A Deductive Database with Datalog and SQL Query Languages

Fernando Sáenz Pérez, Rafael Caballero and Yolanda García-Ruiz

Grupo de Programación Declarativa (GPD)

Universidad Complutense de Madrid (Spain)

Contents

1. Introduction

- Some concepts:
  - Database (DB)
  - Database Management System (DBMS)
  - Data model
    - (Abstract) data structures
    - Operations
    - Constraints

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)
    - bddbddb (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)

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 were asking for such a system?
  - A system oriented at teaching
  - User-friendly:
    - Installation
    - Usability
  - Multiplatform (Windows, Linux, Mac, …)
  - Interactive
  - Query languages
  - …

DES Concrete Features (1/4)

- Free, Open-source, Multiplatform, Portable
- Query languages:
  - (Extended) Datalog
  - (Recursive) SQL following ANSI/ISO standard
- Stratified Negation
- Integrity constraints
- Duplicates
- Null value support *à la* SQL
- Outer joins for both SQL and Datalog
- Aggregates

DES Concrete Features (2/4)

- Declarative debuggers and tracers
- Test case generator for SQL views
- Full-fledged arithmetic
- Database updates
- Temporary Datalog views
- Type system
- Batch processing
- Textual API

DES Concrete Features (3/4)

- Program analysis:
  - Safe rules (classical safety for range restriction)
    - Negation
    - Head variables
  - Safe metapredicates
    - Aggregates
    - Duplicate elimination
- Source-to-source program transformations:
  - Safety (command `/safe on`)
  - Aggregates (solving)
  - Performance (`/simplification on`)
  - ...

DES Concrete Features (4/4)

- But, quite relevant features are:
  - Interactiveness
    - Database updates
    - A wide set of commands (>70)
  - Easy to install and use
    `des.sourceforge.net`
  - Robust (up to bugs)

DES allows

- Teach (Declarative) Query Languages: From SQL to Datalog
- But also for rapid prototyping:
  - Novel features:
    - SQL hypothetical queries
    - Outer joins in Datalog
    - Datalog and SQL declarative (algorithmic) debuggers and tracers
    - Test case generation for SQL views
- ... and Experiment with Datalog for research
  - Theses, Papers, … See DES Facts at its web page

2. Query Languages. Datalog

- A database query language stemming from Prolog

Prolog	Datalog
Predicate	Relation
Goal	Query

  - Goals are solved one answer at a time (backtracking)
  - Queries are solved by computing its meaning once
- Deductive database:
  - Extensional database: Facts
  - Intensional database: Rules.

Datalog

- Datalog differs from Prolog:
  - Datalog does not allow function symbols in arguments
  - Facts are ground (safety)
  - Datalog is truly declarative:
    - Clause order is irrelevant
    - Order of literals in a body is irrelevant
    - No extra-logical constructors as the feared cut

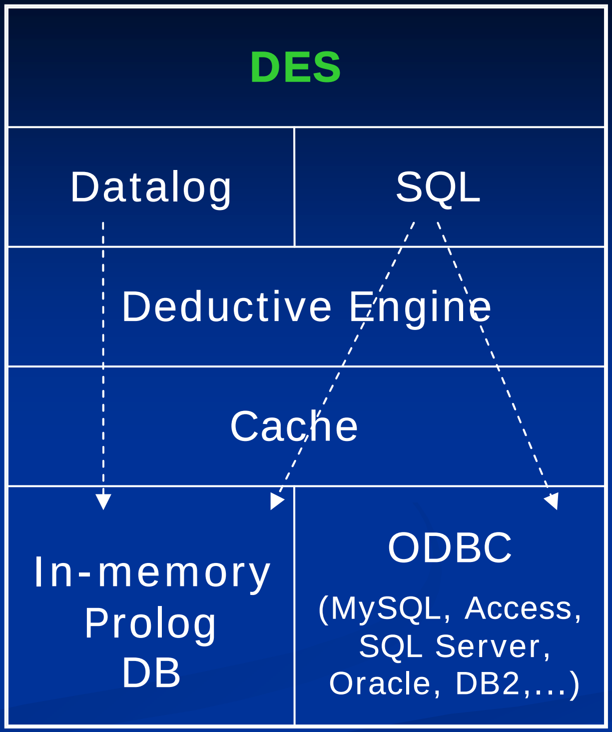
Datalog Syntax

- 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, …)

Query Languages. SQL

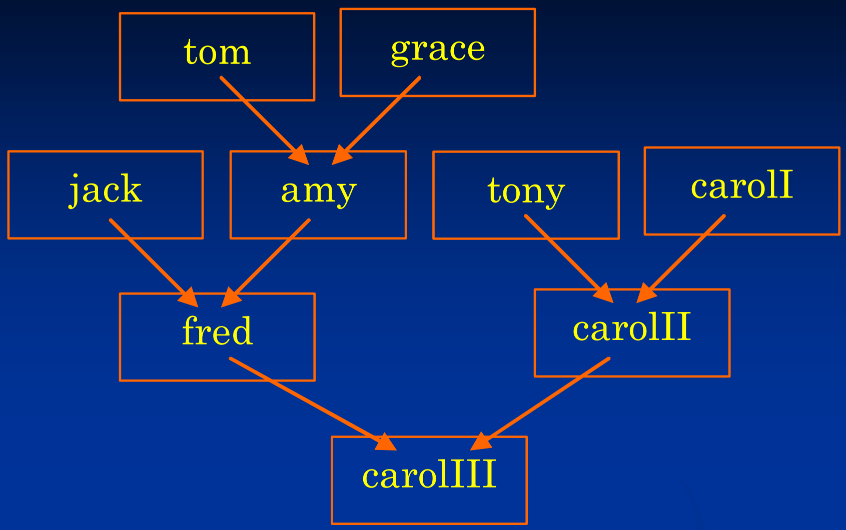
- Follows ISO Standard
- DQL:
  - `SELECT` ***Expressions*** `FROM` ***Relations*** `WHERE` ***Condition***
  - `WITH RECURSIVE` ***LocalViewDefs Statement***
  - `ASSUME` ***LocalViewDefs*** `IN` ***Statement*** (ongoing work)
- DML:
  - `INSERT` …
  - `UPDATE` …
  - `DELETE` …
- DDL:
  - `CREATE [OR 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

Datalog Example



- Facts:

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

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

- Rules:

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

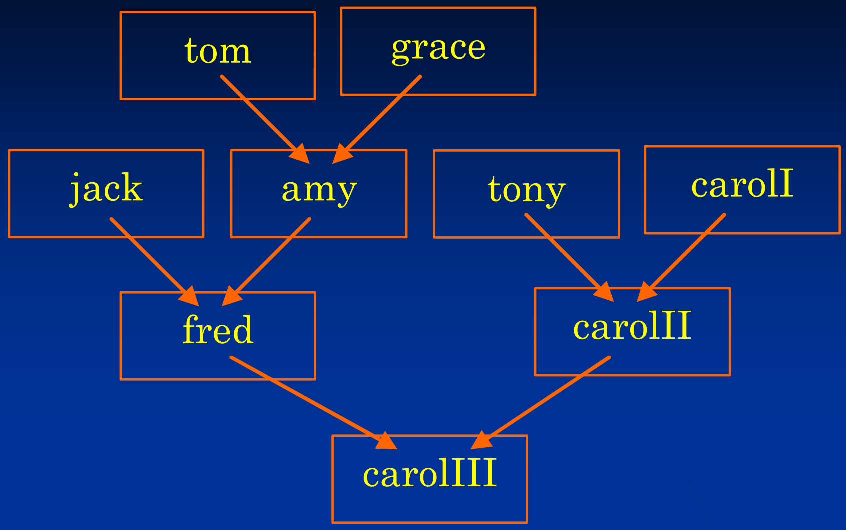
- Query:

```
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).
```

DES implements a fixpoint semantics for recursive Datalog, finding the least fixpoint as answer

```
ancestor(tom,X)

{
  ancestor(tom,amy),
  ancestor(tom,carolIII),
  ancestor(tom,fred)
}
```

SQL Recursion as of Current DBMS's

```
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';
```

SQL Simplified Syntax in DES

- Simply:

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

- Instead of resorting to WITH:

```
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;
```

SQL Hypothetical Queries

```
CREATE TABLE flight(origin string, destination string, time
   real)

INSERT INTO flight VALUES('lon','ny',9.0);
INSERT INTO flight VALUES('mad','par',1.5);
INSERT INTO flight VALUES('par','ny',10.0);

CREATE VIEW travel(origin,destination,time) AS
WITH connected(origin,destination,time) AS
    SELECT * FROM flight
  UNION
    SELECT flight.origin, connected.destination,
           flight.time+connected.time
    FROM flight, connected
    WHERE flight.destination = connected.origin
SELECT * FROM connected;
```

SQL Hypothetical Queries

```
DES-Datalog> SELECT * FROM travel
{
  answer(lon,ny,9.0),
  answer(mad,ny,11.5),
  answer(mad,par,1.5),
  answer(par,ny,10.0)
}
Info: 4 tuples computed.

DES-Datalog> ASSUME SELECT 'mad','lon',2.0
             IN flight(origin,destination,time)
             SELECT * FROM travel;
{
  answer(lon,ny,9.0),
  answer(mad,lon,2.0),
  answer(mad,ny,11.0),
  answer(mad,ny,11.5),
  answer(mad,par,1.5),
  answer(par,ny,10.0)
}
Info: 6 tuples computed.
```

3. 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

Types vs. Domains

- Imposing type constraints:
  - SQL table creation
  - Interactive type assertions (*even* at the command prompt)

- 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.
```

User-defined Integrity Constraints

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

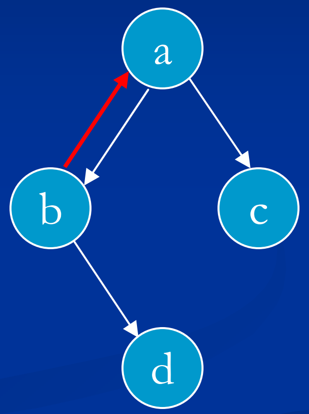
```
CREATE TABLE t(c INT CHECK (c BETWEEN 0 AND 10));
```

- Datalog:

```
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.
```



4. Duplicates

- SQL is not set-oriented, rather it allows duplicates in base relations and query outcomes
- So, for supporting SQL as Datalog programs we need:
  - Multisets
  - Duplicate elimination

Duplicates as of DES

- Duplicates are disabled by default

```
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:

```
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)
}
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
```

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)).
```

5. Outer Joins

- Null values:
  - Cte.: `null`
  - Functions: `is_null(Var)`
    `is_not_null(Var)`
- Outer join built-ins:
  - Left (⟕): `lj(Left_Rel, Right_Rel, ON_Condition)`
  - Right (⟖): `rj(Left_Rel, Right_Rel, ON_Condition)`
  - Full (⟗): `fj(Left_Rel, Right_Rel, ON_Condition)`

Outer Join Examples

- SQL:

```
SELECT * FROM a LEFT JOIN b ON x=y;
```

- Datalog:

```
lj(a(X), b(Y), X=Y)
```

- SQL:

```
SELECT * FROM a LEFT JOIN b WHERE x=y;
```

- Datalog:

```
lj(a(X), b(X), true)
```

- SQL:

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

- Datalog:

```
lj(a(X), rj(b(Y), c(U,V), Y=U), X=Y)
```

6. Aggregates

- DES offers two possibilities:
  - A 'group by' metapredicate with expressions including aggregate functions
  - Aggregate predicates with grouping criteria at the rule head

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`

```
group_by(
   Relation,               % FROM / WHERE
   [Var_1,…,Var_n],        % Grouping columns
   Condition)              % 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.
```

employee

Name	*Department*	*Salary*
anderson	accounting	1200
andrews	accounting	1200
arlingon	accounting	1000
nolan	null	null
norton	null	null
randall	resources	800
sanders	sales	null
silver	sales	1000
smith	sales	1000
Steel	sales	1020
Sullivan	sales	null

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.
```

employee

Name	*Department*	*Salary*
anderson	accounting	1200
andrews	accounting	1200
arlingon	accounting	1000
nolan	null	null
norton	null	null
randall	resources	800
sanders	sales	null
silver	sales	1000
smith	sales	1000
Steel	sales	1020
Sullivan	sales	null

Aggregates

- Aggregate metapredicates:
  - `count(Relation,Result)`
  - `count(Relation,CountedVar,Result)`
  - `min(Relation,Result)`
  - `max(Rel,Result)`
  - `sum(Rel,SummedVar,Result)`
  - `avg(Rel,AvgdVar,Result)`
  - `times(Rel,MultdVar,Var)`

```
DES-Datalog> count(employee(_,_,_),C)
Info: Processing:
  answer(C) :- count(employee(_,_,_),[],C).
```

Aggregate 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)
{
  c(accounting,3),
  c(null,0),
  c(resources,1),
  c(sales,3)
}
```

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).
```

Safety and Aggregates

- 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`

7. Deguggers and Tracers. Datalog Declarative Debugger

- Motivation:
  - Abstract the solving-oriented debugging procedure
- Roots:
  - [Shaphiro83], Algorithmic Program Debugging
- Semantics-oriented

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

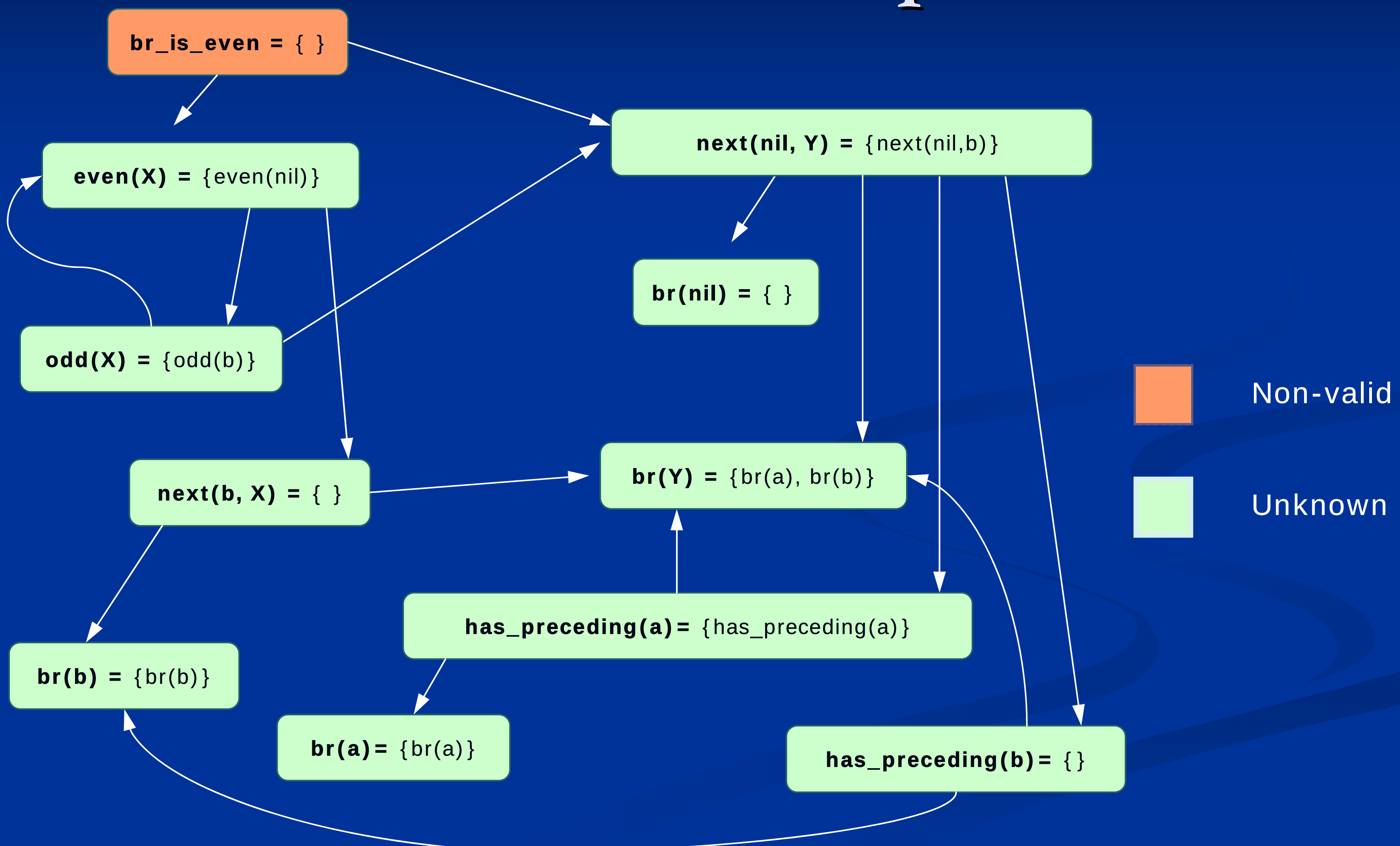
```
br_is_even :- even(X), not(next(X,Y)).
```

Succeeds if the cardinality is even

```
br(a).
br(b).
```

Base relation (sequence of elements)

Declarative Debugging: Semantic Graph



Declarative Debugging: A Practical Session

```
DES> /debug_datalog 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) = { }
```

SQL Debugger

- Motivation as of Datalog Debugger
- Adds traversing strategies
  - Divide & Query

```
/debug_sql Guest order(dq)
```

- Also, trusted tables

```
/debug_sql Guest trust_tables(no)
```

- And trusted specifications:

```
/debug_sql Guest trust_file(pets_trust)
```

SQL Debugger

Owner(id integer primary key,
name varchar(50));

Pet(code integer primary key,
name varchar(50), specie
varchar(20));

PetOwner(id integer, code
integer, primary key (id,code),
foreign key (id) references
Owner(id), foreign key (code)
references Pet(code))

AnimalOwner

LessThan6 (AnimalOwner)

CatsAndDogsOwner
(AnimalOwner)

NoCommonName
(CatsAndDogsOwner)

Guest (NoCommonName,
LessThan6)

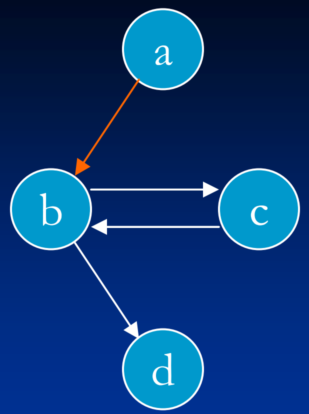
```
DES-SQL> select * from Guest
answer(Guest.id, Guest.name) ->
{
  answer(1,'Mark Costas'),
  answer(2,'Helen Kaye'),
  answer(3,'Robin Scott')
}
Info: 3 tuples computed.
```

```
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: Outcome of view 'AnimalOwner':
{
  AnimalOwner(1,'Kitty',cat),
  AnimalOwner(1,'Wilma',dog),
  AnimalOwner(2,'Lucky',dog),
  AnimalOwner(2,'Wilma',cat),
  AnimalOwner(3,'Oreo',cat),
  AnimalOwner(3,'Rocky',dog),
  AnimalOwner(4,'Cecile',turtle),
  AnimalOwner(4,'Chelsea',dog)
}
Input: Is this view valid? (y/n/a) [y]: y
Info: Buggy view found:
      CatsAndDogsOwner/2.
```

Datalog Tracer

```
a :- not(b).
b :- c,d.
c :- b.
c.
```




```
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]:
Info: Tracing view 'parent'.
{
  parent(amy,fred), ...
  parent(tony,carolII)
}
Info: 8 tuples in the answer table.
Info : Remaining views: [father/2,mother/2]
Input: Continue? (y/n) [y]:
Info: Tracing view 'father'.
{
  father(fred,carolIII), ...
  father(tony,carolII)
}
Info: 4 tuples in the answer table.
```

```
Info : Remaining views: [mother/2]
Input: Continue? (y/n) [y]:
Info: Tracing view 'mother'.
{
  mother(amy,fred), ...
  mother(grace,amy)
}
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,carolII)
}
Info: 4 tuples in the answer table.
Info: No more predicates to trace.
```

8. 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
DES-SQL> /test_case v
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:
  - Integer and string types
  - Aggregates, UNION
  - Options:
    - Adding/replacing results to a table
    - Kind of generated test case (PTC, NTC, PNTC)
    - Test case size

9. Conclusions

- Successful implementation guided by need
- Widely used, both for teaching and research
  - More than 35,000 downloads
  - Up to more than 1,500 downloads/month
- Includes novel features
  - Hypothetical SQL
  - Declarative debuggers
  - Outer joins
- But key factors are also:
  - Datalog and SQL integration
  - Interactive, user-friendly, multiplatform system
  - Just download it and play!

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
- Constraints (*à la* CLP)
- Performance
- … only to name a few!

Canada

- [Efficient Integrity Checking for Databases with Recursive Views]
  Davide Martinenghi and Henning Christiansen
  In Advances in Databases and Information Systems: 9th East European Conference, ADBIS 2005, Tallinn, Estonia, September 12-15, 2005 : Proceedings
  Autor Johann Eder, Hele-Mai Haav, Ahto Kalja, Jaan 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 (bddbddb)
  Other implementations of Datalog/Prolog
- Reach Information
- Ask a Word
- Acronym finder
- Acronym Geek
- .

DES Facts

- University of California, at Los Angeles
  CS240A: Databases and Knowledge Bases
- The University of Arizona
  CsC372
- The State University of New York
  University at Buffalo
  CSE 636: Data Integration
- The University of British Columbia
  CS304: Introduction to Relational Databases
  Datalog Tutorial
- Master's of Information Technology in Arkansas Tech University, Russellville
- The University of Texas at Austin
  CS2

Australia:

- INFO2820: Database Systems 1 (Advanced) (2010 - Semester 1)
  Engineering and Information Technologies
  The University of Sydney
  Tab "Resources"
- INFO2120/2820: Database Systems 1 (2009 - Semester 1)
  School of Information Technologies
  The University of Sydney
  Tutorial 3
- Allan Hancock College >> INFO >> 2120 Fall, 2009
  Description: School of Information Technologies
  INFO2120/2820: Database Systems I 1.Sem./2009
  Tutorial 3: SQL and Relational Algebra 23.03.2009

Africa:

- Faculty of Sciences and Technologies of Mohammedia (FSTM) - Morocco