rules (IDB)

DES-Datalog> /listing
s(a).
t(1).
t(X) :-
    s(X).
Info: 3 rules listed.

DES-Datalog> :-type(t,[int])
Error: No type tuple covers all the loaded rules for t/1:
    t(1).
    t(X) :-
        s(X).
Info: 2 rules listed.
Types. Propositional Relations

- Predicates defined by rules (IDB)

```
DES-Datalog> :-type(a,[])
DES-Datalog> /dbschema a
Info: Table:
* a
```
Types vs. Domains

- Type assertions and SQL table creation are alternative syntax for imposing type constraints

- SQL:
  
  CREATE TABLE s(sno INT, name VARCHAR(10));

- Datalog:
  
  :-type(s(sno:int, name:varchar(10))).

DES-Datalog> /dbschema
Info: Table(s):
 * s(sno:number(integer), name:string(varchar(10)))
Info: No views.
Info: No integrity constraints.
Existency Constraint

- An existency constraint requires concrete values for instances.
- The NULL value as in relational databases in supported in DES (more on this later).

- **SQL:**
  ```sql
  CREATE TABLE s(sno INT, name VARCHAR(10) NOT NULL);
  ```

- **Datalog:**
  ```prolog
  :-type(s(sno:int, name:varchar(10))).
  :-nn(s,[name]).
  ```

```
DES-Datalog> /assert s(1,null)
Error: Not null violation s.[name]
```
Primary Key Constraint

- A primary key (PK) constraint requires unique combinations of concrete values for instances

- SQL:
  
  ```
  CREATE TABLE s(sno INT PRIMARY KEY, name VARCHAR(10));
  ```

- Datalog:
  
  ```
  :-type(s(sno:int, name:varchar(10))).
  :-pk(s,[sno]).
  ```

  DES-Datalog> /assert s(1,'wheel')
  DES-Datalog> /assert s(1,'spoke')
  Error: Primary key violation s.[sno]
  when trying to insert: s(1,spoke)
Candidate Key Constraint

- **AKA Uniqueness constraint**

- **Similar to PK**

- No direct support in standard SQL. You must use non-duplicated indexes (not yet supported by DES):
  
  ```sql
  CREATE INDEX s_index FOR s ON sno NO DUPLICATES;
  ```

- But, DES does provide (extension to standard):
  
  ```sql
  CREATE TABLE s(sno INT, name VARCHAR(10) CANDIDATE KEY);
  ```

- **Datalog:**
  
  ```prolog
  :-type(s(sno:int, name:varchar(10))).
  :-ck(s,[name]).
  ```
Foreign Key Constraint

- Values in a given set of columns of a relation must exist already in the columns declared in the primary key constraint of another relation.

- **SQL:**
  ```sql
  CREATE TABLE n(name VARCHAR(10) PRIMARY KEY);
  CREATE TABLE s(sno INT, name VARCHAR(10) FOREIGN KEY n);
  ```

- **Datalog:**
  ```datalog
  :-type(n(name:varchar(10))).
  :-pk(n,[name]).
  :-type(s(sno:int, name:varchar(10))).
  :-fk(s,[name],n,[name]).
  ```
Functional Dependency Constraint

- Given a set of attributes $A_1$ of a relation $R$, they functionally determine another set $A_2$, i.e., each tuple of values of $A_1$ in $R$ is associated with precisely one tuple of values $A_2$ in the same tuple of $R$.

- No provision in SQL Standard

- **Datalog:**
  
  DES-Datalog> :-type(p, [a:int, b:int])  
  DES-Datalog> :-fd(p, [a], [b])  
  DES-Datalog> /dbschema p  
  Info: Table:  
  * p(a:number(integer), b:number(integer))  
  - FD: [a] -> [b]
User-defined Integrity Constraints

- **SQL:**
  - **CHECK constraints** (not supported by DES, yet)
  - **Triggers**

  ```sql
  CREATE TABLE t(c INT CHECK (c BETWEEN 0 AND 10));
  ```

- **Datalog:**

  ```prolog
  DES-Datalog> :-type(t,[c:int])
  DES-Datalog> :-t(X),(X<0;X>10)
  DES-Datalog> /assert t(11)
  Error: Integrity constraint violation.
  ic(X) :- t(X),(X < 0 ; X > 10).
  Offending values in database: [ic(11)]
  ```
User-defined Integrity Constraints

DES-Datalog> /consult paths
Info: Consulting paths...
  edge(a,b).
  edge(a,c).
  edge(b,a).
  edge(b,d).
  path(X,Y) :- path(X,Z), edge(Z,Y).
  path(X,Y) :- edge(X,Y).
end_of_file.
Info: 6 rules consulted.
DES-Datalog> :-path(X,X)
Error: Integrity constraint violation.
  ic(X) :-
    path(X,X).
  Offending values in database: [ic(b), ic(a)]
Info: Constraint has not been asserted.
Duplicates

- SQL is not set-oriented, rather it allows duplicates in base relations and query outcomes.
- It does more than RA and Datalog.
- So, for supporting SQL we need:
  - Multisets or Bags
  - Duplicate elimination
- But, duplicates suffer issues, and however all practical RDBMSs do support duplicates.
  - "If something is true, saying it twice doesn't make it any more true." [E.F. Codd]
Duplicates. Some Issues

- Identifier and name of suppliers:
  \[ s(\text{sno INT, name STRING}) \]

- Which supplier supplies a given part:
  \[ sp(\text{sno, pno}) \]

```
SELECT sno
FROM s
WHERE sno IN (SELECT sno FROM sp)
```

```
SELECT sno
FROM s NATURAL INNER JOIN sp
```

- Semantically equivalent queries return a different number of tuples
- Also, they preclude some optimizations and make this process cumbersome
Duplicates as of DES

- Duplicates are disabled by default

```datalog
DES-Datalog> /duplicates on
DES-Datalog> /assert t(1)
DES-Datalog> /assert t(1)
DES-Datalog> t(X)
{
    t(1),
    t(1)
}
Info: 2 tuples computed.
```

- Rules can also be source of duplicates, as in:

```datalog
DES-Datalog> /assert s(X):-t(X)
DES-Datalog> s(X)
{
    s(1),
    s(1)
}
Info: 2 tuples computed.
```
Duplicates as of DES

- Duplicates *even* in recursive rules (LDL does not allow this)

```
DES-Datalog> /assert t(X):-t(X)
DES-Datalog> t(X)
{
    t(1),
    t(1),
    t(1),
    t(1),
    t(1)
}
Info: 4 tuples computed.
```

- No SQL implementation support this
Duplicates as of DES

- Discarding duplicates with metapredicates:
  - `distinct/1`
  - `distinct/2`

DES-Datalog> `distinct(t(X))`
Info: Processing:
  `answer(X) :-
  distinct(t(X)).`
{
  `answer(1)`
}
Info: 1 tuple computed.

- SQL

DES-Datalog> `select distinct * from t`
`answer(t.a) ->`
{
  `answer(1)`
}
Info: 1 tuple computed.
Safety and Duplicates

- Set variables in duplicate metapredicates are not bound

\[
distinct([X], t(X,Y))
\]

- Unsafe goal:

\[
distinct([X], t(X,Y)), s(Y)
\]

DES-Datalog> distinct([X], t(X,Y)), s(Y)
Error: Incorrect use of shared set variables in metapredicate:
    [Y]
DES-Datalog> /safe on
DES-Datalog> distinct([X], t(X,Y)), s(Y)
Info: Processing:
    answer(X,Y,C) :- s(Y), distinct([X], t(X,Y)).
Aggregates

- **Aggregate functions:**
  - `count` – `COUNT (*)`
  - `count(V)` – `COUNT (C)`
  - `min(V)` – `MIN (C)`
  - `max(V)` – `MAX (C)`
  - `sum(V)` – `SUM (C)`
  - `avg(Var)` – `AVG (C)`
  - `times(Var)` – *No SQL counterpart*
Aggregates

- **Metapredicate** `group_by/3`

```prolog
group_by(
    Relation_A, % FROM / WHERE
    [Var_1,...,Var_n], % Grouping columns
    Relation_B) % HAVING / Projection
```
Aggregates

Example

- Number of employees for each department:

DES-Datalog> group_by(employee(N,D,S), [D], R=count)

Info: Processing:
answer(D,R) :-
group_by(employee(N,D,S),[D],R = count).
{
  answer(accounting,3),
  answer(null,2),
  answer(resources,1),
  answer(sales,5)
}
Info: 4 tuples computed.

11 counts
Aggregates
Example (contd.)

- Active employees (those with assigned salaries):

  DES-Datalog> group_by(employee(N,D,S),
      [D],
      C=count(S))

  Info: Processing:
  answer(D,C) :-
      group_by(employee(N,D,S),[D],C = count(S)).

  \{  
      answer(accounting,3),
      answer(null,0),
      answer(resources,1),
      answer(sales,3)
  \}

  Info: 4 tuples computed.

  7 counts !

<table>
<thead>
<tr>
<th>Name</th>
<th>Department</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>anderson</td>
<td>accounting</td>
<td>1200</td>
</tr>
<tr>
<td>andrews</td>
<td>accounting</td>
<td>1200</td>
</tr>
<tr>
<td>arlingon</td>
<td>accounting</td>
<td>1000</td>
</tr>
<tr>
<td>nolan</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>norton</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>randall</td>
<td>resources</td>
<td>800</td>
</tr>
<tr>
<td>sanders</td>
<td>sales</td>
<td>null</td>
</tr>
<tr>
<td>silver</td>
<td>sales</td>
<td>1000</td>
</tr>
<tr>
<td>smith</td>
<td>sales</td>
<td>1000</td>
</tr>
<tr>
<td>Steel</td>
<td>sales</td>
<td>1020</td>
</tr>
<tr>
<td>Sullivan</td>
<td>sales</td>
<td>null</td>
</tr>
</tbody>
</table>

Still, 11 tuples
Aggregates
Example (contd.)

- Active employees of departments with more than one active employee:

DES-Datalog> group_by(employee(N,D,S),
                        [D],
                        count(S)>1)

Info: Processing:
answer(D) :-
group_by(employee(N,D,S),
        [D],
        (A = count(S),A > 1)).

{ answer(accounting),
  answer(sales)
}
Info: 2 tuples computed.

<table>
<thead>
<tr>
<th>Name</th>
<th>Department</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>anderson</td>
<td>accounting</td>
<td>1200</td>
</tr>
<tr>
<td>andrews</td>
<td>accounting</td>
<td>1200</td>
</tr>
<tr>
<td>arlingon</td>
<td>accounting</td>
<td>1000</td>
</tr>
<tr>
<td>nolan</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>norton</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>randall</td>
<td>resources</td>
<td>800</td>
</tr>
<tr>
<td>sanders</td>
<td>sales</td>
<td>null</td>
</tr>
<tr>
<td>silver</td>
<td>sales</td>
<td>1000</td>
</tr>
<tr>
<td>smith</td>
<td>sales</td>
<td>1000</td>
</tr>
<tr>
<td>Steel</td>
<td>sales</td>
<td>1020</td>
</tr>
<tr>
<td>Sullivan</td>
<td>sales</td>
<td>null</td>
</tr>
</tbody>
</table>
Aggregates

Aggregate metapredicates:

- count(Rel)
- count(Rel, Var)
- min(Rel, Var)
- max(Rel, Var)
- sum(Rel, Var)
- avg(Rel, Var)
- times(Rel, Var)
Aggregates Predicates and Group By

- Number of employees for each department.
  - Recall predicate group_by and functions:

    DES-Datalog> group_by(employee(N,D,S), [D], C=count)

- With aggregate predicates:

    DES-Datalog> c(D,C) :- count(employee(N,D,S),S,C)
    Info: Processing:
    c(D,C) :-
      count(employee(N,D,S),S,[D],C).
    {
      c(accounting,3),
      c(null,0),
      c(resources,1),
      c(sales,3)
    }
    Info: 4 tuples computed.
Aggregates and Recursion

% SQL Program

CREATE OR REPLACE VIEW
shortest_paths(Origin,Destination,Length) AS
WITH RECURSIVE
  path(Origin,Destination,Length) AS
  (SELECT edge.*,1 FROM edge)
UNION
  (SELECT
    path.Origin,edge.Destination,path.Length+1
  FROM path,edge
  WHERE path.Destination=edge.Origin and
    path.Length <
  (SELECT COUNT(*) FROM Edge) )
SELECT Origin,Destination,MIN(Length)
FROM path
GROUP BY Origin,Destination;

% SQL Query

SELECT * FROM shortest_paths;

% Datalog Program

path(X,Y,1) :-
  edge(X,Y).
path(X,Y,L) :-
  path(X,Z,L0),
  edge(Z,Y),
  count(edge(A,B),Max),
  L0<Max,
  L is L0+1.

% Datalog Query:

shortest_paths(X,Y,L) :-
  min(path(X,Y,Z),Z,L).
Set variables in aggregate metapredicates are not bound

group_by(t(X, Y), [X], C=count)

Unsafe goal:

group_by(t(X, Y), [X], C=count), s(Y)

DES-Datalog> group_by(t(X, Y), [X], C=count), s(Y)
Error: Incorrect use of shared set variables in metapredicate:
   [Y]
DES-Datalog> /safe on
DES-Datalog> group_by(t(X, Y), [X], C=count), s(Y)
Info: Processing:
   answer(X, Y, C) :- s(Y), group_by(t(X, Y), [X], C = count).
Aggregates and DISTINCT

- For discarding duplicates (functions and metapredicates):
  - `sum_distinct`
  - `count_distinct` - SELECT DISTINCT COUNT(*)
  - `count_distinct(V)` - SELECT COUNT(DISTINCT C)
  - `avg_distinct`
  - `times_distinct`
  - No need for `min_distinct` and `max_distinct`
What Outer Joins are All About

CREATE TABLE students(
    name string, subject string, mark int);
:-type(students(name:string, subject:string, mark:int)).

SELECT name
FROM students
WHERE subject='databases'
    AND mark >= 5

students(Name,databases,_Mark),_Mark>=5

<table>
<thead>
<tr>
<th>name</th>
<th>subject</th>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>anderson</td>
<td>programming</td>
<td>6</td>
</tr>
<tr>
<td>andrews</td>
<td>databases</td>
<td>5</td>
</tr>
<tr>
<td>arlingon</td>
<td>databases</td>
<td>3</td>
</tr>
<tr>
<td>arlingon</td>
<td>programming</td>
<td>7</td>
</tr>
<tr>
<td>norton</td>
<td>databases</td>
<td>6</td>
</tr>
<tr>
<td>smith</td>
<td>databases</td>
<td>NULL</td>
</tr>
</tbody>
</table>

answer
name

andrews
norton
CREATE TABLE conversion(mark INT, grade STRING)

:-type(conversion(mark:int, grade:string))

SELECT name, mark, grade
FROM students AS s, conversion AS c
WHERE
    s.subject='databases'
    AND s.mark=c.mark

students(Name,databases,Mark),
conversion(Mark,Grade)
Now, we recall a database student named Smith

Where's Smith in the answer?

SELECT name, mark, grade
FROM students AS s
LEFT OUTER JOIN conversion AS c
ON s.mark = c.mark
WHERE s.subject = 'databases'

<table>
<thead>
<tr>
<th>name</th>
<th>subject</th>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>anderson</td>
<td>programming</td>
<td>6</td>
</tr>
<tr>
<td>andrews</td>
<td>databases</td>
<td>5</td>
</tr>
<tr>
<td>arlingon</td>
<td>databases</td>
<td>3</td>
</tr>
<tr>
<td>arlingon</td>
<td>programming</td>
<td>7</td>
</tr>
<tr>
<td>norton</td>
<td>databases</td>
<td>6</td>
</tr>
<tr>
<td>smith</td>
<td>databases</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>mark</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>andrews</td>
<td>5</td>
<td>C</td>
</tr>
<tr>
<td>arlingon</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>norton</td>
<td>6</td>
<td>C+</td>
</tr>
<tr>
<td>smith</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
But, beware ...

SELECT name
FROM students
WHERE subject='databases' and mark >= 5

SELECT name
FROM students
WHERE subject='databases' and mark < 5
And even worser ...

```
SELECT *  
FROM students  
WHERE mark IN  
  (SELECT mark  
   FROM students)  ;
```

- Where's damned Smith in the answer?
- Hasn't Smith the same mark as itself eventually, even if it is unknown yet?
### Null Semantics: SQL

<table>
<thead>
<tr>
<th>Comparison Operator</th>
<th>Cte.₁</th>
<th>NULL</th>
<th>Left Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cte.₂</td>
<td>True/False</td>
<td>False</td>
<td>?</td>
</tr>
<tr>
<td>NULL</td>
<td>False</td>
<td>False</td>
<td>?</td>
</tr>
<tr>
<td>Right Argument</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Incomplete 3VL
- $a = a$ is False for NULL
- False $\rightarrow$ Unknown
## Null Semantics: DES

<table>
<thead>
<tr>
<th>Comparison Operator</th>
<th>Cte.₁</th>
<th>NULL₁</th>
<th>NULL₂</th>
<th>Left Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cte.₂</td>
<td>True/False</td>
<td>False</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>NULL₁</td>
<td>False</td>
<td>True/False</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>NULL₂</td>
<td>False</td>
<td>False</td>
<td>True/False</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>Right Argument</td>
<td>NULL's are distinguishable:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'$NULL' (Id)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>But still, we are in a 2VL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Duplicates and NULL's

- If duplicates are disabled, each NULL value is considered as the same constant for all of its occurrences in a program (as SQL)

```
DES-Datalog> /assert p(null)
DES-Datalog> /assert p(null)
DES-Datalog> p(X)
{
  p(null)
}
Info: 1 tuple computed.
```

- If duplicates are enabled, each NULL value is considered as a different constant for each occurrence in a program (as SQL)

```
DES-Datalog> /duplicates on
DES-Datalog> p(X)
{
  p(null),
  p(null)
}
Info: 2 tuples computed.
```
So:

- NULL's, although subject of semantic flaws, are widely used in current database applications
- However, they are not usual in deductive databases (perhaps to above?)
  - SPARQL (recursion is not supported)
  - XSB (no native support)
  - 4QL (released in 2011 as a prototype)
Outer Joins as provided by DES

Null values:
- Cte.: \text{null}
- Functions: \text{is\_null(Var)} \quad \text{is\_not\_null(Var)}

Outer join built-ins:
- Left ( ): \text{lj(Left\_Rel, Right\_Rel, ON\_Condition)}
- Right ( ): \text{rj(Left\_Rel, Right\_Rel, ON\_Condition)}
- Full ( ): \text{fj(Left\_Rel, Right\_Rel, ON\_Condition)}
Outer Join Examples

- **SQL:**
  
  ```sql
  SELECT * FROM a LEFT JOIN b ON x=y;
  ```

- **Datalog:**
  
  ```datalog
  lj(a(X), b(Y), X=Y)
  ```

- **SQL:**
  
  ```sql
  SELECT * FROM a LEFT JOIN b WHERE x=y;
  ```

- **Datalog:**
  
  ```datalog
  lj(a(X), b(X), true)
  ```

- **SQL:**
  
  ```sql
  SELECT * FROM a LEFT JOIN (b RIGHT JOIN c ON y=u) ON x=y;
  ```

- **Datalog:**
  
  ```datalog
  lj(a(X), rj(b(Y), c(U,V), Y=U), X=Y)
  ```
Source-to-Source Transformations

- Recall that $lj(A, B, C)$ is the union of:
  - Tuples from $A$ matching $C$ joined with $B$
  - Tuples from $A$ not matching $C$ joined with NULL's

- E.g.:

  $lj(s(X, U), t(V, Y), U>V)$

  $\begin{align*}
  s(1, 4). & \quad \text{answer}(X, Y, U, V) \rightarrow \\
  s(2, 3). & \quad \quad \{ \text{answer}(1, 4, 3, 5), \\
  t(3, 5). & \quad \quad \text{answer}(2, 3, \text{null}, \text{null}) \} 
  \end{align*}$
Source-to-Source Transformations

\[
v(X, Y) :- \text{lj}(s(X, U), t(V, Y), U>V).
\]

\[
v(X, Y) :- \text{lj}('p0'(X, U, V, Y)).
\]

\[
'p0'(A, B, C, D) :-
  'p1'(A, B, C, D).

'p0'(A, B, 'NULL'(C), 'NULL'(D)) :-
  s(A, B),
  not('p1'(A, B, E, F)).
\]

\[
p1'(A, B, C, D) :-
  s(A, B),
  t(C, D),
  B > C.
\]
Source-to-Source Transformations

- But, we get an unsafe rule because of floundering:
  \[ p_0(A, B, \text{'$NULL'(C), '$NULL'(D)')} \]

- However, such NULL specifications are otherwise treated in DES as null providers

- A null provider returns a unique identifier for a given tuple of ground values
Source-to-Source Transformations

\[
\text{'$p0' (A,B, '$NULL' (C), '$NULL' (D)) : - s(A,B), not('p1'(A,B,E,F)).}
\]

\[
\text{'$p0' (2,3, '$NULL' (1), '$NULL' (2))}
\]

\[
\text{'$p1' (A,B,C,D) : - s(A,B), t(C,D), B > C.}
\]

\[
\text{'$p1' (1,4,3,5)}
\]
Transfers to Other Systems

- The transformation includes a floundering rule, as E and F are not range restricted:

\[
\text{'$p0'(A, B, '$NULL'(C), '$NULL'(D))} :- \\
\hspace{1em} s(A, B), \\
\hspace{1em} \text{not('}$p1'(A, B, E, F))').
\]

- The meaning of not('}$p1') is unsafe, as it contains unbounded arguments:

\[
\text{not('}$p1'(2, 3, A, B))
\]
Transfers to Other Systems

- Usually, floundering is not allowed as in DLV
- However, some floundering programs can be translated into non-floundering [Ullman]
- This time we are lucky
Transfers: DLV

\[ v(X,Y) :- \text{lj}(s(X,U), t(V,Y), U>V). \]

\[ v(X,Y) :- '{\$p0}'(X,U,V,Y). \]

\[ '{\$p0}'(A,B, '{\$NULL}'(C), '{\$NULL}'(D)) :-
\]
\[ s(A,B), \text{not}('{\$p1}'(B)). \]

\[ '{\$p0}'(A,B,C,D) :-
\]
\[ s(A,B), t(C,D), B > C. \]

\[ '{\$p1}'(B) :-
\]
\[ s(A,B), t(C,D), B > C. \]

Solved at an extra cost
Transfers to Other Systems

- Another state-of-the-art system is XSB
- Here, built-in `sk_not/1` allows floundering by program transformation
Transfers: XSB

\[ v(X, Y) \leftarrow \text{lj}(s(X, U), t(V, Y), U > V). \]

```prolog
:- table('p0'/4), table('p1'/4).
:- table(s/2), table(t/2).
main(Vs) :- findall(v(X, Y), v(X, Y), Vs).
v(X, Y) :- 'p0'(X, U, V, Y).
'p0'(A, B, 'NULL'(C), 'NULL'(D)) :-
    get_id(C), get_id(D), s(A, B),
    sk_not('p1'(A, B, E, F)).
'p0'(A, B, C, D) :- 'p1'(A, B, C, D).
'p1'(A, B, C, D) :- s(A, B), t(C, D), B > C.
:- dynamic id/1.
id(0).
get_id(X) :-
id(X), retractall(id(X)), Y is X+1, assertz(id(Y)).
```
Recursion precludes:
- Duplicate elimination (IBM DB2, Oracle)
- Aggregates (IBM DB2, Oracle)

Recursion forces:
- Explicit recursion depth (MS SQL Server)
- Linearity (SQL Standard)
- Stratification on other operators (SQL Standard)

MySQL does not allow recursion at all!
Current RDBMS's Limitations

- **Example**
  - UNION ALL is required
  - An aggregate is not allowed in the recursive part

```sql
CREATE OR REPLACE VIEW shortest_paths(Origin, Destination, Length) AS
WITH RECURSIVE
  path(Origin, Destination, Length) AS
  (SELECT edge.*, 1 FROM edge)
  UNION
  (SELECT
    path.Origin, edge.Destination, path.Length + 1
  FROM path, edge
  WHERE path.Destination = edge.Origin and
    path.Length <
    (SELECT COUNT(*) FROM Edge) )
SELECT Origin, Destination, MIN(Length)
FROM path
GROUP BY Origin, Destination;
```
CREATE VIEW parent(parent,child) AS
   SELECT * FROM father
   UNION
   SELECT * FROM mother;

CREATE OR REPLACE VIEW ancestor(ancestor,descendant) AS
   WITH RECURSIVE rec_ancestor(ancestor,descendant) AS
      SELECT * FROM parent
   UNION
   SELECT parent,descendant
   FROM parent,rec_ancestor
   WHERE parent.child=rec_ancestor.ancestor
   SELECT * FROM rec_ancestor;

DES-SQL> SELECT * FROM ancestor WHERE ancestor='tom';
CREATE OR REPLACE VIEW ancestor(ancestor, descendant) AS
    SELECT parent, child FROM parent
    UNION
    SELECT parent, descendant FROM parent, ancestor
    WHERE parent.child=ancestor.ancestor;
Datalog Declarative Debugger

- **Motivation:**
  - Abstract the solving-oriented debugging procedure

- **Roots:**
  - [Shapiro83], Algorithmic Program Debugging

- **Semantics-oriented**

- **cf. Rafa talk**
Declarative Debugger

between(X,Z) :- br(X), br(Y), br(Z), X<Y, Y<Z .

Pairs of non-consecutive elements in the sequence

next(X,Y) :- br(X), br(Y), X<Y, not(between(X,Y)).

Consecutive elements in a sequence (starting at nil)

next(nil,X) :- br(X), not(has_preceding(X)).

has_preceding(X) :- br(X), br(Y), X > Y.

Elements having preceding values in the sequence

even(nil).
even(X) :- odd(Z), next(Z,X).

Elements in an even position+nil

odd(Y) :- even(Z), next(Z,Y).

Elements in an odd position

br_is_even :- even(X), not(next(X,Y)).

Succeeds if the cardinality is even

br(a).

br(b).

Base relation (sequence of elements)
Declarative Debugger

between(X,Z) :- br(X), br(Y), br(Z), X<Y, Y<Z.

Pairs of non-consecutive elements in the sequence

next(X,Y) :- br(X), br(Y), X<Y, not(between(X,Y)).

next(nil,X) :- br(X), not(has_preceding(X)).

Consecutive elements in a sequence (starting at nil)

has_preceding(X) :- br(X), br(Y), X<Y.

Elements having preceding values in the sequence

even(nil).

even(X) :- odd(Z), next(Z,X).

Elements in an even position+nil

odd(Y) :- even(Z), next(Z,Y).

Elements in an odd position

br_is_even :- even(X), not(next(X,Y)).

Succeeds if the cardinality is even

br(a).  

Base relation (sequence of elements)

br(b).
Declarative debugging: a practical session

DES> /debug br_is_even

Debugger started ...
Is \(br(b) = \{br(b)\}\) valid(v)/non-valid(n) [v]? v
Is has_preceding\(b\) = {} valid(v)/non-valid(n) [v]? n
Is \(br(X) = \{br(b),br(a)\}\) valid(v)/non-valid(n) [v]? v
! Error in relation: has_preceding/1
! Witness query: has_preceding\(b\) = {}
Declarative Debugging:
Semantic Graph

- \( \text{br}_\text{is\_even} = \{ \} \)
- \( \text{even}(X) = \{\text{even}(\text{nil})\} \)
- \( \text{odd}(X) = \{\text{odd}(\text{b})\} \)
- \( \text{next}(\text{b}, X) = \{ \} \)
- \( \text{next}(\text{nil}, Y) = \{\text{next}(\text{nil}, \text{b})\} \)
- \( \text{br}(\text{nil}) = \{ \} \)
- \( \text{br}(\text{b}) = \{\text{br}(\text{b})\} \)
- \( \text{br}(\text{a}) = \{\text{br}(\text{a})\} \)
- \( \text{br}(Y) = \{\text{br}(\text{a}), \text{br}(\text{b})\} \)
- \( \text{has\_preceding}(\text{a}) = \{\text{has\_preceding}(\text{a})\} \)
- \( \text{has\_preceding}(\text{b}) = \{ \} \)
- \( \text{has\_preceding}(\text{Y}) = \{ \} \)
SQL Debugger

- Motivation as of Datalog Debugger
- Adds traversing strategies
  - Divide & Query
    ```sql
divide_query
guest order(dq)
```
- Also, trusted tables
  ```sql
divide_query
guest trust_tables(no)
```
- And trusted specifications:
  ```sql
divide_query
guest trust_file(pets_trust)
```
SQL Debugger

DES-SQL> /debug_sql Guest
Info: Outcome of view 'LessThan6':
{ 'LessThan6'(1),
  'LessThan6'(2),
  'LessThan6'(3),
  'LessThan6'(4) }
Input: Is this view valid? (y/n/a) [y]: y
Info: Outcome of view 'NoCommonName':
{ 'NoCommonName'(1),
  'NoCommonName'(2),
  'NoCommonName'(3) }
Input: Is this view valid? (y/n/a) [y]: n
Info: Outcome of view 'CatsAndDogsOwner':
{ 'CatsAndDogsOwner'(1,'Wilma'),
  'CatsAndDogsOwner'(2,'Lucky'),
  'CatsAndDogsOwner'(3,'Rocky') }
Input: Is this view valid? (y/n/a) [y]: n
Info: Buggy view found: CatsAndDogsOwner/2.

DES-SQL> select * from Guest
answer(Guest.id, Guest.name) ->
{ answer(1,'Mark Costas'),
  answer(2,'Helen Kaye'),
  answer(3,'Robin Scott') }
Info: 3 tuples computed

9/21/2011 132 / 141

PROMETIDOS-CM Summer School
Datalog Tracer

DES-Datalog> /c negation
DES-Datalog> /trace_datalog a
Info: Tracing predicate 'a'.
{
  a
}
Info: 1 tuple in the answer table.
Info: Remaining predicates: [b/0,c/0,d/0]
Input: Continue? (y/n) [y]:
Info: Tracing predicate 'b'.
{
  not(b)
}
Info: 1 tuple in the answer table.
Info: Remaining predicates: [c/0,d/0]
Input: Continue? (y/n) [y]:
Info: Tracing predicate 'c'.
{
  c
}
Info: 1 tuple in the answer table.
Info: Remaining predicates: [d/0]
Input: Continue? (y/n) [y]:
Info: Tracing predicate 'd'.
{
}
Info: No more predicates to trace.
SQL Tracer

DES-SQL> /trace_sql ancestor
Info: Tracing view 'ancestor'.
{
    ancestor(amy, carolIII),
    ...
    ancestor(tony, carolIII)
}
Info: 16 tuples in the answer table.
Info: Remaining views: [parent/2, father/2, mother/2]
Input: Continue? (y/n) [y]: [y]:
Info: Tracing view 'parent'.
{
    parent(amy, fred),
    ...
    parent(tony, carolIII)
}
Info: 8 tuples in the answer table.
Info: Remaining views: [father/2, mother/2]
Input: Continue? (y/n) [y]: [y]:
Info: Tracing view 'father'.
{
    father(fred, carolIII),
    ...
    father(tony, carolIII)
}
Info: 4 tuples in the answer table.
Info: No more views to trace.
DES-SQL> /trace_datalog father(X,Y)
Info: Tracing predicate 'father'.
{
    father(fred, carolIII),
    ...
    father(tony, carolIII)
}
Info: 4 tuples in the answer table.
Info: No more predicates to trace.
SQL Test Case Generator

- Provides tuples that can be matched to the intended interpretation of a view

- Test cases
  - Positive (PTC)
  - Negative (NTC)

- Querying a view w.r.t.
  - PTC: One tuple, at least
  - NTC: One tuple, at least, which does not match the WHERE condition

- Predicate coverage:
  - PNTC: Contains both PTC and NTC tuples
SQL Test Case Generator

- **PNTC**
  DES-SQL> create table t(a int primary key)
  DES-SQL> create view v(a) as select a from t where a=5
  Info: Test case over integers:
  [t(5), t(-5)]

- **No PNTC**
  create view v(a) as select a from t
  where a=1 and not exists (select a from t where a<>1);

- **Support for:**
  - Aggregates
  - UNION
  - Options:
    - Adding/replacing results to a table
    - Kind of generated test case (PTC, NTC, PNTC)
    - Test case size
Several other Features of DES

- Tabling with support for relational features
- Preprocessor:
  - Source-to-source program transformation for safety and performance
  - Program analysis for safety
- Development mode
- Batch processing
- A wide set of commands (>70)
- Textual API for communicating to external applications
Conclusions

- Relational vs. Deductive databases:
  - FOL is not enough
  - SQL:
    - Syntax oriented to human speech
    - Many functionalities
  - Datalog:
    - Syntax oriented to formulas
    - Neater formulations
    - New needs coming from Semantic Web, Ontologies, ...

- But, syntax is not all, a practical system is the key
Conclusions. DES

- Successful implementation guided by need
- Widely used, both for teaching and research
- Not really novel for each feature but as a whole
  - Datalog and SQL integration
  - Interactive, user-friendly, multiplatform system
  - Just download it and play!
  - Nevertheless the aforementioned novel features
  - Still, many things to do…
Limitations (Future Work)

- Data are constants, no terms (functions) are allowed
- Datalog database updates
- Beyond 2.5VL
- SQL coverage still incomplete
- Precise syntax error reports
- Single-line inputs
- Constraints (à la CLP)
- Performance
- … only to name a few!
Efficient Integrity Checking for Databases with Recursive Views
Davide Martinenghi and Henning Christiansen
Autor Johann Eder, Hele-Mai Haav, Ahto Kalja, Juan Penjam
ISBN 3540285857, 9783540285854

PhD
Computer Science and Engineering Department
University of Nebraska - Lincoln, USA

PhD
University of Texas at San Antonio, USA

Industry:
- XLOG Technologies GmbH, Zürich
- CaseLab: Applied Operations Research
- Ideacube

Links to DES:
- ACM SIGMOD Online: Publicly Available Database Software from Nonprofit Organizations
- The ALP Newsletter, vol. 21 n. 1
- Datalog Wikipedia: German
- Datalog Wikipedia: English
- Wapedia
- SWI-Prolog: Related Web Resources
- SICStus Prolog: Third Party Software, Other Research Systems
- SOFTPEDIA: Datalog Educational System 1.7.0
- Famouswhy
- DBpedia
- BDD-Based Deductive Database (bddbddd): Other implementations of Datalog/Ficolog
- Reach Information
- Ask a Word
- Acronym finder
- Acronym Geek

DES Facts

PhD Computer Science and Engineering Department
University of Nebraska - Lincoln, USA

PhD Computer Science and Engineering Department
University of Nebraska - Lincoln, USA