Datalog and Emerging Applications: an Interactive Tutorial

Shan Shan Huang       T.J. Green       Boon Thau Loo

LOGICBLOX®           UCDAVIS
                      UNIVERSITY OF CALIFORNIA

SIGMOD 2011            Athens, Greece           June 14, 2011
A Brief History of Datalog

Workshop on Logic and Databases

'77
A Brief History of Datalog

Workshop on Logic and Databases

‘77  ‘80s ...

LDL, NAIL, Coral, ...

3
A Brief History of Datalog

Workshop on Logic and Databases

Data integration

LDL, NAIL, Coral, ...

‘77  ‘80s ...  ‘95
A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases

Data integration

‘77
‘80s ...
‘95

LDL, NAIL, Coral, ...

Controlling + data flow
A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases

Data integration

‘77

’80s ...

‘95

LDL, NAIL, Coral, ...

Workshop on Logic and Databases

Data integration

Control + data flow
A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases

Data integration

‘77 ‘80s

LDL, NAIL, Coral, ...

No practical applications of recursive query theory ... have been found to date.

-- Hellerstein and Stonebraker “Readings in Database Systems”
A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases

Data integration

'77   '80s ...   '95

LDL, NAIL, Coral, ...
A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases

Data integration

LDL, NAIL, Coral, ...

Access control (Binder)

'77

'80s ...

'95

'02
A Brief History of Datalog

- '77: Workshop on Logic and Databases
- '80s...: Data integration
- '95: Control + data flow
- '02: Access control (Binder)
- '05: Declarative networking

LDL, NAIL, Coral, ...
A Brief History of Datalog

- '77: Workshop on Logic and Databases
- '80s: Data integration
- '95: Control + data flow
- '02: Access control (Binder)
- '05: Declarative networking

LDL, NAIL, Coral, ...

Control + data flow

BDDBDDB

Declarative networking
A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases

Data integration

‘77 ’80s … ’95

LDL, NAIL, Coral, ...

Declarative networking

BDDBDDB

Orchestra CDSS

Access control (Binder)

‘02 ‘05
A Brief History of Datalog

- Workshop on Logic and Databases
- Data integration
- Control + data flow

- LDL, NAIL, Coral, ...

- Declarative networking
- Orchestra CDSS
- Information Extraction
- Access control (Binder)

- '77
- '80s ...
- '95
- '02
- '05
- '07
A Brief History of Datalog

Control + data flow

Data integration

Workshop on Logic and Databases

‘77 ‘80s … ‘95

LDL, NAIL, Coral, ...

Declarative networking

BDDBDDB

Orchestra CDSS

Information Extraction

Access control (Binder)

‘02 ‘05 ‘07

.QL
A Brief History of Datalog

- Workshop on Logic and Databases
- Control + data flow
- Data integration
- LDL, NAIL, Coral, ...

- Orchestra CDSS
- Information Extraction
- Access control (Binder)
- .QL
- Declarative networking
- BDDBDBDB
- Doop (pointer-analysis)

Timeline:
- '77
- '80s...
- '95
- '02
- '05
- '07
- '08
A Brief History of Datalog

- Control + data flow
- Data integration
- Workshop on Logic and Databases
- LDL, NAIL, Coral, ...

- Declarative networking
- BDDBDDB
- Orchestra CDSS
- Information Extraction
- Access control (Binder)
- Doop (pointer-analysis)
- Evita Raced
- .QL

- 1977
- 1980s ...
- 1995
- 2002
- 2005
- 2007
- 2008
A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases

Data integration

LDL, NAIL, Coral, ...

Declarative networking

BDDBDDBDB

Orchestra CDSS

Information Extraction

Access control (Binder)

Doop (pointer-analysis)

Evita Raced

.QL

SecureBlox

‘77 ‘80s ... ‘95

‘02 ‘05 ‘07 ‘08 ‘10
A Brief History of Datalog

- '77: Workshop on Logic and Databases
- '80s: Data integration
- '95: Control + data flow

LDL, NAIL, Coral, ...

- '02: Access control (Binder)
- '05: Declarative networking
- '07: Blockchain
- '08: Information Extraction
- '10: SecureBlox

Evita Raced
Doop (pointer-analysis)
.QL
A Brief History of Datalog

'77 Workshop on Logic and Databases
'80s ..., '95 Control + data flow

'77 LDL, NAIL, Coral, ...

'95 Data integration

'02 Access control (Binder)
'05 Declarative networking
'07 BDDBDDB
'08 Orchestra CDSS
'10 SecureBlox

Information Extraction

Doop (pointer-analysis)
Evita Raced
.QL

Evita
Raced

LDL, NAIL, Coral, ...

Control + data flow
A Brief History of Datalog

- Workshop on Logic and Databases
- Data integration
- Control + data flow
- LDL, NAIL, Coral, ...

- Workshop on Logic and Databases
- Data integration
- Control + data flow
- LDL, NAIL, Coral, ...

- Declarative networking
- BDDBDDI
- Orchestra CDSS
- Information Extraction
- Access control (Binder)
- Evita Raced
- .QL

- Workshop on Logic and Databases
- Data integration
- Control + data flow
- LDL, NAIL, Coral, ...

- Workshop on Logic and Databases
- Data integration
- Control + data flow
- LDL, NAIL, Coral, ...

- Declarative networking
- BDDBDDI
- Orchestra CDSS
- Information Extraction
- Access control (Binder)
- Evita Raced
- .QL
A Brief History of Datalog

- Workshop on Logic and Databases
- Data integration
- Control + data flow

- LDL, NAIL, Coral, ...

- Access control (Binder)
- Doop (pointer-analysis)
- Evita Raced
- BDDBDII
- Orchestra CDSS
- Information Extraction
- Declarative networking
- SecureBlox

- '77 '80s ...
- '95
- '02 '05 '07 '08 '10
- .QL
- LOGICBLOX®
- semmle
A Brief History of Datalog

Hey wait... there ARE applications!
Today’s Tutorial, or,
Datalog: Taste it Again for the First Time

• We review the basics and examine several of these recent applications

• Theme #1: *lots* of compelling applications, if we look beyond payroll / bill-of-materials / ...
  
  – Some of the most interesting work coming from *outside* databases community!

• Theme #2: language extensions usually needed
  
  – To go from a toy language to something really usable
(Asynchronously!)

An Interactive Tutorial

- INSTALL_LB : installation guide
- README : structure of distribution files
- Quick-Start guide : usage
- *.logic : Datalog examples
- *.lb : LogicBlox interactive shell script (to drive the Datalog examples)
- Shan Shan and other LogicBlox folks will be available immediately after talk for the “synchronous” version of tutorial
Outline of Tutorial

June 14, 2011: The Second Coming of Datalog!

• Refresher: Datalog 101
• Application #1: Data Integration and Exchange
• Application #2: Program Analysis
• Application #3: Declarative Networking
• Conclusions
Datalog Refresher: Syntax of Rules

Datalog rule syntax:

<result> ← <condition1>, <condition2>, ..., <conditionN>.
Datalog Refresher: Syntax of Rules

Datalog rule syntax:

\[
\text{<result>} \leftarrow \text{<condition1>, <condition2>, ... , <conditionN>}. \\
\]

Body
Datalog Refresher: Syntax of Rules

Datalog rule syntax:

\[ \text{<result>} \leftarrow \text{<condition1>}, \text{<condition2>}, \ldots, \text{<conditionN>} \].

Head \hspace{2cm} Body

\(\times\) Body consists of one or more conditions (input tables)

\(\times\) Head is an output table

\(\hspace{0.5cm}\bullet\) Recursive rules: result of head in rule body
Example: All-Pairs Reachability

R1: reachable(S,D) <- link(S,D).
R2: reachable(S,D) <- link(S,Z), reachable(Z,D).

Input: link(source, destination)
Output: reachable(source, destination)
Example: All-Pairs Reachability

R1: reachable(S,D) ← link(S,D).
R2: reachable(S,D) ← link(S,Z), reachable(Z,D).

link(a,b) – “there is a link from node a to node b”

Input: link(source, destination)
Output: reachable(source, destination)
Example: All-Pairs Reachability

R1: reachable(S,D) ← link(S,D).
R2: reachable(S,D) ← link(S,Z), reachable(Z,D).

`link(a,b)` – “there is a link from node `a` to node `b`”

`reachable(a,b)` – “node `a` can reach node `b`”

Input: link(source, destination)
Output: reachable(source, destination)
Example: All-Pairs Reachability

R1: reachable(S,D) <- link(S,D).
R2: reachable(S,D) <- link(S,Z), reachable(Z,D).

“For all nodes S,D,
If there is a link from S to D, then S can reach D”.

Input: link(source, destination)
Output: reachable(source, destination)
Example: All-Pairs Reachability

R1: \texttt{reachable(S,D) <- link(S,D).}

R2: \texttt{reachable(S,D) <- link(S,Z), reachable(Z,D).}

“For all nodes S, D and Z,
If there is a link from S to Z, AND Z can reach D, then S can reach D’.

\begin{itemize}
  \item \textbf{Input:} link(source, destination)
  \item \textbf{Output:} reachable(source, destination)
\end{itemize}
Terminology and Convention

- An **atom** is a **predicate**, or relation name with **arguments**.
- Convention: Variables begin with a capital, predicates begin with lower-case.
- The **head** is an atom; the **body** is the AND of one or more atoms.
- **Extensional database predicates (EDB)** – source tables
- **Intensional database predicates (IDB)** – derived tables

reachable(S,D) <- link(S,Z), reachable(Z,D) .
Negated Atoms

• We may put ! (NOT) in front of a atom, to negate its meaning.
Negated Atoms

- We may put ! (NOT) in front of a atom, to negate its meaning.

Not “cut” in Prolog. 😊
Negated Atoms

• We may put ! (NOT) in front of a atom, to negate its meaning.
• Example: For any given node S, return all nodes D that are two hops away, where D is not an immediate neighbor of S.

```
twoHop(S,D)
<- link(S,Z),
    link(Z,D)
    ! link(S,D).
```

S --- link(S,Z) --- Z --- link(Z,D) --- D
Safe Rules

• Safety condition:
  – Every variable in the rule must occur in a positive (non-negated) relational atom in the rule body.
  – Ensures that the results of programs are finite, and that their results depend only on the actual contents of the database.
Safe Rules

• Safety condition:
  – Every variable in the rule must occur in a positive (non-negated) relational atom in the rule body.
  – Ensures that the results of programs are finite, and that their results depend only on the actual contents of the database.

• Examples of unsafe rules:
  – \( s(X) \leftarrow r(Y) \).
  – \( s(X) \leftarrow r(Y), ! r(X) \).
Semantics

• Model-theoretic
  – Most “declarative”. Based on model-theoretic semantics of first order logic. View rules as logical constraints.
  – Given input DB I and Datalog program P, find the smallest possible DB instance I’ that extends I and satisfies all constraints in P.
Semantics

• **Model-theoretic**
  
  — Most “declarative”. Based on model-theoretic semantics of first order logic. View rules as logical constraints.
  
  — Given input DB I and Datalog program P, find the smallest possible DB instance I’ that extends I and satisfies all constraints in P.

• **Fixpoint-theoretic**
  
  — Most “operational”. Based on the immediate consequence operator for a Datalog program.
  
  — Least fixpoint is reached after finitely many iterations of the immediate consequence operator.
  
  — Basis for practical, bottom-up evaluation strategy.
Semantics

• **Model-theoretic**
  - Most “declarative”. Based on model-theoretic semantics of first order logic. View rules as logical constraints.
  - Given input DB I and Datalog program P, find the smallest possible DB instance I’ that extends I and satisfies all constraints in P.

• **Fixpoint-theoretic**
  - Most “operational”. Based on the immediate consequence operator for a Datalog program.
  - Least fixpoint is reached after finitely many iterations of the immediate consequence operator.
  - Basis for practical, bottom-up evaluation strategy.

• **Proof-theoretic**
  - Set of provable facts obtained from Datalog program given input DB.
  - Proof of given facts (typically, top-down Prolog style reasoning)
The “Naïve” Evaluation Algorithm

1. Start by assuming all IDB relations are empty.
2. Repeatedly evaluate the rules using the EDB and the previous IDB, to get a new IDB.
3. End when no change to IDB.
Naïve Evaluation

reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z), reachable(Z,D).
Naïve Evaluation

reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z), reachable(Z,D).
Naïve Evaluation

reachable(S,D) ← link(S,D).
reachable(S,D) ← link(S,Z),
reachable(Z,D).
Naïve Evaluation

```
reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z), reachable(Z,D).
```
Naïve Evaluation

reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z), reachable(Z,D).
Naïve Evaluation

reach(S,D) <- link(S,D).
reach(S,D) <- link(S,Z),
reach(Z,D).
Naïve Evaluation

reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z),
reachable(Z,D).

reachable
Naïve Evaluation

reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z), reachable(Z,D).
Semi-naïve Evaluation

• Since the EDB never changes, on each round we only get new IDB tuples if we use at least one IDB tuple that was obtained on the previous round.

• Saves work; lets us avoid rediscovering most known facts.
  – A fact could still be derived in a second way.
Semi-naïve Evaluation

reachable(S,D) ← link(S,D).
reachable(S,D) ← link(S,Z), reachable(Z,D).
Semi-naïve Evaluation

reachable(S,D) ← link(S,D).
reachable(S,D) ← link(S,Z), reachable(Z,D).
Semi-naïve Evaluation

reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z), reachable(Z,D).
Semi-naïve Evaluation

reachable(S, D) ← link(S, D).
reachable(S, D) ← link(S, Z), reachable(Z, D).

~
Semi-naïve Evaluation

reachable(S,D) ← link(S,D).
reachable(S,D) ← link(S,Z),
reachable(Z,D).

reachables and links diagram
Semi-naïve Evaluation

reachable(S,D) ← link(S,D).
reachable(S,D) ← link(S,Z), reachable(Z,D).
Semi-naïve Evaluation

reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z), reachable(Z,D).
Semi-naïve Evaluation

reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z),
reachable(Z,D).
Recursion with Negation

Example: to compute all pairs of disconnected nodes in a graph.

reach(S,D) <- link(S,D).
reach(S,D) <- link(S,Z), reach(Z,D).
unreach(S,D) <- node(S), node(D), ! reach(S,D).
Recursion with Negation

Example: to compute all pairs of disconnected nodes in a graph.

\[
\begin{align*}
\text{reachable}(S,D) & \leftarrow \text{link}(S,D). \\
\text{reachable}(S,D) & \leftarrow \text{link}(S,Z), \text{reachable}(Z,D). \\
\text{unreachable}(S,D) & \leftarrow \text{node}(S), \text{node}(D), \neg \text{reachable}(S,D).
\end{align*}
\]

**Precedence graph**: Nodes = IDB predicates. Edge \( q \leftarrow p \) if predicate \( q \) depends on \( p \). Label this arc “–” if the predicate \( p \) is negated.
Stratified Negation

- Straightforward syntactic restriction.
- When the Datalog program is stratified, we can evaluate IDB predicates lowest-stratum-first.
- Once evaluated, treat it as EDB for higher strata.

reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z),
    reachable(Z,D).
unreachable(S,D) <- node(S),
    node(D),
    ! reachable(S,D).

Stratum 0

reachable

unreachable

Stratum 1

--
Stratified Negation

- Straightforward syntactic restriction.
- When the Datalog program is stratified, we can evaluate IDB predicates lowest-stratum-first.
- Once evaluated, treat it as EDB for higher strata.

Non-stratified example:

\[ p(X) \leftarrow q(X), \neg p(X). \]
A Sneak Preview...

• Data integration
  – Skolem functions
• Program analysis
  – Type-based optimization
• Declarative networking
  – Aggregates, aggregate selections
  – Incremental view maintenance
  – Magic sets
Suggested Readings

• Survey papers:
  • *What you always wanted to know about datalog (and never dared to ask)*, by Ceri, Gottlob, and Tanca.
  • Database Encyclopedia entry on “DATALOG”. Grigoris Karvounarakis.

• Textbooks:
  • *Foundations in Databases*. Abiteboul, Hull, Vianu.
  • *Database Management Systems*, Ramakrishnan and Gehkre. Chapter on “Deductive Databases”.

• Acknowledgements:
  • Jeff Ullman’s CIS 145 class lecture slides.
  • Raghu Ramakrishnan and Johannes Gehrke’s lecture slides for Database Management Systems textbook.
Outline of Tutorial

June 14, 2011: The Second Coming of Datalog!

• Refresher: Datalog 101
• Application #1: Data Integration and Exchange
• Application #2: Program Analysis
• Application #3: Declarative Networking
• Conclusions
Datalog for Data Integration

• Motivation and problem setting

• Two basic approaches:
  – virtual data integration
  – materialized data exchange

• Schema mappings and Datalog with Skolem functions
The Data Integration Problem

• Have a collection of related data sources with
  – different schemas
  – different data models (relational, XML, plain text, ...)
  – different attribute domains
  – different capabilities / availability

• Need to cobble them together and provide a uniform interface

• Want to keep track of what came from where

• Focus here: solving problem of different schemas (schema heterogeneity) for relational data
Mediator-Based Data Integration

Basic idea: use a **global mediated schema** to provide a uniform query interface for the heterogeneous data sources.
Mediator-Based Virtual Data Integration

- Local data sources
- Global mediated schema
- Declarative schema mappings
- Source schemas
- Local data sources
Mediator-Based Virtual Data Integration

Query over global schema

Global *mediated* schema

Declarative schema mappings

Source schemas

Local data sources
Mediator-Based Virtual Data Integration

Query over global schema

Global mediated schema

Reformulated query over local schemas

Declarative schema mappings

Source schemas

Local data sources
Mediator-Based Virtual Data Integration

Local data sources

Global mediated schema

Declarative schema mappings

Reformulated query over local schemas

Query results

Query over global schema

Source schemas

Local data sources
Mediator-Based Virtual Data Integration

Query over global schema
Integrated query results

Global mediated schema

Reformulated query over local schemas

Query results
Declarative schema mappings

Source schemas
Local data sources
Mediator-Based Virtual Data Integration

Query over global schema → Integrated query results

Query results → Reformulated query over local schemas

Global mediated schema

Declarative schema mappings

Source schemas

Local data sources

Query may be recursive
Mediator-Based Virtual Data Integration

- **Query** may be recursive

- Reformulation may be (necessarily) recursive

- Query results

- Declarative schema mappings

- Integrated query results

- Query over global schema

- Global mediated schema

- Source schemas

- Local data sources
Materialized Data Exchange

Declarative schema mappings

Global mediated schema (aka target schema)

Declarative schema mappings

Source schema(s)

Local data source(s)
Materialized Data Exchange

- Source schema(s)
- Local data source(s)
- Global mediated schema (aka target schema)
- Declarative schema mappings
- Mappings may be recursive

Diagram:
- Red cylinder: Source schema(s)
- Green cylinder: Local data source(s)
- Purple rectangle: Global mediated schema (aka target schema)
- Blue arrows: Declarative schema mappings
Materialized Data Exchange

- Global mediated schema (aka **target** schema)
- Declarative schema mappings
- Source schema(s)
- Local data source(s)
Materialized Data Exchange

- Source schema(s)
- Local data source(s)
- Global mediated schema (aka target schema)
- Declarative schema mappings
- Data exchange step (construct mediated DB)
Materialized Data Exchange

Declarative schema mappings

Global mediated schema (aka target schema)

Data exchange step (construct mediated DB)

Source schema(s)

Local data source(s)
Materialized Data Exchange

- Local data source(s)
- Source schema(s)
- Global mediated schema (aka target schema)
- Data exchange step (construct mediated DB)
- Declarative schema mappings
- Declarative schema mappings
- Local data source(s)
Materialized Data Exchange

Declarative schema mappings

Global mediated schema
(aka target schema)

Data exchange step
(construct mediated DB)

Source schema(s)

Local data source(s)
Materialized Data Exchange

Materialized mediated (target) database

Data exchange step (construct mediated DB)

Global mediated schema (aka target schema)

Declarative schema mappings

Source schema(s)

Local data source(s)
Materialized Data Exchange

Materialized mediated (target) database

Global mediated schema (aka target schema)

Declarative schema mappings

Source schema(s)

Local data source(s)
Materialized Data Exchange

Materialized mediated (target) database

Query

Declarative schema mappings

Global mediated schema (aka target schema)

Declarative schema mappings

Source schema(s)

Local data source(s)
Materialized Data Exchange

Materialized mediated (target) database

Source schema(s)

Local data source(s)

Query

Global mediated schema (aka target schema)

Declarative schema mappings

Declarative schema mappings
Materialized Data Exchange

Materialized mediated (target) database

Source schema(s)

Local data source(s)

Global mediated schema (aka target schema)

Declarative schema mappings

Query results

Query

Declarative schema mappings
Peer-to-Peer Data Integration (Virtual or Materialized)
Peer-to-Peer Data Integration (Virtual or Materialized)

Recursion arises naturally as peers add mappings to each other.
Peer-to-Peer Data Integration (Virtual or Materialized)
Peer-to-Peer Data Integration
(Virtual or Materialized)
Peer-to-Peer Data Integration (Virtual or Materialized)
Peer-to-Peer Data Integration
(Virtual or Materialized)
Peer-to-Peer Data Integration (Virtual or Materialized)
Peer-to-Peer Data Integration
(Virtual or Materialized)
Peer-to-Peer Data Integration (Virtual or Materialized)
Peer-to-Peer Data Integration (Virtual or Materialized)
Peer-to-Peer Data Integration (Virtual or Materialized)
Peer-to-Peer Data Integration (Virtual or Materialized)
How to Specify Mappings?

• Many flavors of mapping specifications: LAV, GAV, GLAV, P2P, “sound” versus “exact”, ...

• Unifying formalism: *integrity constraints*
  – different flavors of specifications correspond to different classes of integrity constraints

• We focus on mappings specified using *tuple-generating dependencies* (a kind of integrity constraint)

• These capture (sound) LAV and GAV as special cases, and much of GLAV and P2P as well
  – and, close relationship with Datalog!
Logical Schema Mappings via Tuple-Generating Dependencies (tgds)

• A tuple-generating dependency (tgd) is a first-order constraint of the form

\[ \forall X \, \phi(X) \rightarrow \exists Y \, \psi(X,Y) \]

where \( \phi \) and \( \psi \) are conjunctions of relational atoms
Logical Schema Mappings via Tuple-Generating Dependencies (tgds)

- A **tuple-generating dependency** (tgd) is a first-order constraint of the form

\[
\forall X \phi(X) \rightarrow \exists Y \psi(X,Y)
\]

where \( \phi \) and \( \psi \) are **conjunctions** of relational atoms.

For example:

\[
\forall \text{Eid, Name, Addr} \ \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \ \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr})
\]

“The name and address of every **employee** should also be recorded in the **name** and **address** tables, indexed by ssn.”
What Answers Should Queries Return?

- **Challenge**: constraints leave problem “under-defined”: for given local source instance, many possible mediated instances may satisfy the constraints.
What Answers Should Queries Return?

• **Challenge**: constraints leave problem “under-defined”: for given local source instance, many possible mediated instances may satisfy the constraints.

CONSTRANINT:

\[\forall \text{Eid}, \text{Name}, \text{Addr} \ \text{employee}(\text{Eid}, \text{Name}, \text{Addr}) \rightarrow \exists \text{Ssn} \ \text{name}(\text{Ssn}, \text{Name}) \land \text{address}(\text{Ssn}, \text{Addr})\]
What Answers Should Queries Return?

• **Challenge**: constraints leave problem “under-defined”: for given local source instance, many possible mediated instances may satisfy the constraints.

CONSTRANT: \[
\forall \text{Eid, Name, Addr} \exists \text{Ssn} \quad \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \quad \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr})
\]

LOCAL SOURCE

<table>
<thead>
<tr>
<th>Eid</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>23</td>
<td>Bob</td>
<td>16 Elm St</td>
</tr>
</tbody>
</table>
What Answers Should Queries Return?

- **Challenge**: constraints leave problem “under-defined”: for given local source instance, many possible mediated instances may satisfy the constraints.

**CONSTRAINT:**
\[
\forall \text{Eid, Name, Addr} \quad \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \quad \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr})
\]

**LOCAL SOURCE**

<table>
<thead>
<tr>
<th>employee</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17 Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>23 Bob</td>
<td>16 Elm St</td>
</tr>
</tbody>
</table>

**MEDIATED DB #1**

<table>
<thead>
<tr>
<th>name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>050-66 Alice</td>
<td></td>
</tr>
<tr>
<td>010-12 Bob</td>
<td></td>
</tr>
<tr>
<td>040-66 Carol</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>050-66 1 Main St</td>
<td></td>
</tr>
<tr>
<td>010-12 16 Elm St</td>
<td></td>
</tr>
<tr>
<td>040-66 7 11th Ave</td>
<td></td>
</tr>
</tbody>
</table>
What Answers Should Queries Return?

- **Challenge**: constraints leave problem “under-defined”: for given local source instance, many possible mediated instances may satisfy the constraints.

CONSTRANT:

\[
\forall \text{Eid, Name, Addr} \ \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \ \text{name}(\text{Ssn, Name}) \land \text{address} (\text{Ssn, Addr})
\]

<table>
<thead>
<tr>
<th>LOCAL SOURCE</th>
<th>MEDIATED DB #1</th>
<th>MEDIATED DB #2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>employee</strong></td>
<td><strong>name</strong></td>
<td><strong>name</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 Alice</td>
<td>050-66 Alice</td>
<td>27 Alice</td>
</tr>
<tr>
<td>23 Bob</td>
<td>010-12 Bob</td>
<td>42 Bob</td>
</tr>
<tr>
<td></td>
<td>040-66 Carol</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>address</strong></td>
<td><strong>address</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>050-66 1 Main St</td>
<td>27 1 Main St</td>
</tr>
<tr>
<td></td>
<td>010-12 16 Elm St</td>
<td>42 16 Elm St</td>
</tr>
<tr>
<td></td>
<td>040-66 7 11th Ave</td>
<td></td>
</tr>
</tbody>
</table>
What Answers Should Queries Return?

- **Challenge**: constraints leave problem “under-defined”: for given local source instance, many possible mediated instances may satisfy the constraints.

CONSTRANT:

\[
\forall \, \text{Eid, Name, Addr} \quad \text{employee(}\text{Eid, Name, Addr}\text{)} \rightarrow \\
\exists \, \text{Ssn} \quad \text{name(}\text{Ssn, Name}\text{)} \land \text{address(}\text{Ssn, Addr}\text{)}
\]

| LOCAL SOURCE | MEDIATED DB #1 | MEDIATED DB #2 | ...ETC...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>employee</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 Alice 1 Main St</td>
<td>name</td>
<td></td>
<td>name</td>
</tr>
<tr>
<td>23 Bob 16 Elm St</td>
<td>050-66 Alice</td>
<td></td>
<td>27 Alice</td>
</tr>
<tr>
<td></td>
<td>010-12 Bob</td>
<td></td>
<td>42 Bob</td>
</tr>
<tr>
<td></td>
<td>040-66 Carol</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td><strong>address</strong></td>
<td></td>
<td><strong>address</strong></td>
</tr>
<tr>
<td></td>
<td>050-66 1 Main St</td>
<td></td>
<td>27 1 Main St</td>
</tr>
<tr>
<td></td>
<td>010-12 16 Elm St</td>
<td></td>
<td>42 16 Elm St</td>
</tr>
<tr>
<td></td>
<td>040-66 7 11th Ave</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
What Answers Should Queries Return?

- **Challenge**: constraints leave problem “under-defined”: for given local source instance, many possible mediated instances may satisfy the constraints.

CONSTRANT:

\[
\forall \text{Eid}, \text{Name}, \text{Addr} \quad \text{employee}(\text{Eid}, \text{Name}, \text{Addr}) \rightarrow \\
\exists \text{Ssn} \quad \text{name}(\text{Ssn}, \text{Name}) \land \text{address}(\text{Ssn}, \text{Addr})
\]

| LOCAL SOURCE | MEDIATED DB #1 | MEDIATED DB #2 | ...ETC...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>employee</strong></td>
<td><strong>name</strong></td>
<td><strong>name</strong></td>
<td></td>
</tr>
<tr>
<td>17 Alice 1 Main St</td>
<td>050-66 Alice</td>
<td>27 Alice</td>
<td></td>
</tr>
<tr>
<td>010-12 Bob 16 Elm St</td>
<td>010-12 Bob</td>
<td>42 Bob</td>
<td></td>
</tr>
<tr>
<td>040-66 Carol</td>
<td>040-66 Carol</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td><strong>address</strong></td>
<td><strong>address</strong></td>
<td><strong>address</strong></td>
<td></td>
</tr>
<tr>
<td>050-66 1 Main St</td>
<td>050-66 1 Main St</td>
<td>27 1 Main St</td>
<td></td>
</tr>
<tr>
<td>010-12 16 Elm St</td>
<td>010-12 16 Elm St</td>
<td>42 16 Elm St</td>
<td></td>
</tr>
<tr>
<td>040-66 7 11th Ave</td>
<td>040-66 7 11th Ave</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Which mediated DB should be materialized?
What Answers Should Queries Return?

- **Challenge**: constraints leave problem “under-defined”: for given local source instance, many possible mediated instances may satisfy the constraints.

CONSTRANT:

\[
\forall \text{Eid, Name, Addr} \quad \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \quad \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr})
\]

LOCAL SOURCE

<table>
<thead>
<tr>
<th>employee</th>
<th>MEDIATED DB #1</th>
<th>MEDIATED DB #2</th>
<th>...ETC...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>17 Alice</td>
<td>050-66 Alice</td>
<td>27 Alice</td>
<td></td>
</tr>
<tr>
<td>1 Main St</td>
<td>010-12 Bob</td>
<td>42 Bob</td>
<td></td>
</tr>
<tr>
<td>23 Bob</td>
<td>040-66 Carol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 Elm St</td>
<td>050-66 1 Main St</td>
<td>27 1 Main St</td>
<td></td>
</tr>
<tr>
<td></td>
<td>010-12 16 Elm St</td>
<td>42 16 Elm St</td>
<td></td>
</tr>
<tr>
<td></td>
<td>040-66 7 11th Ave</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**QUERY:**

\[ q(\text{Name}) \leftarrow \text{name}(\text{Ssn, Name}), \text{address}(\text{Ssn, _}). \]
What Answers Should Queries Return?

- **Challenge**: constraints leave problem “under-defined”: for given local source instance, many possible mediated instances may satisfy the constraints.

CONSTRAINT:

\[
\forall E_{id}, N_{ame}, A_{ddr}\ \text{employee} (E_{id}, N_{ame}, A_{ddr}) \rightarrow \\
\exists S_{sn}\ \text{name} (S_{sn}, N_{ame}) \land \text{address} (S_{sn}, A_{ddr})
\]

**LOCAL SOURCE**

<table>
<thead>
<tr>
<th>employee</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17 Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>23 Bob</td>
<td>16 Elm St</td>
</tr>
</tbody>
</table>

**MEDIATED DB #1**

<table>
<thead>
<tr>
<th>name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>050-66 Alice</td>
<td></td>
</tr>
<tr>
<td>010-12 Bob</td>
<td></td>
</tr>
<tr>
<td>040-66 Carol</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>050-66 1 Main St</td>
<td></td>
</tr>
<tr>
<td>010-12 16 Elm St</td>
<td></td>
</tr>
<tr>
<td>040-66 7 11th Ave</td>
<td></td>
</tr>
</tbody>
</table>

**MEDIATED DB #2**

<table>
<thead>
<tr>
<th>name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>27 Alice</td>
<td></td>
</tr>
<tr>
<td>42 Bob</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>27 1 Main St</td>
<td></td>
</tr>
<tr>
<td>42 16 Elm St</td>
<td></td>
</tr>
</tbody>
</table>

**...ETC...**

**QUERY:**

\[ q(N_{ame}) \leftarrow \text{name} (S_{sn}, N_{ame}), \text{address} (S_{sn}, \_). \]

**What** answers should \( q \) return?

**Which** mediated DB should be materialized?
Certain Answers Semantics

**Basic idea**: query should return those answers that would be present for **any** mediated DB instance (satisfying the constraints).
Certain Answers Semantics

**Basic idea:** query should return those answers that would be present for *any* mediated DB instance (satisfying the constraints).

| LOCAL SOURCE | MEDIATED DB #1 | MEDIATED DB #2 | ...ETC...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>employee</strong></td>
<td>name</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>17 Alice</td>
<td>050-66 Alice</td>
<td>27 Alice</td>
<td></td>
</tr>
<tr>
<td>23 Bob</td>
<td>010-12 Bob</td>
<td>42 Bob</td>
<td></td>
</tr>
<tr>
<td></td>
<td>040-66 Carol</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>address</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>050-66 1 Main St</td>
<td>27 1 Main St</td>
<td></td>
</tr>
<tr>
<td></td>
<td>010-12 16 Elm St</td>
<td>42 16 Elm St</td>
<td></td>
</tr>
<tr>
<td></td>
<td>040-66 7 11th Ave</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Certain Answers Semantics**

**Basic idea:** query should return those answers that would be present for any mediated DB instance (satisfying the constraints).

| LOCAL SOURCE | MEDIATED DB #1 | MEDIATED DB #2 | ...ETC...
|--------------|----------------|----------------|----------------
| **employee** | **name**       | **name**       | **address**    |
| 17 Alice 1 Main St | 050-66 Alice | 27 Alice | 050-66 1 Main St |
| 23 Bob 16 Elm St  | 010-12 Bob   | 42 Bob       | 010-12 16 Elm St |
|               | 040-66 Carol  |             | 040-66 7 11th Ave |

**QUERY:**

\[
q(\text{Name}) \leftarrow \\
\text{name}(\text{Ssn, Name}), \\
\text{address}(\text{Ssn, _}).
\]
Certain Answers Semantics

Basic idea: query should return those answers that would be present for any mediated DB instance (satisfying the constraints).

| LOCAL SOURCE | MEDIATED DB #1 | MEDIATED DB #2 | ...ETC...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>employee</td>
<td>name</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>17 Alice 1 Main St</td>
<td>050-66 Alice</td>
<td>27 Alice</td>
<td></td>
</tr>
<tr>
<td>23 Bob 16 Elm St</td>
<td>010-12 Bob</td>
<td>42 Bob</td>
<td></td>
</tr>
<tr>
<td></td>
<td>040-66 Carol</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>050-66 1 Main St</td>
<td>27 1 Main St</td>
<td></td>
</tr>
<tr>
<td></td>
<td>010-12 16 Elm St</td>
<td>42 16 Elm St</td>
<td></td>
</tr>
<tr>
<td></td>
<td>040-66 7 11th Ave</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

QUERY:

\[ q(\text{Name}) \leftarrow \text{name}(\text{Ssn, Name}), \text{address}(\text{Ssn, _}). \]
Certain Answers Semantics

**Basic idea**: query should return those answers that would be present for **any** mediated DB instance (satisfying the constraints).

| LOCAL SOURCE employee | MEDIATED DB #1 name | MEDIATED DB #2 name | MEDIATED DB #2 address | ...ETC...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17 Alice 1 Main St</td>
<td>050-66 Alice</td>
<td>27 Alice</td>
<td>27 1 Main St</td>
<td>...</td>
</tr>
<tr>
<td>23 Bob 16 Elm St</td>
<td>010-12 Bob</td>
<td>42 Bob</td>
<td>42 16 Elm St</td>
<td></td>
</tr>
</tbody>
</table>

**QUERY:**

\[
\text{q}(\text{Name}) \leftarrow \text{name}(\text{Ssn}, \text{Name}), \text{address}(\text{Ssn}, \_).
\]
Certain Answers Semantics

**Basic idea:** query should return those answers that would be present for any mediated DB instance (satisfying the constraints).

| LOCAL SOURCE | MEDIATED DB #1 | MEDIATED DB #2 | ...ETC...
|---------------|----------------|----------------|----------------
| **employee**  | **name**       | **name**       |                |
| 17 Alice 1 Main St | 050-66 Alice | 27 Alice       |                |
| 23 Bob 16 Elm St   | 010-12 Bob    | 42 Bob         | ...            |
|                | 040-66 Carol  |                |                |
| **address**     | **address**   |                |                |
| 050-66 1 Main St | 27 1 Main St  |                |                |
| 010-12 16 Elm St | 42 16 Elm St  |                |                |
| 040-66 7 11th Ave|                |                |                |

**QUERY:**

\[ q(\text{Name}) \leftarrow \text{name}(\text{Ssn, Name}), \text{address}(\text{Ssn, \_}) \]

```sql
q
  Alice
  Bob
  Carol
```

```sql
q
  Alice
  Bob
```

```sql
...etc...
```
**Certain Answers Semantics**

**Basic idea:** query should return those answers that would be present for any mediated DB instance (satisfying the constraints).

| LOCAL SOURCE | MEDIATED DB #1 | MEDIATED DB #2 | ...ETC...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>employee</strong></td>
<td><strong>name</strong></td>
<td><strong>name</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>050-66 Alice</td>
<td>27 Alice</td>
<td></td>
</tr>
<tr>
<td>17 Alice</td>
<td>010-12 Bob</td>
<td>042 Bob</td>
<td></td>
</tr>
<tr>
<td>23 Bob</td>
<td>040-66 Carol</td>
<td>42 Bob</td>
<td></td>
</tr>
</tbody>
</table>

|           | **address**    |                   |
|           | 050-66 1 Main St | 27 1 Main St |
|           | 010-12 16 Elm St | 42 16 Elm St |
|           | 040-66 7 11\textsuperscript{th} Ave | |

**QUERY:**

\[ q(\text{Name}) \leftarrow \text{name}(\text{Ssn, Name}), \text{address}(\text{Ssn, _}). \]
Certain Answers Semantics

**Basic idea:** query should return those answers that would be present for any mediated DB instance (satisfying the constraints).

| LOCAL SOURCE | MEDIATED DB #1 | MEDIATED DB #2 | ...ETC...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>employee</strong></td>
<td><strong>name</strong></td>
<td><strong>name</strong></td>
<td></td>
</tr>
<tr>
<td>17 Alice 1 Main St</td>
<td>050-66 Alice</td>
<td>27 Alice</td>
<td></td>
</tr>
<tr>
<td>23 Bob 16 Elm St</td>
<td>010-12 Bob</td>
<td>42 Bob</td>
<td></td>
</tr>
<tr>
<td><strong>address</strong></td>
<td><strong>address</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>050-66 1 Main St</td>
<td>010-12 16 Elm St</td>
<td>27 1 Main St</td>
<td>...</td>
</tr>
<tr>
<td>040-66 7 11th Ave</td>
<td></td>
<td>42 16 Elm St</td>
<td>...</td>
</tr>
</tbody>
</table>

**QUERY:**

$q(\text{Name}) \leftarrow \text{name}(\text{Ssn}, \text{Name}), \text{address}(\text{Ssn}, \_).$
Certain Answers Semantics

**Basic idea:** query should return those answers that would be present for **any** mediated DB instance (satisfying the constraints).

| LOCAL SOURCE | MEDIATED DB #1 | MEDIATED DB #2 | ...ETC...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>employee</strong></td>
<td><strong>name</strong></td>
<td><strong>name</strong></td>
<td></td>
</tr>
<tr>
<td>17 Alice</td>
<td>050-66 Alice</td>
<td>27 Alice</td>
<td></td>
</tr>
<tr>
<td>23 Bob</td>
<td>010-12 Bob</td>
<td>040-66 Bob</td>
<td></td>
</tr>
<tr>
<td></td>
<td>040-66 Carol</td>
<td>42 Bob</td>
<td></td>
</tr>
</tbody>
</table>

**QUERY:**

\[ q(\text{Name}) \leftarrow \text{name}(\text{Ssn, Name}), \text{address}(\text{Ssn, _}). \]

**certain answers to q**

\[ \{ \text{Alice}, \text{Bob}, \text{Carol} \} = \bigcap \text{q} \]

\[ \bigcap \text{q} \]

\[ \ldots \]
Computing the Certain Answers

• A number of methods have been developed
  – Bucket algorithm [Levy+ 1996]
  – Minicon [Pottinger & Halevy 2000]
  – Inverse rules method [Duschka & Genesereth 1997]
  – ...

• We focus on the Datalog-based inverse rules method

• Same method works for both virtual data integration, and materialized data exchange
  – Assuming constraints are given by tgds
Inverse Rules: Computing Certain Answers with Datalog

- Basic idea: a tgd looks a lot like a Datalog rule (or rules)

\[
\forall X, Y, Z \text{ foo}(X,Y) \land \text{ bar}(X,Z) \rightarrow \text{ biz}(Y,Z) \land \text{ baz}(Z)
\]

Datalog rules:

\[
\text{biz}(X,Y,Z) \leftarrow \text{ foo}(X,Y), \text{ bar}(X,Z).
\]

\[
\text{baz}(Z) \leftarrow \text{ foo}(X,Y), \text{ bar}(X,Z).
\]
Inverse Rules: Computing Certain Answers with Datalog

• Basic idea: a tgd looks a lot like a Datalog rule (or rules)

\[
\forall X, Y, Z \ foo(X,Y) \land bar(X,Z) \rightarrow biz(Y,Z) \land baz(Z)
\]

tgd: 

Datalog rules:

\[
biz(X,Y,Z) \leftarrow foo(X,Y), \ bar(X,Z).
\]

\[
baz(Z) \leftarrow foo(X,Y), \ bar(X,Z).
\]

• So just interpret tgds as Datalog rules! (“Inverse” rules.) Can use these to compute the certain answers.
Inverse Rules: Computing Certain Answers with Datalog

• Basic idea: a tgd looks a lot like a Datalog rule (or rules)

\[
\forall X, Y, Z \ foo(X,Y) \land bar(X,Z) \rightarrow biz(Y,Z) \land baz(Z)
\]

Datalog rules:

\[
biz(X,Y,Z) \leftarrow foo(X,Y), \ bar(X,Z).
\]

\[
baz(Z) \leftarrow foo(X,Y), \ bar(X,Z).
\]

• So just interpret tgds as Datalog rules! (“Inverse” rules.) Can use these to compute the certain answers.

  – Why called “inverse” rules? In work on LAV data integration, constraints written in the other direction, with sources thought of as views over the (hypothetical) mediated database instance
Inverse Rules: Computing Certain Answers with Datalog

- Basic idea: a tgd looks a lot like a Datalog rule (or rules)

\[
\forall X, Y, Z \ foo(X,Y) \land \ bar(X,Z) \rightarrow biz(Y,Z) \land baz(Z)
\]

Datalog rules:

\[
\begin{align*}
biz(X,Y,Z) & \leftarrow foo(X,Y), \ bar(X,Z). \\
baz(Z) & \leftarrow foo(X,Y), \ bar(X,Z). 
\end{align*}
\]

- So just interpret tgds as Datalog rules! (“Inverse” rules.) Can use these to compute the certain answers.

  - Why called “inverse” rules? In work on LAV data integration, constraints written in the other direction, with sources thought of as views over the (hypothetical) mediated database instance

The catch: what to do about \textbf{existentially quantified variables}...
Inverse Rules: Computing Certain Answers with Datalog (2)

- Challenge: **existentially quantified variables** in tgd's

\[
\forall \text{Eid, Name, Addr} \quad \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \quad \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr})
\]
Inverse Rules: Computing Certain Answers with Datalog (2)

• Challenge: **existentially quantified variables** in tgd$s

\[ \forall \text{Eid, Name, Addr} \quad \text{employee}(\text{Eid, Name, Addr}) \rightarrow \]
\[ \exists \text{Ssn} \quad \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr}) \]

• Key idea: use **Skolem functions**
  – think: “memoized value invention” (or “labeled nulls”)
Inverse Rules: Computing Certain Answers with Datalog (2)

• Challenge: **existentially quantified variables** in tgds

\[
\forall \text{Eid, Name, Addr} \quad \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \quad \text{name}(\text{Ssn}, \text{Name}) \land \text{address}(\text{Ssn}, \text{Addr})
\]

• Key idea: use **Skolem functions**
  – think: “memoized value invention” (or “labeled nulls”)

\[
\text{name}(\text{ssn}(\text{Name, Addr}), \text{Name}) \leftarrow \text{employee}(\_, \text{Name, Addr}). \\
\text{address}(\text{ssn}(\text{Name, Addr}), \text{Addr}) \leftarrow \text{employee}(\_, \text{Name, Addr}).
\]
Inverse Rules: Computing Certain Answers with Datalog (2)

• Challenge: **existentially quantified variables** in tgd

\[ \forall \text{Eid, Name, Addr} \quad \text{employee}(\text{Eid, Name, Addr}) \rightarrow \]
\[ \exists \text{Ssn} \quad \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr}) \]

• Key idea: use **Skolem functions**
  – think: “memoized value invention” (or “labeled nulls”)

```prolog
name(\text{ssn}(\text{Name, Addr}), \text{Name}) \leftarrow \text{employee}(\_ , \text{Name, Addr}).
address(\text{ssn}(\text{Name, Addr}), \text{Addr}) \leftarrow \text{employee}(\_ , \text{Name, Addr}).
```

ssn is a Skolem function
Inverse Rules: Computing Certain Answers with Datalog (2)

- Challenge: **existentially quantified variables** in tgd

\[
\forall \text{Eid, Name, Addr} \text{ } \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn } \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr})
\]

- Key idea: use **Skolem functions**
  - think: “memoized value invention” (or “labeled nulls”)

\[
\text{name}(\text{ssn}(\text{Name, Addr}), \text{Name}) <\text{employee}(\_, \text{Name, Addr}) . \\
\text{address}(\text{ssn}(\text{Name, Addr}), \text{Addr}) <\text{employee}(\_, \text{Name, Addr}) .
\]
Inverse Rules: Computing Certain Answers with Datalog (2)

- **Challenge:** existentially quantified variables in tgd

  $\forall$ Eid, Name, Addr  $\text{employee}(Eid, Name, Addr) \rightarrow$
  
  $\exists$ Ssn  $\text{name}(Ssn, Name) \land \text{address}(Ssn, Addr)$

- **Key idea:** use Skolem functions
  
  - think: “memoized value invention” (or “labeled nulls”)

  $\text{name}(\text{ssn}(Name, Addr), Name) \leftarrow \text{employee}(\_, Name, Addr)$.
  
  $\text{address}(\text{ssn}(Name, Addr), Addr) \leftarrow \text{employee}(\_, Name, Addr)$.

- Unlike SQL nulls, can join on Skolem values:
Inverse Rules: Computing Certain Answers with Datalog (2)

- Challenge: **existentially quantified variables** in tgds

\[
\forall \text{Eid, Name, Addr} \quad \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \quad \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr})
\]

- Key idea: use **Skolem functions**
  - think: “memoized value invention” (or “labeled nulls”)

\[
\text{name}(\text{ssn}(\text{Name, Addr}), \text{Name}) \leftarrow \text{employee}(\_, \text{Name, Addr}). \\
\text{address}(\text{ssn}(\text{Name, Addr}), \text{Addr}) \leftarrow \text{employee}(\_, \text{Name, Addr}).
\]

- Unlike SQL nulls, can join on Skolem values:

\[
\text{query } \_{(\text{Name, Addr}) \leftarrow \\
\text{name}(\text{Ssn, Name}), \\
\text{address}(\text{Ssn, Addr})}.
\]
Semantics of Skolem Functions in Datalog
Semantics of Skolem Functions in Datalog

• Skolem functions interpreted “as themselves,” like constants (Herbrand interpretations): not to be confused with user-defined functions
  
  – e.g., can think of interpretation of term
    
    \[
    \text{ssn}(\text{“Alice”}, \text{“1 Main St”})
    \]

    as just the string (or null labeled by the string)
    
    \[
    \text{ssn}(\text{“Alice”}, \text{“1 Main St”})
    \]
Semantics of Skolem Functions in Datalog

• Skolem functions interpreted “as themselves,” like constants (Herbrand interpretations): not to be confused with user-defined functions
  – e.g., can think of interpretation of term
    \[
    \text{ssn(“Alice”, “1 Main St”)}
    \]
    as just the string (or null labeled by the string)
    \[
    \text{ssn(“Alice”, “1 Main St”)}
    \]

• Datalog programs with Skolem functions continue to have minimal models, which can be computed via, e.g., bottom-up seminaive evaluation
  – Can show that the certain answers are precisely the query answers that contain no Skolem terms. (We’ll revisit this shortly...)
Semantics of Skolem Functions in Datalog

• Skolem functions interpreted “as themselves,” like constants (Herbrand interpretations): not to be confused with user-defined functions
  
  – e.g., can think of interpretation of term
    
    \text{ssn(“Alice”, “1 Main St”)}

    as just the string (or null labeled by the string)

    \text{ssn(“Alice”, “1 Main St”)}

• Datalog programs with Skolem functions continue to have minimal models, which can be computed via, e.g., bottom-up seminaive evaluation
  
  – Can show that the \textbf{certain answers} are precisely the query answers that contain no Skolem terms. (We’ll revisit this shortly...)

• But: the models may now be \textbf{infinite}!
Termination and Infinite Models

• **Problem:** Skolem terms “invent” new values, which might be fed back in a loop to “invent” more new values, ad infinitum
Termination and Infinite Models

- **Problem**: Skolem terms “invent” new values, which might be fed back in a loop to “invent” more new values, ad infinitum—e.g., “every manager has a manager”

```prolog
manager(X) <-
    employee(_, X, _).
manager(m(X)) <-
    manager(X).
```
Termination and Infinite Models

• **Problem:** Skolem terms “invent” new values, which might be fed back in a loop to “invent” more new values, ad infinitum — e.g., “every manager has a manager”

```
manager(X) <-
    employee(_, X, _).
manager(m(X)) <-
    manager(X).
```

*m* is a Skolem function
Termination and Infinite Models

- **Problem**: Skolem terms “invent” new values, which might be fed back in a loop to “invent” more new values, ad infinitum – e.g., “every manager has a manager”

```
manager(X) <-
    employee(_, X, _).
manager(m(X)) <-
    manager(X).
```

<table>
<thead>
<tr>
<th>employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>17  Alice  1 Main St</td>
</tr>
<tr>
<td>23  Bob    16 Elm St</td>
</tr>
</tbody>
</table>
Termination and Infinite Models

- **Problem**: Skolem terms “invent” new values, which might be fed back in a loop to “invent” more new values, ad infinitum—e.g., “every manager has a manager”

```prolog
manager(X) :-
    employee(_, X, _).
manager(m(X)) :-
    manager(X).
```

<table>
<thead>
<tr>
<th>manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>m(Alice)</td>
</tr>
<tr>
<td>m(Bob)</td>
</tr>
<tr>
<td>m(m(Alice))</td>
</tr>
<tr>
<td>m(m(Bob))</td>
</tr>
<tr>
<td>m(m(m(Alice)))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 Alice 1 Main St</td>
</tr>
<tr>
<td>23 Bob 16 Elm St</td>
</tr>
</tbody>
</table>

...
Termination and Infinite Models

- **Problem**: Skolem terms “invent” new values, which might be fed back in a loop to “invent” more new values, ad infinitum
  - e.g., “every manager has a manager”

```
manager(X) <-
  employee(_, X, _).
manager(m(X)) <-
  manager(X).
```

```
<table>
<thead>
<tr>
<th>employee</th>
<th>manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 Alice</td>
<td>m(Alice)</td>
</tr>
<tr>
<td>23 Bob</td>
<td>m(Bob)</td>
</tr>
<tr>
<td>m(m(Alice))</td>
<td>m(m(Alice))</td>
</tr>
<tr>
<td>m(m(Bob))</td>
<td>m(m(Bob))</td>
</tr>
</tbody>
</table>
```

- Option 1: let ‘er rip and see what happens! (Coral, LB)
Termination and Infinite Models

- **Problem**: Skolem terms “invent” new values, which might be fed back in a loop to “invent” more new values, ad infinitum – e.g., “every manager has a manager”

- Option 1: let ‘er rip and see what happens! (Coral, LB)
- Option 2: use syntactic restrictions to ensure termination...

```
manager(X) <-
  employee(_, X, _).
manager(m(X)) <-
  manager(X).
```

```
manager
m(Alice)
m(Bob)
m(m(Alice))
m(m(Bob))
m(m(m(Alice)))
...
```

```
employee

<table>
<thead>
<tr>
<th>17</th>
<th>Alice</th>
<th>1 Main St</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Bob</td>
<td>16 Elm St</td>
</tr>
</tbody>
</table>
```

145
Ensuring Termination of Datalog Programs with Skolems via Weak Acyclicity

• Draw graph for Datalog program as follows:

```
manager(X) <-
    employee(_, X, _).
manager(m(X)) <-
    manager(X).
```
Ensuring Termination of Datalog Programs with Skolems via Weak Acyclicity

• Draw graph for Datalog program as follows:

\[
\text{manager}(X) \leftarrow \\
\quad \text{employee}(\_, X, \_). \\
\text{manager}(m(X)) \leftarrow \\
\quad \text{manager}(X).
\]

vertex for each (predicate, index)

(employee, 1)  (employee, 2)  (employee, 3)

(manager, 1)
Ensuring Termination of Datalog Programs with Skolems via Weak Acyclicity

• Draw graph for Datalog program as follows:

\[
\text{manager}(X) \leftarrow \text{employee}(\_, X, \_).
\]
\[
\text{manager}(m(X)) \leftarrow \text{manager}(X).
\]

- **vertex for each (predicate, index)**
  - \((\text{employee}, 2)\)
  - \((\text{employee}, 1)\)
  - \((\text{employee}, 3)\)
  - \((\text{manager}, 1)\)

- **variable occurs as arg #2 to employee in body, arg #1 to manager in head**
Ensuring Termination of Datalog Programs with Skolems via Weak Acyclicity

• Draw graph for Datalog program as follows:

```
manager(X) <-
    employee(_, X, _).
manager[m(X)] <-
    manager(X).
```

vertex for each (predicate, index)

(manager, 1)
(employee, 1)
(employee, 2)
(employee, 3)

variable occurs as arg #2 to `employee` in body, arg #1 to `manager` in head

variable occurs as arg #1 to `manager` in body and as argument to Skolem (hence dashes) in arg #1 to `manager` in head
Ensuring Termination of Datalog Programs with Skolems via Weak Acyclicity

- Draw graph for Datalog program as follows:

  \[
  \text{manager}(X) \leftarrow \\
  \quad \text{employee}(\_ , X , \_ ) . \\
  \text{manager}(m(X)) \leftarrow \\
  \quad \text{manager}(X). \\
  \]

- If graph contains no cycle through a dashed edge, then P is called **weakly acyclic**
Ensuring Termination of Datalog Programs with Skolems via Weak Acyclicity

• Draw graph for Datalog program as follows:

\[
\begin{align*}
\text{manager}(X) & \leftarrow \text{employee}(\_, X, \_). \\
\text{manager}(m(X)) & \leftarrow \text{manager}(X).
\end{align*}
\]

vertex for each (predicate, index)

Cycle through dashed edge! Not weakly acyclic ☹️

• If graph contains no cycle through a dashed edge, then P is called \textbf{weakly acyclic}
Ensuring Termination via Weak Acyclicity (2)

• Another example, this one weakly acyclic:
Ensuring Termination via Weak Acyclicity (2)

• Another example, this one weakly acyclic:

```prolog
name(ssn(Name,Addr),Name)  <- emp(_,Name,Addr).
addr(ssn(Name,Addr),Addr)  <- emp(_,Name,Addr).
query _(Name,Addr)
   <- name(Ssn,Name),
       address(Ssn,Addr) ;
   _(Addr,Name).
```
Ensuring Termination via Weak Acyclicity (2)

• Another example, this one weakly acyclic:

```prolog
name(ssn(Name,Addr),Name)  <- emp(_,Name,Addr).
addr(ssn(Name,Addr),Addr)  <- emp(_,Name,Addr).
query _(Name,Addr)  <- name(Ssn,Name),
                address(Ssn,Addr) ;
                _(Addr,Name).
```

Actual example:

- (emp, 2)
- (emp, 3)
- (emp, 1)
- (name, 1)
- (addr, 1)
- (name, 2)
- (addr, 2)
- (_, 1)
- (_, 2)
Ensuring Termination via Weak Acyclicity (2)

- Another example, this one weakly acyclic:

```prolog
name(ssn(Name,Addr),Name)  <- emp(_,Name,Addr).
addr(ssn(Name,Addr),Addr)  <- emp(_,Name,Addr).
query _ (Name,Addr)        <- name(Ssn,Name), address(Ssn,Addr) ; _ (Addr,Name).
```

[Diagram of the relationship between entities]
Ensuring Termination via Weak Acyclicity (2)

- Another example, this one weakly acyclic:

```
name(ssn(Name,Addr),Name)  <- emp(_,Name,Addr).
addr(ssn(Name,Addr),Addr)  <- emp(_,Name,Addr).
query _(Name,Addr)         <- name(Ssn,Name)
                           addr(Ssn,Addr)
                           _ (Addr,Name).
```

The graph has a cycle, but no cycle through the dashed edge; weakly acyclic 😊
Ensuring Termination via Weak Acyclicity (2)

- Another example, this one weakly acyclic:

```
name(ssn(Name,Addr),Name)  
  <- emp(_,Name,Addr).
addr(ssn(Name,Addr),Addr)  
  <- emp(_,Name,Addr).
query _ (Name,Addr)  
  <- name(Ssn,Name),
    address(Ssn,Addr),
    _ (Addr,Name).
```

Theorem: bottom-up evaluation of weakly acyclic Datalog programs with Skolems terminates in \# steps polynomial in size of source database.
Once Computation Stops, What Do We Have?
Once Computation Stops, What Do We Have?

**tgd:**
\[
\forall \text{Eid, Name, Addr} \; \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \; \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr})
\]

**datalog rules:**

\[
\text{name}(\text{ssn}(\text{Name, Addr}), \text{Name}) <- \text{employee}(\_ , \text{Name, Addr}). \\
\text{address}(\text{ssn}(\text{Name, Addr}), \text{Addr}) <- \text{employee}(\_ , \text{Name, Addr}).
\]
Once Computation Stops, What Do We Have?

∀ Eid, Name, Addr \texttt{employee}(Eid, Name, Addr) → \\
∃ Ssn \texttt{name}(Ssn, Name) ∧ \texttt{address}(Ssn, Addr)

datalog rules:
\texttt{name}(ssn(\texttt{Name}, \texttt{Addr}), \texttt{Name}) <- \texttt{employee}(\_, \texttt{Name}, \texttt{Addr}). \\
\texttt{address}(ssn(\texttt{Name}, \texttt{Addr}), \texttt{Addr}) <- \texttt{employee}(\_, \texttt{Name}, \texttt{Addr}).

LOCAL SOURCE

\texttt{employee}

<table>
<thead>
<tr>
<th>Eid</th>
<th>Name</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>23</td>
<td>Bob</td>
<td>16 Elm St</td>
</tr>
</tbody>
</table>
Once Computation Stops, What Do We Have?

Let's consider the following logical expressions and data tables:

**Logical expressions:**
\[
\forall \text{Eid, Name, Addr} \land \text{employee}(\text{Eid, Name, Addr}) \rightarrow \\
\exists \text{Ssn} \land \text{name}(\text{Ssn, Name}) \land \text{address}(\text{Ssn, Addr})
\]

**Datalog rules:**
- `name(ssn(Name, Addr), Name) ← employee(_, Name, Addr).`
- `address(ssn(Name, Addr), Addr) ← employee(_, Name, Addr).`

**Local Source (employee table):**
- Eid: 17, Name: Alice, Addr: 1 Main St
- Eid: 23, Name: Bob, Addr: 16 Elm St

**Mediated DB #2 (name and address tables):**
- **Name table:**
  - `ssn(A..)` → Alice
  - `ssn(B..)` → Bob
- **Address table:**
  - `ssn(A..)` → 1 Main St
  - `ssn(B..)` → 16 Elm St
Once Computation Stops, What Do We Have?

∀ Eid, Name, Addr employee(Eid, Name, Addr) →
∃ Ssn name(Ssn, Name) ∧ address(Ssn, Addr)

datalog rules:

name(ssn(Name, Addr), Name) <- employee(_, Name, Addr).
address(ssn(Name, Addr), Addr) <- employee(_, Name, Addr).

<table>
<thead>
<tr>
<th>LOCAL SOURCE</th>
<th>MEDIATED DB #1</th>
<th>MEDIATED DB #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>employee</td>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>17 Alice</td>
<td>050-66 Alice</td>
<td>ssn(A..) Alice</td>
</tr>
<tr>
<td>23 Bob</td>
<td>010-12 Bob</td>
<td>ssn(B..) Bob</td>
</tr>
<tr>
<td></td>
<td>040-66 Carol</td>
<td></td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>address</td>
</tr>
<tr>
<td></td>
<td>050-66 1 Main St</td>
<td>ssn(A..) 1 Main St</td>
</tr>
<tr>
<td></td>
<td>010-12 16 Elm St</td>
<td>ssn(B..) 16 Elm St</td>
</tr>
<tr>
<td></td>
<td>040-66 7 11th Ave</td>
<td></td>
</tr>
</tbody>
</table>
Once Computation Stops, What Do We Have?

∀ Eid, Name, Addr \textbf{employee}(Eid, Name, Addr) \rightarrow
\exists Ssn \textbf{name}(Ssn, Name) \land \textbf{address}(Ssn, Addr)

datalog rules:

\textbf{name}(\texttt{ssn}(\text{Name, Addr}), \text{Name}) \leftarrow \textbf{employee}(_, \text{Name, Addr}).
\textbf{address}(\texttt{ssn}(\text{Name, Addr}), \text{Addr}) \leftarrow \textbf{employee}(_, \text{Name, Addr}).

<table>
<thead>
<tr>
<th>LOCAL SOURCE</th>
<th>MEDITED DB #1</th>
<th>MEDITED DB #2</th>
<th>MEDITED DB #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{employee}</td>
<td>\textbf{name}</td>
<td>\textbf{name}</td>
<td>\textbf{name}</td>
</tr>
<tr>
<td>17 Alice 1 Main St</td>
<td>050-66 Alice</td>
<td>\texttt{ssn(A..)} Alice</td>
<td>27 Alice</td>
</tr>
<tr>
<td>23 Bob 16 Elm St</td>
<td>010-12 Bob</td>
<td>\texttt{ssn(B..)} Bob</td>
<td>42 Bob</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\textbf{addres}</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>050-66 1 Main St</td>
<td></td>
<td></td>
<td>27 1 Main St</td>
</tr>
<tr>
<td>010-12 16 Elm St</td>
<td></td>
<td></td>
<td>42 16 Elm St</td>
</tr>
<tr>
<td>040-66 7 11th Ave</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Once Computation Stops, What Do We Have?

∀ Eid, Name, Addr \( \text{employee} \)(Eid, Name, Addr) → ∃ Ssn \( \text{name} \)(Ssn, Name) ∧ \( \text{address} \)(Ssn, Addr)

datalog rules:
\[
\begin{align*}
\text{name}(\text{ssn}(\text{Name}, \text{Addr}), \text{Name}) & \leftarrow \text{employee}(\_, \text{Name}, \text{Addr}). \\
\text{address}(\text{ssn}(\text{Name}, \text{Addr}), \text{Addr}) & \leftarrow \text{employee}(\_, \text{Name}, \text{Addr}).
\end{align*}
\]

<table>
<thead>
<tr>
<th>LOCAL SOURCE</th>
<th>MEDIATED DB #1</th>
<th>MEDIATED DB #2</th>
<th>MEDIATED DB #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>employee</td>
<td>name</td>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>address</td>
<td>address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Eid</th>
<th>Name</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>23</td>
<td>Bob</td>
<td>16 Elm St</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ssn</th>
<th>Name</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>050-66</td>
<td>Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>010-12</td>
<td>Bob</td>
<td>16 Elm St</td>
</tr>
<tr>
<td>040-66</td>
<td>Carol</td>
<td>7 11th Ave</td>
</tr>
</tbody>
</table>

Among all the mediated DB instances satisfying the constraints (solutions), #2 above is universal: can be homomorphically embedded in any other solution.
Once Computation Stops, What Do We Have?

∀ Eid, Name, Addr \( \text{employee}(Eid, Name, Addr) \rightarrow \exists Ssn \text{name}(Ssn, Name) \land \text{address}(Ssn, Addr) \)

datalog rules:
\[ \text{name}(\text{ssn}(\text{Name}, \text{Addr}), \text{Name}) \leftarrow \text{employee}(\_\_, \text{Name}, \text{Addr}). \]
\[ \text{address}(\text{ssn}(\text{Name}, \text{Addr}), \text{Addr}) \leftarrow \text{employee}(\_\_, \text{Name}, \text{Addr}). \]

LOCAL SOURCE

<table>
<thead>
<tr>
<th>Eid</th>
<th>Name</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>23</td>
<td>Bob</td>
<td>16 Elm St</td>
</tr>
</tbody>
</table>

MEDIATED DB #1

<table>
<thead>
<tr>
<th>Ssn</th>
<th>Name</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>050-66</td>
<td>Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>010-12</td>
<td>Bob</td>
<td>16 Elm St</td>
</tr>
<tr>
<td>040-66</td>
<td>Carol</td>
<td>7 11th Ave</td>
</tr>
</tbody>
</table>

MEDIATED DB #2

<table>
<thead>
<tr>
<th>Ssn</th>
<th>Name</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ssn(A..)}</td>
<td>Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>\text{ssn(B..)}</td>
<td>Bob</td>
<td>16 Elm St</td>
</tr>
</tbody>
</table>

MEDIATED DB #3

<table>
<thead>
<tr>
<th>Ssn</th>
<th>Name</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Alice</td>
<td>1 Main St</td>
</tr>
<tr>
<td>42</td>
<td>Bob</td>
<td>16 Elm St</td>
</tr>
</tbody>
</table>

Among all the mediated DB instances satisfying the constraints (solutions), #2 above is universal: can be homomorphically embedded in any other solution.
Once Computation Stops, What Do We Have?

∀ Eid, Name, Addr \(\text{employee}(Eid, Name, Addr) \rightarrow\)

\(\exists Ssn\) \(\text{name}(Ssn, Name) \land \text{address}(Ssn, Addr)\)

datalog rules:

\(\text{name}(\text{ssn}(\text{Name}, \text{Addr}), \text{Name}) \leftarrow \text{employee}(\_\_, \text{Name}, \text{Addr})\).

\(\text{address}(\text{ssn}(\text{Name}, \text{Addr}), \text{Addr}) \leftarrow \text{employee}(\_\_, \text{Name}, \text{Addr})\).

LOCAL SOURCE

<table>
<thead>
<tr>
<th>employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 Alice 1 Main St</td>
</tr>
<tr>
<td>23 Bob 16 Elm St</td>
</tr>
</tbody>
</table>

MEDIATED DB #1

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>050-66 Alice</td>
</tr>
<tr>
<td>010-12 Bob</td>
</tr>
<tr>
<td>040-66 Carol</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>050-66 1 Main St</td>
</tr>
<tr>
<td>010-12 16 Elm St</td>
</tr>
<tr>
<td>040-66 11^{th} Ave</td>
</tr>
</tbody>
</table>

MEDIATED DB #2

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ssn}(A..) Alice</td>
</tr>
<tr>
<td>\text{ssn}(B..) Bob</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ssn}(A..) 1 Main St</td>
</tr>
<tr>
<td>\text{ssn}(B..) 16 Elm St</td>
</tr>
</tbody>
</table>

MEDIATED DB #3

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>27 Alice</td>
</tr>
<tr>
<td>42 Bob</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>27 1 Main St</td>
</tr>
<tr>
<td>42 16 Elm St</td>
</tr>
</tbody>
</table>

Among all the mediated DB instances satisfying the constraints (solutions), #2 above is universal: can be homomorphically embedded in any other solution.
Universal Solutions Are Just What is Needed to Compute the Certain Answers
Universal Solutions Are Just What is Needed to Compute the Certain Answers

**Theorem:** can compute certain answers to Datalog program $q$ over target/mediated schema by:

1. evaluating $q$ on materialized mediated DB (computed using inverse rules); then
2. crossing out rows containing Skolem terms.
Universal Solutions Are Just What is Needed to Compute the Certain Answers

**Theorem:** can compute certain answers to Datalog program \( q \) over target/mediated schema by:

1. evaluating \( q \) on materialized mediated DB (computed using inverse rules); then
2. crossing out rows containing Skolem terms.

*Proof (crux):* use universality of materialized DB.
Notes on Skolem Functions in Datalog

• Notion of weak acyclicity introduced by Deutsch and Popa, as a way to ensure termination of the chase procedure for logical dependencies (but applies to Datalog too).

• **Crazy idea**: what if we allow arbitrary use of Skolems, and forget about computing complete output idb’s bottom-up, but only partially enumerate their contents, on demand, using **top-down** evaluation?
  
  – And, while we’re at it, allow unsafe rules too?

• This is actually a beautiful idea: it’s called **logic programming**

  – Skolem functions (aka “functor terms”) are how you build data structures like lists, trees, etc. in Prolog

  – Resulting language is Turing-complete
Summary: Datalog for Data Integration and Exchange

• Datalog serves as very nice language for **schema mappings**, as needed in data integration, provided we extend it with Skolem functions
  – Can use Datalog to compute certain answers
  – Fancier kinds of schema mappings than tgds require further language extensions; e.g., Datalog +/- [Cali et al 09]

• Can also extend Datalog to track various kinds of data **provenance**, very useful in data integration
  – Using semiring-based framework [Green+ 07]
Some Datalog-Based Data Integration/Exchange Systems

• Information Manifold [Levy+ 96]
  – Virtual approach
  – No recursion

• Clio [Miller+ 01]
  – Materialized approach
  – Skolem terms, no recursion, rich data model
  – Ships as part of IBM WebSphere

• Orchestra CDSS [Ives+ 05]
  – Materialized approach
  – Skolem terms, recursion, provenance, updates
Datalog for Data Integration: Some Open Issues

• Materialized data exchange: renewed need for efficient **incremental view maintenance** algorithms
  – Source databases are dynamic entities, need to propagate changes
  – Classical algorithm DRed [Gupta+ 93] often performs very badly; newer provenance-based algorithms [Green+ 07, Liu+ 08] faster but incur space overhead; can we do better?

• **Termination** for Datalog with Skolems
  – Improvements on weak ayclicity for chase termination, translate to Datalog; more permissive conditions always useful!
  – Is termination even decidable? (Undecidable if we allow Skolems *and* unsafe rules, of course.)
Outline of Tutorial

June 14, 2011: The Second Coming of Datalog!

• Refresher: basics of Datalog
• Application #1: Data Integration and Exchange
• Application #2: Program Analysis
• Application #3: Declarative Networking
• Conclusion
Program Analysis

• What is it?

• Why in Datalog?

• How does it work?
Program Analysis

• What is it?
  – Fundamental analysis aiding software development
  – Help make programs run fast, help you find bugs

• Why in Datalog?

• How does it work?
Program Analysis

• **What is it?**
  – Fundamental analysis aiding software development
  – Help make programs run fast, help you find bugs

• **Why in Datalog?**
  – Declarative recursion

• **How does it work?**
Program Analysis

• What is it?
  – Fundamental analysis aiding software development
  – Help make programs run fast, help you find bugs

• Why in Datalog?
  – Declarative recursion

• How does it work?
  – Really well! An order-of-magnitude faster than hand-tuned, Java tools
Program Analysis

• What is it?
  – Fundamental analysis aiding software development
  – Help make programs run fast, help you find bugs

• Why in Datalog?
  – Declarative recursion

• How does it work?
  – Really well! An order-of-magnitude faster than hand-tuned, Java tools
  – Datalog optimizations are crucial in achieving performance
WHAT IS PROGRAM ANALYSIS
Understanding Program Behavior

animal.eat((Food) thing);
Understanding Program Behavior
(without actually running the program)

animal.eat((Food) thing);
Understanding Program Behavior

testing
(without actually running the program)

animal.eat((Food) thing);
Understanding Program Behavior

(testing)
(without actually running the program)

what is animal?

animal.eat((Food) thing);
Understanding Program Behavior

testing (without actually running the program)

what is animal?

points-to analyses

animal.eat((Food) thing);
Understanding Program Behavior

(testing (without actually running the program))

What is *animal*?

Points-to analyses

Through what method does it *eat*?

```cpp
animal.eat((Food) thing);
```
Understanding Program Behavior (without actually running the program)

Through what method does it eat?

What is animal?

What is thing?

Points-to analyses

animal.eat((Food) thing);
Optimizations

what is \textit{animal}?

what is \textit{thing}?

animal.eat((Food) \textit{thing});

through what method does it \textit{eat}?
Optimizations

it’s a Dog

what is thing?

animal.eat((Food) thing);

through what method does it eat?
Optimizations

class Dog {
  void eat(Food f) {
  ...  }
}

animal.eat((Food) thing);

what is thing?

it's a Dog
Optimizations

animal.eat((Food) thing);

it's a Dog

what is thing?

t class Dog {
    void eat(Food f) { ... }
}

virtual call resolution
Optimizations

`animal.eat( (Food) thing);`

it’s a **Dog**

virtual call resolution

class Dog {
    void eat(Food f) { ... }
}

it’s **Chocolate**
Optimizations

```cpp
class Dog {
    void eat(Food f) { ... }
}
```

virtual call resolution

```cpp
animal.eat((Food) thing);
```

it's a Dog

it's Chocolate

193
animal.eat((Food) thing); through what method does it eat? what is thing? what is animal? it's a Dog class Dog {
    void eat(Food f) { ... }
}
Bug Finding

animal.eat((Food) thing);

what is thing?
what is animal?

it’s a Dog
it’s Chocolate

class Dog {
    void eat(Food f) { ... }
}

195
Bug Finding

animal.eat((Food) thing);

what is thing?
what is animal?

it's a Dog
it's Chocolate

Dog + Chocolate = BUG

class Dog {
    void eat(Food f) { ... }
}

196
Bug Finding

```java
class Dog {
    void eat(Food f) { ... }
}

ChokeException never caught = BUG

Dog + Chocolate = BUG
```

- it’s a Dog
- it’s Chocolate
- `animal.eat(Food thing);`
- `ChokeException` never caught = BUG
- `Dog + Chocolate = BUG`
Precise, Fast Program Analysis Is Hard

- necessarily an approximation
Precise, Fast Program Analysis Is Hard

• necessarily an approximation
  – because Alan Turing said so
Precise, Fast Program Analysis Is Hard

• necessarily an approximation
  – because Alan Turing said so
• a *lot* of possible execution paths to analyze
Precise, Fast Program Analysis Is Hard

• necessarily an approximation
  – because Alan Turing said so

• a *lot* of possible execution paths to analyze
  – $10^{14}$ acyclic paths in an average Java program, *Whaley et al., ‘05*
WHY PROGRAM ANALYSIS IN DATALOG?
WHY PROGRAM ANALYSIS IN A DECLARATIVE LANGUAGE?
WHY PROGRAM ANALYSIS IN A DECLARATIVE LANGUAGE?

WHY DATALOG?
Program Analysis: A Complex Domain

1. Pointer analysis: haven't we solved this problem yet?
   Michael Hunt
   June 2001 PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering
   Publisher: ACM
   Full text available: PDF (199.83 KB)
   Bibilometrics: Downloads (6 Weeks): 25, Downloads (12 Months): 191, Downloads (Overall): 1523, Citation Count: 100
   
   During the past twenty-one years, over seventy-five papers and nine Ph.D. theses have been published on pointer analysis. Given the tomes of work on this topic one may wonder, "Haven't we solved this problem yet?" With input from many researchers...

2. A schema for interprocedural modification side-effect analysis with pointer aliasing
   Barbara G. Ryder, William A. Land, Philip A. Stocks, Sean Zhang, Rita Attacher
   March 2001 Transactions on Programming Languages and Systems (TOPLAS), Volume 23 Issue 2
   Publisher: ACM
   Full text available: PDF (1.72 MB)
   Bibilometrics: Downloads (6 Weeks): 5, Downloads (12 Months): 59, Downloads (Overall): 675, Citation Count: 31
   
   The first interprocedural modification side-effects analysis for C (MIDC) that obtains better than worst-case precision on programs with general-purpose pointer usage is presented with empirical results. The analysis...

3. Semi-sparse flow-sensitive pointer analysis
   Ben Hardtкоп, Calvin Lin
   Publisher: ACM
   Full text available: PDF (246.09 KB)
   Bibilometrics: Downloads (6 Weeks): 12, Downloads (12 Months): 108, Downloads (Overall): 348, Citation Count: 6
   
   Pointer analysis is a prerequisite for many program analyses, and the effectiveness of these analyses depends on the precision of the pointer information they receive. Two major axes of pointer analysis precision are flow-sensitivity and context-sensitivity...
   
   Keywords: alias analysis, pointer analysis
   Also published in:
   January 2009 SIGPLAN Notices Volume 44 Issue 1

4. Efficient field-sensitive pointer analysis of C
   David J. Pearce, Paul H.J. Kelly, Chris Hankin
   November 2007 Transactions on Programming Languages and Systems (TOPLAS), Volume 30 Issue 1
Program Analysis: A Complex Domain

1. Pointer analysis: haven't we solved this problem yet?
   Michael Hrid
   June 2001 PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering
   Publisher: ACM  Request Permissions
   Full text available: pdf (159.83 KB)
   Bibliometrics: Downloads (6 Weeks): 25, Downloads (12 Months): 101, Downloads (Overall): 1523, Citation Count: 100

   During the past twenty-one years, over seventy-five papers and nine Ph.D. theses have been published on pointer analysis. Given the tombs of work on this topic one may wonder, "Haven't we solved this problem yet?" With input from many researchers...

2. A schema for interprocedural modification side-effect analysis with pointer aliasing
   Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, Rita Attacher
   March 2001 Transactions on Programming Languages and Systems (TOPLAS), Volume 23 Issue 2
   Publisher: ACM  Request Permissions
   Full text available: pdf (1.72 MB)
   Bibliometrics: Downloads (6 Weeks): 5, Downloads (12 Months): 59, Downloads (Overall): 675, Citation Count: 31

   The first interprocedural modification side-effects analysis for C (MODC) that obtains better than worst-case precision on programs with general-purpose pointer usage is presented with empirical results. The analysis...

3. Semi-sparse flow-sensitive pointer analysis
   Ben Hardock, Calvin Lin
   Publisher: ACM  Request Permissions
   Full text available: pdf (246.09 KB)
   Bibliometrics: Downloads (6 Weeks): 12, Downloads (12 Months): 108, Downloads (Overall): 348, Citation Count: 6

   Pointer analysis is a prerequisite for many program analyses, and the effectiveness of these analyses depends on the precision of the pointer information they receive. Two major axes of pointer analysis precision are flow-sensitivity and context-sensitivity,...

   Keywords: alias analysis, pointer analysis

   Also published in:
   January 2001 SIGPLAN Notices Volume 44 Issue 1

4. Efficient field-sensitive pointer analysis of C
   David J. Pearce, Paul H.J. Kelly, Chris Hankin
   November 2007 Transactions on Programming Languages and Systems (TOPLAS), Volume 30 Issue 1

   Efficient field-sensitive pointer analysis of C...
Program Analysis: A Complex Domain

- flow-sensitive
- inclusion-based
- unification-based
- k-cfa
- object-sensitive
- context-sensitive
- field-based
- field-sensitive
- BDDs
- heap-sensitive
Algorithm in 10-page Conf. Papers

procedure exhaustive_aliasing(G)
  G: an interprocedural control flow graph (ICFG);
begin
  /* 1. only performed implicitly */
  1. initialize may_hold with a default value NO;
     create an empty worklist;
  2. for each node N in G
     2.1 if N is a pointer assignment
         aliases_intro_by_assignment(N,YES);
     2.2 else if N is a call node
         aliases_intro_by_call(N,YES);
  3. while worklist is not empty
     3.1 remove (N, AA, PA) from worklist;
     3.2 if N is a call node
         alias_at_call_implies(N, AA, PA, YES);
     3.3 else if N is an exit node
         alias_at_exit_implies(N, AA, PA, YES);
     3.4 else for each M ∈ successor(N)
         3.4.1 if M is a pointer assignment
             alias_implies_thru_assign(M,
             AA, PA, YES);
         3.4.2 else
             make_true(M, AA, PA);
end

Figure 1: Exhaustive algorithm for pointer aliasing
Algorithms in 10-page Conf. Papers

**Figure 1:** Exhaustive algorithm for **pointer aliasing**

```
procedure exhaustive_aliasing(G)
begin
    procedure incremental_aliasing(G,N)
        G: an ICFG;
        N: a statement to be changed;
    begin
        1. falsify the affected aliases, which are either generated
           at N, or depend on other affected aliases.
        2. update G to reflect the change to statement N;
        3. worklist=reintroduce_aliases(G);
        4. reiterate_worklist(worklist,YES);
    end
    Figure 2: Incremental aliasing algorithm for handling
    addition/deletion of a statement

    3.4 else for each M ∈ successor(N)
        3.4.1 if M is a pointer assignment
            alias_implies_thru_assign(M,
            AA,PA,YES);
        3.4.2 else
            make_true(M,AA,PA);
    end
```

209
Algorithms in 10-page Conf. Papers

begin
1. N: a statement to be changed
2. set all may-hold(N, AA, PA) to NO;
3. if N is marked TOUCHEd, return;
4. set all may-hold(N, AA, PA) to FALSEFIEd;
5. for each call node C which calls the function
     containing N:
     1. if C is marked TOUCHEd, disable-aliases at the changed node N *;
     2. if C is marked TOUCHEd, N: a statement to be changed;
     3. if C is marked TOUCHEd, disable-aliases at the changed node N *;
     4. if C is marked TOUCHEd, disable-aliases at the changed node N *;
     5. else if C is a call node
     6. else for each M ∈ successor(N):

end
procedure exhaustive_aliasing(G)
begin
/* Alias falsification corresponding to step 1 in Figure 2 */
procedure reintroduce_aliases(G)
begin
1. if
2. return
3. if
4. if
end
end
end
procedure reintroduce_aliases(G)
begin
1. if
2. if
3. if
4. return
end
end
procedure repropagate_aliases(N, worklist)
N: a program node in the ICFG;
worklist: a worklist for keeping the reintroduced aliases;
begin
for each may_hold(N, AA, PA) = YES
   end
Algorithms in 10-page Conf. Papers

```plaintext
procedure exhaustive_aliasing(G)
begin
    procedure alias_falsification corresponding to step 1 in Figure 2 */
    procedure alias reintroduction corresponding to step 3 in Figure 2 */

    procedure reiterate_worklist(worklist, value)
        worklist: a worklist for keeping the aliases to process;
        value: value that will be given to (N, AA, PA);
        begin
            1. while worklist is not empty do
                1.1 remove (N, AA, PA) from worklist;
                1.2 if N is a call node
                    aliases_propagated_at_call(N, AA, PA, value);
                1.3 else if N is an exit node
                    alias_at_exit_implies(N, AA, PA, value);
                1.4 else for each M ∈ successor(N)
                    1.4.1 if M is a pointer assignment
                        alias_implies_thru_assign(M, AA, PA, value);
                    1.4.2 else if value is YES
                        make_true(M, AA, PA);
                    1.4.3 else /* value is FALSIFIED */
                        make_false(M, AA, PA);
```
Algorithms in 10-page Conf. Papers

procedure exhaustive_aliasing(G)
begin
procedure exhaustively_reintroduce(G)
begin
/* Alias falsification corresponding to step 1 in Figure 2 */
procedure falsify_for_deleting_assign(N)
begin
/* Alias falsification for deleting a pointer assignment corresponding to step 1 in Figure 2 */
procedure falsify_for_deleting_call(N)
begin
/* Alias falsification for deleting a pointer assignment corresponding to step 1 in Figure 2 */
procedure falsify_for_deleting_assignment(N)
begin
/* Alias falsification for deleting a pointer assignment corresponding to step 1 in Figure 2 */
procedure falsify_for_deleting_call(N)
begin
/* Alias falsification for deleting a pointer assignment corresponding to step 1 in Figure 2 */
procedure falsify_for_deleting_assignment(N)
begin
/* Alias falsification for deleting a pointer assignment corresponding to step 1 in Figure 2 */
procedure falsify_for_deleting_call(N)
begin
Algorithms in 10-page Conf. Papers

```plaintext
procedure exhaustive_aliasing(G)
    begin
        N: a node in G;
        begin
            /* Alias falsification corresponding to step 1 in Figure 2 */
            procedure alias_falsification(G, )
                begin
                    G: a graph representing the program;
                    begin
                        /* Alias reintroduction corresponding to step 3 in Figure 2 */
                        procedure alias_reintroduction(G, )
                            begin
                                G: a graph representing the program;
                                begin
                                    /* Reiteration corresponding to step 6 in Figure 2 */
                                    procedure reiteration(G, )
                                        begin
                                            G: a graph representing the program;
                                            begin
                                                procedure update_for_adding_assign(N, M)
                                                    begin
                                                        N: a pointer assignment to be added;
                                                        M: the statement after which statement N is added;
                                                        begin
                                                            1. make N as a successor of M, and leave N without any successors;
                                                            2. create an empty worklist;
                                                            3. aliases_intro_by_assignment(N, YES);
                                                            4. repropagate_aliases(M, worklist);
                                                            5. reiterate_worklist(worklist, YES);
                                                            6. for each may_hold(M, AA, PA = (o1, o2)) = YES, and may_hold(N, AA, PA) = NO
                                                                add (M, AA, PA) to worklist;
                                                            7. reiterate_worklist(worklist, FALSIFIED);
                                                    end
                                                end
                                            end
                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
```
Algorithms in 10-page Conf. Papers

variation points unclear

procedure update_for_adding_assign(N, M)
    N: a pointer assignment to be added;
    M: the statement after which statement N is added;
begin
    1. make N as a successor of M, and leave N without any successors;
    2. create an empty worklist;
    3. aliases_intro_by_assignment(N, YES);
    4. repropagate_aliases(M, worklist);
    5. reiterate_worklist(worklist, YES);
    6. for each may_hold(M, AA, PA = (o1, o2)) = YES, and may_hold(N, AA, PA) = NO
       add (M, AA, PA) to worklist;
    7. reiterate_worklist(worklist, FALSIFIED);
end

Figure 8: Procedure for falsifying aliases that are potentially affected by adding a pointer assignment
Algorithms in 10-page Conf. Papers

Variation points unclear

Every variation new algorithm
Algorithms in 10-page Conf. Papers

**variation points unclear**

**every variation new algorithm**

**correctness unclear**
Algorithms in 10-page Conf. Papers

- variation points unclear
- every variation new algorithm
- correctness unclear
- incomparable in precision

**Figure 8:** Procedure for falsifying aliases that are potentially affected by adding a pointer assignment
Algorithms in 10-page Conf. Papers

- Variation points unclear
- Every variation new algorithm
- Correctness unclear
- Incomparable in precision
- Incomparable in performance
Want: Specification + Implementation
Want: Specification + Implementation
Want: Specification + Implementation

Specifications

Declarative Language Runtime

Implementation
DECLARATIVE = GOOD

WHY DATALOG?
Program Analysis: Domain of Mutual Recursion

var points-to
Program Analysis: Domain of Mutual Recursion

\[ x = y; \]
\[ \text{var points-to} \]
Program Analysis: Domain of Mutual Recursion

\[ x = y; \]

\text{var points-to}
Program Analysis: Domain of Mutual Recursion

\[ x = f(); \]

\[ \text{var points-to} \]
Program Analysis: Domain of Mutual Recursion

\[ x = f(); \]
Program Analysis: Domain of Mutual Recursion

```plaintext
x = y.f();
```

```
var points-to
```

```
call graph
```
Program Analysis: Domain of Mutual Recursion

\[ x = y.f(); \]

\[ \text{var points-to} \]

\[ \text{call graph} \]
Program Analysis: Domain of Mutual Recursion

x.f = y;

var points-to

call graph

fields points-to
Program Analysis: Domain of Mutual Recursion

var points-to

x.f = y;
call graph

fields points-to
Program Analysis: Domain of Mutual Recursion

\[ x = y.f; \]

var points-to

fields points-to

call graph
Program Analysis: Domain of Mutual Recursion

\[ x = y.f; \]

- \text{var points-to}
- \text{call graph}
- \text{fields points-to}
Program Analysis: Domain of Mutual Recursion

```
var points-to

call graph

fields points-to

throw e

exceptions
```
Program Analysis: Domain of Mutual Recursion

- var points-to
- throw e
- call graph
- exceptions
- fields points-to
Program Analysis: Domain of Mutual Recursion

- var points-to
  - catch (E e)
  - call graph
  - exceptions
  - fields points-to
Program Analysis: Domain of Mutual Recursion

- var points-to
- catch (E e)
- call graph
- exceptions
- fields points-to
Program Analysis: Domain of Mutual Recursion

var points-to

g()
call graph

fields points-to

exceptions
Program Analysis: Domain of Mutual Recursion

var points-to

```
points
```

call graph

exceptions

fields points-to

```
g()```

A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases

Data integration

'77 '80s ... '95

LDL, NAIL, Coral, ...

Declerative networking

BDDBDDI

Orchestra CDSS

Information Extraction

'02 '05 '07 '08 '10

Access control (Binder)

Doop (pointer-analysis)

Evita Raced

QL

SecureBlox

LiXto
A Brief History of Datalog

- Workshop on Logic and Databases
- Data integration
- Control + data flow

- LDL, NAIL, Coral, ...

- '77
- '80s...
- '95

- Declarative networking
- BDDBDI
- Orchestra CDSS
- Information Extraction
- SecureBlox
- Access control (Binder)
- Doop (pointer-
- Evita Raced
- .QL

- '02
- '05
- '07
- '08
- '10

- ...
A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases
LDL, NAIL, Coral, ...

Data integration

‘77 ‘80s ... ‘95

Declarative networking
BDDBDDBI

Orchestra CDSS
Information Extraction

Access control (Binder)

‘02 ‘05 ‘07 ‘08 ‘10

Doop (pointer-
analysis)

Evita Raced

.info

QL

Evita

secureblox

logicblox

LiXto

semmler
A Brief History of Datalog

Control + data flow

Workshop on Logic and Databases

Data integration

'77    '80s ...    '95

LDL, NAIL, Coral, ...

Declarative networking

BDDBDDI

Orchestra CDSS

Information Extraction

Access control (Binder)

Doop (pointer-)

Evita Raced

.QL

Information Extraction
PROGRAM ANALYSIS IN DATALOG
Points-to Analyses for A Simple Language

program

a = new A();
b = new B();
c = new C();
a = b;
a = b;
b = a;
b = a;
c = b;
Points-to Analyses for A Simple Language

```
program

a = new A();
b = new B();
c = new C();
a = b;
a = b;
b = a;
b = a;
c = b;
```

248
Points-to Analyses for A Simple Language

```
program

a = new A();
b = new B();
c = new C();
a = b;
a = b;
b = a;
c = b;
```
Points-to Analyses for A Simple Language

```java
program
    a = new A();
    b = new B();
    c = new C();
    a = b;
    a = b;
    b = a;
    c = b;
```
Points-to Analyses for A Simple Language

What objects can a variable point to?

```
program
a = new A();
b = new B();
c = new C();
a = b;
```
Points-to Analyses for A Simple Language

What objects can a variable point to?

```
program

a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```
Points-to Analyses for A Simple Language

What objects can a variable point to?

program

```java
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

assignObjectAllocation
What objects can a variable point to?

```java
program
a = new A();
b = new B();
c = new C();
a = b;
a = b;
```

assignObjectAllocation
- a: new A()
Points-to Analyses for A Simple Language

What objects can a variable point to?

**Program**

```java
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

**assignObjectAllocation**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
</tbody>
</table>
Points-to Analyses for A Simple Language

What objects can a variable point to?

**program**

```java
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

**assignObjectAllocation**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>
Points-to Analyses for A Simple Language

What objects can a variable point to?

**program**

```java
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

**assignObjectAllocation**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>
What objects can a variable point to?

```
program
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```
Points-to Analyses for A Simple Language

What objects can a variable point to?

**program**

a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;

**assignObjectAllocation**

<table>
<thead>
<tr>
<th>a</th>
<th>new A()</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

**assign**

| b   | a       |
Points-to Analyses for A Simple Language

What objects can a variable point to?

**program**

- `a = new A();`
- `b = new B();`
- `c = new C();`
- `a = b;`
- `b = a;`
- `c = b;`

**assignObjectAllocation**

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td><code>new A()</code></td>
</tr>
<tr>
<td><code>b</code></td>
<td><code>new B()</code></td>
</tr>
<tr>
<td><code>c</code></td>
<td><code>new C()</code></td>
</tr>
</tbody>
</table>

**assign**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b</code></td>
<td><code>a</code></td>
</tr>
<tr>
<td><code>a</code></td>
<td><code>b</code></td>
</tr>
</tbody>
</table>
Points-to Analyses for A Simple Language

What objects can a variable point to?

**program**

```java
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

**assignObjectAllocation**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

**assign**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>
Defining varPointsTo

**program**

a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;

<table>
<thead>
<tr>
<th>assignObjectAllocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>assign</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>
Defining varPointsTo

<table>
<thead>
<tr>
<th>program</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = new A();</td>
</tr>
<tr>
<td>b = new B();</td>
</tr>
<tr>
<td>c = new C();</td>
</tr>
<tr>
<td>a = b;</td>
</tr>
<tr>
<td>b = a;</td>
</tr>
<tr>
<td>c = b;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>assignObjectAllocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>assign</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>
Defining varPointsTo

<table>
<thead>
<tr>
<th>program</th>
<th>assignObjectAllocation</th>
<th>varPointsTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = new A();</td>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b = new B();</td>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c = new C();</td>
<td>c</td>
<td>new C()</td>
</tr>
<tr>
<td>a = b;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b = a;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c = b;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Defining `varPointsTo`

```plaintext
program
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

<table>
<thead>
<tr>
<th>assignObjectAllocation</th>
<th>varPointsTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>assign</th>
<th>varPointsTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>
### Defining varPointsTo

**program**

```plaintext
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

**assignObjectAllocation**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

**assign**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

**varPointsTo**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

**varPointsTo(Var, Obj)**

<- assignObjectAllocation(Var, Obj).
Defining varPointsTo

**Program**

```plaintext
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

**assignObjectAllocation**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

**assign**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

**varPointsTo**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

varPointsTo(Var, Obj) <- assignObjectAllocation(Var, Obj).
### Defining `varPointsTo`

**Program**

```java
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

**`assignObjectAllocation`**

<table>
<thead>
<tr>
<th>Var</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td><code>new A()</code></td>
</tr>
<tr>
<td><code>b</code></td>
<td><code>new B()</code></td>
</tr>
<tr>
<td><code>c</code></td>
<td><code>new C()</code></td>
</tr>
</tbody>
</table>

**`assign`**

<table>
<thead>
<tr>
<th>Var</th>
<th>Var</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b</code></td>
<td><code>a</code></td>
</tr>
<tr>
<td><code>a</code></td>
<td><code>b</code></td>
</tr>
<tr>
<td><code>b</code></td>
<td><code>c</code></td>
</tr>
</tbody>
</table>

**`varPointsTo`**

```java
varPointsTo(Var, Obj) <- assignObjectAllocation(Var, Obj).
```
Defining \texttt{varPointsTo}

\begin{align*}
\texttt{program} & \quad \texttt{assignObjectAllocation} & \quad \texttt{varPointsTo} \\
\begin{array}{ll}
a = \text{new } A(); & \text{a} \rightarrow \text{new } A() \\
b = \text{new } B(); & \text{b} \rightarrow \text{new } B() \\
c = \text{new } C(); & \text{c} \rightarrow \text{new } C() \\
a = b; & \text{assign} \\
b = a; & \text{assign} \\
c = b; & \end{array} & \begin{array}{ll}
a & \text{a} \\
b & \text{b} \\
c & \text{c} \\
\end{array} & \begin{array}{ll}
a & \text{new } A() \\
b & \text{new } B() \\
c & \text{new } C() \\
\end{array} \\
\end{align*}

\texttt{varPointsTo(Var, Obj)} \\
\quad \leftarrow \texttt{assignObjectAllocation(Var, Obj)}.

\texttt{varPointsTo(To, Obj)} \\
\quad \leftarrow \texttt{assign(From, To), varPointsTo(From, Obj)}.
### Defining varPointsTo

**program**

\[
\begin{align*}
    a &= \text{new A();} \\
    b &= \text{new B();} \\
    c &= \text{new C();} \\
    a &= b; \\
    b &= a; \\
    c &= b;
\end{align*}
\]

**assignObjectAllocation**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

**assign**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

\[
\text{varPointsTo(Var, Obj) <- assignObjectAllocation(Var, Obj).}
\]

\[
\text{varPointsTo(To, Obj) <- assign(From, To), varPointsTo(From, Obj).}
\]
Defining varPointsTo

**program**

a = new A();
b = new B();
c = new C();

**assignObjectAllocation**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

**assign**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

**varPointsTo**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

\[
\text{varPointsTo}(\text{Var}, \text{Obj}) <\to \text{assignObjectAllocation}(\text{Var, Obj}).
\]

\[
\text{varPointsTo}(\text{To}, \text{Obj}) <\to \text{assign}(\text{From, To}), \text{varPointsTo}(\text{From, Obj}).
\]
Defining varPointsTo

**program**

```java
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

**assignObjectAllocation**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

**assign**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

**varPointsTo**

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

**varPointsTo** function:

- `varPointsTo(Var, Obj) <- assignObjectAllocation(Var, Obj).`
- `varPointsTo(To, Obj) <- assign(From, To), varPointsTo(From, Obj).`
### Defining varPointsTo

#### program

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

#### assignObjectAllocation

<table>
<thead>
<tr>
<th>Var</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

#### assign

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

#### varPointsTo

```
varPointsTo(Var, Obj) <- assignObjectAllocation(Var, Obj).
varPointsTo(To, Obj) <- assign(From, To), varPointsTo(From, Obj).
```
Introducing Fields

```plaintext
program
a.F1 = b;
c = b.F2;
```
Introducing Fields

```plaintext
program
a.F1 = b;
c = b.F2;
```
Introducing Fields

program
a.F1 = b;
c = b.F2;
Introducing Fields

```
program
a.F1 = b;
c = b.F2;
```

storeField
Introducing Fields

program

a.F1 = b;
c = b.F2;

storeField

b   a   F1
Introducing Fields

program
a.F1 = b;
c = b.F2;

storeField
| b | a | F1 |
Introducing Fields

```
program
a.F1 = b;
c = b.F2;
```

storeField
b a F1

loadField
Introducing Fields

```
program
a.F1 = b;
c = b.F2;
```

```
storeField
b  a  F1
loadField
b  F2  c
```
Introducing Fields

**program**

```plaintext
a.F1 = b;
c = b.F2;
```

**storeField**

```
b  a  F1
```

**loadField**

```
b  F2  c
```

**fieldPointsTo(BaseObj, Fld, Obj)**
Introducing Fields

**Program**

\[
\begin{align*}
a.F1 &= b; \\
c &= b.F2;
\end{align*}
\]

**Field Points To** (BaseObj, Fld, Obj)

**Store Field**

<table>
<thead>
<tr>
<th>storeField</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

**Load Field**

<table>
<thead>
<tr>
<th>loadField</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

**BaseObj.Fld**

**Obj**
Introducing Fields

program

\(a.F1 = b;\)
\(c = b.F2;\)

\[
\text{storeField}
\begin{array}{ccc}
  b & a & F1 \\
\end{array}
\]

\[
\text{loadField}
\begin{array}{ccc}
  b & F2 & c \\
\end{array}
\]

\[
\text{fieldPointsTo}(\text{BaseObj, Fld, Obj})
\]
\(<-\text{storeField(From, Base, Fld)},\)

\[
\text{BaseObj.Fld} \quad \text{Obj}
\]
Introducing Fields

```
program
a.F1 = b;
c = b.F2;
```

```
storeField
b  a  F1

loadField
b  F2  c

fieldPointsTo(BaseObj, Fld, Obj) <- storeField(From, Base, Fld),
```

```
BaseObj.Fld  Obj
Base.Fld = From
```
Introducing Fields

program
a.F1 = b;
c = b.F2;

storeField
b a F1

loadField
b F2 c

fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Fld),
varPointsTo(Base, BaseObj),
BaseObj.Fld
Obj

Base.Fld = From
Introducing Fields

**program**

\[
a \cdot F1 = b; \\
c = b \cdot F2;
\]

**storeField**

\[
\begin{array}{c}
b \\
a \\
F1
\end{array}
\]

**loadField**

\[
\begin{array}{c}
b \\
F2 \\
c
\end{array}
\]

\[
\text{fieldPointsTo}(\text{BaseObj}, Fld, Obj) \\
\leftarrow \text{storeField}(\text{From}, \text{Base}, Fld), \\
\text{varPointsTo}(\text{Base}, \text{BaseObj}), \\
\text{BaseObj.Fld} \\
\text{Obj}
\]

\[
\text{Base.Fld} = \text{From}
\]
Introducing Fields

program
a.F1 = b;
c = b.F2;

fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Fld),
  varPointsTo(Base, BaseObj),
  varPointsTo(From, Obj).

storeField
b a F1

loadField
b F2 c

Base.Fld = From
Introducing Fields

Program

a.F1 = b;
c = b.F2;

FieldPointsTo(BaseObj, Fld, Obj) <- storeField(From, Base, Fld),
varPointsTo(Base, BaseObj),
varPointsTo(From, Obj).

storeField
b  a  F1

loadField
b  F2  c

BaseObj.Fld

Obj

Base.Fld = From
Introducing Fields

```
program
a.F1 = b;
c = b.F2;
```

```
fieldPointsTo(BaseObj, Fld, Obj) <- storeField(From, Base, Fld),
varPointsTo(Base, BaseObj),
varPointsTo(From, Obj).
```

```
storeField
b a F1
loadField
b F2 c
```

```
BaseObj.Fld = Obj
Base.Fld = From
```
Introducing Fields

(program)

\( a.F1 = b; \)
\( c = b.F2; \)

**storeField**

\( b \quad a \quad F1 \)

**loadField**

\( b \quad F2 \quad c \)

fieldPointsTo(BaseObj, Fld, Obj) <- storeField(From, Base, Fld),
varPointsTo(Base, BaseObj),
varPointsTo(From, Obj).

varPointsTo(To, Obj)
Introducing Fields

```
program
a.F1 = b;
c = b.F2;
```

```
fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Fld),
  varPointsTo(Base, BaseObj),
  varPointsTo(From, Obj).
```

```
varPointsTo(To, Obj)
<- loadField(Base, Fld, To),
```

```
storeField
b a F1
loadField
b F2 c
```
Introducing Fields

program

\[
a.F1 = b; \\
c = b.F2;
\]

\[
\text{fieldPointsTo}(\text{BaseObj}, \text{Fld}, \text{Obj}) \leftarrow \text{storeField}(\text{From}, \text{Base}, \text{Fld}), \\
\text{varPointsTo}(\text{Base}, \text{BaseObj}), \\
\text{varPointsTo}(\text{From}, \text{Obj}).
\]

\[
\text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{loadField}(\text{Base}, \text{Fld}, \text{To}),
\]

\[
\text{storeField} \\
b \quad a \quad F1
\]

\[
\text{loadField} \\
b \quad F2 \quad c
\]
Introducing Fields

```
program
a.F1 = b;
c = b.F2;
```

```
storeField
b a F1
```

```
loadField
b F2 c
```

```
fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Fld),
  varPointsTo(Base, BaseObj),
  varPointsTo(From, Obj).
```

```
varPointsTo(To, Obj)
<- loadField(Base, Fld, To),
  varPointsTo(Base, BaseObj),
```

```
BaseObj.Fld  Obj
Base.Fld = From
```

```
To = Base.Fld
```
Introducing Fields

program
a.F1 = b;
c = b.F2;

storeField
b  a  F1

loadField
b  F2  c

fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Fld),
  varPointsTo(Base, BaseObj),
  varPointsTo(From, Obj).

varPointsTo(To, Obj)
<- loadField(Base, Fld, To),
  varPointsTo(Base, BaseObj),
  To = Base.Fld
Introducing Fields

program
a.F1 = b;
c = b.F2;

storeField
b a F1

loadField
b F2 c

fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Fld),
   varPointsTo(Base, BaseObj),
   varPointsTo(From, Obj).

varPointsTo(To, Obj)
<- loadField(Base, Fld, To),
   varPointsTo(Base, BaseObj),
   fieldPointsTo(BaseObj, Fld, Obj).
Introducing Fields

```
program
a.F1 = b;
c = b.F2;
```

```
storeField
b  a  F1
```

```
loadField
b  F2  c
```

```
fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Fld),
  varPointsTo(Base, BaseObj),
  varPointsTo(From, Obj).
```

```
varPointsTo(To, Obj)
<- loadField(Base, Fld, To),
  varPointsTo(Base, BaseObj),
  fieldPointsTo(BaseObj, Fld, Obj).
```
Introducing Fields

program
a.F1 = b;
c = b.F2;

fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Fld),
varPointsTo(Base, BaseObj),
varPointsTo(From, Obj).

varPointsTo(To, Obj)
<- loadField(Base, Fld, To),
varPointsTo(Base, BaseObj),
fieldPointsTo(BaseObj, Fld, Obj).

storeField
b a F1
loadField
b F2 c
Introducing Fields

```
program
a.F1 = b;
c = b.F2;
```

```
fieldPointsTo(BaseObj, Fld, Obj)
 <- storeField(From, Base, Fld),
  varPointsTo(Base, BaseObj),
  varPointsTo(From, Obj).
```

```
varPointsTo(To, Obj)
 <- loadField(Base, Fld, To),
  varPointsTo(Base, BaseObj),
  fieldPointsTo(BaseObj, Fld, Obj).
```

**Enhance specification without changing base code**
Introducing Fields

Program

\[ a.F1 = b; \]
\[ c = b.F2; \]

\[ \text{storeField} \]
\[ b \quad a \quad F1 \]

\[ \text{loadField} \]
\[ b \quad F2 \quad c \]

\[ \text{fieldPointsTo} \]
\[ \text{(BaseObj, Fld, Obj)} \]
\[ \leftarrow \text{storeField(From, Base, Fld)}, \]
\[ \text{varPointsTo(Base, BaseObj)}, \]
\[ \text{varPointsTo(From, Obj)}. \]

Enhance specification without changing base code
Introducing Fields

program
a.F1 = b;
c = b.F2;

storeField
b  a  F1

loadField
b  F2  c

fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Fld),
  varPointsTo(Base, BaseObj),
  varPointsTo(From, Obj).

varPointsTo(To, Obj)
<- loadField(Base, Fld, To),
  varPointsTo(Base, BaseObj),
  fieldPointsTo(BaseObj, Fld, Obj).

Enhance specification without changing base code
Introducing Fields

Program

\[
a.F1 = b; \\
c = b.F2;
\]

fieldPointsTo(BaseObj, Fld, Obj)

\[
<- \text{storeField} (\text{From}, \text{Base}, \text{Fld}), \\
\text{varPointsTo} (\text{Base}, \text{BaseObj}), \\
\text{varPointsTo} (\text{From}, \text{Obj}).
\]

varPointsTo(To, Obj)

\[
<- \text{loadField} (\text{Base}, \text{Fld}, \text{To}), \\
\text{varPointsTo} (\text{Base}, \text{BaseObj}), \\
\text{fieldPointsTo} (\text{BaseObj}, \text{Fld}, \text{Obj}).
\]

storeField

\[
\begin{align*}
b & | a & | F1 \\
b & | F2 & | c
\end{align*}
\]

loadField

Enhance specification without changing base code
Specification + Implementation

Specifications

**varPointsTo** (Var, Obj)
<- assignObjectAllocation(...).

**varPointsTo** (To, Obj)
<- assign(From, To),
  varPointsTo(From, Obj).

**fieldPointsTo** (BaseObj, Fld, Obj)
<- storeField(From, Base, Field),
  varPointsTo(Base, BaseObj),
  varPointsTo(From, Obj).

Implementation
varPointsTo(Var, Obj) <- assignObjectAllocation(...).

varPointsTo(To, Obj) <- assign(From, To),
varPointsTo(From, Obj).

fieldPointsTo(BaseObj, Fld, Obj) <- storeField(From, Base, Field),
varPointsTo(Base, BaseObj),
varPointsTo(From, Obj).

varPointsTo(To, Obj) <- loadField(Base, Field, To),
varPointsTo(Base, BaseObj),
fieldPointsTo(BaseObj, ...).

Doop: ~2500 lines of logic
**Specifications**

- `varPointsTo(Var, Obj) <- assignObjectAllocation(...).`

- `varPointsTo(To, Obj) <- assign(From, To), varPointsTo(From, Obj).`

- `fieldPointsTo(BaseObj, Fld, Obj) <- storeField(From, Base, Field), varPointsTo(Base, BaseObj), varPointsTo(From, Obj).`

- `varPointsTo(To, Obj) <- loadField(Base, Field, To), varPointsTo(Base, BaseObj), fieldPointsTo(BaseObj, ...).`

**Implementation**
Specifications + Implementation

Specifications

\[ \text{varPointsTo(Var, Obj)} \leftarrow \text{assignObjectAllocation(...).} \]

\[ \text{varPointsTo(To, Obj)} \leftarrow \text{assign(From, To),} \]
\[ \quad \text{varPointsTo(From, Obj).} \]

\[ \text{fieldPointsTo(BaseObj, Fld, Obj)} \leftarrow \text{storeField(From, Base, Field),} \]
\[ \quad \text{varPointsTo(Base, BaseObj),} \]
\[ \quad \text{varPointsTo(From, Obj).} \]

\[ \text{varPointsTo(To, Obj)} \leftarrow \text{loadField(Base, Field, To),} \]
\[ \quad \text{varPointsTo(Base, BaseObj),} \]
\[ \quad \text{fieldPointsTo(BaseObj, ...).} \]

Implementation

Datalog Engine
Specifications

\textbf{varPointsTo}(\text{Var, Obj})
\begin{align*}
\iff & \text{assignObjectAllocation}(\ldots).
\end{align*}

\textbf{varPointsTo}(\text{To, Obj})
\begin{align*}
\iff & \text{assign}(\text{From, To}),
\text{varPointsTo}(\text{From, Obj}).
\end{align*}

\textbf{fieldPointsTo}(\text{BaseObj, Fld, Obj})
\begin{align*}
\iff & \text{storeField}(\text{From, Base, Field}),
\text{varPointsTo}(\text{Base, BaseObj}),
\text{varPointsTo}(\text{From, Obj}).
\end{align*}

\textbf{varPointsTo}(\text{To, Obj})
\begin{align*}
\iff & \text{loadField}(\text{Base, Field, To}),
\text{varPointsTo}(\text{Base, BaseObj}),
\text{fieldPointsTo}(\text{BaseObj, \ldots}).
\end{align*}
Specifications + Implementation

Specifications

\[ \text{varPointsTo}(\text{Var}, \text{Obj}) \leftarrow \text{assignObjectAllocation}(\ldots). \]

\[ \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{assign}(\text{From}, \text{To}), \]
\[ \quad \text{varPointsTo}(\text{From}, \text{Obj}). \]

\[ \text{fieldPointsTo}(\text{BaseObj}, \text{Fld}, \text{Obj}) \leftarrow \text{storeField}(\text{From}, \text{Base}, \text{Field}), \]
\[ \quad \text{varPointsTo}(\text{Base}, \text{BaseObj}), \]
\[ \quad \text{varPointsTo}(\text{From}, \text{Obj}). \]

\[ \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{loadField}(\text{Base}, \text{Field}, \text{To}), \]
\[ \quad \text{varPointsTo}(\text{Base}, \text{BaseObj}), \]
\[ \quad \text{fieldPointsTo}(\text{BaseObj}, \ldots). \]

Implementation

Top-down

Bottom-up

Control

Datalog Engine
**Specifications**

- \( \text{varPointsTo}(\text{Var}, \text{Obj}) \leftarrow \text{assignObjectAllocation}(\ldots). \)

- \( \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{assign}(\text{From}, \text{To}), \text{varPointsTo}(\text{From}, \text{Obj}). \)

- \( \text{fieldPointsTo}(\text{BaseObj}, \text{Fld}, \text{Obj}) \leftarrow \text{storeField}(\text{From}, \text{Base}, \text{Field}), \text{varPointsTo}(\text{Base}, \text{BaseObj}), \text{varPointsTo}(\text{From}, \text{Obj}). \)

- \( \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{loadField}(\text{Base}, \text{Field}, \text{To}), \text{varPointsTo}(\text{Base}, \text{BaseObj}), \text{fieldPointsTo}(\text{BaseObj}, \ldots). \)

**Implementation**

- Control: Top-down, Bottom-up, Tabled

- Datalog Engine

**Specification + Implementation**
Specifications

\( \text{varPointsTo}(\text{Var}, \text{Obj}) \leftarrow \text{assignObjectAllocation}(\ldots). \)

\( \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{assign}(\text{From}, \text{To}), \text{varPointsTo}(\text{From}, \text{Obj}). \)

\( \text{fieldPointsTo}(\text{BaseObj}, \text{Fld}, \text{Obj}) \leftarrow \text{storeField}(\text{From}, \text{Base}, \text{Field}), \text{varPointsTo}(\text{Base}, \text{BaseObj}), \text{varPointsTo}(\text{From}, \text{Obj}). \)

\( \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{loadField}(\text{Base}, \text{Field}, \text{To}), \text{varPointsTo}(\text{Base}, \text{BaseObj}), \text{fieldPointsTo}(\text{BaseObj}, \ldots). \)

Implementation

- Top-down
- Bottom-up
- Tabled
- Semi-naive
- Naive

Control
Specifications

\[ \text{varPointsTo}(\text{Var}, \text{Obj}) \leftarrow \text{assignObjectAllocation}(\ldots). \]

\[ \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{assign}(\text{From}, \text{To}), \]
\[ \quad \text{varPointsTo}(\text{From}, \text{Obj}). \]

\[ \text{fieldPointsTo}(\text{BaseObj}, \text{Fld}, \text{Obj}) \leftarrow \text{storeField}(\text{From}, \text{Base}, \text{Field}), \]
\[ \quad \text{varPointsTo}(\text{Base}, \text{BaseObj}), \]
\[ \quad \text{varPointsTo}(\text{From}, \text{Obj}). \]

\[ \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{loadField}(\text{Base}, \text{Field}, \text{To}), \]
\[ \quad \text{varPointsTo}(\text{Base}, \text{BaseObj}), \]
\[ \quad \text{fieldPointsTo}(\text{BaseObj}, \ldots). \]

Implementation

Control

- Top-down
- Bottom-up
- Tabled
- Naive
- Semi-naive
- Counting
- DReD

Datalog

Engine

Specifications + Implementation
Specifications

\texttt{varPointsTo(Var, Obj)}
\texttt{<- assignObjectAllocation(...).}

\texttt{varPointsTo(To, Obj)}
\texttt{<- assign(From, To),}
\texttt{\hspace{1em} varPointsTo(From, Obj).}

\texttt{fieldPointsTo(BaseObj, Fld, Obj)}
\texttt{<- storeField(From, Base, Field),}
\texttt{\hspace{1em} varPointsTo(Base, BaseObj),}
\texttt{\hspace{1em} varPointsTo(From, Obj).}

\texttt{varPointsTo(To, Obj)}
\texttt{<- loadField(Base, Field, To),}
\texttt{\hspace{1em} varPointsTo(Base, BaseObj),}
\texttt{\hspace{1em} fieldPointsTo(BaseObj, ...).}

Implementation

Control

Top-down

Bottom-up

Tabled

Naive

Semi-naive

DReD

Data Structures
Specifications

\texttt{varPointsTo}(Var, Obj) \leftarrow \text{assignObjectAllocation}(\ldots).

\texttt{varPointsTo}(To, Obj) \leftarrow \text{assign}(From, To), \texttt{varPointsTo}(From, Obj).

\texttt{fieldPointsTo}(BaseObj, Fld, Obj) \leftarrow \text{storeField}(From, Base, Field), \texttt{varPointsTo}(Base, BaseObj), \texttt{varPointsTo}(From, Obj).

\texttt{varPointsTo}(To, Obj) \leftarrow \text{loadField}(Base, Field, To), \texttt{varPointsTo}(Base, BaseObj), \texttt{fieldPointsTo}(BaseObj, \ldots).
Specifications + Implementation

**Specifications**

\[ \text{varPointsTo}(\text{Var}, \text{Obj}) \leftarrow \text{assignObjectAllocation}(\ldots). \]

\[ \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{assign}(\text{From}, \text{To}), \]
\[ \text{varPointsTo}(\text{From}, \text{Obj}). \]

\[ \text{fieldPointsTo}(\text{BaseObj}, \text{Fld}, \text{Obj}) \leftarrow \text{storeField}(\text{From}, \text{Base}, \text{Field}), \]
\[ \text{varPointsTo}(\text{Base}, \text{BaseObj}), \]
\[ \text{varPointsTo}(\text{From}, \text{Obj}). \]

\[ \text{varPointsTo}(\text{To}, \text{Obj}) \leftarrow \text{loadField}(\text{Base}, \text{Field}, \text{To}), \]
\[ \text{varPointsTo}(\text{Base}, \text{BaseObj}), \]
\[ \text{fieldPointsTo}(\text{BaseObj}, \ldots). \]

**Implementation**

- Top-down
- Bottom-up
- Tabled
- Naive
- Semi-naive
- Counting
- DReD
- BTree
- KDTree

**Control**

- Data Structures

**Data Structures**

- BTree
- KDTree
Specifications

varPointsTo(Var, Obj)
<- assignObjectAllocation(…).

varPointsTo(To, Obj)
<- assign(From, To),
    varPointsTo(From, Obj).

fieldPointsTo(BaseObj, Fld, Obj)
<- storeField(From, Base, Field),
    varPointsTo(Base, BaseObj),
    varPointsTo(From, Obj).

varPointsTo(To, Obj)
<- loadField(Base, Field, To),
    varPointsTo(Base, BaseObj),
    fieldPointsTo(BaseObj, …).

Implementation

Control

Top-down

Bottom-up

Tabled

Naive

Semi-naive

Counting

DReD

Data Structures

BDDs

BTree

KDTree
Specifications

\texttt{varPointsTo(Var, Obj)}
\texttt{\quad \leftarrow assignObjectAllocation(...).}

\texttt{varPointsTo(To, Obj)}
\texttt{\quad \leftarrow assign(From, To),}
\texttt{\quad \texttt{varPointsTo(From, Obj).}}

\texttt{fieldPointsTo(BaseObj, Fld, Obj)}
\texttt{\quad \leftarrow storeField(From, Base, Field),}
\texttt{\quad \texttt{varPointsTo(Base, BaseObj),}
\texttt{\quad \texttt{varPointsTo(From, Obj).}}

\texttt{varPointsTo(To, Obj)}
\texttt{\quad \leftarrow loadField(Base, Field, To),}
\texttt{\quad \texttt{varPointsTo(Base, BaseObj),}
\texttt{\quad \texttt{fieldPointsTo(BaseObj, ...).}}

Implementation

\textbf{Control}

- Top-down
- Bottom-up
- Tabled
- Naive
- Semi-naive
- Counting
- DReD

\textbf{Data Structures}

- BDDs
- BTree
- KDTree
- transitive closure
Specification + Implementation

Specifications

Control

Implementation

Top-down

Bottom-up

Naive

Semi-naive

Counting

DReD

Data Structures

BDDs

BTree

transitive closure

KDTree

Datalog Engine

Transitive closure

Semi-naive

Tabled

Naive

Counting

DReD

BTree

transitive closure

KDTree
Specification + Implementation

Does It Run Fast?!?
Doop vs. Paddle: 1-call-site-sensitive-heap
Crucial Optimizations

• something old

• something new(-ish)

• something borrowed (from PL)
Crucial Optimizations

• something old
  – semi-naïve evaluation, folding, index selection
• something new(-ish)

• something borrowed (from PL)
Crucial Optimizations

• something old
  – semi-naïve evaluation, folding, index selection
• something new(-ish)
  – magic-sets
• something borrowed (from PL)
Crucial Optimizations

• something old
  – semi-naïve evaluation, folding, index selection
• something new(-ish)
  – magic-sets
• something borrowed (from PL)
  – type-based
Crucial Optimizations

• something old
  – semi-naïve evaluation, folding, index selection

• something new(-ish)
  – magic-sets

• something borrowed (from PL)
  – type-based
TYPE-BASED OPTIMIZATIONS
Types: Sets of Values
Types: Sets of Values

animal

universe
Types: Sets of Values
Types: Sets of Values

- animal
- food
- thing
- universe
animal(X) -> .

Types: Sets of Values

- animal
- food
- thing
- universe
Types: Sets of Values

animal(X) -> .
animal(X) -> .
bird(X) -> animal(X) .

Types: Sets of Values

- bird
- animal
- food
- thing
Types: Sets of Values

animal(X) -> .

bird(X) -> animal(X).
Types: Sets of Values

animal(X) -> .
bird(X) -> animal(X) .
dog(X) -> animal(X) .

[Diagram showing a universe divided into animal (green), bird (green), dog (red), and food (yellow) categories.]

thing

animal

bird
dog

food

universe
Types: Sets of Values

animal(X) ->  .
bird(X) -> animal(X) .
dog(X) -> animal(X) .
dog(X) -> !bird(X).
bird(X) -> !dog(X).
Types: Sets of Values

animal(X) -> .

bird(X) -> animal(X) .

dog(X) -> animal(X) .

dog(X) -> !bird(X).
bird(X) -> !dog(X).
Types: Sets of Values

animal(X) -> .
bird(X) -> animal(X) .
dog(X) -> animal(X) .
dog(X) -> !bird(X).
bird(X) -> !dog(X).
pet(X) -> animal(X).
“Virtual Call Resolution”

query _(D)
    <- dog(D), eat(D, Thing),
        food(Thing),
        chocolate(Thing).
“Virtual Call Resolution”

query _(D)
    <- dog(D), eat(D, Thing),
        food(Thing),
        chocolate(Thing).

eat(A, Food)
    <- dogChews(A,Food)
        ; birdSwallows(A,Food).
“Virtual Call Resolution”

query _ (D)
   <- dog(D), eat(D, Thing),
      food(Thing),
      chocolate(Thing).

eat(A, Food)
   <- dogChews(A,Food)
      ; birdSwallows(A,Food).
“Virtual Call Resolution”

query _(D)
  <- **dog(D)**, eat(D, Thing),
  food(Thing),
  chocolate(Thing).

D :: dog

eat(A, Food)
  <- dogChews(A,Food)
  ; birdSwallows(A,Food).
“Virtual Call Resolution”

query _(_(D)
  <- dog(D), eat(D, Thing),
        food(Thing),
        chocolate(Thing).

D :: dog

eat(A, Food)
  <- dogChews(A,Food)
    ; birdSwallows(A,Food).
“Virtual Call Resolution”

query _(_D)  
  <- dog(D), eat(D, Thing),  
      food(Thing),  
      chocolate(Thing).

dogChews :: (dog, food)

eat(A, Food)  
  <- dogChews(A,Food)  
    ; birdSwallows(A,Food).
“Virtual Call Resolution”

query _(D)  
  <- dog(D), eat(D, Thing),  
      food(Thing),  
      chocolate(Thing).

eat(A, Food)  
  <- dogChews(A,Food)  
  ; birdSwallows(A,Food).

dogChews :: (dog, food)  
birdSwallows :: (bird, food)
“Virtual Call Resolution”

query _(_D)
  <- dog(_D), eat(_D, Thing),
      food(Thing),
      chocolate(Thing).

eat(A, Food)
  <- dogChews(A,Food)
      ; birdSwallows(A,Food).

D :: dog

dogChews :: (dog, food)
birdSwallows :: (bird, food)
Type Erasure

query _(D)
  <- dog(D), eat(D, Thing),
      food(Thing),
      chocolate(Thing).

eat(A, Food)
  <- dogChews(A,Food)
    ; birdSwallows(A,Food).

dogChews :: (dog, food)
birdSwallows :: (bird, food)
Type Erasure

query _ (D)
    <- dog(D), eat(D, Thing),
        food(Thing),
        chocolate(Thing).

eat(A, Food)
    <- dogChews(A, Food)
        ; birdSwallows(A, Food).

d :: dog

eat :: (dog, food)
Type Erasure

query _(_\text{D})
  \text{<-} \text{dog}(\text{D}), \text{eat}(\text{D}, \text{Thing}),
  \text{food}(\text{Thing}),
  \text{chocolate}(\text{Thing}).

\text{eat}(\text{A}, \text{Food})
  \text{<-} \text{dogChews}(\text{A}, \text{Food})
  ; \text{birdSwallows}(\text{A}, \text{Food}).

\text{D} :: \text{dog}

\text{eat} :: (\text{dog, food})
Type Erasure

query \(_{(D)}\)
  \(<-\) \texttt{dog}(D), \texttt{eat}(D, \texttt{Thing}),
  \texttt{food}(\texttt{Thing}),
  \texttt{chocolate}(\texttt{Thing}).

\texttt{eat}(A, \texttt{Food})
  \(<-\) \texttt{dogChews}(A,\texttt{Food})
  ; \texttt{birdSwallows}(A,\texttt{Food}).

\texttt{D :: dog}

\texttt{eat :: (dog, food)}
Type Erasure

query _(_D)
  <- dog(_D), eat(_D, Thing), food(Thing), chocolate(Thing).

eat(_A, _Food)
  <- dogChews(_A, _Food)
  ; birdSwallows(_A, _Food).

D :: dog

Thing :: chocolate

eat :: (dog, food)
Type Erasure

query _(D)
  <- \texttt{dog(D)}, \texttt{eat(D, Thing)},
   \texttt{food(Thing)},
   \texttt{chocolate(Thing)}.

\texttt{eat(A, Food)}
  <- \texttt{dogChews(A,Food)}
   ; \texttt{birdSwallows(A,Food)}.

\texttt{D :: dog}

\texttt{Thing :: chocolate}

\texttt{eat :: (dog, food)}
Clean Up

query _(D)
  <- dog(D), eat(D, Thing),
      food(Thing),
      chocolate(Thing).

eat(A, Food)
  <- dogChews(A,Food)
     ; birdSwallows(A,Food).

D :: dog

Thing :: chocolate

eat :: (dog, food)
query _(D)  
  <- eat(D,Thing),  
      chocolate(Thing).

eat(A, Food)  
  <- dogChews(A,Food).
References on Datalog and Types

- “Type inference for datalog and its application to query optimisation”, de Moor et al., PODS ‘08
- “Type inference for datalog with complex type hierarchies”, Schafer and de Moor, POPL ‘10
- “Semantic Query Optimization in the Presence of Types”, Meier et al., PODS ‘10
Datalog Program Analysis Systems

• BDD
  – Data structure: BDD

• Semmle (.QL)
  – Object-oriented syntax
  – No update

• Doop
  – Points-to analysis for full Java
  – Supports for many variants of context and heap sensitivity.
REVIEW
Program Analysis

• **What is it?**
  – Fundamental analysis aiding software development
  – Help make programs run fast, help you find bugs

• **Why in Datalog?**
  – Declarative recursion

• **How does it work?**
  – Really well! order of magnitude faster than hand-tuned, Java tools
  – Datalog optimizations are crucial in achieving performance
Program Analysis

understanding program behavior
Program Analysis

imperative
understanding program behavior
Program Analysis
Program Analysis
Program Analysis

Datalog
understanding program behavior
Program Analysis

• “Evita Raced: Meta-compilation for declarative networks”, Condie et al., VLDB ‘08
OPEN CHALLENGES
Traditional View
Datalog: Data Querying Language
Traditional View
Datalog: Data Querying Language

Application Logic
Java
C++
...
Ruby

Middleware
Queries
Traditional View

Datalog: Data Querying Language

UI Logic + Rendering
- Java
- OracleForms
- ...
- JavaScript

Application Logic
- Java
- C++
- ...
- Ruby

Middleware

Queries
New View

Datalog: General Purpose Language
Challenges Raised by Program Analysis

• Datalog **Programming** in the large
Challenges Raised by Program Analysis

• Datalog **Programming** in the large
  – Modularization support
  – Reuse (generic programming)
  – Debugging and Testing
Challenges Raised by Program Analysis

• Datalog **Programming** in the large
  – Modularization support
  – Reuse (generic programming)
  – Debugging and Testing

• Expressiveness:
  – Recursion through negation, aggregation
  – Declarative state
Challenges Raised by Program Analysis

• Datalog **Programming** in the large
  – Modularization support
  – Reuse (generic programming)
  – Debugging and Testing

• Expressiveness:
  – Recursion through negation, aggregation
  – Declarative state

• Optimization, optimization, optimization
  – In the presence of recursion!
Acknowledgements

• Slides:
  – Martin Bravenboer & LogicBlox, Inc.
  – Damien Sereni & Semmle, Inc.
  – Matt Might, University of Utah
Outline of Tutorial

June 14, 2011: The Second Coming of Datalog!

• Refresher: basics of Datalog
• Application #1: Data Integration and Exchange
• Application #2: Program Analysis
• Application #3: Declarative Networking
• Conclusions
Declarative Networking

• A declarative framework for networks:
  – Declarative language: “ask for what you want, not how to implement it”
  – Declarative specifications of networks, compiled to distributed dataflows
  – Runtime engine to execute distributed dataflows
Declarative Networking

• A declarative framework for networks:
  – Declarative language: “ask for what you want, not how to implement it”
  – Declarative specifications of networks, compiled to distributed dataflows
  – Runtime engine to execute distributed dataflows
• Observation: Recursive queries are a natural fit for routing
A Declarative Network

Traditional Networks

Declarative Networks
A Declarative Network

Traditional Networks

- Network State

Declarative Networks

- Distributed database
A Declarative Network

Traditional Networks
- Network State
- Network protocol

Declarative Networks
- Distributed database
- Recursive Query Execution

Distributed recursive query
A Declarative Network

Traditional Networks
- Network State
- Network protocol
- Network messages

Declarative Networks
- Distributed database
- Recursive Query Execution
- Distributed Dataflow
Declarative* in Distributed Systems Programming

- IP Routing [SIGCOMM’05, SIGCOMM’09 demo]
- Overlay networks [SOSP’05]
- Network Datalog [SIGMOD’06]
- Distributed debugging [Eurosys’06]
- Sensor networks [SenSys’07]
- Network composition [CoNEXT’08]
- Fault tolerant protocols [NSDI’08]
- Secure networks [ICDE’09, NDSS’10, SIGMOD’10]
- Replication [NSDI’09]
- Hybrid wireless routing [ICNP’09], channel selection [PRESTO’10]
- Formal network verification [HotNets’09, SIGCOMM’11 demo]
- Network provenance [SIGMOD’10, SIGMOD’11 demo]
- Cloud programming [Eurosys ’10], Cloud testing (NSDI’11)
- ... <More to come>
Open-source systems

• P2 declarative networking system
  – The “original” system
  – Based on modifications to the Click modular router.
  – [http://p2.cs.berkeley.edu](http://p2.cs.berkeley.edu)

• RapidNet
  – Integrated with network simulator 3 (ns-3), ORBIT wireless testbed, and PlanetLab testbed.
  – Security and provenance extensions.
  – Demonstrations at SIGCOMM’09, SIGCOMM’11, and SIGMOD’11
  – [http://netdb.cis.upenn.edu/rapidnet](http://netdb.cis.upenn.edu/rapidnet)

• BOOM – Berkeley Orders of Magnitude
  – BLOOM (DSL in Ruby, uses Dedalus, a temporal logic programming language as its formal basis).
Network Datalog

R1: reachable(@S,D) <- link(@S,D)
R2: reachable(@S,D) <- link(@S,Z), reachable(@Z,D)
Network Datalog

R1: reachable(@S,D) <- link(@S,D)

R2: reachable(@S,D) <- link(@S,Z), reachable(@Z,D)

Location Specifier “@S”

Diagram:

a --- b --- c --- d
R1: reachable(@S,D) <- link(@S,D)

R2: reachable(@S,D) <- link(@S,Z), reachable(@Z,D)

Network Datalog

Location Specifier "@S"

Input table:

```
<table>
<thead>
<tr>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@b</td>
<td>c</td>
</tr>
<tr>
<td>@b</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@c</td>
<td>b</td>
</tr>
<tr>
<td>@c</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@d</td>
<td>c</td>
</tr>
</tbody>
</table>
```
Network Datalog

R1: reachable(@S,D) <- link(@S,D)

R2: reachable(@S,D) <- link(@S,Z), reachable(@Z,D)

query _(M,N) <- reachable(M,N)

Input table:

```
link
@S  D
@a  b

link
@S  D
@b  c
@b  a

link
@S  D
@c  b
@c  d

link
@S  D
@d  c
```
Network Datalog

R1: \( \text{reachable}(\text{@S}, \text{D}) \leftarrow \text{link}(\text{@S}, \text{D}) \)

R2: \( \text{reachable}(\text{@S}, \text{D}) \leftarrow \text{link}(\text{@S}, \text{Z}), \text{reachable}(\text{@Z}, \text{D}) \)

query \( \_\_\_\_\@M, \text{N}\) \( \leftarrow \) \( \text{reachable}(\@M, \text{N}) \) → All-Pairs Reachability

Input table:

<table>
<thead>
<tr>
<th>link (@S)</th>
<th>(\text{D})</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>(b)</td>
</tr>
</tbody>
</table>

Output table:

<table>
<thead>
<tr>
<th>reachable (@S)</th>
<th>(\text{D})</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>(b)</td>
</tr>
<tr>
<td>@a</td>
<td>(c)</td>
</tr>
<tr>
<td>@a</td>
<td>(d)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link (@S)</th>
<th>(\text{D})</th>
</tr>
</thead>
<tbody>
<tr>
<td>@b</td>
<td>(c)</td>
</tr>
<tr>
<td>@b</td>
<td>(a)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link (@S)</th>
<th>(\text{D})</th>
</tr>
</thead>
<tbody>
<tr>
<td>@c</td>
<td>(b)</td>
</tr>
<tr>
<td>@c</td>
<td>(d)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link (@S)</th>
<th>(\text{D})</th>
</tr>
</thead>
<tbody>
<tr>
<td>@d</td>
<td>(a)</td>
</tr>
<tr>
<td>@d</td>
<td>(b)</td>
</tr>
<tr>
<td>@d</td>
<td>(c)</td>
</tr>
</tbody>
</table>
Network Datalog

R1: reachable(@S,D) <- link(@S,D)
R2: reachable(@S,D) <- link(@S,Z), reachable(@Z,D)

query _(@a,N) <- reachable(@a,N)

### Input table:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@S</td>
<td>D</td>
</tr>
<tr>
<td>@a</td>
<td>b</td>
</tr>
<tr>
<td>@b</td>
<td>c</td>
</tr>
<tr>
<td>@b</td>
<td>a</td>
</tr>
<tr>
<td>@c</td>
<td>b</td>
</tr>
<tr>
<td>@c</td>
<td>d</td>
</tr>
<tr>
<td>@d</td>
<td>c</td>
</tr>
</tbody>
</table>

### Output table:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@S</td>
<td>D</td>
</tr>
<tr>
<td>@a</td>
<td>b</td>
</tr>
<tr>
<td>@a</td>
<td>c</td>
</tr>
<tr>
<td>@a</td>
<td>d</td>
</tr>
</tbody>
</table>

Query: reachable(@a,N)
Implicit Communication

- A networking language with no explicit communication:

\[ \text{R2: } \text{reachable}(\text{@S}, \text{D}) \leftarrow \text{link}(\text{@S}, \text{Z}), \text{reachable}(\text{@Z}, \text{D}) \]

Data placement induces communication
Path Vector Protocol Example

- Advertisement: entire path to a destination
- Each node receives advertisement, adds itself to path and forwards to neighbors
Path Vector Protocol Example

- Advertisement: entire path to a destination
- Each node receives advertisement, adds itself to path and forwards to neighbors

```
path=[c,d]
c advertises [c,d]
```
Path Vector Protocol Example

- Advertisement: entire path to a destination
- Each node receives advertisement, adds itself to path and forwards to neighbors
Path Vector Protocol Example

- Advertisement: entire path to a destination
- Each node receives advertisement, adds itself to path and forwards to neighbors
Path Vector in Network Datalog

R1: \( \text{path}(@S,D,P) \leftarrow \text{link}(@S,D), P=(S,D). \)

R2: \( \text{path}(@S,D,P) \leftarrow \text{link}(@Z,S), \text{path}(@Z,D,P_2), P=S\cdot P_2. \)

query _(@S,D,P) \leftarrow \text{path}(@S,D,P)

- Input: link(@source, destination)
- Query output: path(@source, destination, pathVector)

Courtesy of Bill Marczak (UC Berkeley)
Path Vector in Network Datalog

\[
\begin{align*}
\text{R1: } & \text{path}(\text{@S}, \text{D}, P) \leftarrow \text{link}(\text{@S}, \text{D}), \quad P = (\text{S}, \text{D}). \\
\text{R2: } & \text{path}(\text{@S}, \text{D}, P) \leftarrow \text{link}(\text{@Z}, \text{S}), \quad \text{path}(\text{@Z}, \text{D}, P_2), \quad P = \text{S} \bullet P_2. \\
& \text{query } (\text{@S}, \text{D}, P) \leftarrow \text{path}(\text{@S}, \text{D}, P)
\end{align*}
\]

- Input: link(@source, destination)
- Query output: path(@source, destination, pathVector)

Courtesy of Bill Marczak (UC Berkeley)
Path Vector in Network Datalog

R1: path(@S,D,P) <- link(@S,D), P=(S,D).
R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P₂), P=S•P₂.
query _(@S,D,P) <- path(@S,D,P)  Add S to front of P₂

Input: link(@source, destination)
Query output: path(@source, destination, pathVector)

Courtesy of Bill Marczak (UC Berkeley)
Query Execution

R1: path(@S,D,P) <- link(@S,D), P=(S,D).
R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P₂), P=S•P₂.
query _(@a,d,P) <- path(@a,d,P)

Neighbor table:

```
link
@S D
@a b
```

```
link
@S D
@b c
@b a
```

```
link
@S D
@c b
@c d
```

```
link
@S D
@d c
```

Forwarding table:

```
path
@S D P
```

```
path
@S D P
```

```
path
@S D P
```
Query Execution

R1: \( \text{path}(\@S, D, P) \leftarrow \text{link}(\@S, D), \, P=(S, D). \)

R2: \( \text{path}(\@S, D, P) \leftarrow \text{link}(\@Z, S), \, \text{path}(\@Z, D, P_2), \)
\( \text{query} \, (\@a, d, P) \leftarrow \text{path}(\@a, d, P) \)
\( P=S\bullet P_2. \)
Query Execution

R1: \( \text{path}(\@S, \@D, \@P) \leftarrow \text{link}(\@S, \@D), \@P = (\@S, \@D) \).

R2: \( \text{path}(\@S, \@D, \@P) \leftarrow \text{link}(\@Z, \@S), \text{path}(\@Z, \@D, \@P_2), \text{query}(\@a, \@d, \@P) \leftarrow \text{path}(\@a, \@d, \@P) \).
Query Execution

R1: \( \text{path}(\text{@S,D,P}) \leftarrow \text{link}(\text{@S,D}), \text{P}=(\text{S,D}). \)

R2: \( \text{path}(\text{@S,D,P}) \leftarrow \text{link}(\text{@Z,S}), \text{path}(\text{@Z,D,P_2}), \text{P}=(\text{S} \bullet \text{P_2}). \)

query \( \_=(\text{@a,d,P}) \leftarrow \text{path}(\text{@a,d,P}) \)

Matching variable \( Z = \text{“Join”} \)

Neighbor table:

Forwarding table:
Query Execution

R1: \( \text{path}(\text{@S}, \text{D}, \text{P}) \leftarrow \text{link}(\text{@S}, \text{D}), \text{P}=(\text{S}, \text{D}). \)

R2: \( \text{path}(\text{@S}, \text{D}, \text{P}) \leftarrow \text{link}(\text{@Z}, \text{S}), \text{path}(\text{@Z}, \text{D}, \text{P}_2), \text{P}=(\text{S} \bullet \text{P}_2). \)

query \( \text{_}(\text{@a}, \text{d}, \text{P}) \leftarrow \text{path}(\text{@a}, \text{d}, \text{P}) \)

Matching variable \( \text{Z} = \text{“Join”} \)

Neighbor table:

<table>
<thead>
<tr>
<th>Link</th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>@b</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>@c</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link</th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>@b</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>@c</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link</th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>@b</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>@c</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

Forwarding table:

<table>
<thead>
<tr>
<th>Path</th>
<th>@S</th>
<th>D</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>@S</td>
<td>D</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>@S</td>
<td>D</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>@c</td>
<td>d</td>
<td>[c,d]</td>
<td></td>
</tr>
</tbody>
</table>
Query Execution

R1: \( \text{path}(\text{@S}, \text{D}, P) \leftarrow \text{link}(\text{@S}, \text{D}), P=(\text{S}, \text{D}). \)

R2: \( \text{path}(\text{@S}, \text{D}, P) \leftarrow \text{link}(\text{@Z}, \text{S}), \text{path}(\text{@Z}, \text{D}, P_2), P=\text{S}\cdot P_2. \)

query \( \text{(_(@a, d, P))} \leftarrow \text{path}(\text{@a}, \text{d}, P) \) Matching variable \( Z = \text{“Join”} \)

Neighbor table:

Forwarding table:
Query Execution

R1: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P₂), P=S•P₂.
query _(@a,d,P) <- path(@a,d,P)  Matching variable Z = “Join”

Neighbor table:

<table>
<thead>
<tr>
<th></th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>@b</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@c</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>@c</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@d</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

Forwarding table:

<table>
<thead>
<tr>
<th></th>
<th>@S</th>
<th>D</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>@b</td>
<td>d</td>
<td>[b,c,d]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>@S</th>
<th>D</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>@c</td>
<td>d</td>
<td>[c,d]</td>
<td></td>
</tr>
</tbody>
</table>

404
Query Execution

R1: \text{path}(\text{@S,D,P}) \leftarrow \text{link}(\text{@S,D}), \ P=(\text{S,D}).

R2: \text{path}(\text{@S,D,P}) \leftarrow \text{link}(\text{@Z,S}), \ \text{path}(\text{@Z,D,P}_2), \ P=S\cdot P_2.

\text{query} \ _{\text{Z}}(\text{@a,d,P}) \leftarrow \text{path}(\text{@a,d,P}) \quad \text{Matching variable Z = “Join”}

Neighbor table:

<table>
<thead>
<tr>
<th>link</th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link</th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>@b</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link</th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@c</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>@c</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link</th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>d</td>
<td>[a,b,c,d]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link</th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@b</td>
<td>d</td>
<td>[b,c,d]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>link</th>
<th>@S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>@c</td>
<td>d</td>
<td>[c,d]</td>
</tr>
</tbody>
</table>

Forwarding table:

<table>
<thead>
<tr>
<th>@S</th>
<th>D</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>d</td>
<td>[a,b,c,d]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>@S</th>
<th>D</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>@b</td>
<td>d</td>
<td>[b,c,d]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>@S</th>
<th>D</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>@c</td>
<td>d</td>
<td>[c,d]</td>
</tr>
</tbody>
</table>
Query Execution

R1: \text{path}(\text{@S}, \text{D}, P) \leftarrow \text{link}(\text{@S}, \text{D}), P=(\text{S}, \text{D}).

R2: \text{path}(\text{@S}, \text{D}, P) \leftarrow \text{link}(\text{@Z}, \text{S}), \text{path}(\text{@Z}, \text{D}, P_2), P=\text{S} \bullet P_2.

\text{query } _{(\text{@a}, \text{d}, P) \leftarrow \text{path}(\text{@a}, \text{d}, P) \text{ Matching variable } Z = \text{“Join”}}

Communication patterns are identical to those in the actual path vector protocol

Forwarding table:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@S</td>
<td>D</td>
<td>P</td>
</tr>
<tr>
<td>@a</td>
<td>d</td>
<td>[a,b,c,d]</td>
</tr>
<tr>
<td>@b</td>
<td>d</td>
<td>[b,c,d]</td>
</tr>
<tr>
<td>@c</td>
<td>d</td>
<td>[c,d]</td>
</tr>
</tbody>
</table>

Matching variable Z = “Join”
All-pairs Shortest-path

R1: path(@S,D,P,C) <- link(@S,D,C), P=(S,D).
R2: path(@S,D,P,C) <- link(@S,Z,C_1), path(@Z,D,P_2,C_2), C=C_1+C_2, P=S \bullet P_2.
All-pairs Shortest-path

R1: path(@S,D,P,C) <- link(@S,D,C), P=(S,D).
R2: path(@S,D,P,C) <- link(@S,Z,C_1), path(@Z,D,P_2,C_2), C=C_1+C_2, P=S\mathbin{\bullet}P_2.
R3: bestPathCost(@S,D,min<C>) <- path(@S,D,P,C).
R4: bestPath(@S,D,P,C) <- bestPathCost(@S,D,C), path(@S,D,P,C). query_(@S,D,P,C) <- bestPath(@S,D,P,C)
Distributed Semi-naïve Evaluation

• Semi-naïve evaluation:
  – Iterations (rounds) of synchronous computation
  – Results from iteration \( i^{th} \) used in \( (i+1)^{th} \)

<table>
<thead>
<tr>
<th>Link Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Path Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

1-hop

Network
Distributed Semi-naïve Evaluation

- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration $i^{th}$ used in $(i+1)^{th}$
Distributed Semi-naïve Evaluation

- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration $i^{th}$ used in $(i+1)^{th}$
Distributed Semi-naïve Evaluation

• Semi-naïve evaluation:
  – Iterations (rounds) of synchronous computation
  – Results from iteration $i^{th}$ used in $(i+1)^{th}$

![Diagram of network and tables]

Link Table
Path Table
Network
Distributed Semi-naïve Evaluation

- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration $i^{th}$ used in $(i+1)^{th}$

Link Table

Path Table

Network
Distributed Semi-naïve Evaluation

• Semi-naïve evaluation:
  – Iterations (rounds) of synchronous computation
  – Results from iteration $i^{th}$ used in $(i+1)^{th}$

Link Table

Path Table

Network
Distributed Semi-naïve Evaluation

• Semi-naïve evaluation:
  – Iterations (rounds) of synchronous computation
  – Results from iteration $i^{th}$ used in $(i+1)^{th}$

![Diagram with Link Table, Path Table, and Network]
Distributed Semi-naïve Evaluation

- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration $i^{th}$ used in $(i+1)^{th}$

Problem: How do nodes know that an iteration is completed? Unpredictable delays and failures make synchronization difficult/expensive.
Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows
Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows

![Diagram of Link Table, Path Table, and Network]
Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows
Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows

Link Table

Path Table

Network
Pipelined Semi-naïve (PSN)

• Fully-asynchronous evaluation:
  – Computed tuples in *any* iteration are pipelined to next iteration
  – Natural for distributed dataflows

Link Table

Path Table

Network
Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in any iteration are pipelined to next iteration
  - Natural for distributed dataflows

![Diagram](image_url)
Nodes in dataflow graph (“elements”):
- Network elements (send/recv, rate limitation, jitter)
- Flow elements (mux, demux, queues)
- Relational operators (selects, projects, joins, aggregates)
Dataflow Graph

Nodes in dataflow graph ("elements"):
- Network elements (send/recv, rate limitation, jitter)
- Flow elements (mux, demux, queues)
- Relational operators (selects, projects, joins, aggregates)
Nodes in dataflow graph ("elements"):

- Network elements (send/recv, rate limitation, jitter)
- Flow elements (mux, demux, queues)
- Relational operators (selects, projects, joins, aggregates)
Rule → Dataflow “Strands”

R2: path(@S,D,P) <- link(@S,Z), path(@Z,D,P₂), P=S • P₂.
Rule → Dataflow “Strands”
Localization Rewrite

• Rules may have body predicates at different locations:

R2: \text{path(@S,D,P)} \leftarrow \text{link(@S,Z), path(@Z,D,P_2), P=S\cdot P_2.}

Matching variable \text{Z = “Join”}
Localization Rewrite

• Rules may have body predicates at different locations:

R2: path(@S,D,P) ← link(@S,Z), path(@Z,D,P₂), P=S•P₂.

Matching variable Z = “Join”

Rewritten rules:

R2a: linkD(S,@D) ← link(@S,D)

R2b: path(@S,D,P) ← linkD(S,@Z), path(@Z,D,P₂), P=S•P₂.
Localization Rewrite

• Rules may have body predicates at different locations:

R2: path(@S,D,P) ← link(@S,Z), path(@Z,D,P_2), P=S•P_2.

Matching variable Z = “Join”

Rewritten rules:

R2a: linkD(S,@D) ← link(@S,D)

R2b: path(@S,D,P) ← linkD(S,@Z), path(@Z,D,P_2), P=S•P_2.
Localization Rewrite

• Rules may have body predicates at different locations:

R2: path(@S,D,P) <- link(@S,Z), path(@Z,D,P₂), P=S•P₂.

Rewritten rules:

R2a: linkD(S,@D) <- link(@S,D)

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P₂), P=S•P₂.
Localization Rewrite

• Rules may have body predicates at different locations:

R2: \( \text{path}(\@S, D, P) \leftarrow \text{link}(\@S, Z), \text{path}(\@Z, D, P_2), P=S\bullet P_2. \)

Matching variable \( Z = “Join” \)

Rewritten rules:

R2a: \( \text{linkD}(S, @D) \leftarrow \text{link}(@S, D) \)

R2b: \( \text{path}(@S, D, P) \leftarrow \text{linkD}(S, @Z), \text{path}(@Z, D, P_2), P=S\bullet P_2. \)

Matching variable \( Z = “Join” \)
Physical Execution Plan

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P₂), P=S•P₂.
Physical Execution Plan

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P₂), P=S\cdot P₂.
R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P₂), P=S\join P₂.
R2b: \( \text{path}(\ast S, D, P) \leftarrow \text{linkD}(S, \ast Z), \text{path}(\ast Z, D, P_2), P = S \bullet P_2. \)

Physical Execution Plan

Strand Elements

Network In

\( \Delta \text{path} \)

Join

\( \text{path}.Z = \text{linkD}.Z \)

Project

\( \text{path}(S, D, P) \)

linkD

Network In
Physical Execution Plan

$$R2b:\ \text{path}(\@S,D,P) \leftarrow \text{linkD}(S,\@Z), \text{path}(\@Z,D,P_2), \ P=S\bullet P_2.$$
Physical Execution Plan

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P), P=S\cdot P_2.
Pipelined Evaluation

• Challenges:
  – Does PSN produce the correct answer?
  – Is PSN bandwidth efficient?
    • I.e. does it make the minimum number of inferences?
Pipelined Evaluation

• Challenges:
  – Does PSN produce the correct answer?
  – Is PSN bandwidth efficient?
    • I.e. does it make the minimum number of inferences?

• Theorems [SIGMOD’06]:
  – $RS_{SN}(p) = RS_{PSN}(p)$, where RS is results set
  – No repeated inferences in computing $RS_{PSN}(p)$
  – Require per-tuple timestamps in delta rules and FIFO and reliable channels
Incremental View Maintenance

• Leverages insertion and deletion delta rules for state modifications.
• Complications arise from duplicate evaluations.
• Consider the Reachable query. What if there are many ways to route between two nodes a and b, i.e. many possible derivations for reachable(a,b)?
Incremental View Maintenance

• Leverages insertion and deletion delta rules for state modifications.
• Complications arise from duplicate evaluations.
• Consider the Reachable query. What if there are many ways to route between two nodes a and b, i.e. many possible derivations for reachable(a,b)?
• Mechanisms: still use delta rules, but additionally, apply
  – Count algorithm (for non-recursive queries).
  – Delete and Rederive (SIGMOD’93). Expensive in distributed settings.

Recent PSN Enhancements

- Provenance-based approach
  - Condensed form of provenance piggy-backed with each tuple for derivability test.

- Relaxation of FIFO requirements:
Optimizations

• Traditional:
  – Aggregate Selections
  – Magic Sets rewrite
  – Predicate Reordering
Optimizations

• Traditional:
  – Aggregate Selections
  – Magic Sets rewrite
  – Predicate Reordering

\[
\text{PV/DV} \rightarrow \text{DSR}
\]
Optimizations

• **Traditional:**
  – Aggregate Selections
  – Magic Sets rewrite
  – Predicate Reordering

• **New:**
  – Multi-query optimizations:
    • Query Results caching
    • Opportunistic message sharing
  – Cost-based optimizations
    • Network statistics (e.g. density, route request rates, etc.)
    • Combining top-down and bottom-up evaluation

\[ \text{PV/DV} \to \text{DSR} \]
Suggested Readings

• Networking use cases:

• Distributed recursive query processing:
Challenges and Opportunities

• Declarative networking adoption:
  – Leverage well-known open-source software-based projects, e.g. ns-3, Quagga, OpenFlow
  – Wrappers for legacy code
  – Usability studies
  – Open-source code release and demonstrations

• Formal network verification:
  – Integration of formal tools (e.g. theorem provers, SMT solvers), formal network models (e.g. routing algebra)
  – Operational semantics of Network Datalog and subsequent extensions
  – Other properties: timing, security

• Opportunities for automated program synthesis
Outline of Tutorial

June 14, 2011: The Second Coming of Datalog!

• Refresher: basics of Datalog
• Application #1: Data Integration and Exchange
• Application #2: Program Analysis
• Application #3: Declarative Networking
• Modern System Implementations
• Open Questions
Outline of Tutorial

June 14, 2011: The Second Coming of Datalog!

• Refresher: basics of Datalog
• Application #1: Data Integration and Exchange
• Application #2: Program Analysis
• Application #3: Declarative Networking
• Conclusions
What Is A Program?

program = algorithms + data structures
What Is A Program?

\[ \text{program} = \text{algorithms} + \text{data structures} \]

\[ \text{algorithm} = \text{logic} + \text{control} \]
What Is A Program?

program = logic + control + data structures

An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used. The logic component determines the meaning of the algorithm whereas the control component only affects its efficiency. The efficiency of an algorithm can often be improved by improving the control component without changing the logic of the algorithm. We argue that computer programs would be more often correct and more easily improved and modified if their logic and control aspects were identified and separated in the program text.

Key Words and Phrases: control language, logic programming, nonprocedural language, programming methodology, program specification, relational data structures
Logic + Control + Data Structures

Specifications

Datalog Engine

Implementation

Control

Top-down

Bottom-up

Tabled

Naive

Semi-naive

Counting

DReD

Data Structures

transitive closure

BDDs

BTree

KDTree
THE END... OR IS IT THE BEGINNING?