Datalog and Emerging Applications: an Interactive Tutorial

Shan Shan Huang — LOGICBLOX®

T.J. Green — UCDAVIS UNIVERSITY OF CALIFORNIA

Boon Thau Loo — Penn

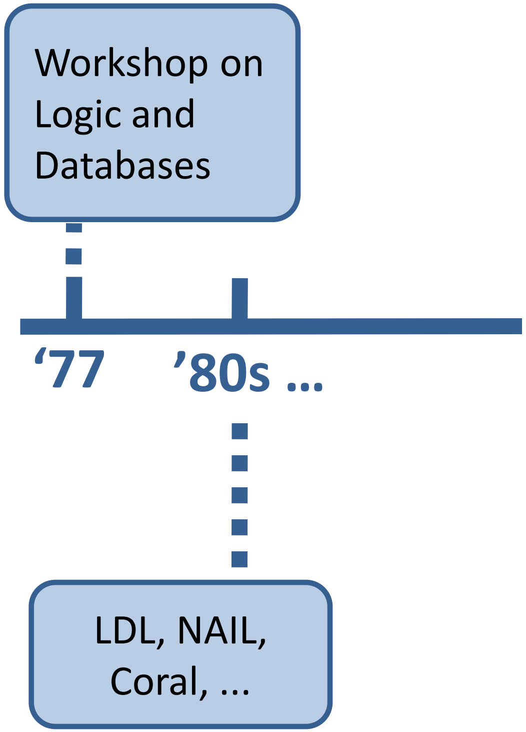
SIGMOD 2011 Athens, Greece June 14, 2011

A Brief History of Datalog

Workshop on Logic and Databases

‘77

A Brief History of Datalog



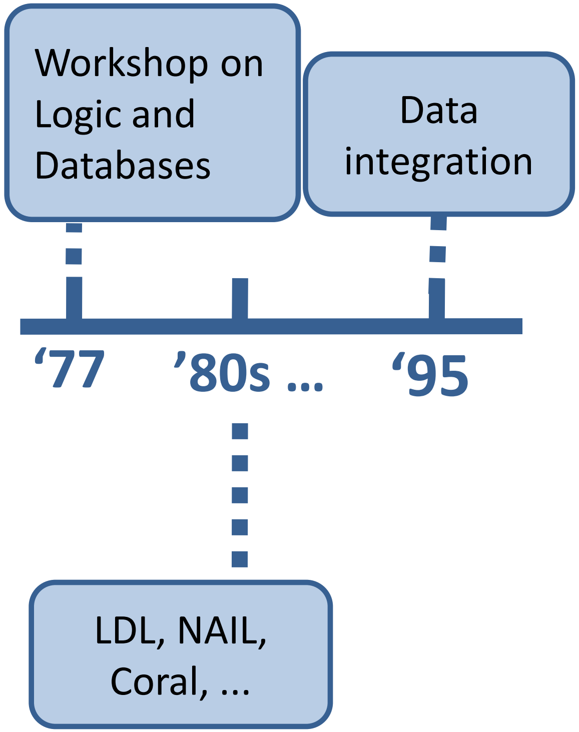
Workshop on Logic and Databases

‘77

’80s …

LDL, NAIL, Coral, ...

A Brief History of Datalog



Workshop on Logic and Databases

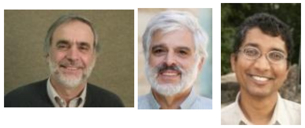
Data integration

‘77

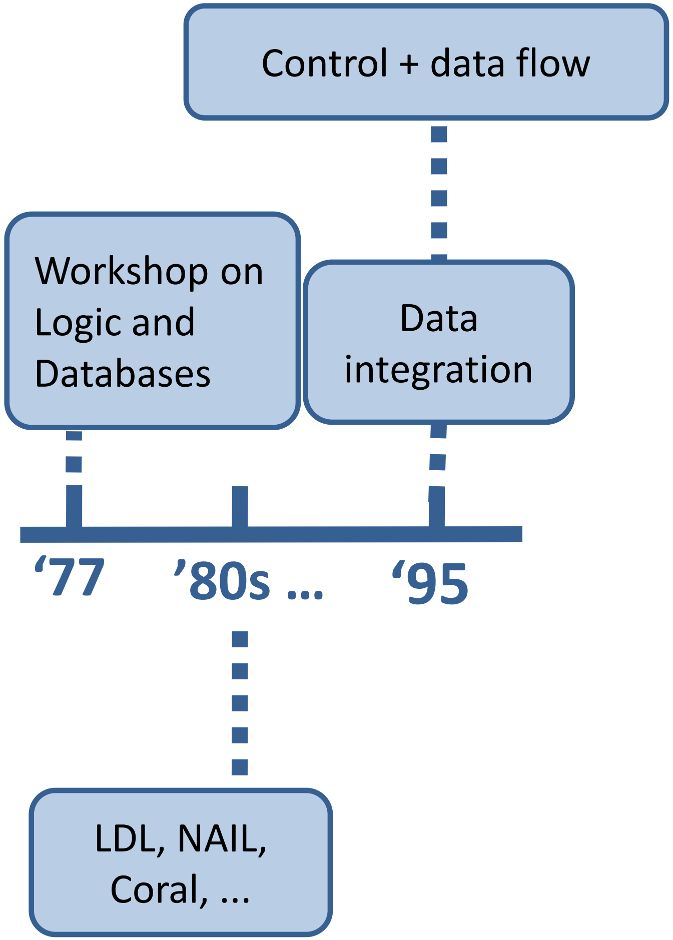
’80s …

‘95

LDL, NAIL, Coral, ...



A Brief History of Datalog



Control + data flow

Workshop on Logic and Databases

Data integration

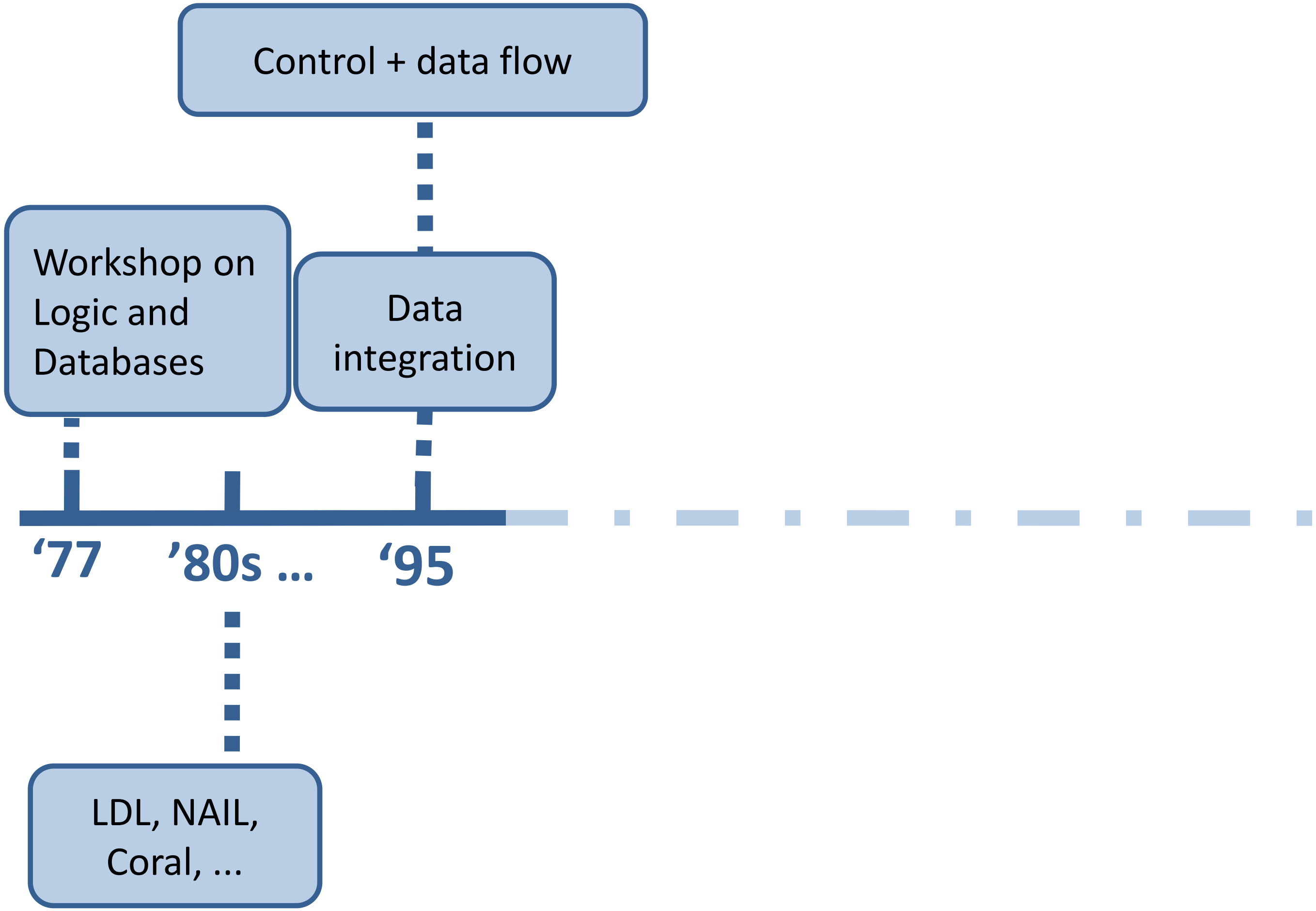
‘77

’80s …

‘95

LDL, NAIL, Coral, …

A Brief History of Datalog



Control + data flow

Workshop on Logic and Databases

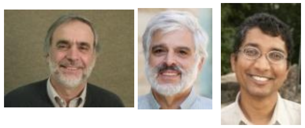
Data integration

‘77

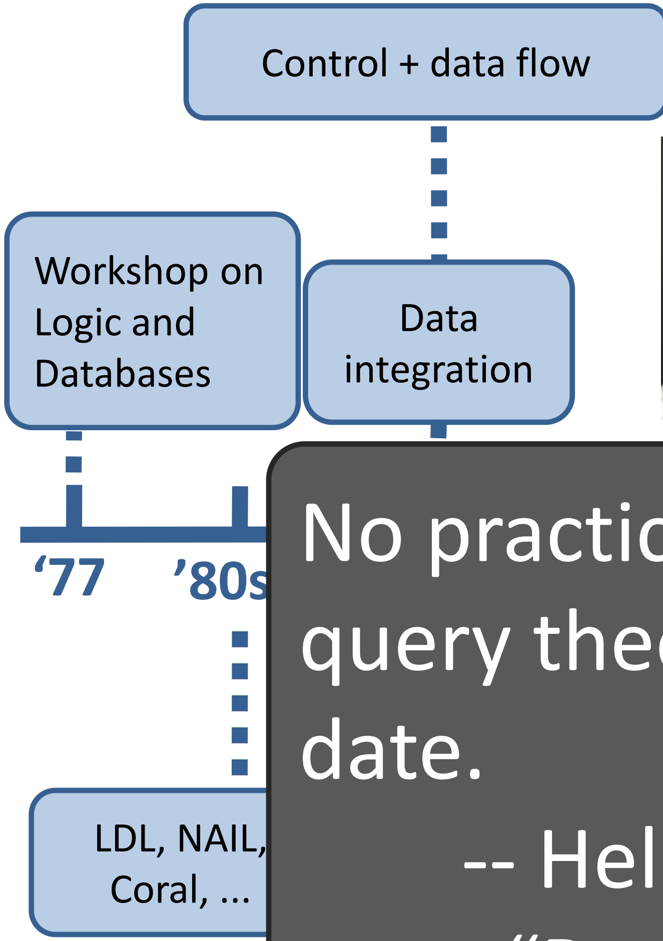
’80s …

‘95

LDL, NAIL, Coral, ...



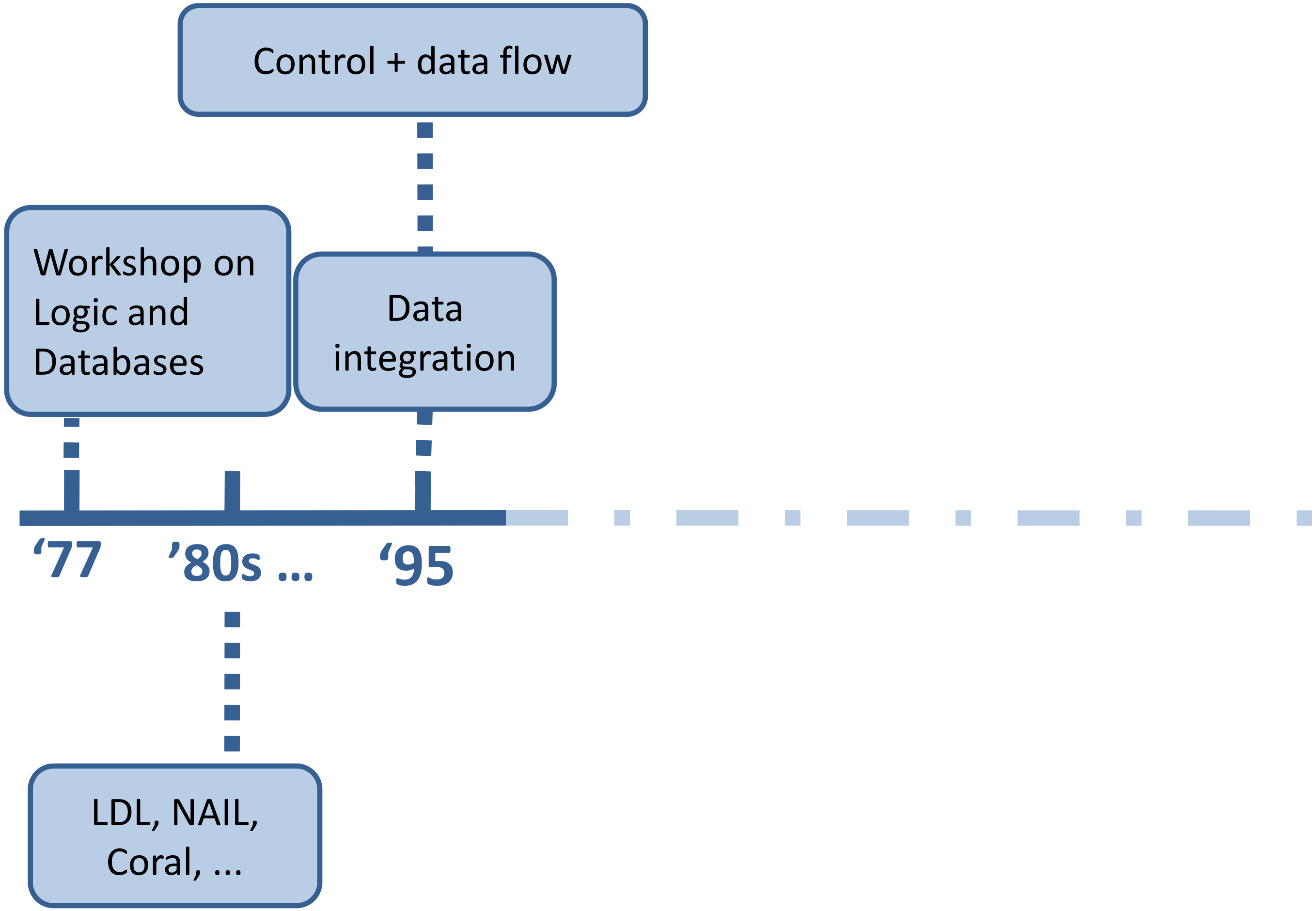
A Brief History of Datalog



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

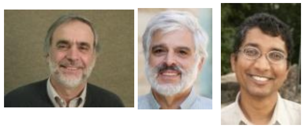
Data integration

‘77

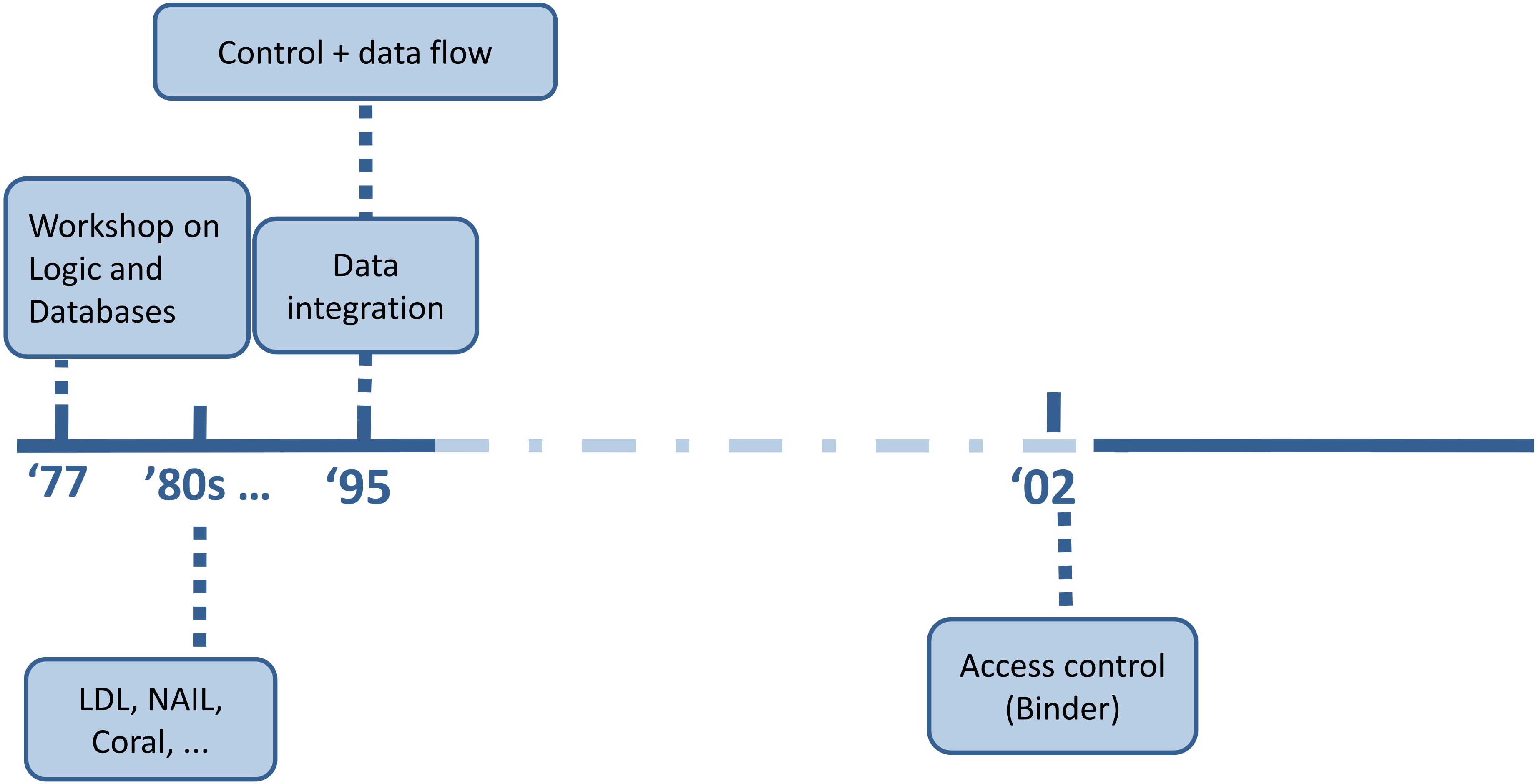
’80s …

‘95

LDL, NAIL, Coral, ...



A Brief History of Datalog



Control + data flow

Workshop on Logic and Databases

Data integration

‘77

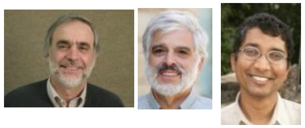
’80s …

‘95

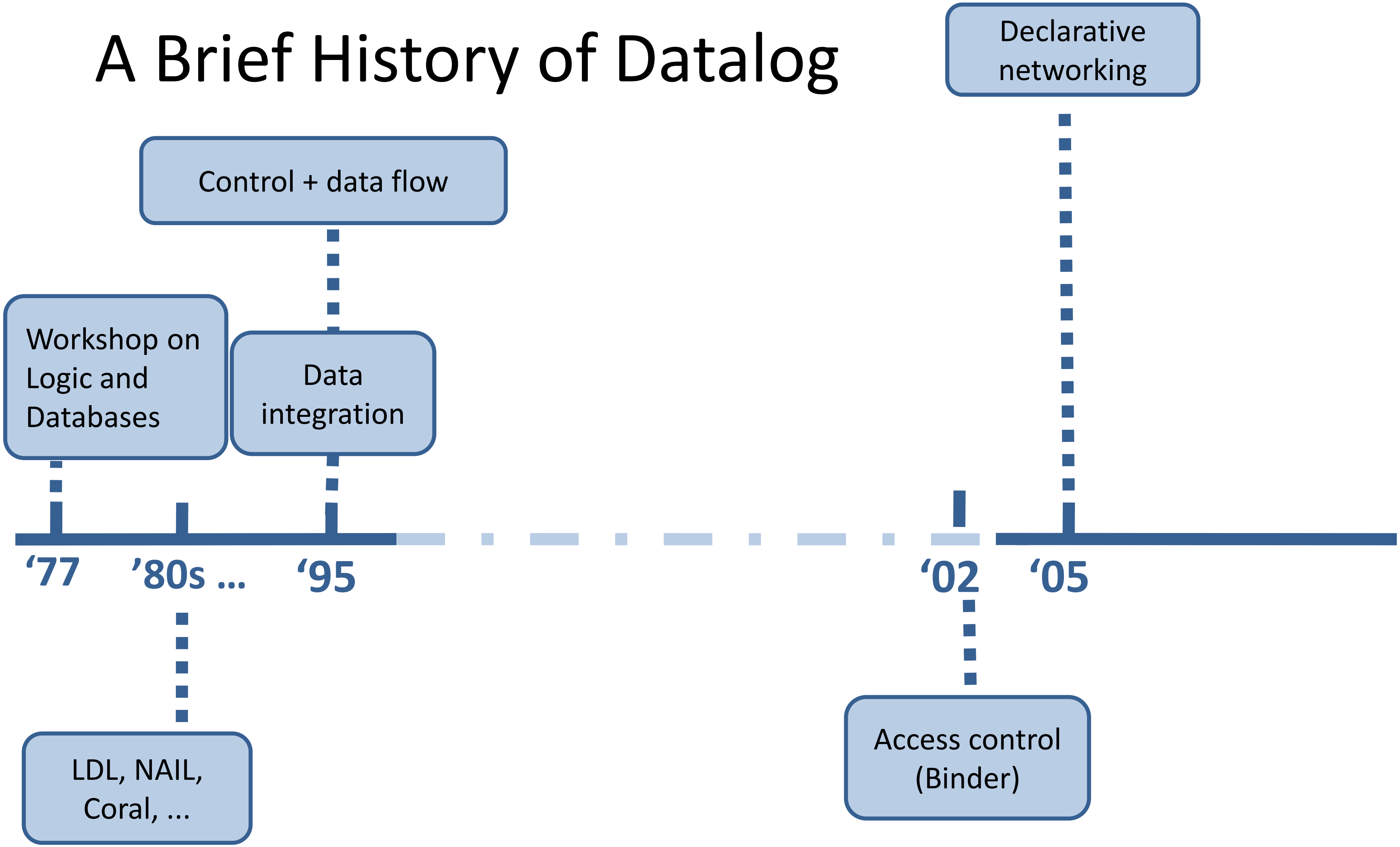
‘02

LDL, NAIL, Coral, …

Access control (Binder)



A Brief History of Datalog



‘77

Workshop on Logic and Databases

’80s …

LDL, NAIL, Coral, ...

‘95

Data integration

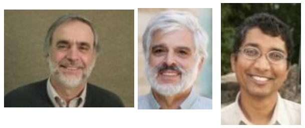
Control + data flow

‘02

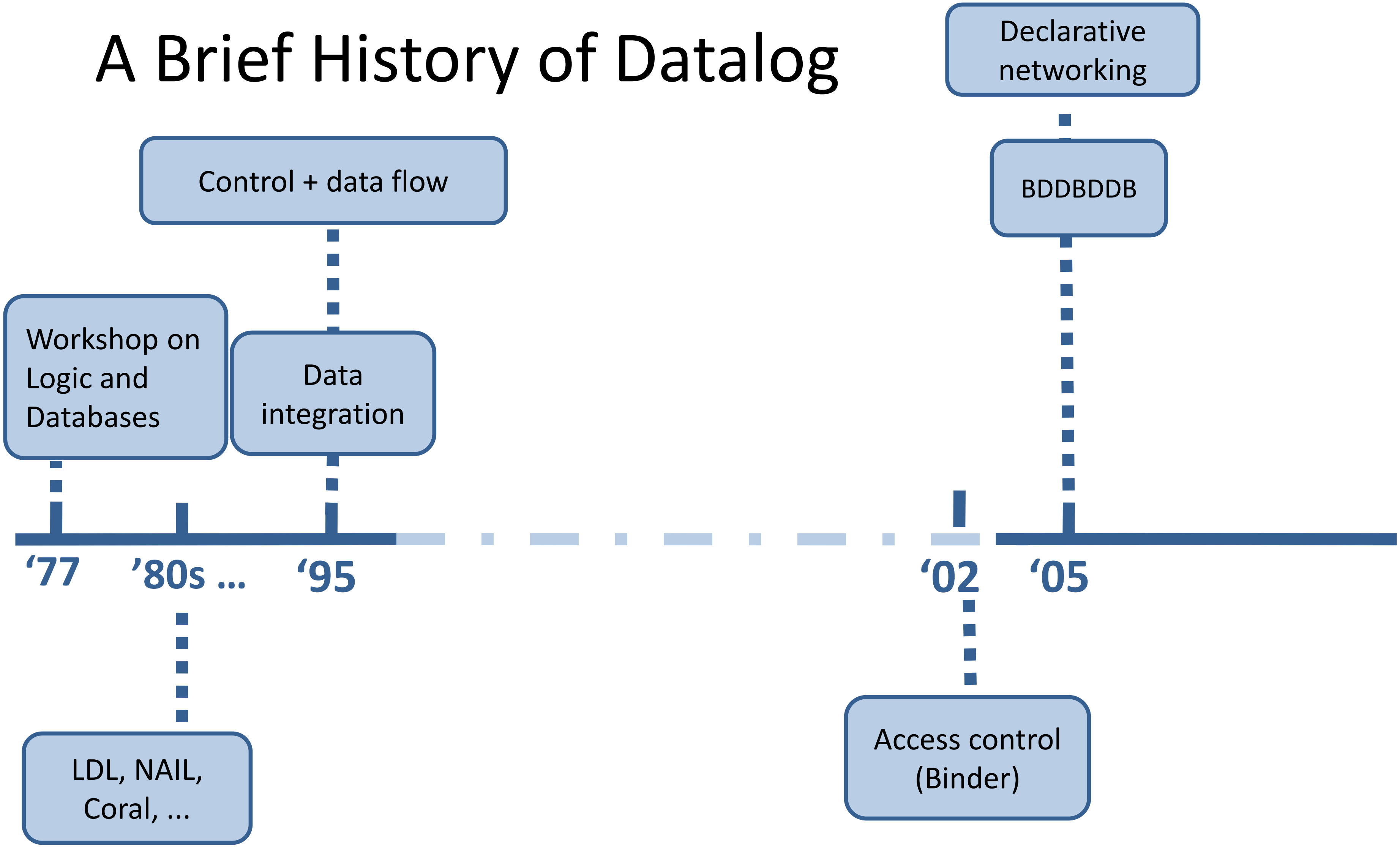
Access control (Binder)

‘05

Declarative networking



A Brief History of Datalog



'77

Workshop on Logic and Databases

'80s …

LDL, NAIL, Coral, ...

'95

Data integration

Control + data flow

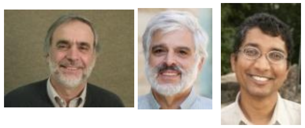
'02

Access control (Binder)

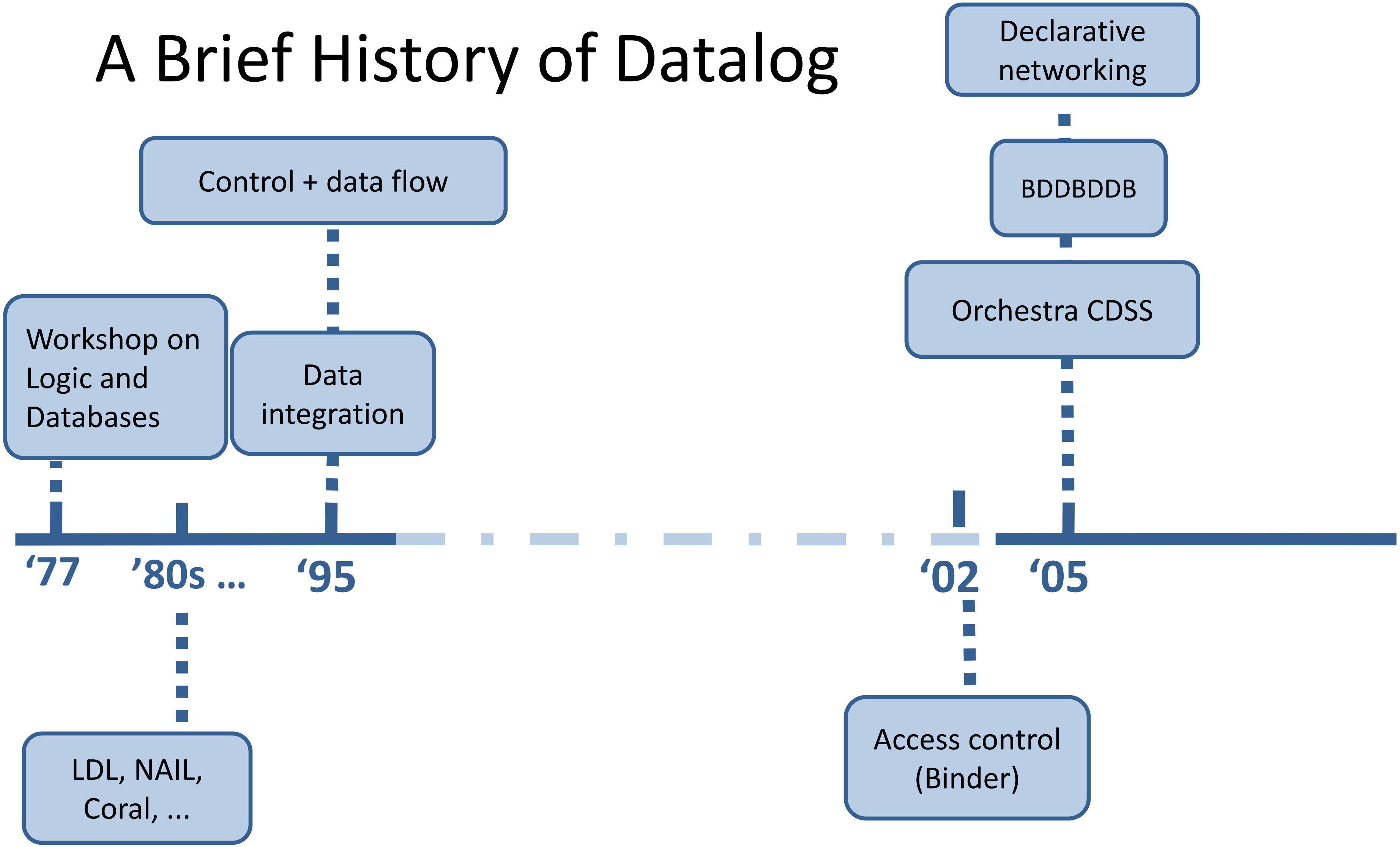
'05

BDDBDDB

Declarative networking



A Brief History of Datalog



Workshop on Logic and Databases

Control + data flow

Data integration

Declarative networking

BDDBDDB

Orchestra CDSS

‘77

’80s …

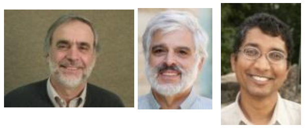
‘95

‘02

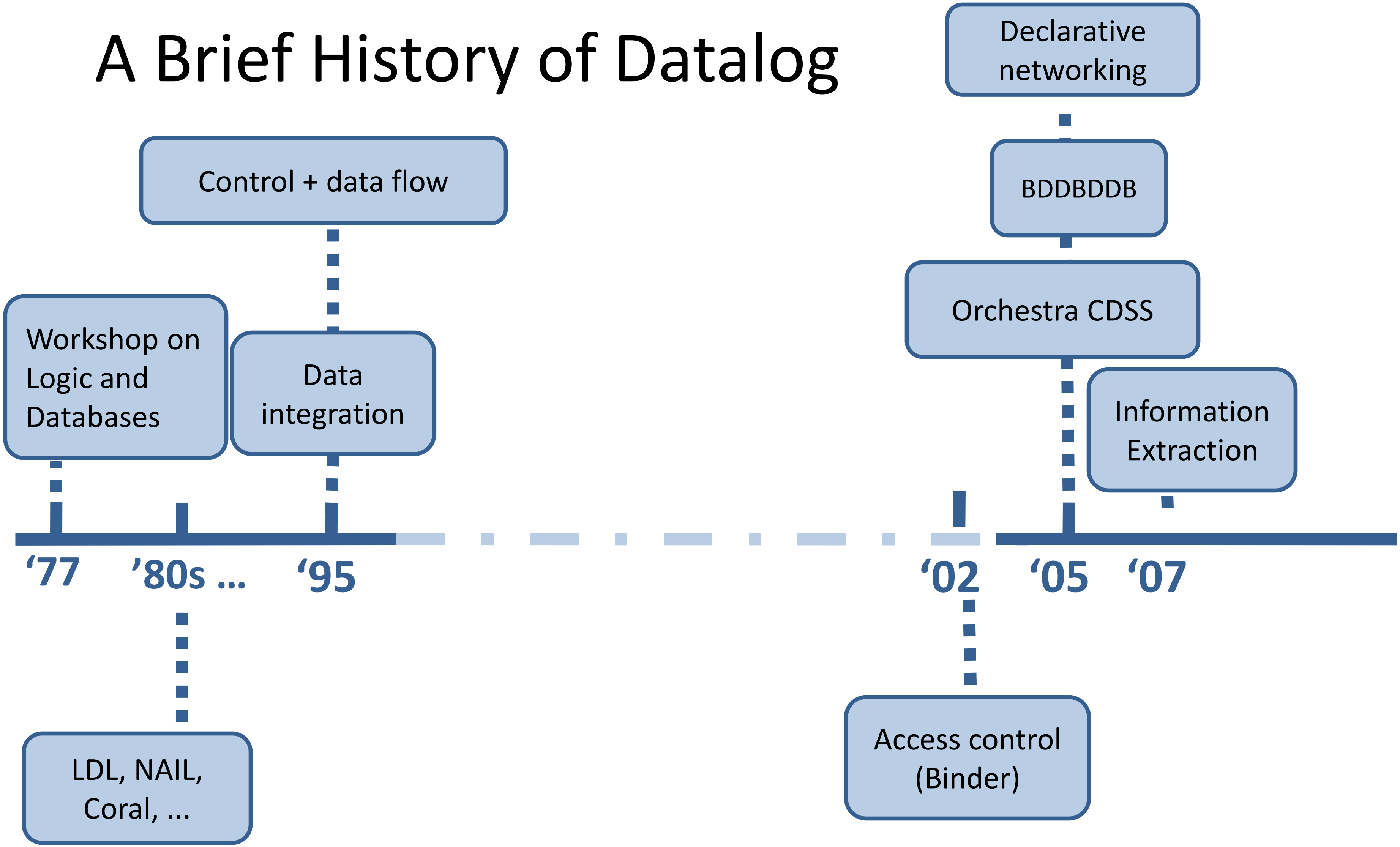
‘05

LDL, NAIL, Coral, …

Access control (Binder)



A Brief History of Datalog



Declarative networking

BDDBDDB

Control + data flow

Orchestra CDSS

Workshop on Logic and Databases

Data integration

Information Extraction

‘77

’80s …

‘95

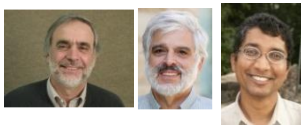
‘02

‘05

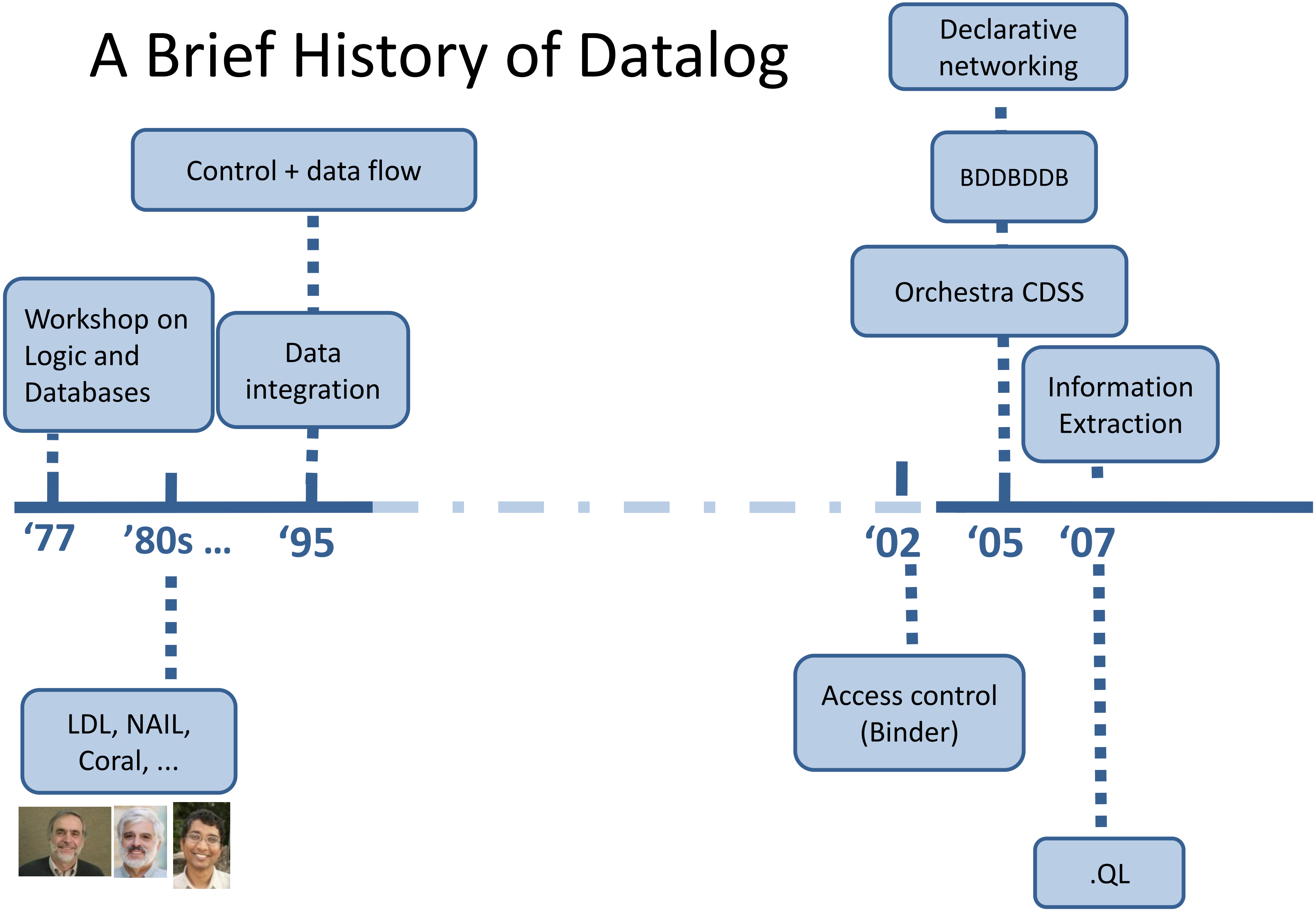
‘07

LDL, NAIL, Coral, ...

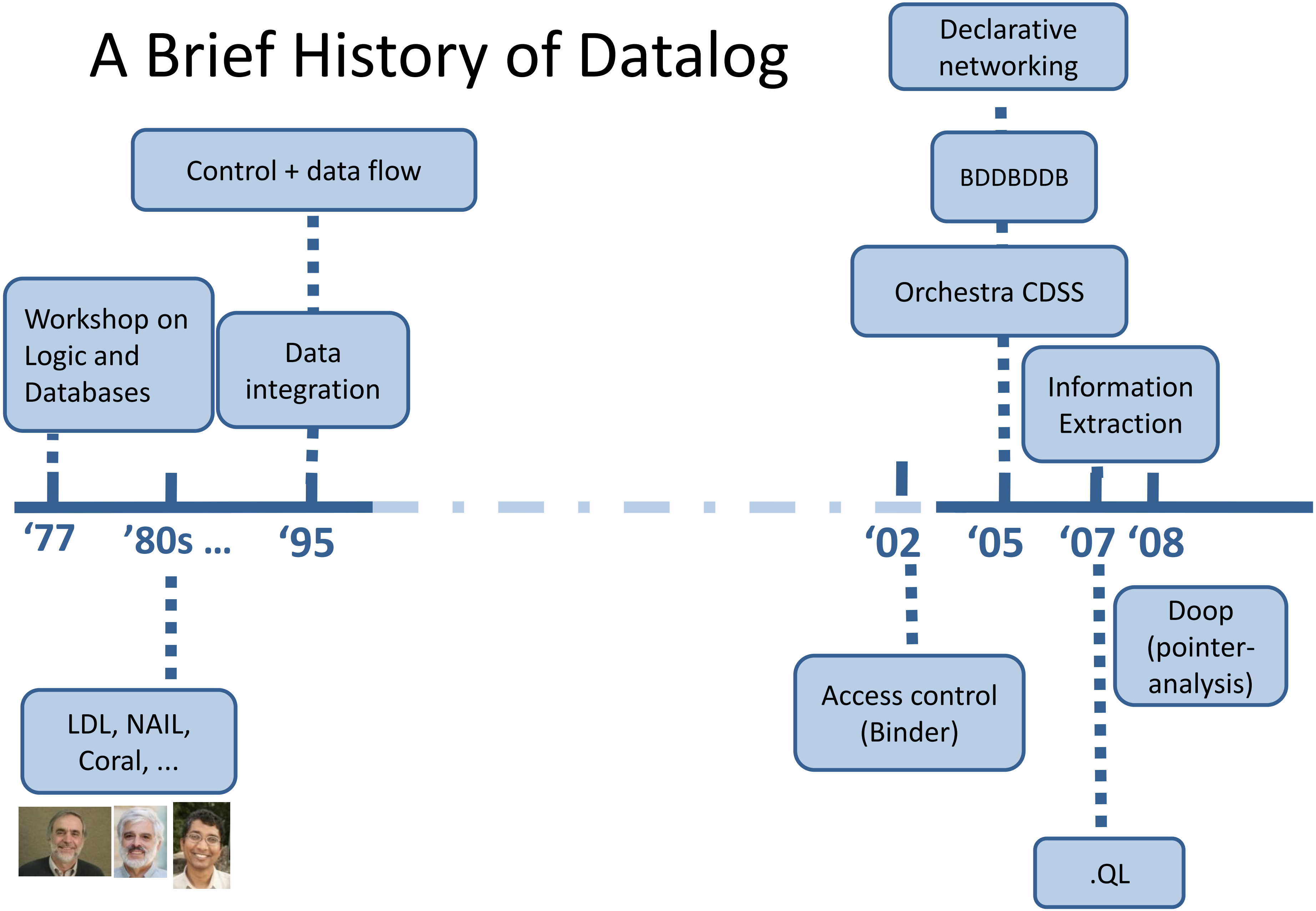
Access control (Binder)



A Brief History of Datalog



A Brief History of Datalog



‘77

Workshop on Logic and Databases

’80s …

LDL, NAIL, Coral, ...

‘95

Control + data flow

Data integration

‘02

Access control (Binder)

‘05

Declarative networking

BDDBDDB

Orchestra CDSS

‘07

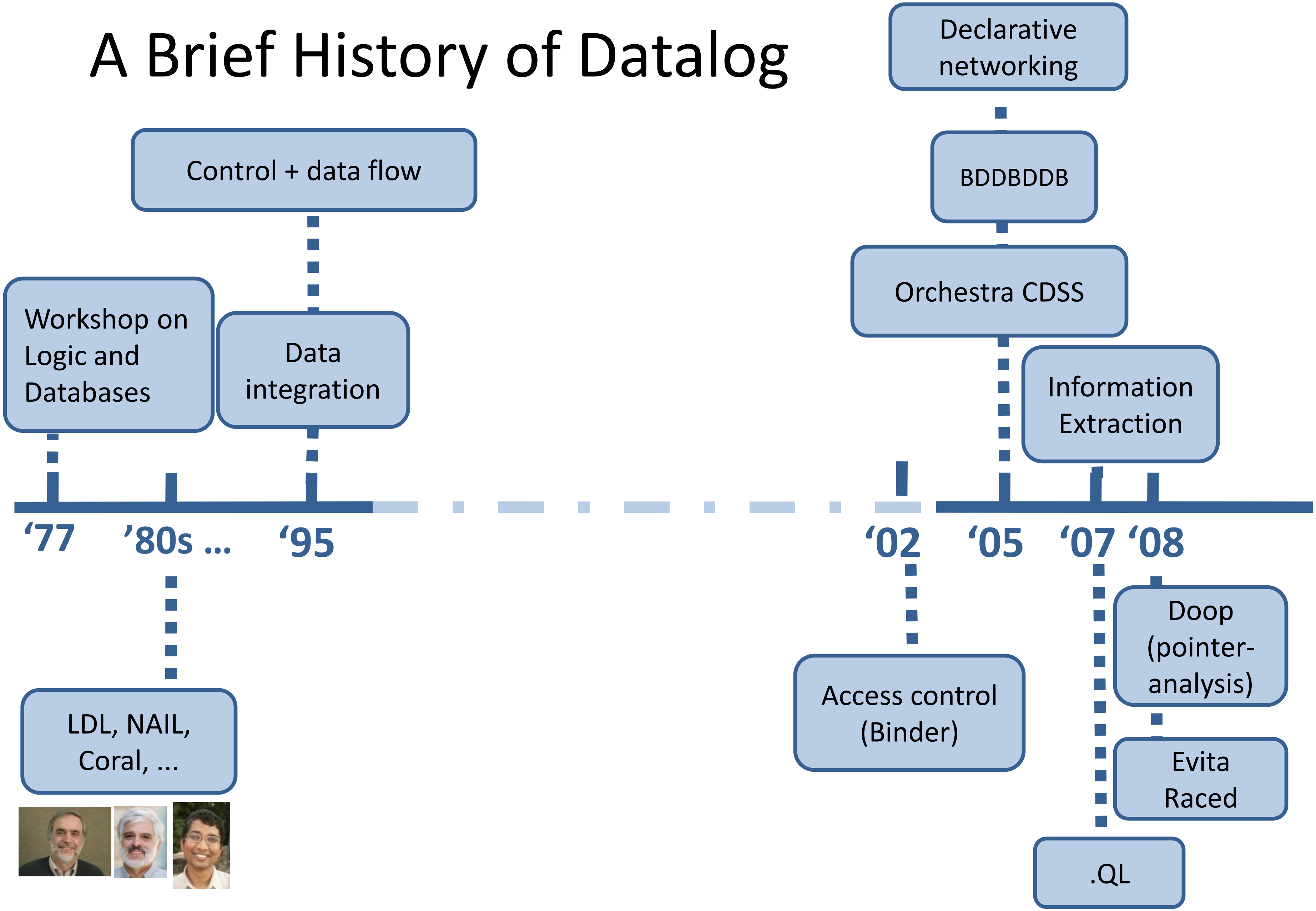
Information Extraction

.QL

‘08

Doop (pointer-analysis)

A Brief History of Datalog



Workshop on Logic and Databases

Control + data flow

Data integration

‘77

’80s …

‘95

LDL, NAIL, Coral, …

Declarative networking

BDDBDDB

Orchestra CDSS

Information Extraction

‘02

‘05

‘07

‘08

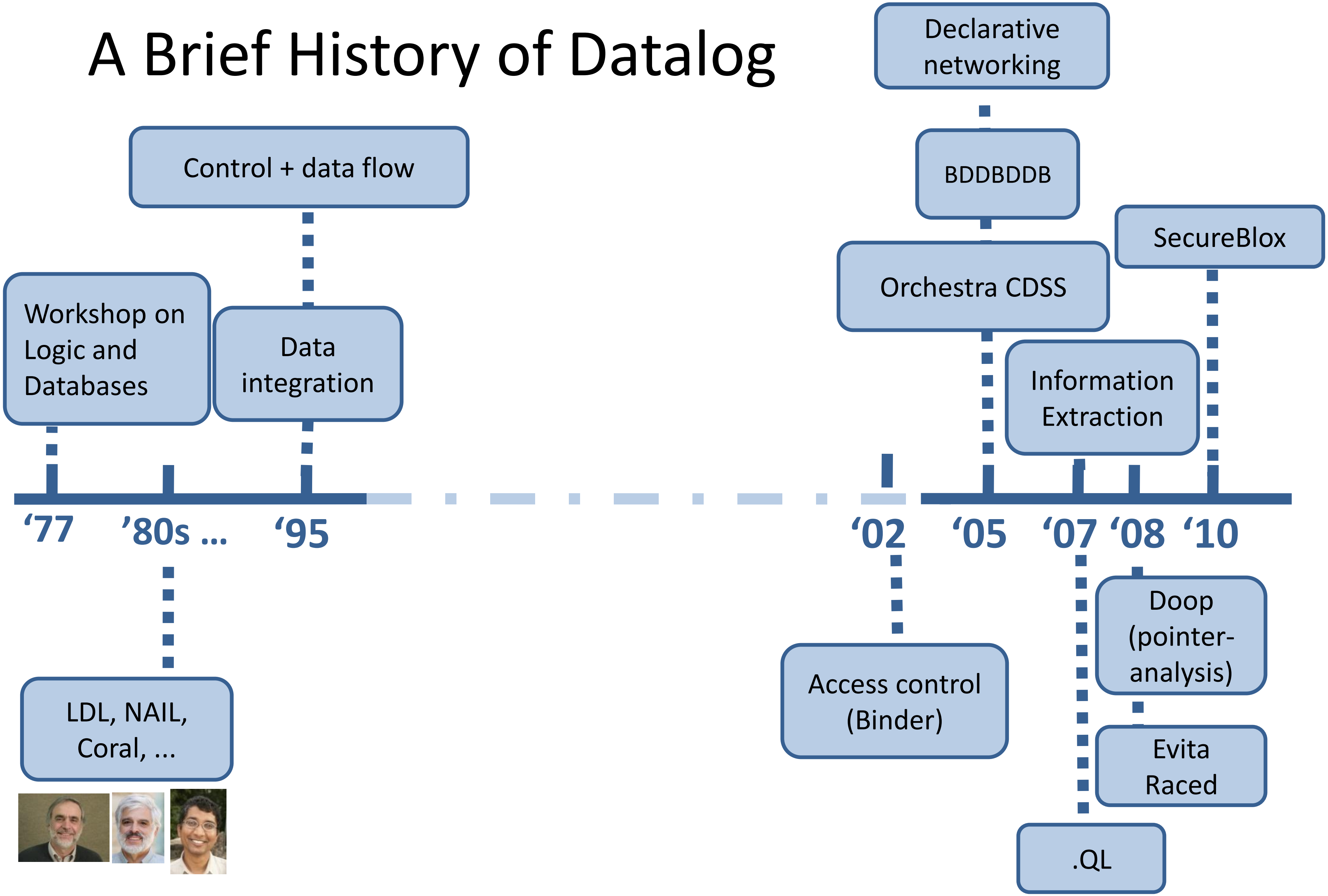
Access control (Binder)

Doop (pointer-analysis)

Evita Raced

.QL

A Brief History of Datalog



‘77

Workshop on Logic and Databases

’80s …

LDL, NAIL, Coral, ...

‘95

Control + data flow

Data integration

‘02

Access control (Binder)

‘05

Declarative networking

BDDBDDB

Orchestra CDSS

‘07

Information Extraction

.QL

‘08

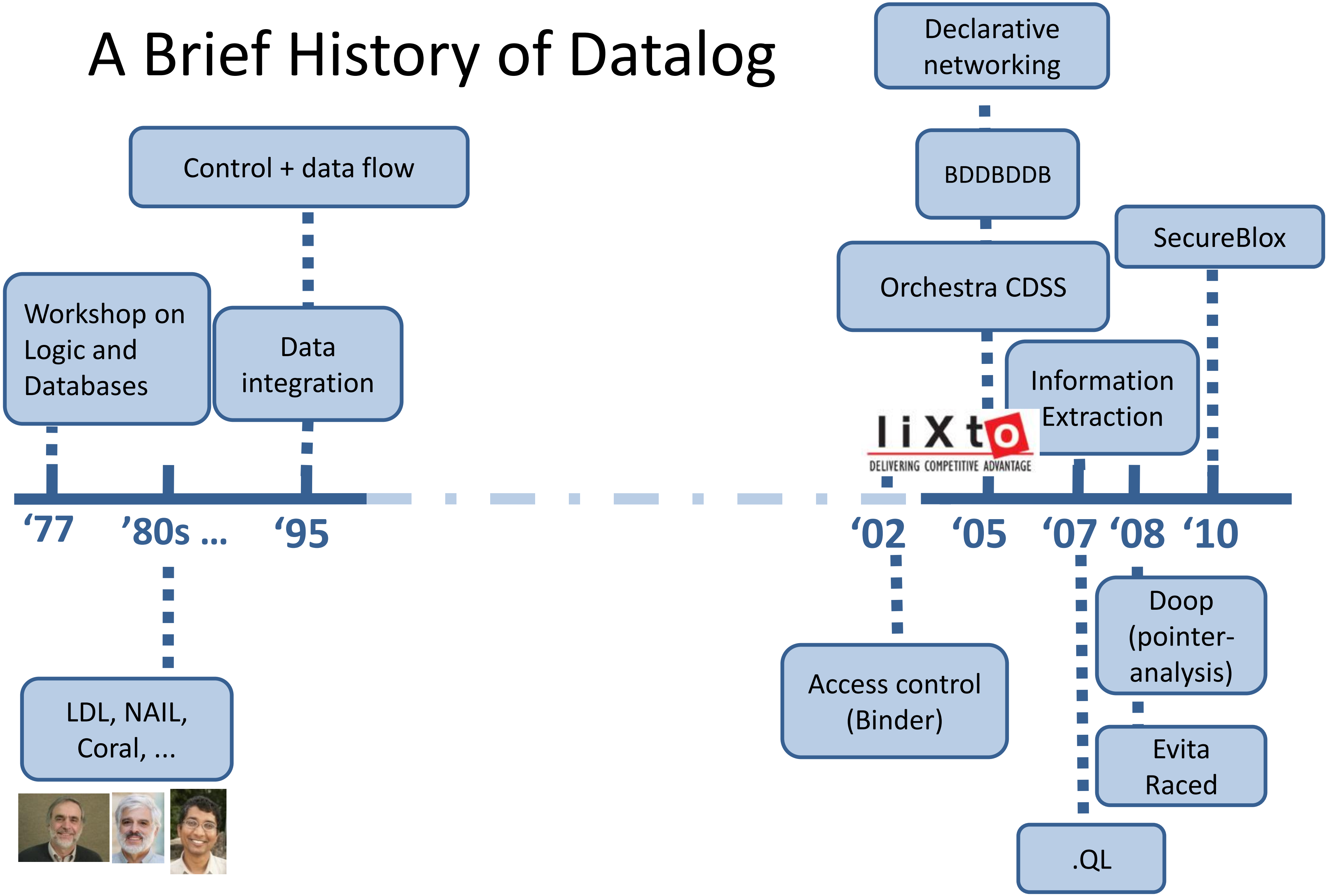
Doop (pointer-analysis)

Evita Raced

‘10

SecureBlox

A Brief History of Datalog



‘77

Workshop on Logic and Databases

’80s …

LDL, NAIL, Coral, ...

‘95

Data integration

Control + data flow

‘02

Access control (Binder)

Information Extraction

‘05

Orchestra CDSS

BDDBDDB

Declarative networking

‘07

.QL

‘08

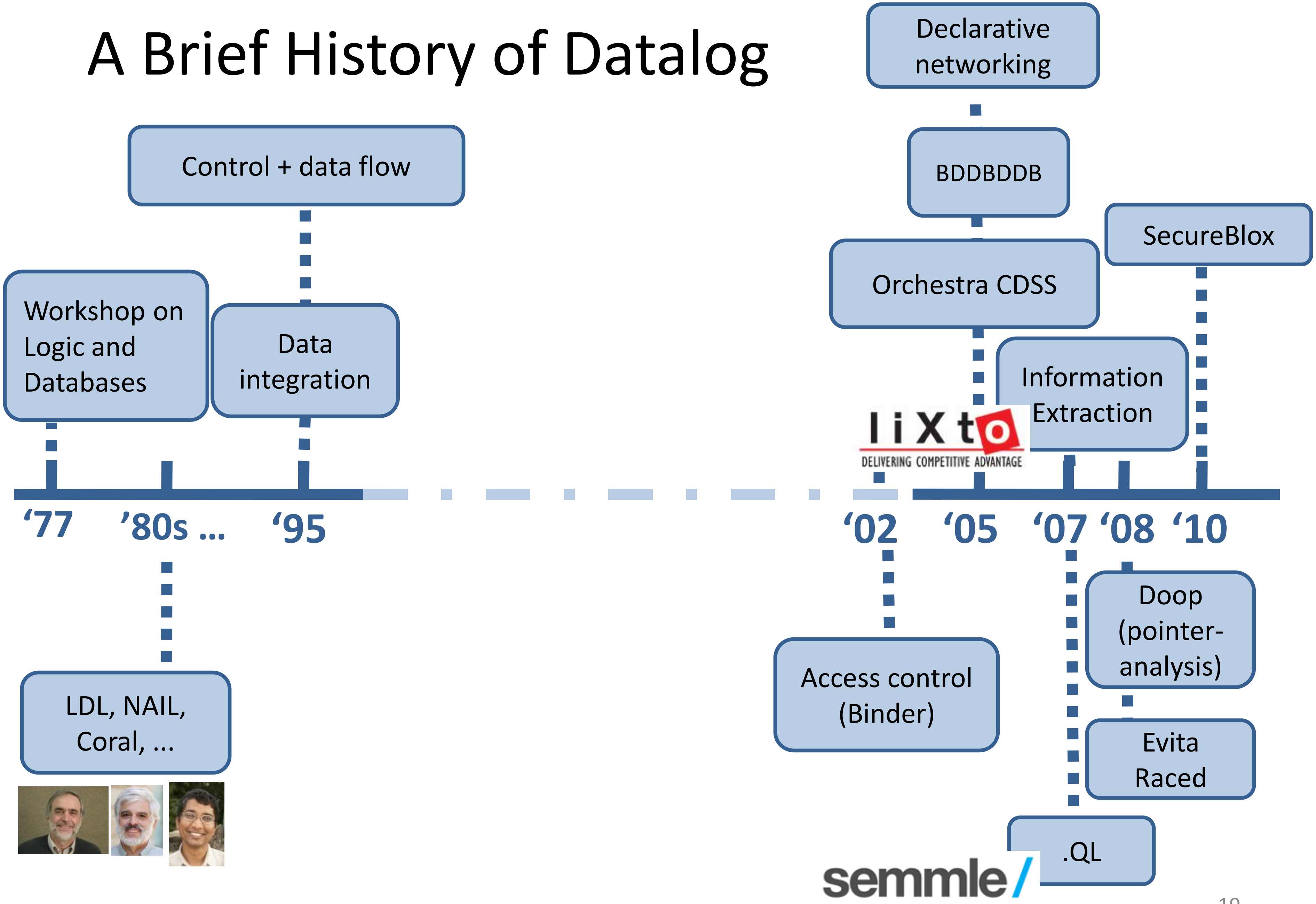
Doop (pointer-analysis)

Evita Raced

‘10

SecureBlox

A Brief History of Datalog



‘77

Workshop on Logic and Databases

’80s …

LDL, NAIL, Coral, ...

‘95

Control + data flow

Data integration

‘02

Access control (Binder)

lixto DELIVERING COMPETITIVE ADVANTAGE

‘05

Declarative networking

BDDBDDB

Orchestra CDSS

‘07

Information Extraction

.QL

semmle

‘08

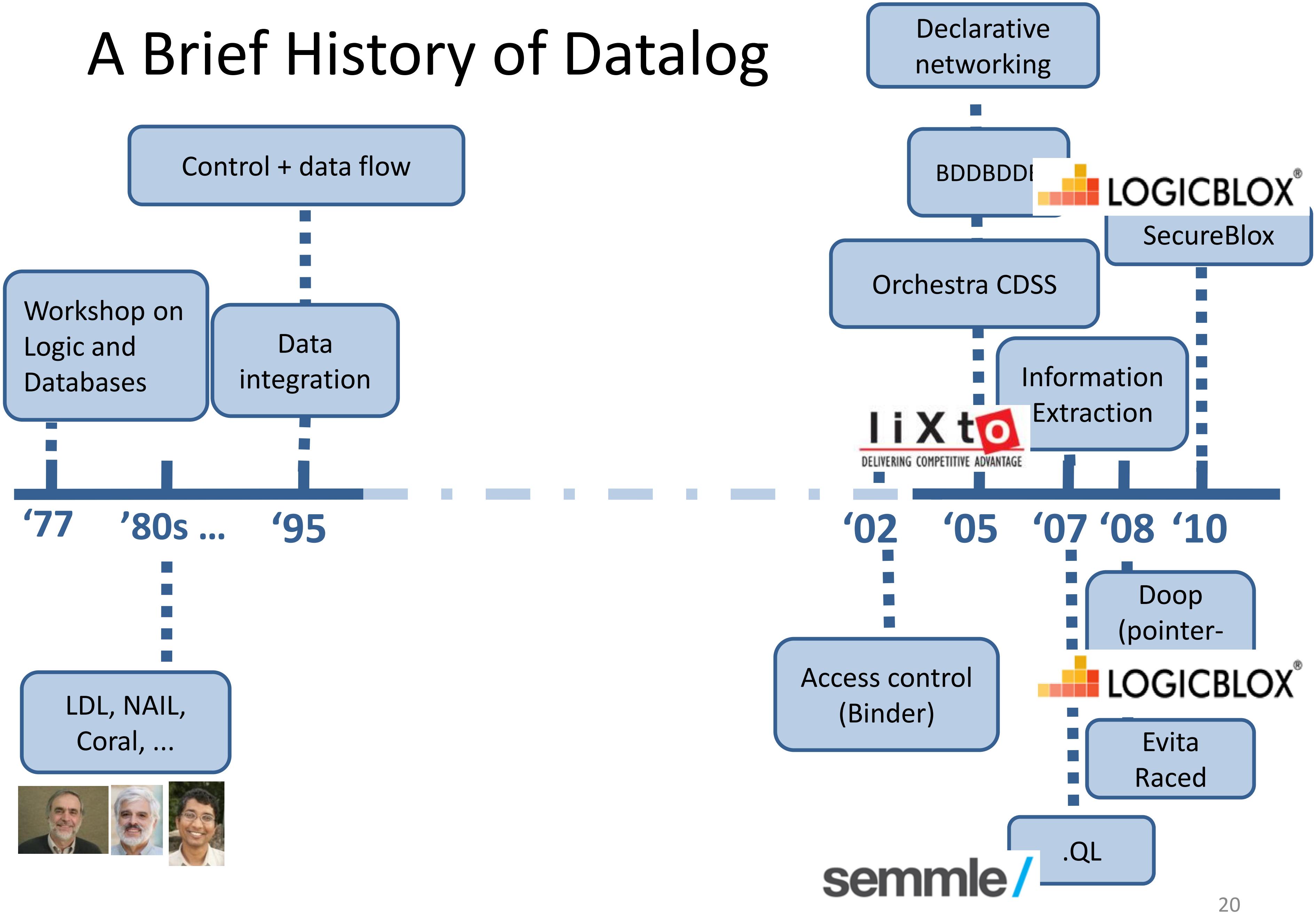
Doop (pointer-analysis)

Evita Raced

‘10

SecureBlox

A Brief History of Datalog



‘77 ’80s … ‘95 ‘02 ‘05 ‘07 ‘08 ‘10

Workshop on Logic and Databases

LDL, NAIL, Coral, ...

Control + data flow

Data integration

Access control (Binder)

Declarative networking

BDDBDD

LOGICBLOX

SecureBlox

Orchestra CDSS

lixto DELIVERING COMPETITIVE ADVANTAGE

Information Extraction

Doop (pointer-

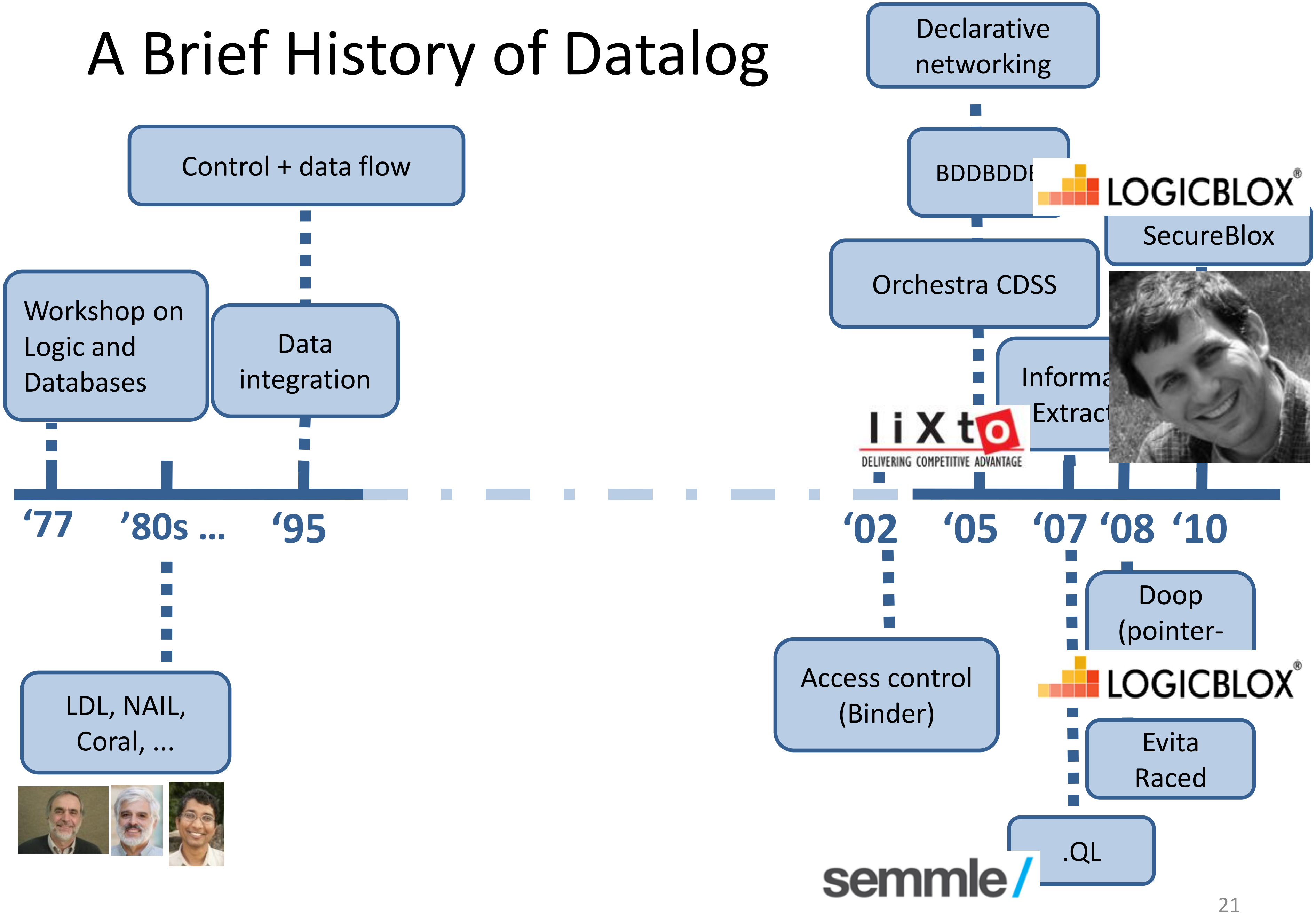
LOGICBLOX

Evita Raced

.QL

semmle

A Brief History of Datalog



‘77

Workshop on Logic and Databases

’80s …

LDL, NAIL, Coral, ...

‘95

Control + data flow

Data integration

‘02

Access control (Binder)

‘05

Declarative networking

BDDBDDI

Orchestra CDSS

‘07

Informa Extract

.QL

‘08

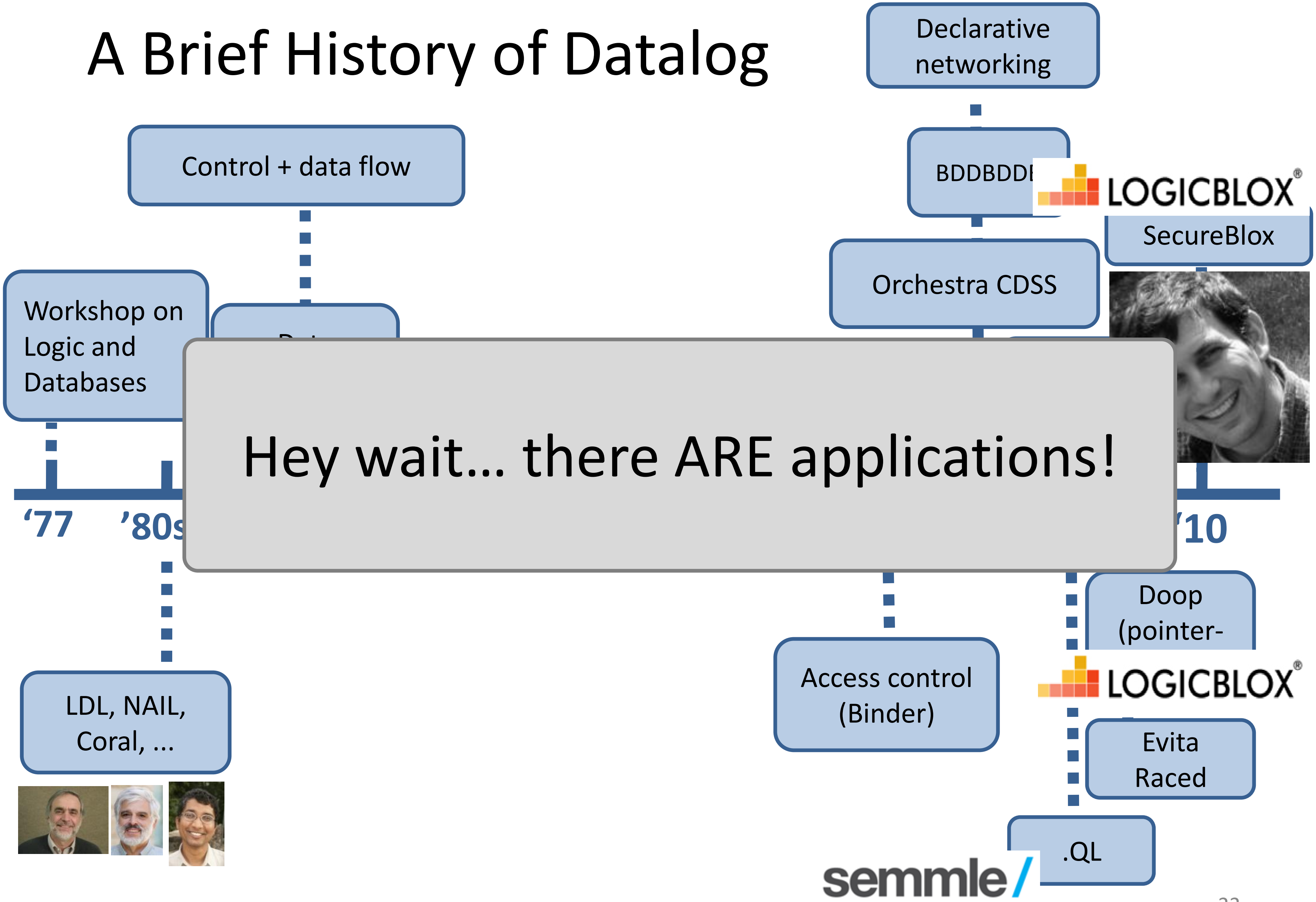
Doop (pointer-

Evita Raced

‘10

SecureBlox

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

An (Asynchronously!) 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> $\leftarrow$ <condition1>, <condition2>, … , <conditionN>.

Datalog Refresher: Syntax of Rules

Datalog rule syntax:

<result> ← <condition1>, <condition2>, … , <conditionN>.

Body

Datalog Refresher: Syntax of Rules

Datalog rule syntax:

<result> ← <condition1>, <condition2>, … , <conditionN>.

Head

Body

- Body consists of one or more conditions (input tables)
- Head is an output table
  - 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: reachable(S,D) <- link(S,D).

R2: 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”.

- Input: link(source, destination)
- Output: reachable(source, destination)

Terminology and Convention

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

- 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

Negated Atoms

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

Negated Atoms

Not “cut” in Prolog. ☺

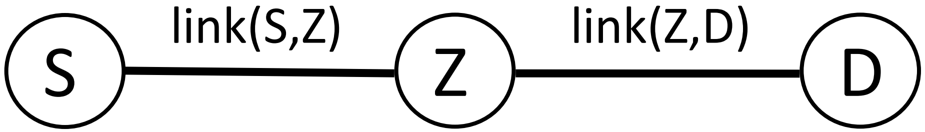
- We may put ! (NOT) in front of a atom, to negate its meaning.

Negated Atoms

Not “cut” in Prolog. ☺

- 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).
```



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) <- r(Y).
  - s(X) <- 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

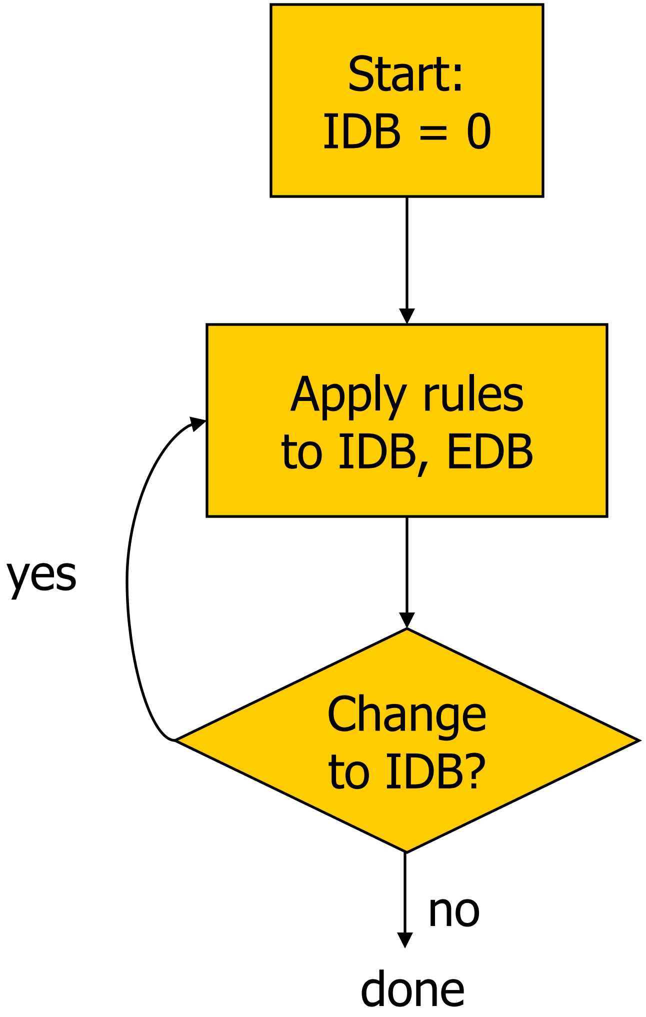
- Model-theoretic
  - Most “declarative”. Based on model-theoretic semantics of first order logic. View rules as logical constraints.
  - Given input DB I and Datalog program P, find the smallest possible DB instance I’ that extends I and satisfies all constraints in P.
- Fixpoint-theoretic
  - Most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



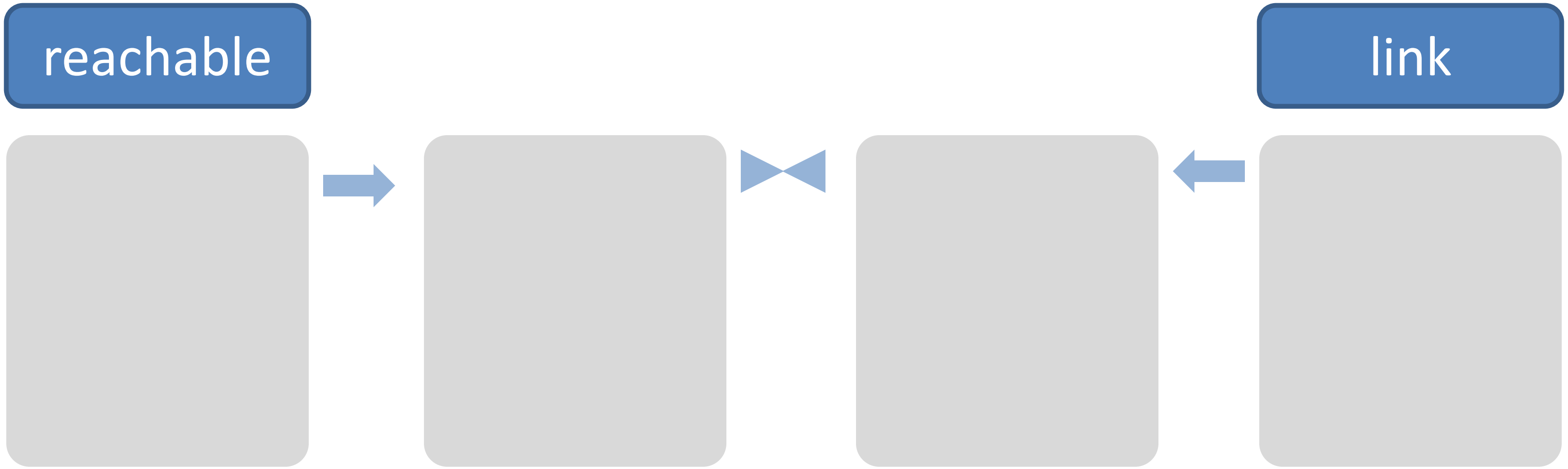
Naïve Evaluation

reachable

link

```
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
link

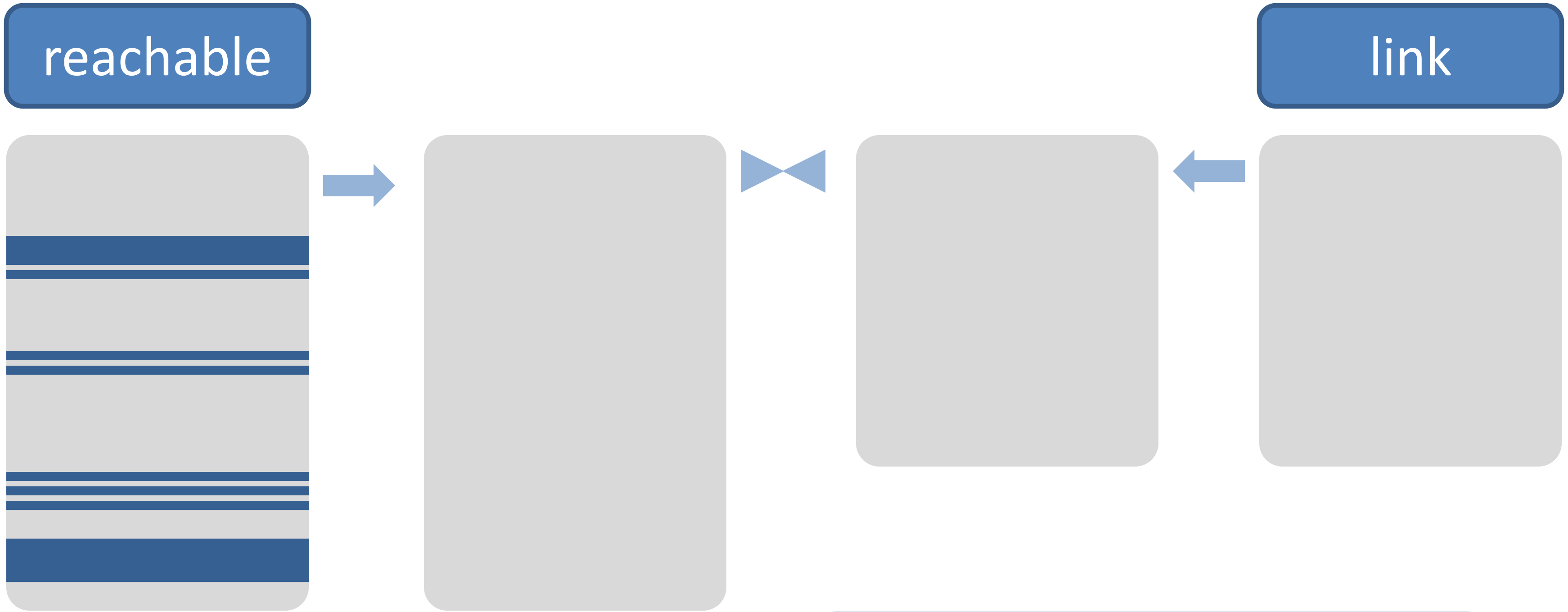
```
reachable(S,D) <- link(S,D).
reachable(S,D)  <- link(S,Z),
          reachable(Z,D).
```

Naïve Evaluation

reachable
link

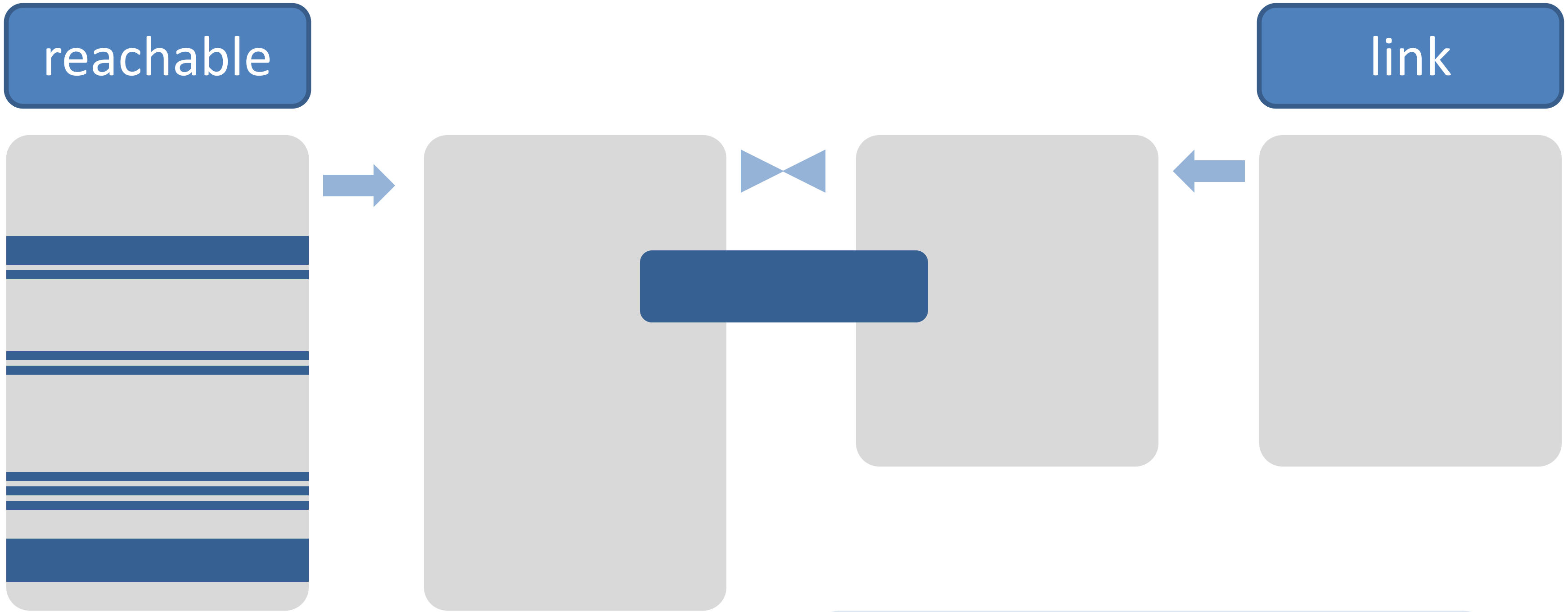
```
reachable(S,D) <- link(S,D).
reachable(S,D)  <- link(S,Z),
        reachable(Z,D).
```

Naïve Evaluation



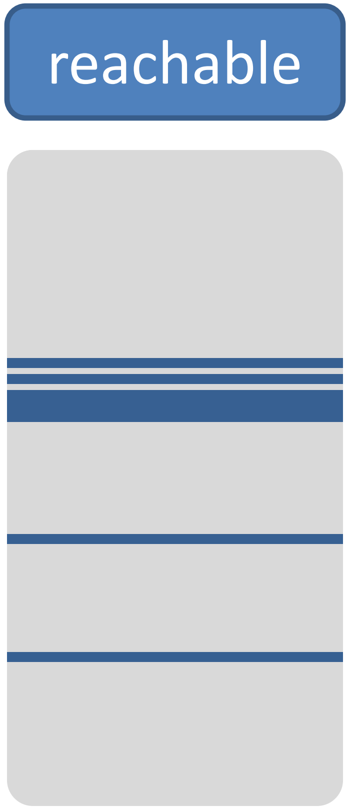
```
reachable(S,D) <- link(S,D).
reachable(S,D)  <- link(S,Z),
          reachable(Z,D).
```

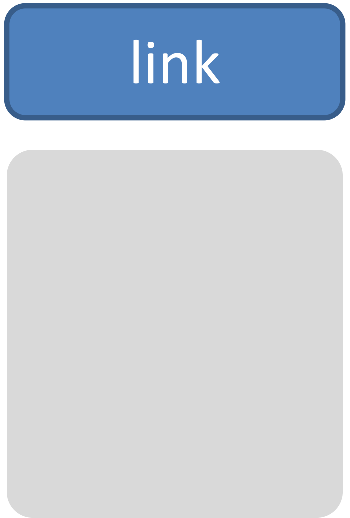
Naïve Evaluation



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

Naïve Evaluation

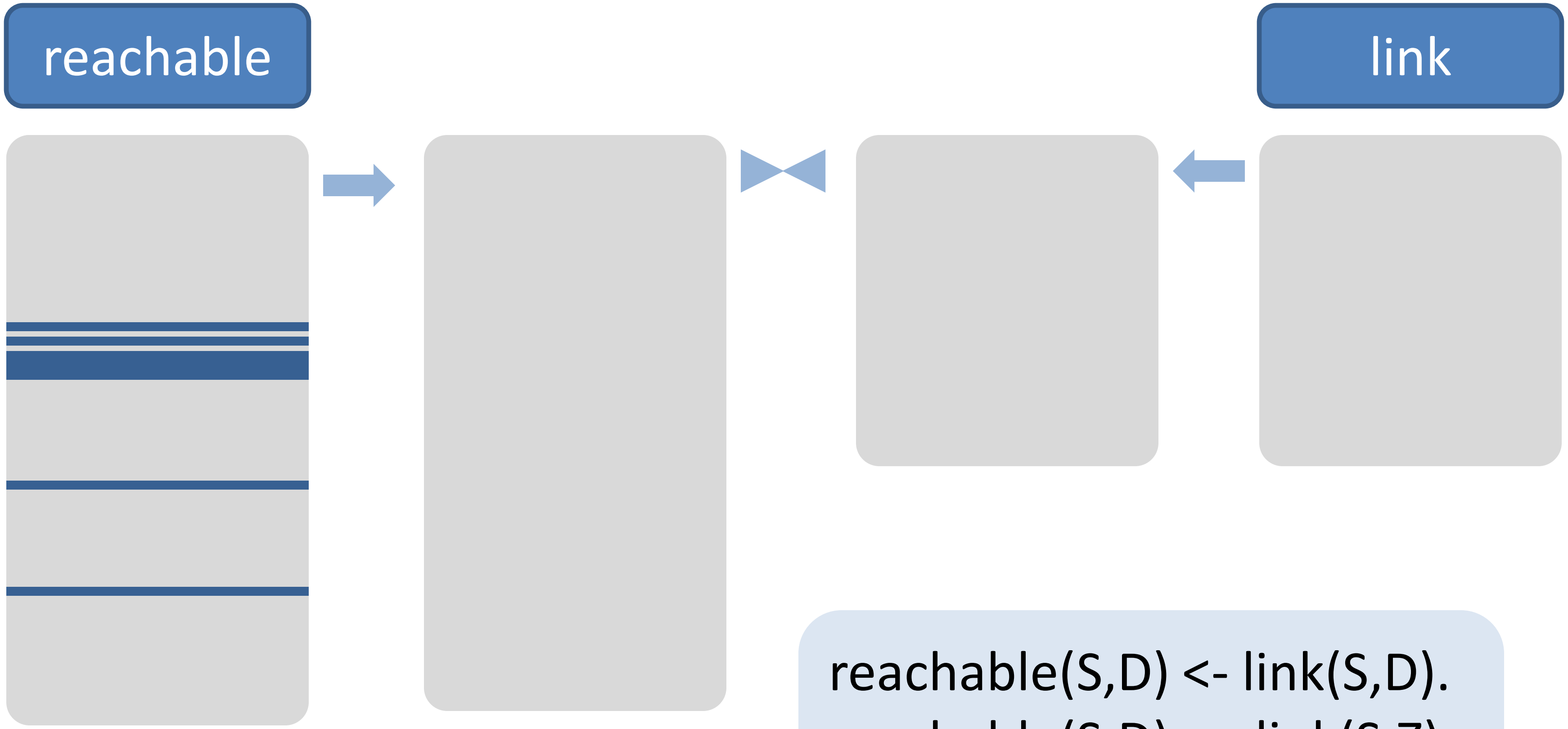




reachable(S,D) <- link(S,D).

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).

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.

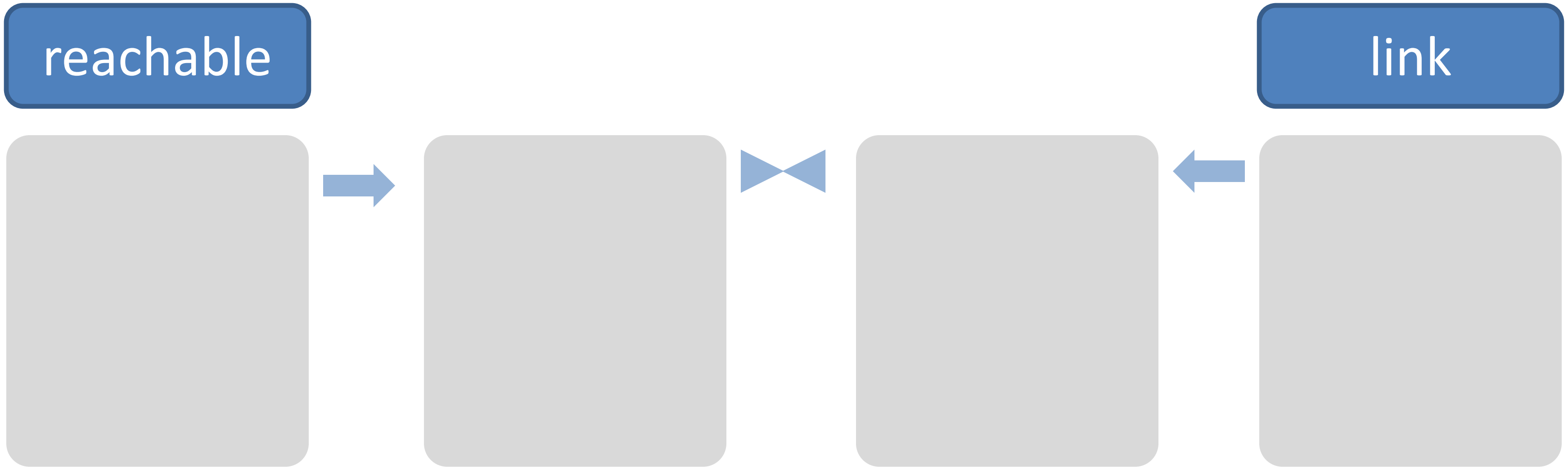
Semi-naïve Evaluation

reachable

link

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

Semi-naïve Evaluation



reachable

link

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

Semi-naïve Evaluation

reachable
link

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

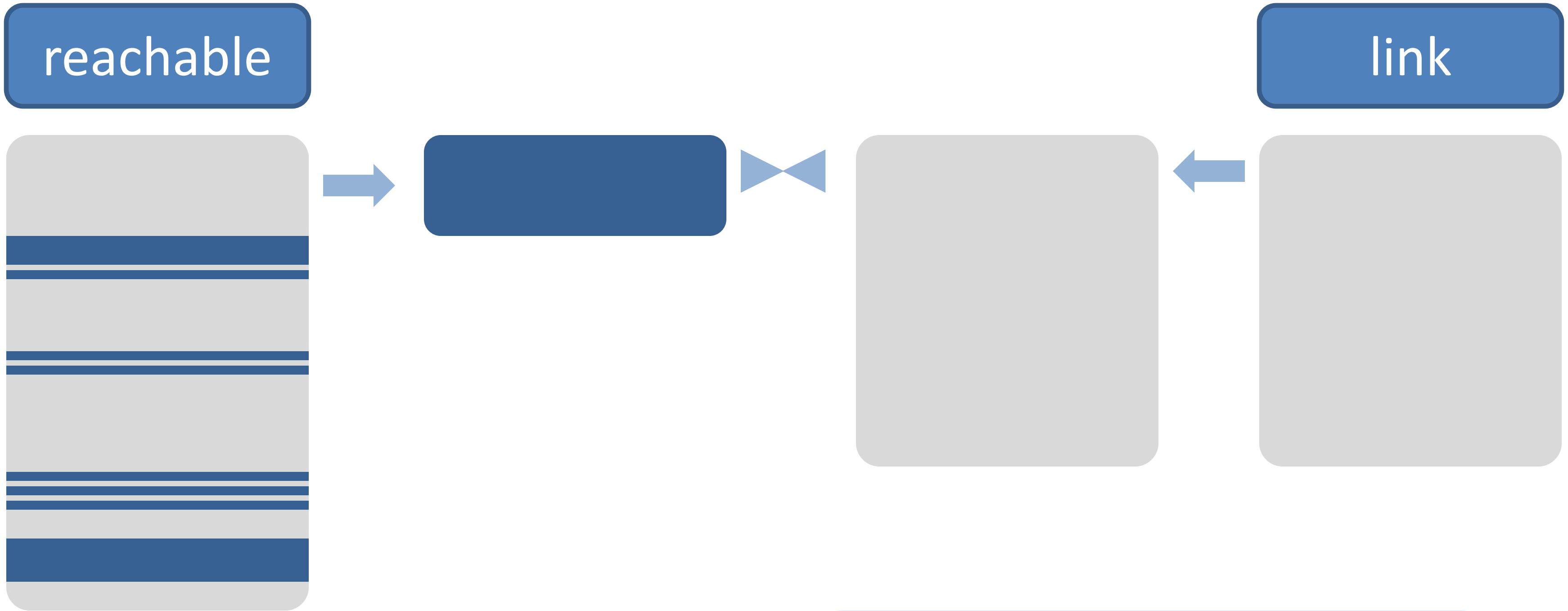
Semi-naïve Evaluation

reachable

link

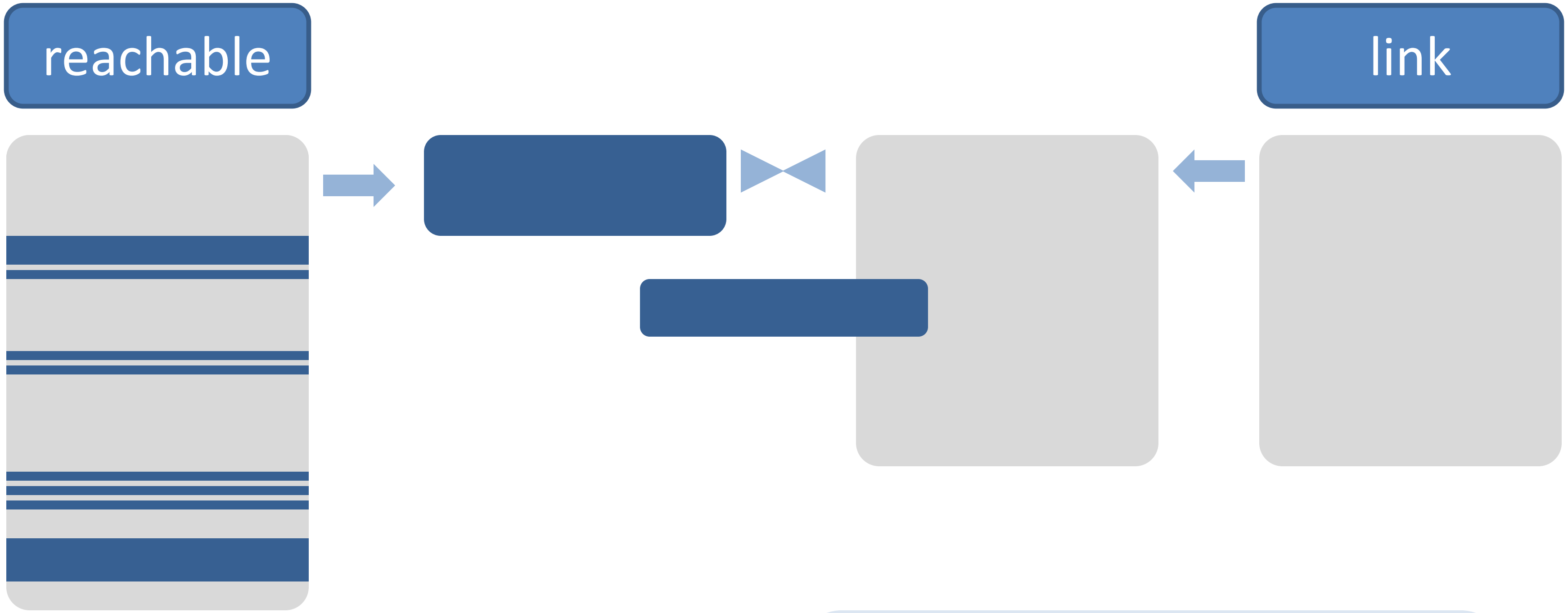
```
reachable(S,D) <- link(S,D).
reachable(S,D)  <- link(S,Z),
          reachable(Z,D).
```

Semi-naïve Evaluation



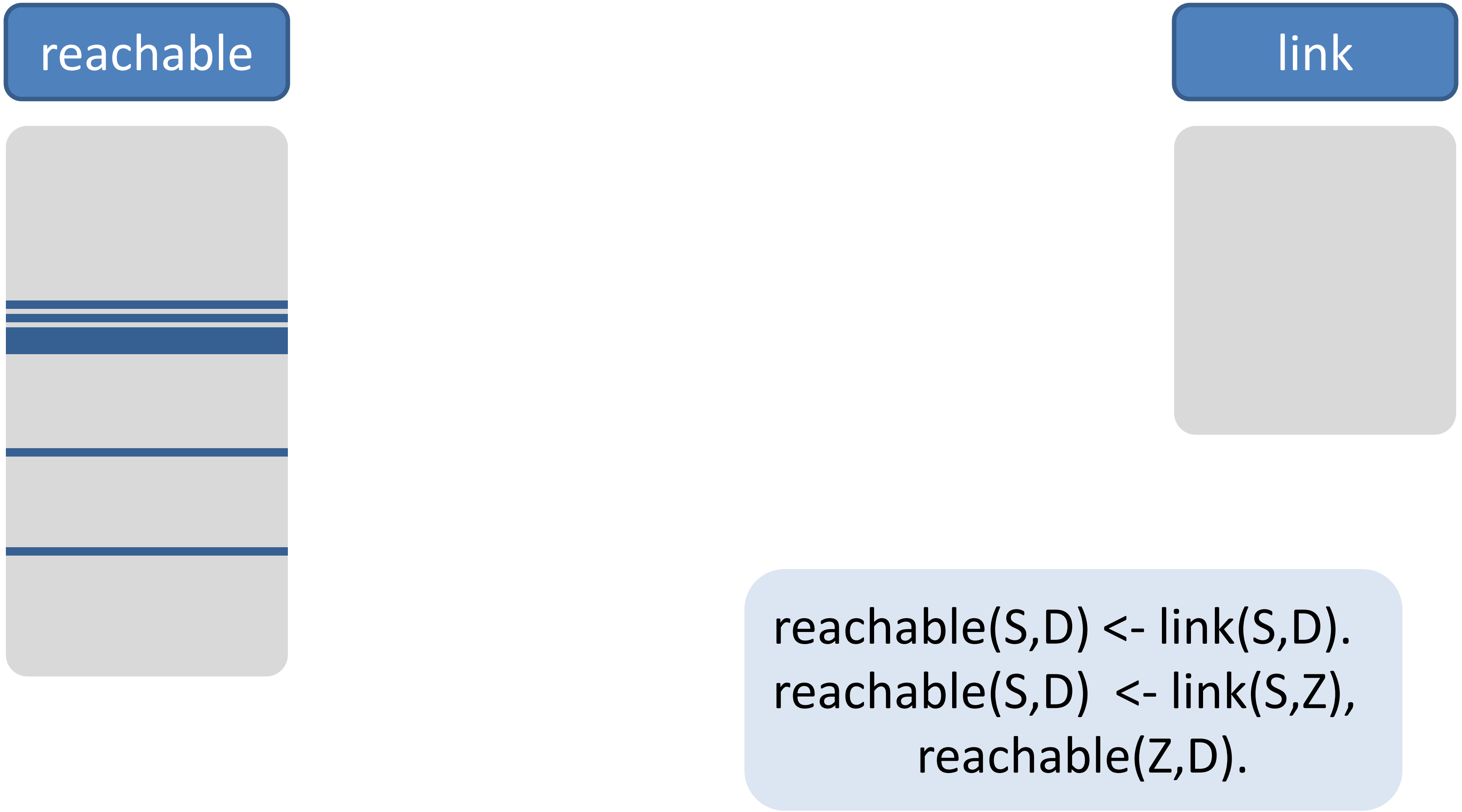
reachable(S,D) <- link(S,D).
reachable(S,D) <- link(S,Z),
reachable(Z,D).

Semi-naïve Evaluation



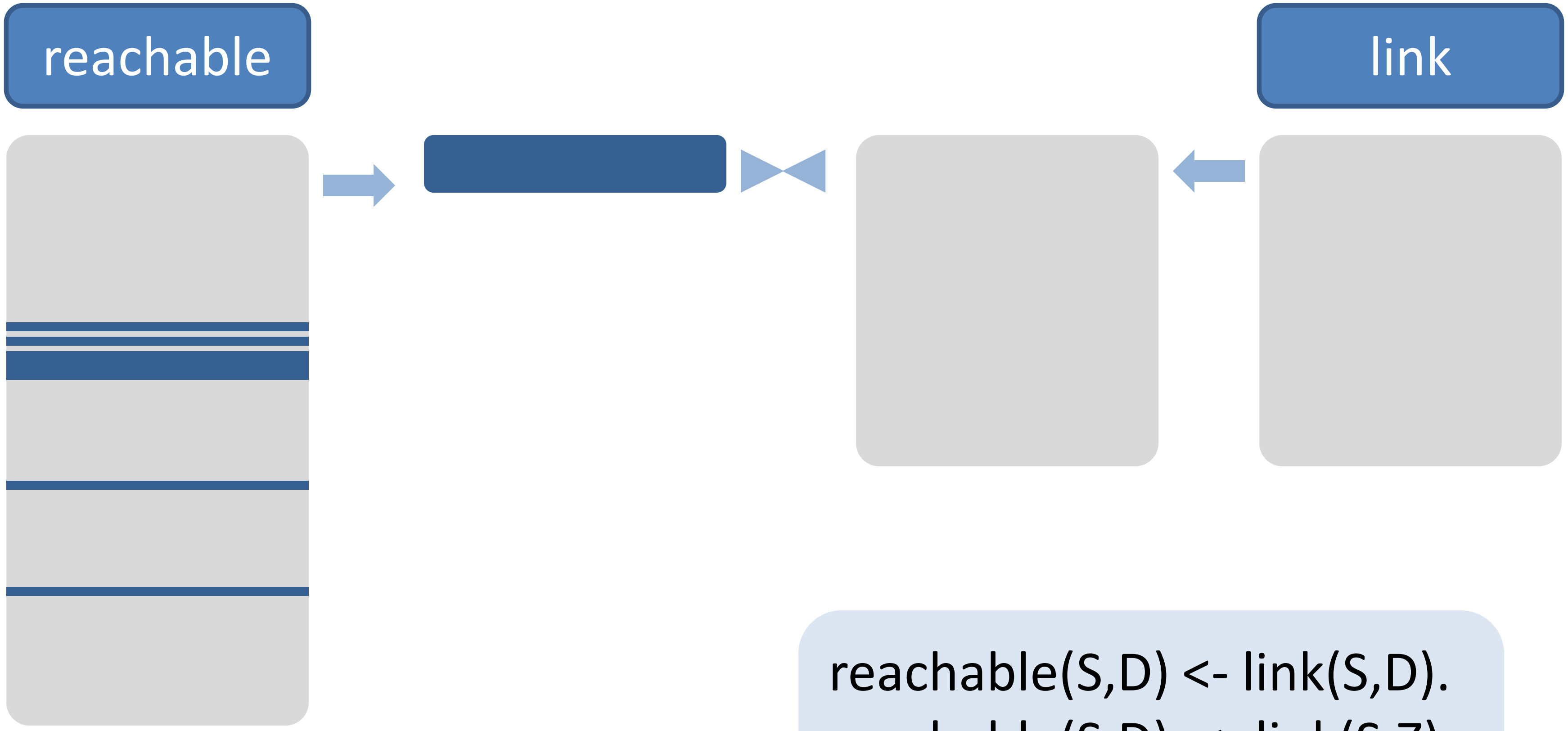
```
reachable(S,D) <- link(S,D).
reachable(S,D)  <- link(S,Z),
           reachable(Z,D).
```

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).
```

Recursion with Negation

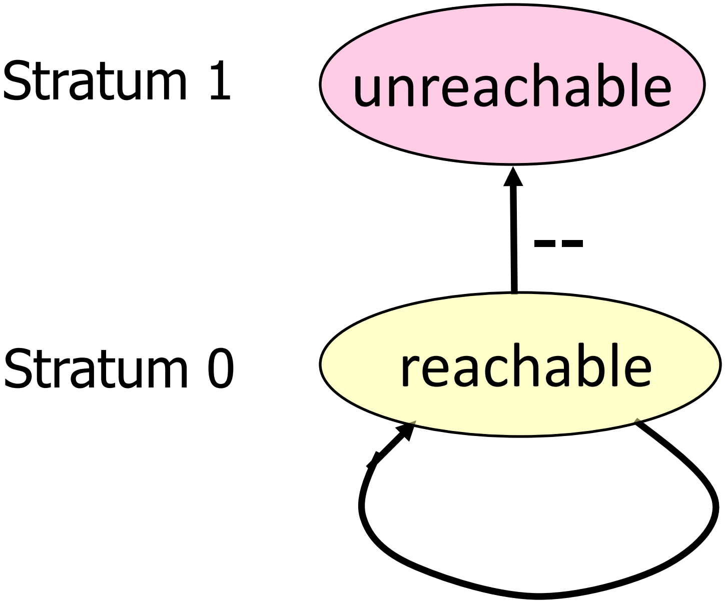
Example: to compute all pairs of disconnected nodes in a graph.

```
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).
```

Recursion with Negation

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

```
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).
```

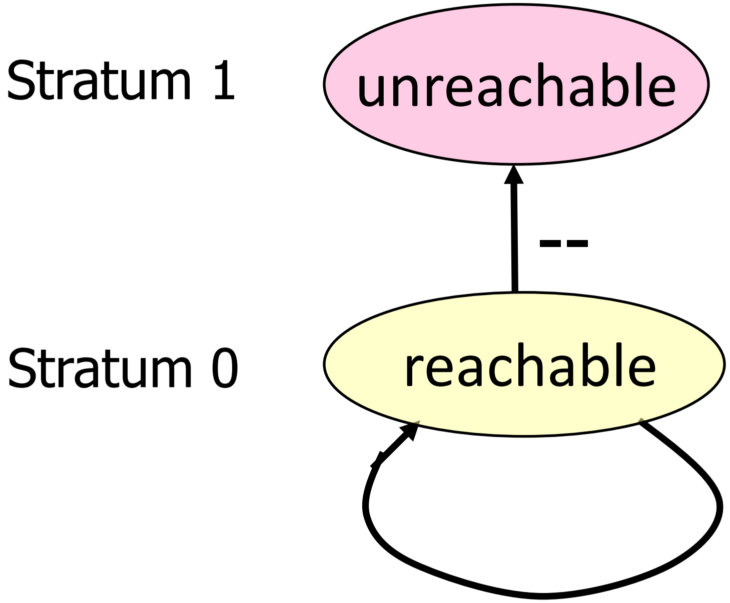


Precedence graph :

Nodes = IDB predicates.
Edge *q* <- *p* if predicate *q* depends on *p*.
Label this arc "–" if the predicate p is negated.

Stratified Negation

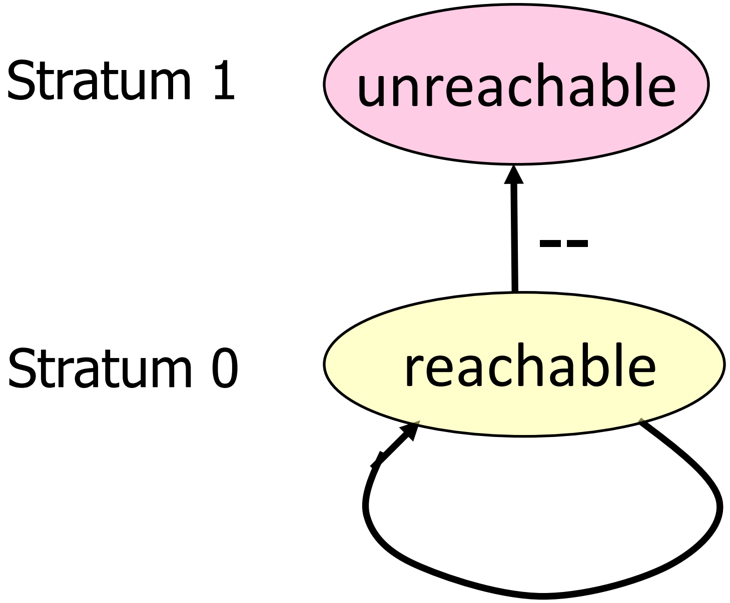
```
reachable(S,D) <- link(S,D).
reachable(S,D)  <- link(S,Z),
                          reachable(Z,D).
unreachable(S,D)  <- node(S),
                           node(D),
                          ! reachable(S,D).
```



- 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.

Stratified Negation

```
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).
```



- 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) <- q(X), ! 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:
  - **A Survey of Research on Deductive Database Systems**, Ramakrishnan and Ullman, Journal of Logic Programming, 1993
  - **What you always wanted to know about datalog (and never dared to ask)**, by Ceri, Gottlob, and Tanca.
  - **An Amateur's Expert's Guide to Recursive Query Processing**, Bancilhon and Ramakrishnan, SIGMOD Record.
  - **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

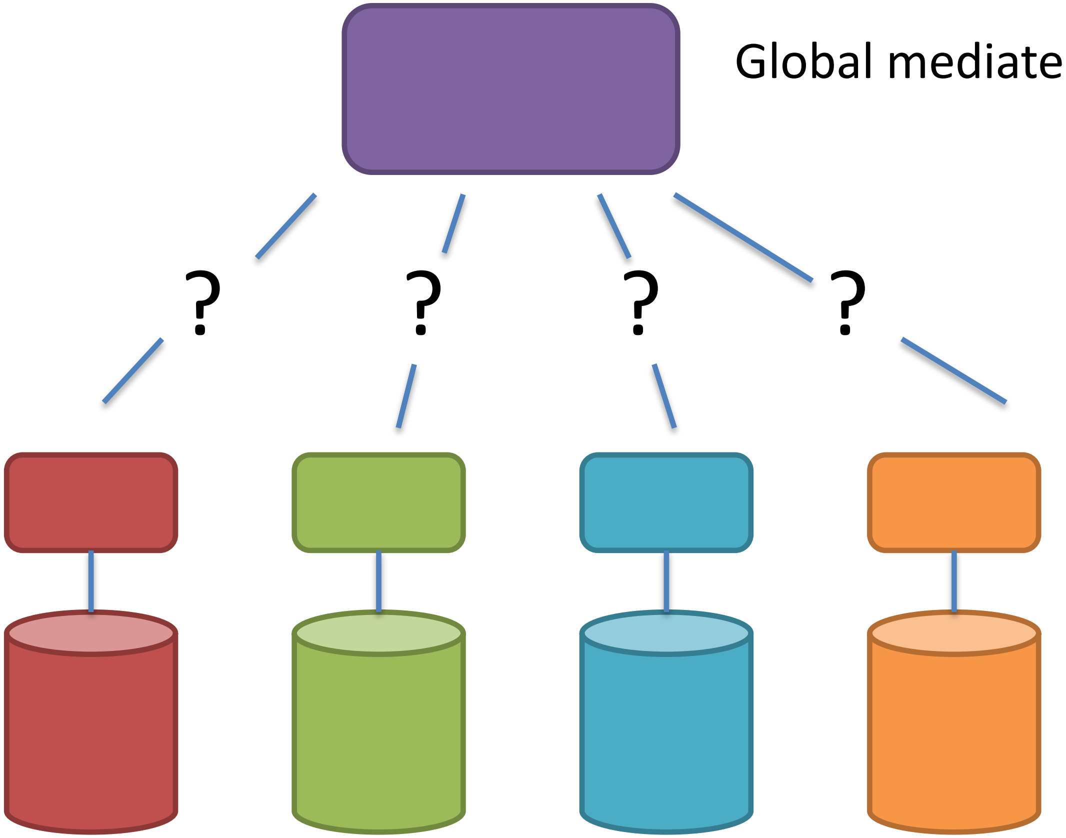
- Motivation and problem setting
- Two basic approaches:
  - virtual data integration
  - materialized data exchange
- Schema mappings and Datalog with **Skolem functions**

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 .

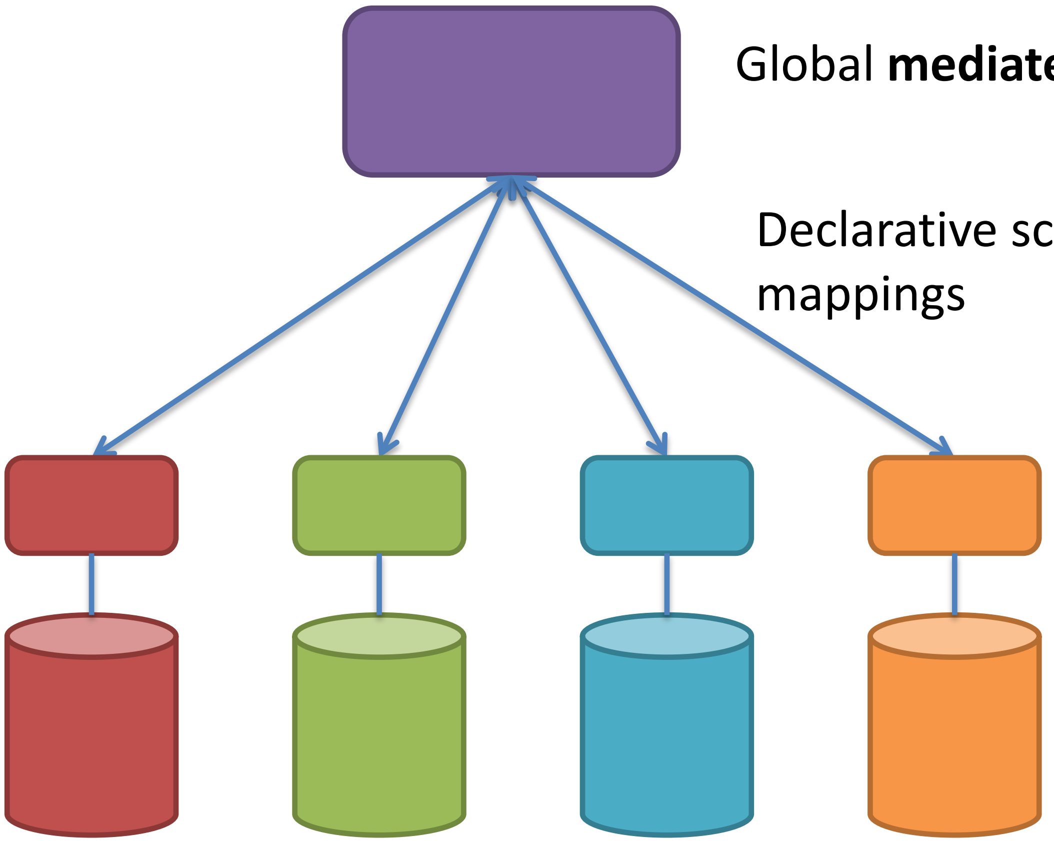


Global mediated schema

Source schemas

Local data sources

Mediator-Based Virtual Data Integration



Global **mediated** schema

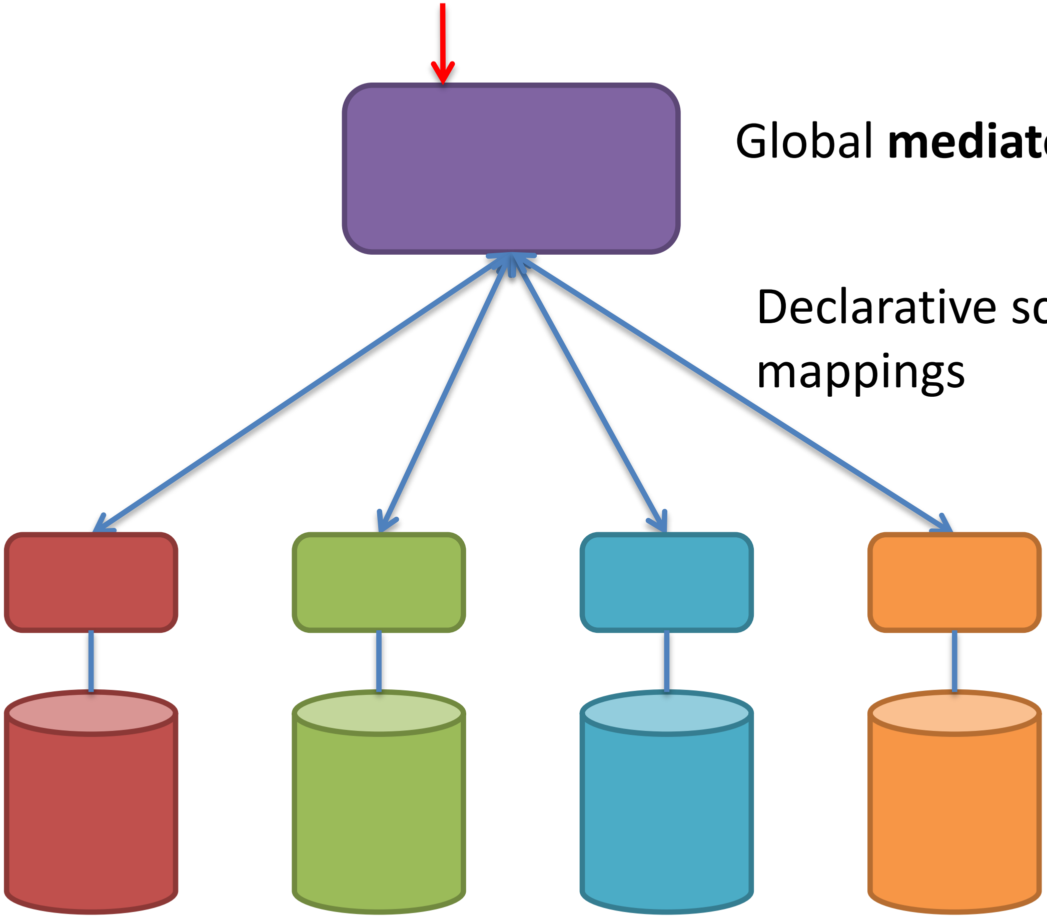
Declarative schema mappings

Source schemas

Local data sources

Mediator-Based Virtual Data Integration

Query over
global schema



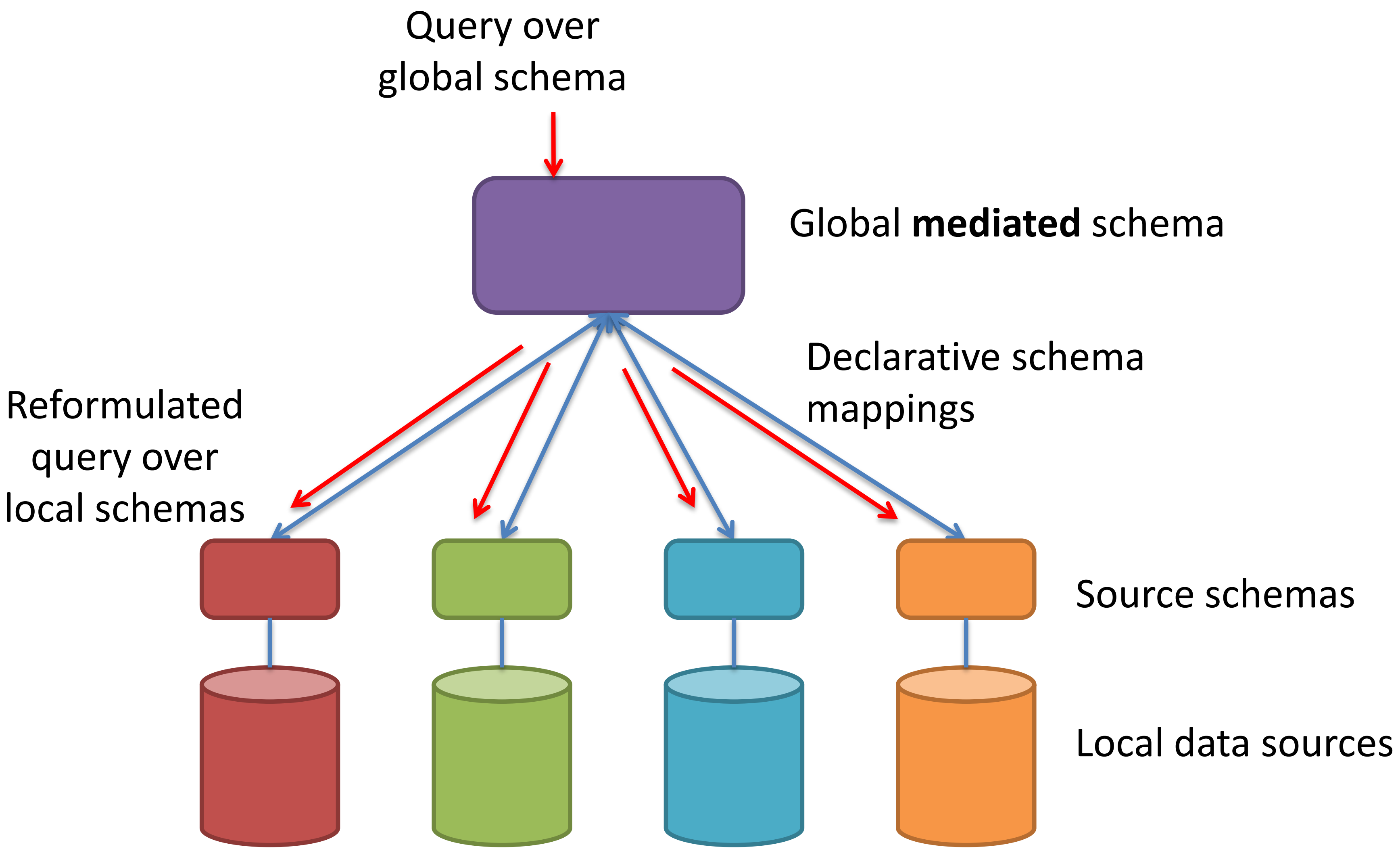
Global **mediated** schema

Declarative schema
mappings

Source schemas

Local data sources

Mediator-Based Virtual Data Integration



Query over
global schema

Global **mediated** schema

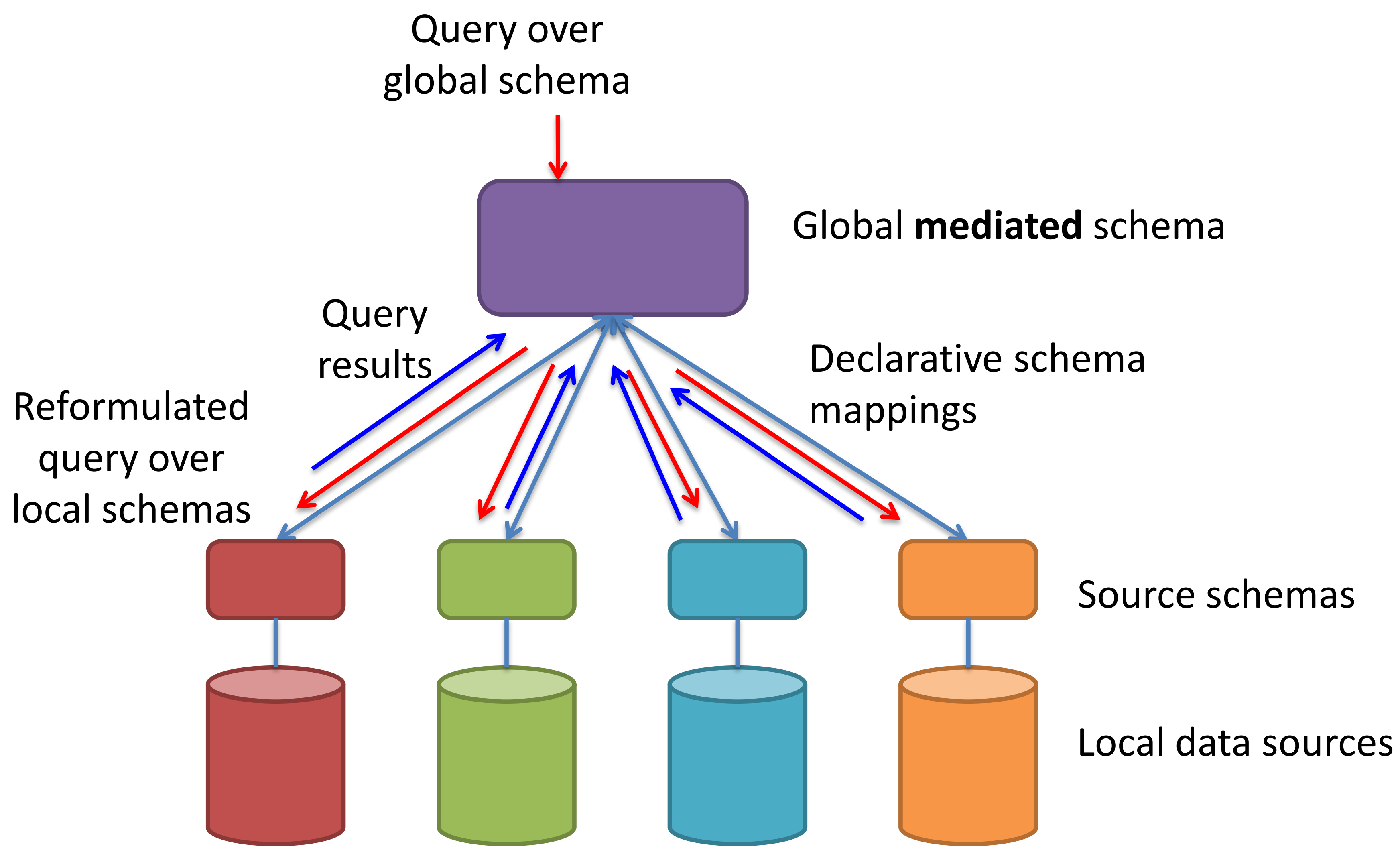
Declarative schema
mappings

Reformulated
query over
local schemas

Source schemas

Local data sources

Mediator-Based Virtual Data Integration



Query over global schema

Global **mediated** schema

Query results

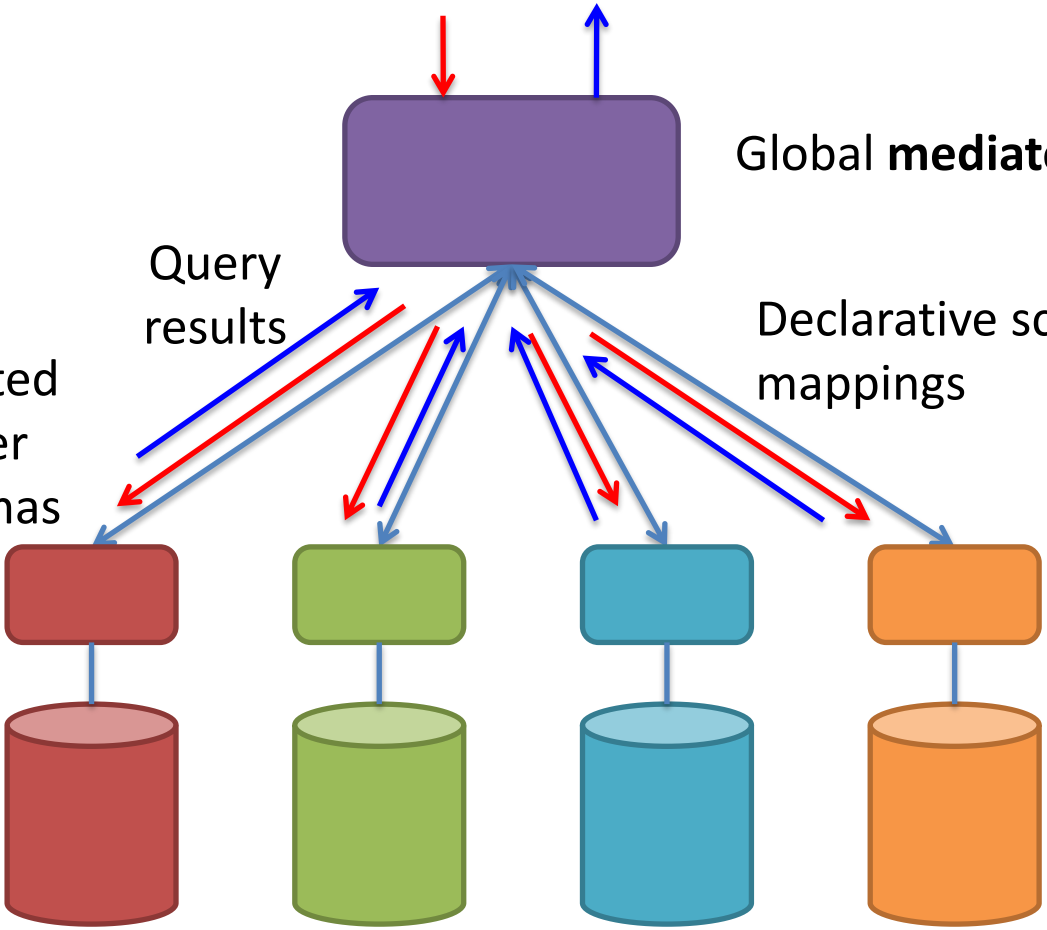
Declarative schema mappings

Reformulated query over local schemas

Source schemas

Local data sources

Mediator-Based Virtual Data Integration



Query over global schema

Integrated query results

Global **mediated** schema

Query results

Reformulated query over local schemas

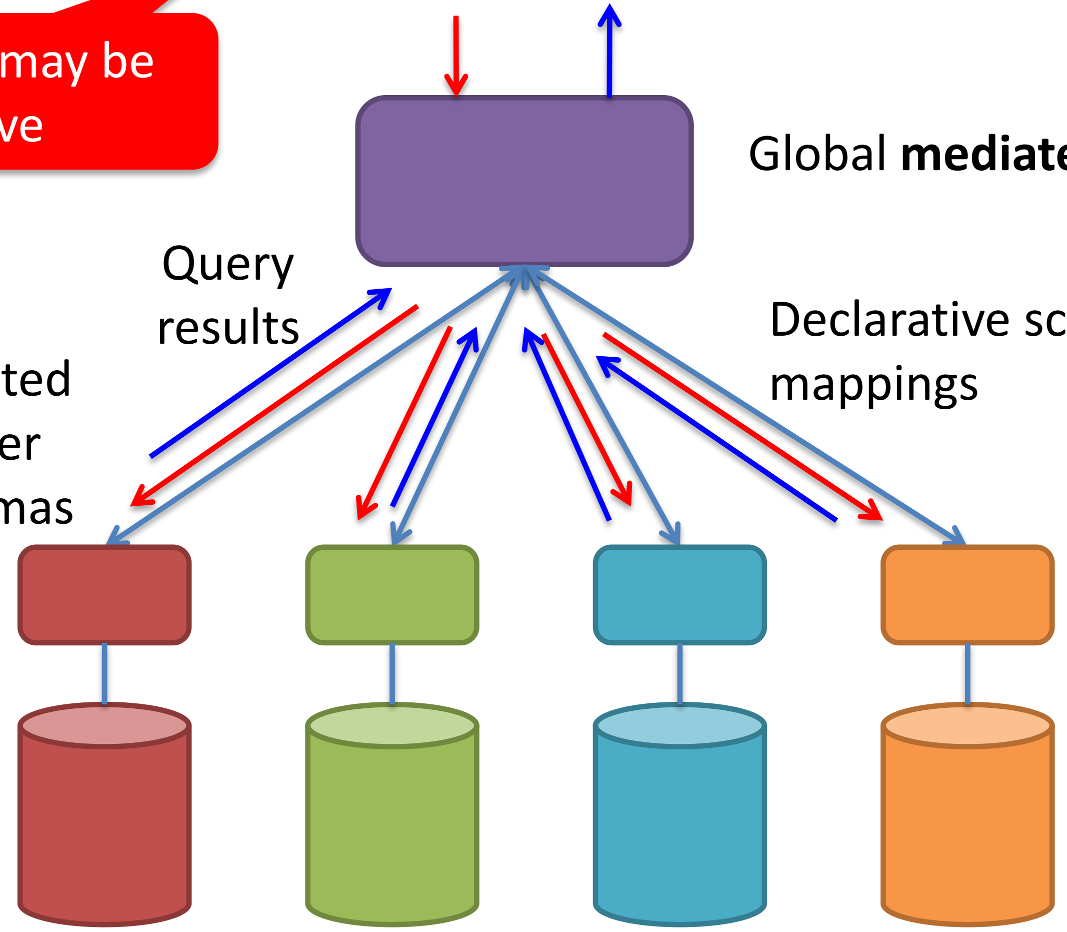
Declarative schema mappings

Source schemas

Local data sources

Mediator-Based Virtual Data Integration

Query may be recursive



Query over global schema

Integrated query results

Global mediated schema

Query results

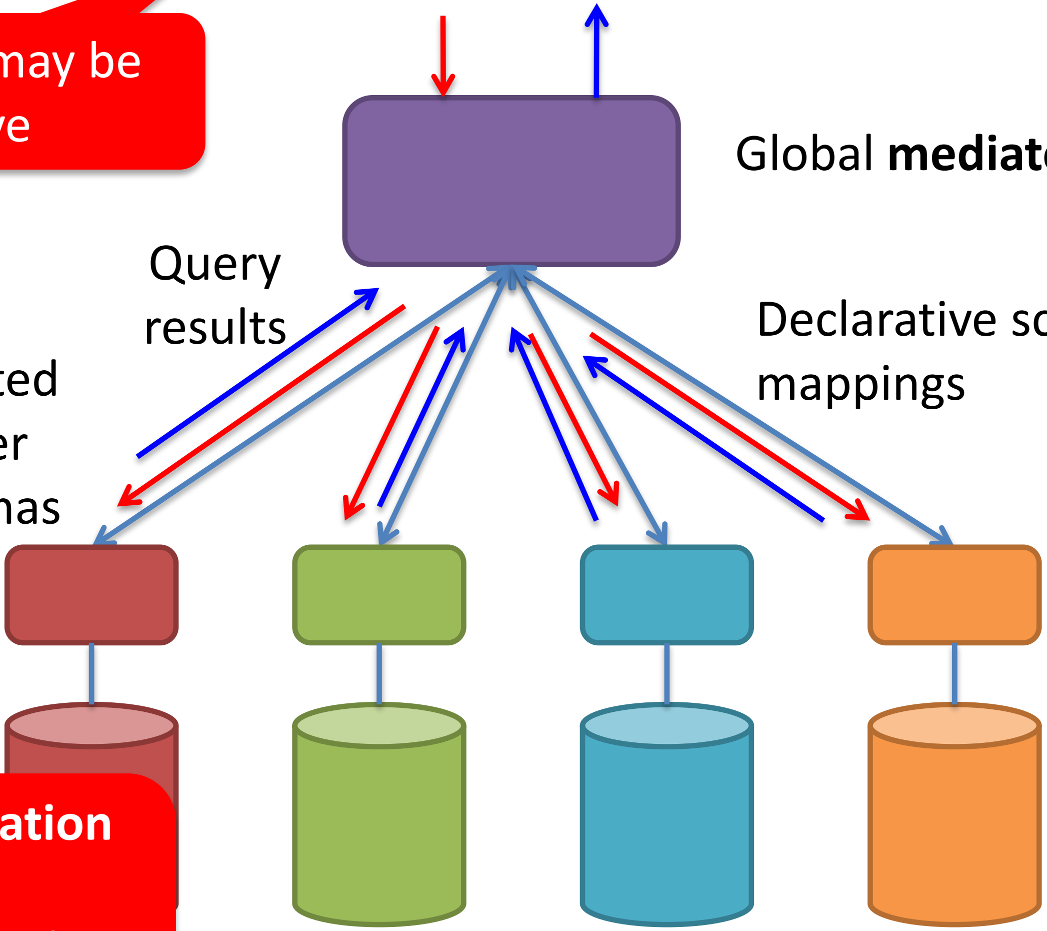
Reformulated query over local schemas

Declarative schema mappings

Source schemas

Local data sources

Mediator-Based Virtual Data Integration



Query may be recursive

Query over global schema

Integrated query results

Global **mediated** schema

Query results

Declarative schema mappings

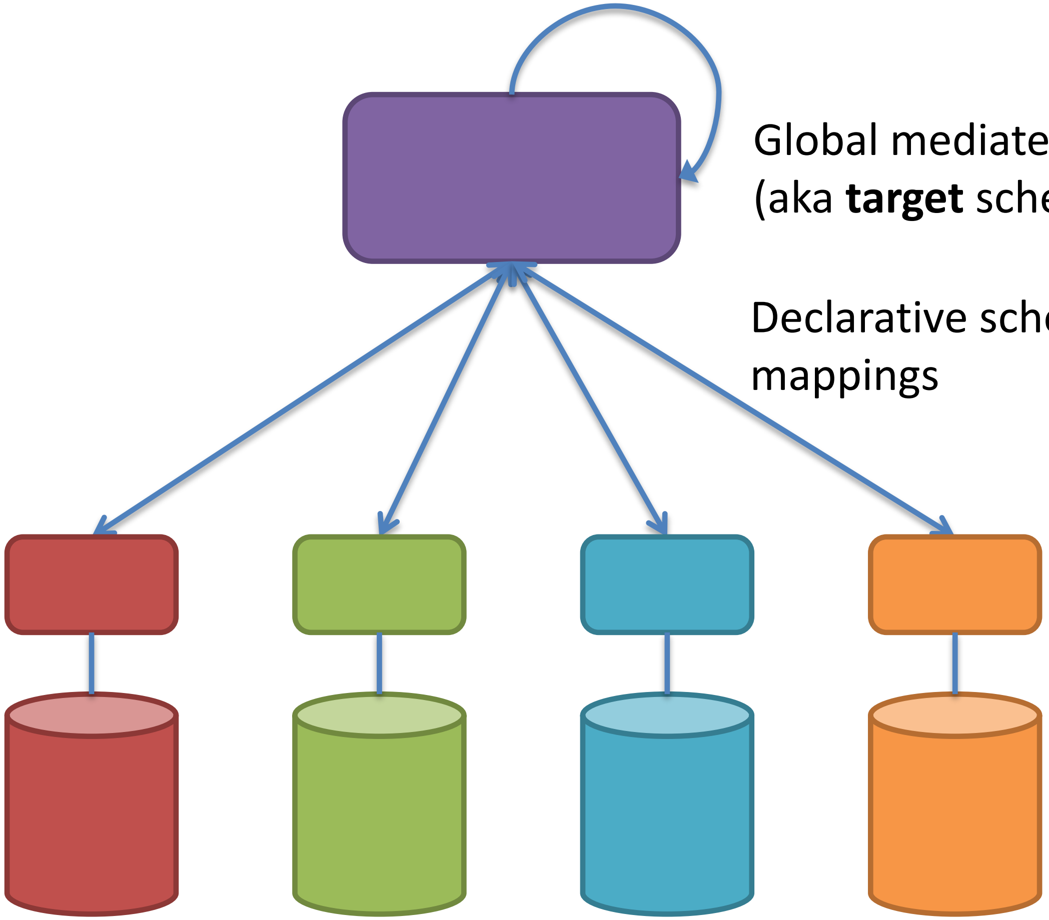
Reformulated query over local schemas

Source schemas

Local data sources

Reformulation may be (necessarily) recursive

Materialized Data Exchange



Declarative schema mappings

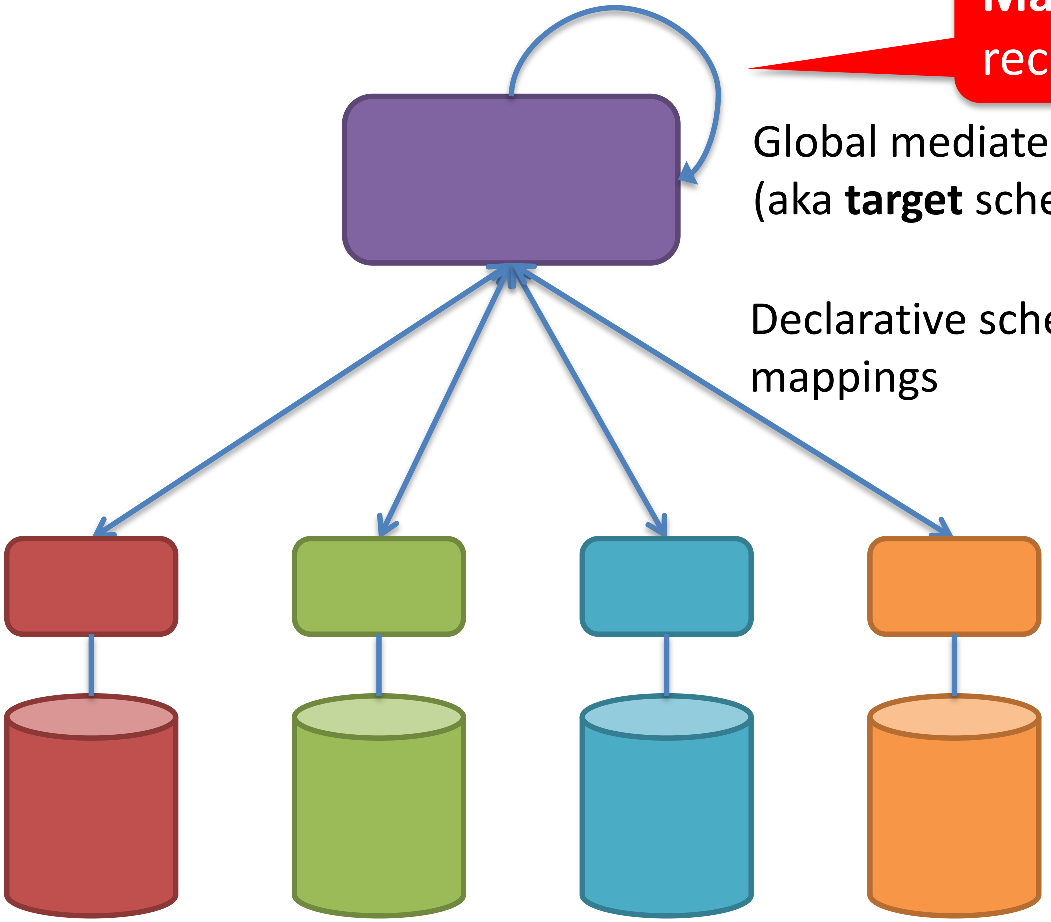
Global mediated schema (aka **target** schema)

Declarative schema mappings

Source schema(s)

Local data source(s)

Materialized Data Exchange



Declarative schema mappings

Mappings may be recursive

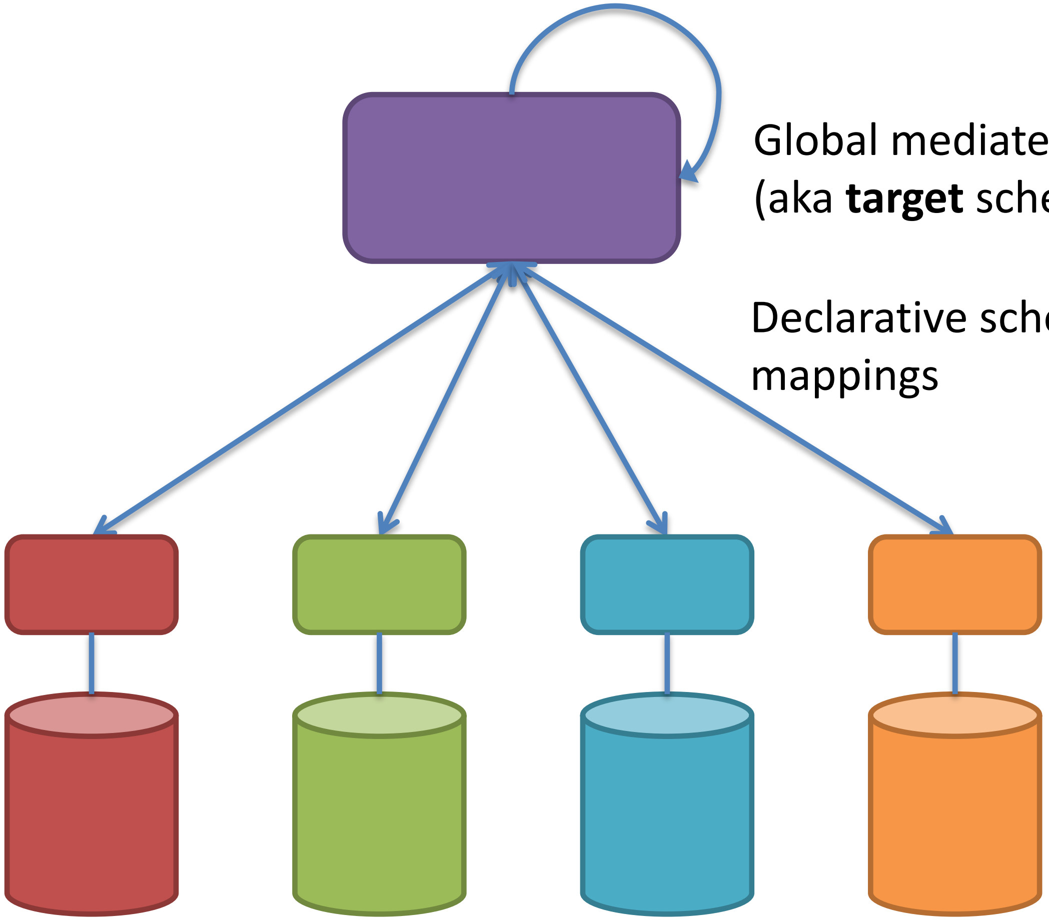
Global mediated schema (aka **target** schema)

Declarative schema mappings

Source schema(s)

Local data source(s)

Materialized Data Exchange



Declarative schema mappings

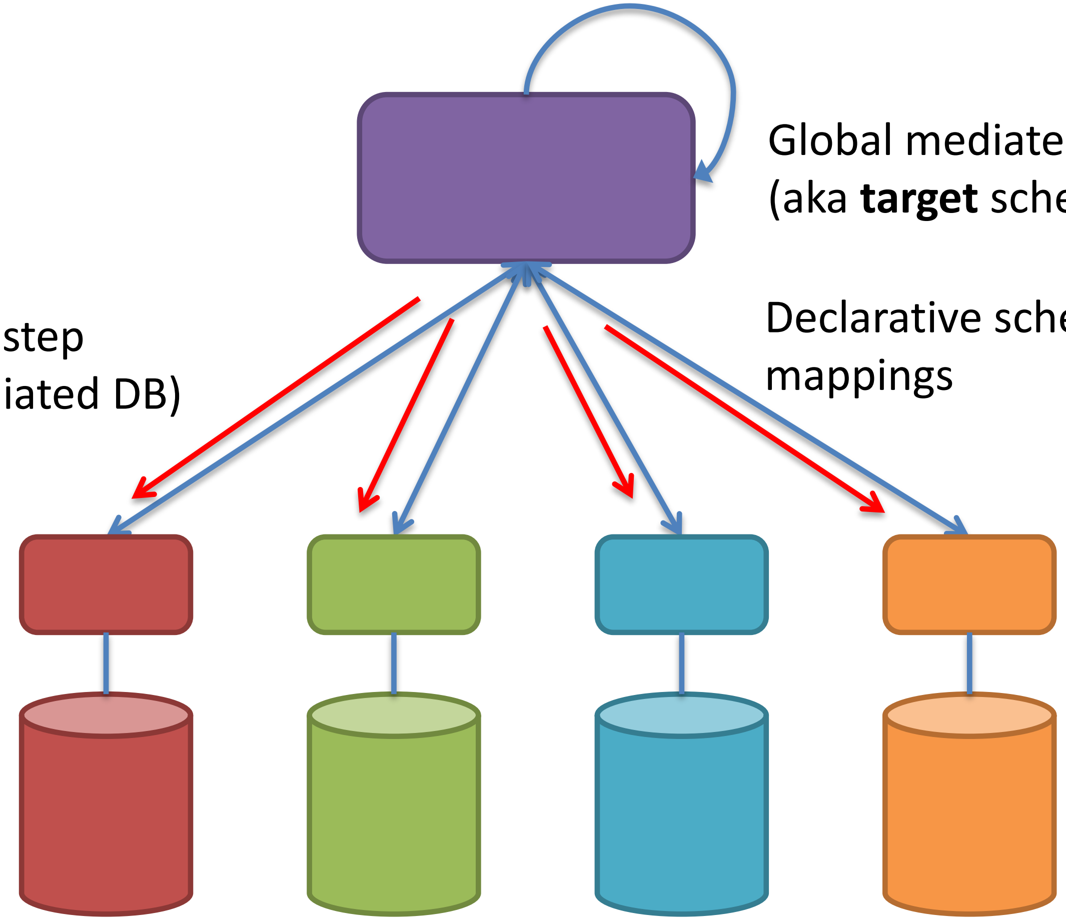
Global mediated schema (aka **target** schema)

Declarative schema mappings

Source schema(s)

Local data source(s)

Materialized Data Exchange



Declarative schema mappings

Global mediated schema (aka **target** schema)

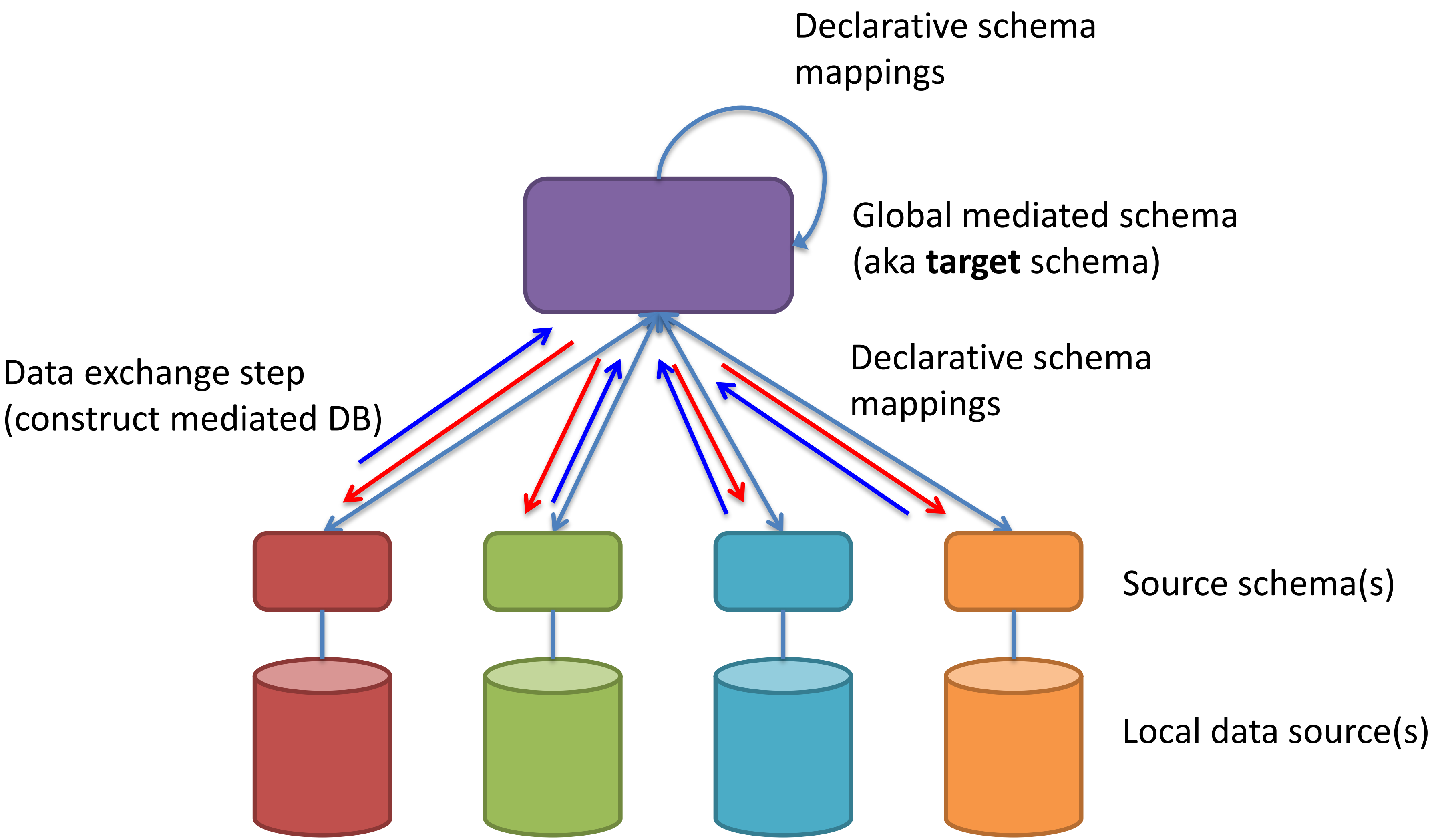
Data exchange step (construct mediated DB)

Declarative schema mappings

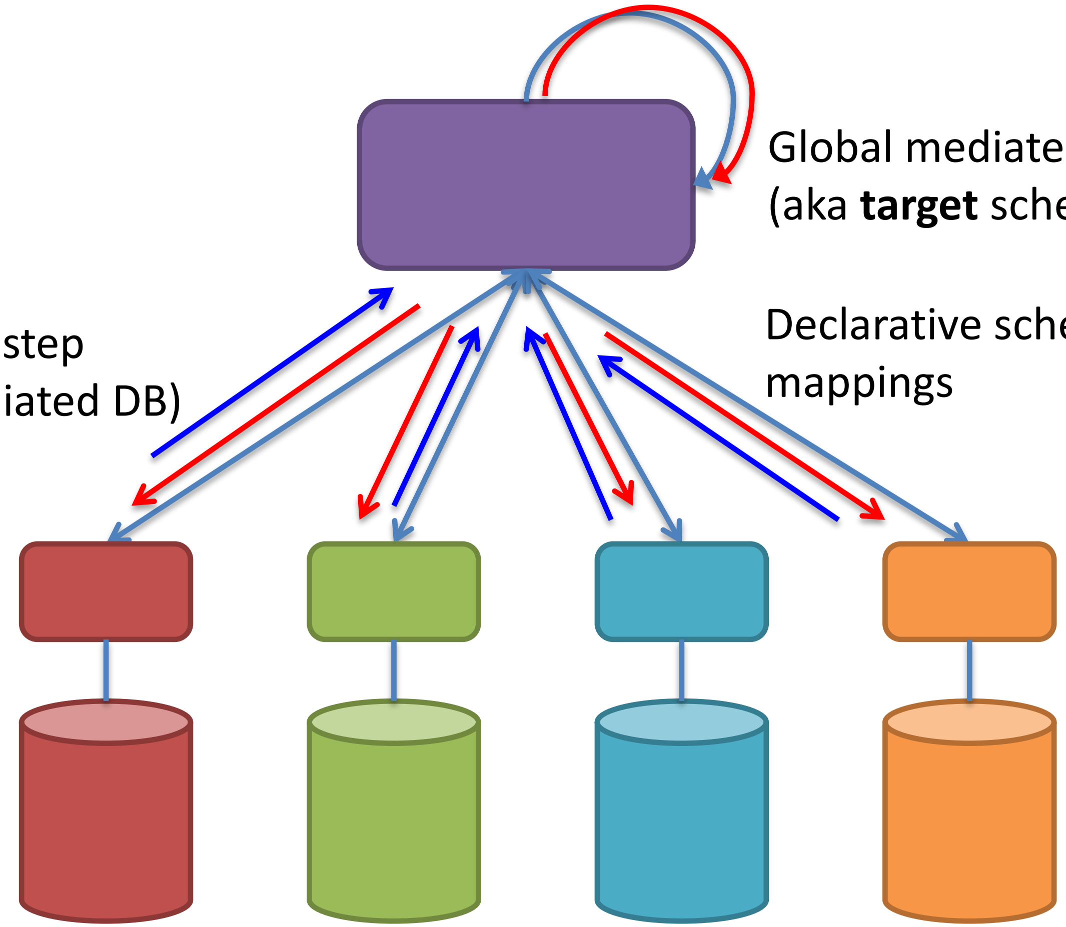
Source schema(s)

Local data source(s)

Materialized Data Exchange


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

Materialized Data Exchange



Declarative schema mappings

Global mediated schema (aka **target** schema)

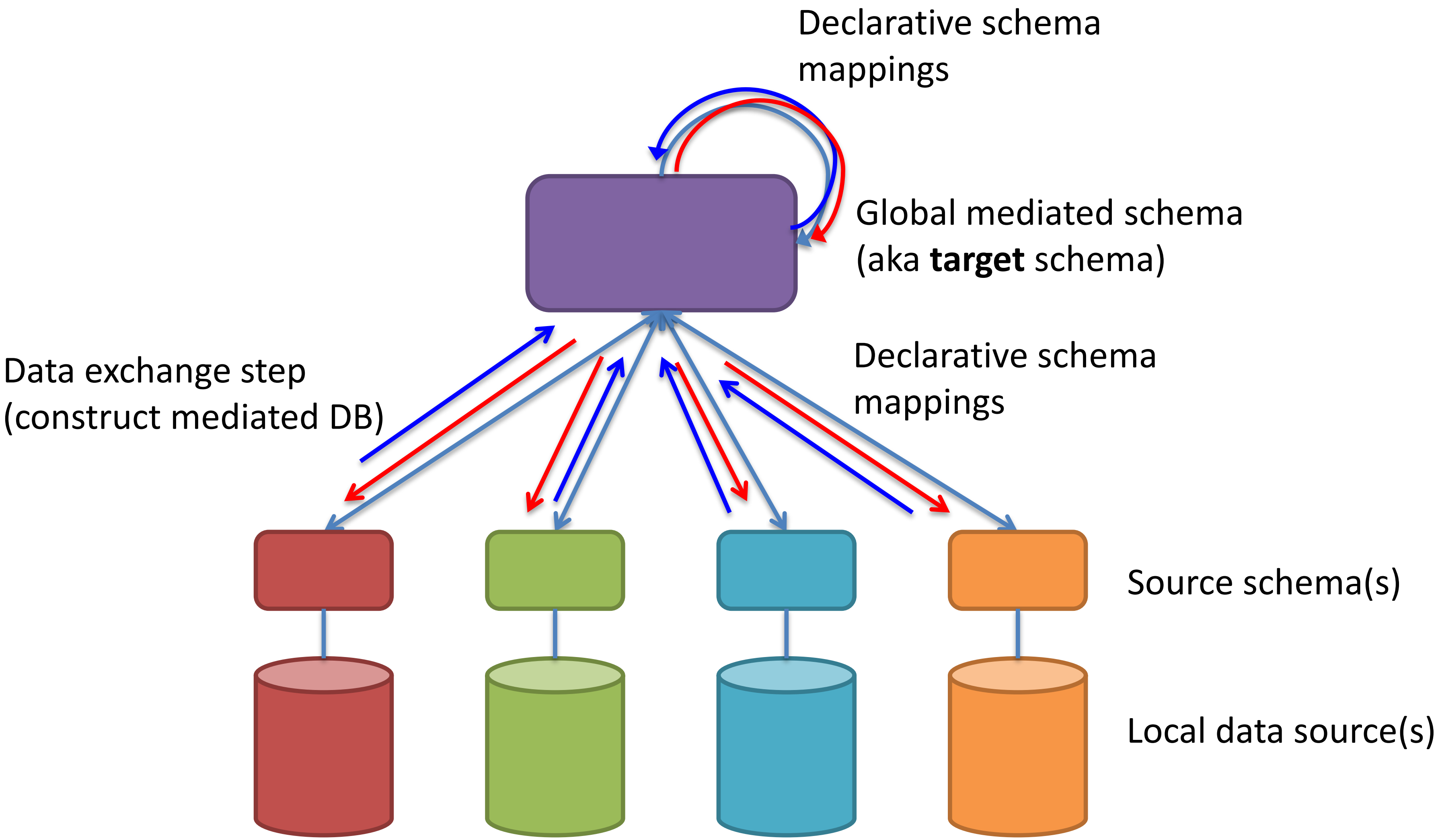
Data exchange step (construct mediated DB)

Declarative schema mappings

Source schema(s)

Local data source(s)

Materialized Data Exchange



Declarative schema mappings

Global mediated schema (aka **target** schema)

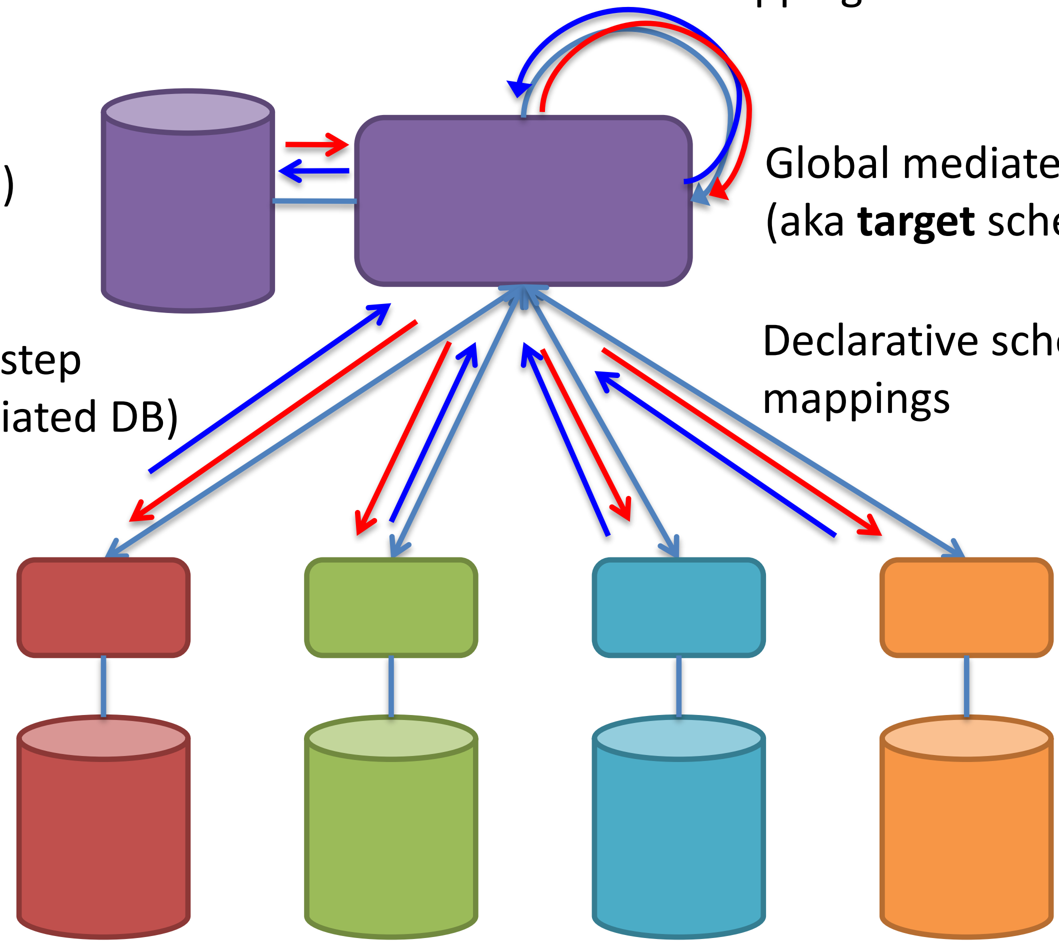
Data exchange step (construct mediated DB)

Declarative schema mappings

Source schema(s)

Local data source(s)

Materialized Data Exchange



Declarative schema mappings

Materialized mediated (target) database

Global mediated schema (aka **target** schema)

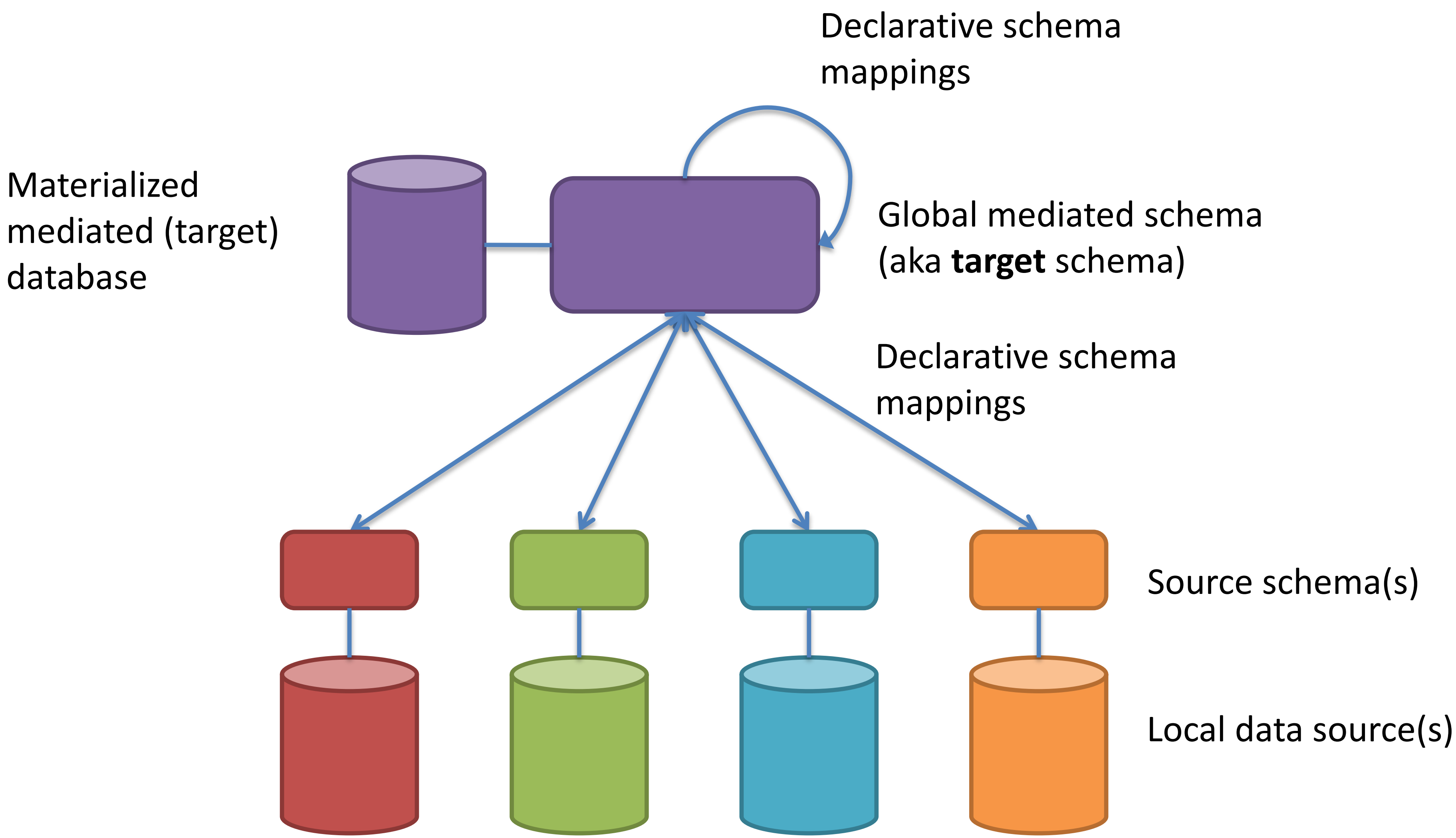
Data exchange step (construct mediated DB)

Declarative schema mappings

Source schema(s)

Local data source(s)

Materialized Data Exchange



Declarative schema mappings

Materialized mediated (target) database

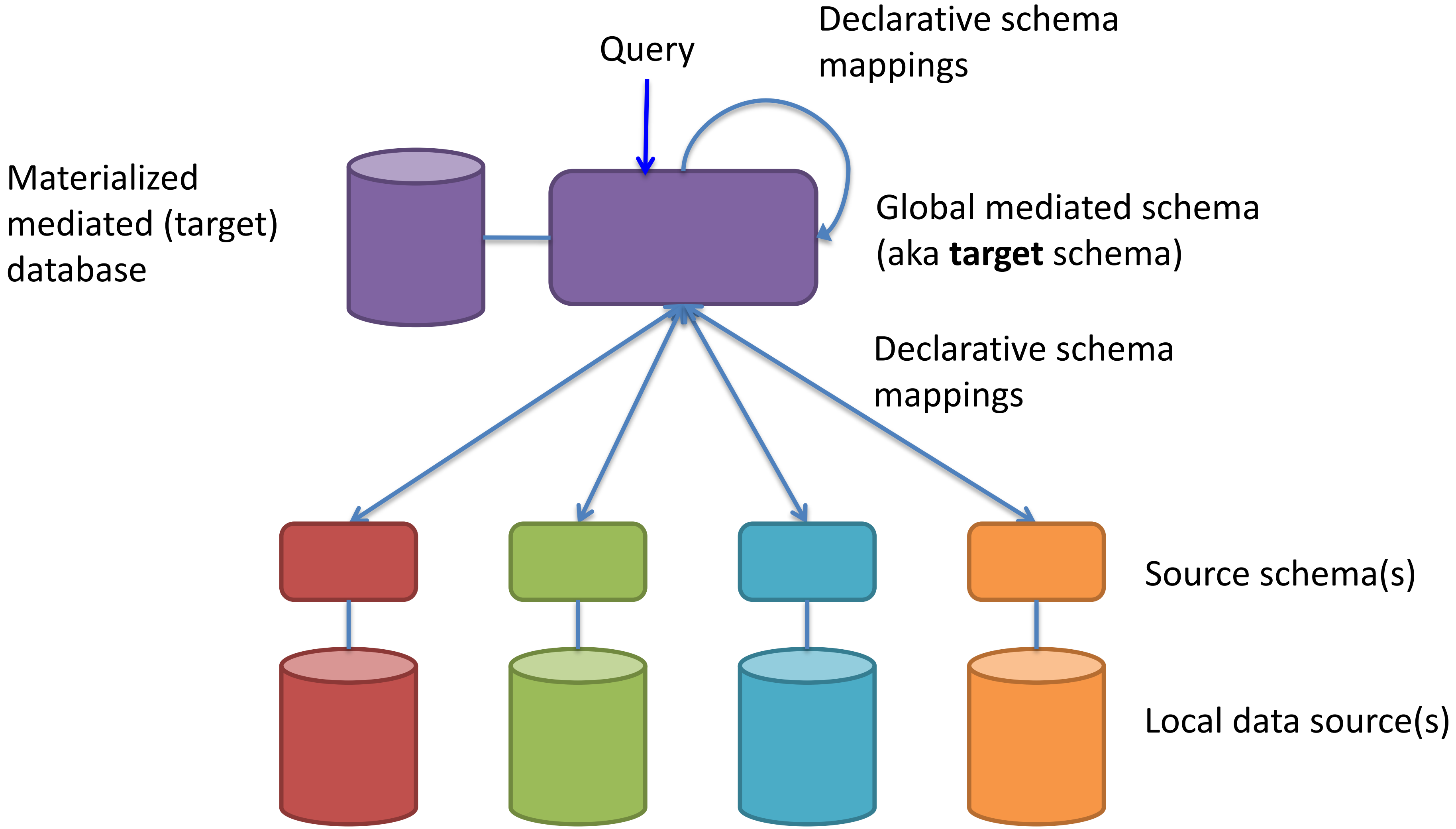
Global mediated schema (aka **target** schema)

Declarative schema mappings

Source schema(s)

Local data source(s)

Materialized Data Exchange



Query

Declarative schema mappings

Materialized mediated (target) database

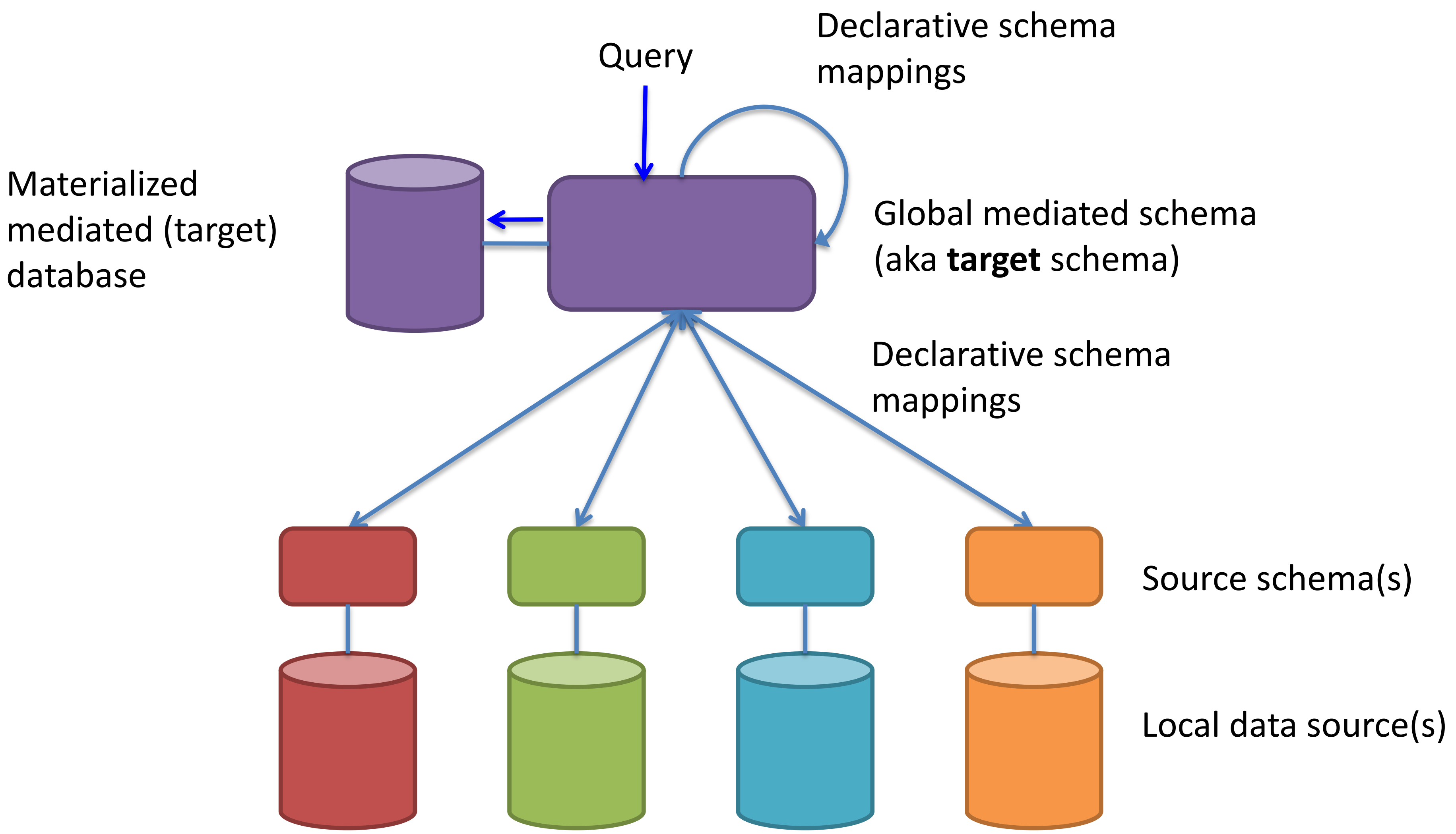
Global mediated schema (aka **target** schema)

Declarative schema mappings

Source schema(s)

Local data source(s)

Materialized Data Exchange



Query

Declarative schema mappings

Materialized mediated (target) database

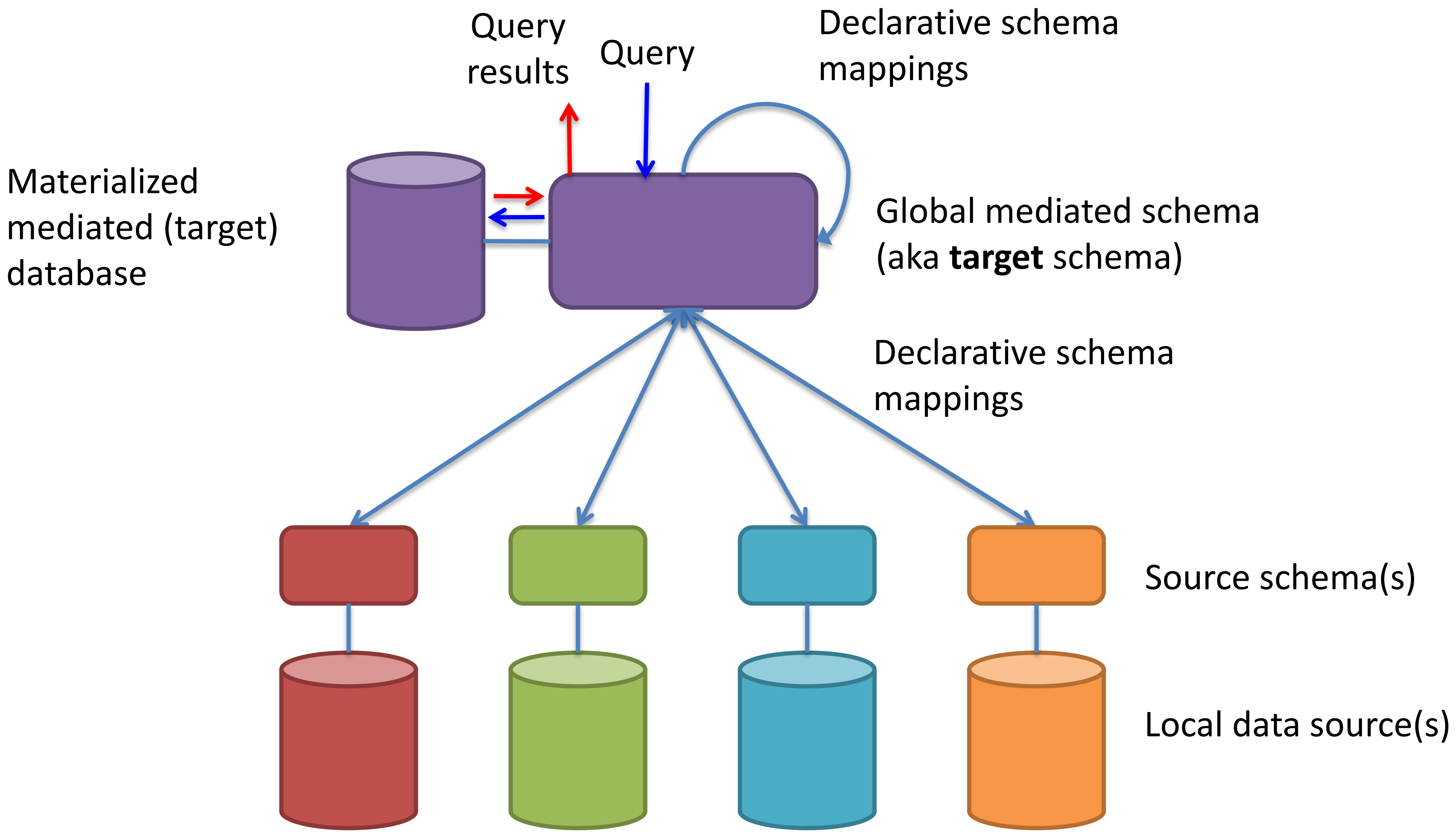
Global mediated schema (aka **target** schema)

Declarative schema mappings

Source schema(s)

Local data source(s)

Materialized Data Exchange



Query results

Query

Declarative schema mappings

Materialized mediated (target) database

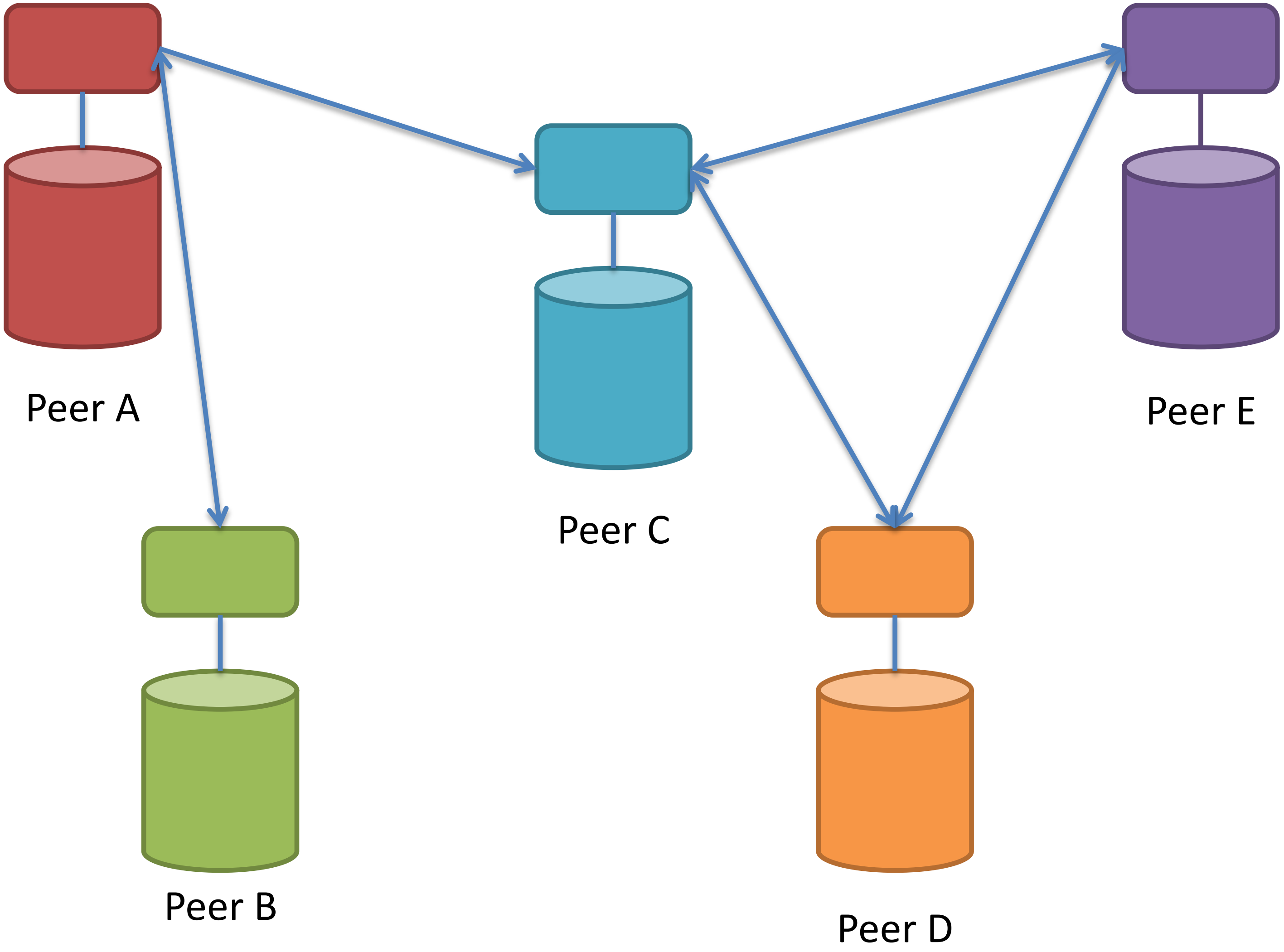
Global mediated schema (aka **target** schema)

Declarative schema mappings

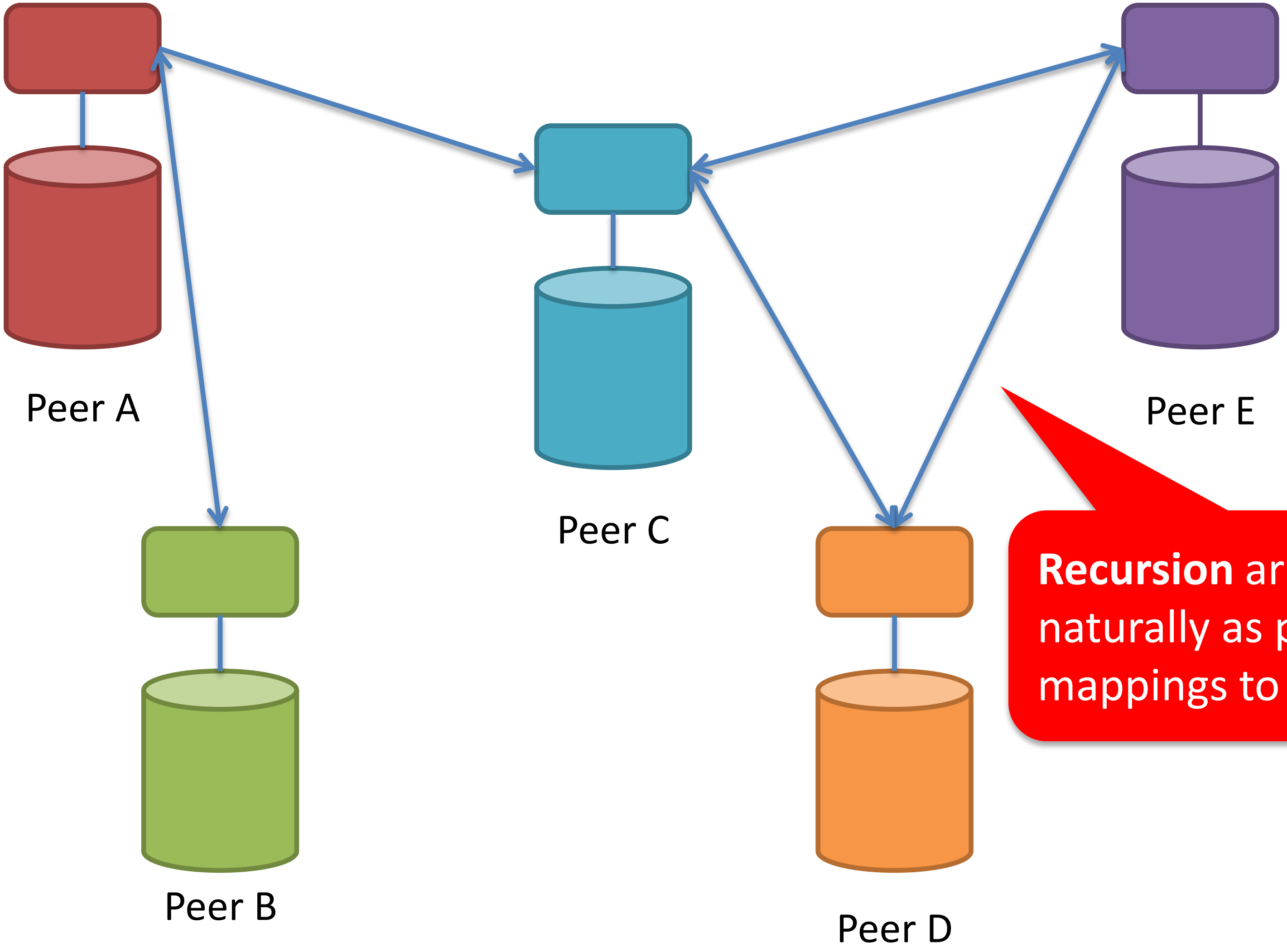
Source schema(s)

Local data source(s)

Peer-to-Peer Data Integration (Virtual or Materialized)

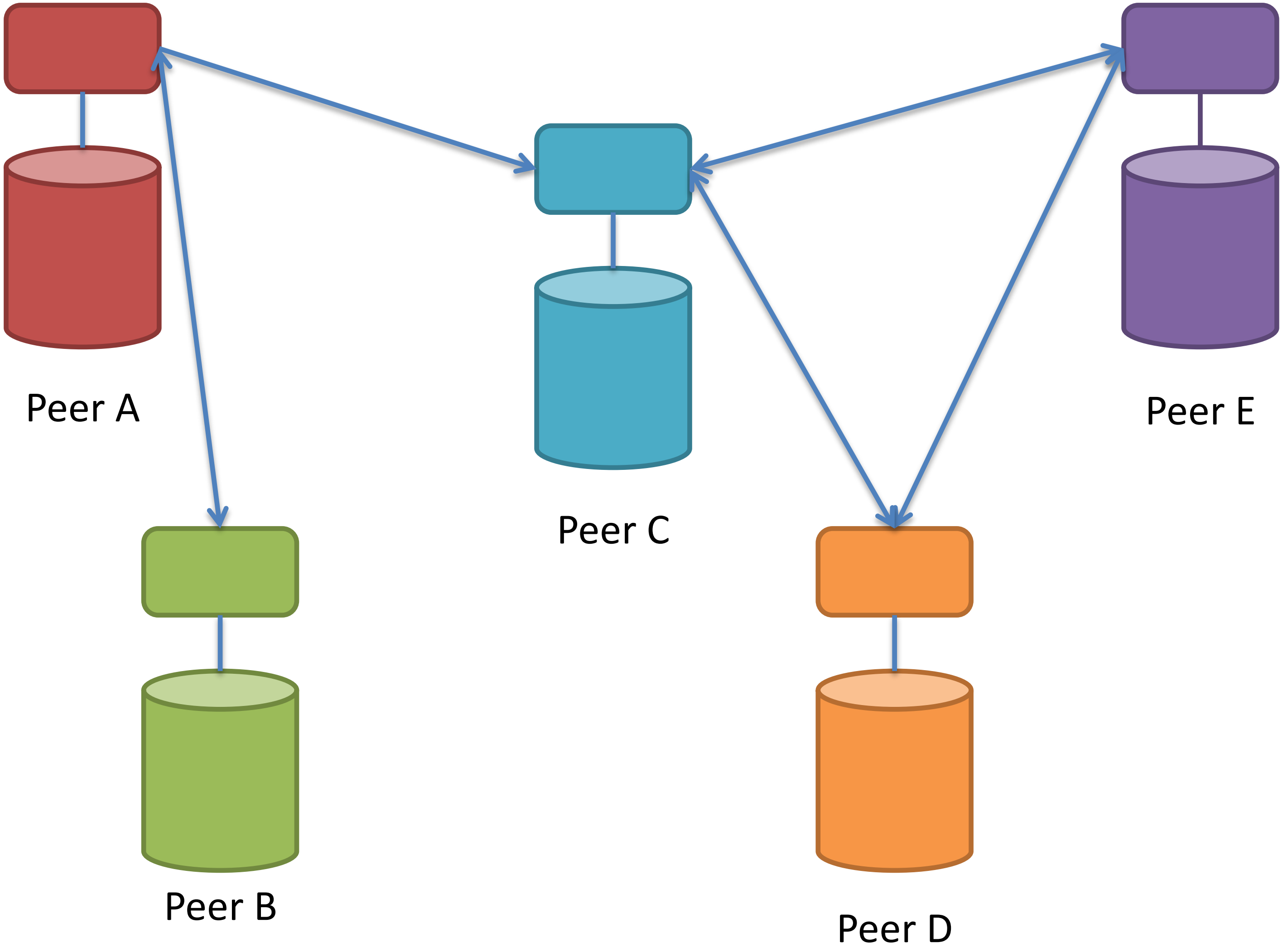


Peer-to-Peer Data Integration (Virtual or Materialized)

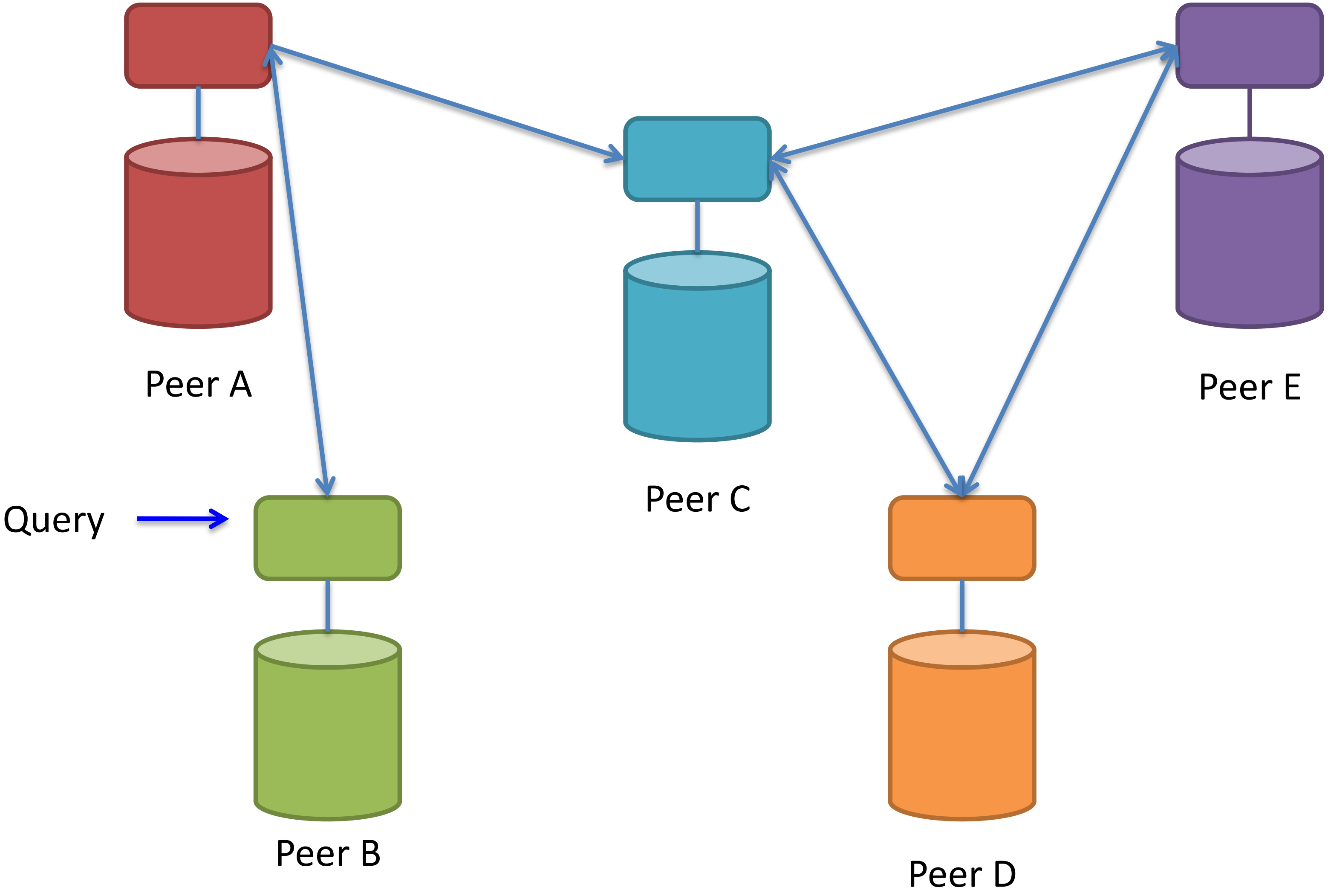


Recursion arises naturally as peers add mappings to each other

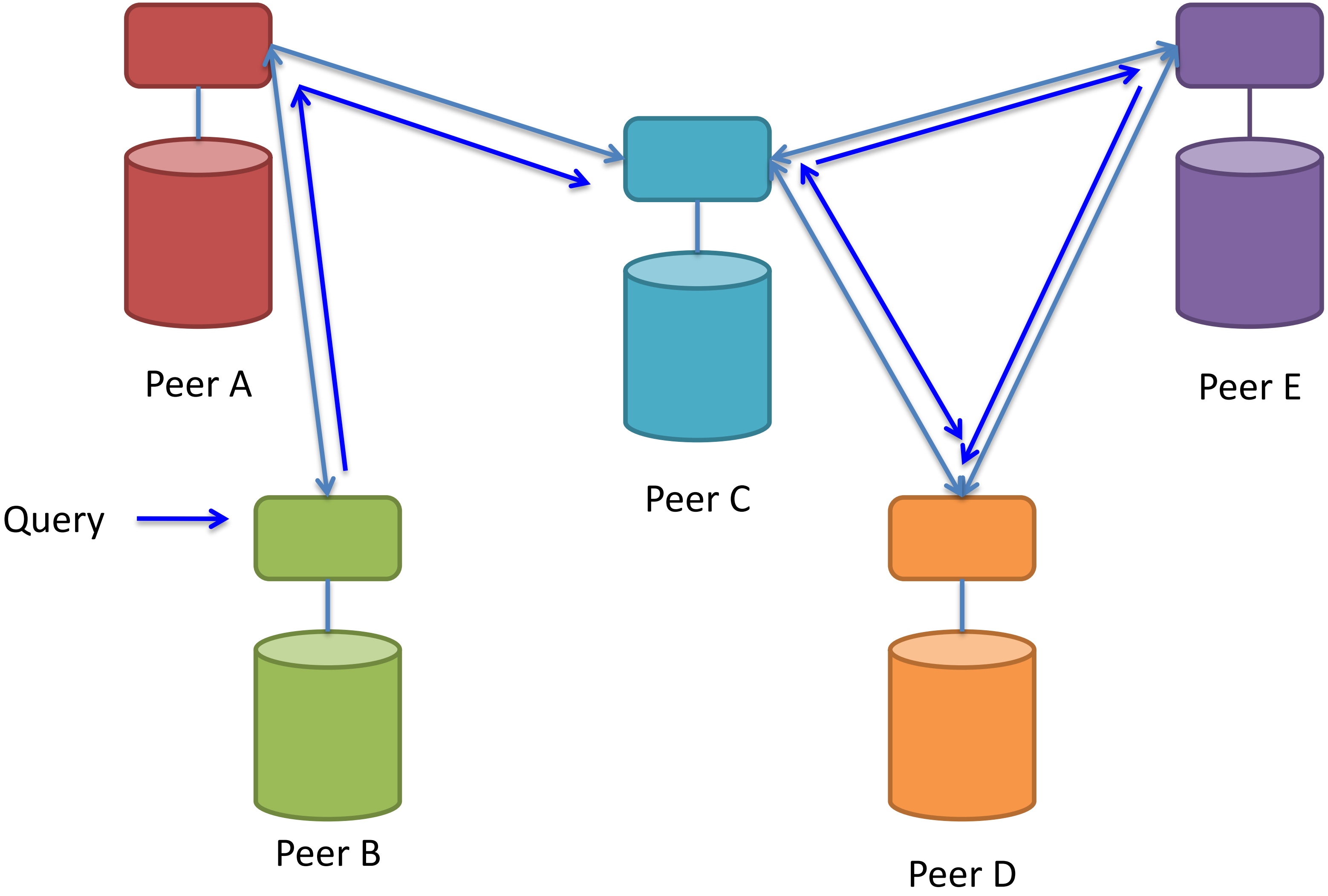
Peer-to-Peer Data Integration (Virtual or Materialized)



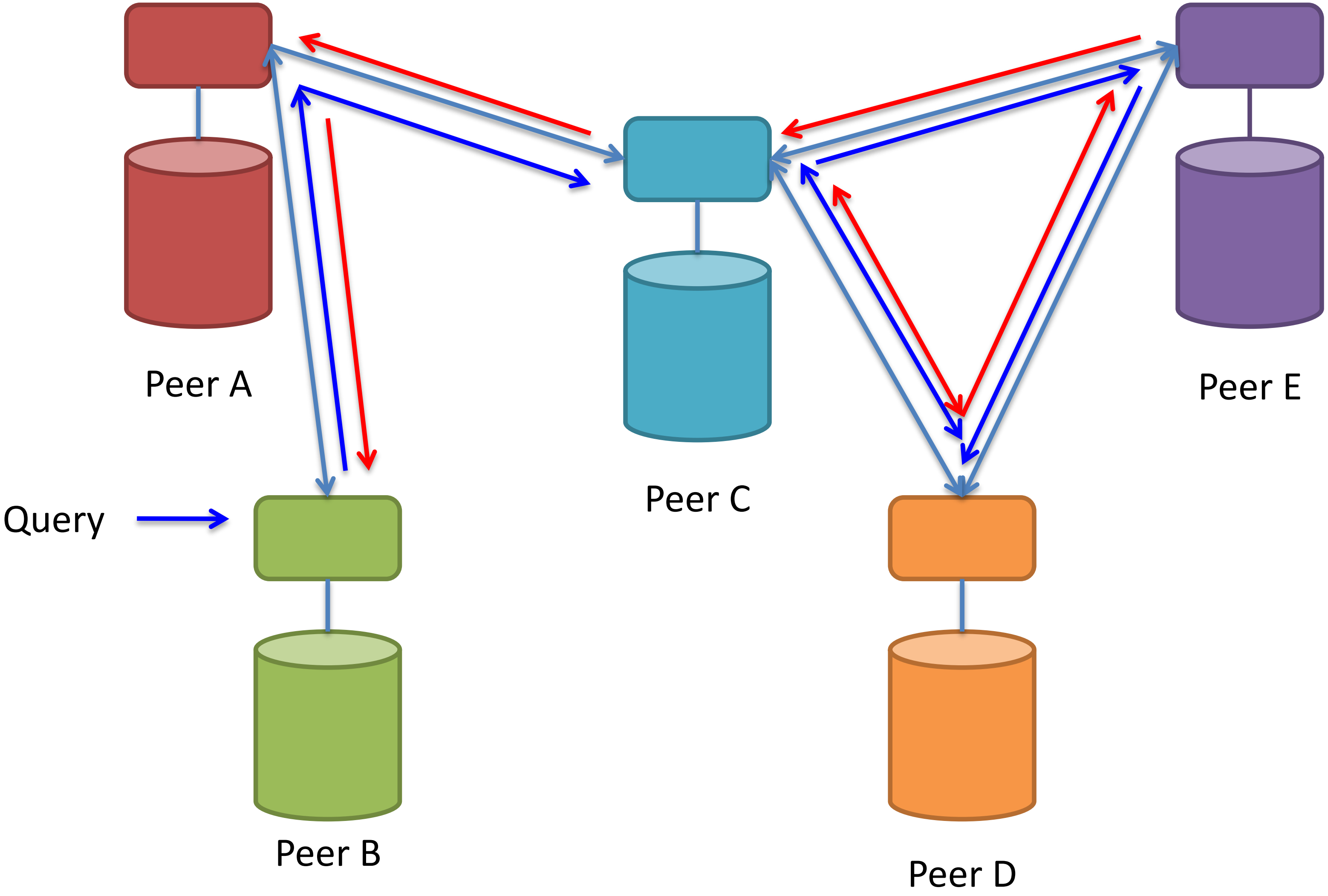
Peer-to-Peer Data Integration (Virtual or Materialized)



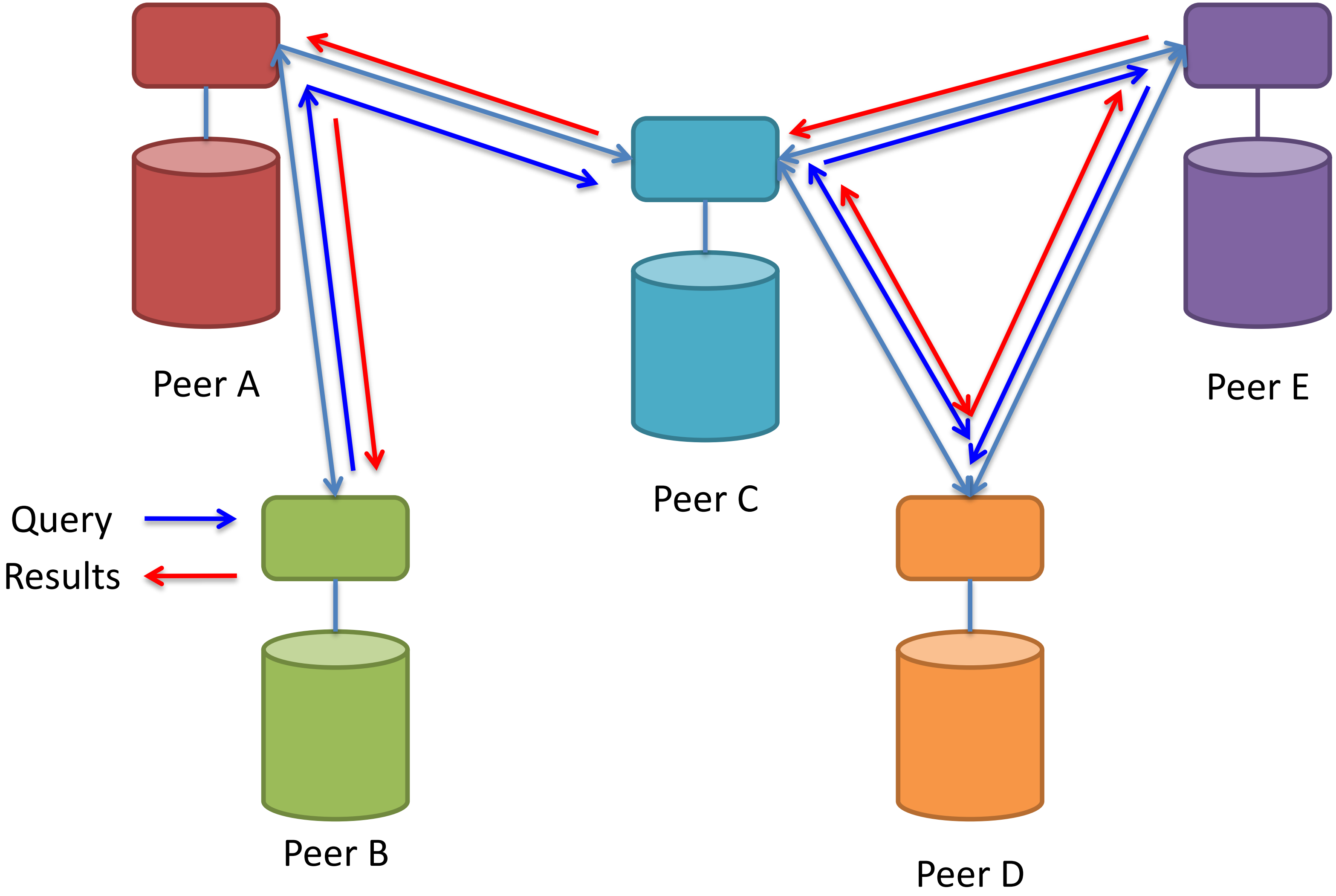
Peer-to-Peer Data Integration (Virtual or Materialized)



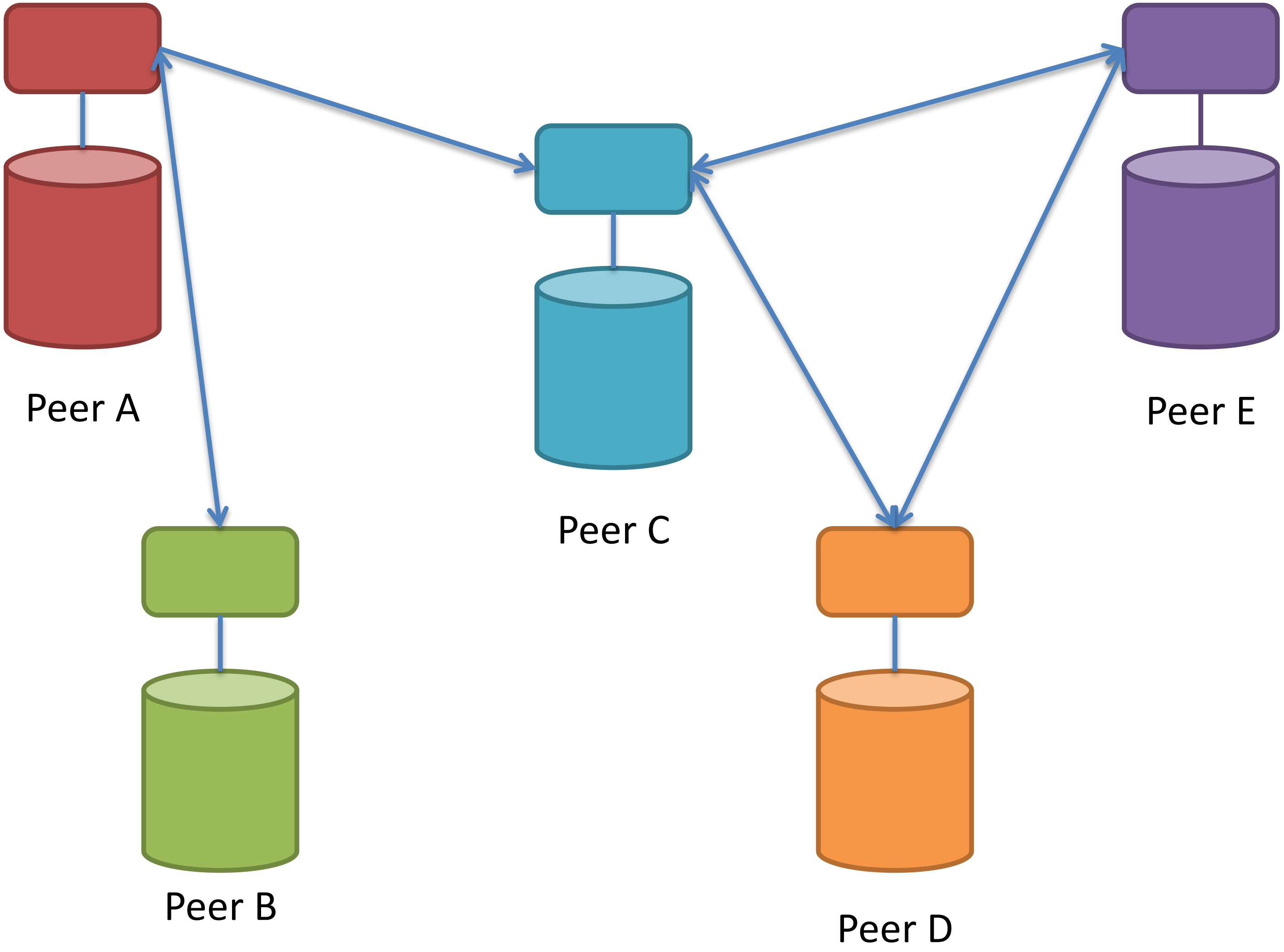
Peer-to-Peer Data Integration (Virtual or Materialized)



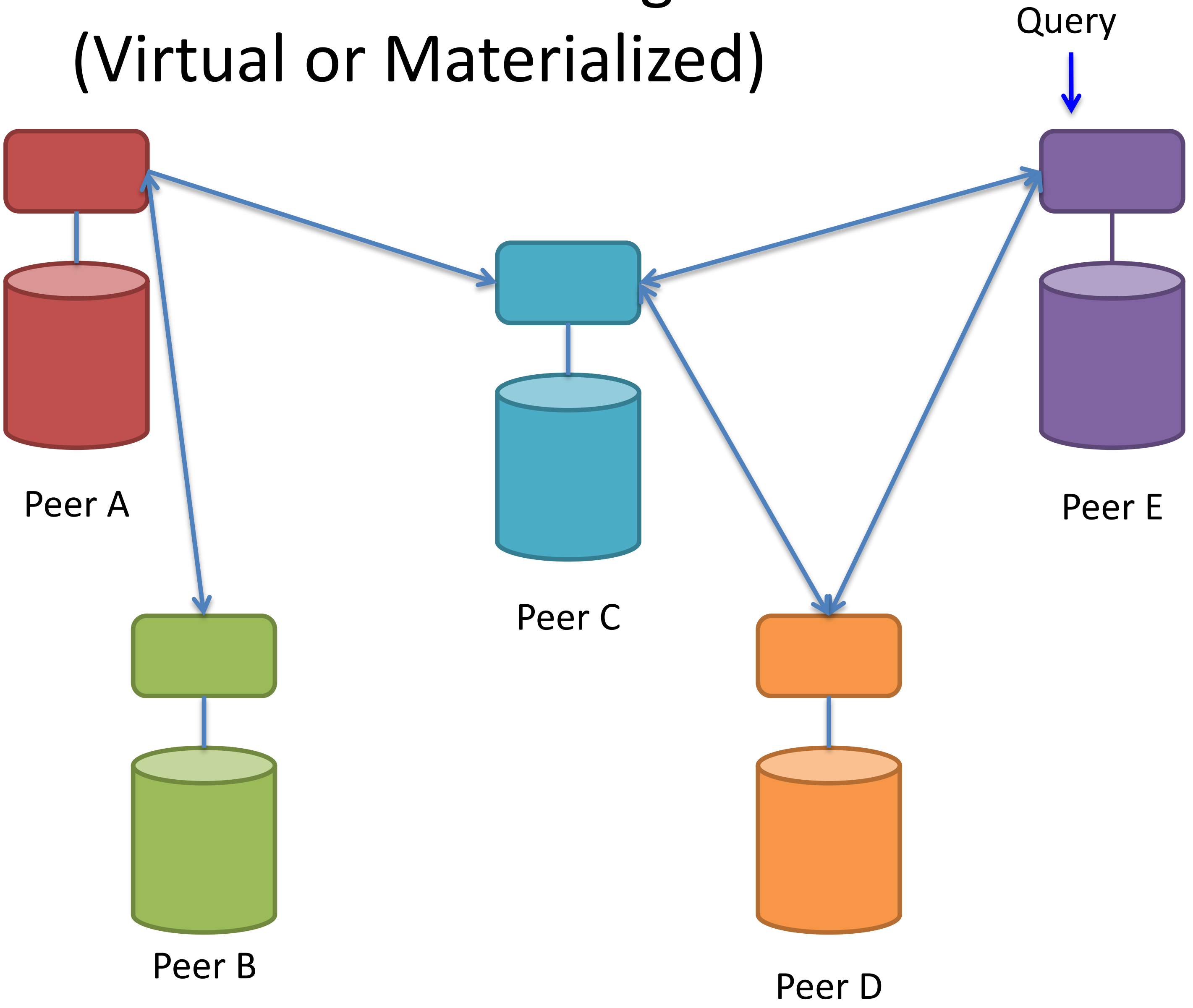
Peer-to-Peer Data Integration (Virtual or Materialized)



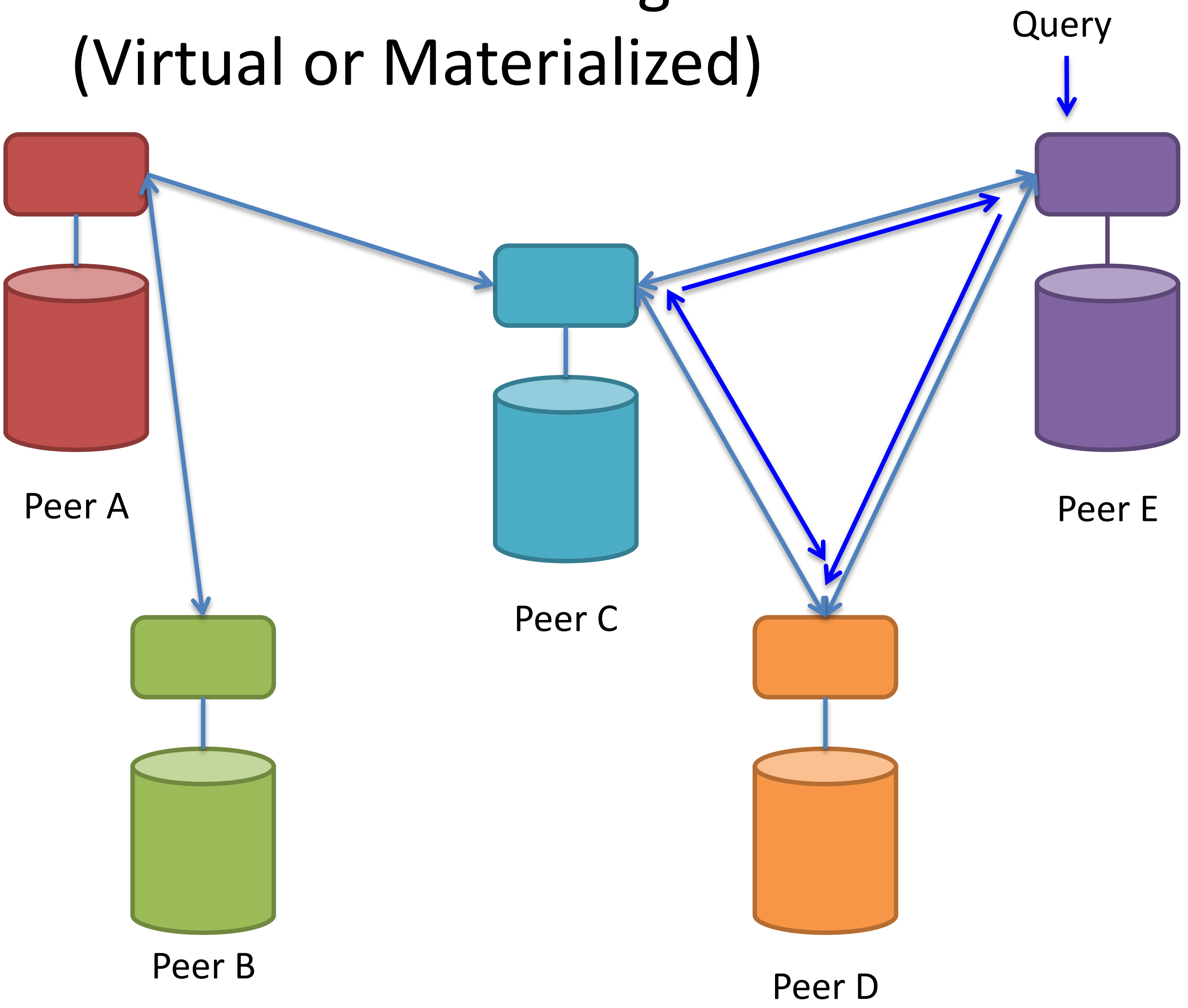
Peer-to-Peer Data Integration (Virtual or Materialized)



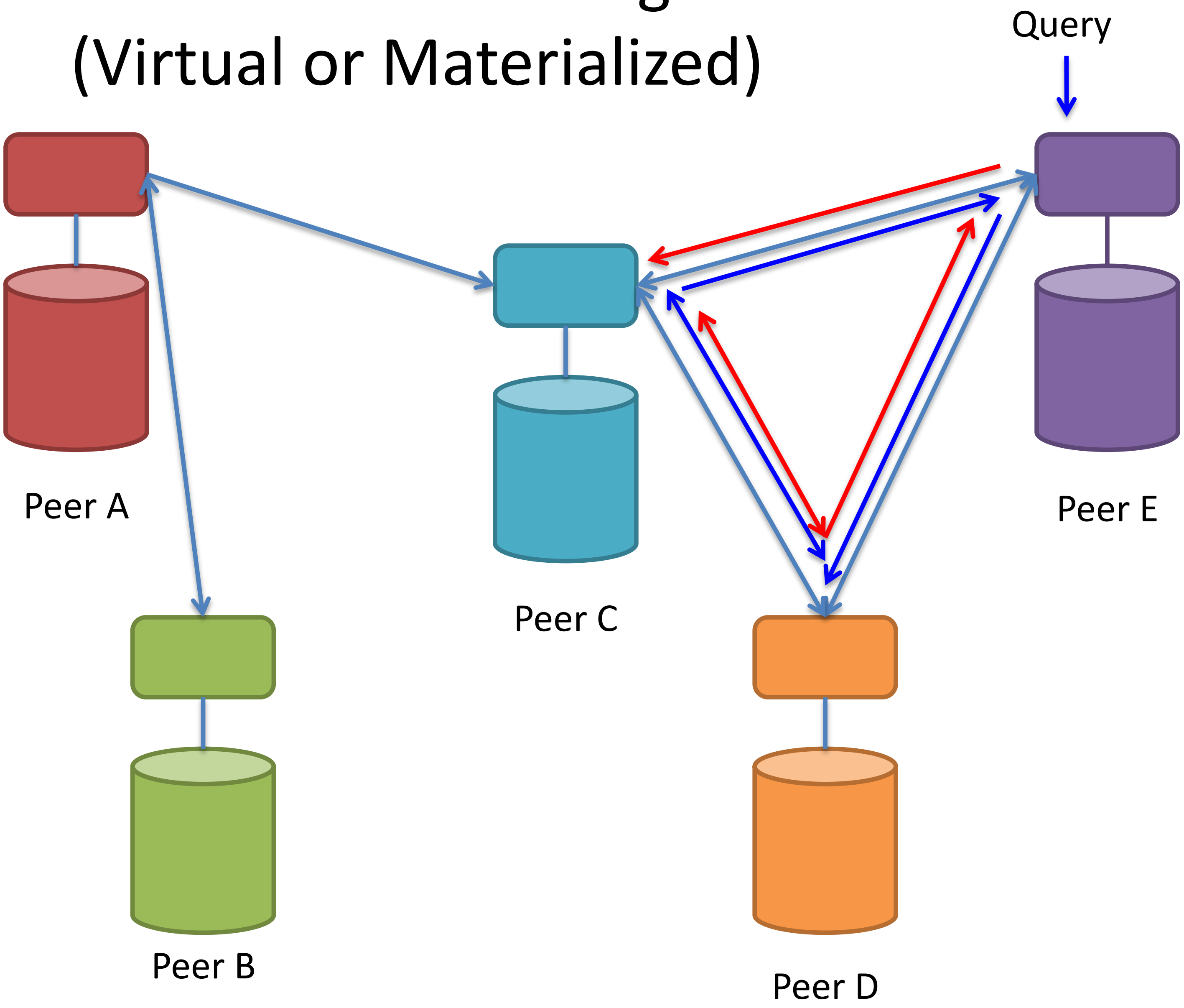
Peer-to-Peer Data Integration (Virtual or Materialized)



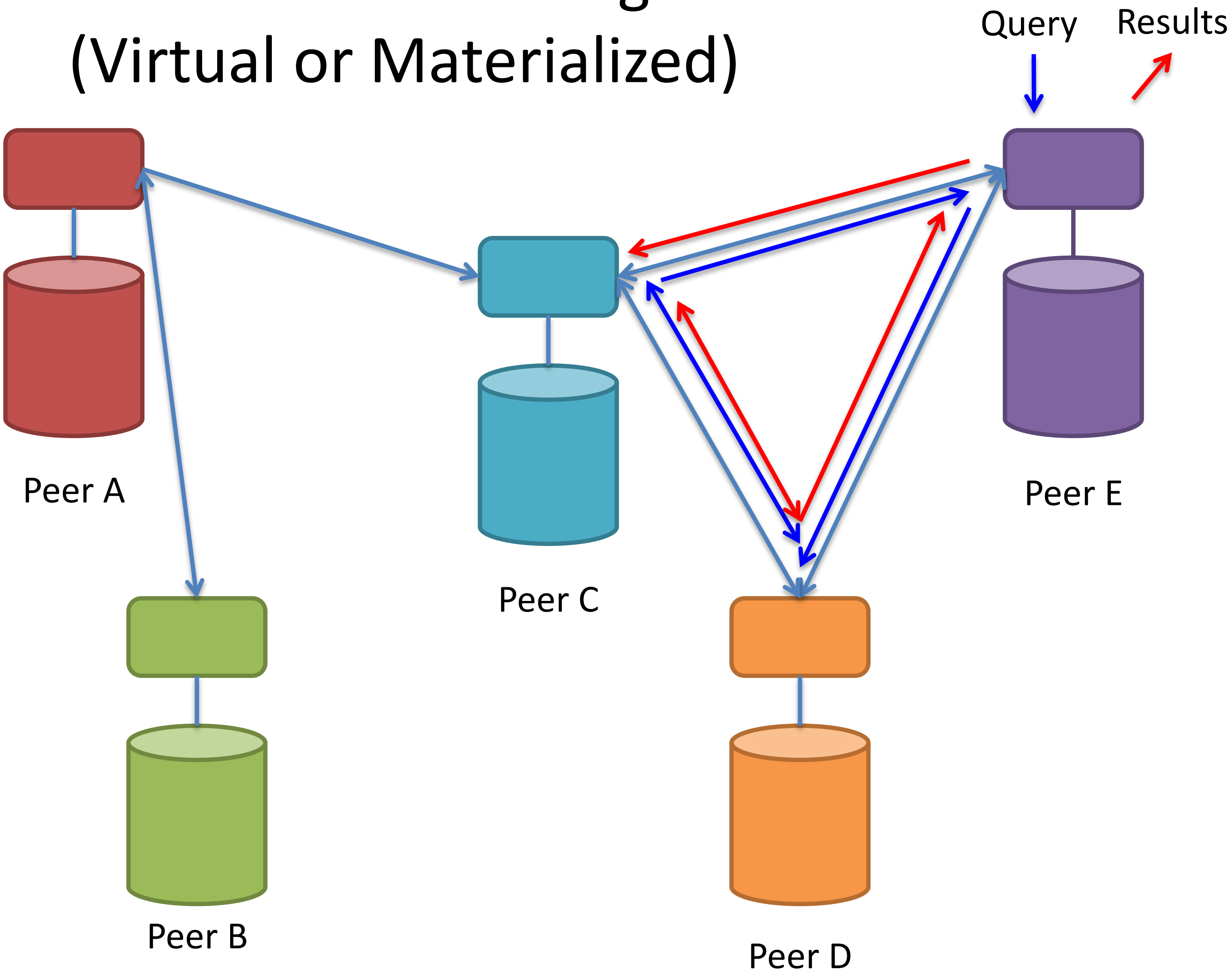
Peer-to-Peer Data Integration (Virtual or Materialized)



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

$$\forall \mathbf{X}\ \phi(\mathbf{X}) \rightarrow \exists \mathbf{Y}\ \psi(\mathbf{X},\mathbf{Y})$$

where ϕ and ψ 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

$$\forall X\ \phi(\mathbf{X}) \rightarrow \exists \mathbf{Y}\ \psi(\mathbf{X},\mathbf{Y})$$

where ϕ and ψ are **conjunctions** of relational atoms

For example:

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

"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.

CONSTRAINT:

$$\begin{array}{c}\forall\ \text{Eid, Name, Addr}\ \ \textbf{employee}(\text{Eid, Name, Addr}) \rightarrow \\ \exists\ \text{Ssn}\ \ \textbf{name}(\text{Ssn, Name}) \wedge \textbf{address}(\text{Ssn, Addr})\end{array}$$

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:

$$\forall\ \mathrm{Eid, Name, Addr}\ \ \textbf{employee}(\mathrm{Eid, Name, Addr}) \rightarrow$$

$$\exists\ \mathrm{Ssn}\ \ \textbf{name}(\mathrm{Ssn, Name}) \wedge \textbf{address}(\mathrm{Ssn, Addr})$$

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

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:

∀ Eid, Name, Addr **employee**(Eid, Name, Addr) →

∃ Ssn **name**(Ssn, Name) ∧ **address**(Ssn, Addr)

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

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:

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

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

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:

$$\forall \text{ Eid, Name, Addr } \; \textbf{employee}(\text{Eid, Name, Addr}) \rightarrow$$

$$\exists \text{ Ssn } \; \textbf{name}(\text{Ssn, Name}) \wedge \textbf{address}(\text{Ssn, Addr})$$

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

...ETC...

…

…

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:

$\forall$ Eid, Name, Addr **employee**(Eid, Name, Addr) → $\exists$ Ssn **name**(Ssn, Name) $\wedge$ **address**(Ssn, Addr)

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 M
42	16 E

...ETC...

...

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.

CONSTRAINT:

∀ Eid, Name, Addr **employee**(Eid, Name, Addr) →

∃ Ssn **name**(Ssn, Name) ∧ **address**(Ssn, Addr)

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 M
42	16 E

...ETC...

...

Which mediated DB should be materialized?

QUERY:

q(Name) <- **name**(Ssn, Name), **address**(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:

∀ Eid, Name, Addr **employee**(Eid, Name, Addr) →

∃ Ssn **name**(Ssn, Name) ∧ **address**(Ssn, Addr)

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 M
42	16 E

...ETC...

...

What answers should q return?

Which mediated DB should be materialized?

QUERY:

q(Name) <- **name**(Ssn, Name), **address**(Ssn, _).

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

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

...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

employee

17	Alice	1 Main St
23	Bob	16 Elm St

QUERY:

q(Name) <-
  name(Ssn, Name),
  address(Ssn, _).

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

...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

employee

17	Alice	1 Main St
23	Bob	16 Elm St

QUERY:

```
q(Name) <-
   name(Ssn, Name),
   address(Ssn, _).
```

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

q

Alice
Bob
Carol

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

...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

employee

17	Alice	1 Main St
23	Bob	16 Elm St

QUERY:

q(Name) <-
  name(Ssn, Name),
  address(Ssn, _).

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

q

Alice
Bob
Carol

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

q

Alice
Bob

...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

employee

17	Alice	1 Main St
23	Bob	16 Elm St

QUERY:

```
q(Name) <-
   name(Ssn, Name),
   address(Ssn, _).
```

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

q

Alice
Bob
Carol

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

q

Alice
Bob

...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

employee

17	Alice	1 Main St
23	Bob	16 Elm St

QUERY:

```
q(Name) <-
   name(Ssn, Name),
   address(Ssn, _).
```

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

q

Alice
Bob
Carol

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

q

Alice
Bob

...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

employee

17	Alice	1 Main St
23	Bob	16 Elm St

QUERY:

q(Name) <-
  name(Ssn, Name),
  address(Ssn, _).

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

q

Alice
Bob
Carol

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

q

Alice
Bob

...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

employee

17	Alice	1 Main St
23	Bob	16 Elm St

QUERY:

```
q(Name) <-
   name(Ssn, Name),
   address(Ssn, _).
```

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

...ETC...

…

…

certain answers to q

Alice
Bob

=

q

Alice
Bob
Carol

∩

q

Alice
Bob

∩ …

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)

tgd:

$$\forall X, Y, Z\ \mathbf{foo}(X,Y) \land \mathbf{bar}(X,Z) \rightarrow \mathbf{biz}(Y,Z) \land \mathbf{baz}(Z)$$

Datalog rules:

biz(X,Y,Z) <- **foo**(X,Y), **bar**(X,Z).
baz(Z) <- **foo**(X,Y), **bar**(X,Z).

Inverse Rules: Computing Certain Answers with Datalog

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

tgd:

∀ X, Y, Z **foo**(X,Y) ∧ **bar**(X,Z) → **biz**(Y,Z) ∧ **baz**(Z)

Datalog rules:

biz(X,Y,Z) <- **foo**(X,Y), **bar**(X,Z).
baz(Z) <- **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)

tgd:

$$\forall X, Y, Z\ \mathbf{foo}(X,Y) \land \mathbf{bar}(X,Z) \rightarrow \mathbf{biz}(Y,Z) \land \mathbf{baz}(Z)$$

Datalog rules:

biz(X,Y,Z) <- **foo**(X,Y), **bar**(X,Z).
baz(Z) <- **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)

tgd:

$$\forall\ X, Y, Z\ \mathbf{foo}(X,Y) \wedge \mathbf{bar}(X,Z) \rightarrow \mathbf{biz}(Y,Z) \wedge \mathbf{baz}(Z)$$

Datalog rules:

biz(X,Y,Z) <- **foo**(X,Y), **bar**(X,Z).
baz(Z) <- **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

The catch: what to do about **existentially quantified variables**...

Inverse Rules: Computing Certain Answers with Datalog (2)

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

$$\forall\ \mathit{Eid}, \mathit{Name}, \mathit{Addr}\ \ \textbf{employee}(\mathit{Eid}, \mathit{Name}, \mathit{Addr}) \rightarrow$$

$$\exists\ \mathit{Ssn}\ \ \textbf{name}(\mathit{Ssn}, \mathit{Name}) \land \textbf{address}(\mathit{Ssn}, \mathit{Addr})$$

Inverse Rules: Computing Certain Answers with Datalog (2)

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

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

- 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

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

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

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

Inverse Rules: Computing Certain Answers with Datalog (2)

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

∀ Eid, Name, Addr **employee**(Eid, Name, Addr) →

∃ Ssn **name**(Ssn, Name) ∧ **address**(Ssn, Addr)

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

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

ssn is a Skolem function

Inverse Rules: Computing Certain Answers with Datalog (2)

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

∀ Eid, Name, Addr **employee**(Eid, Name, Addr) →

∃ Ssn **name**(Ssn, Name) ∧ **address**(Ssn, Addr)

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

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

Inverse Rules: Computing Certain Answers with Datalog (2)

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

∀ Eid, Name, Addr **employee**(Eid, Name, Addr) →

∃ Ssn **name**(Ssn, Name) ∧ **address**(Ssn, Addr)

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

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

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

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

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

- Unlike SQL nulls, can join on Skolem values:

```
query _(Name,Addr) <-
        name(Ssn,Name),
        address(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

    ssn(“Alice”, “1 Main St”)

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

    ssn(“Alice”, “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

    ssn("Alice", "1 Main St")

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

    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

    ssn("Alice", "1 Main St")

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

    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...)
- But: the models may now be **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”

```
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).
```

employee

17	Alice	1 Main St
23	Bob	16 Elm St

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).
```

employee

17	Alice	1 Main St
23	Bob	16 Elm St

manager

m(Alice)
m(Bob)
m(m(Alice))
m(m(Bob))
m(m(m(Alice)))
…

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).
```

employee

17	Alice	1 Main St
23	Bob	16 Elm St

manager

m(Alice)
m(Bob)
m(m(Alice))
m(m(Bob))
m(m(m(Alice)))
…

- 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”

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

employee

17	Alice	1 Main St
23	Bob	16 Elm St

manager

m(Alice)
m(Bob)
m(m(Alice))
m(m(Bob))
m(m(m(Alice)))
...

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

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:

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

(employee, 2)

vertex for each (predicate, index)

(employee, 1)

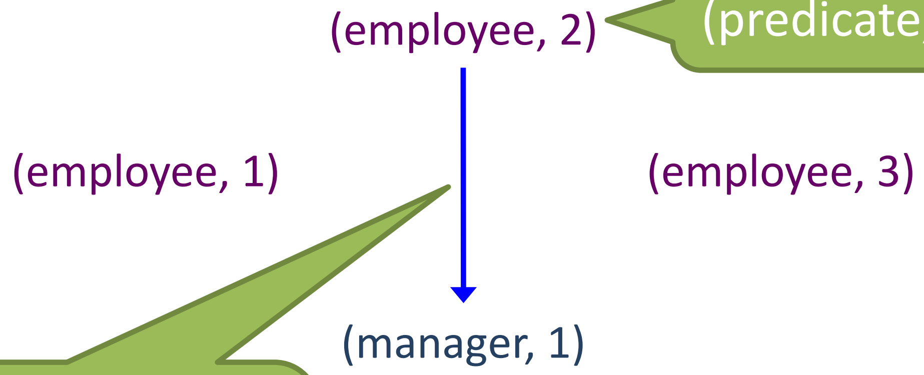
(employee, 3)

(manager, 1)

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).
```



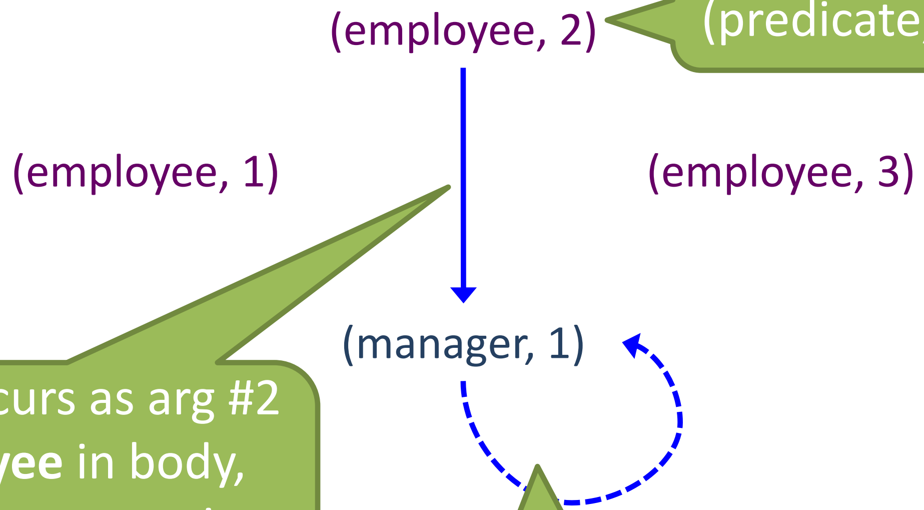
vertex for each (predicate, index)

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)

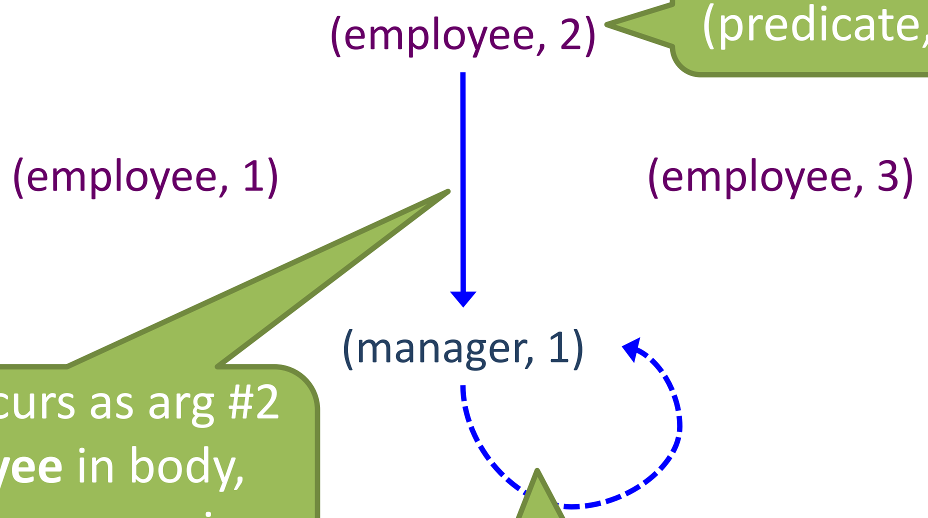
variable occurs as arg #2 to **employee** in body, arg #1 to **manager** in head

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:

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



vertex for each (predicate, index)

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

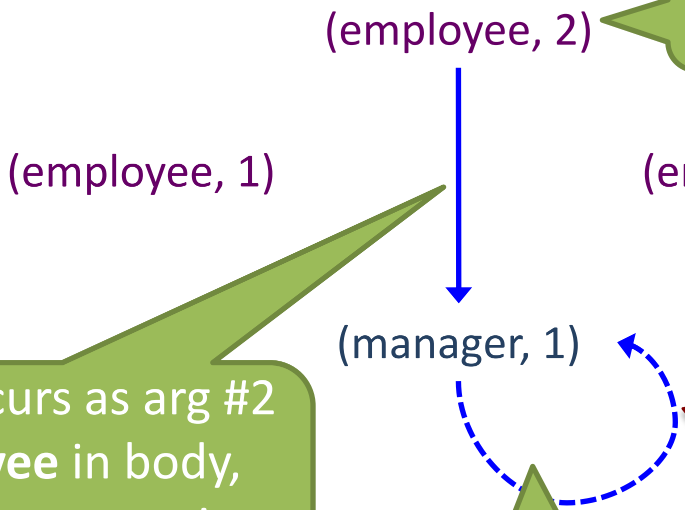
variable occurs as arg #1 to **manager** in body and as argument to Skolem (hence dashes) in arg #1 to **manager** in head

- 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:

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



vertex for each (predicate, index)

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

Cycle through dashed edge! Not weakly acyclic ☹

variable occurs as arg #1 to **manager** in body and as argument to Skolem (hence dashes) in arg #1 to **manager** in head

- If graph contains no cycle through a dashed edge, then P is called **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:

```
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:

```
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).
```

(emp, 2) (emp, 3)

(emp, 1)

(name, 1) (addr, 1)

(name, 2) (addr, 2)

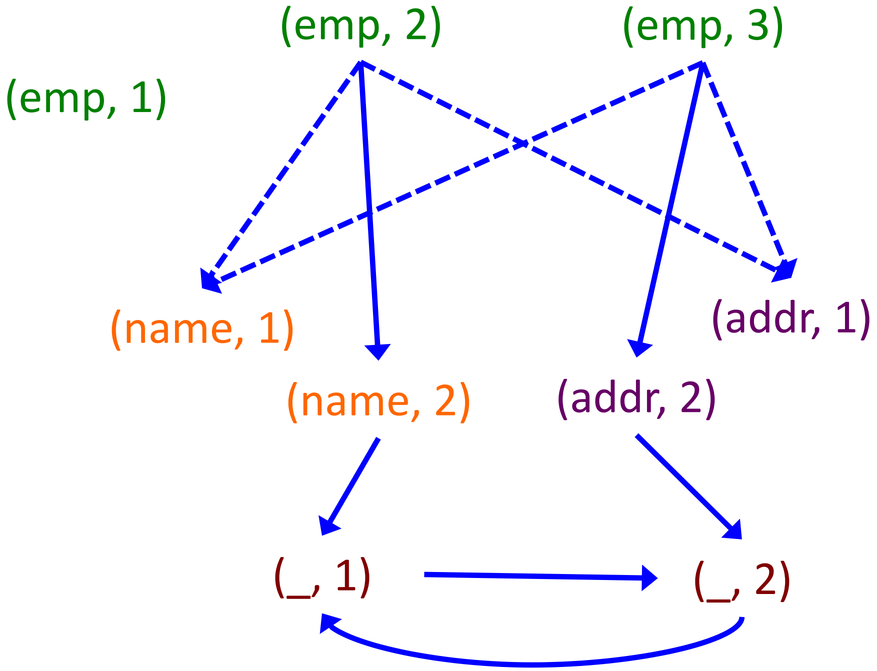
(_, 1) (_, 2)

Ensuring Termination via Weak Acyclicity (2)

- Another example, this one weakly acyclic:

```
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).
```



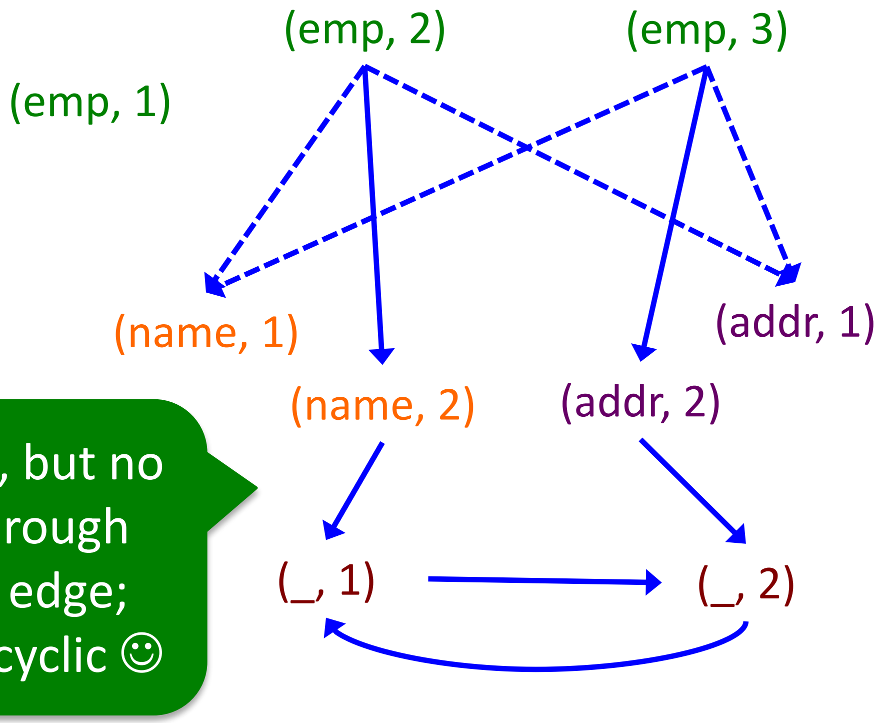
Ensuring Termination via Weak Acyclicity (2)

- Another example, this one weakly acyclic:

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

query _(Name,Addr)
          <- name(Ssn,Nar
            address(Ssn,Ac
            _(Addr,Name).
```

has cycle, but no cycle through dashed edge; weakly acyclic ☺



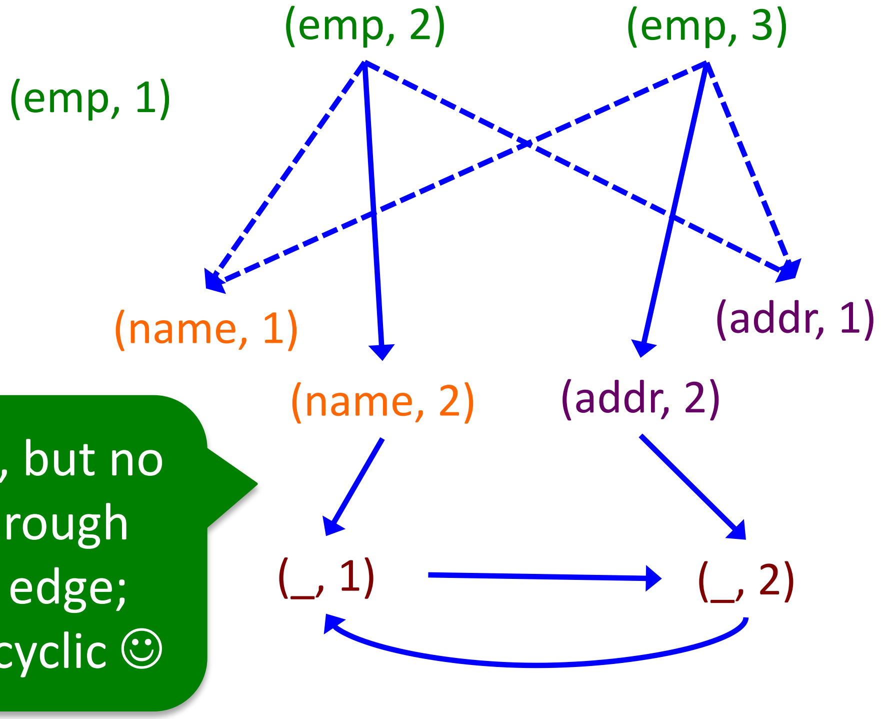
Ensuring Termination via Weak Acyclicity (2)

- 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,Nar
            address(Ssn,Ad
            _(Addr,Name).
```

has cycle, but no cycle through dashed edge; weakly acyclic ☺



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:

∀ 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).
```

Once Computation Stops, What Do We Have?

tgd:

∀ 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).
```

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

Once Computation Stops, What Do We Have?

tgd:

∀ 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).
```

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #2

name

ssn(A..)	Alice
ssn(B..)	Bob

address

ssn(A..)	1 Main St
ssn(B..)	16 Elm St

Once Computation Stops, What Do We Have?

tgd:

∀ 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).

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

ssn(A..)	Alice
ssn(B..)	Bob

address

ssn(A..)	1 Main St
ssn(B..)	16 Elm St

Once Computation Stops, What Do We Have?

tgd:

∀ 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).
```

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

ssn(A..)	Alice
ssn(B..)	Bob

address

ssn(A..)	1 Main St
ssn(B..)	16 Elm St

MEDIATED DB #3

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

…

…

Once Computation Stops, What Do We Have?

tgd:

∀ 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).
```

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

ssn(A..)	Alice
ssn(B..)	Bob

address

ssn(A..)	1 Main St
ssn(B..)	16 Elm St

MEDIATED DB #3

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

…

…

Among all the mediated DB instances satisfying the constraints (**solutions**), #2 above is **universal**: can be homomorphically embedded in **any** other solution.

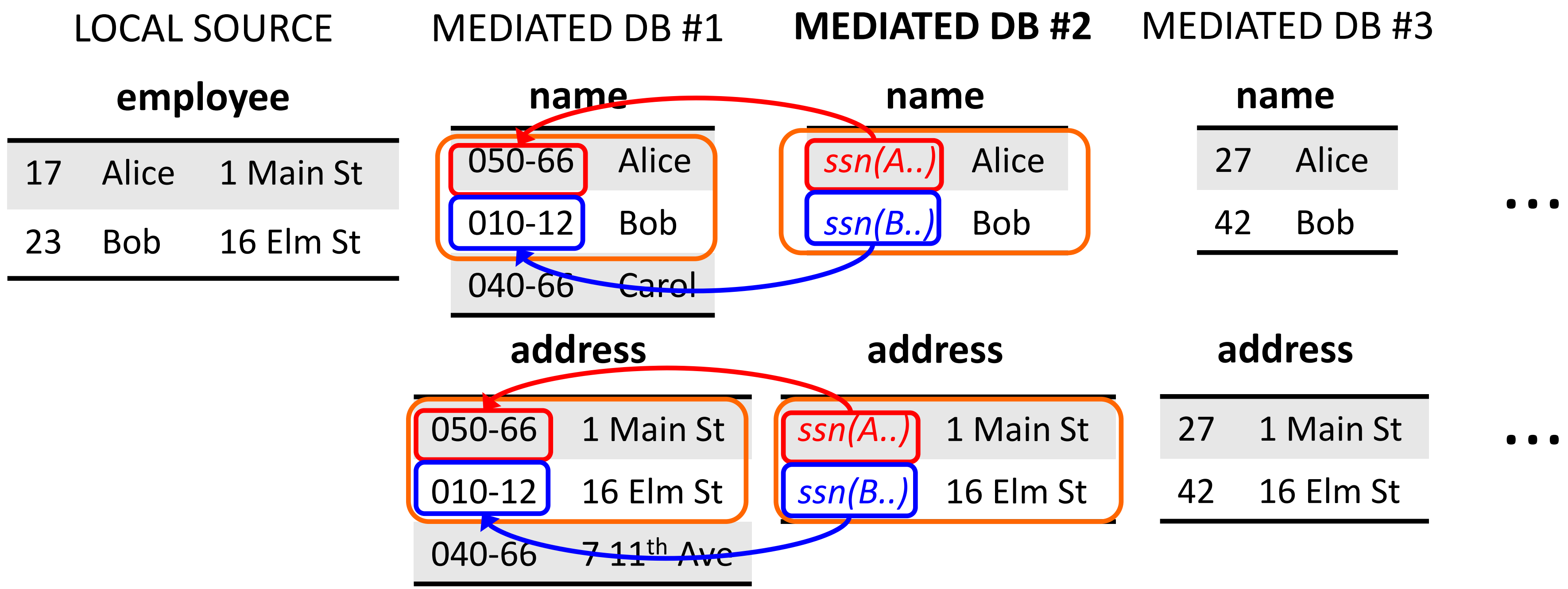
Once Computation Stops, What Do We Have?

tgd:

∀ 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).



LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2

name

ssn(A..)	Alice
ssn(B..)	Bob

address

ssn(A..)	1 Main St
ssn(B..)	16 Elm St

MEDIATED DB #3

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

…

…

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?

tgd:

∀ 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).

LOCAL SOURCE

employee

17	Alice	1 Main St
23	Bob	16 Elm St

MEDIATED DB #1

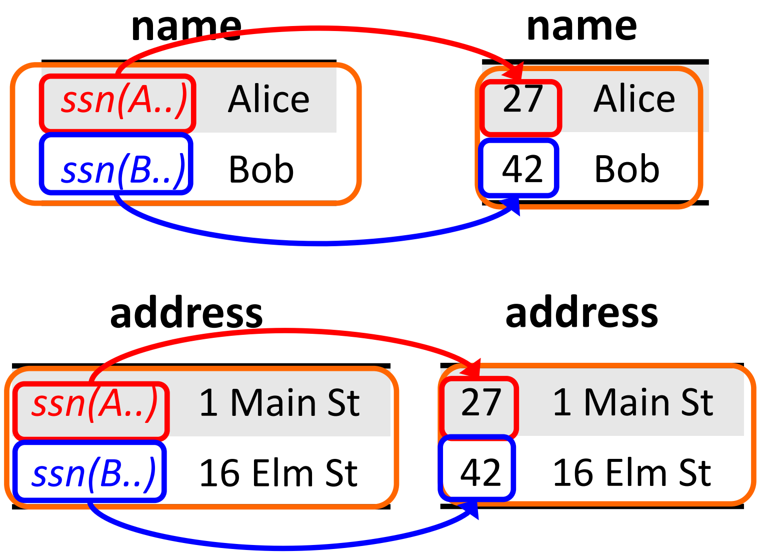
name

050-66	Alice
010-12	Bob
040-66	Carol

address

050-66	1 Main St
010-12	16 Elm St
040-66	7 11th Ave

MEDIATED DB #2 MEDIATED DB #3



name

ssn(A..)	Alice
ssn(B..)	Bob

address

ssn(A..)	1 Main St
ssn(B..)	16 Elm St

name

27	Alice
42	Bob

address

27	1 Main St
42	16 Elm St

…

…

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

AT&T
Bell Laboratories

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

IBM

UNIVERSITY OF
TORONTO

- Orchestra CDSS [Ives+ 05]
  - Materialized approach
  - Skolem terms, recursion, provenance, updates

ORCHESTRA

Penn

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

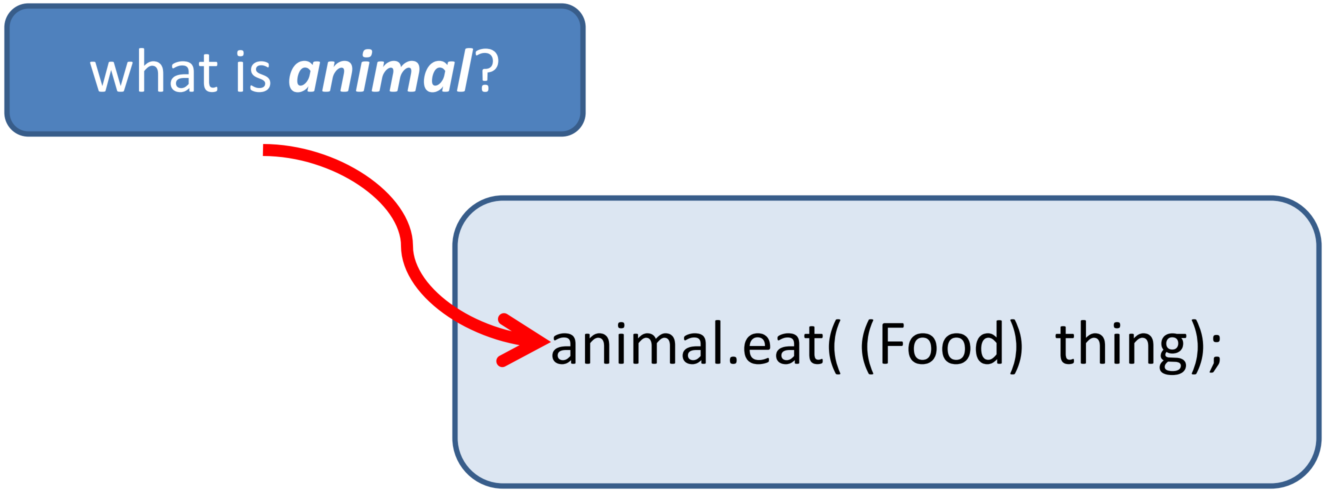
testing

(without actually ~~running~~ the program)

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

Understanding Program Behavior

(without actually ~~running~~ testing the program)



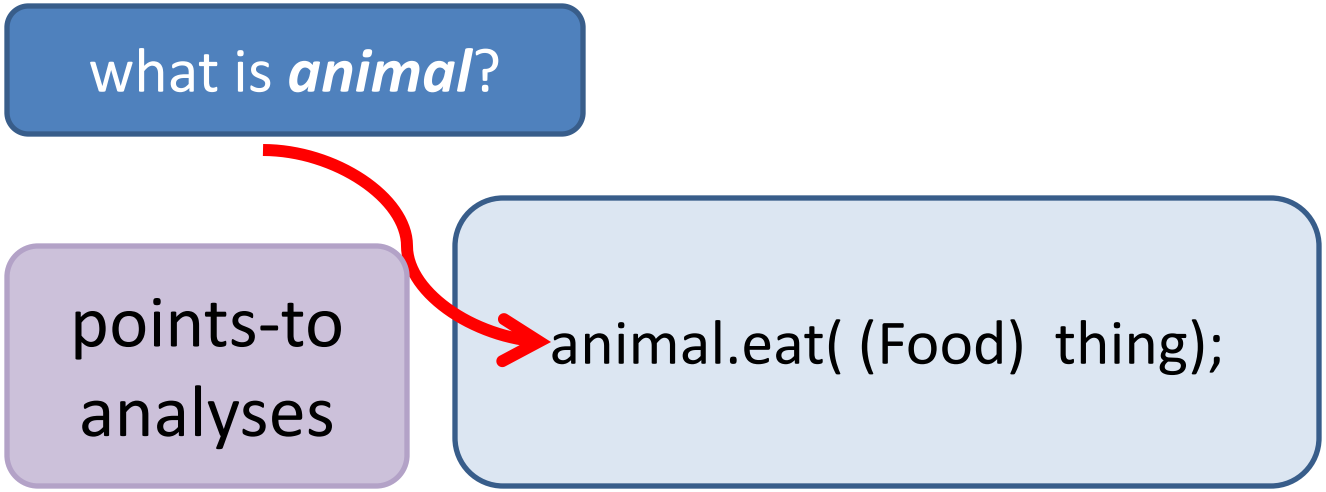
what is ***animal***?

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

Understanding Program Behavior

testing

(without actually ~~running~~ the program)



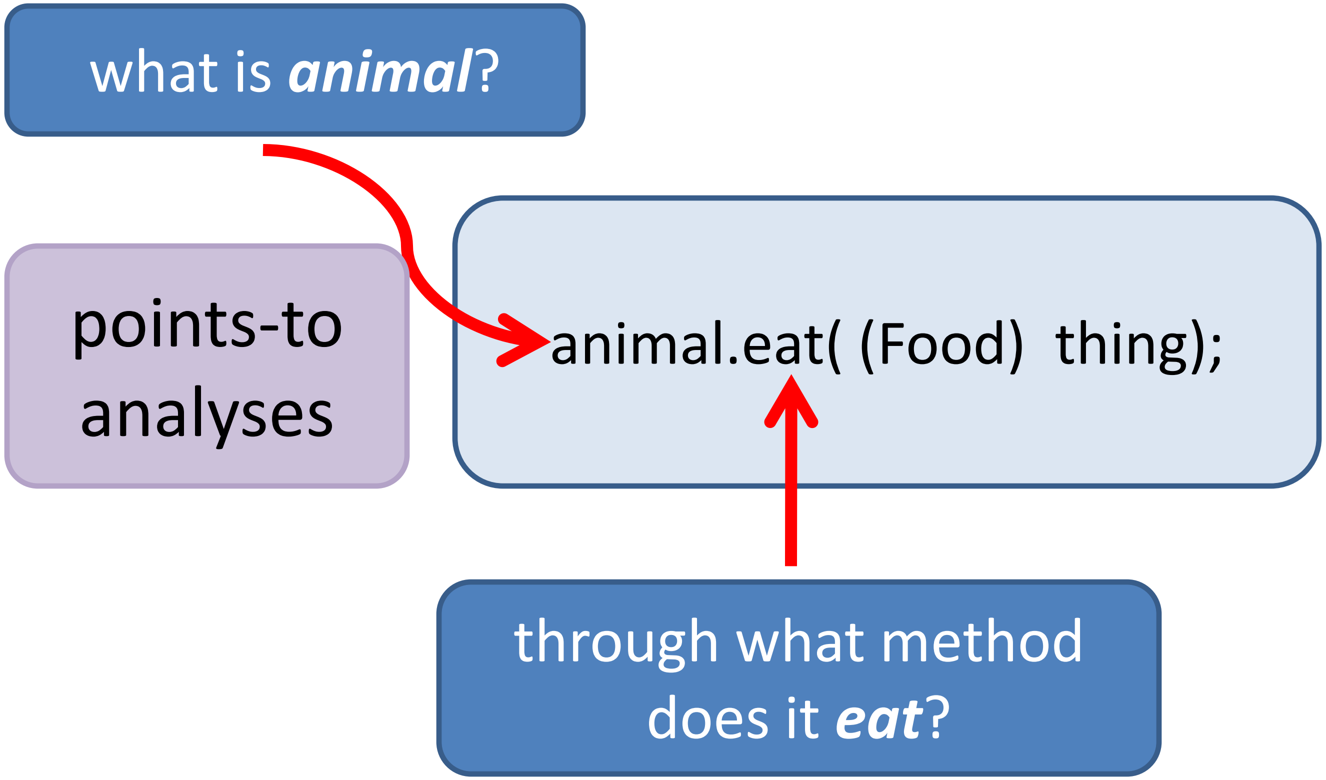
what is ***animal***?

points-to analyses

animal.eat((Food) thing);

Understanding Program Behavior

(without actually ~~running~~ testing the program)



what is ***animal***?

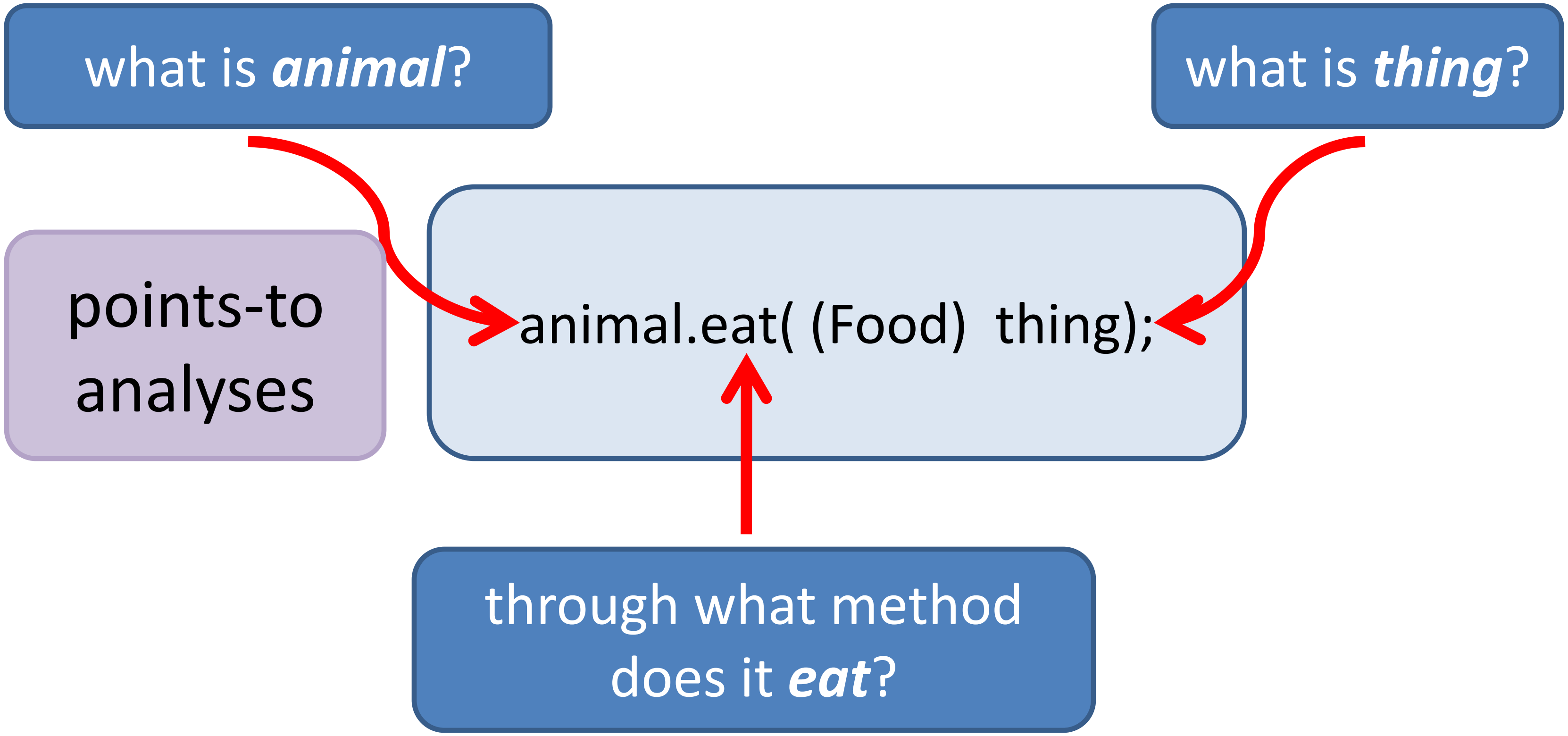
points-to analyses

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

through what method does it ***eat***?

Understanding Program Behavior

testing
(without actually ~~running~~ the program)



what is ***animal***?

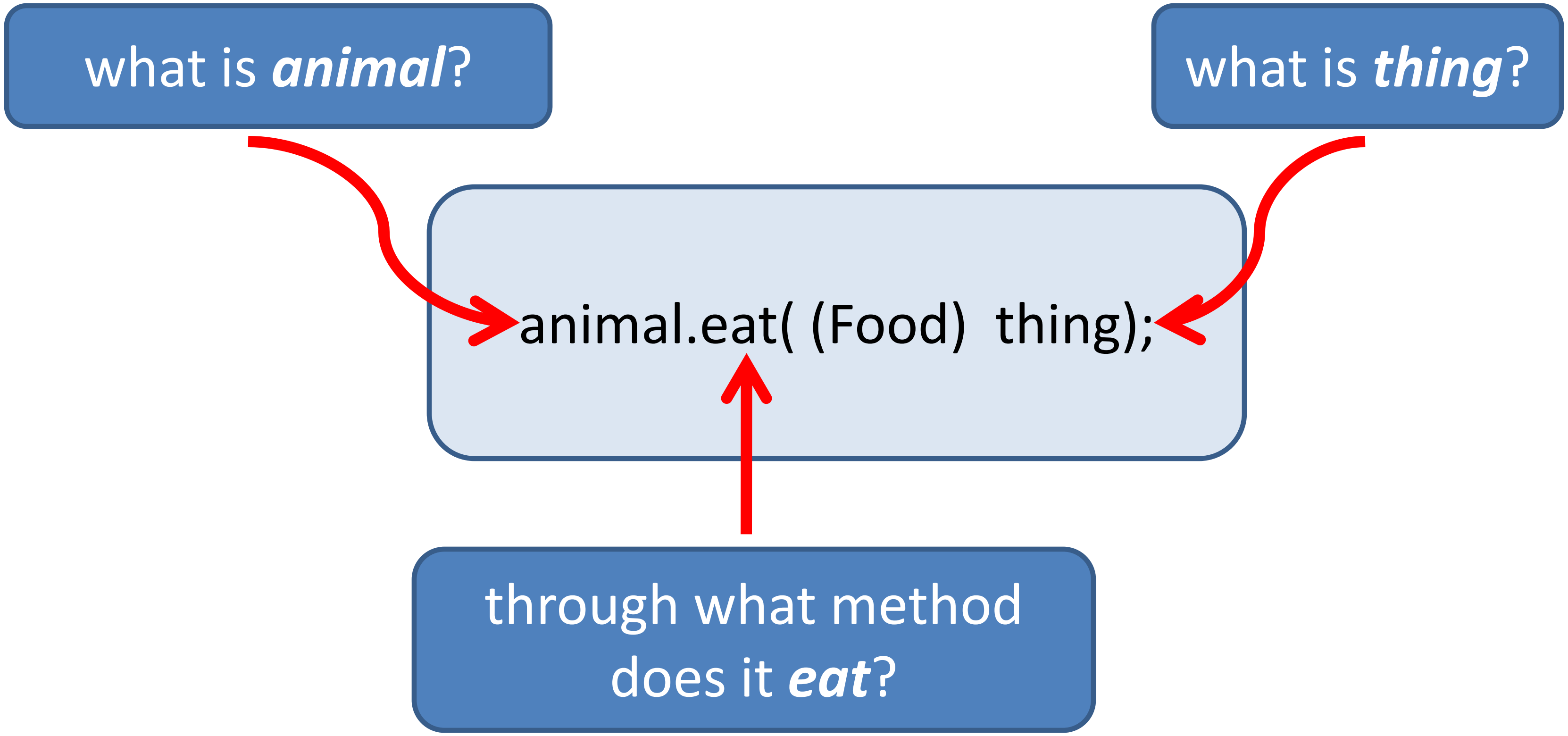
what is ***thing***?

points-to analyses

animal.eat((Food) thing);

through what method does it ***eat***?

Optimizations



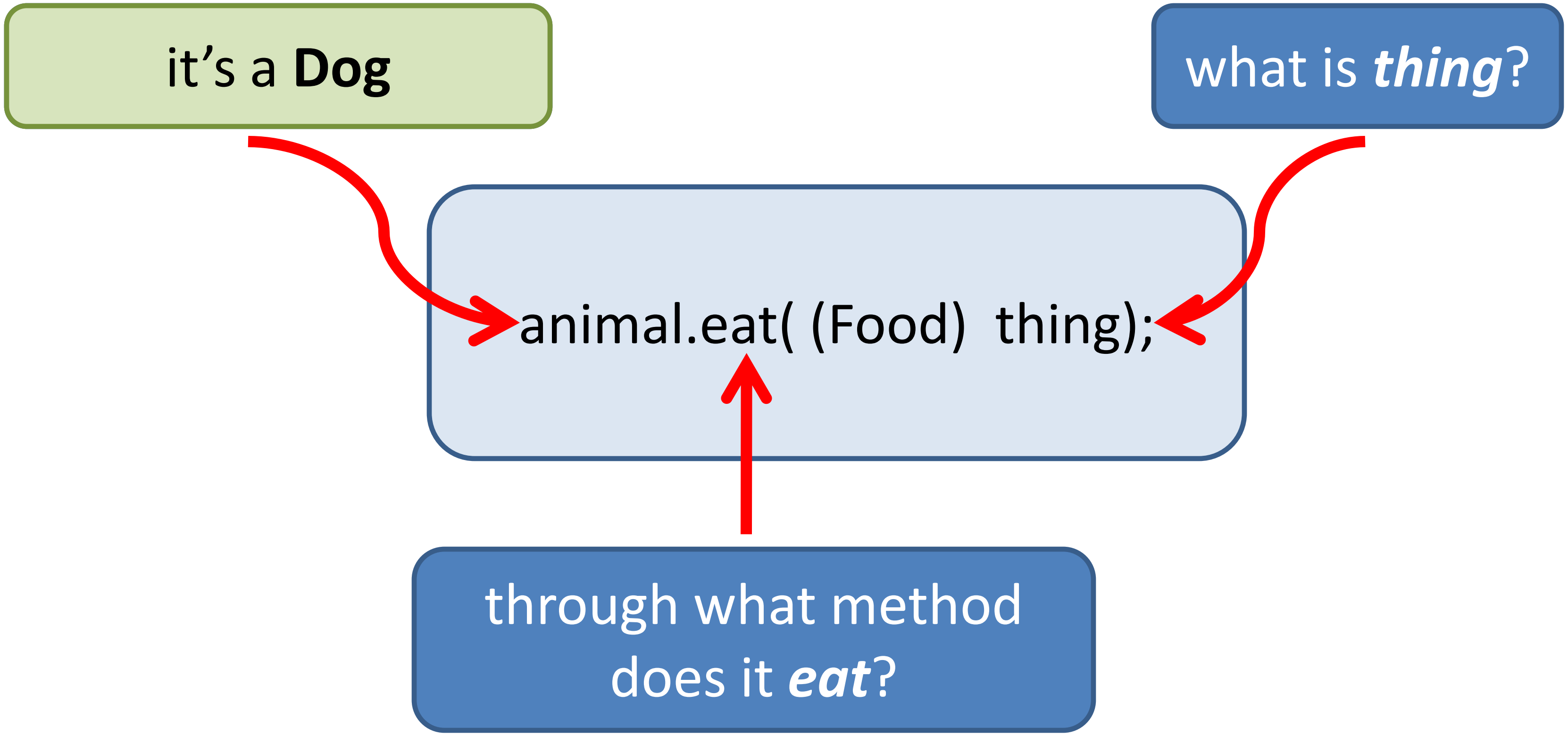
what is ***animal***?

what is ***thing***?

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

through what method does it ***eat***?

Optimizations



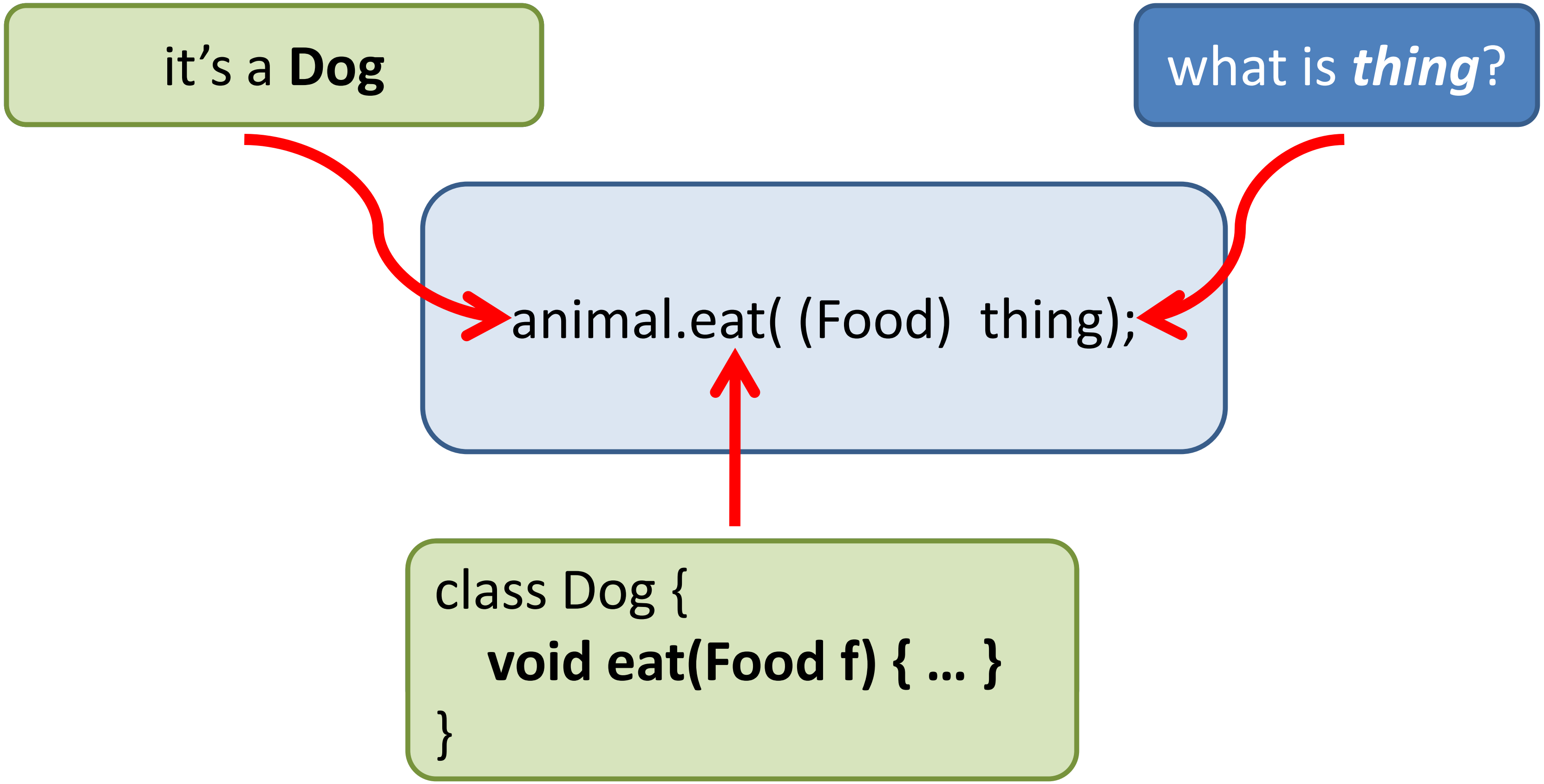
it's a **Dog**

what is ***thing***?

animal.eat((Food) thing);

through what method does it ***eat***?

Optimizations



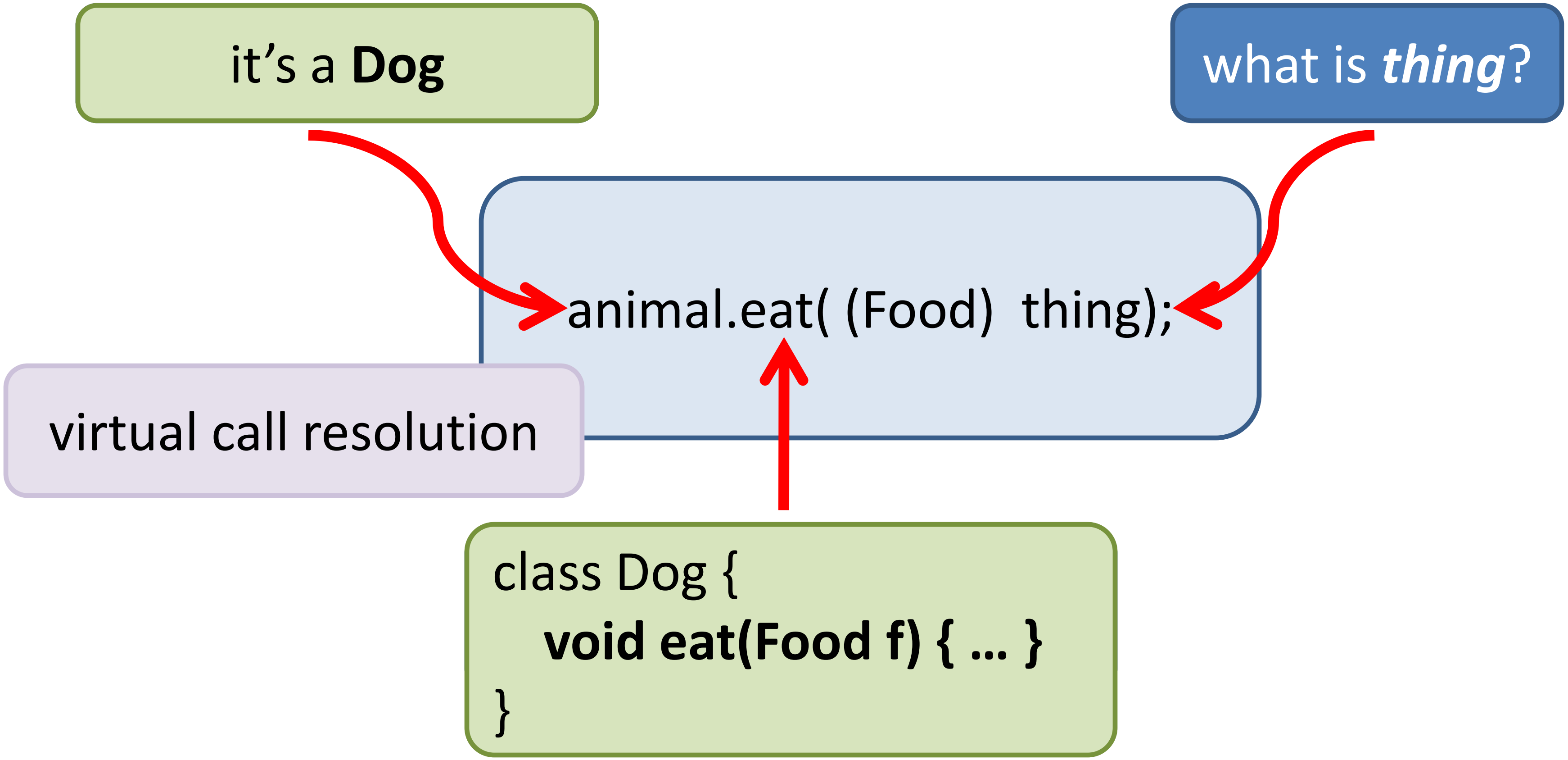
it’s a **Dog**

what is ***thing***?

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

```
class Dog {
   void eat(Food f) { … }
}
```

Optimizations



it’s a **Dog**

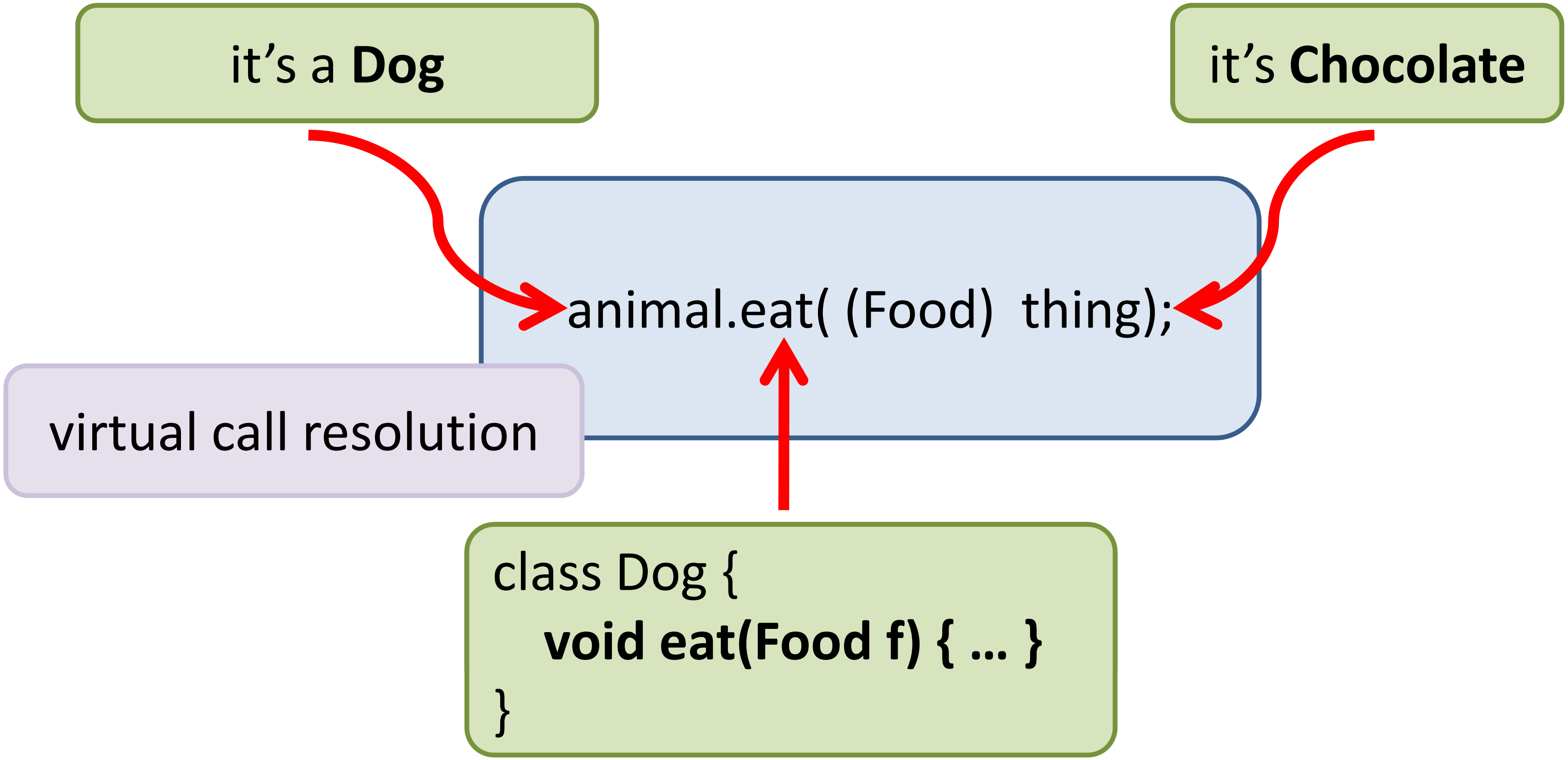
what is ***thing***?

animal.eat((Food) thing);

virtual call resolution

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

Optimizations



it’s a **Dog**

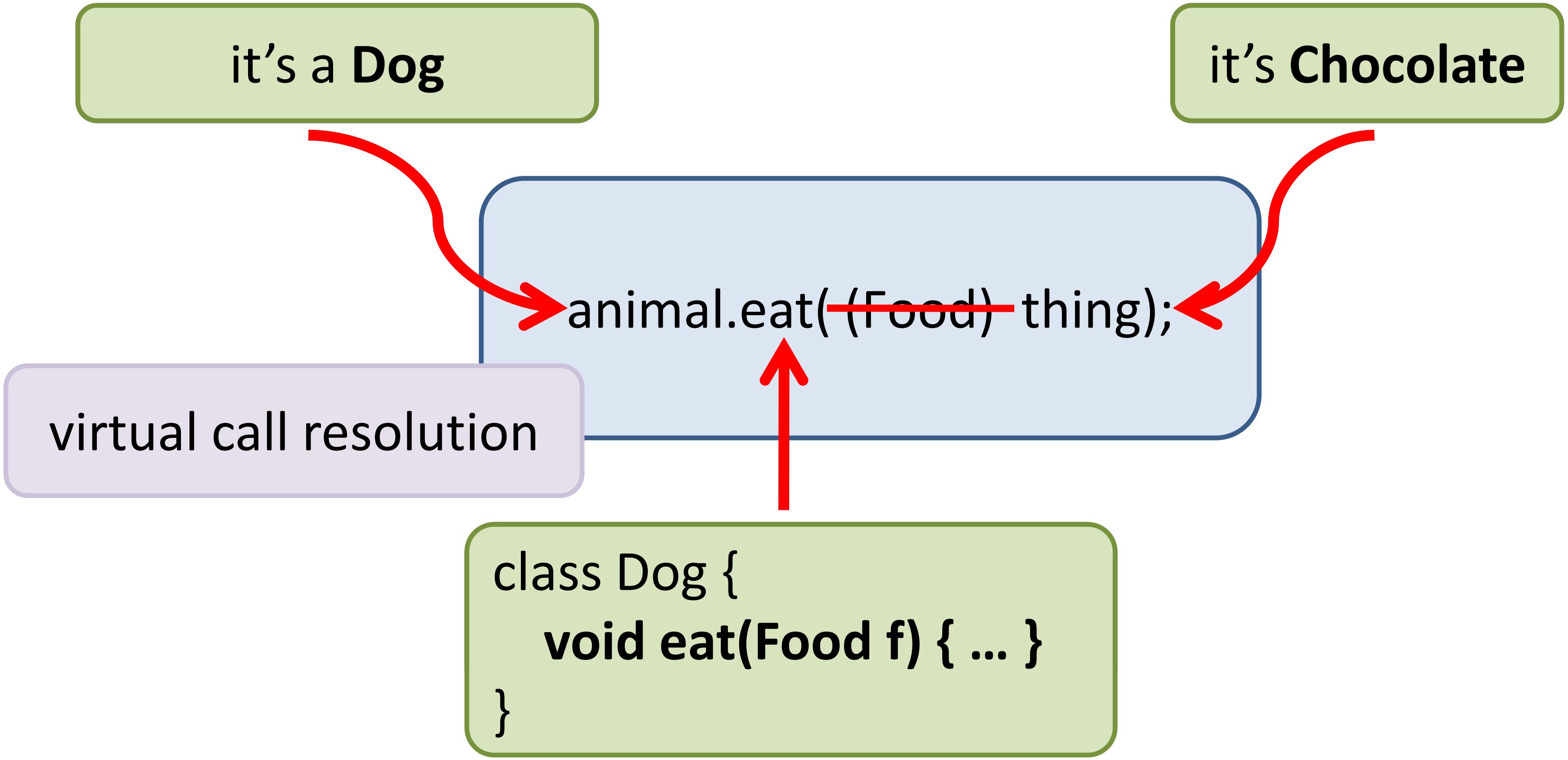
it’s **Chocolate**

animal.eat((Food) thing);

virtual call resolution

```
class Dog {
   void eat(Food f) { … }
}
```

Optimizations



it’s a **Dog**

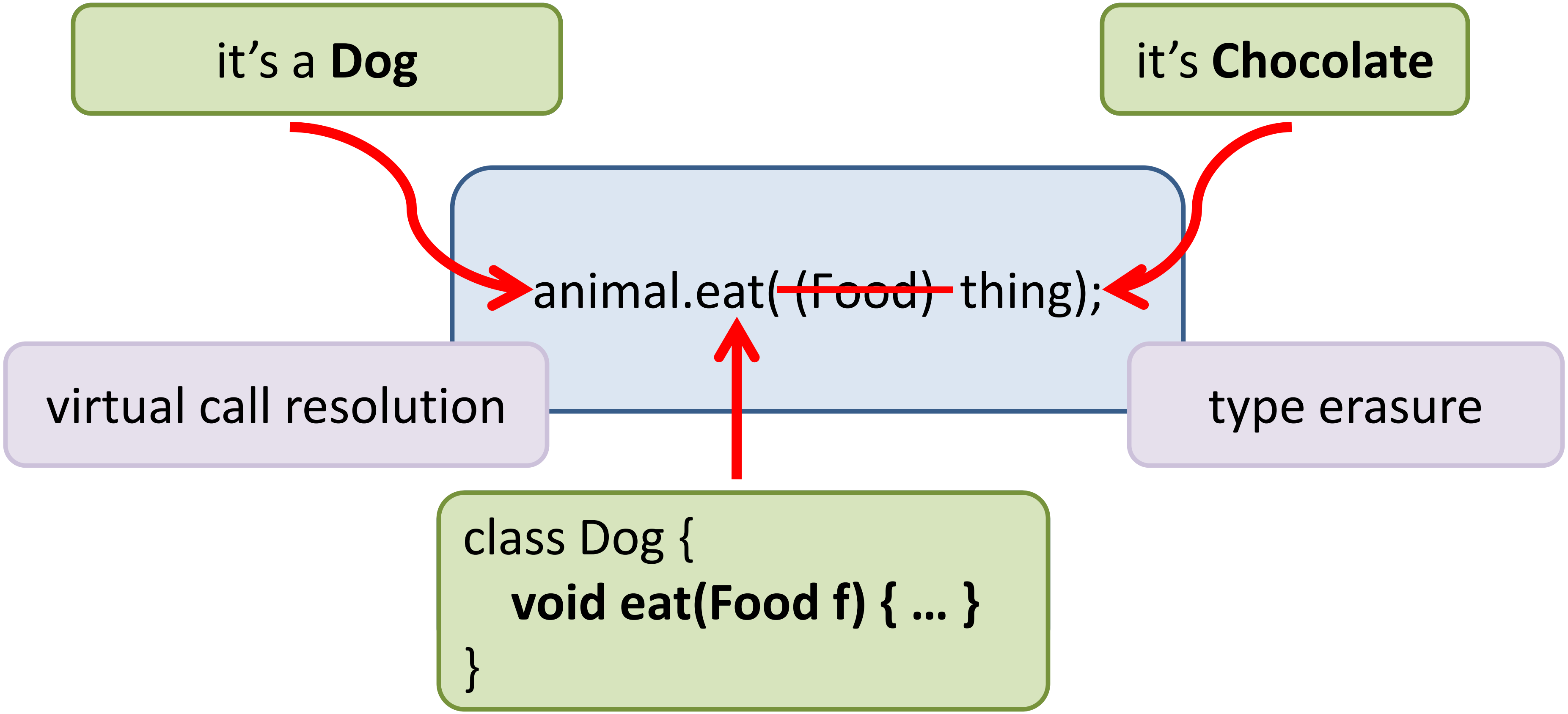
it’s **Chocolate**

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

virtual call resolution

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

Optimizations



it’s a **Dog**

it’s **Chocolate**

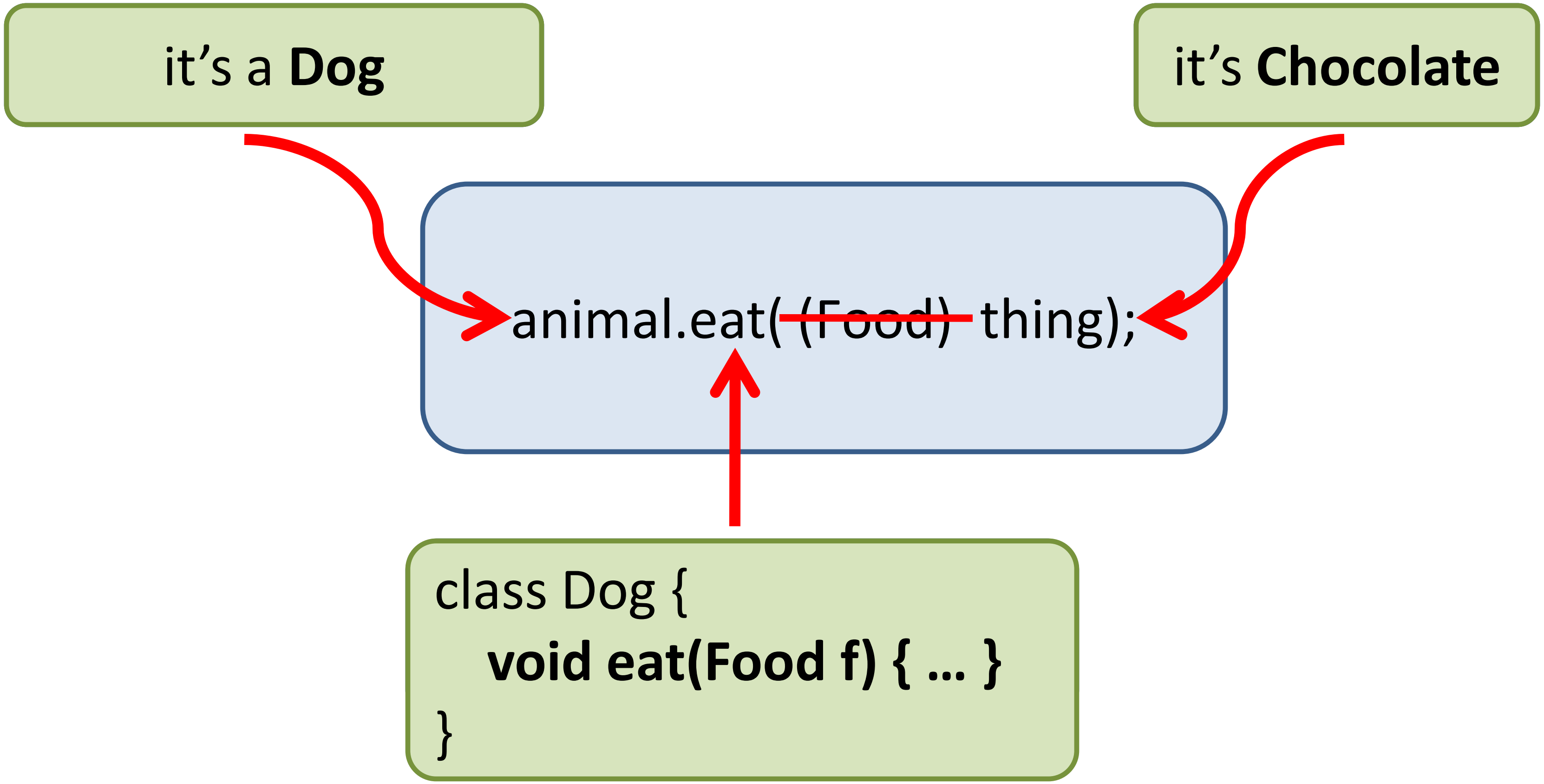
animal.eat(~~(Food)~~ thing);

virtual call resolution

type erasure

```
class Dog {
    void eat(Food f) { … }
}
```

Bug Finding



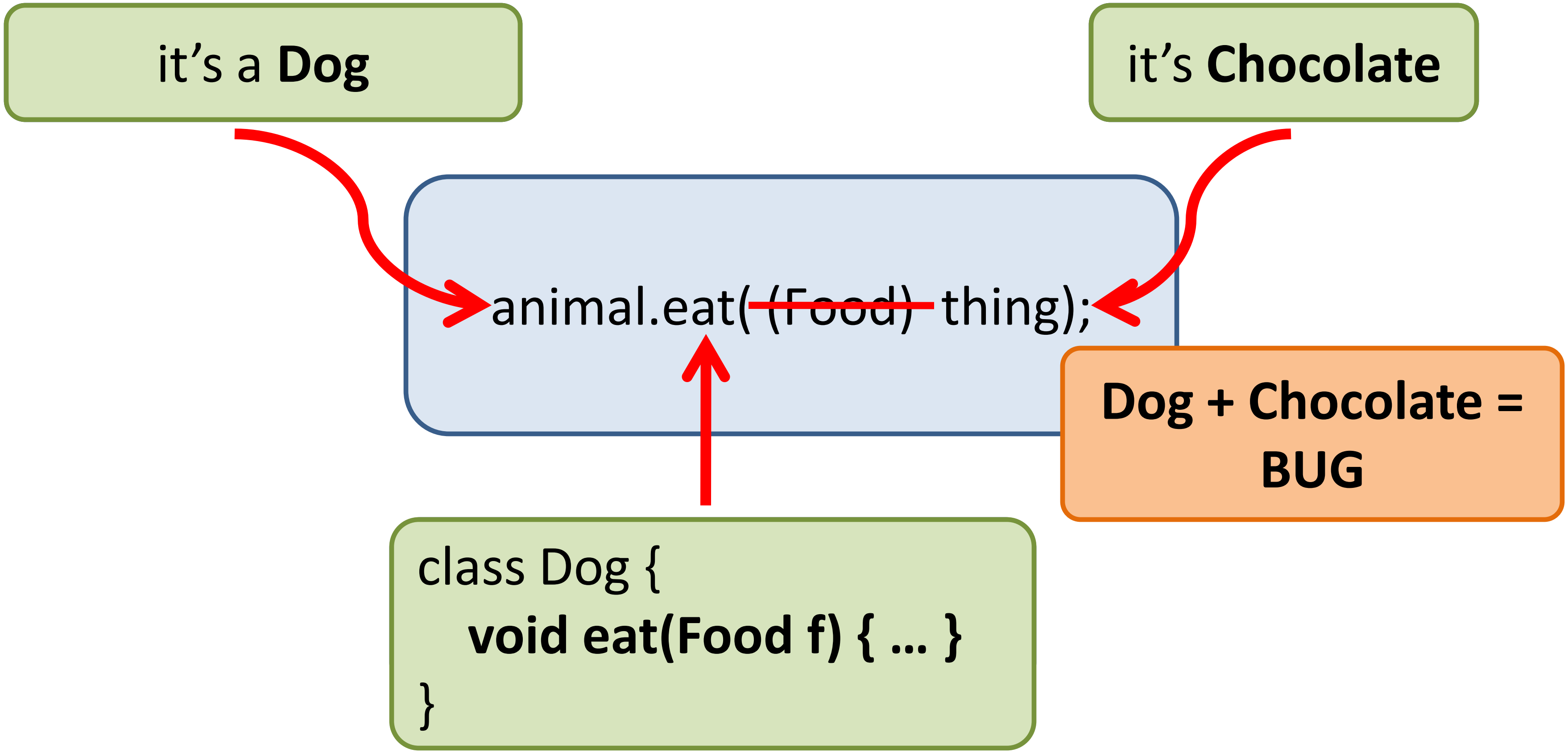
it’s a **Dog**

it’s **Chocolate**

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

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

Bug Finding



it's a **Dog**

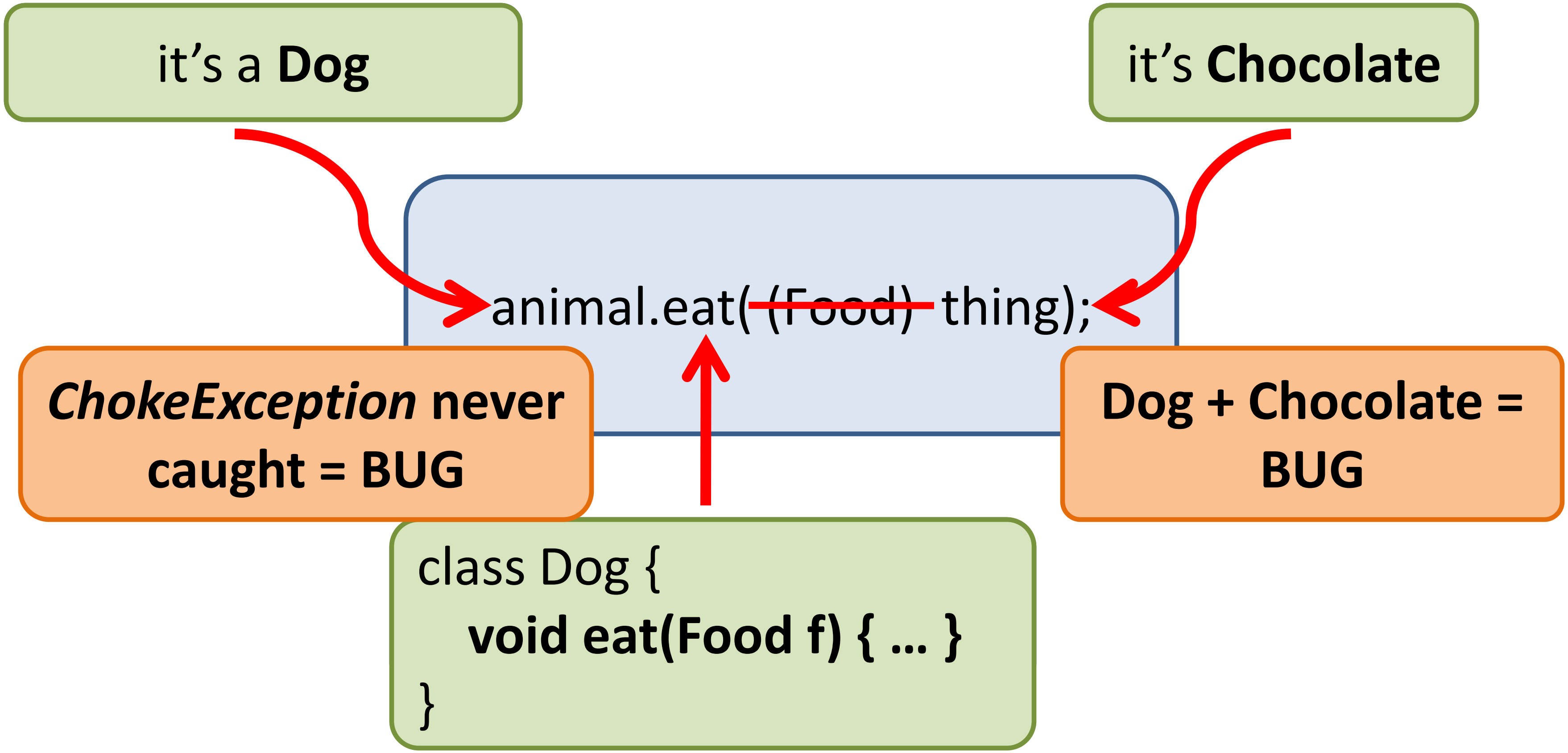
it's **Chocolate**

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

Dog + Chocolate = BUG

```
class Dog {
   void eat(Food f) { … }
}
```

Bug Finding



it's a **Dog**

it's **Chocolate**

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

***ChokeException* never caught = BUG**

Dog + Chocolate = BUG

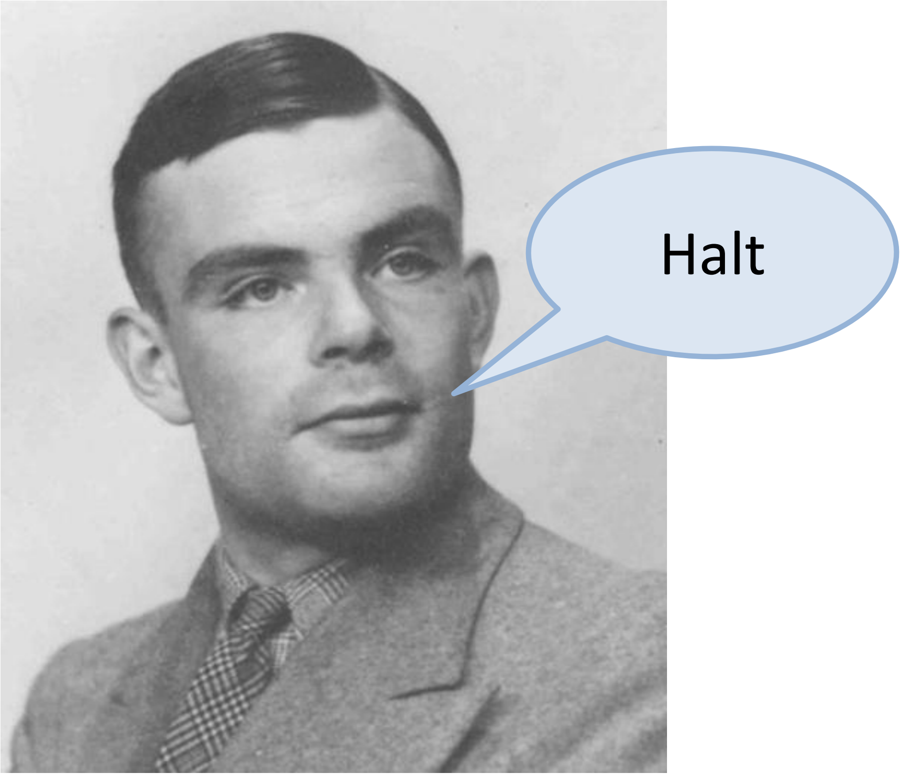
```
class Dog {
   void eat(Food f) { … }
}
```

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*

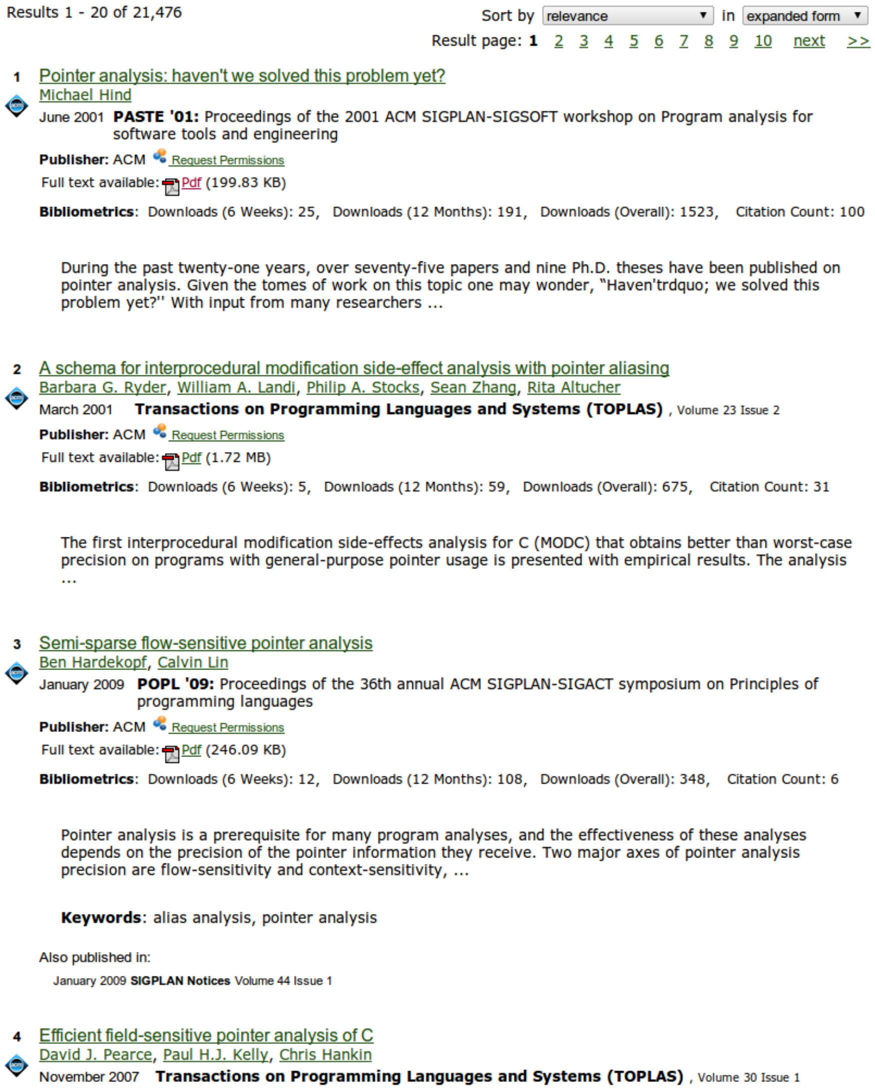
WHY PROGRAM ANALYSIS IN DATALOG?

WHY PROGRAM ANALYSIS IN A DECLARATIVE LANGUAGE?

WHY PROGRAM ANALYSIS IN A DECLARATIVE LANGUAGE?

WHY DATALOG?

Program Analysis: A Complex Domain



Results 1 - 20 of 21,476

Sort by relevance in expanded form

Result page: **1** 2 3 4 5 6 7 8 9 10 next >>

1 Pointer analysis: haven't we solved this problem yet?
Michael Hind
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 (199.83 KB)
Bibliometrics: 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'trdquo; 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 Altucher
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 Hardekopf, Calvin Lin
January 2009 **POPL '09:** Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages
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

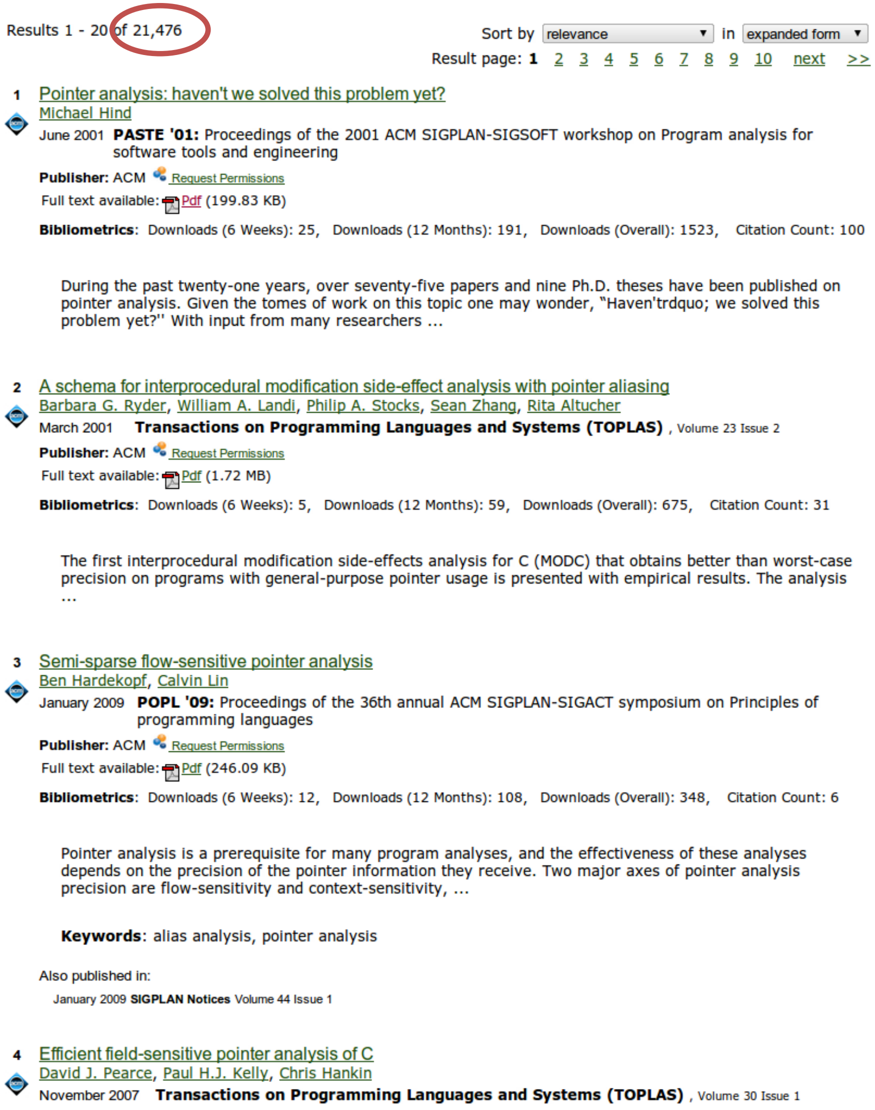
Pointer analysis is a prerequisite for many program analyses, and the effectiveness of these analyses depends on the precision of the pointer information they receive. Two major axes of pointer analysis precision are flow-sensitivity and context-sensitivity, ...

Keywords: alias analysis, pointer analysis

Also published in:
January 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



Results 1 - 20 of 21,476

Sort by relevance ▾ in expanded form ▾

Result page: **1** 2 3 4 5 6 7 8 9 10 next >>

1 Pointer analysis: haven't we solved this problem yet?
Michael Hind
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 (199.83 KB)
Bibliometrics: 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'trdquo; 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 Altucher
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 Hardekopf, Calvin Lin
January 2009 **POPL '09:** Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages
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, ...

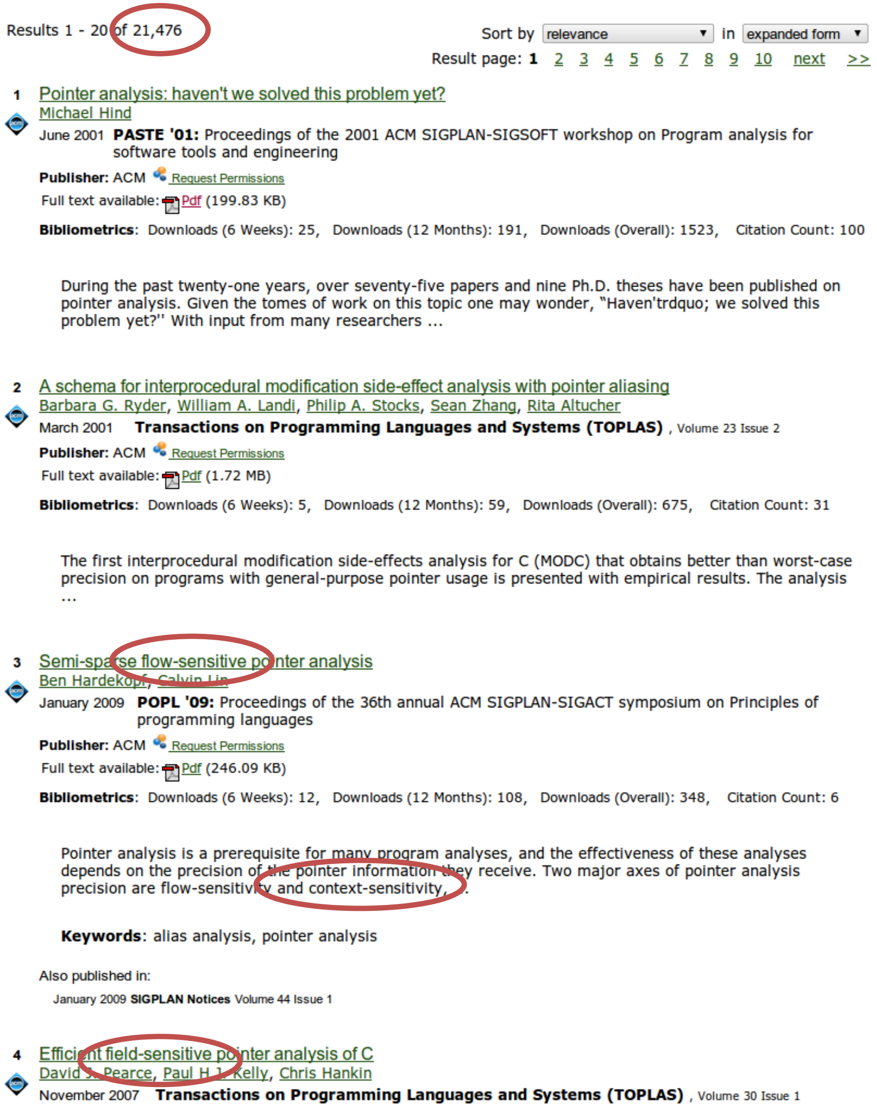
Keywords: alias analysis, pointer analysis

Also published in:
January 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

- flow-sensitive
- inclusion-based
- unification-based
- k-cfa
- object-sensitive
- context-sensitive
- field-based
- field-sensitive
- BDDs
- heap-sensitive



Results 1 - 20 of 21,476

Sort by relevance ▾ in expanded form ▾

Result page: **1** 2 3 4 5 6 7 8 9 10 next >>

1 Pointer analysis: haven't we solved this problem yet?
Michael Hind
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 (199.83 KB)
Bibliometrics: 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'trdquo; 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 Altucher
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 Hardekopf, Calvin Lin
January 2009 **POPL '09:** Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages
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

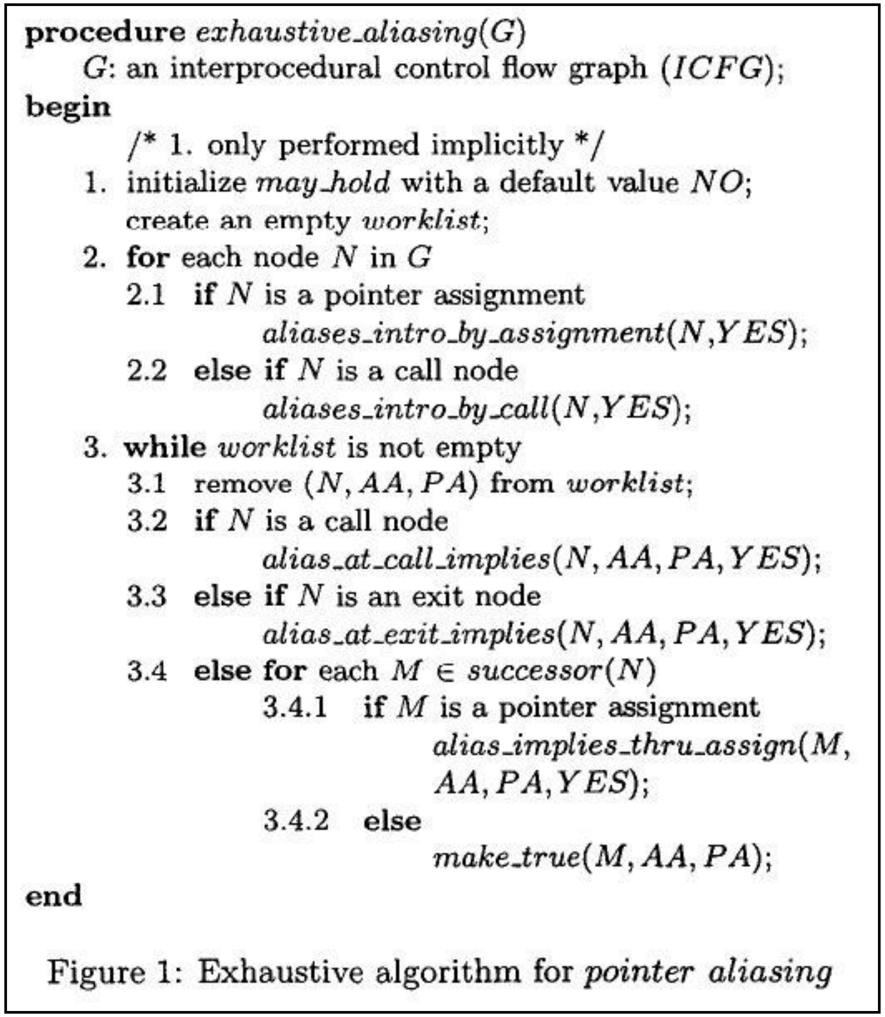
Pointer analysis is a prerequisite for many program analyses, and the effectiveness of these analyses depends on the precision of the pointer information they receive. Two major axes of pointer analysis precision are flow-sensitivity and context-sensitivity, ...

Keywords: alias analysis, pointer analysis

Also published in:
January 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

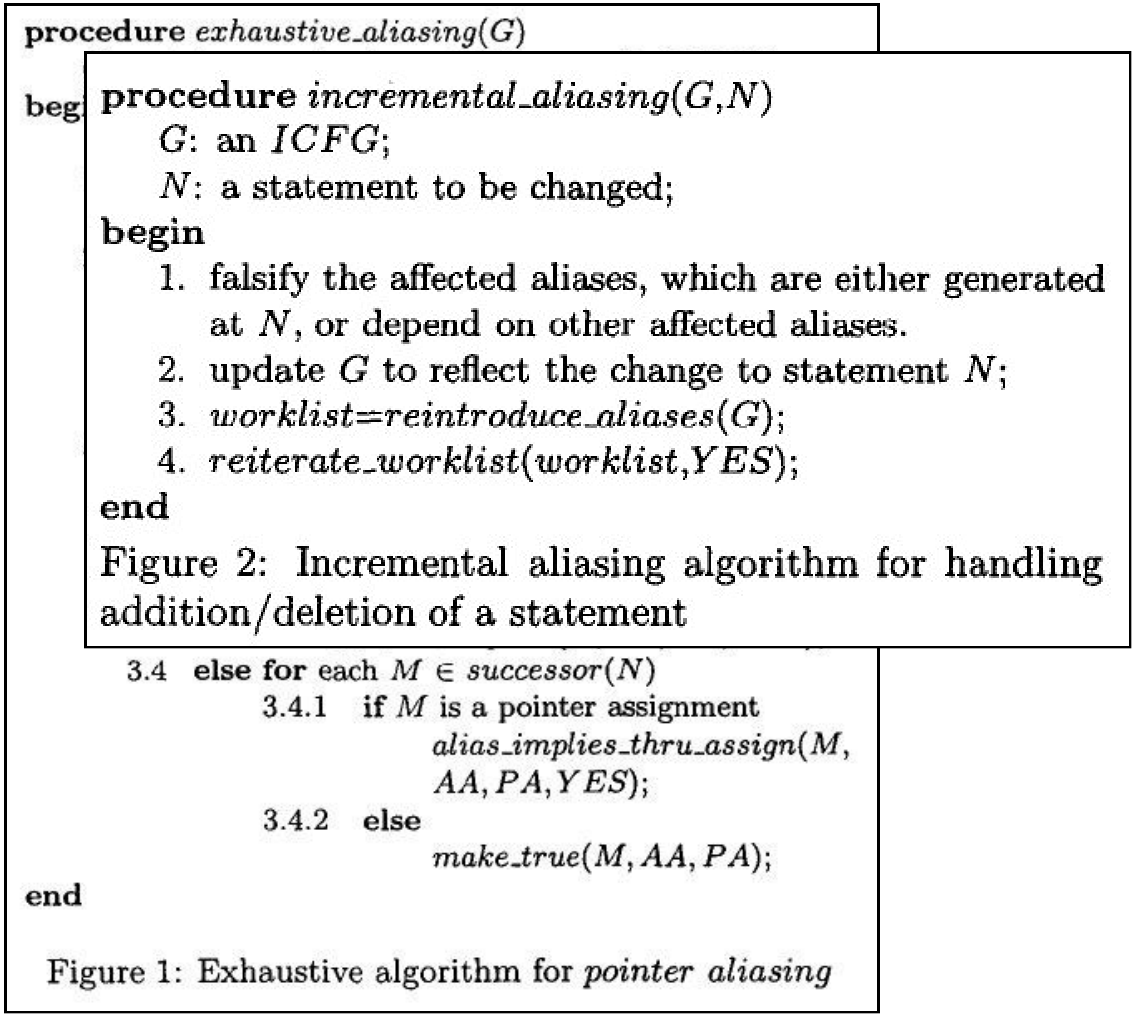
Algorithms in 10-page Conf. Papers



```
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



```
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

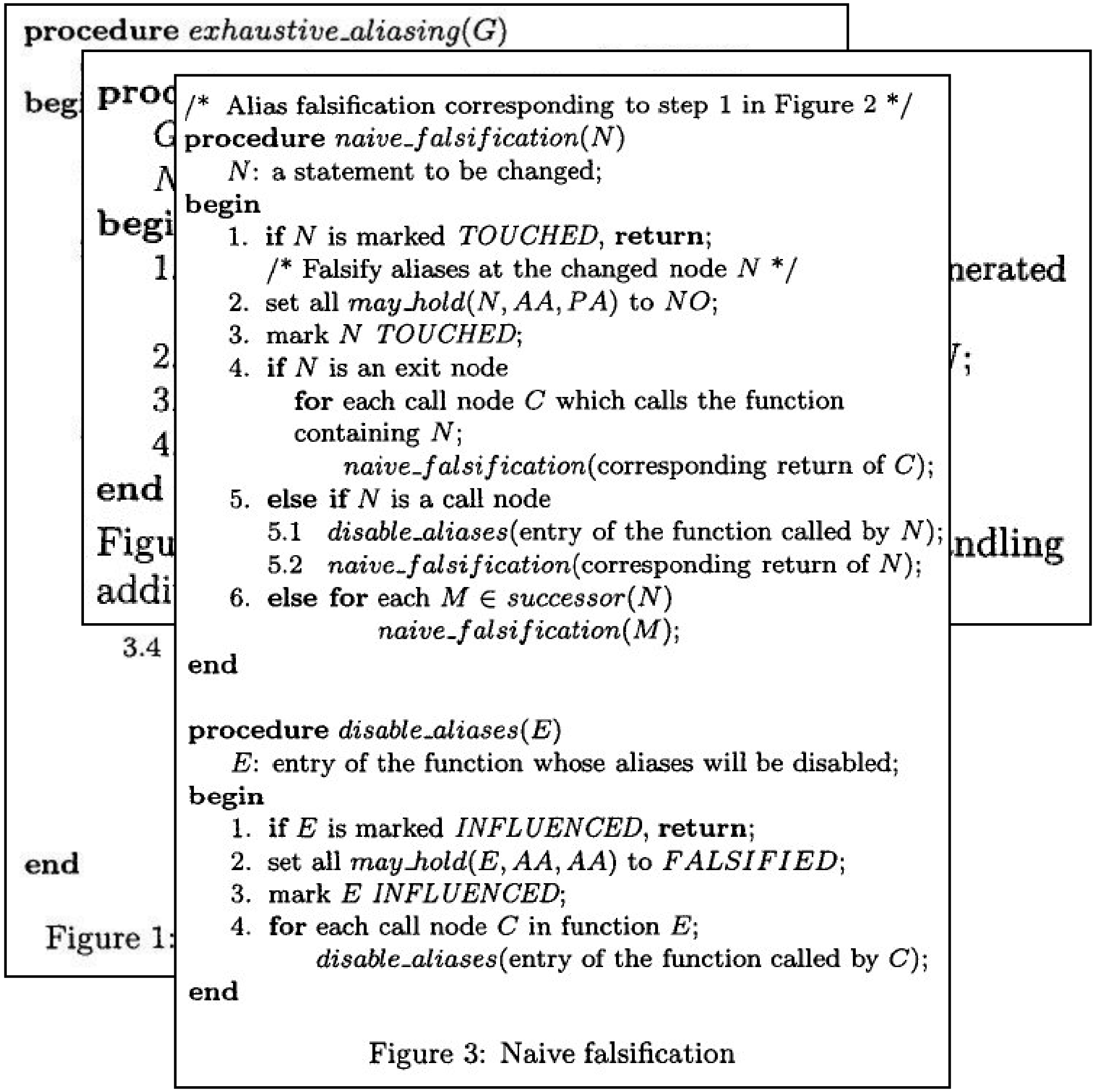
```
procedure exhaustive_aliasing(G)

beg

    3.4  else for each $M \in successor(N)$
             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



procedure *exhaustive_aliasing*(*G*)

beg

end

Figure 1:

proc

G

N

begi

1.

2.

3.

4.

end

Figu

addi

nerated

V;

ndling

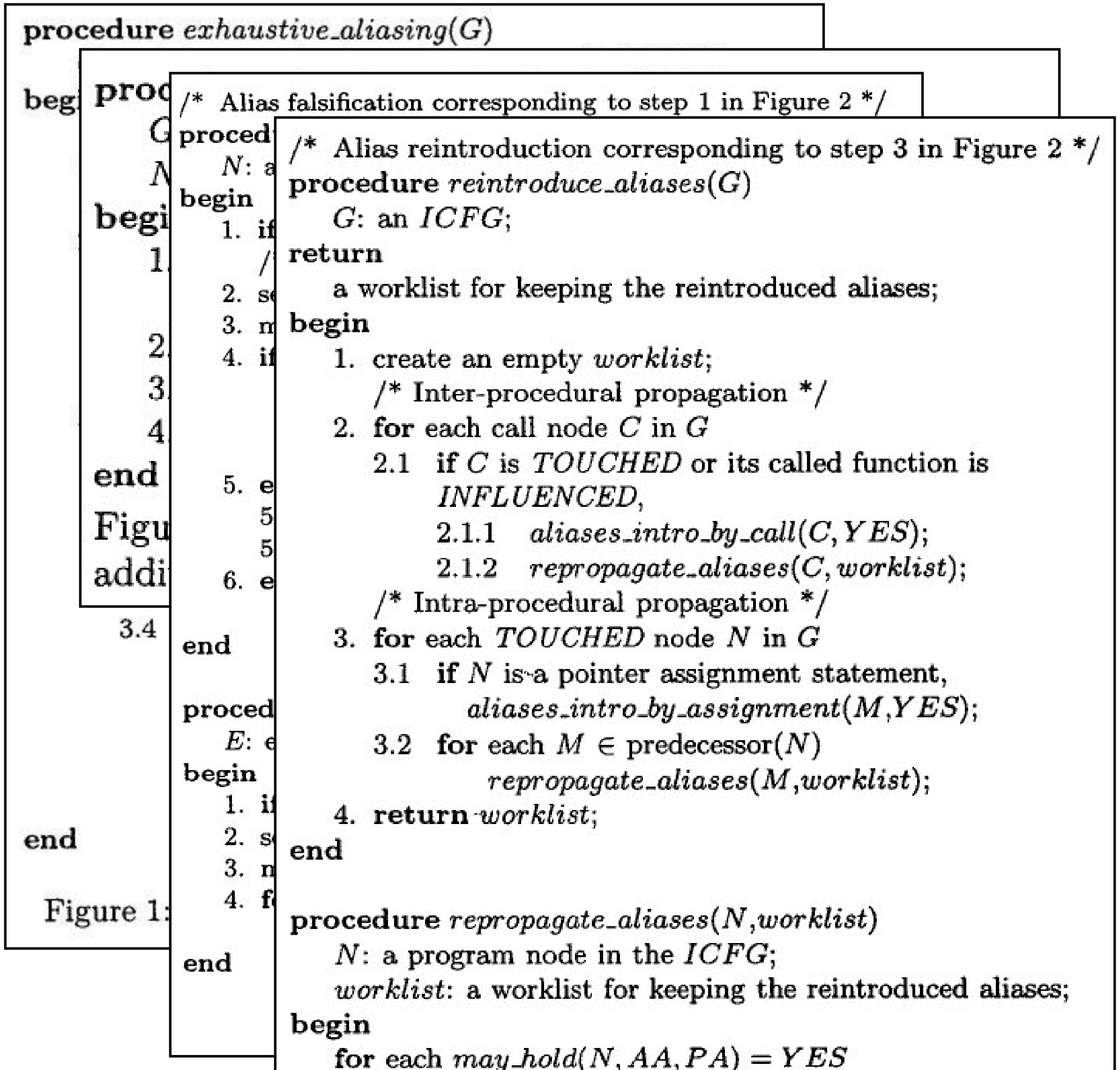
3.4

```
/* Alias falsification corresponding to step 1 in Figure 2 */
procedure naive_falsification($N$)
    $N$: a statement to be changed;
begin
    1. if $N$ is marked $TOUCHED$, return;
       /* Falsify aliases at the changed node $N$ */
    2. set all $may\_hold(N, AA, PA)$ to $NO$;
    3. mark $N$ $TOUCHED$;
    4. if $N$ is an exit node
          for each call node $C$ which calls the function
          containing $N$;
              naive_falsification(corresponding return of $C$);
    5. else if $N$ is a call node
       5.1  disable_aliases(entry of the function called by $N$);
       5.2  naive_falsification(corresponding return of $N$);
    6. else for each $M \in successor(N)$
                naive_falsification($M$);
end

procedure disable_aliases($E$)
    $E$: entry of the function whose aliases will be disabled;
begin
    1. if $E$ is marked $INFLUENCED$, return;
    2. set all $may\_hold(E, AA, AA)$ to $FALSIFIED$;
    3. mark $E$ $INFLUENCED$;
    4. for each call node $C$ in function $E$;
           disable_aliases(entry of the function called by $C$);
end
```

Figure 3: Naive falsification

Algorithms in 10-page Conf. Papers



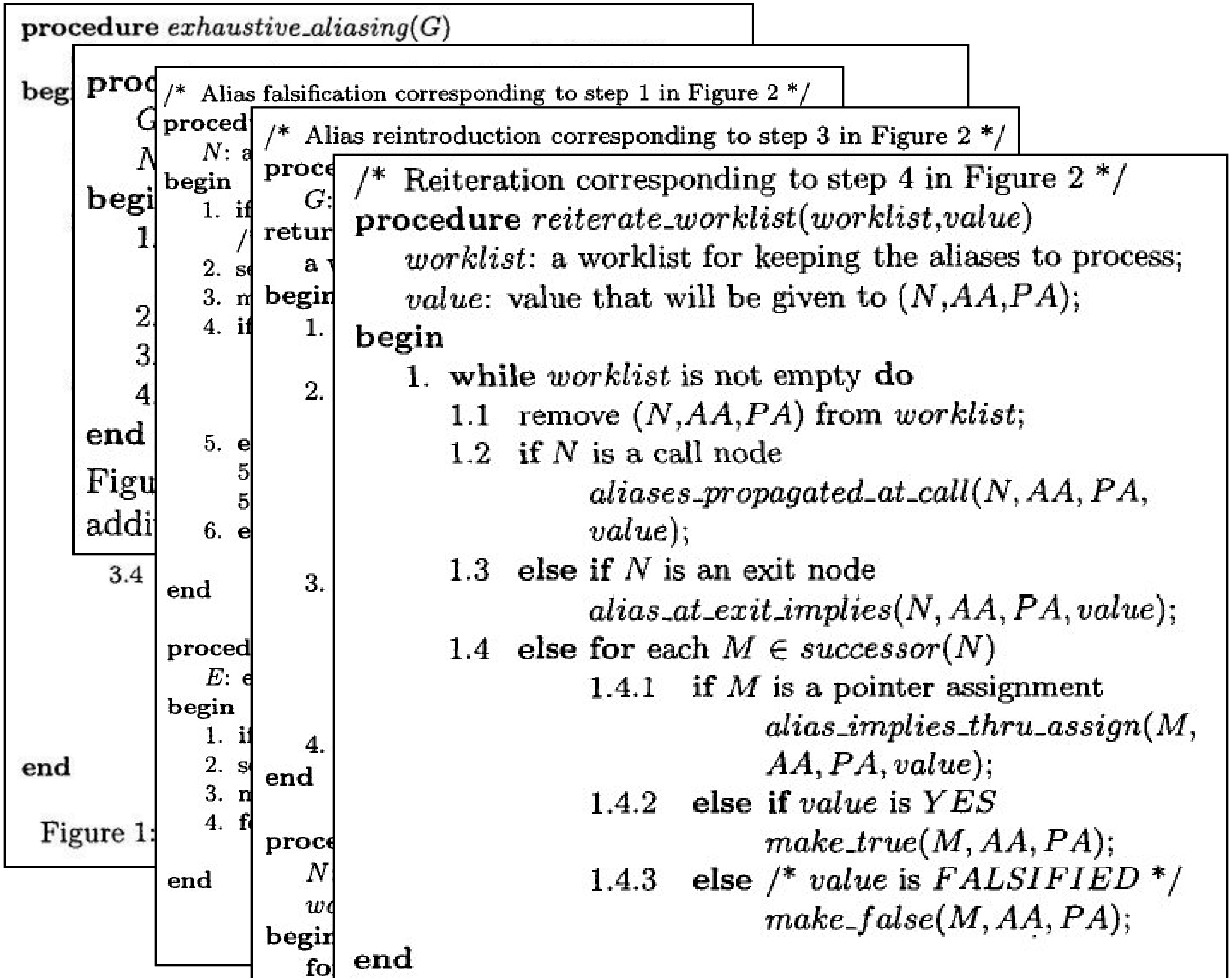
```
procedure exhaustive_aliasing(G)
```

```
/* Alias falsification corresponding to step 1 in Figure 2 */
```

```
/* Alias reintroduction corresponding to step 3 in Figure 2 */
procedure reintroduce_aliases(G)
    G: an ICFG;
return
    a worklist for keeping the reintroduced aliases;
begin
    1. create an empty worklist;
       /* Inter-procedural propagation */
    2. for each call node C in G
       2.1  if C is TOUCHED or its called function is
            INFLUENCED,
            2.1.1   aliases_intro_by_call(C,YES);
            2.1.2   repropagate_aliases(C, worklist);
       /* Intra-procedural propagation */
    3. for each TOUCHED node N in G
       3.1  if N is a pointer assignment statement,
               aliases_intro_by_assignment(M,YES);
       3.2  for each M ∈ predecessor(N)
                repropagate_aliases(M,worklist);
    4. return worklist;
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
```

Algorithms in 10-page Conf. Papers



```
procedure exhaustive_aliasing(G)
beg
end
Figure 1:
```

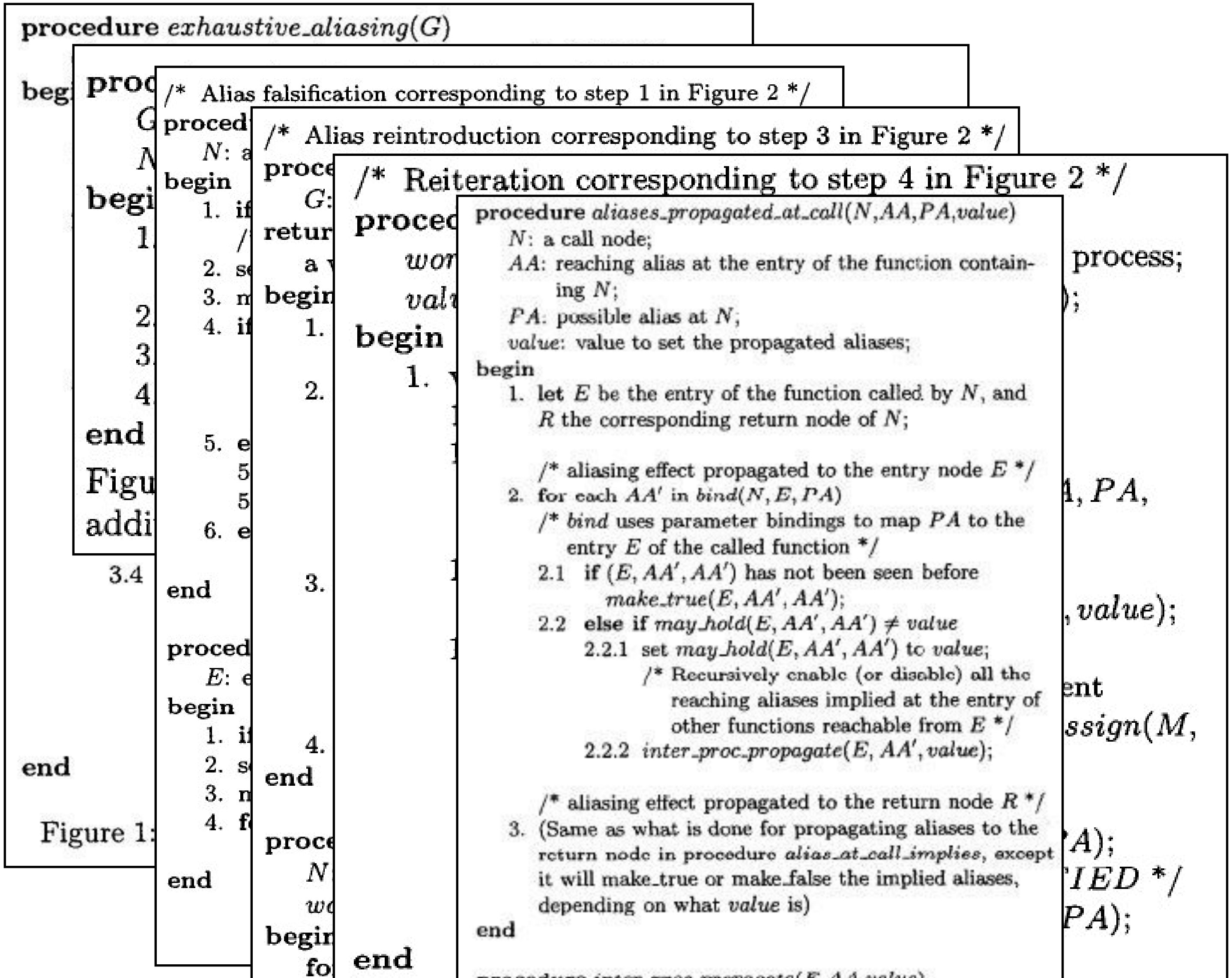
```
proc
    G
    N
begi
    1.
    2.
    3.
    4.
end
Figu
addi
3.4
```

```
/* Alias falsification corresponding to step 1 in Figure 2 */
proced
    N: a
begin
    1. if
       /
    2. s
    3. m
    4. if
    5. e
       5
       5
    6. e
end
proced
    E: e
begin
    1. i
    2. s
    3. n
    4. f
end
```

```
/* Alias reintroduction corresponding to step 3 in Figure 2 */
proce
    G:
retur
    a
begin
    1.
    2.
    3.
    4.
end
proce
    N
    w
begin
    fo
```

```
/* Reiteration corresponding to step 4 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 $M \in successor(N)$
                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);
end
```

Algorithms in 10-page Conf. Papers



procedure *exhaustive_aliasing*(*G*)

/* Alias falsification corresponding to step 1 in Figure 2 */

/* Alias reintroduction corresponding to step 3 in Figure 2 */

/* Reiteration corresponding to step 4 in Figure 2 */

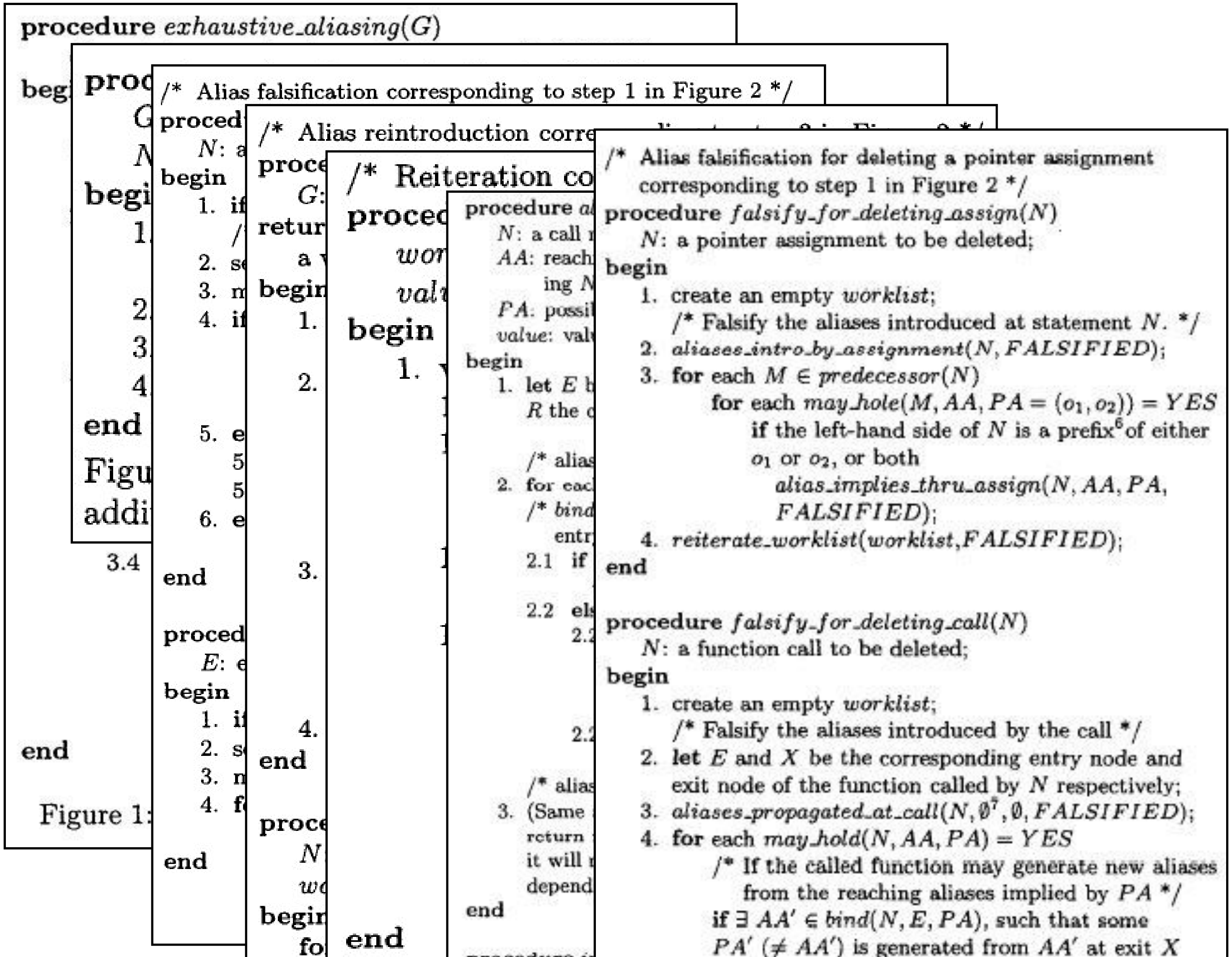
```
procedure aliases_propagated_at_call(N,AA,PA,value)
    N: a call node;
    AA: reaching alias at the entry of the function containing N;
    PA: possible alias at N;
    value: value to set the propagated aliases;
begin
    1. let E be the entry of the function called by N, and
       R the corresponding return node of N;

       /* aliasing effect propagated to the entry node E */
    2. for each AA' in bind(N,E,PA)
       /* bind uses parameter bindings to map PA to the
          entry E of the called function */
       2.1 if (E,AA',AA') has not been seen before
              make_true(E,AA',AA');
       2.2 else if may_hold(E,AA',AA') ≠ value
           2.2.1 set may_hold(E,AA',AA') to value;
                 /* Recursively enable (or disable) all the
                    reaching aliases implied at the entry of
                    other functions reachable from E */
           2.2.2 inter_proc_propagate(E,AA',value);

       /* aliasing effect propagated to the return node R */
    3. (Same as what is done for propagating aliases to the
       return node in procedure alias_at_call_implies, except
       it will make_true or make_false the implied aliases,
       depending on what value is)
end

procedure inter_proc_propagate(E,AA,value)
```

Algorithms in 10-page Conf. Papers



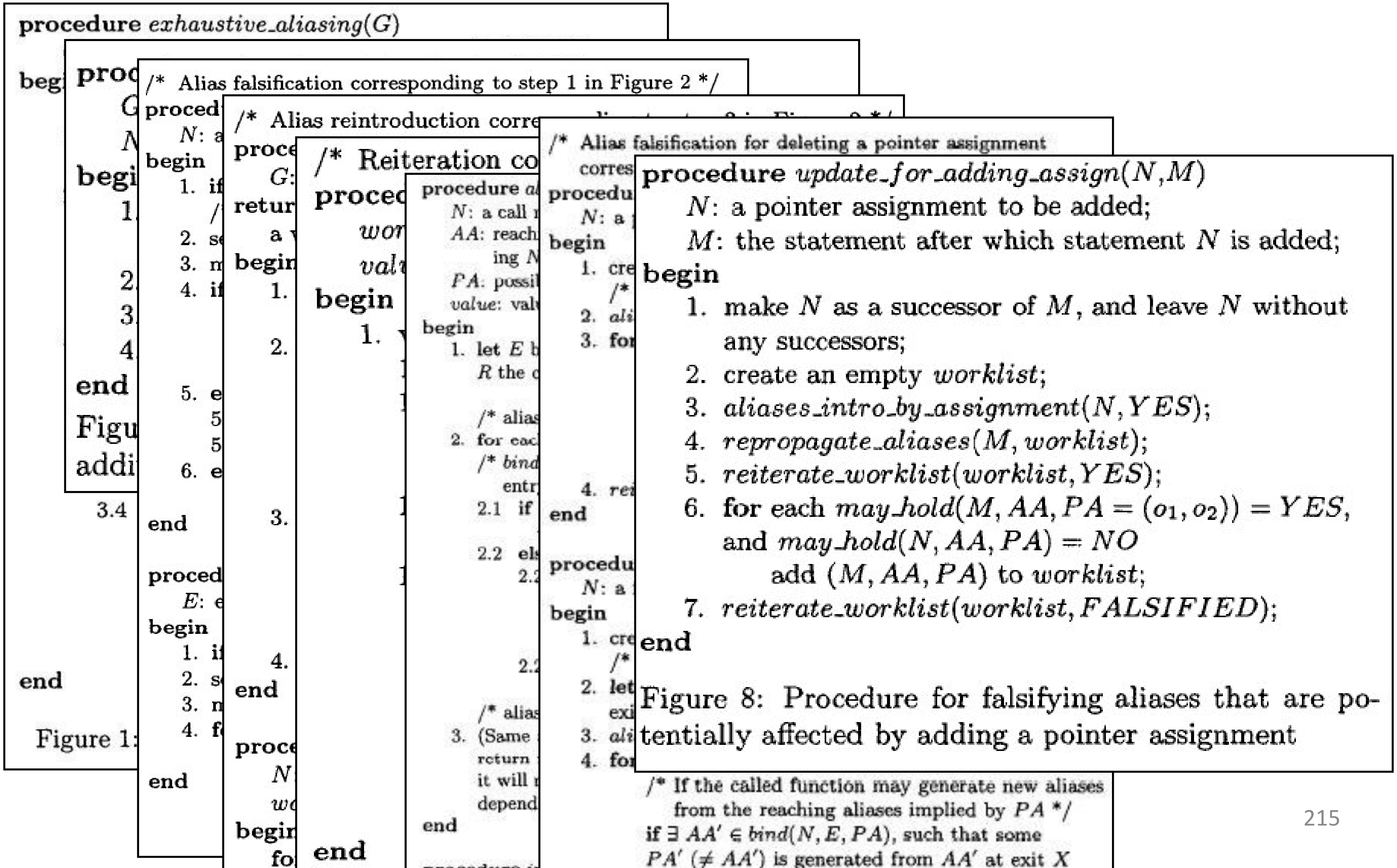
```
procedure exhaustive_aliasing(G)
```

```
/* Alias falsification corresponding to step 1 in Figure 2 */
```

```
/* Alias falsification for deleting a pointer assignment
   corresponding to step 1 in Figure 2 */
procedure falsify_for_deleting_assign(N)
    N: a pointer assignment to be deleted;
begin
    1. create an empty worklist;
       /* Falsify the aliases introduced at statement N. */
    2. aliases_intro_by_assignment(N, FALSIFIED);
    3. for each M ∈ predecessor(N)
           for each may_hole(M, AA, PA = (o1, o2)) = YES
               if the left-hand side of N is a prefix[6] of either
               o1 or o2, or both
                   alias_implies_thru_assign(N, AA, PA,
                   FALSIFIED);
    4. reiterate_worklist(worklist, FALSIFIED);
end

procedure falsify_for_deleting_call(N)
    N: a function call to be deleted;
begin
    1. create an empty worklist;
       /* Falsify the aliases introduced by the call */
    2. let E and X be the corresponding entry node and
       exit node of the function called by N respectively;
    3. aliases_propagated_at_call(N, ∅[7], ∅, FALSIFIED);
    4. for each may_hold(N, AA, PA) = YES
           /* If the called function may generate new aliases
              from the reaching aliases implied by PA */
           if ∃ AA' ∈ bind(N, E, PA), such that some
           PA' (≠ AA') is generated from AA' at exit X
```

Algorithms in 10-page Conf. Papers



```
procedure exhaustive_aliasing(G)
```

```
/* Alias falsification corresponding to step 1 in Figure 2 */
```

```
/* Alias reintroduction corre
```

```
/* Reiteration co
```

```
/* Alias falsification for deleting a pointer assignment
   corres
```

```
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 = (o_1, o_2)) = YES,
       and may_hold(N, AA, PA) = NO
           add (M, AA, PA) to worklist;
    7. reiterate_worklist(worklist, FALSIFIED);
end
```

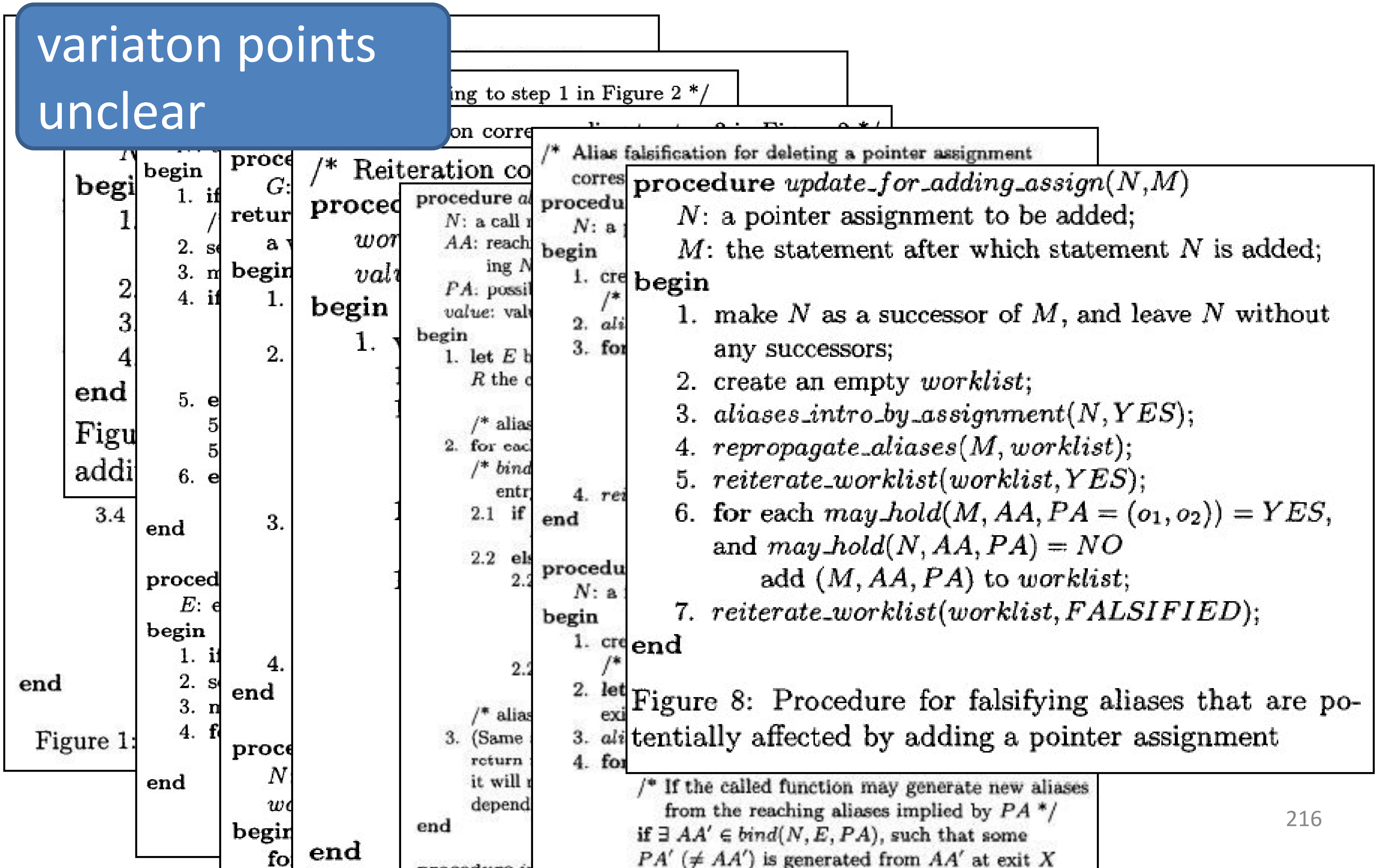
Figure 8: Procedure for falsifying aliases that are potentially affected by adding a pointer assignment

```
/* If the called function may generate new aliases
   from the reaching aliases implied by PA */
if ∃ AA' ∈ bind(N, E, PA), such that some
PA' (≠ AA') is generated from AA' at exit X
```

Figure 1:

Algorithms in 10-page Conf. Papers

variaton points unclear



/* Alias falsification for deleting a pointer assignment

```
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 = (o_1, o_2)) = YES,
       and may_hold(N, AA, PA) = NO
           add (M, AA, PA) to worklist;
    7. reiterate_worklist(worklist, FALSIFIED);
end
```

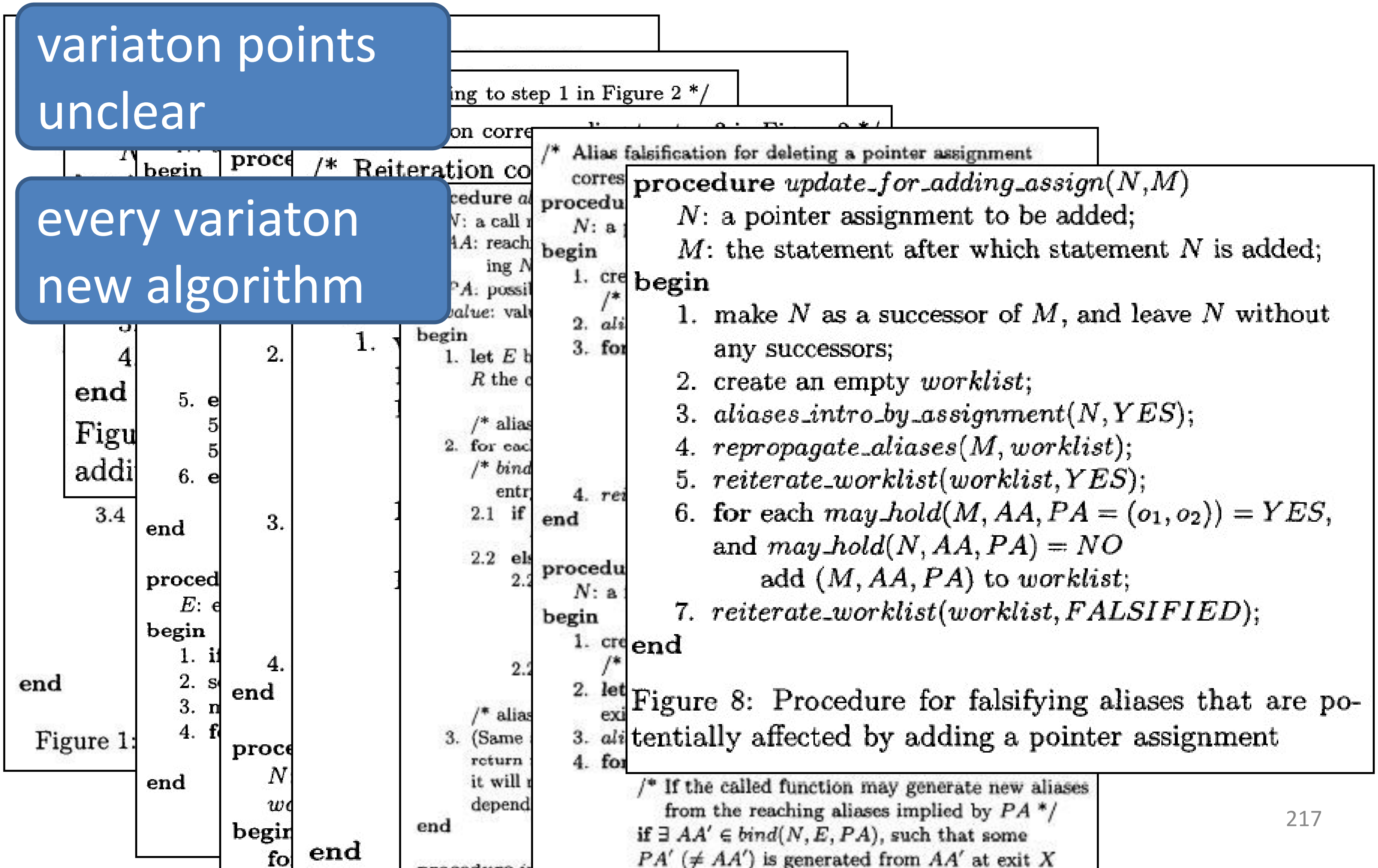
Figure 8: Procedure for falsifying aliases that are potentially affected by adding a pointer assignment

```
/* If the called function may generate new aliases
   from the reaching aliases implied by PA */
if ∃ AA' ∈ bind(N, E, PA), such that some
PA' (≠ AA') is generated from AA' at exit X
```

Algorithms in 10-page Conf. Papers

variaton points unclear

every variaton new algorithm



/* Alias falsification for deleting a pointer assignment

```
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 = (o_1, o_2)) = 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

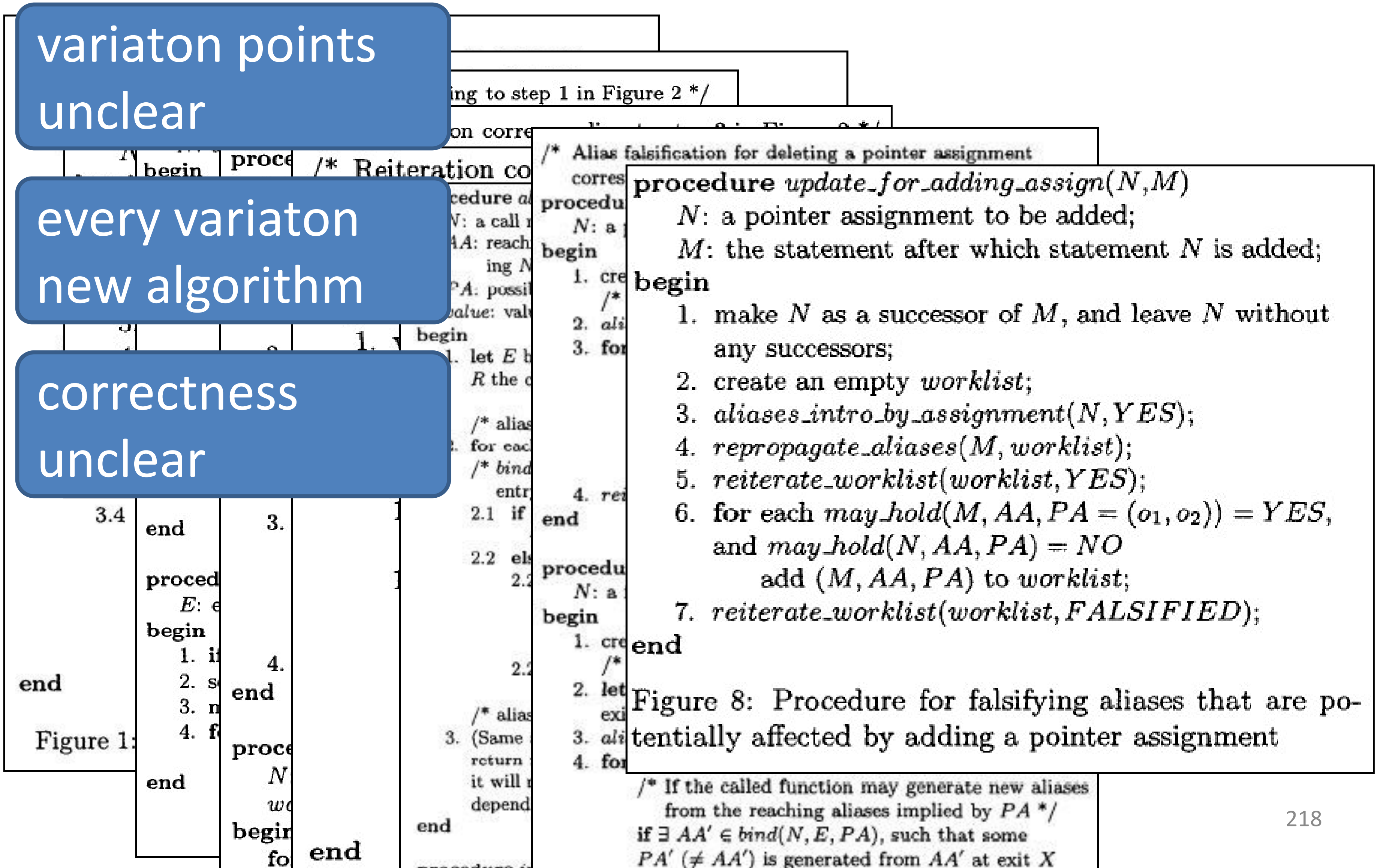
/* If the called function may generate new aliases from the reaching aliases implied by PA */
if $\exists$ AA′ $\in$ bind(N, E, PA), such that some PA′ ($\neq$ AA′) is generated from AA′ at exit X

Algorithms in 10-page Conf. Papers

variaton points unclear

every variaton new algorithm

correctness unclear



/* Alias falsification for deleting a pointer assignment

```
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 = (o_1,o_2)) = 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

/* If the called function may generate new aliases from the reaching aliases implied by PA */
if ∃ AA′ ∈ bind(N,E,PA), such that some PA′ (≠ AA′) is generated from AA′ at exit X

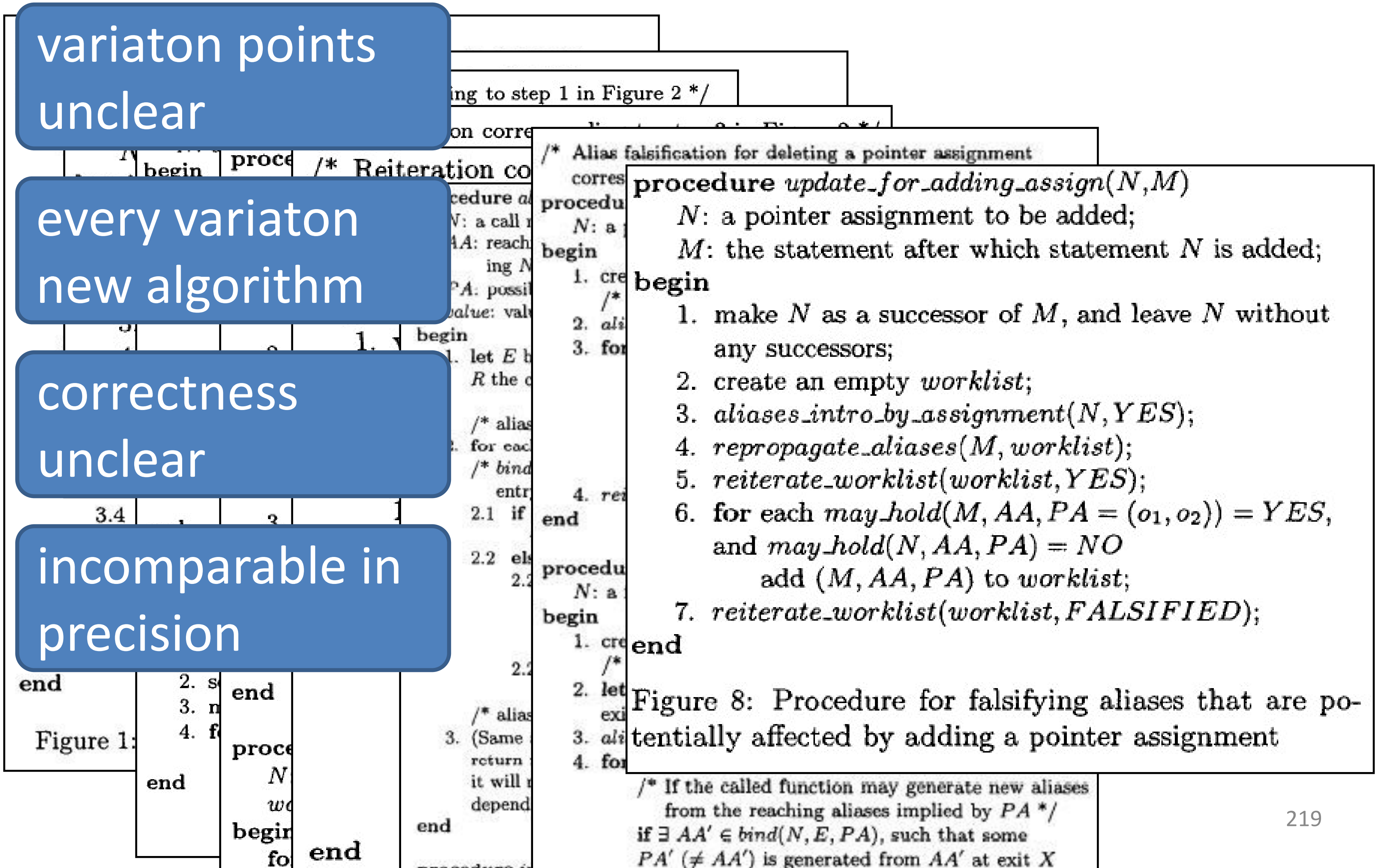
Algorithms in 10-page Conf. Papers

variaton points unclear

every variaton new algorithm

correctness unclear

incomparable in precision



```
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 = (o_1, o_2)) = 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

/* Alias falsification for deleting a pointer assignment

/* If the called function may generate new aliases from the reaching aliases implied by PA */
if $\exists AA' \in bind(N, E, PA)$, such that some PA' ($\neq AA'$) is generated from AA' at exit X

Algorithms in 10-page Conf. Papers

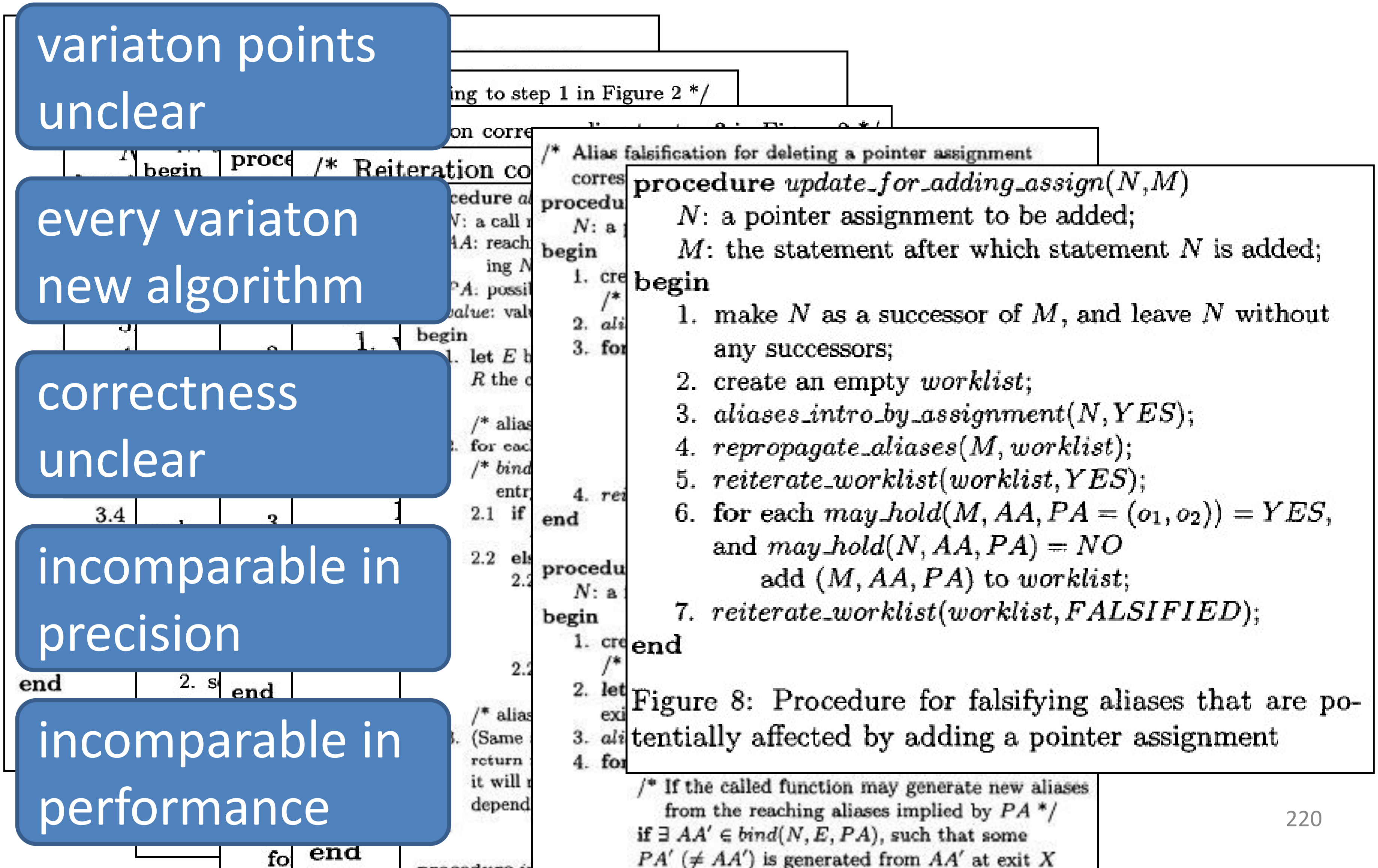
variaton points unclear

every variaton new algorithm

correctness unclear

incomparable in precision

incomparable in performance



```
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 = (o_1, o_2)) = 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

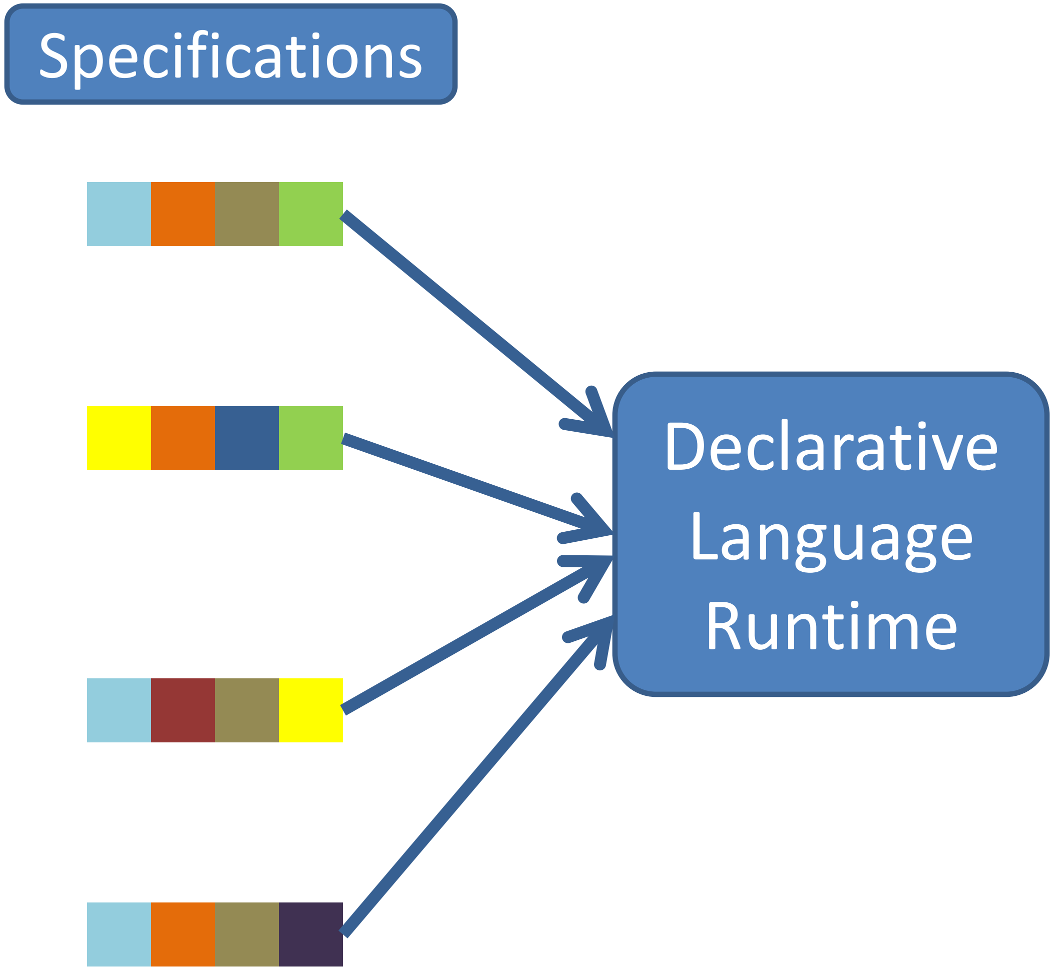
```
/* Alias falsification for deleting a pointer assignment
```

```
/* If the called function may generate new aliases
   from the reaching aliases implied by PA */
if ∃ AA′ ∈ bind(N, E, PA), such that some
PA′ (≠ AA′) is generated from AA′ at exit X
```

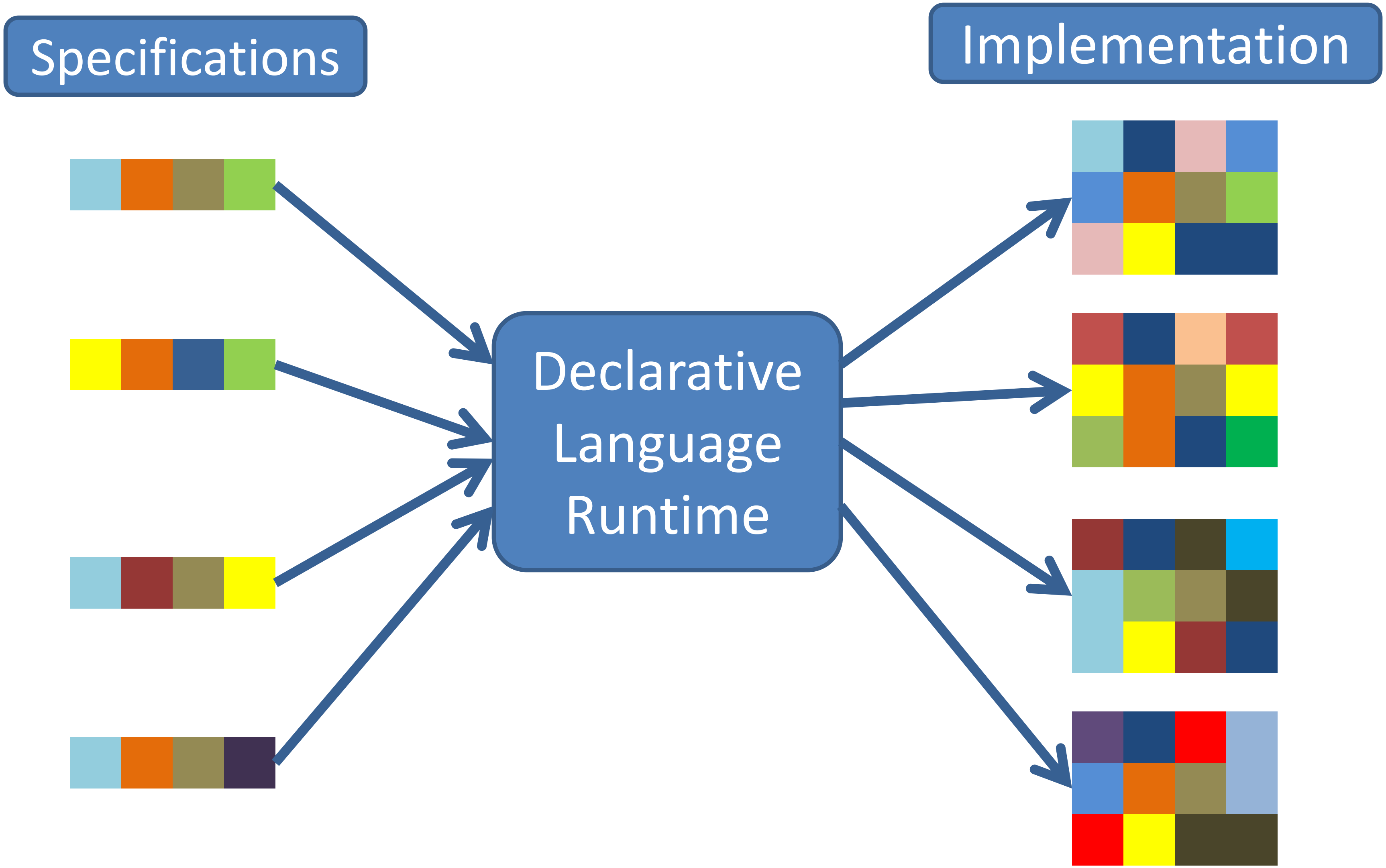
Want: Specification + Implementation

Specifications

Want: Specification + Implementation



Want: Specification + Implementation



Specifications

Implementation

Declarative Language Runtime

DECLARATIVE = GOOD

WHY DATALOG?

Program Analysis: Domain of Mutual Recursion

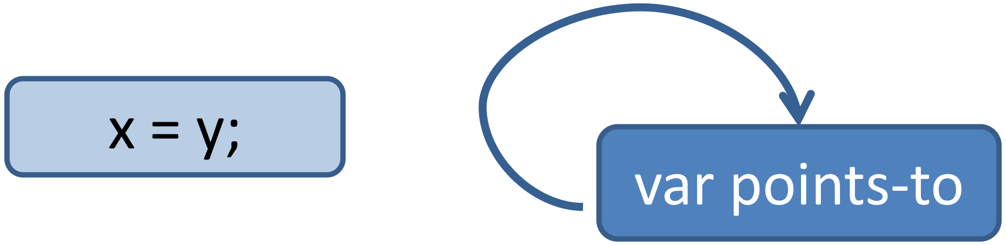
var points-to

Program Analysis: Domain of Mutual Recursion

```
x = y;
```

var points-to

Program Analysis: Domain of Mutual Recursion

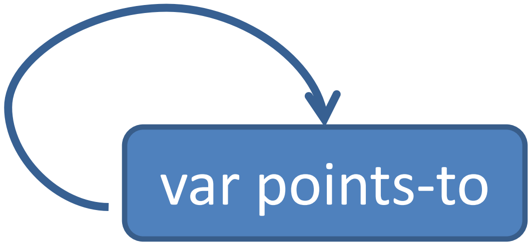


x = y;

var points-to

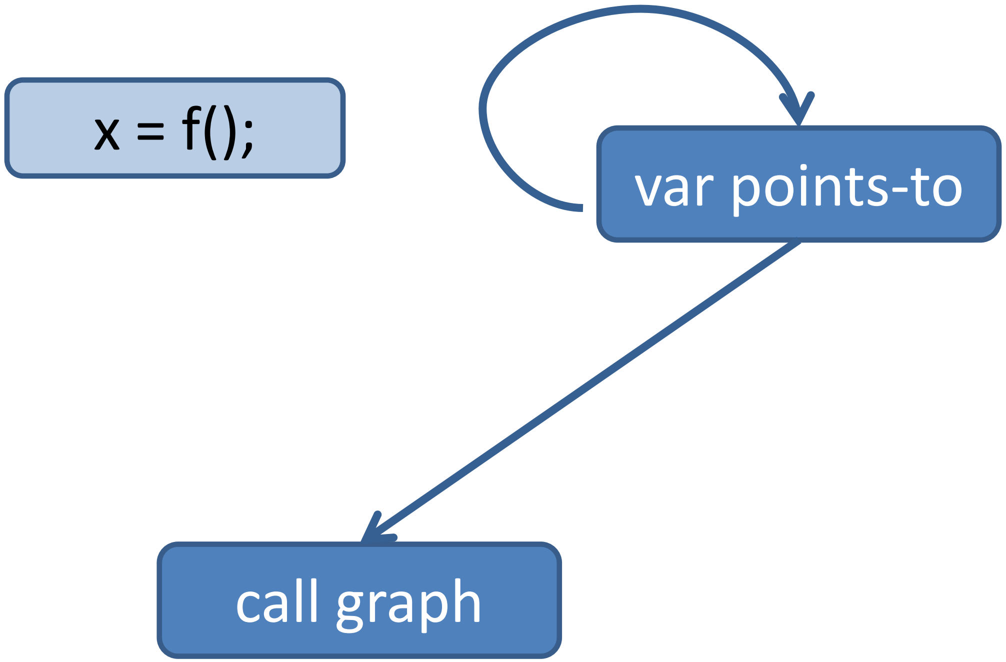
Program Analysis: Domain of Mutual Recursion

```
x = f();
```



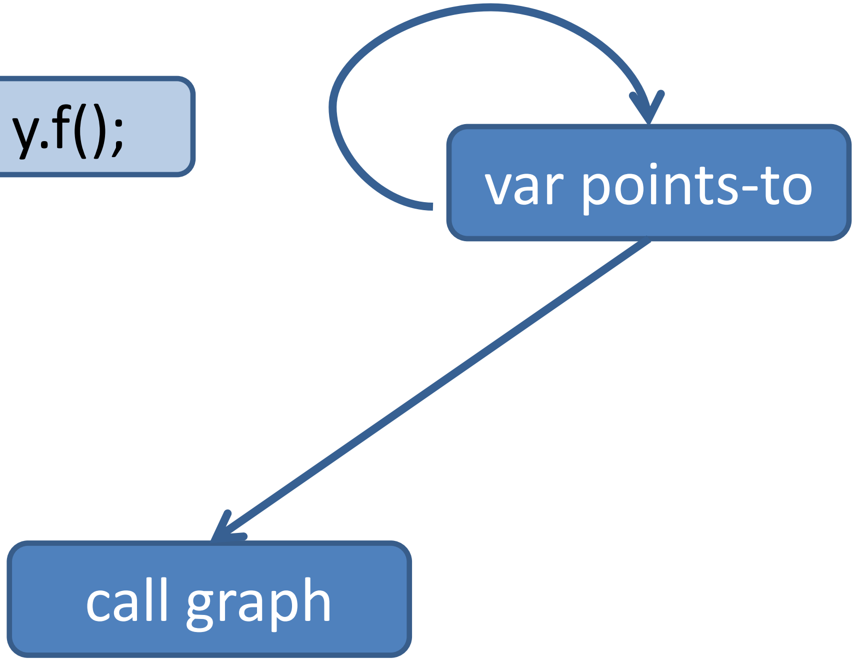
var points-to

Program Analysis: Domain of Mutual Recursion



Program Analysis: Domain of Mutual Recursion

`x = y.f();`

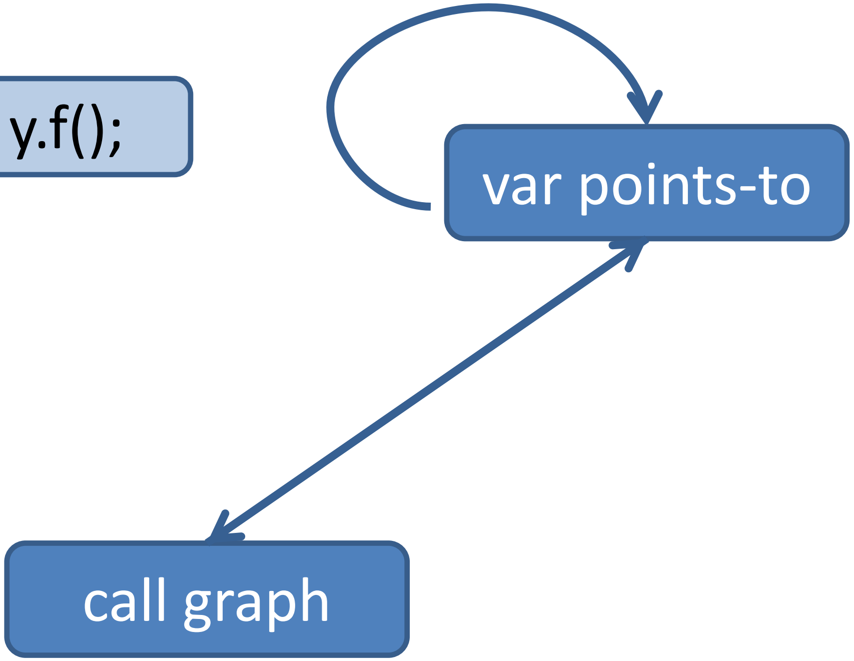


var points-to

call graph

Program Analysis: Domain of Mutual Recursion

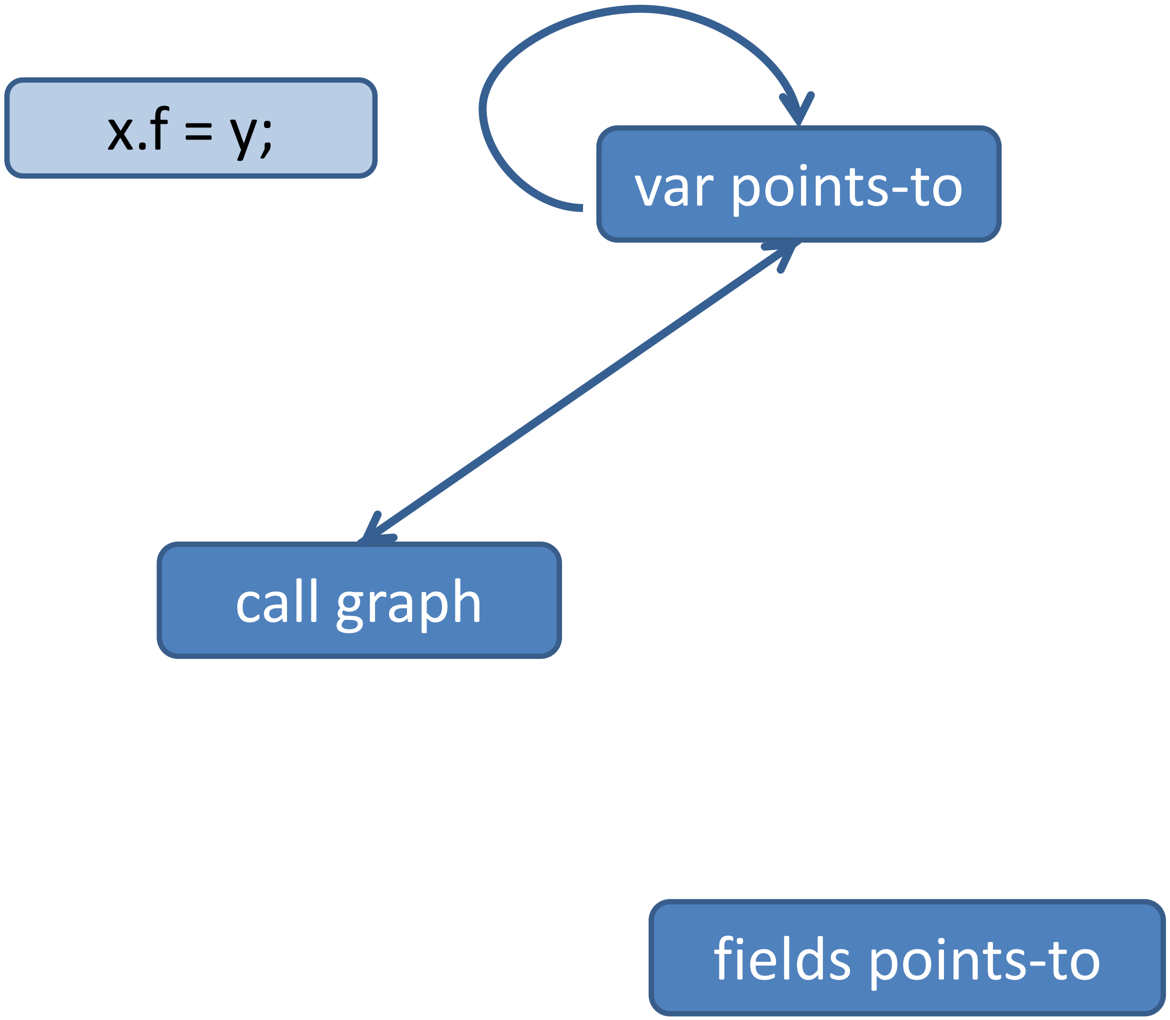
```
x = y.f();
```



var points-to

call graph

Program Analysis: Domain of Mutual Recursion



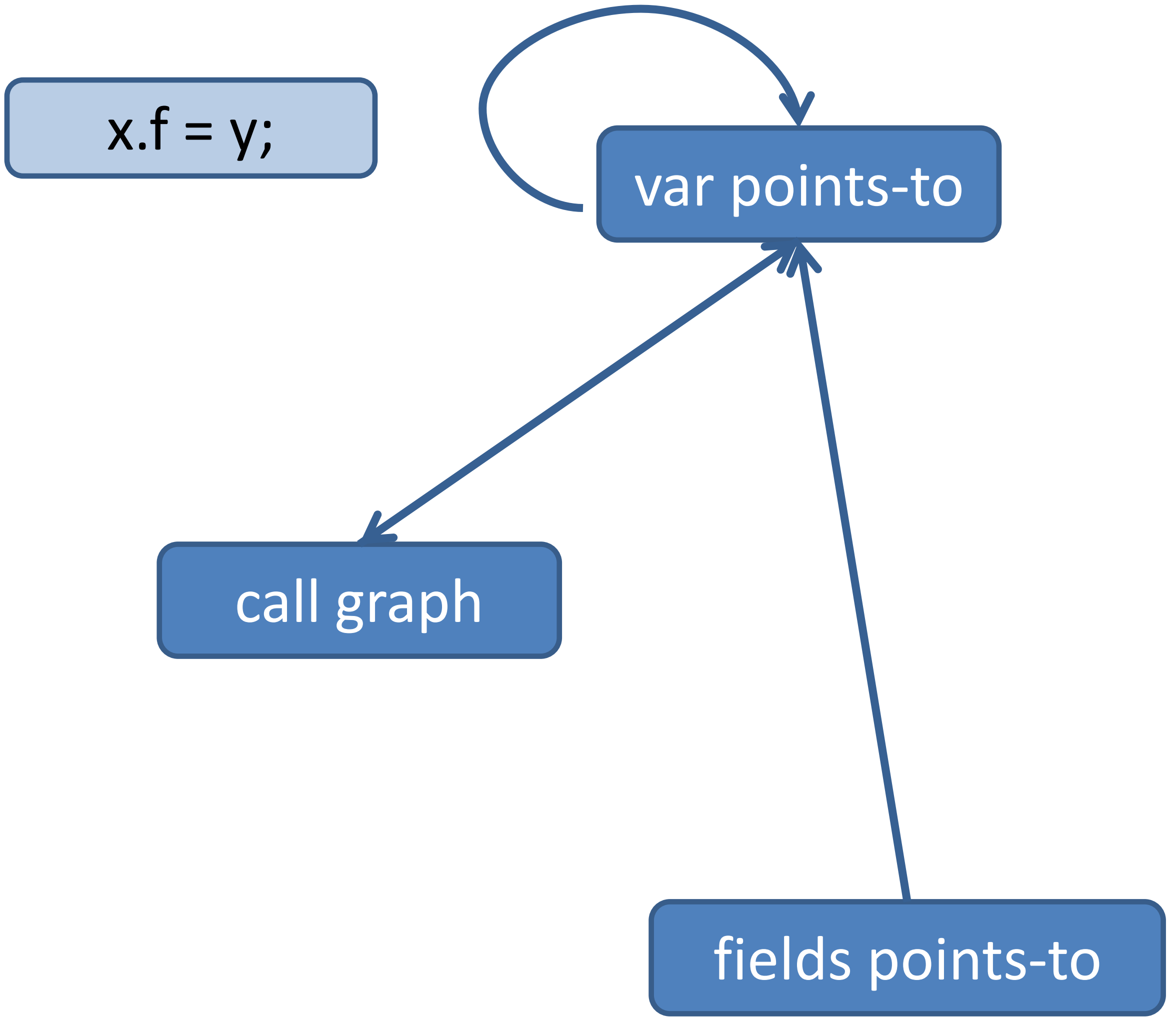
x.f = y;

var points-to

call graph

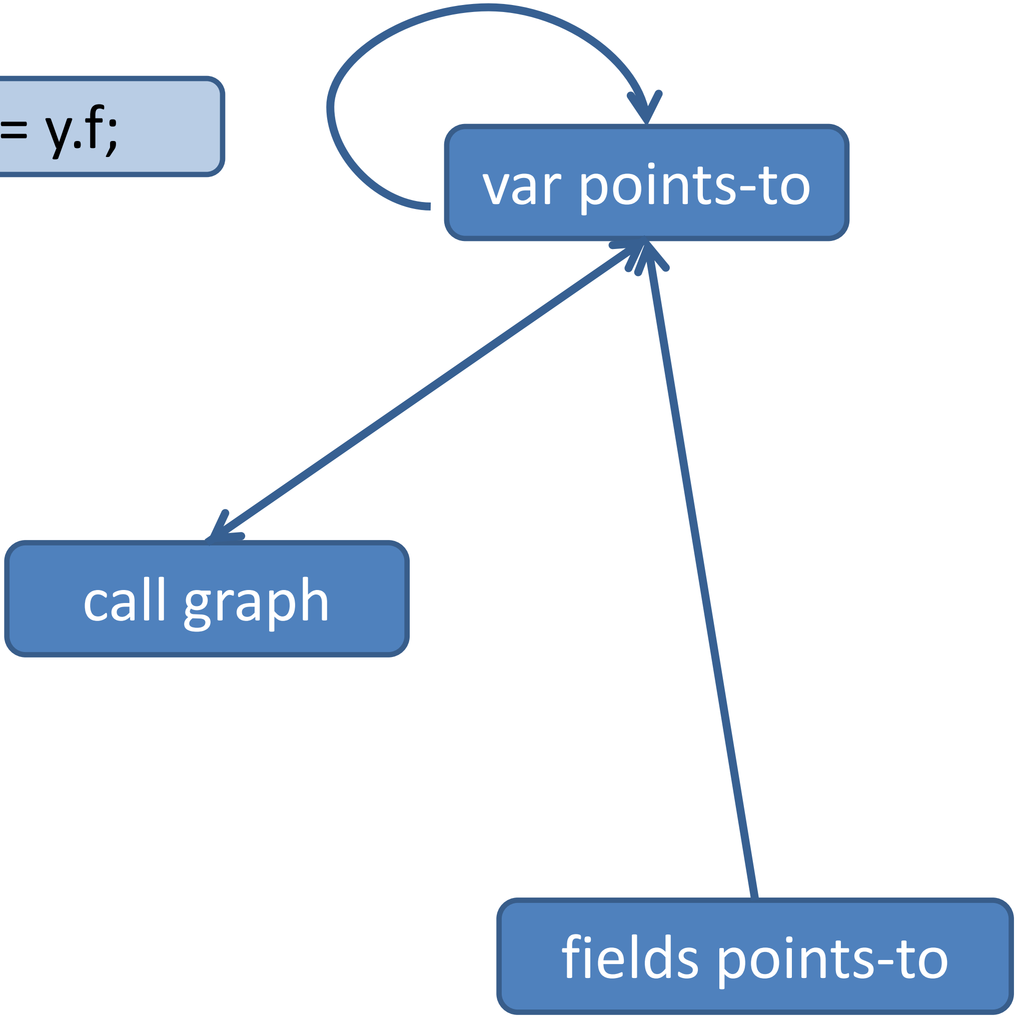
fields points-to

Program Analysis: Domain of Mutual Recursion

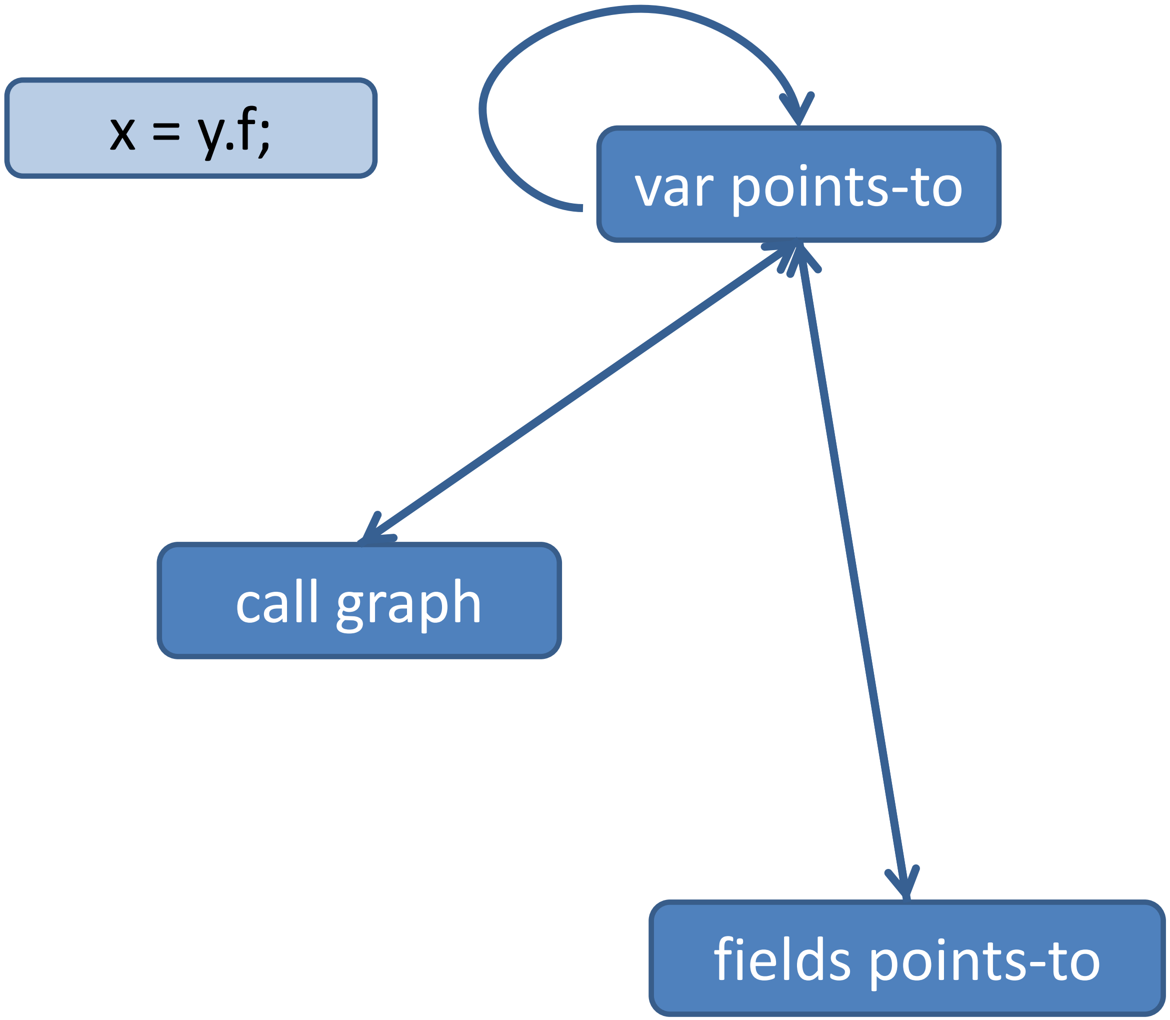


Program Analysis: Domain of Mutual Recursion

```
x = y.f;
```

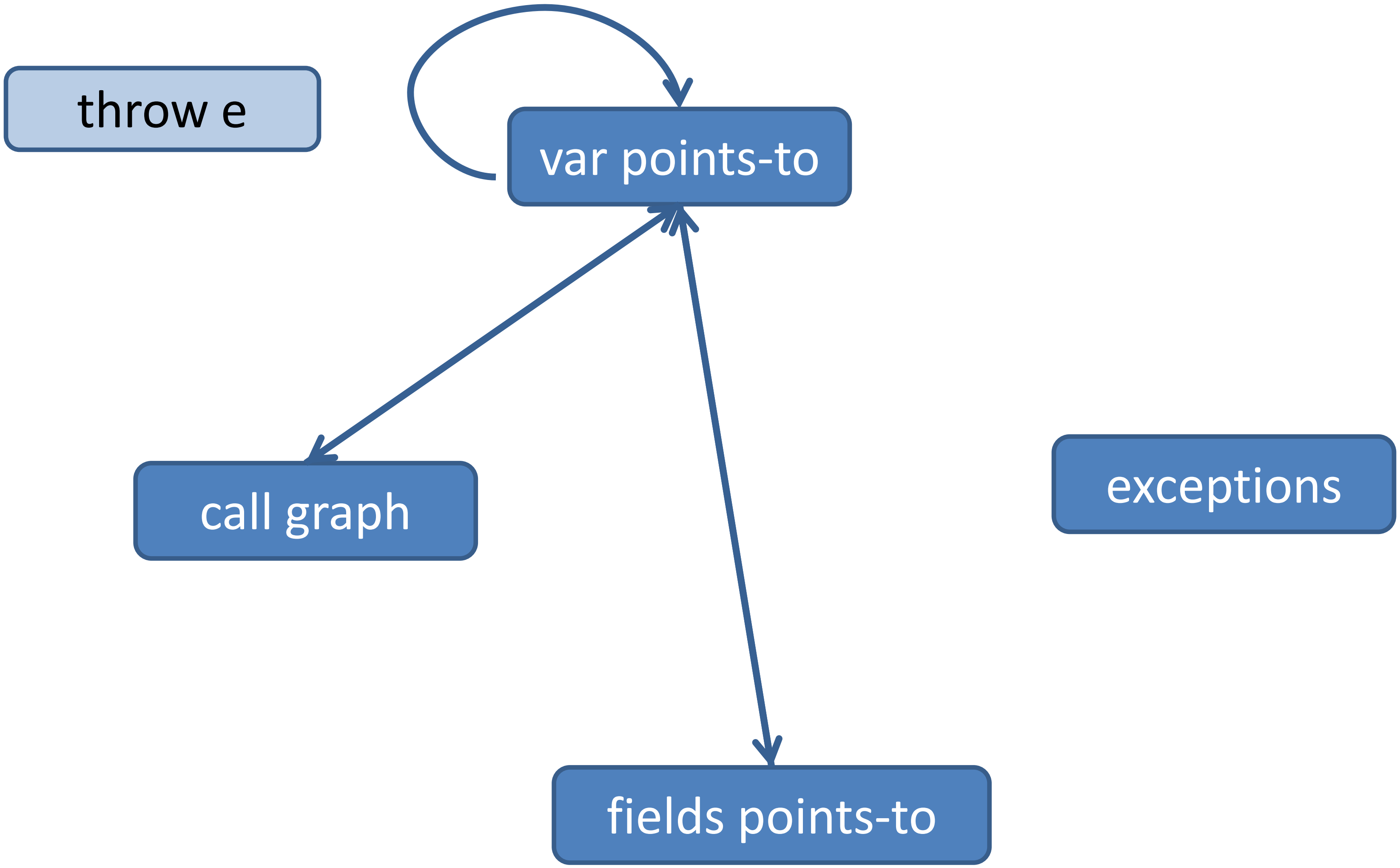


Program Analysis: Domain of Mutual Recursion

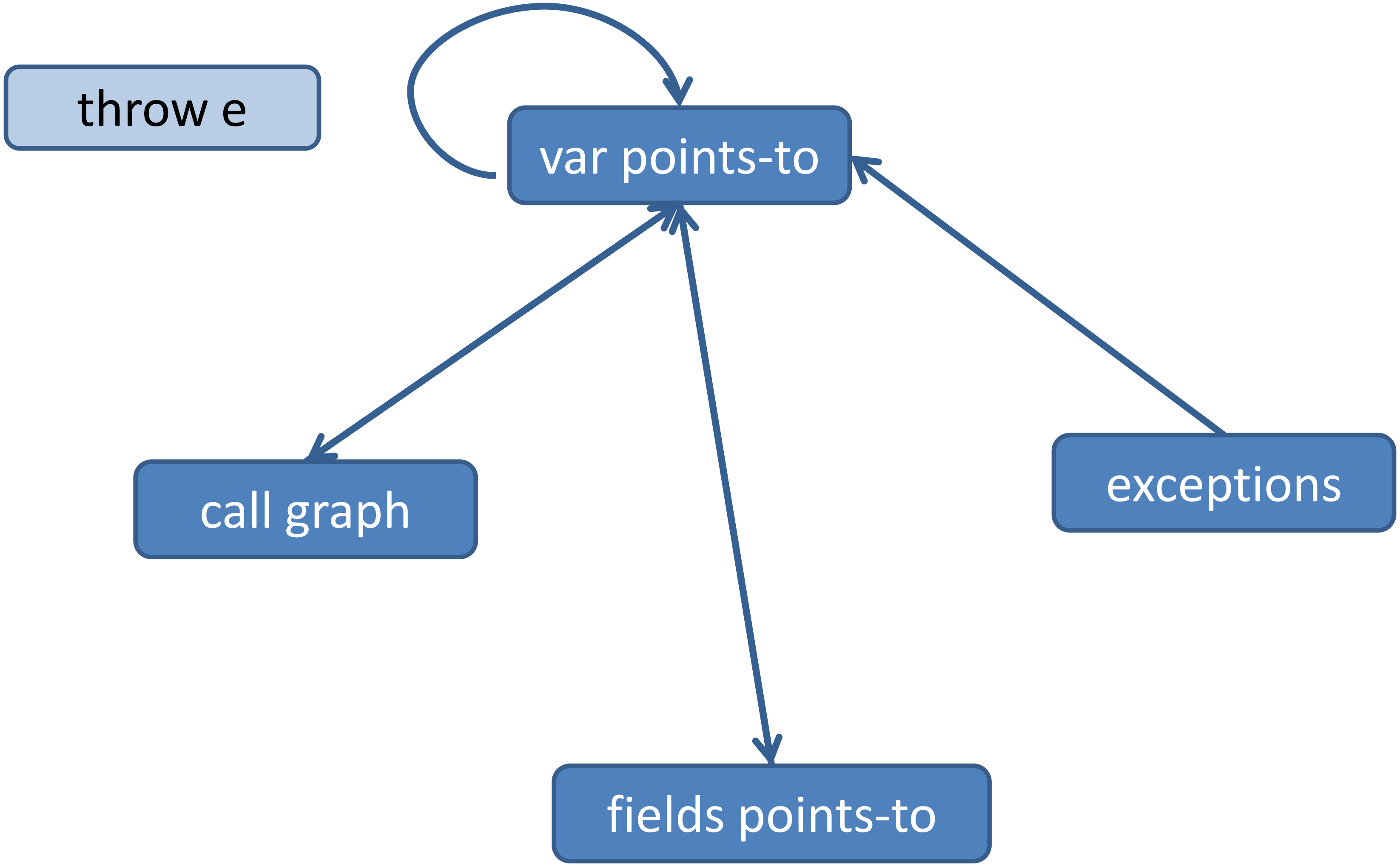


```
x = y.f;
```

Program Analysis: Domain of Mutual Recursion



Program Analysis: Domain of Mutual Recursion



throw e

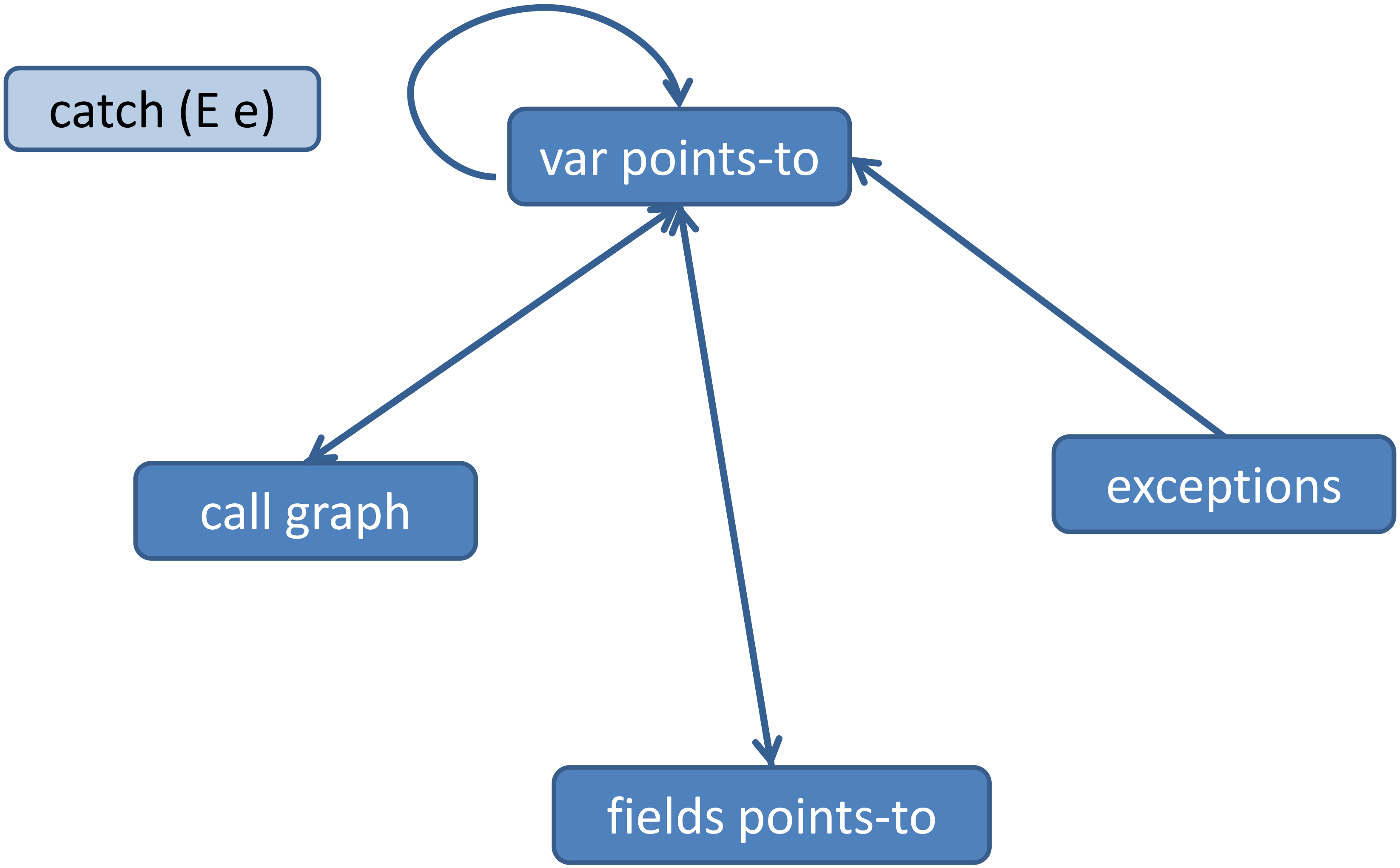
var points-to

call graph

exceptions

fields points-to

Program Analysis: Domain of Mutual Recursion



catch (E e)

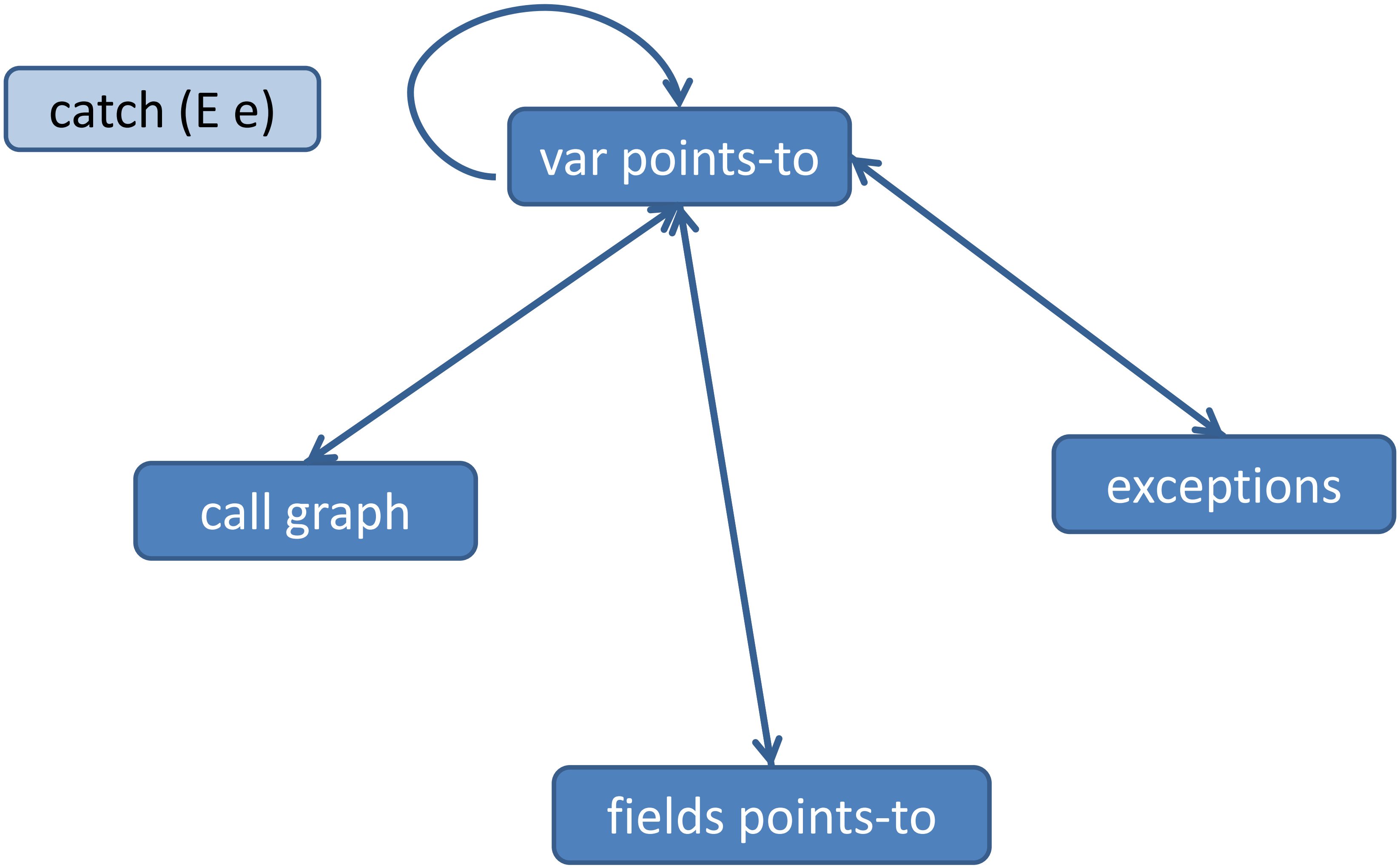
var points-to

call graph

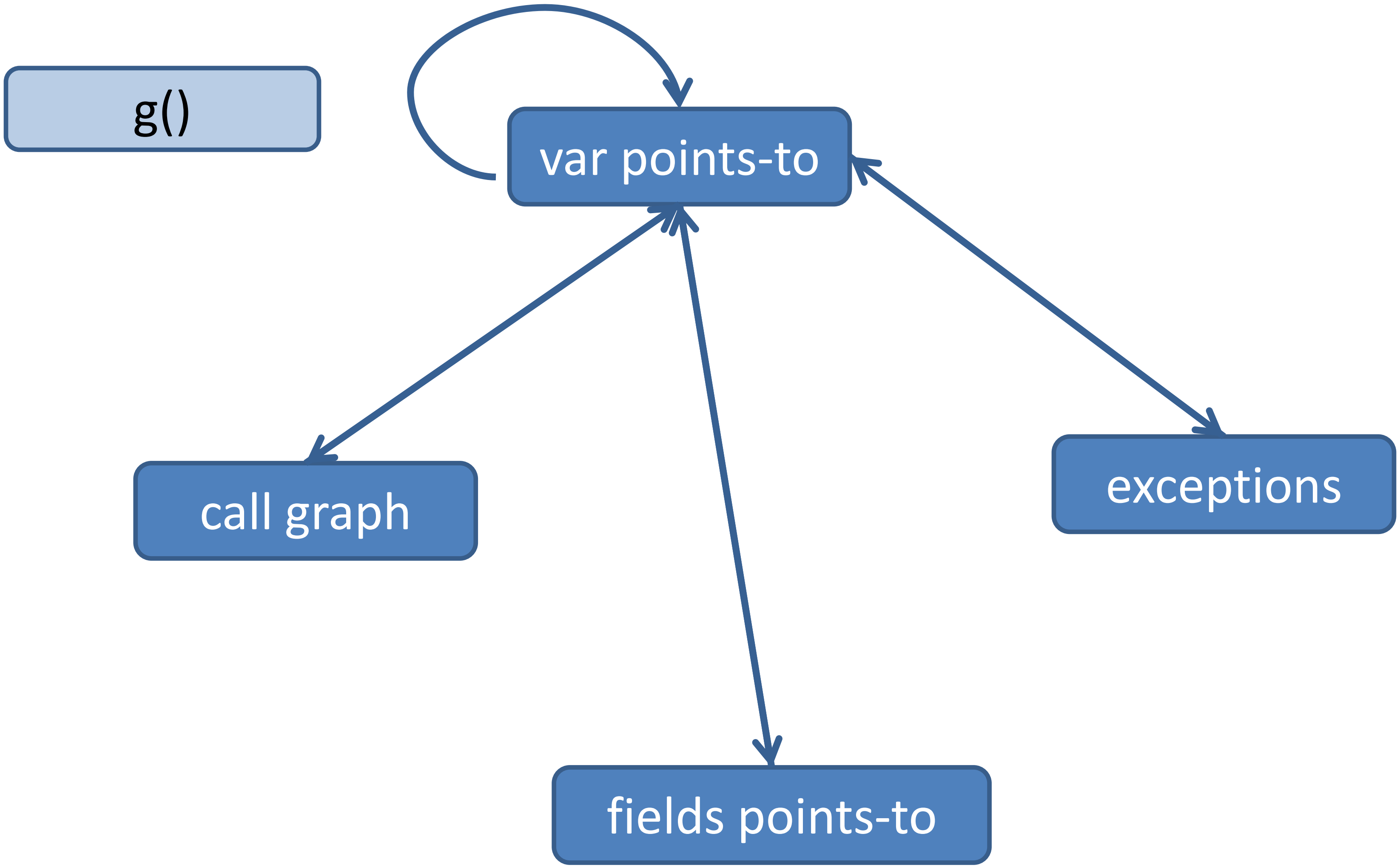
exceptions

fields points-to

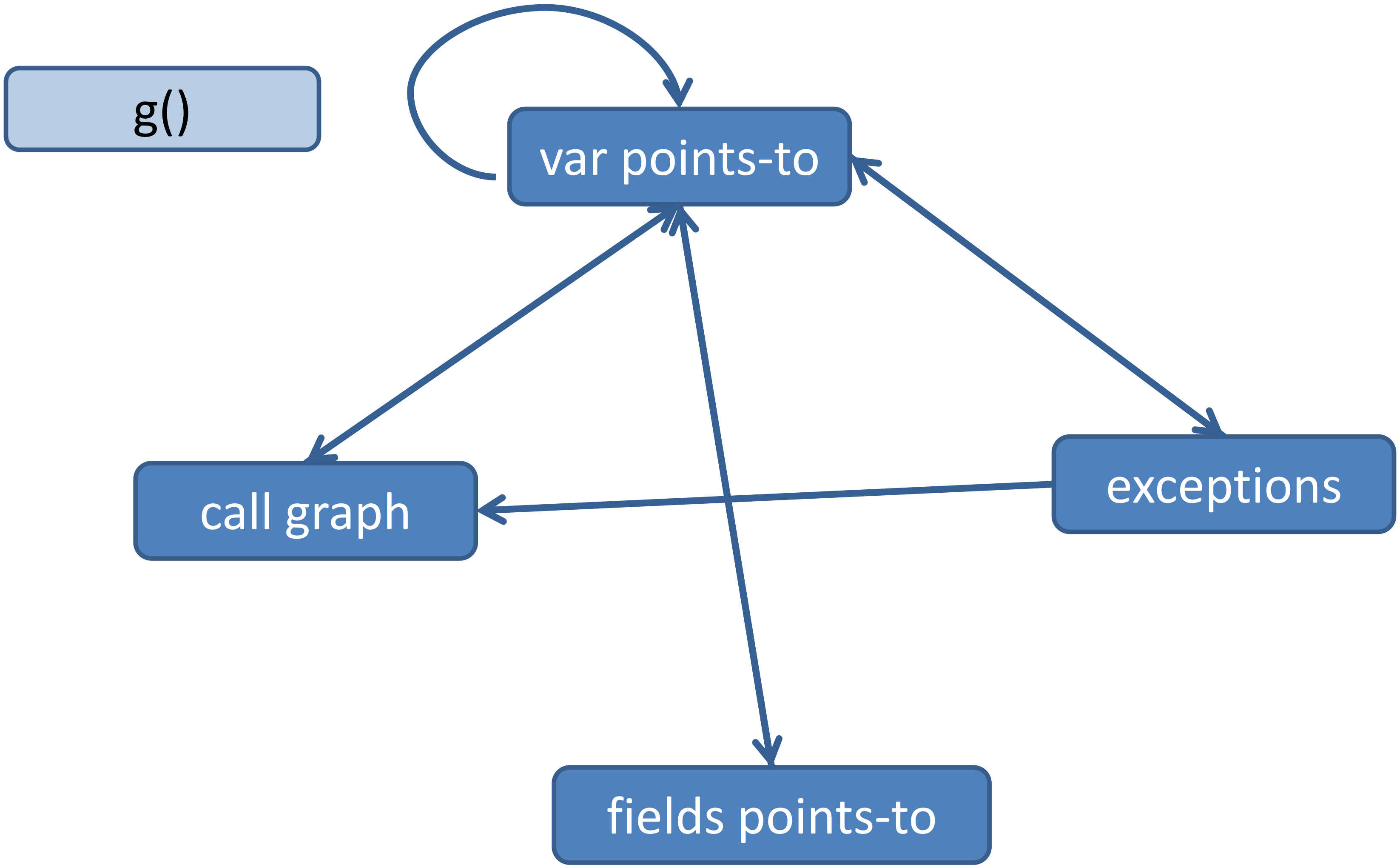
Program Analysis: Domain of Mutual Recursion



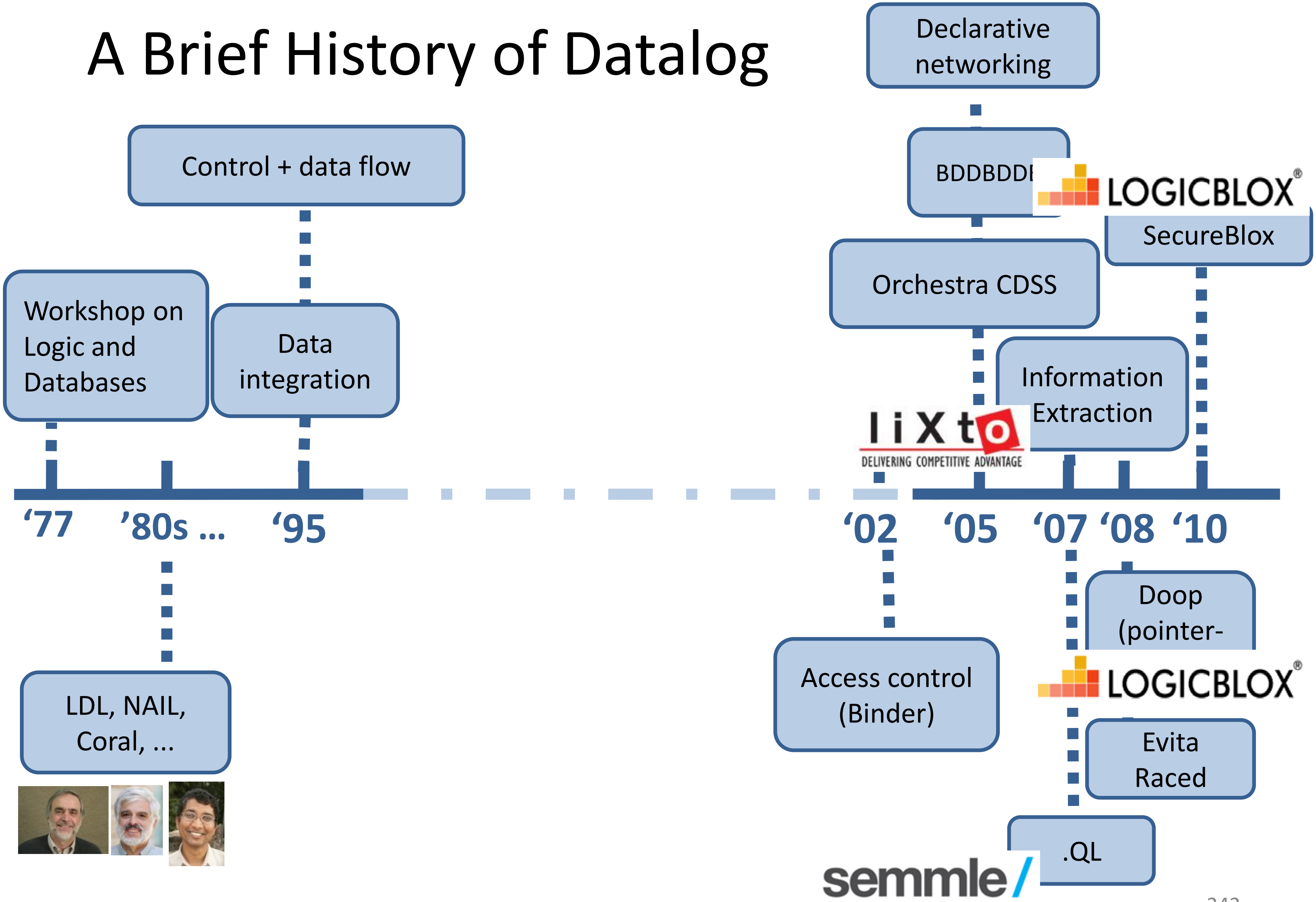
Program Analysis: Domain of Mutual Recursion



Program Analysis: Domain of Mutual Recursion



A Brief History of Datalog



Workshop on Logic and Databases

Control + data flow

Data integration

LDL, NAIL, Coral, ...

‘77

’80s ...

‘95

‘02

‘05

‘07

‘08

‘10

Access control (Binder)

Declarative networking

BDDBDDB

LOGICBLOX

SecureBlox

Orchestra CDSS

Information Extraction

liXto

DELIVERING COMPETITIVE ADVANTAGE

Doop (pointer-

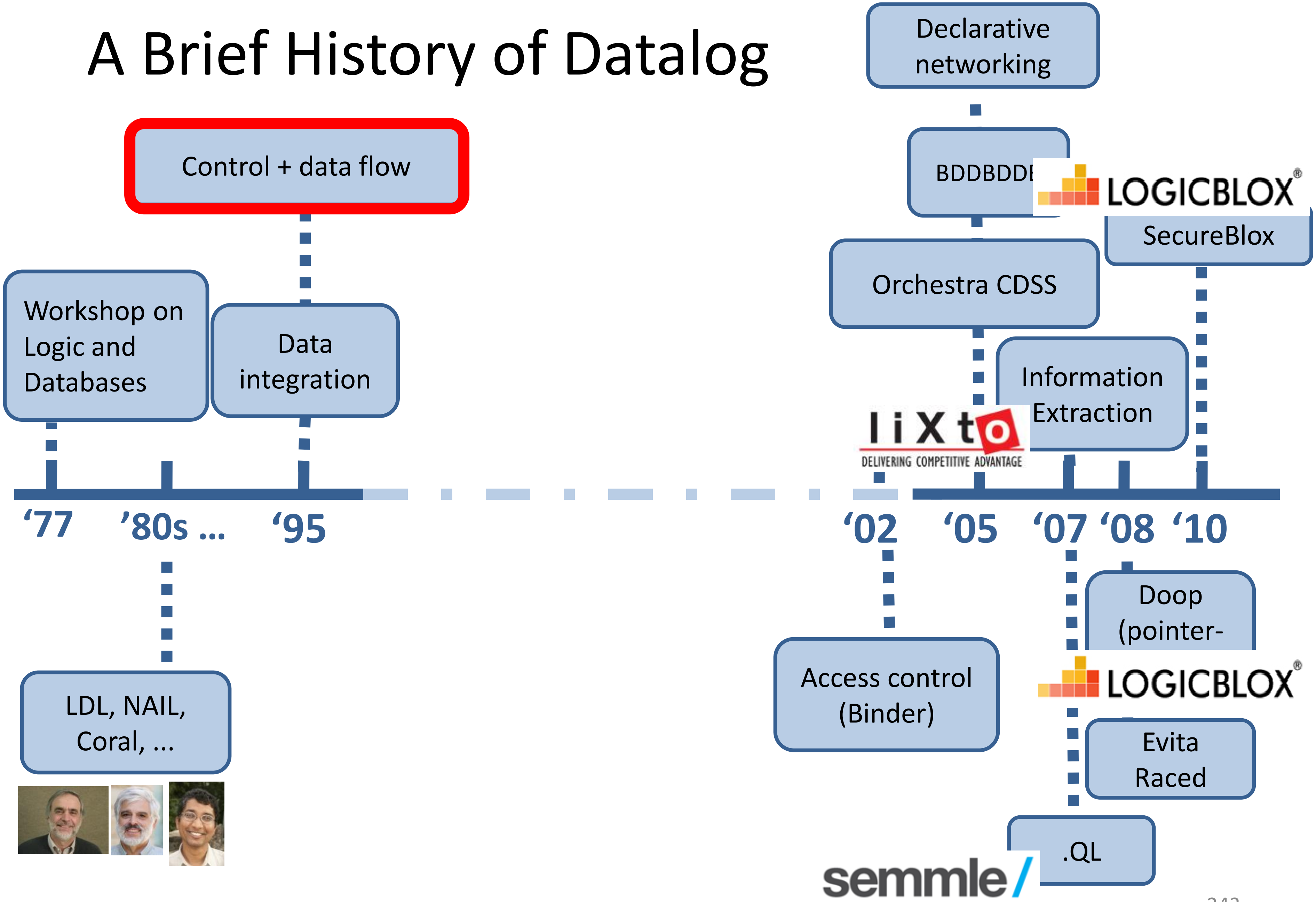
LOGICBLOX

Evita Raced

.QL

semmle

A Brief History of Datalog



Control + data flow

Workshop on Logic and Databases

Data integration

LDL, NAIL, Coral, ...

Declarative networking

BDDBDDB

SecureBlox

Orchestra CDSS

Information Extraction

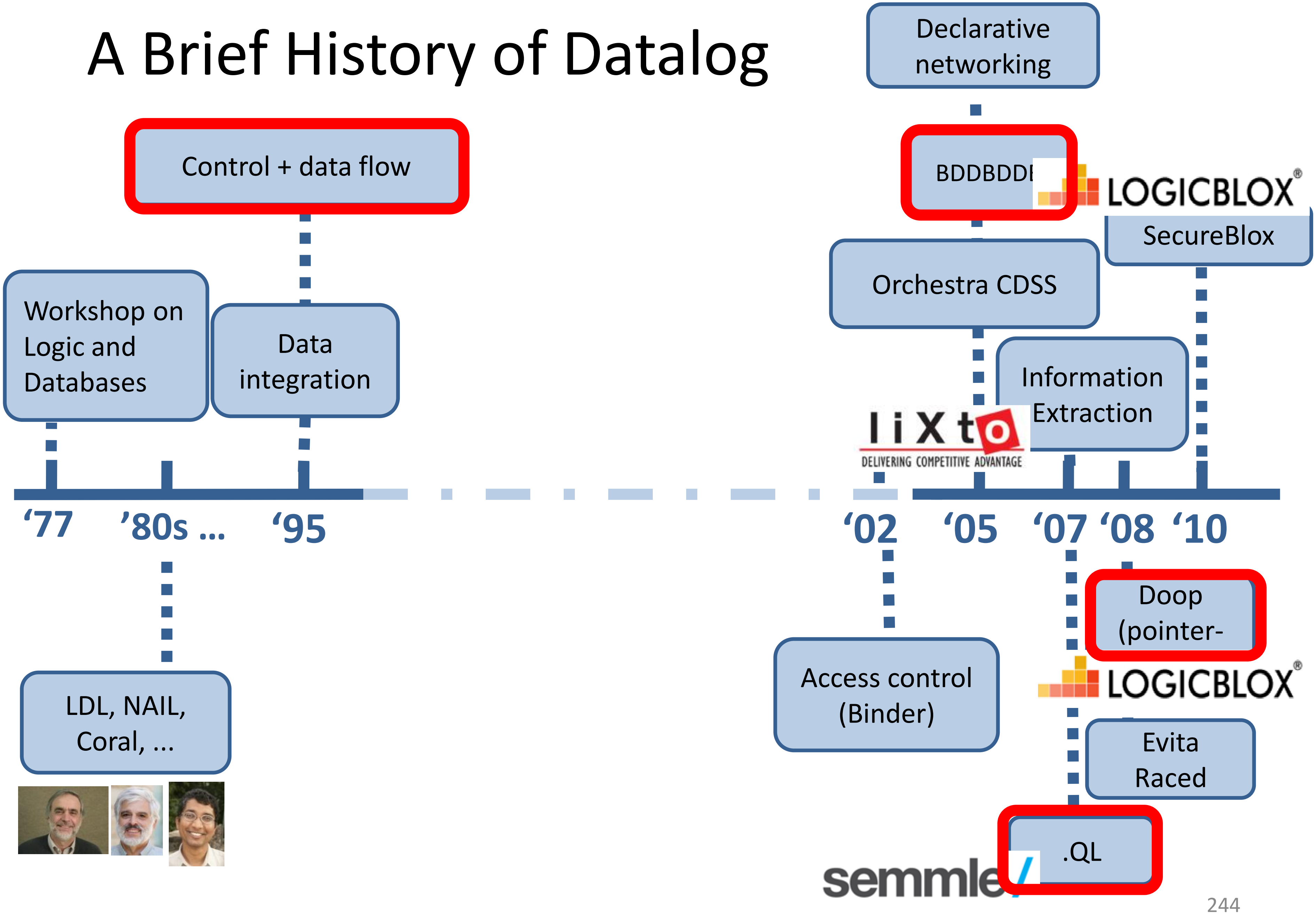
Access control (Binder)

Doop (pointer-

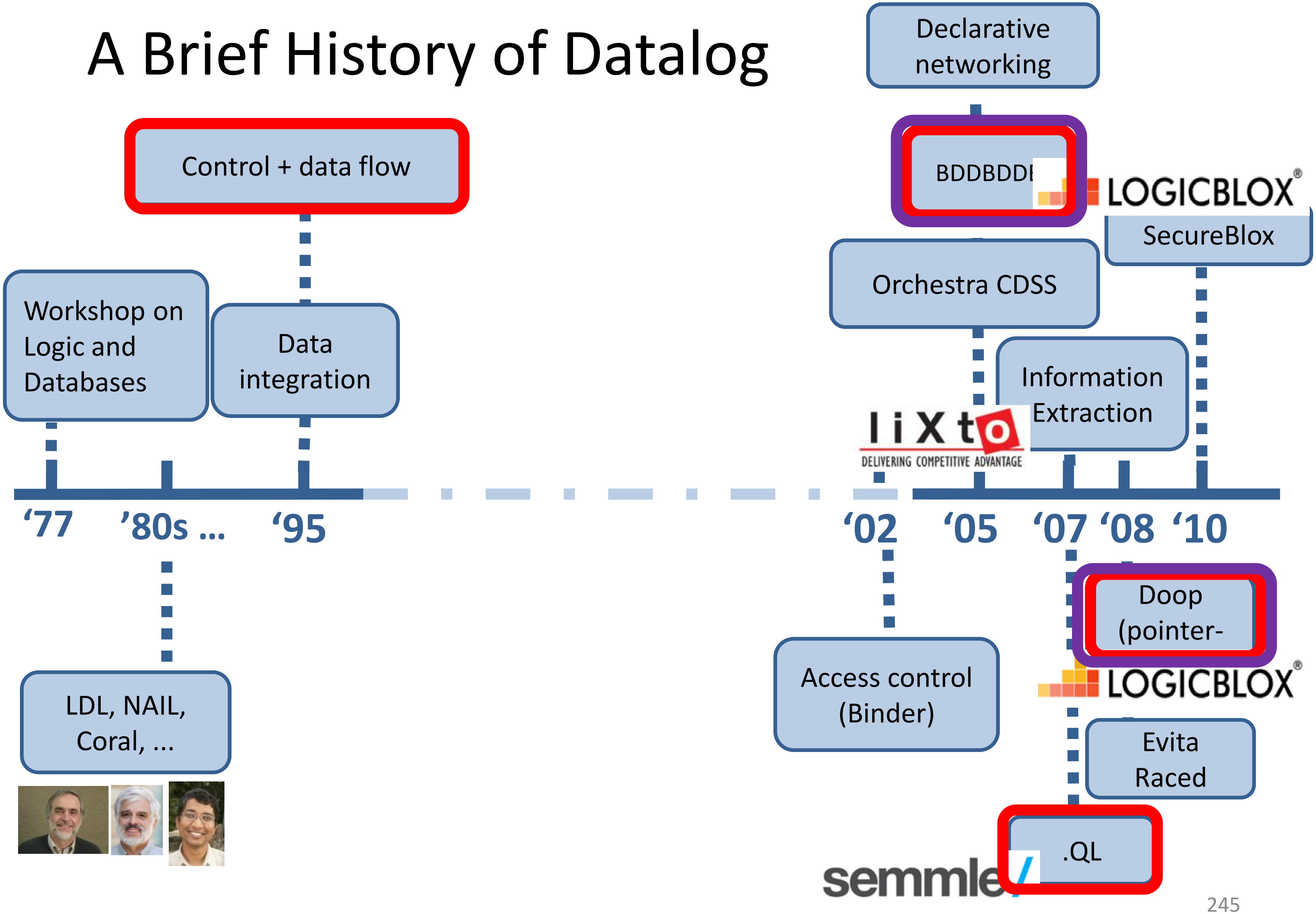
Evita Raced

.QL

A Brief History of Datalog



A Brief History of Datalog



Control + data flow

Workshop on Logic and Databases

Data integration

‘77

’80s …

‘95

LDL, NAIL, Coral, …

Declarative networking

BDDBDDB

LOGICBLOX

SecureBlox

Orchestra CDSS

Information Extraction

lixto DELIVERING COMPETITIVE ADVANTAGE

‘02

‘05

‘07

‘08

‘10

Access control (Binder)

Doop (pointer-

LOGICBLOX

Evita Raced

.QL

semmle

PROGRAM ANALYSIS IN DATALOG

Points-to Analyses for A Simple Language

program

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

Points-to Analyses for A Simple Language

program

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

Points-to Analyses for A Simple Language

program

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

Points-to Analyses for A Simple Language

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

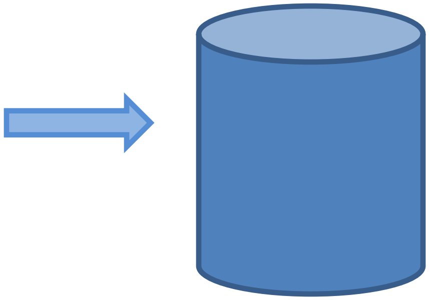
```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

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;
```

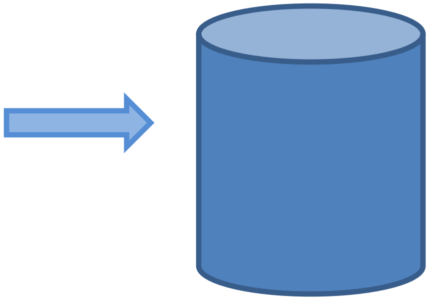


Points-to Analyses for A Simple Language

What objects can a variable point to?

program

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

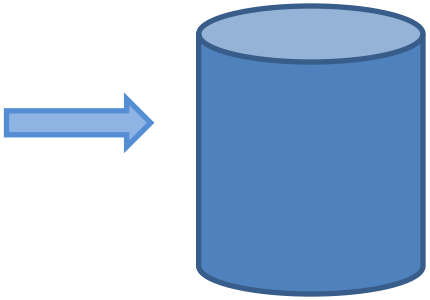


assignObjectAllocation

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;



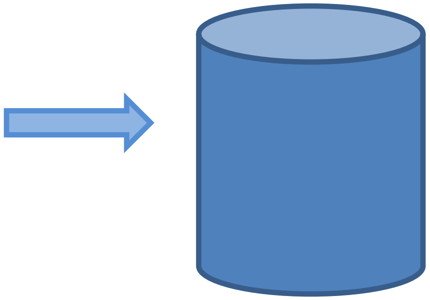
assignObjectAllocation	
a	new A()

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;
```



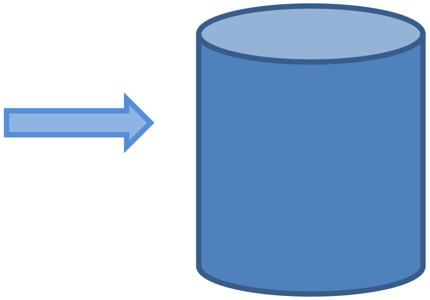
assignObjectAllocation	
a	new A()
b	new B()

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;
```



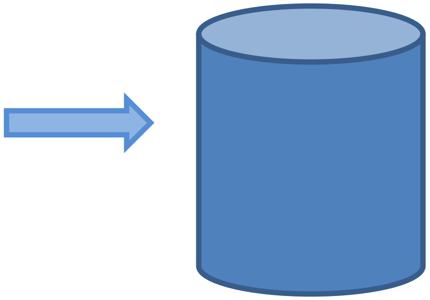
assignObjectAllocation	
a	new A()
b	new B()
c	new C()

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;
```



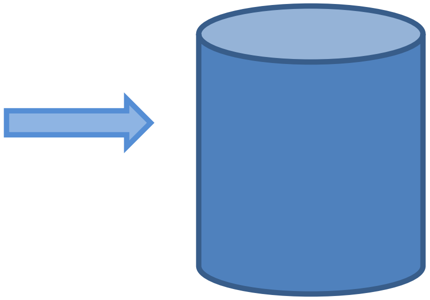
assignObjectAllocation	
a	new A()
b	new B()
c	new C()

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

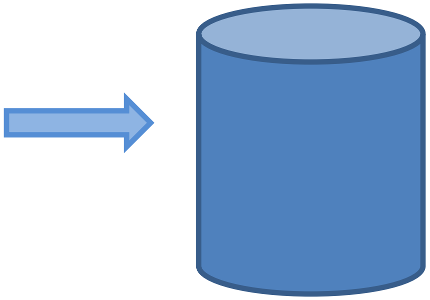
assign

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

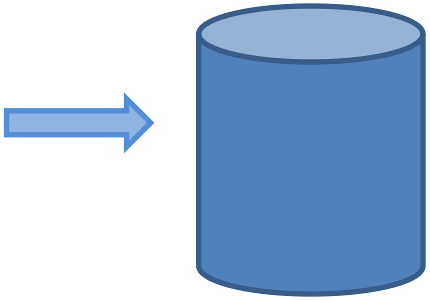
assign	
b	a

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

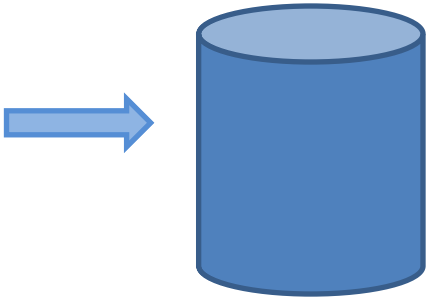
assign	
b	a
a	b

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;
```



assignObjectAllocation	
a	new A()
b	new B()
c	new C()

assign	
b	a
a	b
b	c

Defining varPointsTo

program

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

assignObjectAllocation	
a	new A()
b	new B()
c	new C()

assign	
b	a
a	b
b	c

Defining varPointsTo

program

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

assignObjectAllocation	
a	new A()
b	new B()
c	new C()

assign	
b	a
a	b
b	c

varPointsTo

Defining varPointsTo

program

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

assignObjectAllocation	
a	new A()
b	new B()
c	new C()

assign	
b	a
a	b
b	c

varPointsTo	

Defining varPointsTo

program

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

assignObjectAllocation	
a	new A()
b	new B()
c	new C()

assign	
b	a
a	b
b	c

varPointsTo

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

Defining varPointsTo

program

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

assignObjectAllocation	
a	new A()
b	new B()
c	new C()

assign	
b	a
a	b
b	c

varPointsTo	
a	new A()
b	new B()
c	new C()

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

Defining varPointsTo

program

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

assignObjectAllocation	
a	new A()
b	new B()
c	new C()

assign	
b	a
a	b
b	c

varPointsTo	
a	new A()
b	new B()
c	new C()

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

Defining varPointsTo

program

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

assignObjectAllocation	
a	new A()
b	new B()
c	new C()

assign	
b	a
a	b
b	c

varPointsTo	
a	new A()
b	new B()
c	new C()

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

Defining varPointsTo

program

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

assignObjectAllocation	
a	new A()
b	new B()
c	new C()

assign	
b	a
a	b
b	c

varPointsTo	
a	new A()
b	new B()
c	new C()

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

assign	
b	a
a	b
b	c

varPointsTo	
a	new A()
b	new B()
c	new C()
a	new B()

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

assign	
b	a
a	b
b	c

varPointsTo	
a	new A()
b	new B()
c	new C()
a	new B()

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

assign	
b	a
a	b
b	c

varPointsTo	
a	new A()
b	new B()
c	new C()
a	new B()
b	new A()

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

assign	
b	a
a	b
b	c

varPointsTo	
a	new A()
b	new B()
c	new C()
a	new B()
b	new A()
c	new B()
c	new A()

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

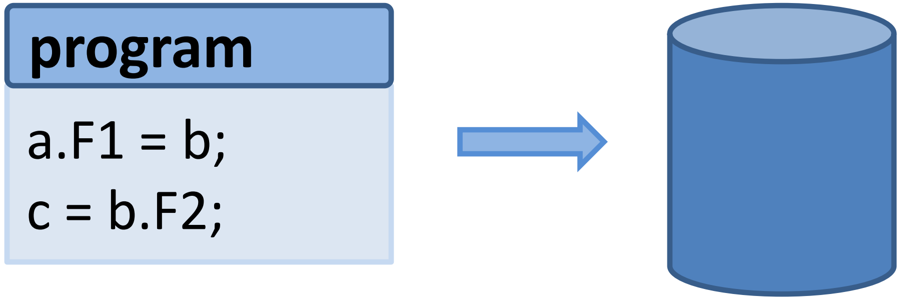
```
varPointsTo(To, Obj)
  <- assign(From, To), varPointsTo(From,Obj).
```

Introducing Fields

program

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

Introducing Fields

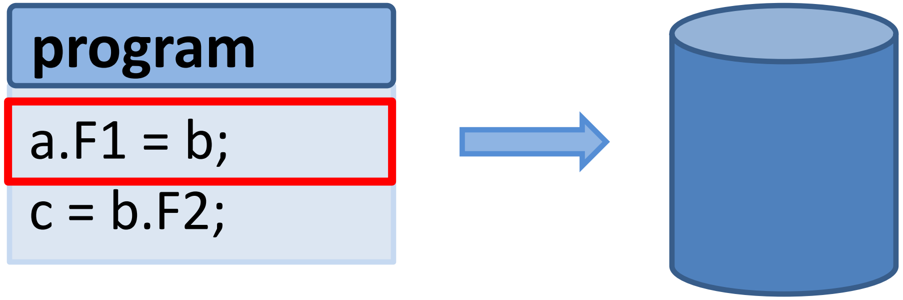


program

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

275

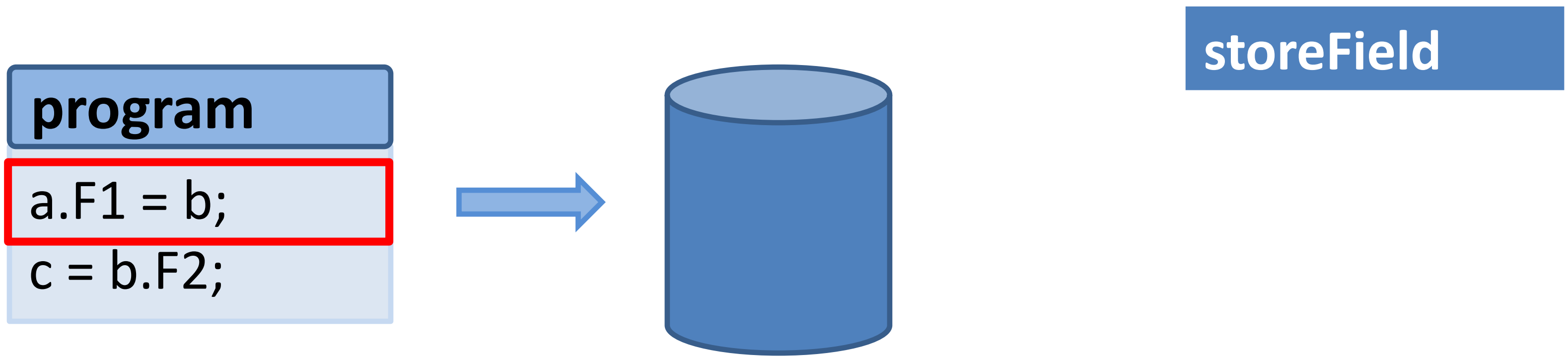
Introducing Fields



program

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

Introducing Fields

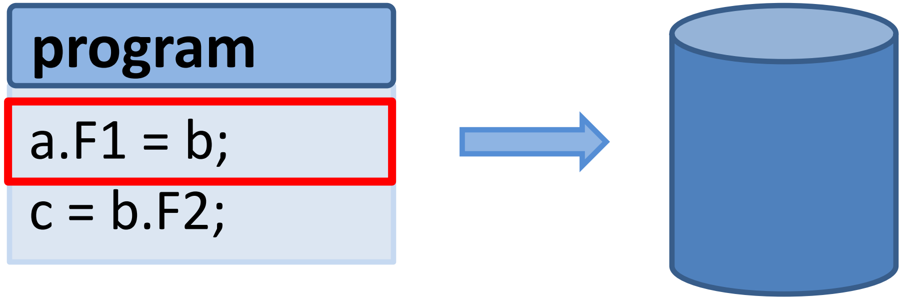


storeField

program

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

Introducing Fields



program

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

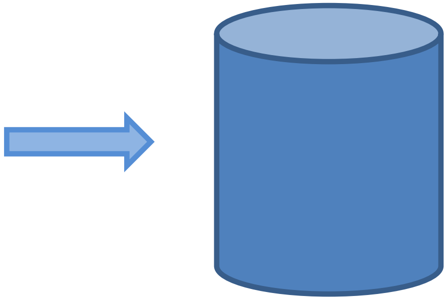
storeField		
b	a	F1

278

Introducing Fields

program

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



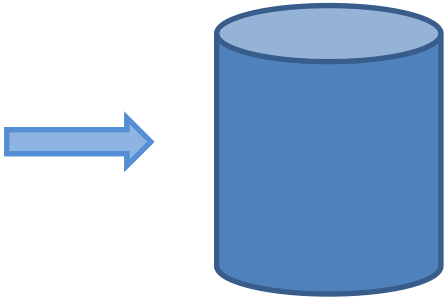
storeField		
b	a	F1

279

Introducing Fields

program

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

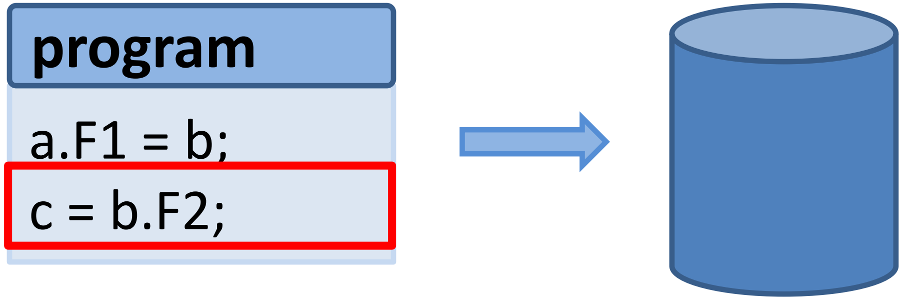


storeField		
b	a	F1

loadField

280

Introducing Fields



program

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

storeField		
b	a	F1

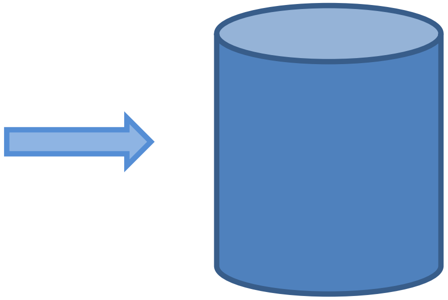
loadField		
b	F2	c

281

Introducing Fields

program

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

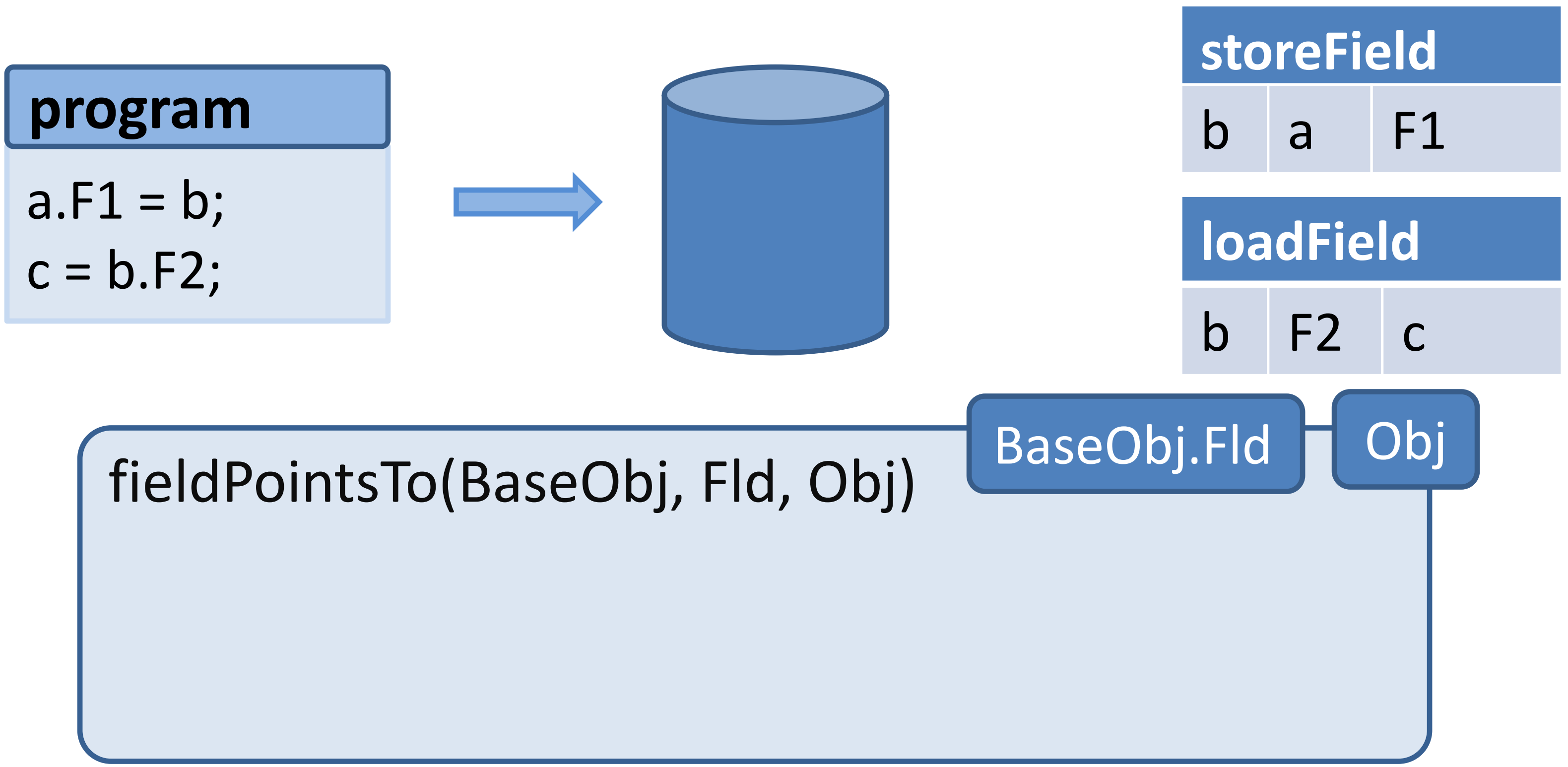


storeField		
b	a	F1

loadField		
b	F2	c

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)

BaseObj.Fld

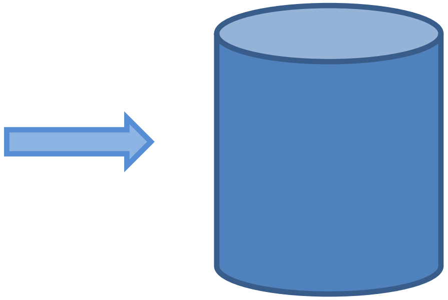
Obj

283

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),
```

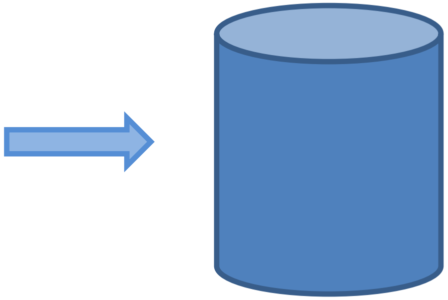
BaseObj.Fld

Obj

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

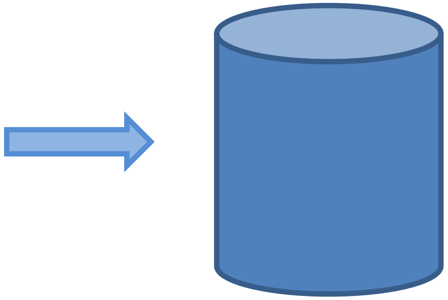
Obj

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

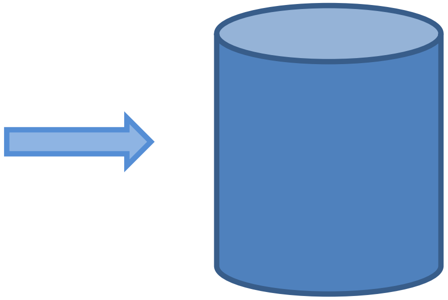
Base.Fld = From

286

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

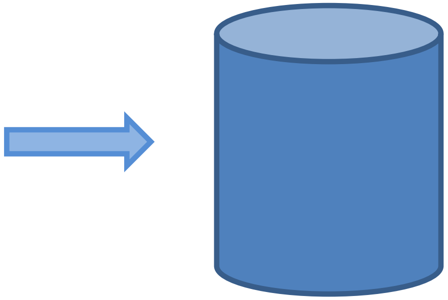
Obj

Base.Fld = From

Introducing Fields

program

```
a.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).
```

BaseObj.Fld

Obj

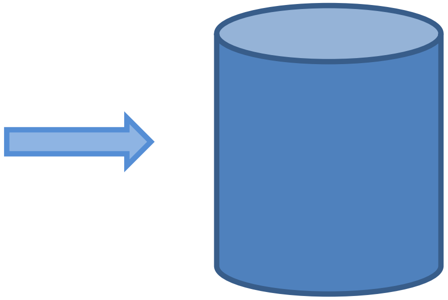
Base.Fld = From

288

Introducing Fields

program

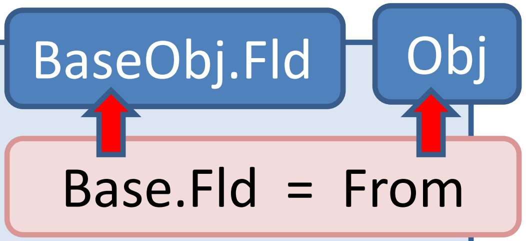
```
a.F1 = b;
c = b.F2;
```



storeField		
b	a	F1

loadField		
b	F2	c

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



BaseObj.Fld

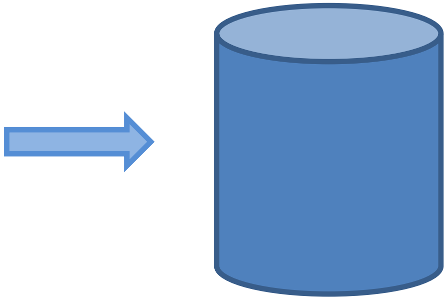
Obj

Base.Fld = From

Introducing Fields

program

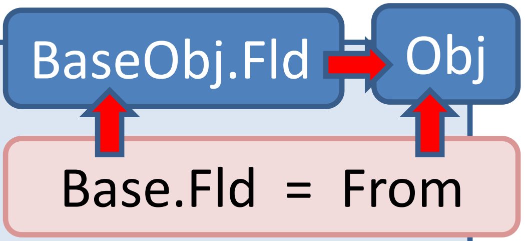
```
a.F1 = b;
c = b.F2;
```



storeField		
b	a	F1

loadField		
b	F2	c

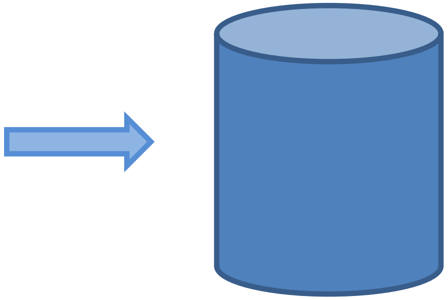
```
fieldPointsTo(BaseObj, Fld, Obj)
    <- storeField(From, Base, Fld),
        varPointsTo(Base, BaseObj),
        varPointsTo(From, Obj).
```



Introducing Fields

program

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



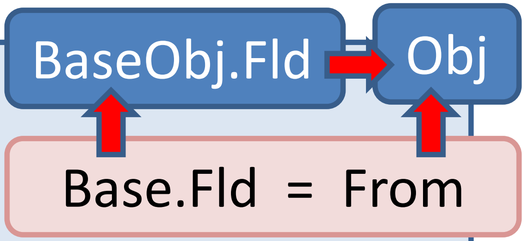
storeField

b	a	F1

loadField

b	F2	c

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



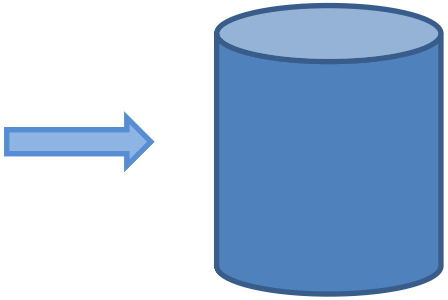
```
varPointsTo(To, Obj)
```

291

Introducing Fields

program

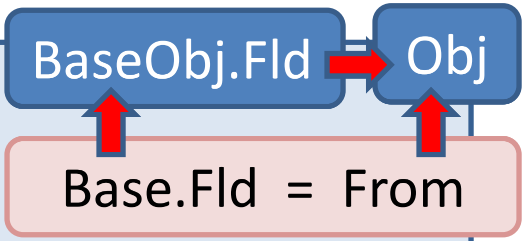
```
a.F1 = b;
c = b.F2;
```



storeField		
b	a	F1

loadField		
b	F2	c

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

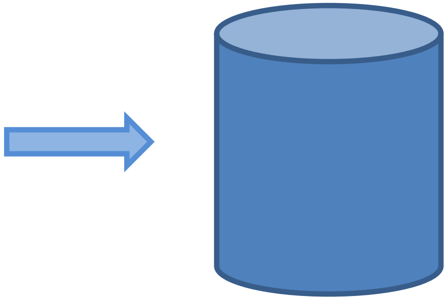


```
varPointsTo(To, Obj)
   <- loadField(Base, Fld, To),
```

Introducing Fields

program

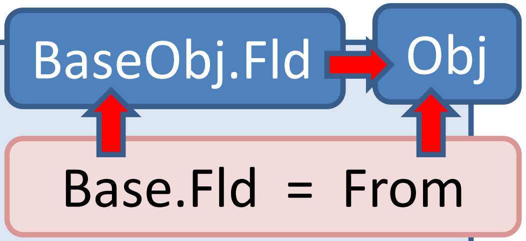
```
a.F1 = b;
c = b.F2;
```



storeField		
b	a	F1

loadField		
b	F2	c

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


BaseObj.Fld → Obj

Base.Fld = From

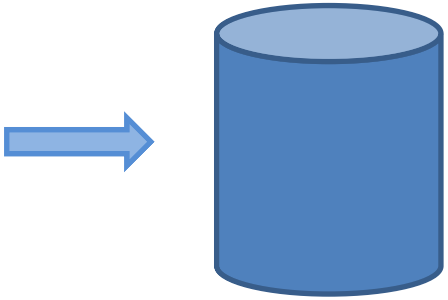
```
varPointsTo(To, Obj)
   <- loadField(Base, Fld, To),
```

To = Base.Fld

Introducing Fields

program

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



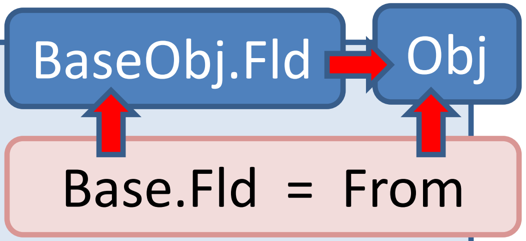
storeField

b	a	F1

loadField

b	F2	c

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



BaseObj.Fld

Obj

Base.Fld = From

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

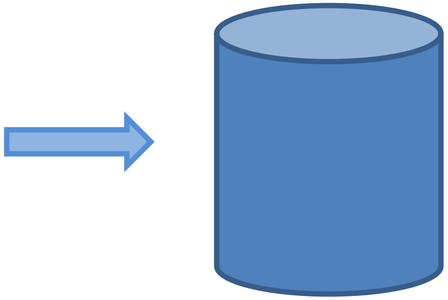
To = Base.Fld

294

Introducing Fields

program

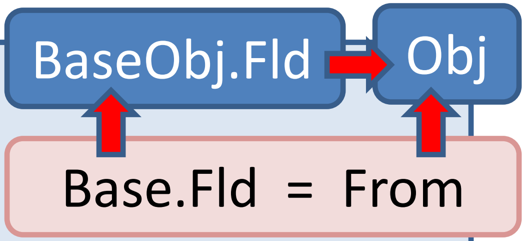
```
a.F1 = b;
c = b.F2;
```



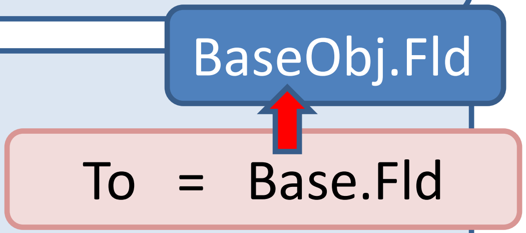
storeField		
b	a	F1

loadField		
b	F2	c

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



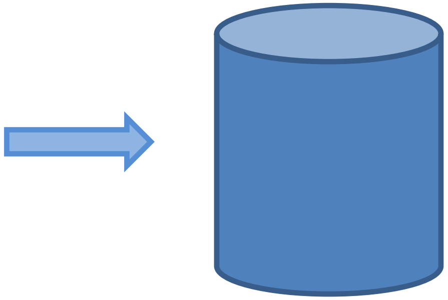
```
varPointsTo(To, Obj)
   <- loadField(Base, Fld, To),
      varPointsTo(Base, BaseObj),
```



Introducing Fields

program

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



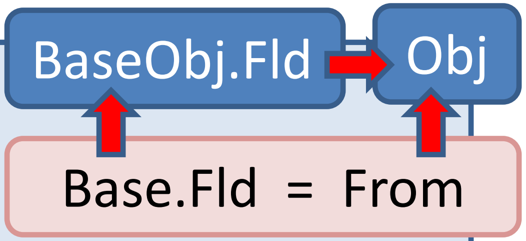
storeField

b	a	F1

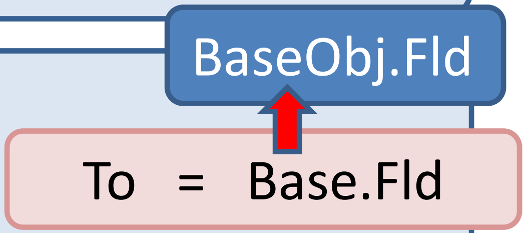
loadField

b	F2	c

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



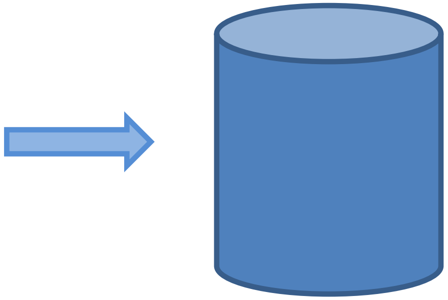
```
varPointsTo(To, Obj)
   <- loadField(Base, Fld, To),
      varPointsTo(Base, BaseObj),
      fieldPointsTo(BaseObj,  Fld, Obj).
```



Introducing Fields

program

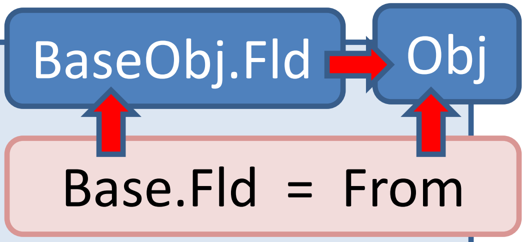
```
a.F1 = b;
c = b.F2;
```



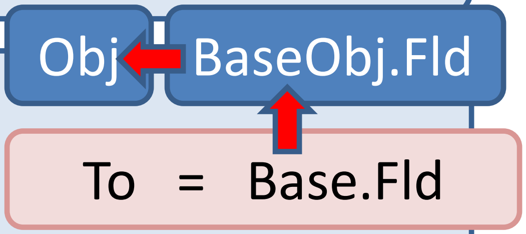
storeField		
b	a	F1

loadField		
b	F2	c

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



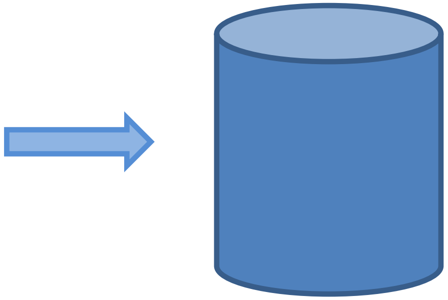
```
varPointsTo(To, Obj)
    <- loadField(Base, Fld, To),
        varPointsTo(Base, BaseObj),
        fieldPointsTo(BaseObj,  Fld, Obj).
```



Introducing Fields

program

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



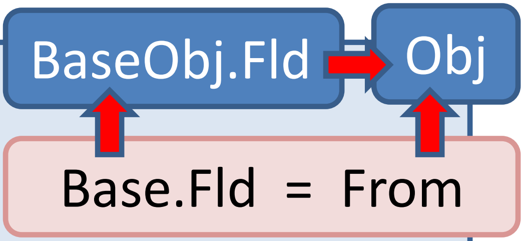
storeField

b	a	F1

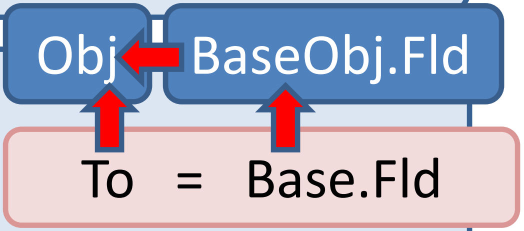
loadField

b	F2	c

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



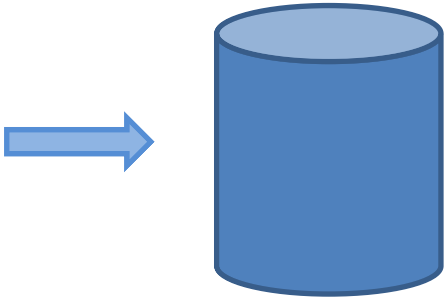
```
varPointsTo(To, Obj)
   <- loadField(Base, Fld, To),
      varPointsTo(Base, BaseObj),
      fieldPointsTo(BaseObj,  Fld, Obj).
```



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).
```

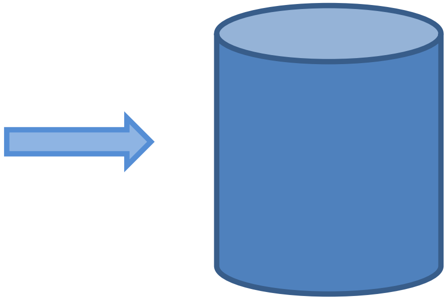
```
varPointsTo(To, Obj)
   <- loadField(Base, Fld, To),
       varPointsTo(Base, BaseObj),
       fieldPointsTo(BaseObj,  Fld, Obj).
```

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).
```

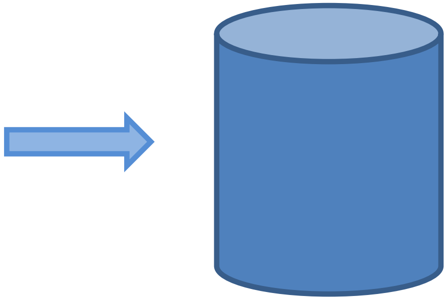
```
varPointsTo(To, Obj)
   <- loadField(Base, Fld, To),
      varPointsTo(Base, BaseObj),
      fieldPointsTo(BaseObj,  Fld, Obj).
```

Enhance specification without changing base code

Introducing Fields

program

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



storeField		
b	a	F1

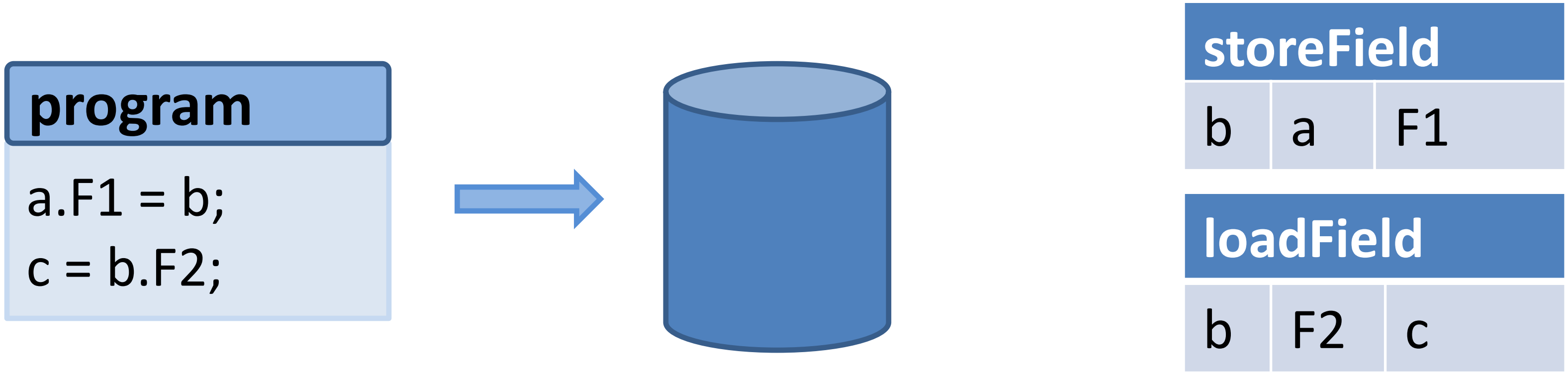
loadField		
b	F2	c

```
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;
```

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

Specification + Implementation

Specifications

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, …).
```

Specification + Implementation

Specifications

Implementation

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

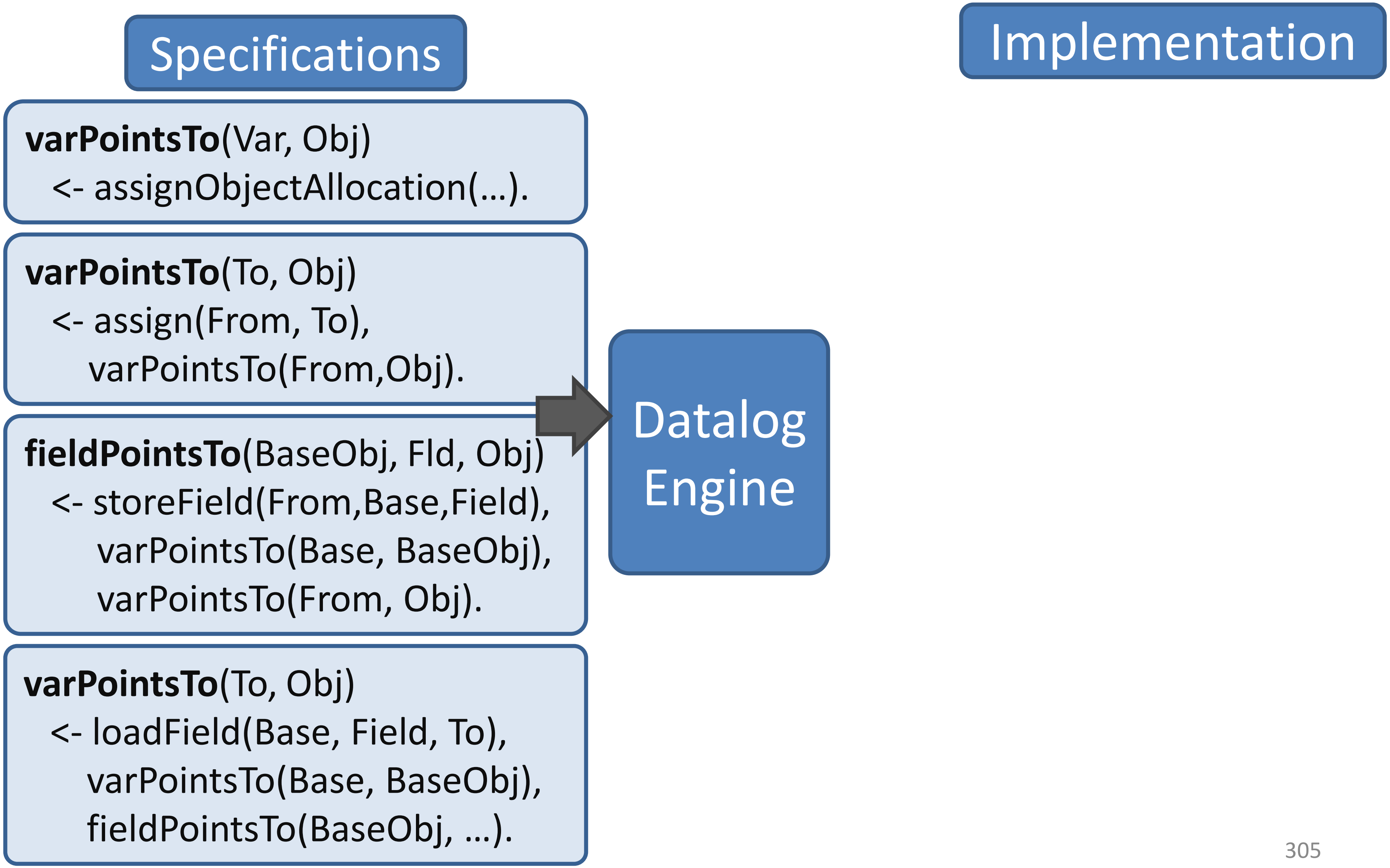
```
varPointsTo(To, Obj)
  <- assign(From, To),
     varPointsTo(From,Obj).
```

```
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

Specification + Implementation



Specifications

Implementation

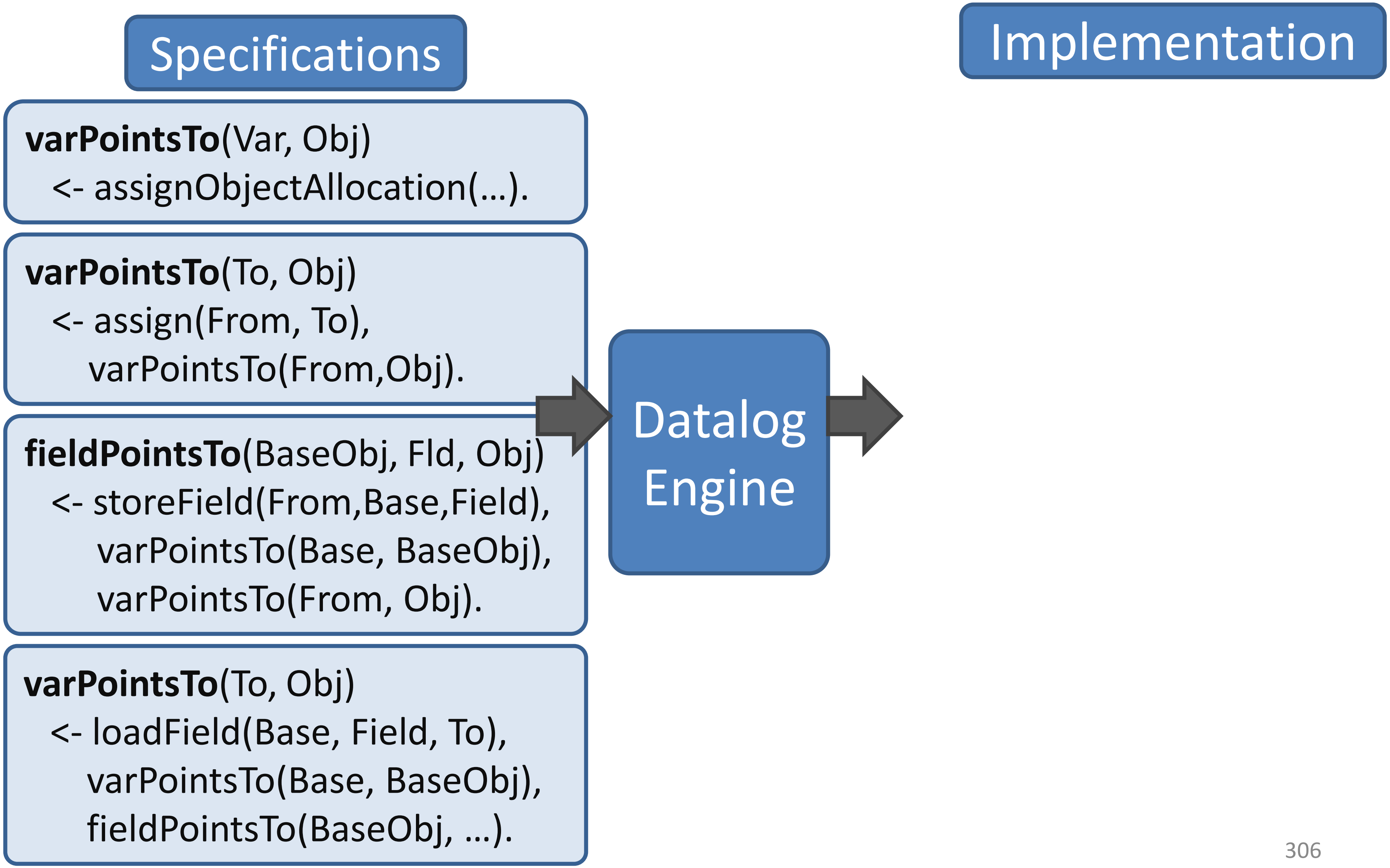
```
varPointsTo(Var, Obj)
  <- assignObjectAllocation(…).
```

```
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, …).
```

Specification + Implementation



Specifications

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

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

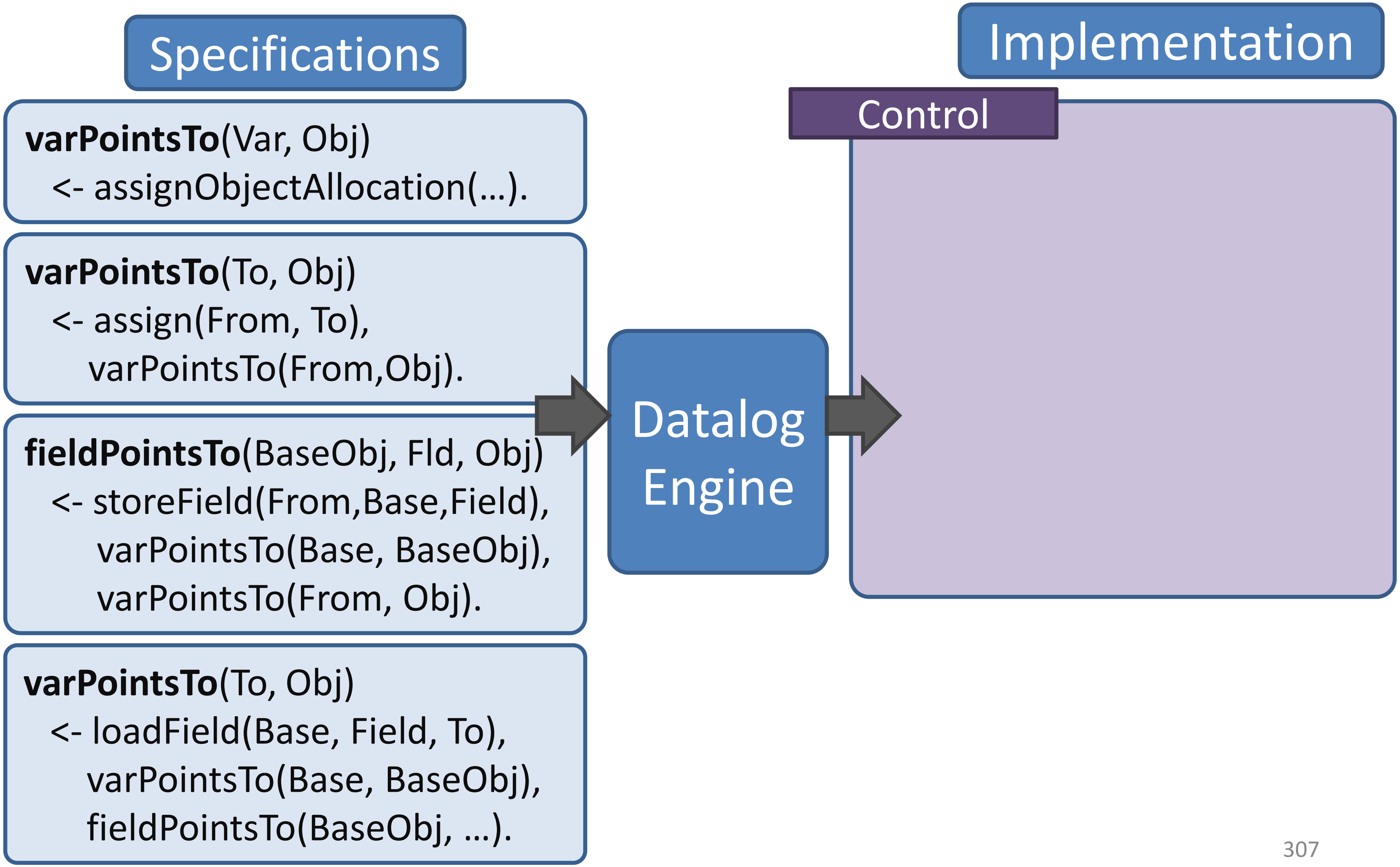
```
fieldPointsTo(BaseObj, Fld, Obj)
  <- storeField(From,Base,Field),
      varPointsTo(Base, BaseObj),
      varPointsTo(From, Obj).
```

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

Datalog Engine

Implementation

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).
```

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

Datalog Engine

Implementation

Control

Specification + Implementation

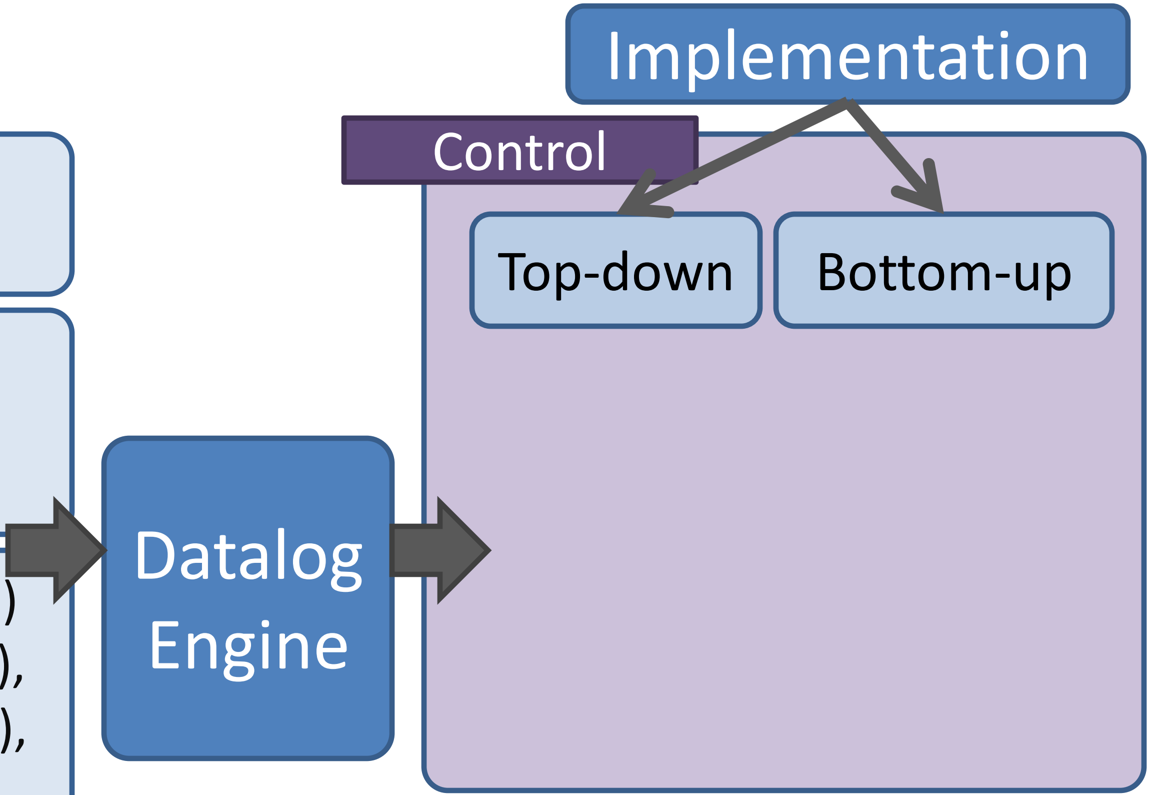
Specifications

```
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).

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



Implementation

Control

Top-down

Bottom-up

Datalog Engine

Specification + Implementation

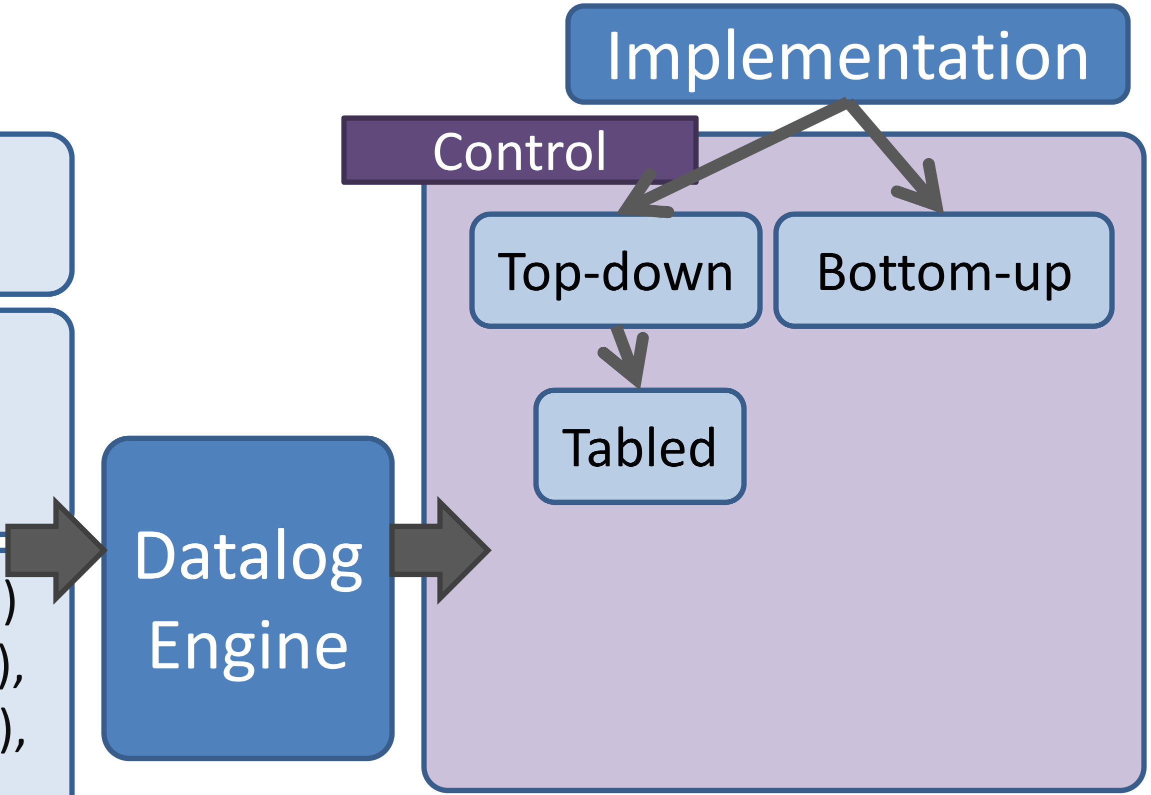
Specifications

```
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).
```

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



Specification + Implementation

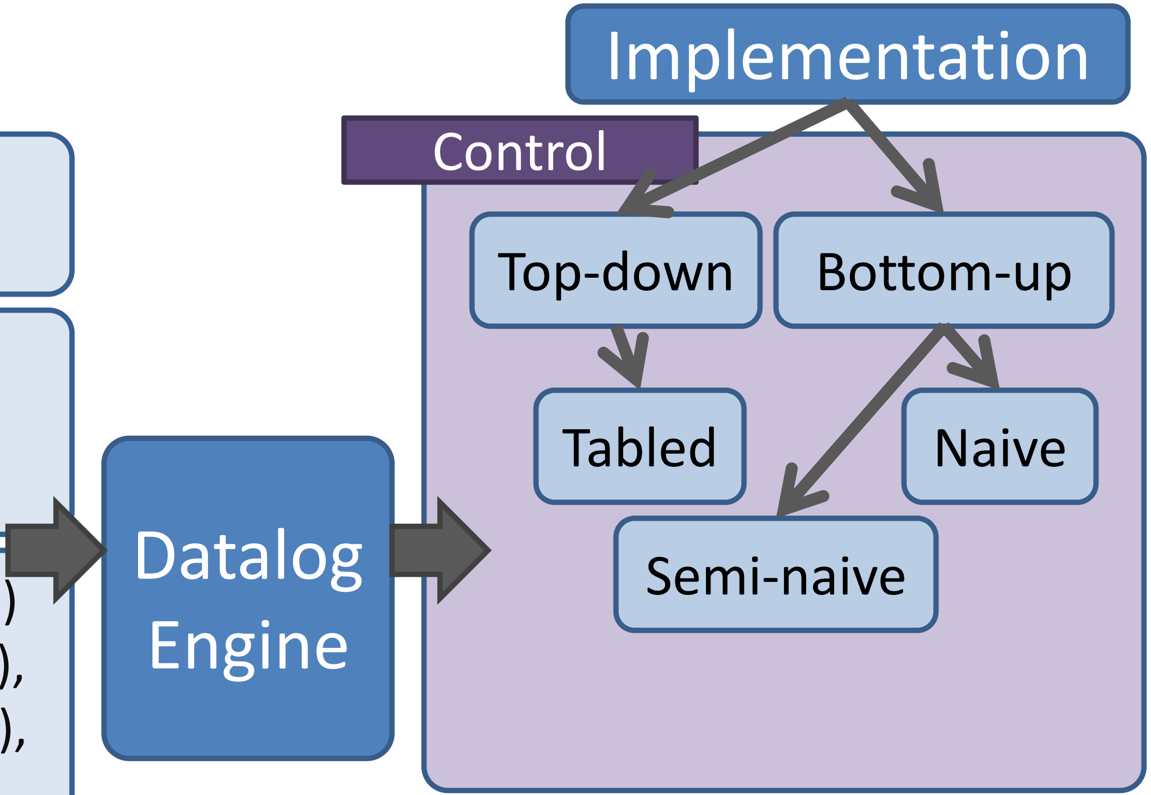
Specifications

```
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).
```

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



Specification + Implementation

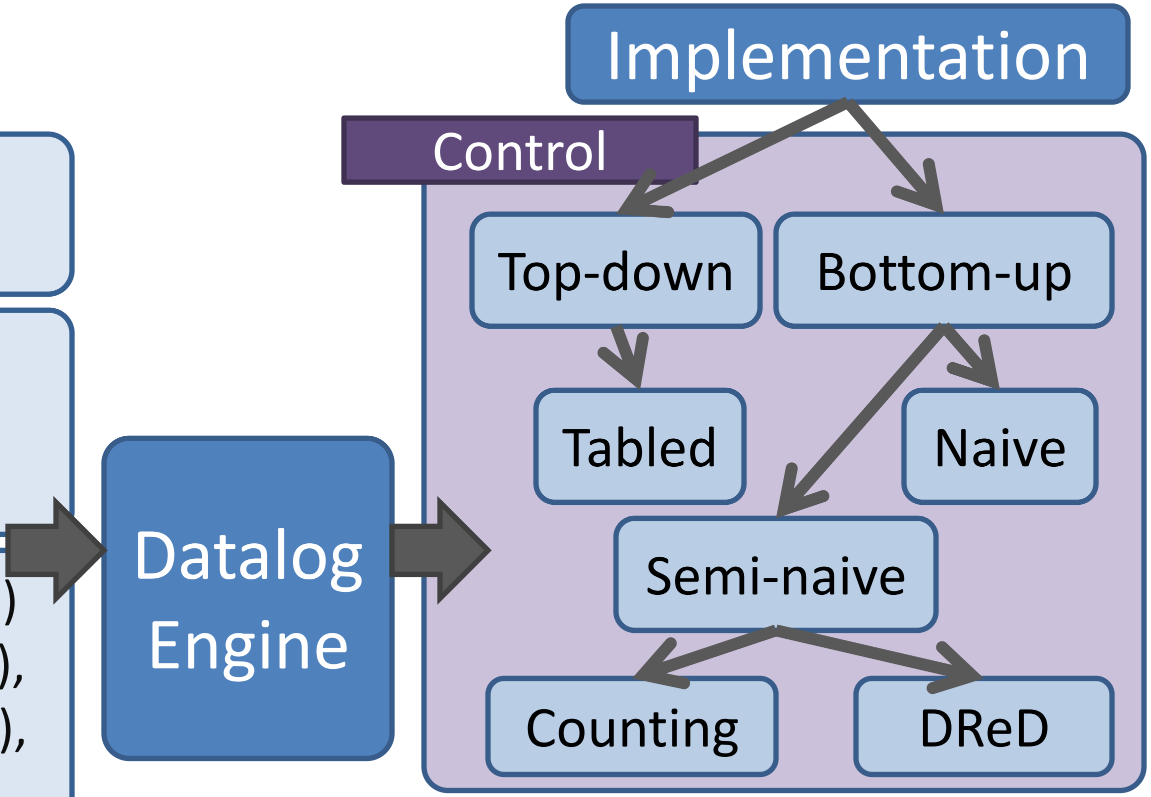
Specifications

```
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).
```

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



Datalog Engine

Implementation

Control

Top-down

Bottom-up

Tabled

Naive

Semi-naive

Counting

DReD

Specification + Implementation

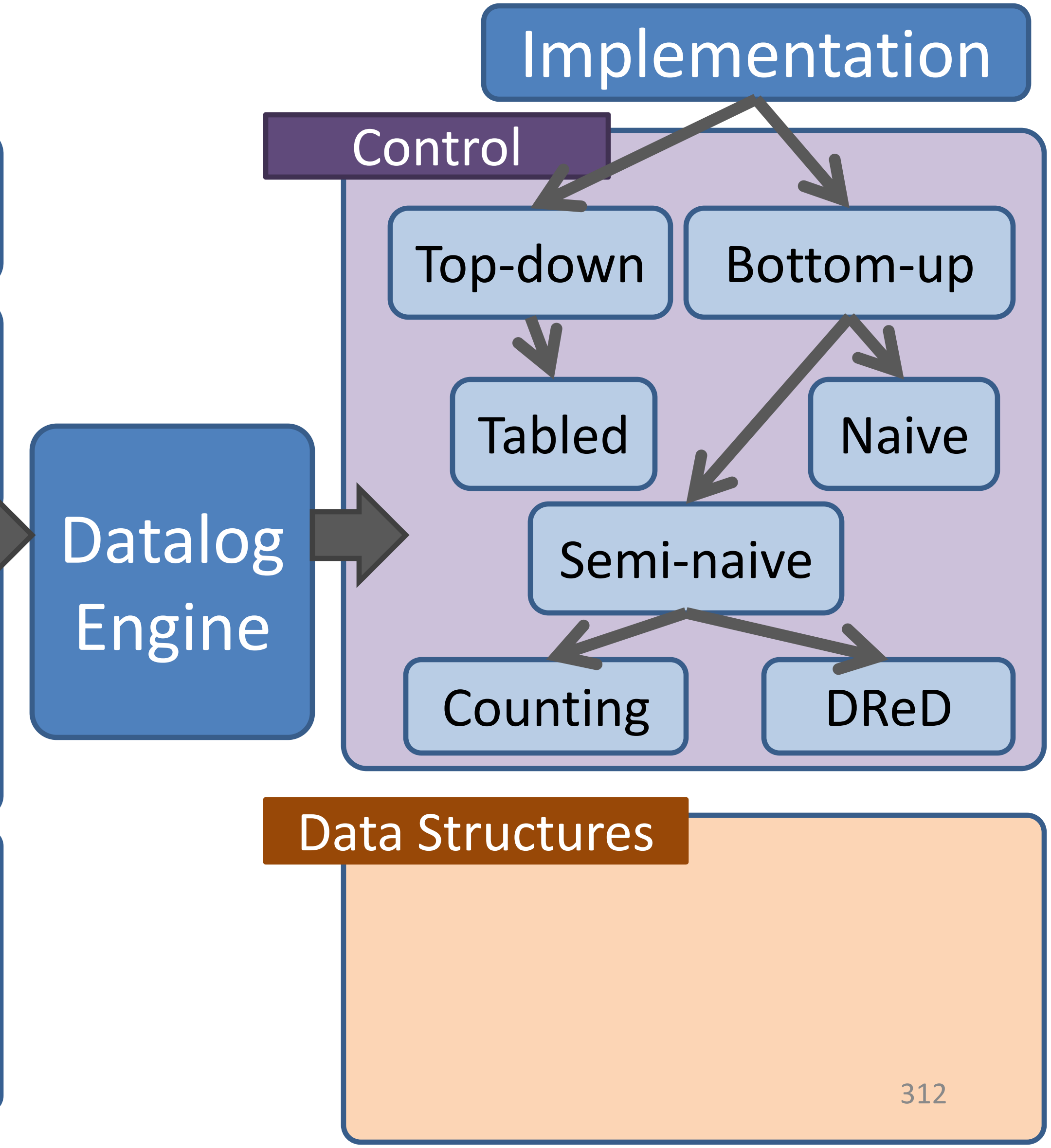
Specifications

```
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).
```

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



Specification + Implementation

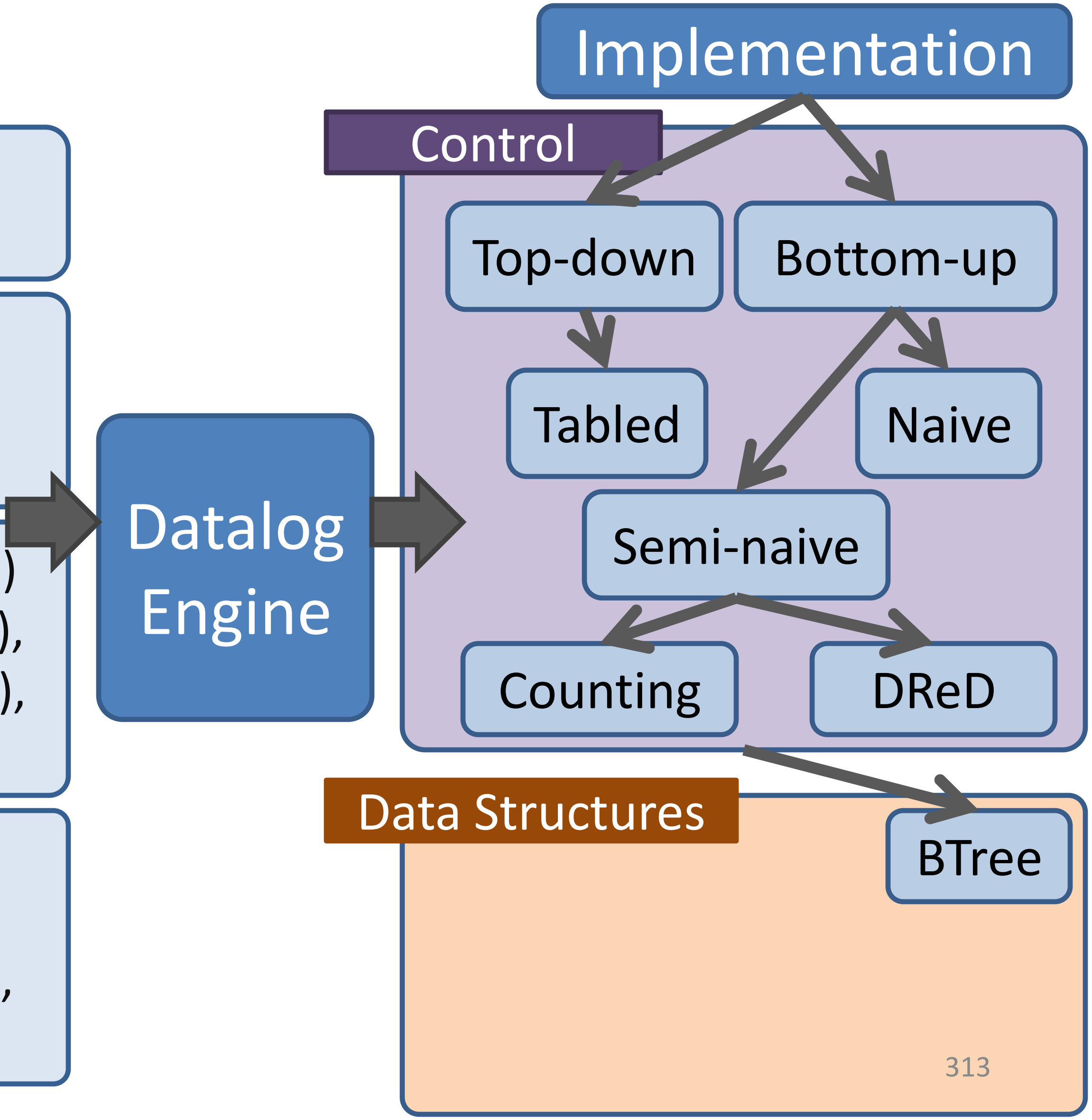
Specifications

```
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).
```

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



Specification + Implementation

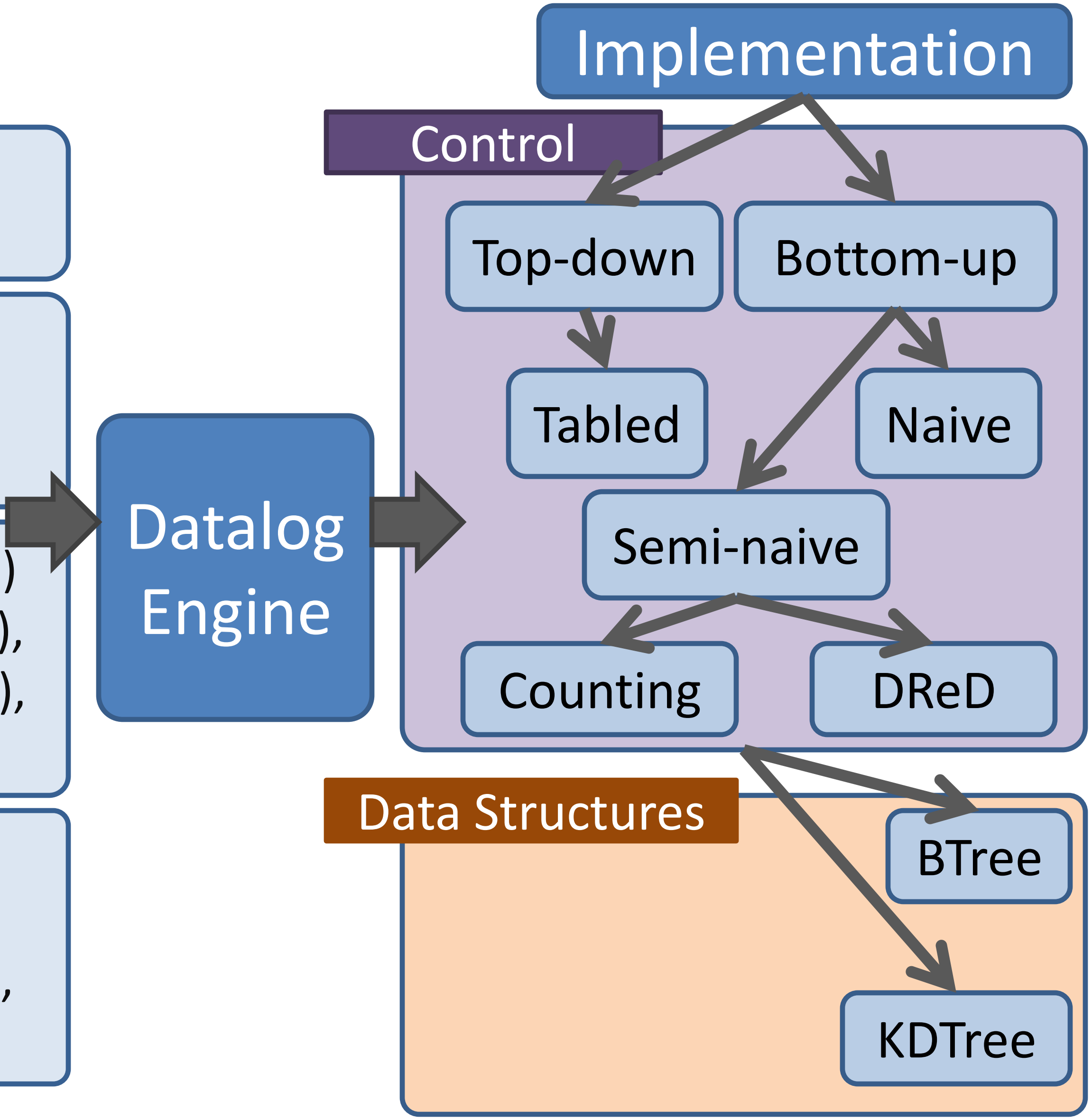
Specifications

```
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).
```

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



Specification + Implementation

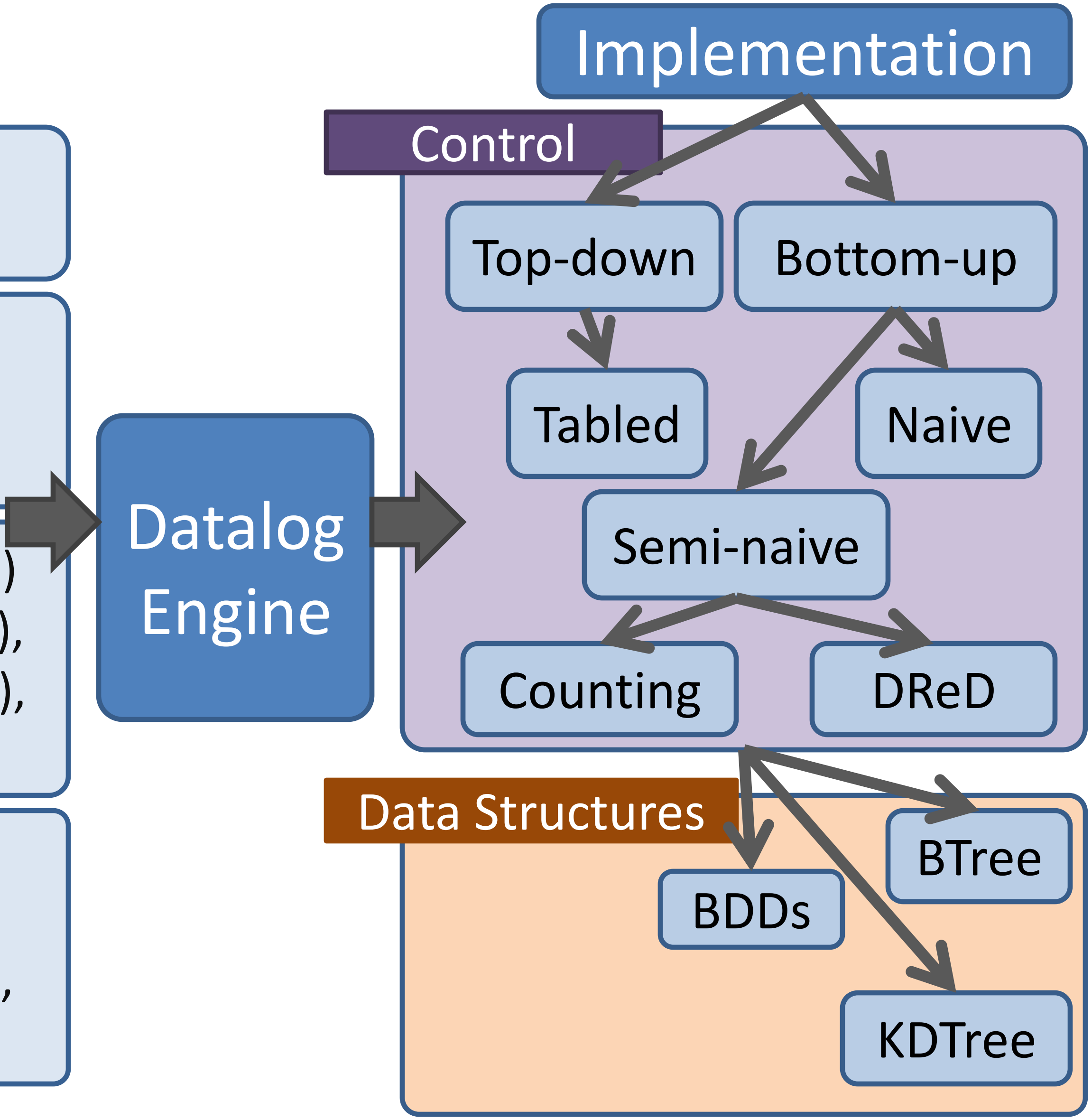
Specifications

```
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).
```

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



Specification + Implementation

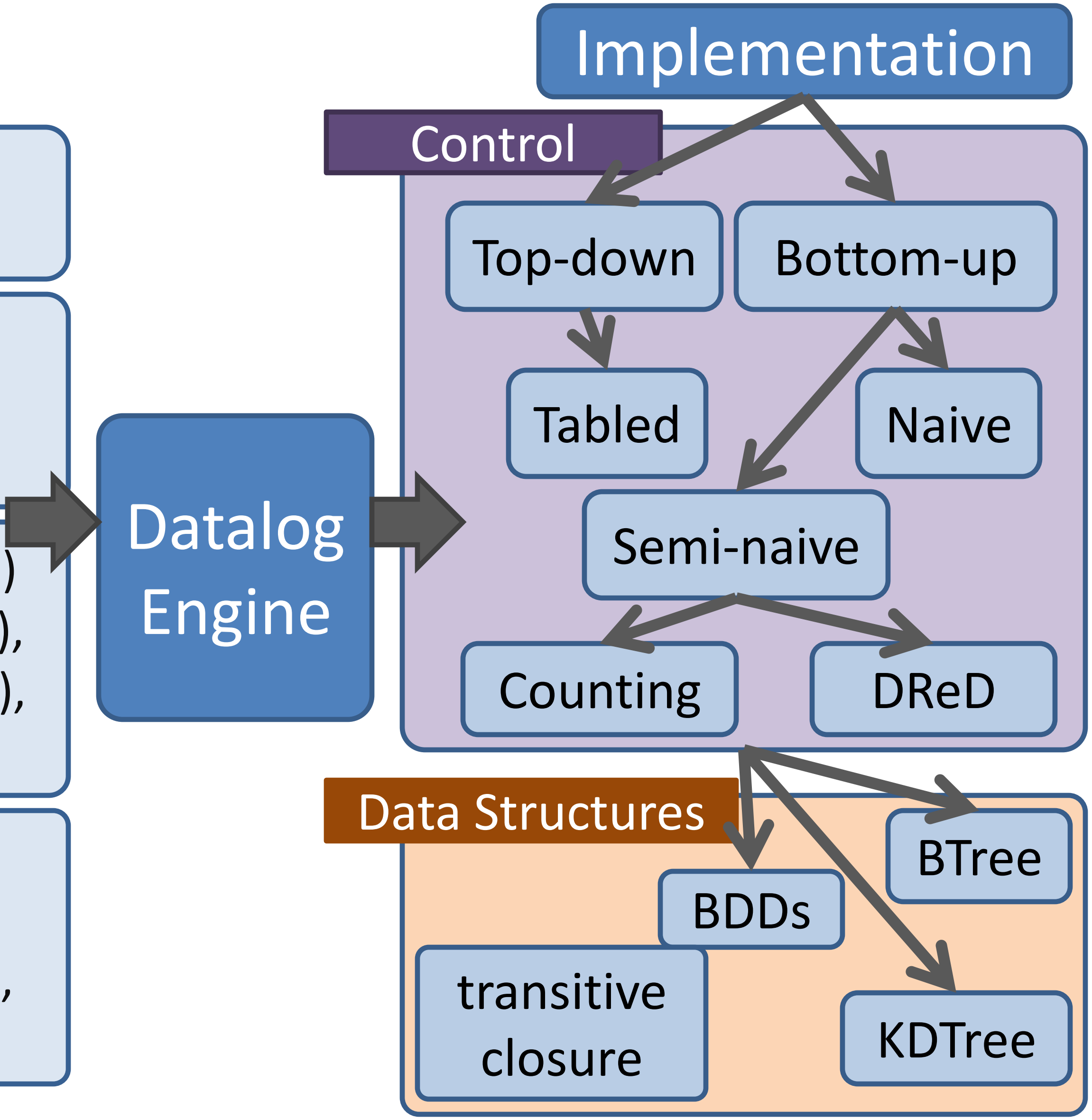
Specifications

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

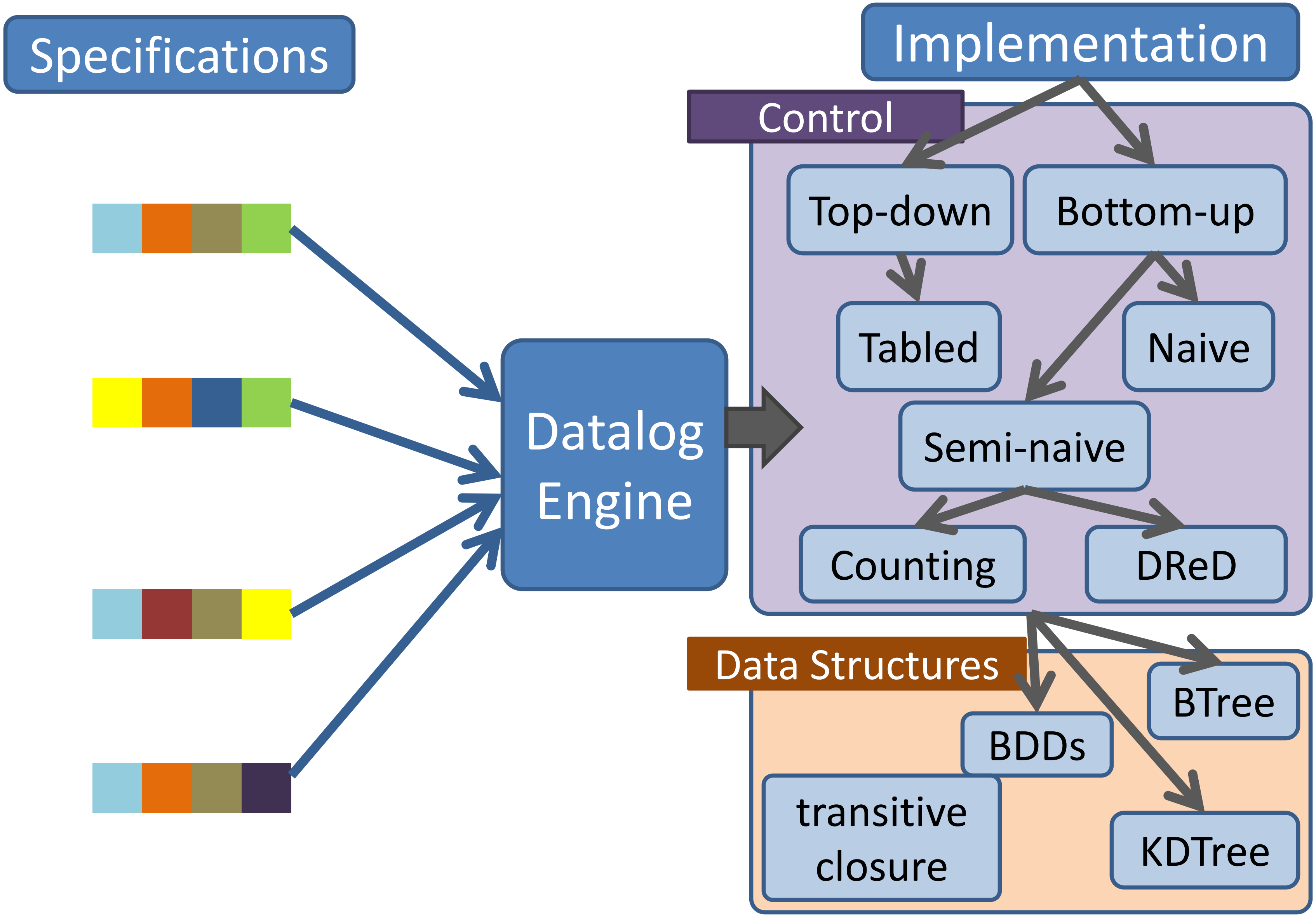
```
varPointsTo(To, Obj)
  <- assign(From, To),
     varPointsTo(From,Obj).
```

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

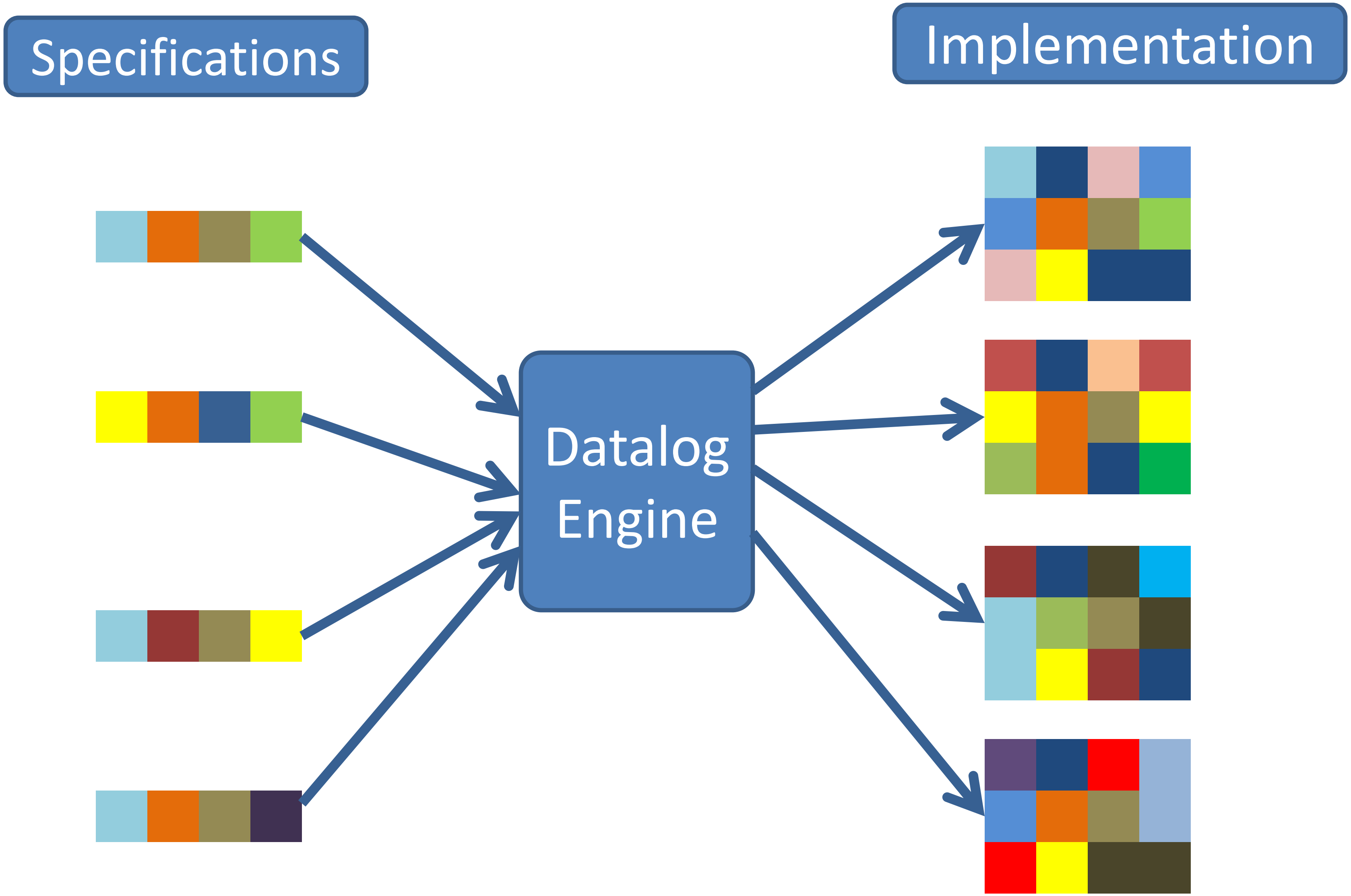
```
varPointsTo(To, Obj)
  <- loadField(Base, Field, To),
     varPointsTo(Base, BaseObj),
     fieldPointsTo(BaseObj, …).
```



Specification + Implementation



Specification + Implementation

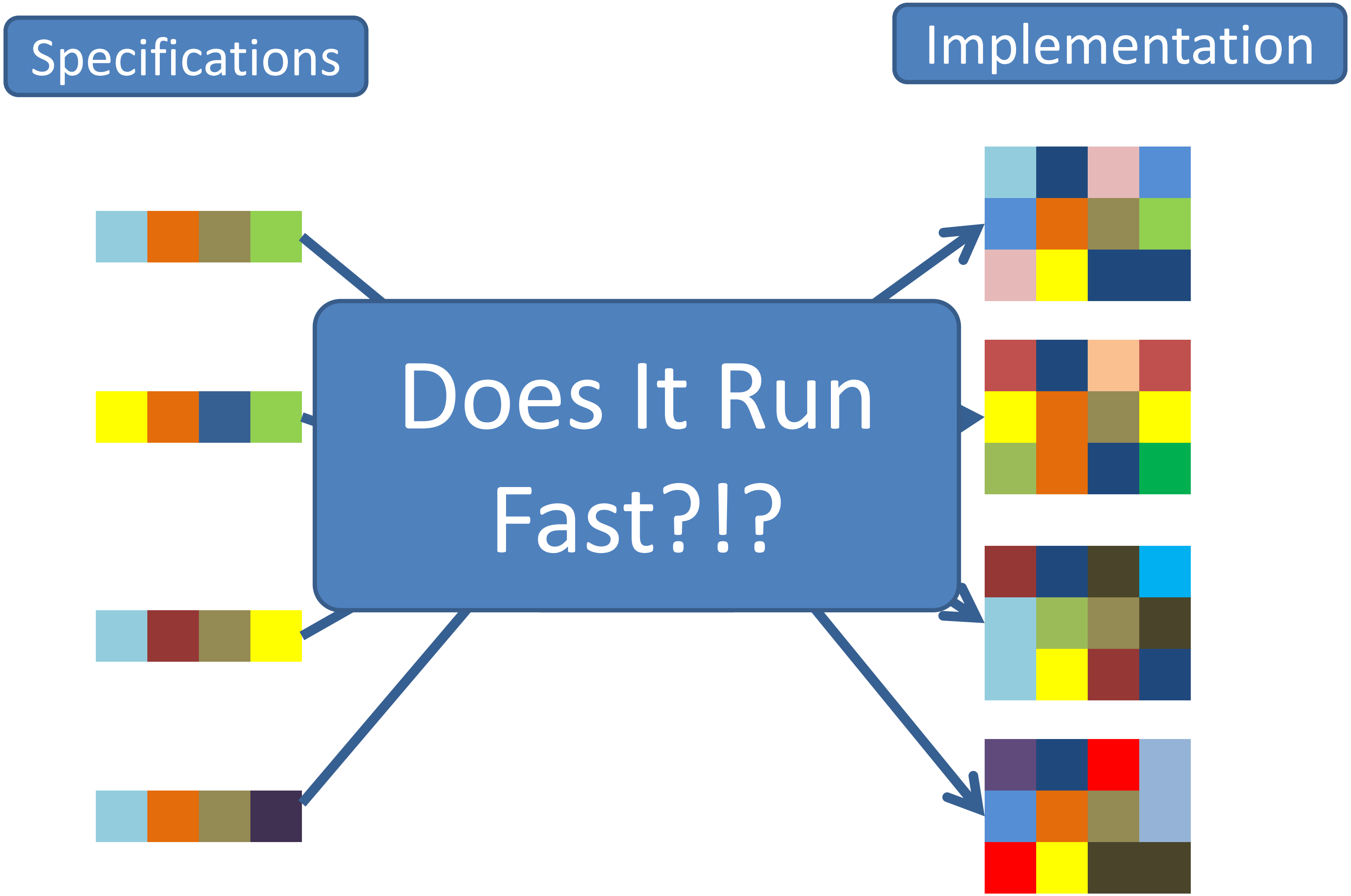


Specifications

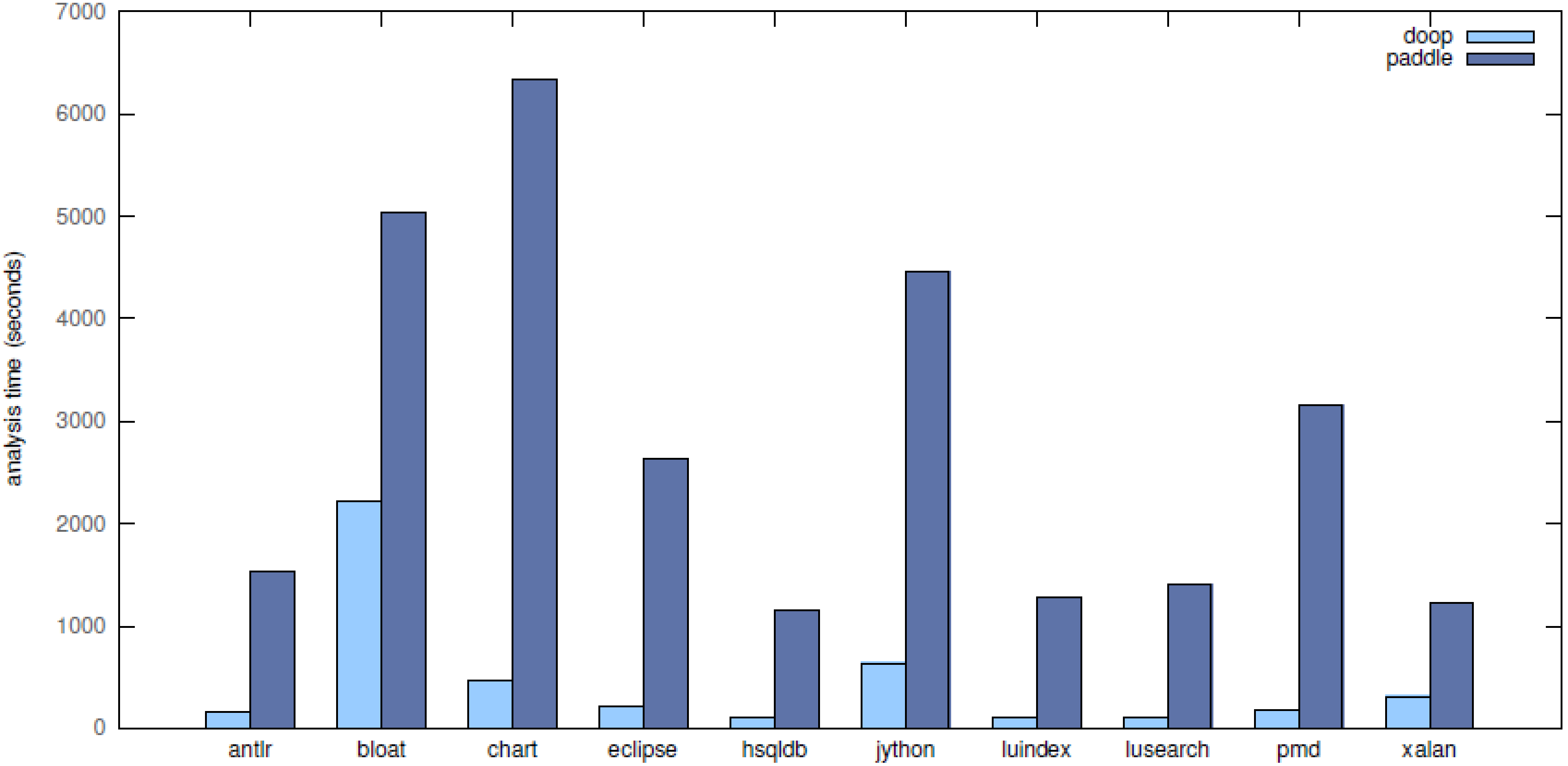
Implementation

Datalog Engine

Specification + Implementation



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

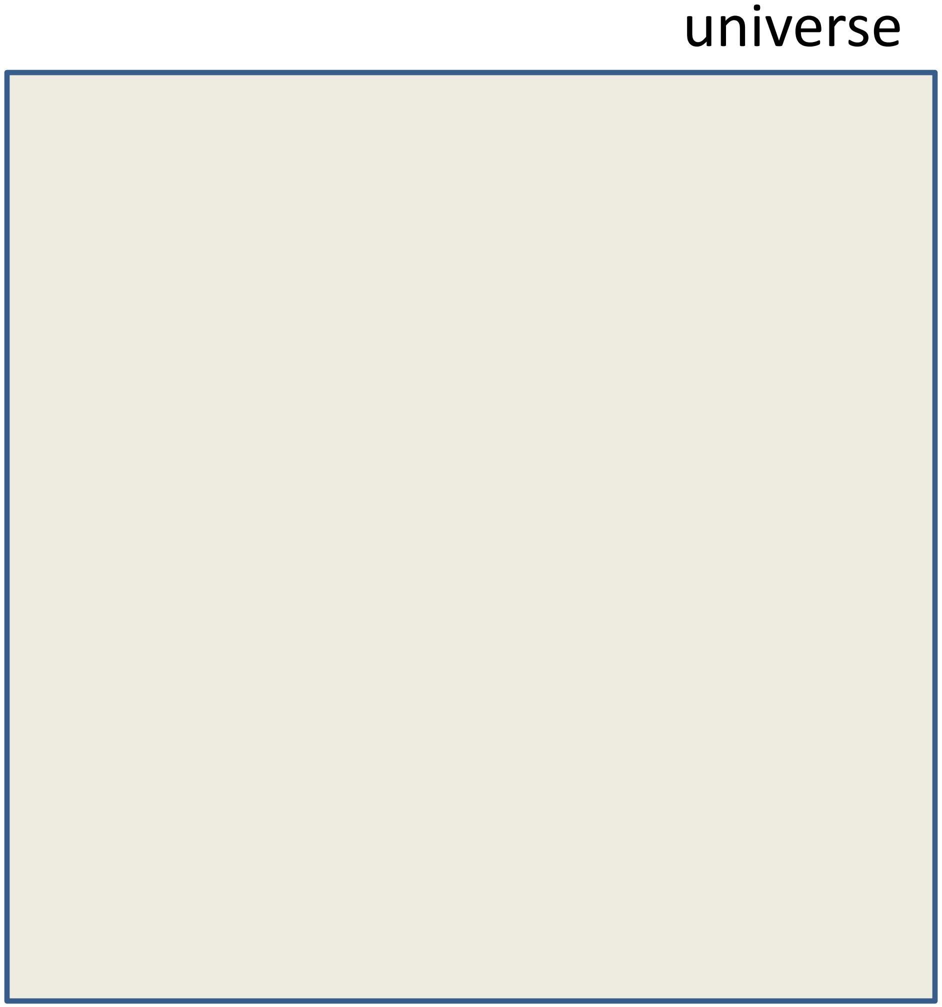
- something old
  - semi-naïve evaluation, folding, index selection
- something new(-ish)
  - magic-sets
- something borrowed (from PL)
  - type-based

Crucial Optimizations

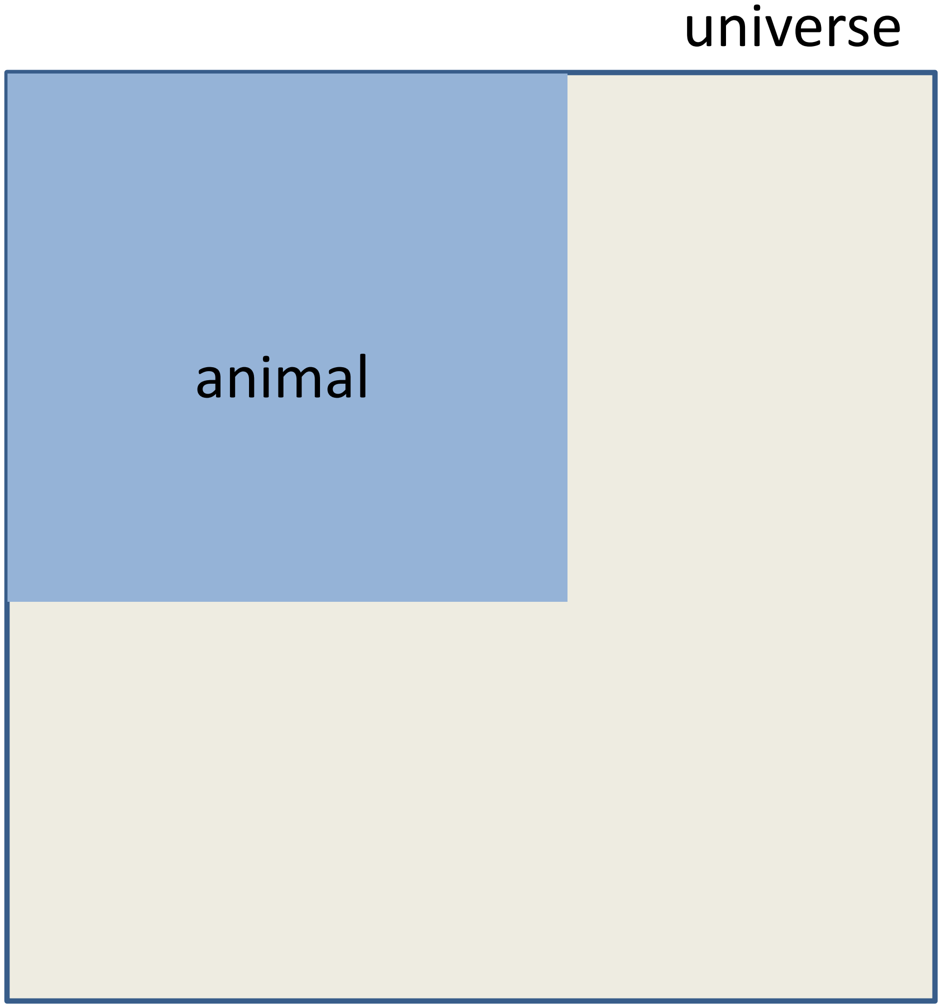
- something old
  - semi-naïve evaluation, folding, index selection
- something new(-ish)
  - magic-sets
- something borrowed (from PL)
  - type-based

TYPE-BASED OPTIMIZATIONS

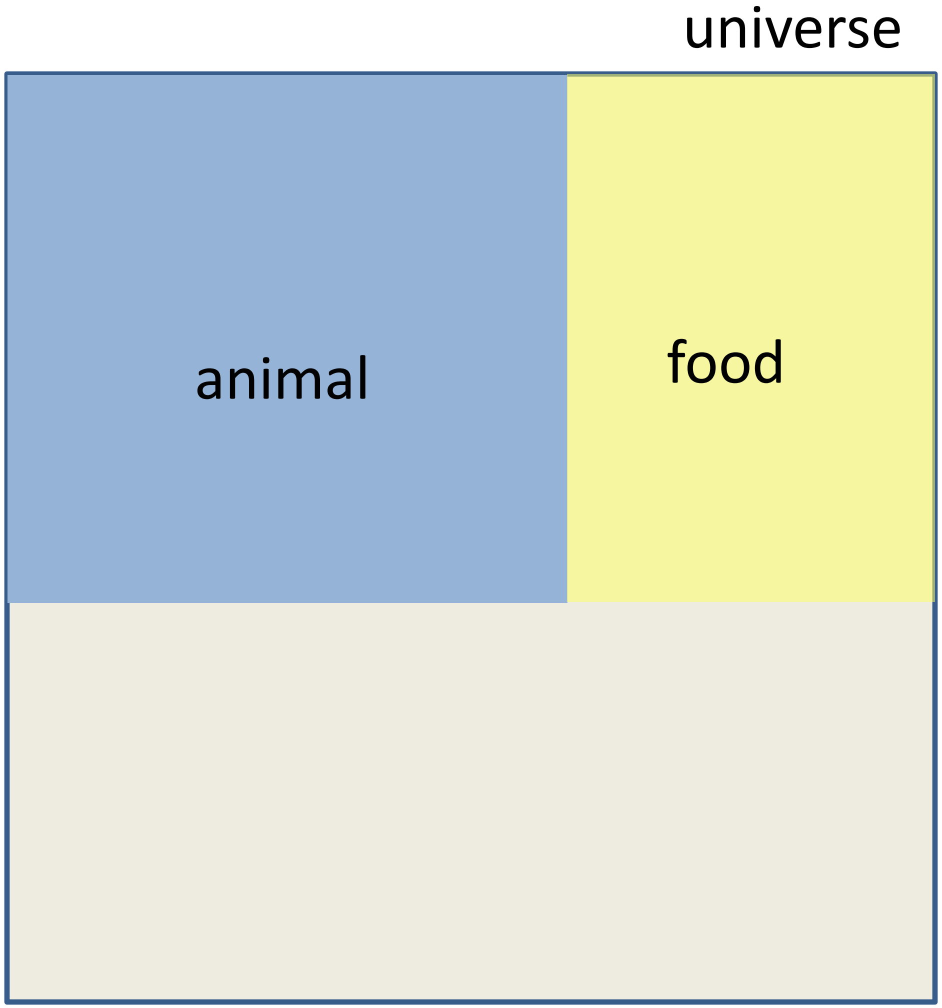
Types: Sets of Values



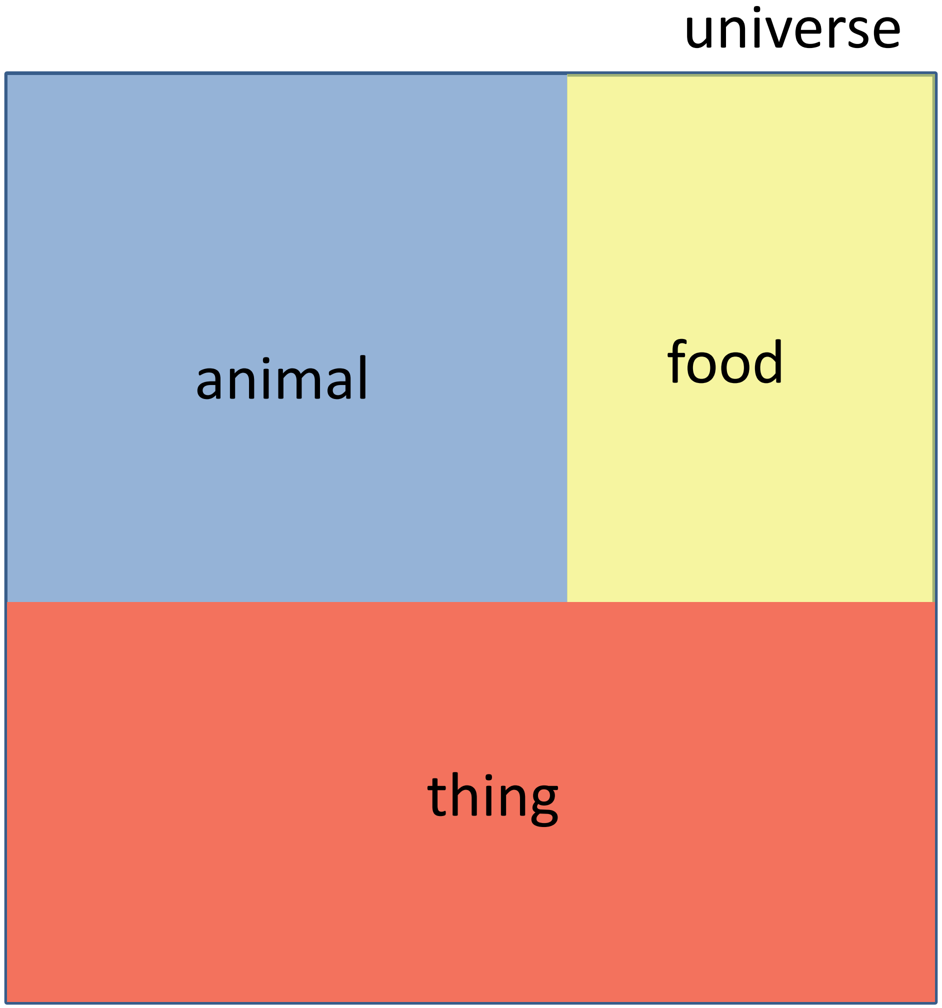
Types: Sets of Values



Types: Sets of Values

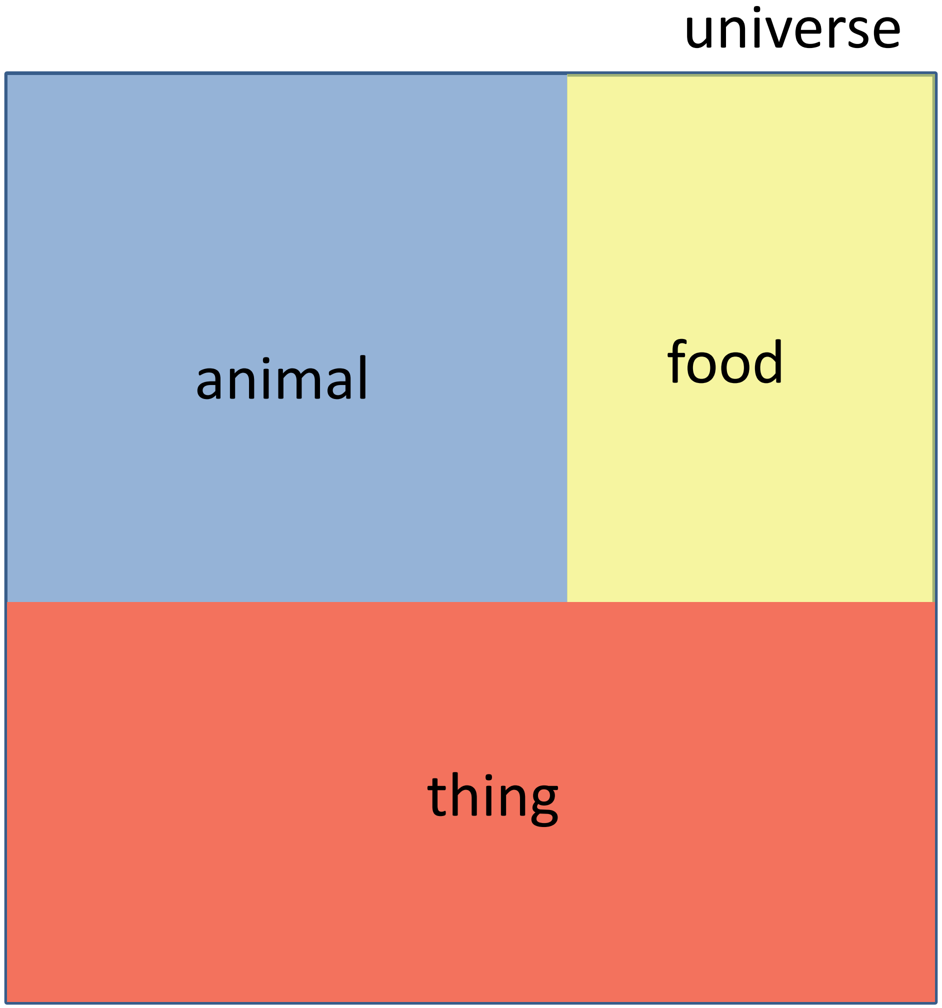


Types: Sets of Values



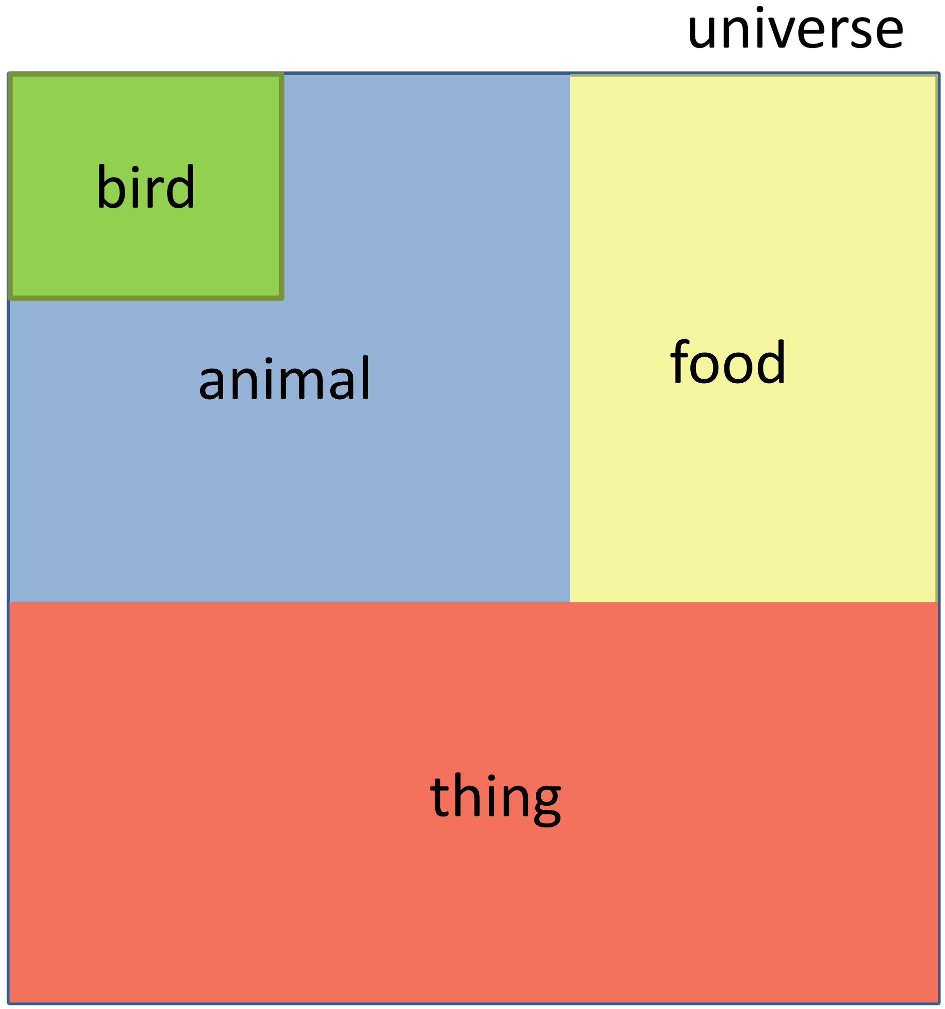
Types: Sets of Values

```
animal(X)  ->  .
```



Types: Sets of Values

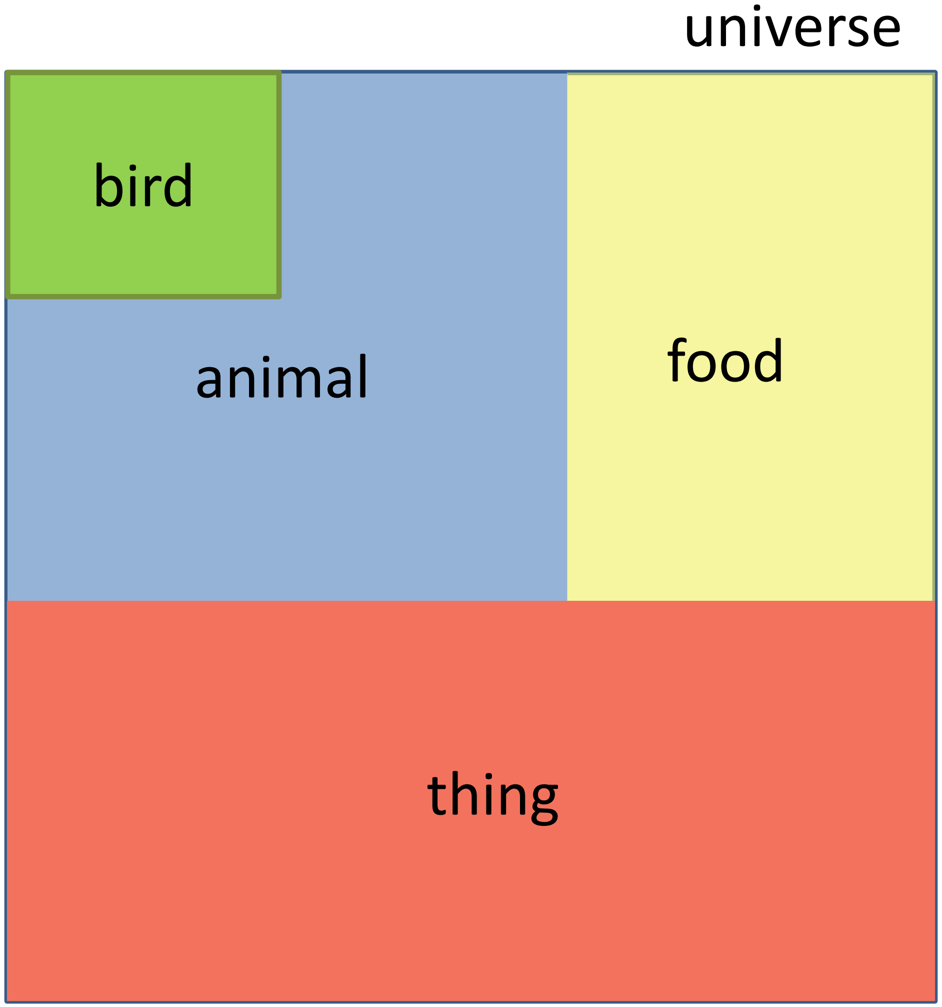
```
animal(X)  ->  .
```



Types: Sets of Values

```
animal(X)  ->  .
```

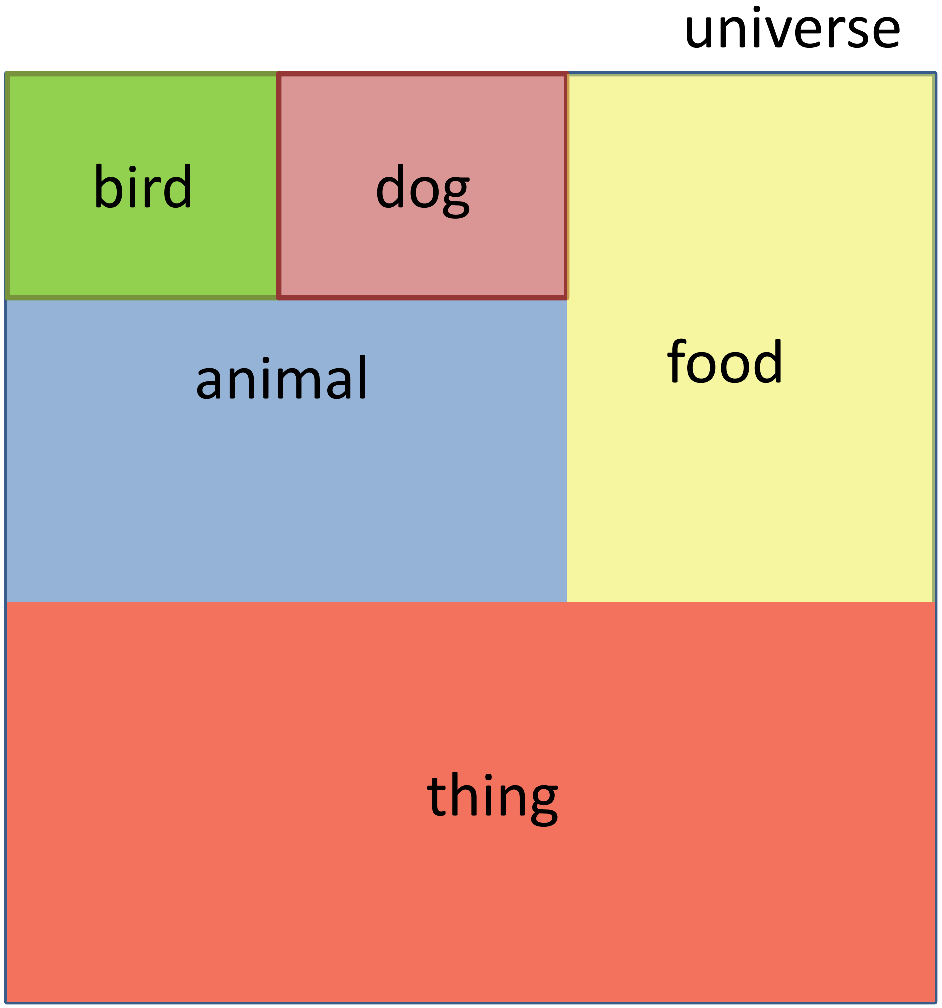
```
bird(X) -> animal(X) .
```



Types: Sets of Values

```
animal(X)  ->  .
```

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

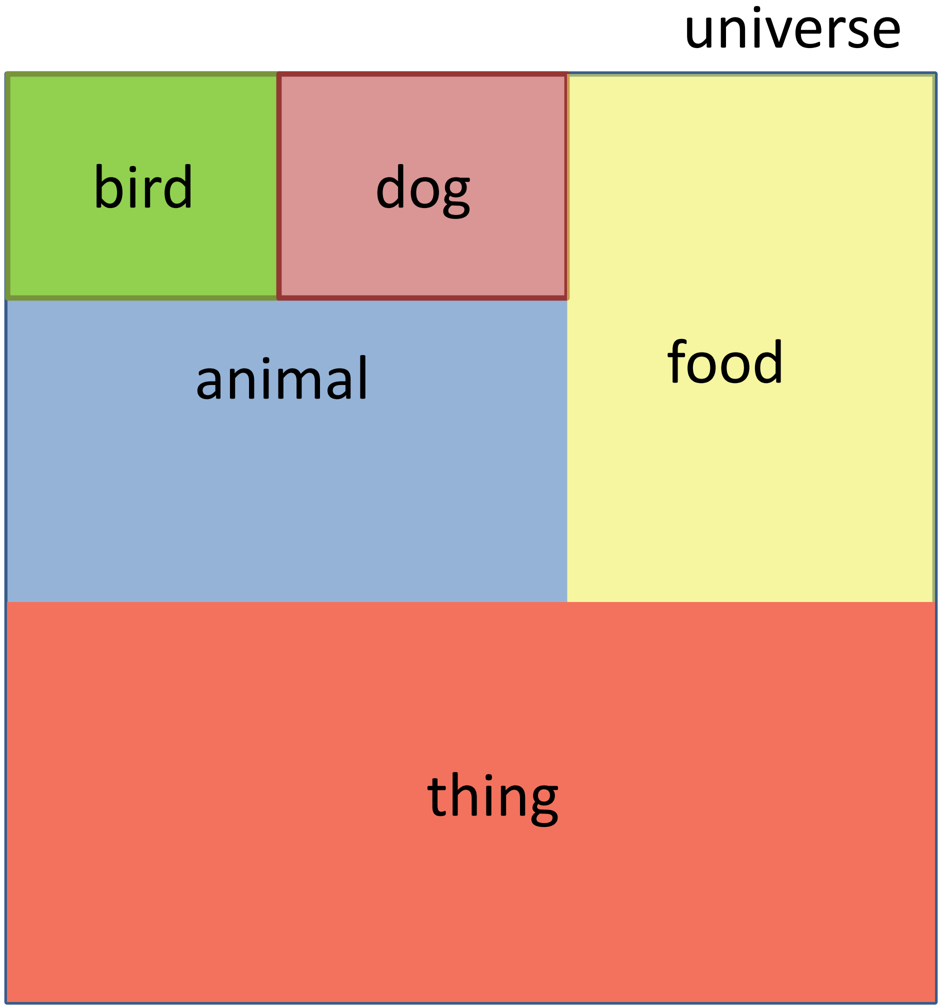


Types: Sets of Values

```
animal(X)  ->  .
```

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

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



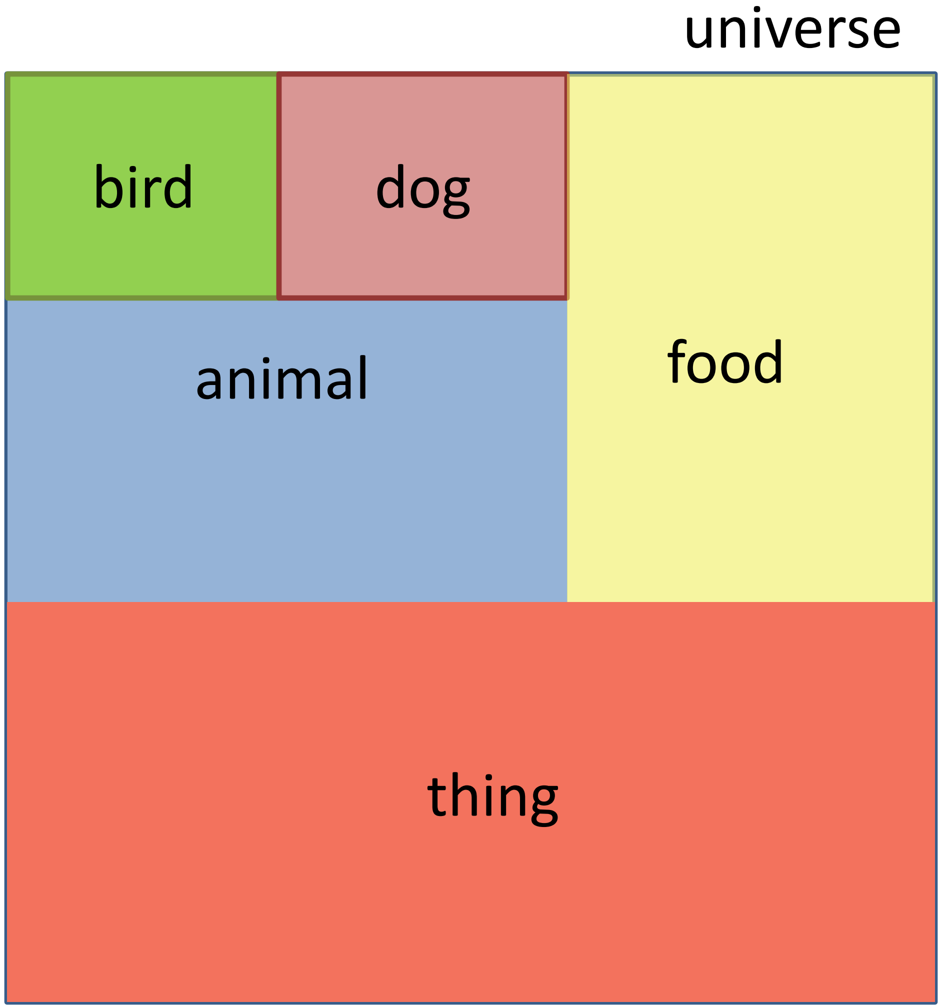
Types: Sets of Values

```
animal(X)  ->  .
```

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

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

```
dog(X) -> !bird(X).
bird(X) -> !dog(X).
```



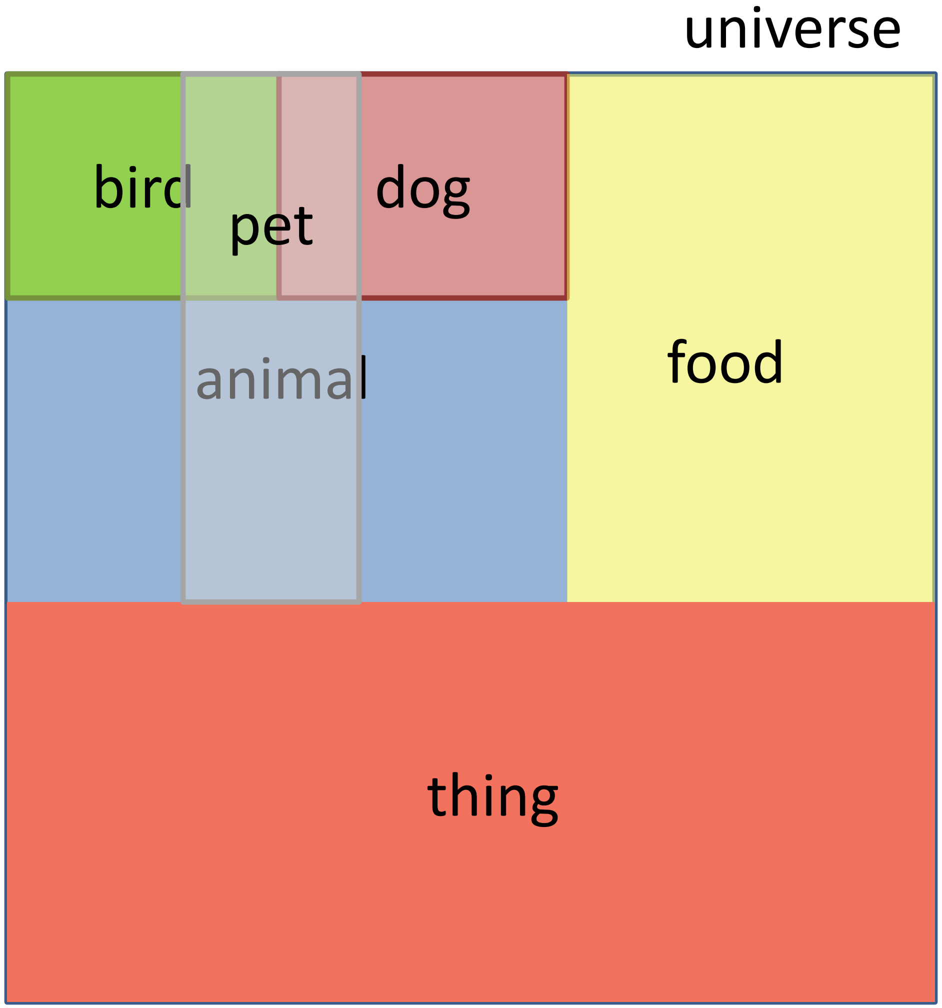
Types: Sets of Values

```
animal(X)  ->  .
```

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

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

```
dog(X) -> !bird(X).
bird(X) -> !dog(X).
```



Types: Sets of Values

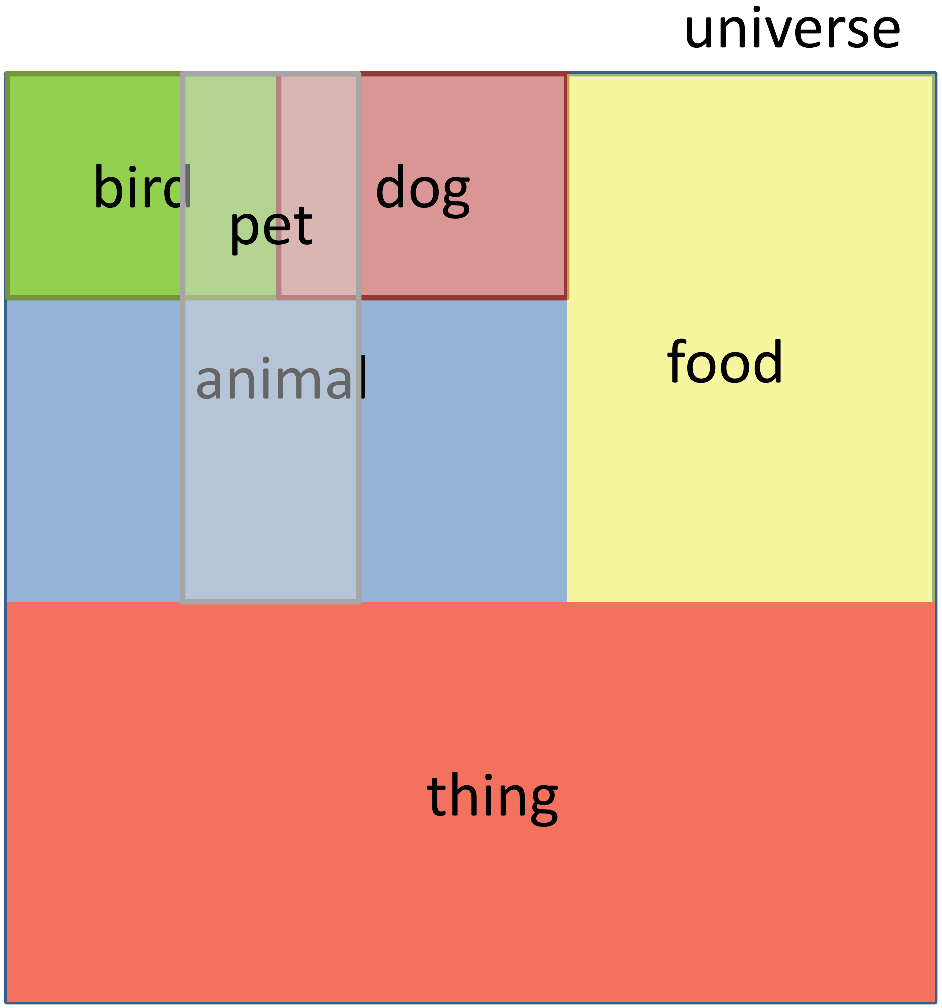
```
animal(X)  ->  .
```

```
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).
```

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

D :: dog

“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

“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).
```

dogChews :: (dog, 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).
```

dogChews :: (dog, food)

birdSwallows :: (bird, 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).~~
```

dogChews :: (dog, food)

birdSwallows :: (bird, food)

Type Erasure

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

D :: dog

```
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).
```

D :: dog

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

eat :: (dog, food)

Type Erasure

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

D :: dog

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

eat :: (dog, food)

Type Erasure

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

D :: dog

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

eat :: (dog, food)

Type Erasure

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

D :: dog

Thing :: chocolate

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

eat :: (dog, food)

Type Erasure

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

D :: dog

Thing :: chocolate

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

eat :: (dog, food)

Clean Up

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

D :: dog

Thing :: chocolate

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

eat :: (dog, food)

Clean Up

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

D :: dog

Thing :: chocolate

```
eat(A, Food)
   <-   dogChews(A,Food).
```

eat :: (dog, 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

- BDDBDDB
  - 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

functional
understanding program behavior
^

Program Analysis

logic

understanding program behavior

^

Program Analysis

Datalog

understanding program behavior

^

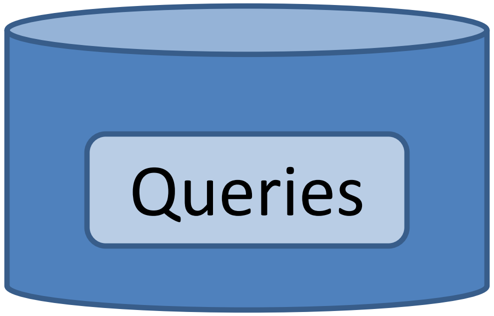
Program Analysis

Datalog
understanding program behavior
^

- ***"Evita Raced: Meta-compilation for declarative networks"***, Condie et al., VLDB '08

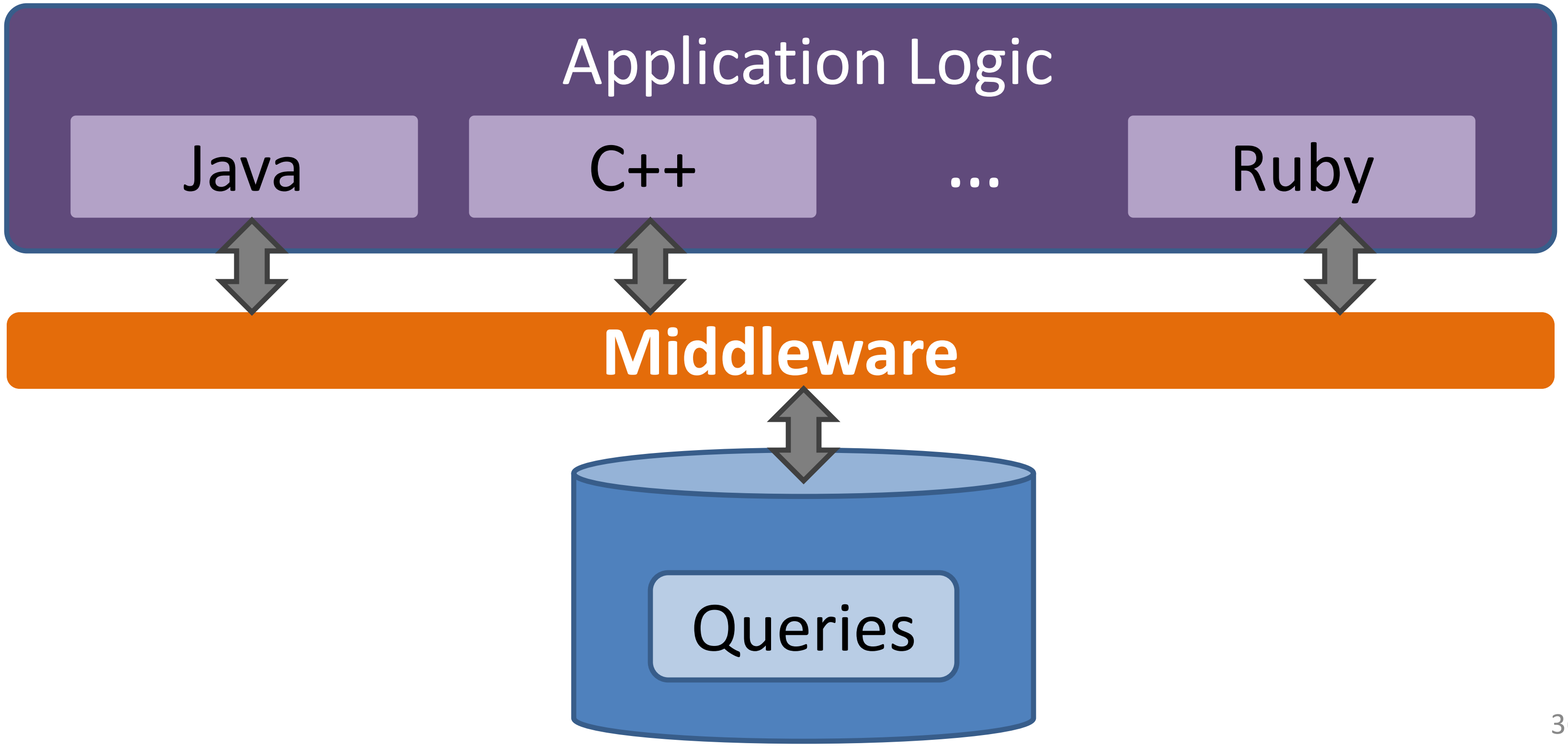
OPEN CHALLENGES

Traditional View
Datalog: Data Querying Language

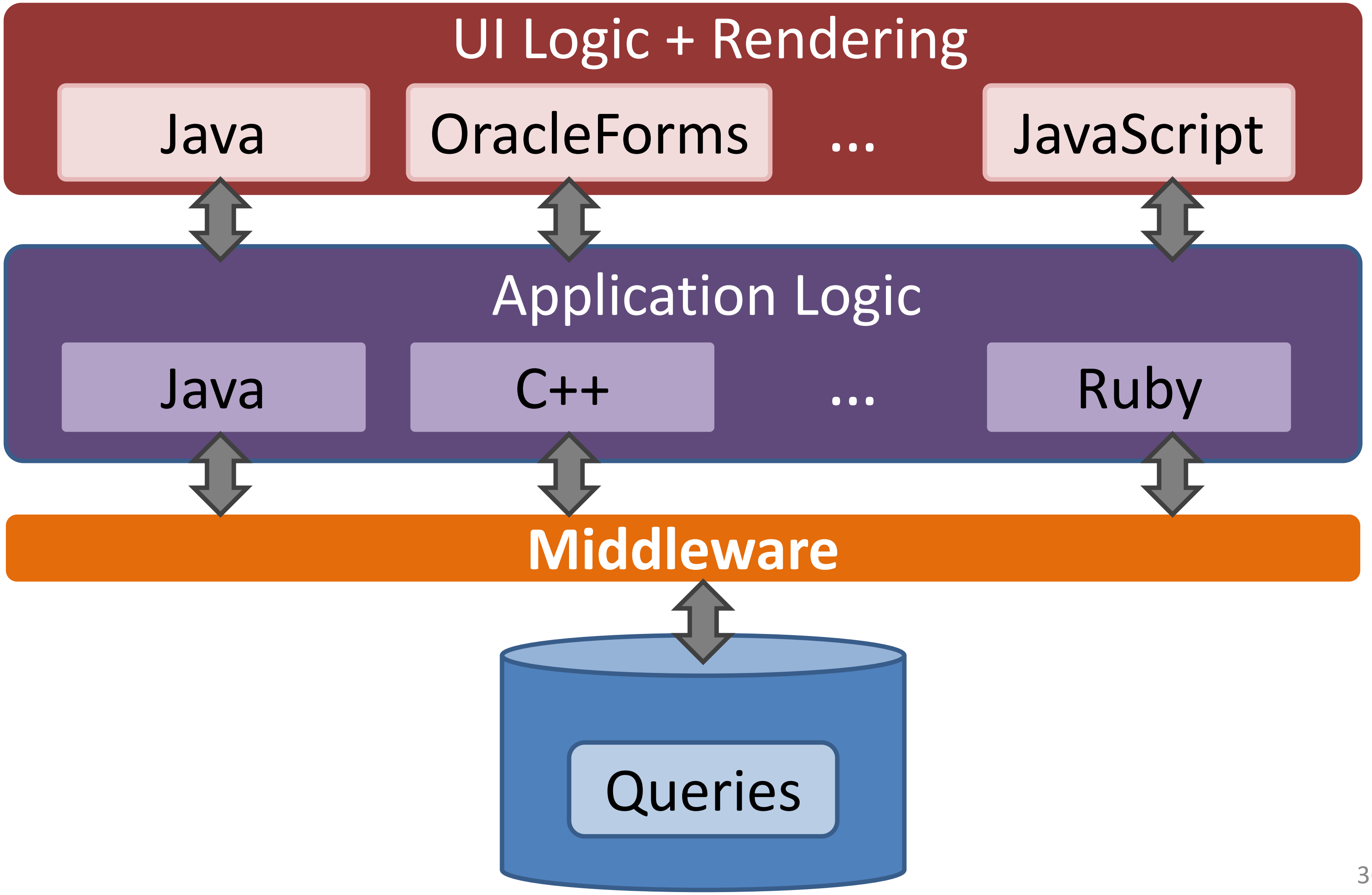


Queries

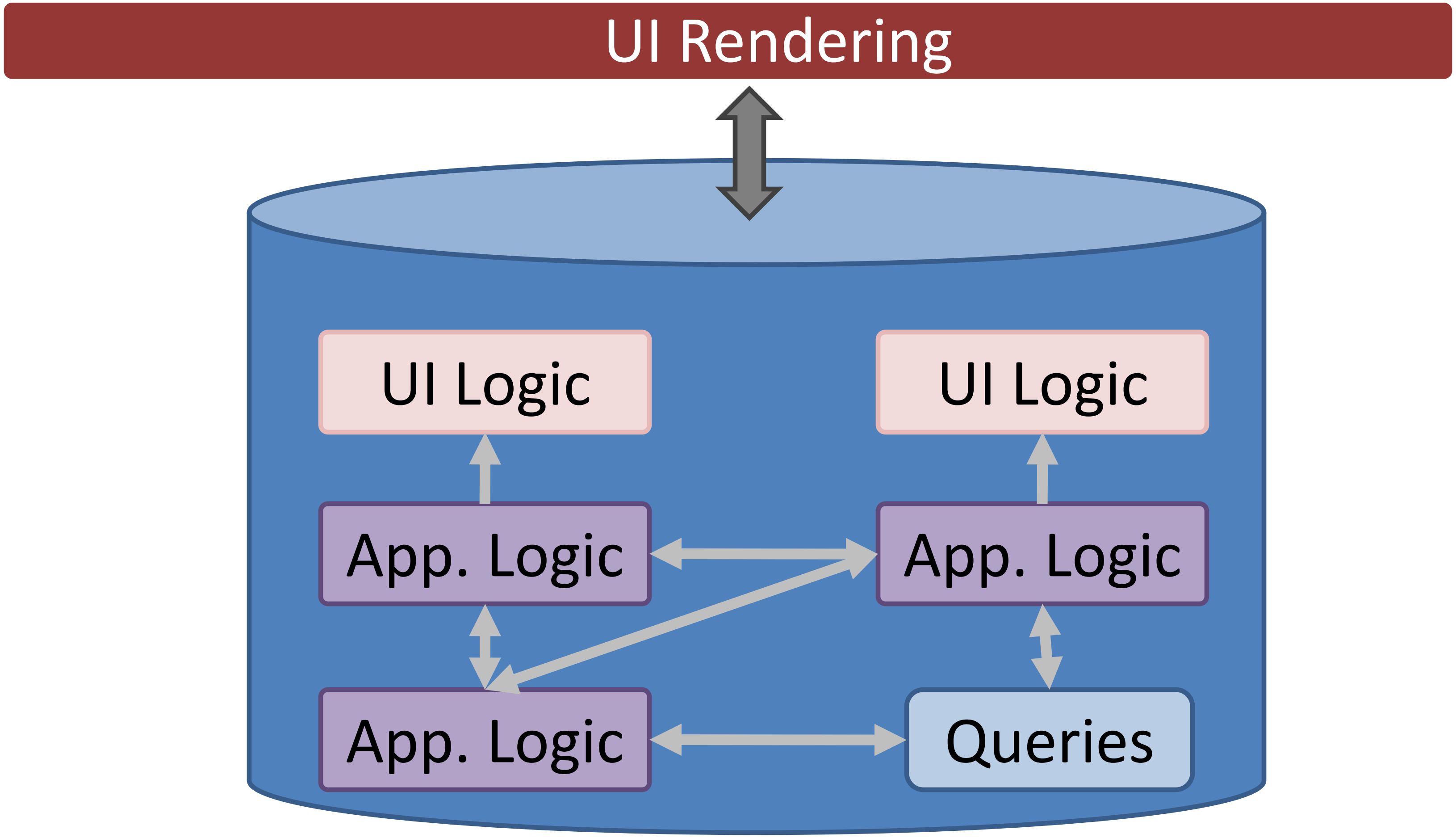
Traditional View
Datalog: Data Querying Language



Traditional View
Datalog: Data Querying Language



New View
Datalog: General Purpose Language


UI Rendering

UI Logic

UI Logic

App. Logic

App. Logic

App. Logic

Queries

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

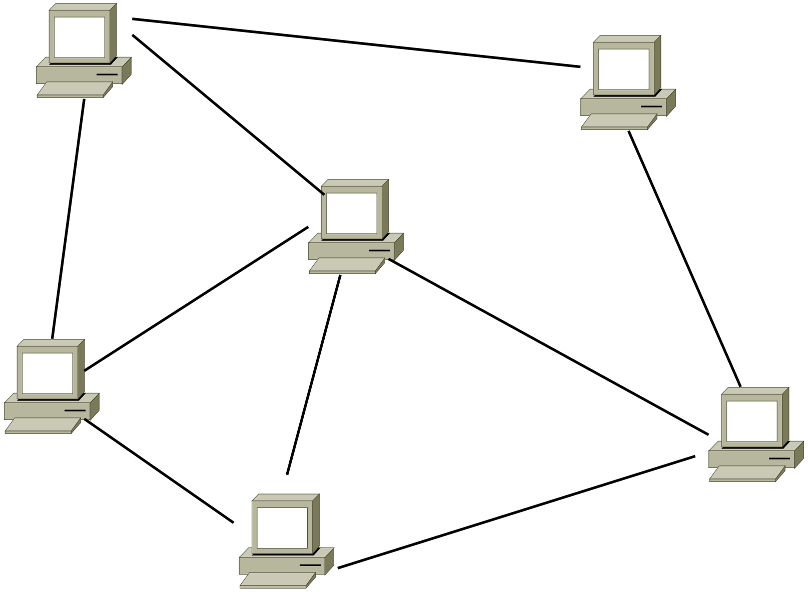
Declarative Networking

- 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

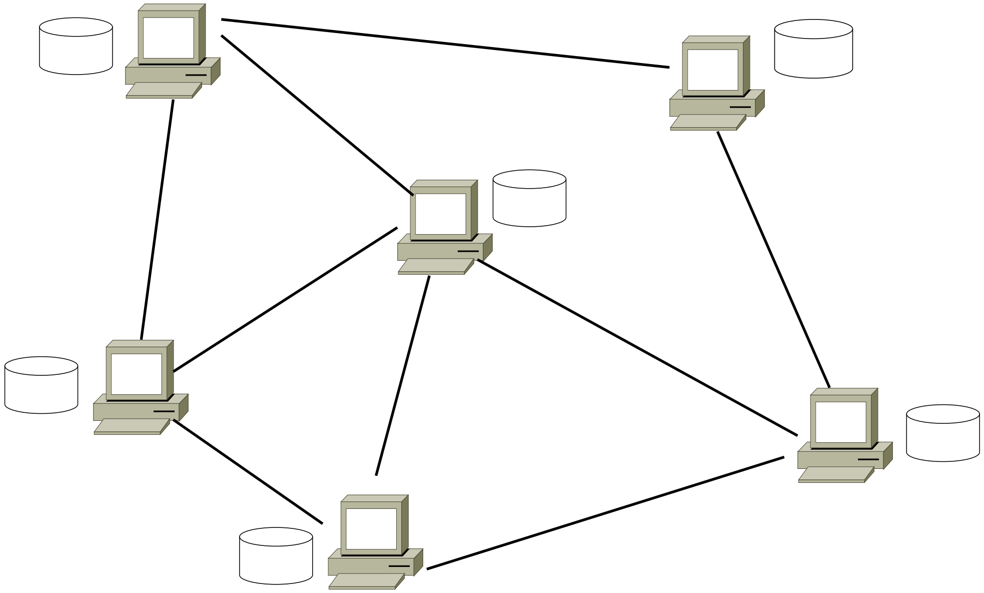
A Declarative Network



Traditional Networks

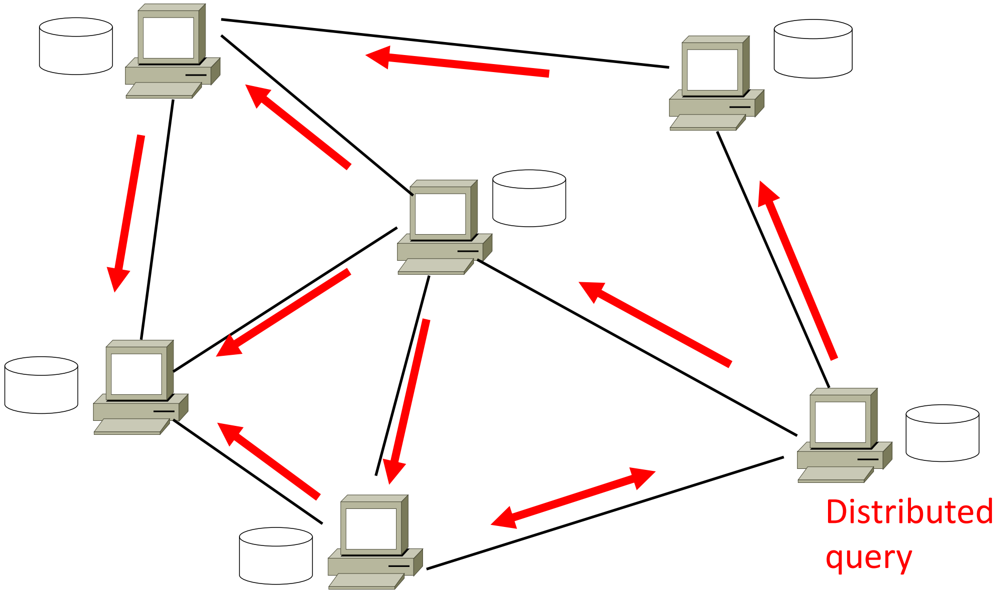
Declarative Networks

A Declarative Network



Traditional Networks	Declarative Networks
Network State	Distributed database

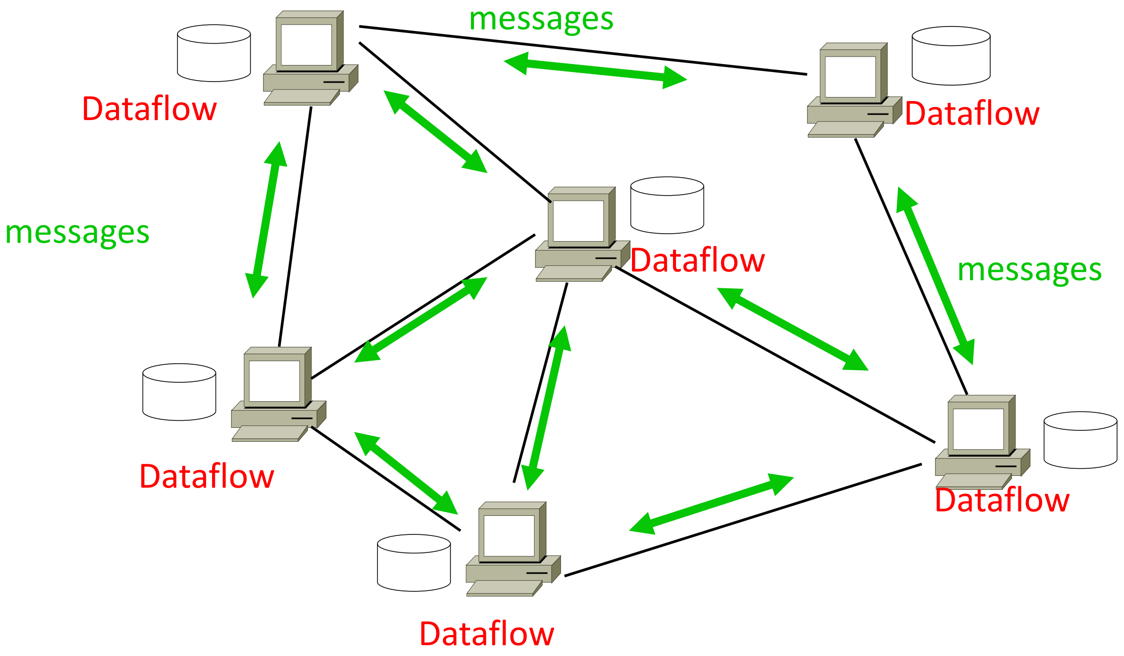
A Declarative Network



Distributed recursive query

Traditional Networks	Declarative Networks
Network State	Distributed database
Network protocol	Recursive Query Execution

A Declarative Network



Traditional Networks	Declarative Networks
Network State	Distributed database
Network protocol	Recursive Query Execution
Network messages	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>

Databases (5)
Networking (11)
Security (1)
Systems (2)

Open-source systems

- P2 declarative networking system
  - The “original” system
  - Based on modifications to the Click modular router.
  - http://p2.cs.berkeley.edu

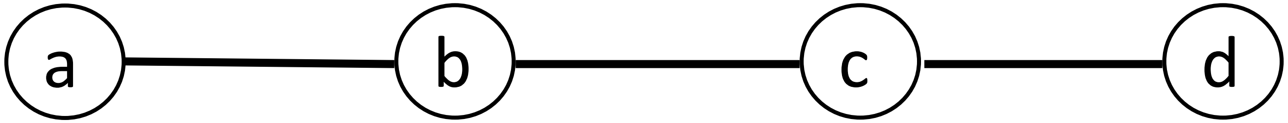
- RapidNet
  - Integrated with network simulator 3 (ns-3), ORBIT wireless testbed, and PlanetLab testbed.
  - Security and provenance extensions.
  - Demonstrations at SIGCOMM’09, SIGCOMM’11, and SIGMOD’11
  - http://netdb.cis.upenn.edu/rapidnet

- BOOM – Berkeley Orders of Magnitude
  - BLOOM (DSL in Ruby, uses Dedalus, a temporal logic programming language as its formal basis).
  - http://boom.cs.berkeley.edu/

Network Datalog

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

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

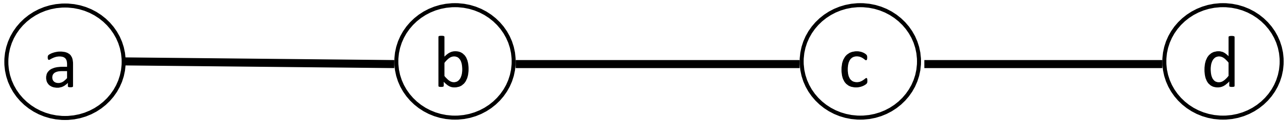


Network Datalog

Location Specifier “@S”

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

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



Network Datalog

Location Specifier “@S”

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

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

Input table:

link

@S	**D**
@a	b

link

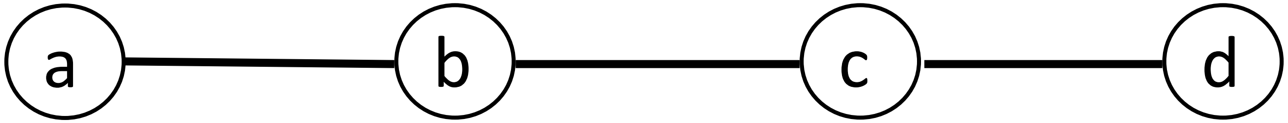
@S	**D**
@b	c
@b	a

link

@S	**D**
@c	b
@c	d

link

@S	**D**
@d	c



Network Datalog

R1: 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

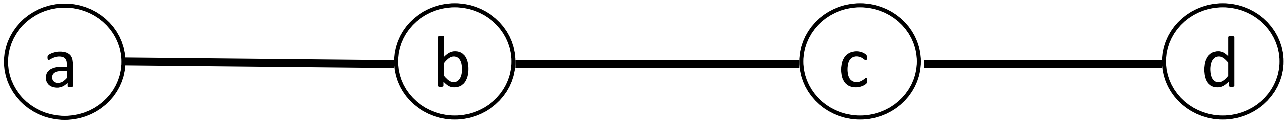
@S	D
@b	c
@b	a

link

@S	D
@c	b
@c	d

link

@S	D
@d	c



Network Datalog

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

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

query _(@M,N) <- reachable(@M,N) ← All-Pairs Reachability

Input table:

link

@S	D
@a	b

link

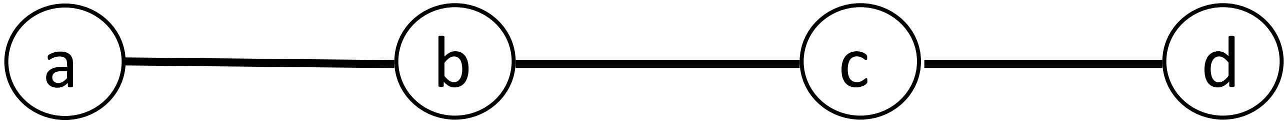
@S	D
@b	c
@b	a

link

@S	D
@c	b
@c	d

link

@S	D
@d	c



Output table:

reachable

@S	D
@a	b
@a	c
@a	d

reachable

@S	D
@b	a
@b	c
@b	d

reachable

@S	D
@c	a
@c	b
@c	d

reachable

@S	D
@d	a
@d	b
@d	c

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:

link

@S	D
@a	b

link

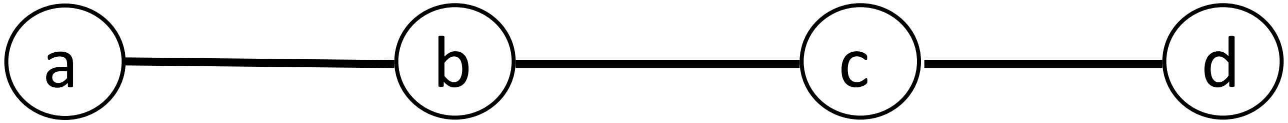
@S	D
@b	c
@b	a

link

@S	D
@c	b
@c	d

link

@S	D
@d	c



Output table:

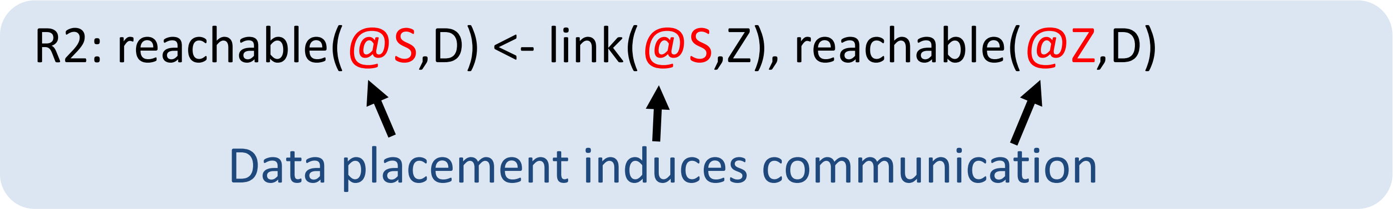
reachable

@S	D
@a	b
@a	c
@a	d

Query: reachable(@a,N)

Implicit Communication

- A networking language with no explicit communication:

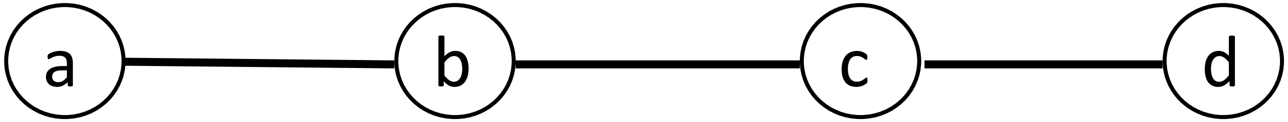


```
R2: reachable(@S,D) <- link(@S,Z), reachable(@Z,D)
```

Data placement induces communication

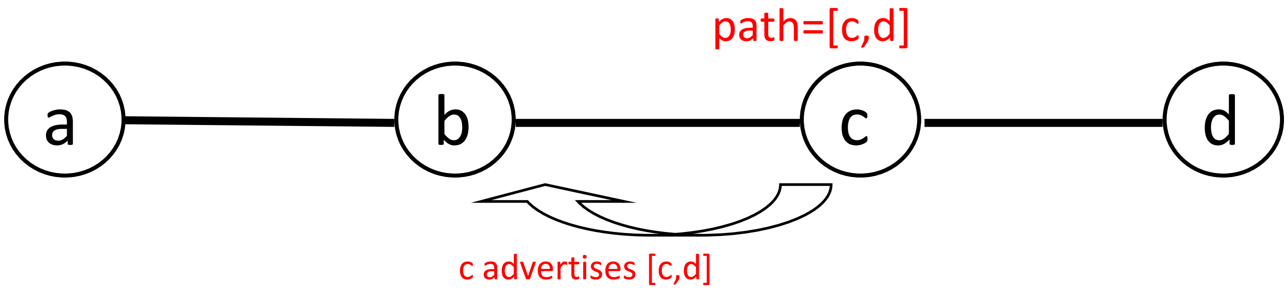
Path Vector Protocol Example

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



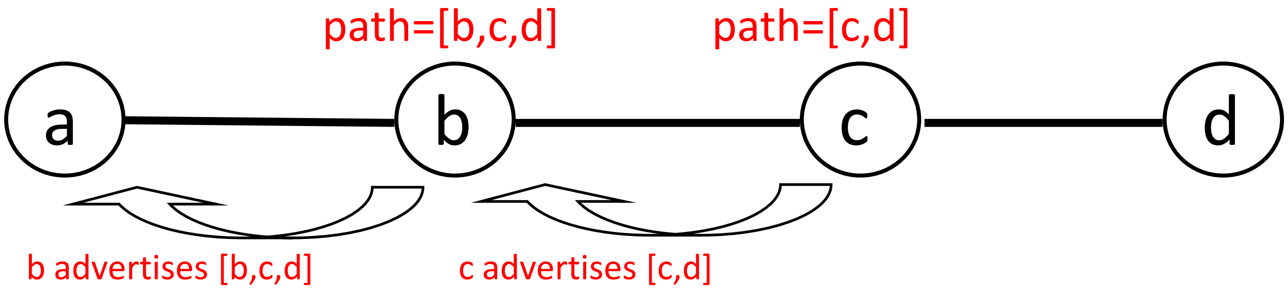
Path Vector Protocol Example

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



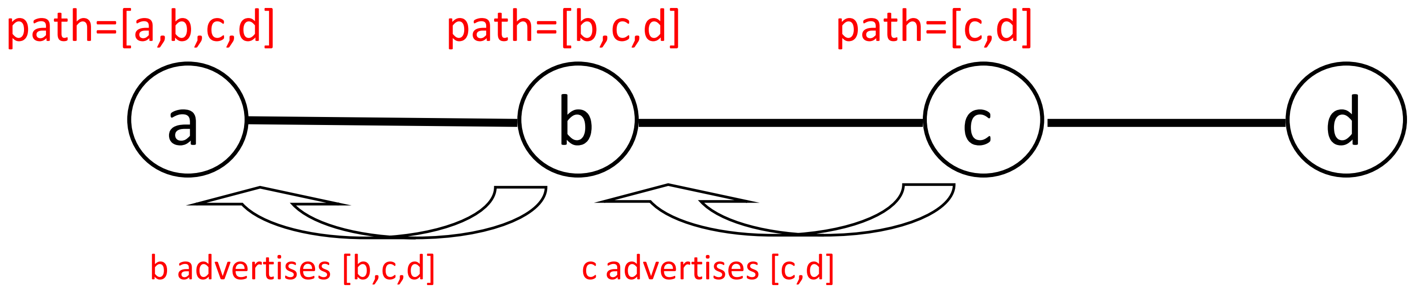
Path Vector Protocol Example

- 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: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query _(@S,D,P) <- 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

R1: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query _(@S,D,P) <- 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

R1: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query _(@S,D,P) <- path(@S,D,P)

Add S to front of P_2

- 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,P_2), P=S•P_2.

query _(@a,d,P) <- path(@a,d,P)

Neighbor table:

link

@S	D
@a	b

link

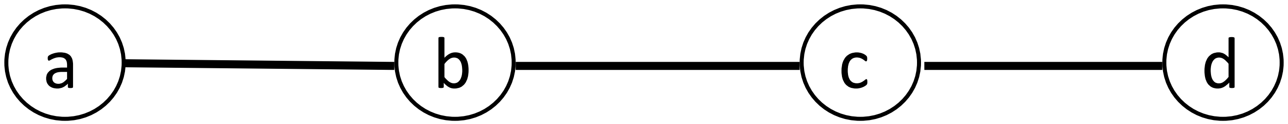
@S	D
@b	c
@b	a

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: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query(@a,d,P) <- path(@a,d,P)

Neighbor table:

link

@S	D
@a	b

link

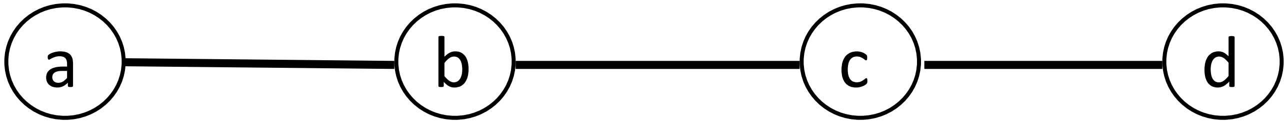
@S	D
@b	c
@b	a

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: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query(@a,d,P) <- path(@a,d,P)

Neighbor table:

link

@S	D
@a	b

link

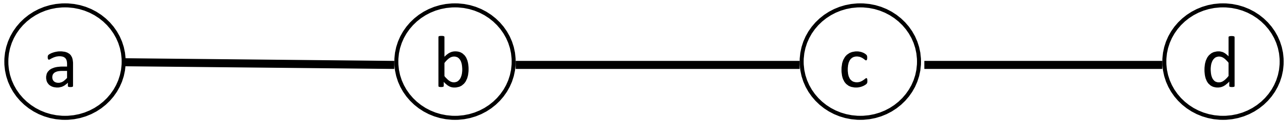
@S	D
@b	c
@b	a

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
@c	d	[c,d]

Query Execution

R1: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query _(@a,d,P) <- path(@a,d,P)

Matching variable Z = “Join” ⋈

Neighbor table:

link

@S	D
@a	b

link

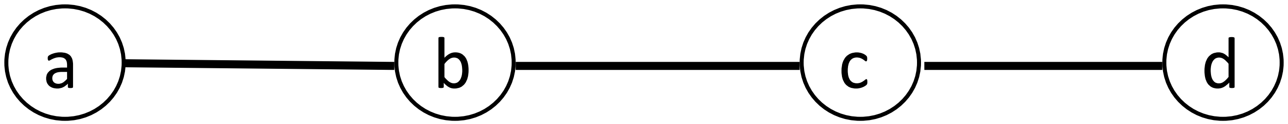
@S	D
@b	c
@b	a

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
@c	d	[c,d]

Query Execution

R1: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query _(@a,d,P) <- path(@a,d,P)

Matching variable Z = “Join” ⋈

Neighbor table:

link

@S	D
@a	b

link

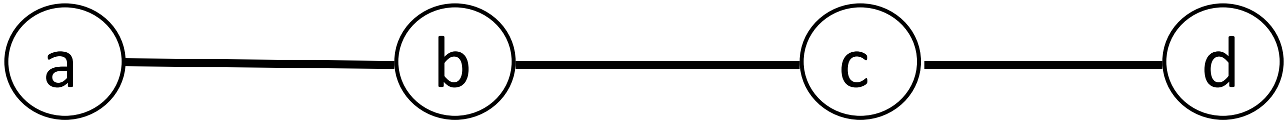
@S	D
@b	c
@b	a

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
@c	d	[c,d]

Query Execution

R1: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query _(@a,d,P) <- path(@a,d,P) Matching variable Z = “Join” ⋈

Neighbor table:

link

@S	D
@a	b

link

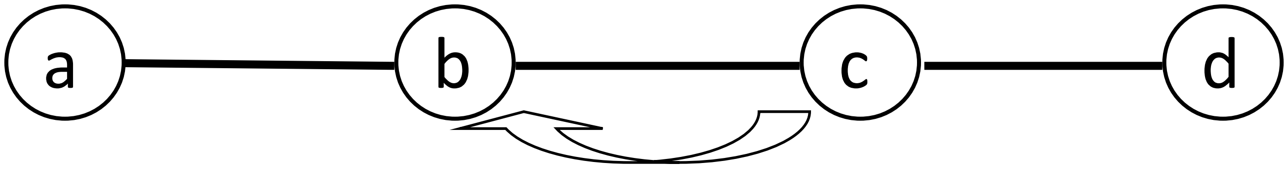
@S	D
@b	c
@b	a

link

@S	D
@c	b
@c	d

link

@S	D
@d	c



path(@b,d,[b,c,d])

Forwarding table:

path

@S	D	P

path

@S	D	P
@b	d	[b,c,d]

path

@S	D	P
@c	d	[c,d]

Query Execution

R1: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query _(@a,d,P) <- path(@a,d,P)

Matching variable Z = “Join” ⋈

Neighbor table:

link

@S	D
@a	b

link

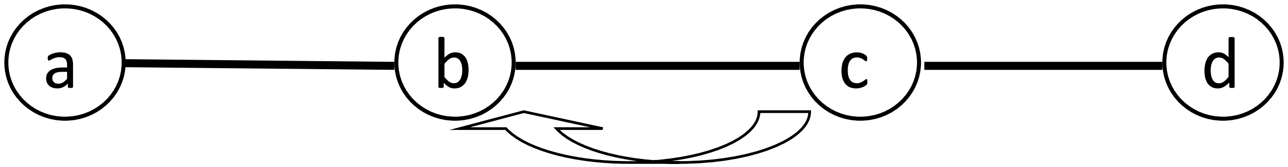
@S	D
@b	c
@b	a

link

@S	D
@c	b
@c	d

link

@S	D
@d	c



path(@b,d,[b,c,d])

Forwarding table:

path

@S	D	P

path

@S	D	P
@b	d	[b,c,d]

path

@S	D	P
@c	d	[c,d]

Query Execution

R1: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query _(@a,d,P) <- path(@a,d,P)

Matching variable Z = "Join" ⋈

Neighbor table:

link

@S	D
@a	b

link

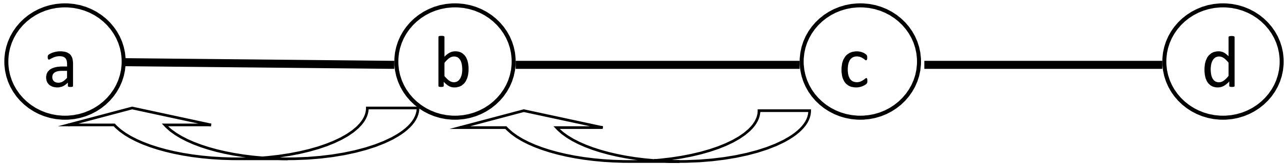
@S	D
@b	c
@b	a

link

@S	D
@c	b
@c	d

link

@S	D
@d	c



path(@a,d,[a,b,c,d])

path(@b,d,[b,c,d])

Forwarding table:

path

@S	D	P
@a	d	[a,b,c,d]

path

@S	D	P
@b	d	[b,c,d]

path

@S	D	P
@c	d	[c,d]

Query Execution

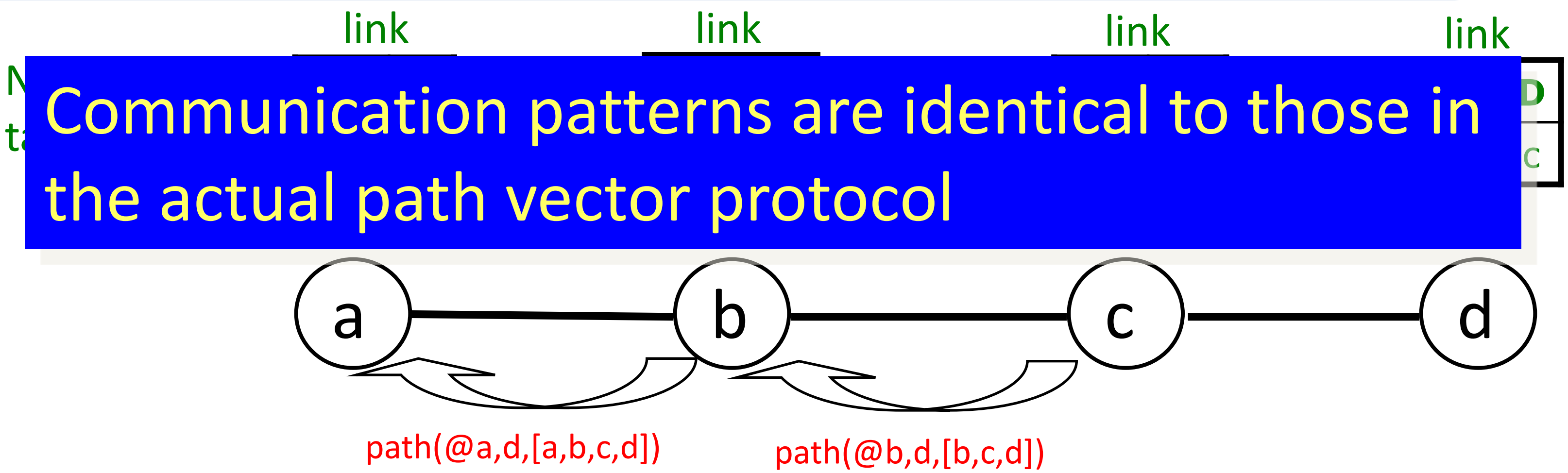
R1: path(@S,D,P) <- link(@S,D), P=(S,D).

R2: path(@S,D,P) <- link(@Z,S), path(@Z,D,P_2), P=S•P_2.

query _(@a,d,P) <- path(@a,d,P)

Matching variable Z = "Join" ⋈

Communication patterns are identical to those in the actual path vector protocol



Forwarding table:

path

@S	D	P
@a	d	[a,b,c,d]

path

@S	D	P
@b	d	[b,c,d]

path

@S	D	P
@c	d	[c,d]

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), $C=C_1+C_2$, $P=S \bullet P_2$.

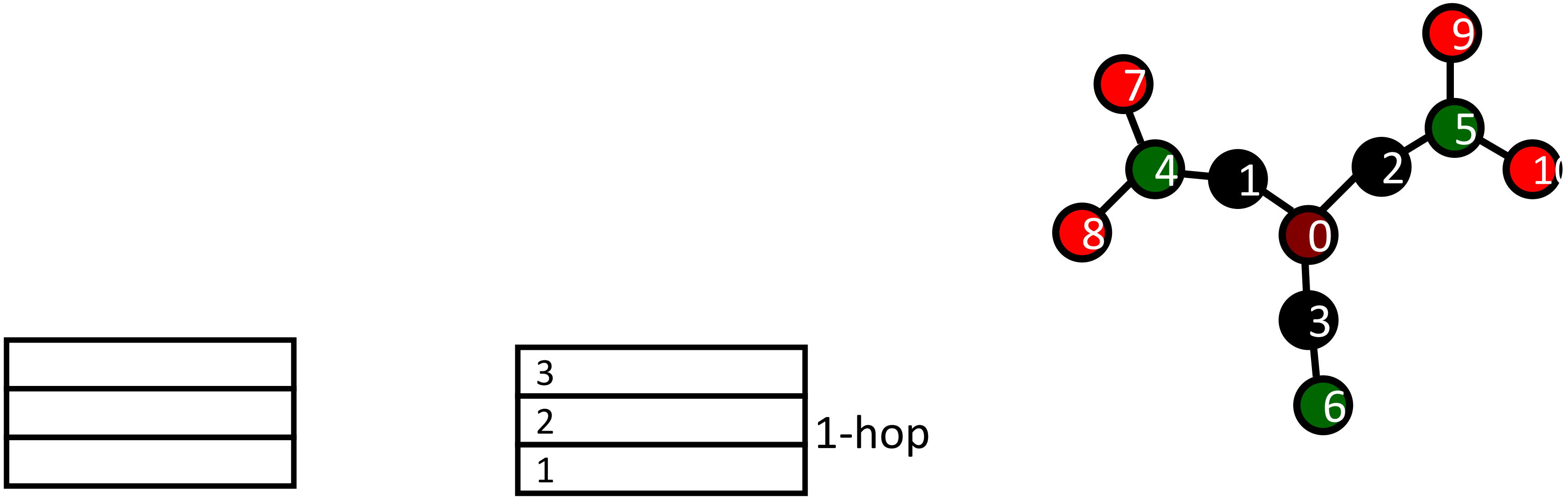
R3: bestPathCost(@S,D,min<C>) <- path(@S,D,P,C).

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 ith used in (i+1)th



Link Table

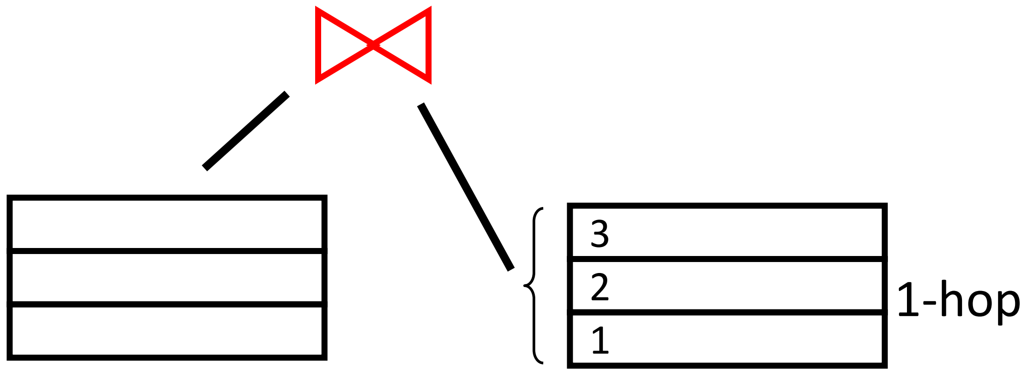
3	
2	1-hop
1	

Path Table

Network

Distributed Semi-naïve Evaluation

- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration i^{th} used in $(i+1)^{th}$



3

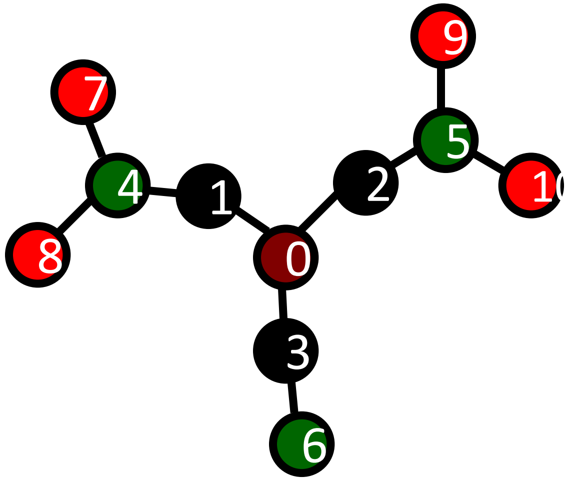
2

1

1-hop

Link Table

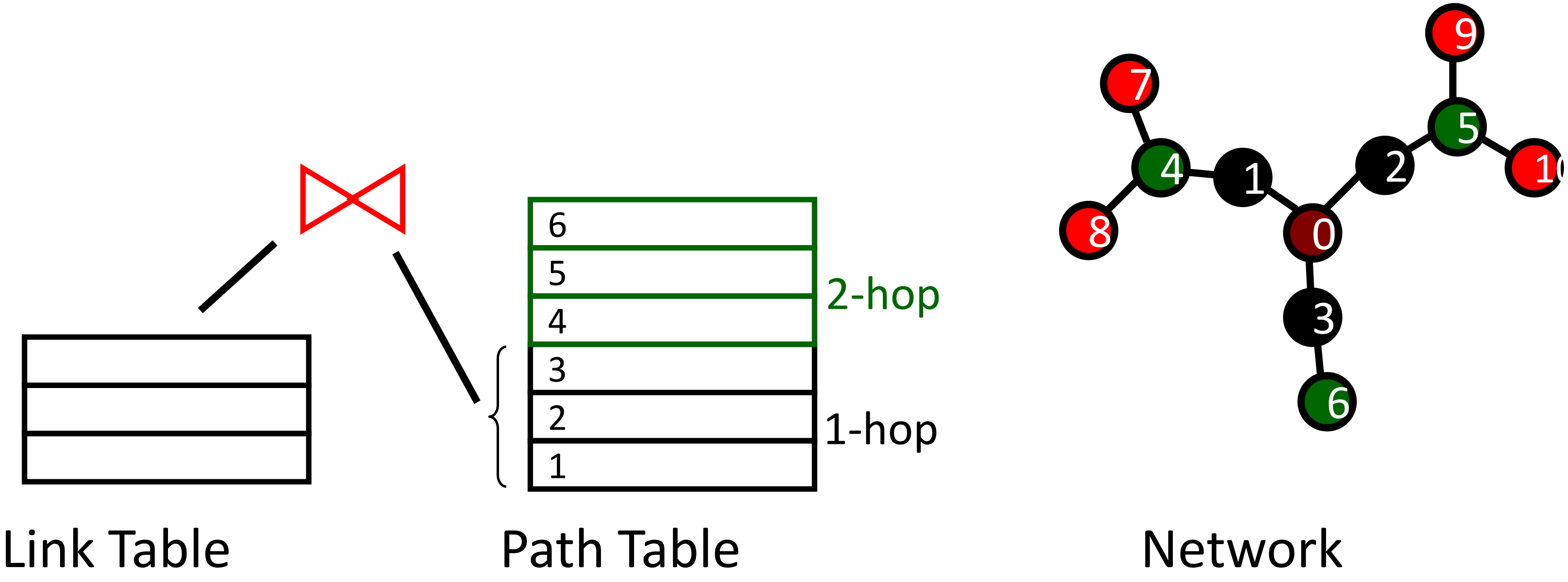
Path Table



Network

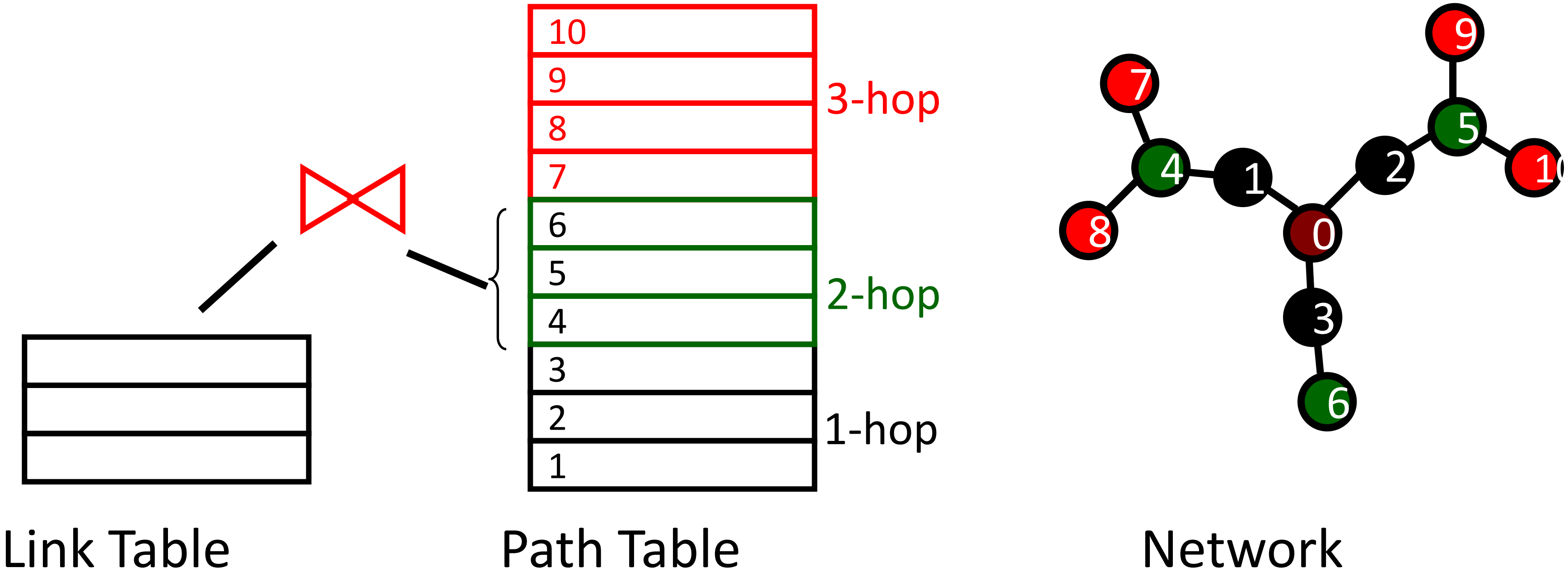
Distributed Semi-naïve Evaluation

- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration ith used in (i+1)th



Distributed Semi-naïve Evaluation

- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration ith used in (i+1)th



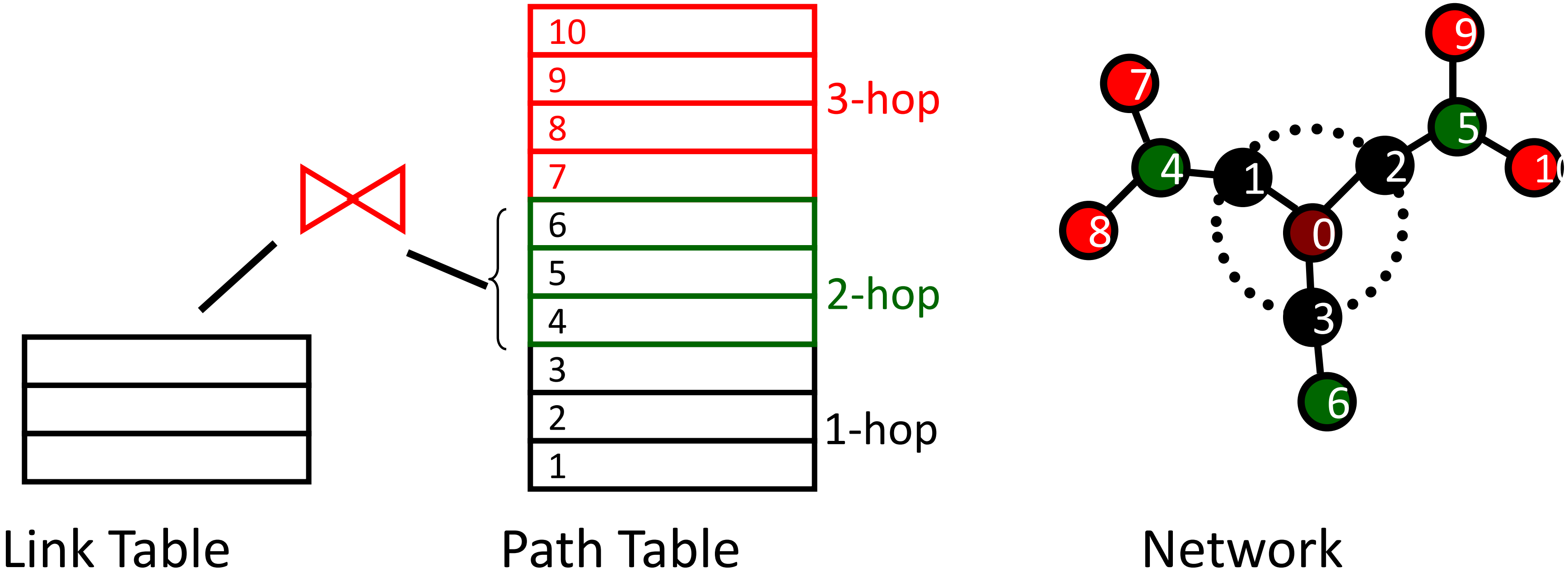
Link Table

Path Table

Network

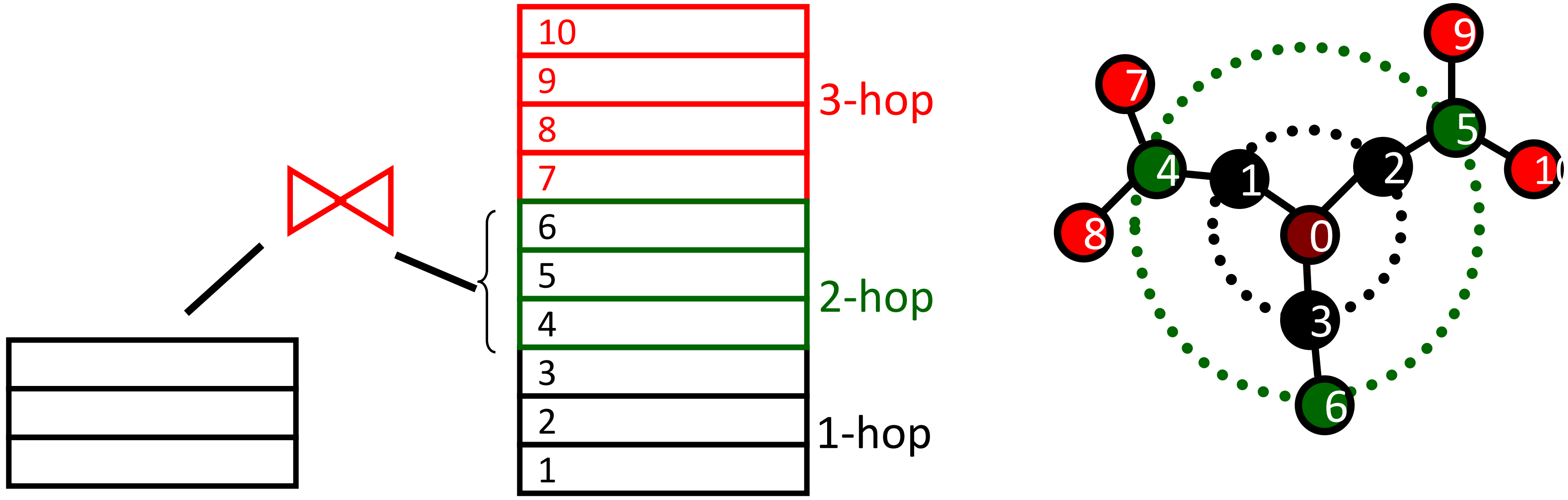
Distributed Semi-naïve Evaluation

- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration ith used in (i+1)th



Distributed Semi-naïve Evaluation

- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration ith used in (i+1)th



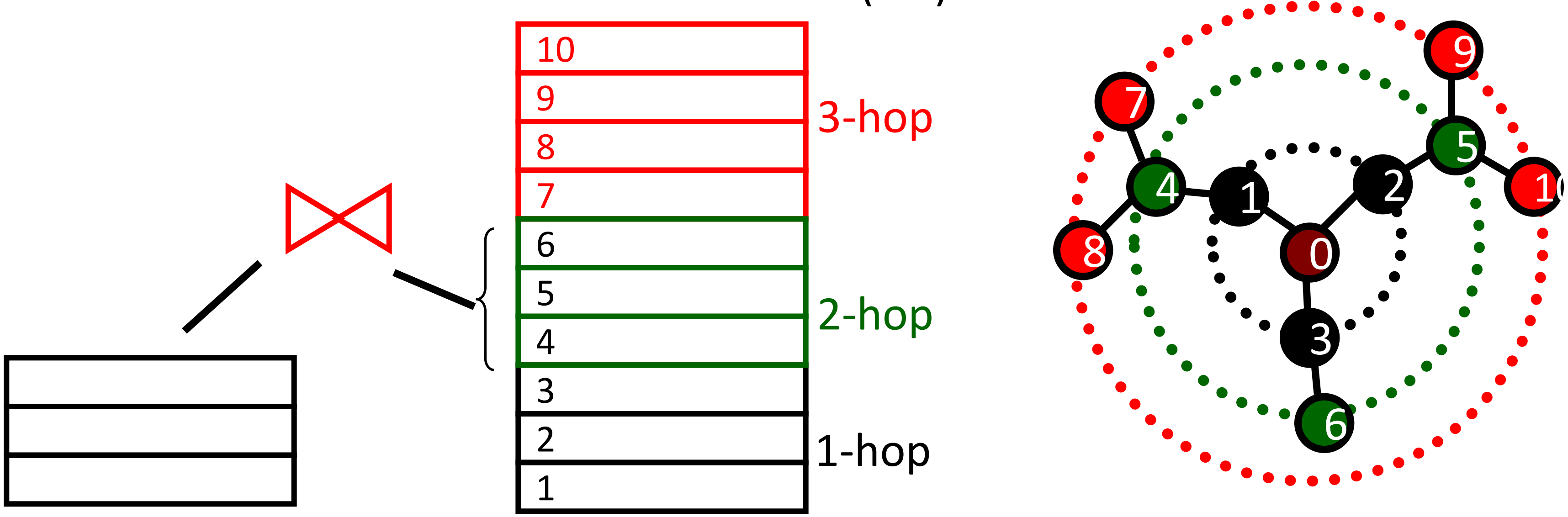
Link Table

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



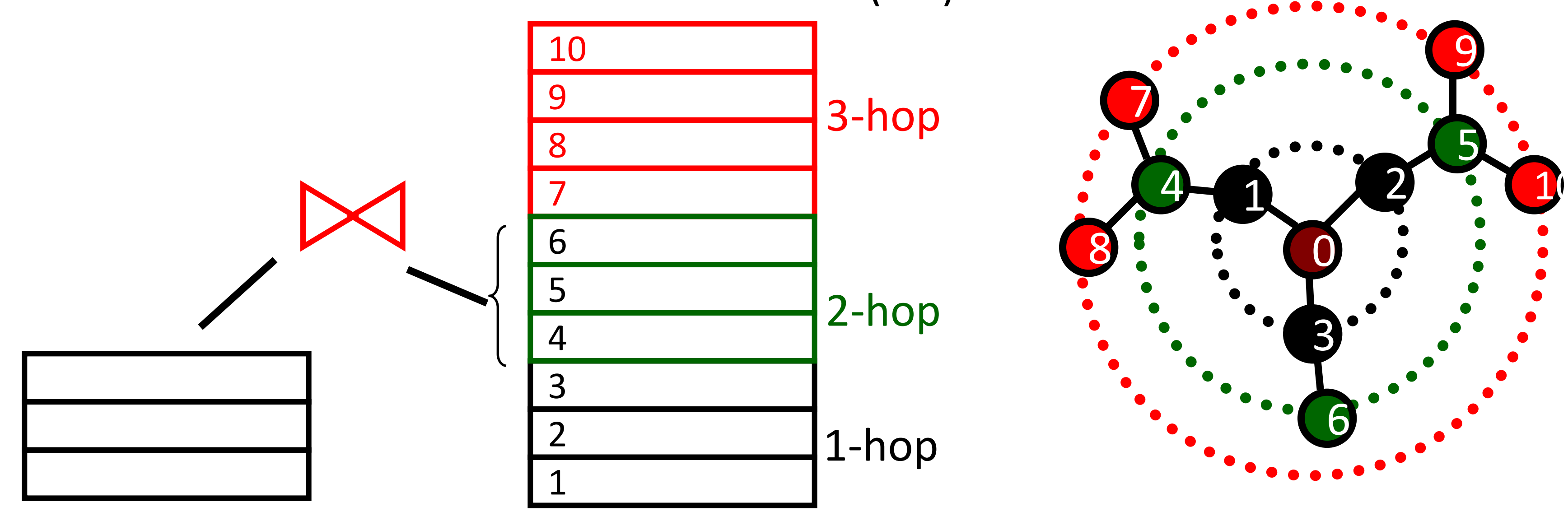
Link Table

Path Table

Network

Distributed Semi-naïve Evaluation

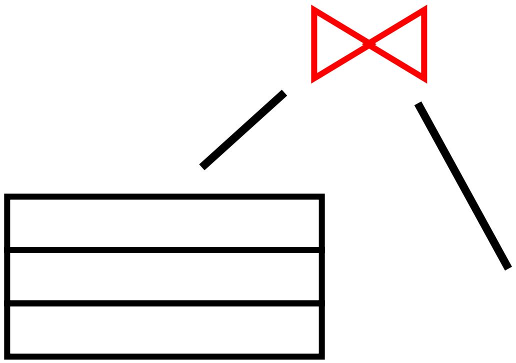
- Semi-naïve evaluation:
  - Iterations (rounds) of synchronous computation
  - Results from iteration ith used in (i+1)th



Problem: How do nodes know that an iteration is completed? Unpredictable delays and failures make synchronization difficult/expensive.

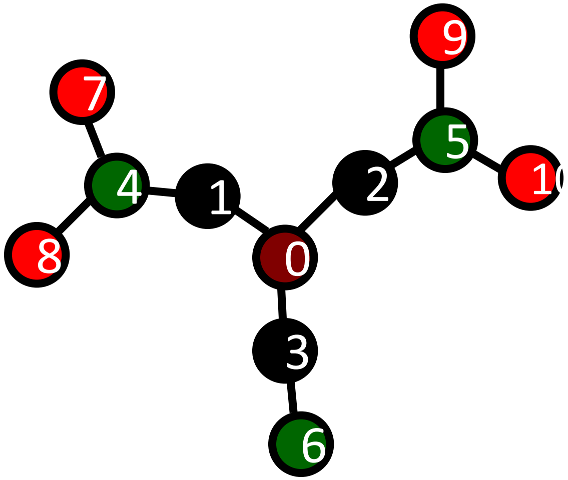
Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows



Link Table

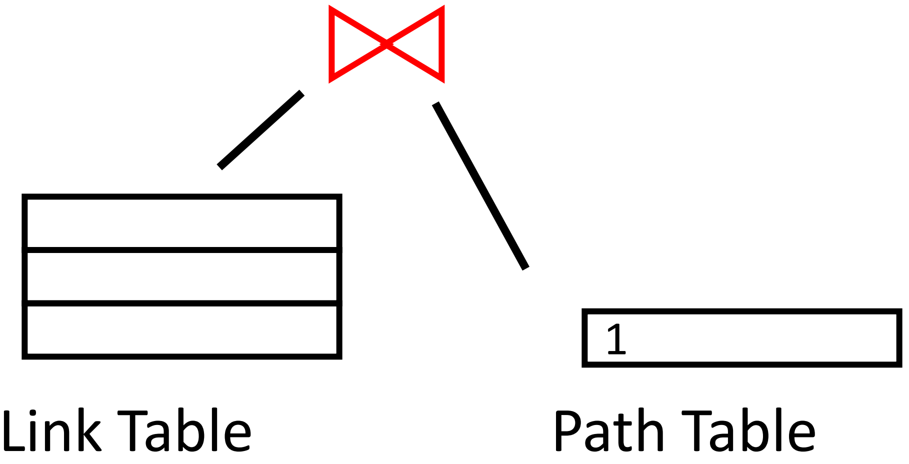
Path Table

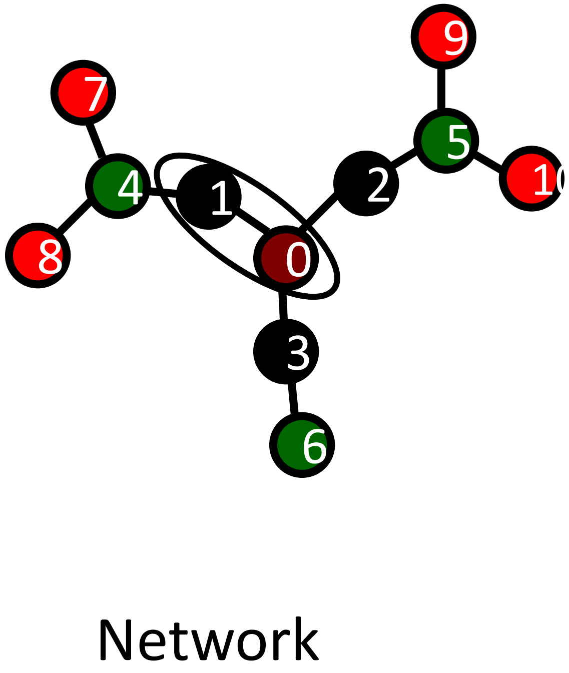


Network

Pipelined Semi-naïve (PSN)

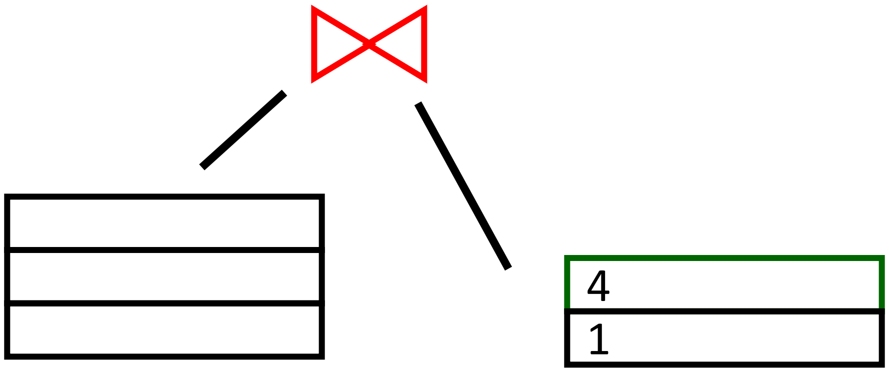
- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows





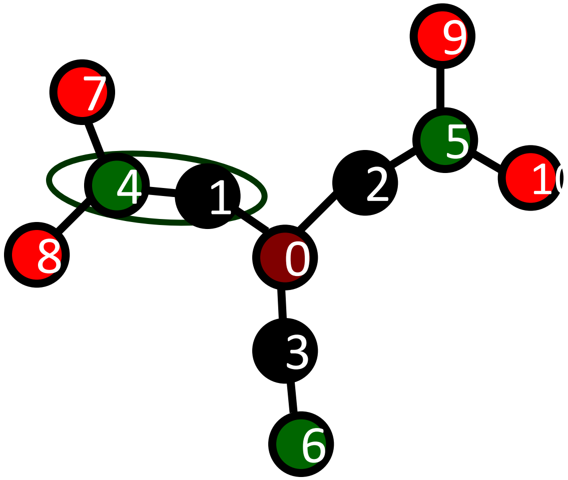
Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows



Link Table

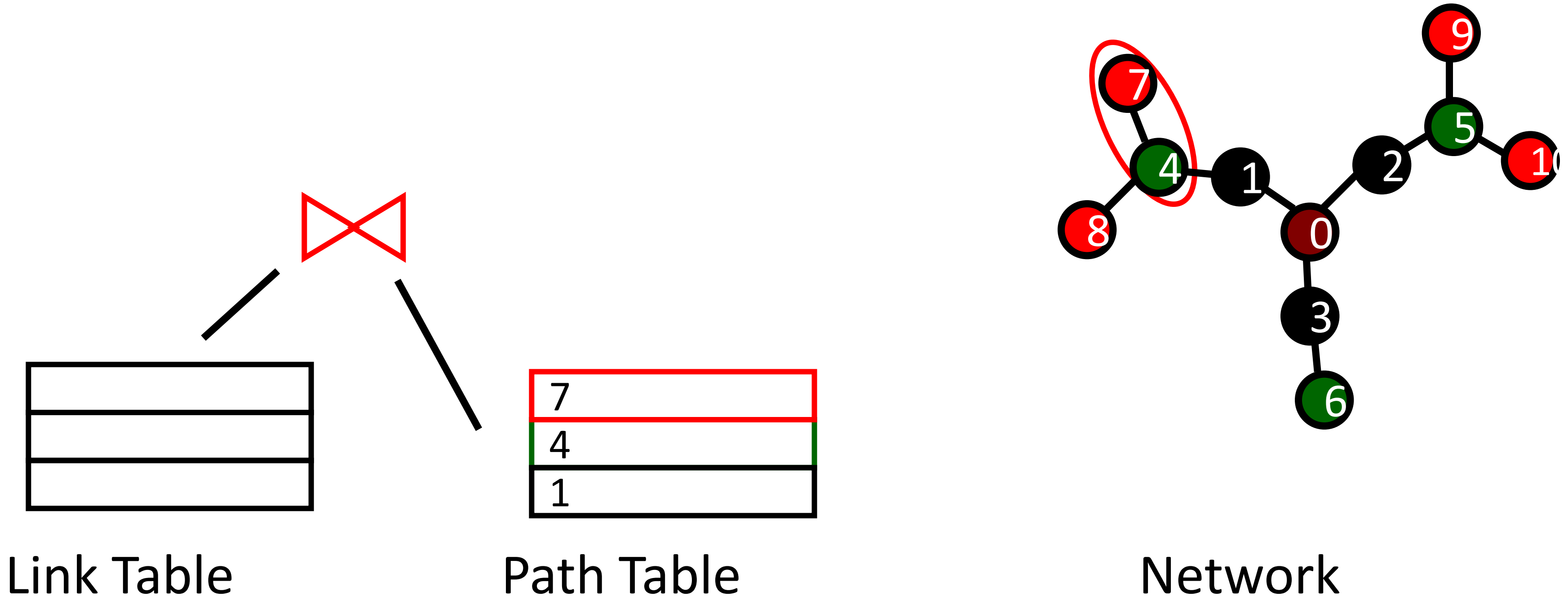
Path Table



Network

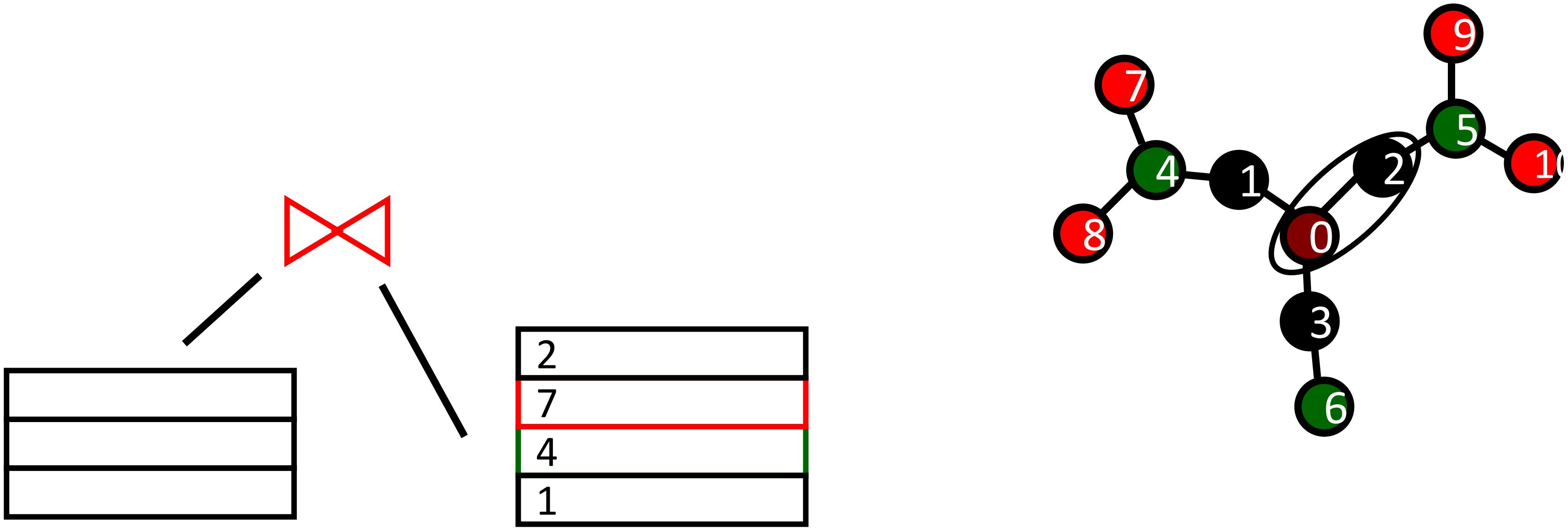
Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows



Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows



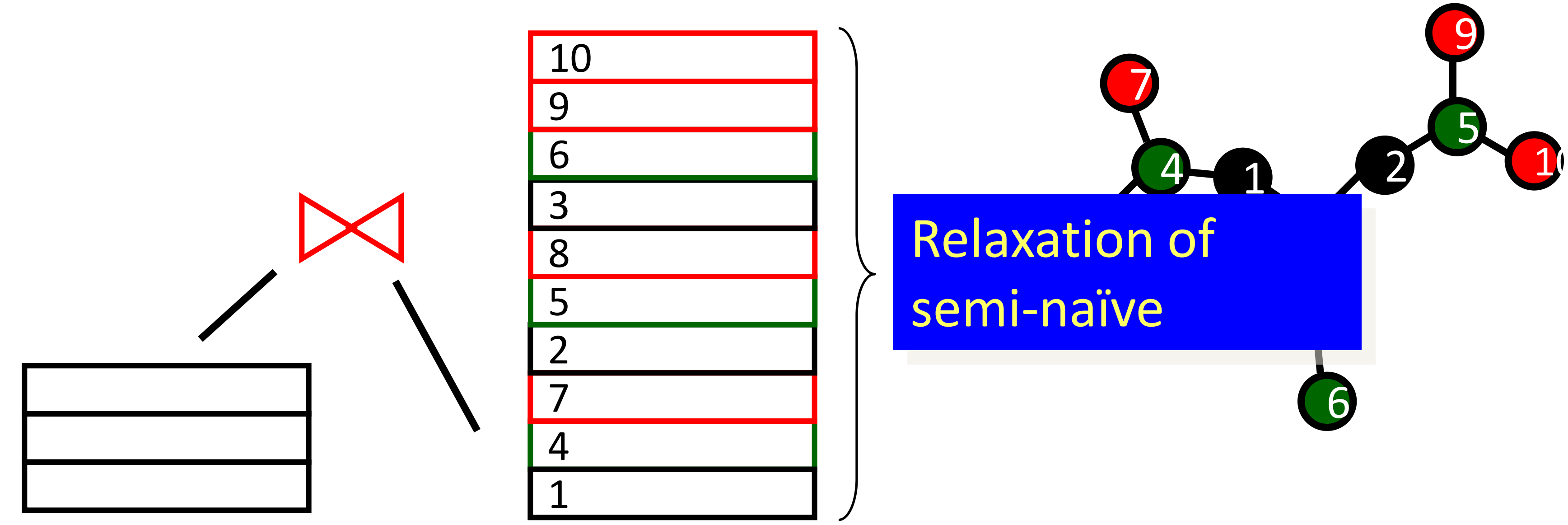
Link Table

Path Table

Network

Pipelined Semi-naïve (PSN)

- Fully-asynchronous evaluation:
  - Computed tuples in *any* iteration are pipelined to next iteration
  - Natural for distributed dataflows



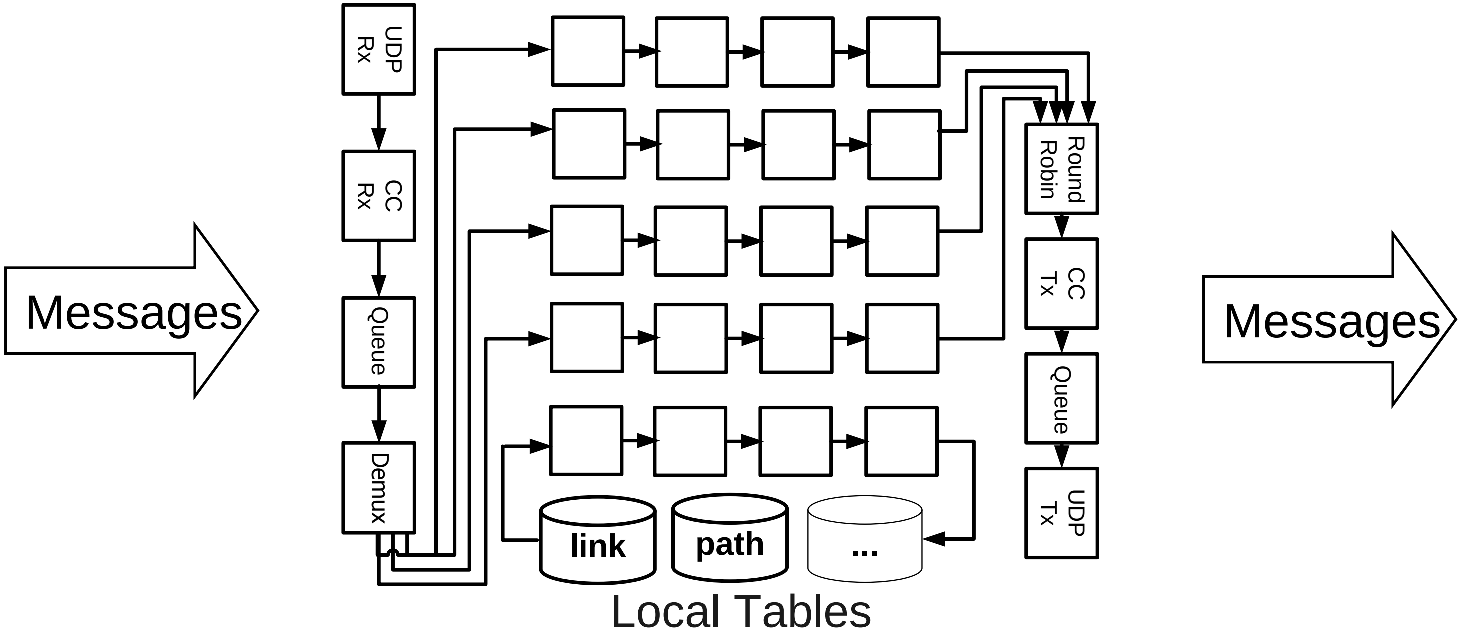
Relaxation of semi-naïve

Link Table

Path Table

Network

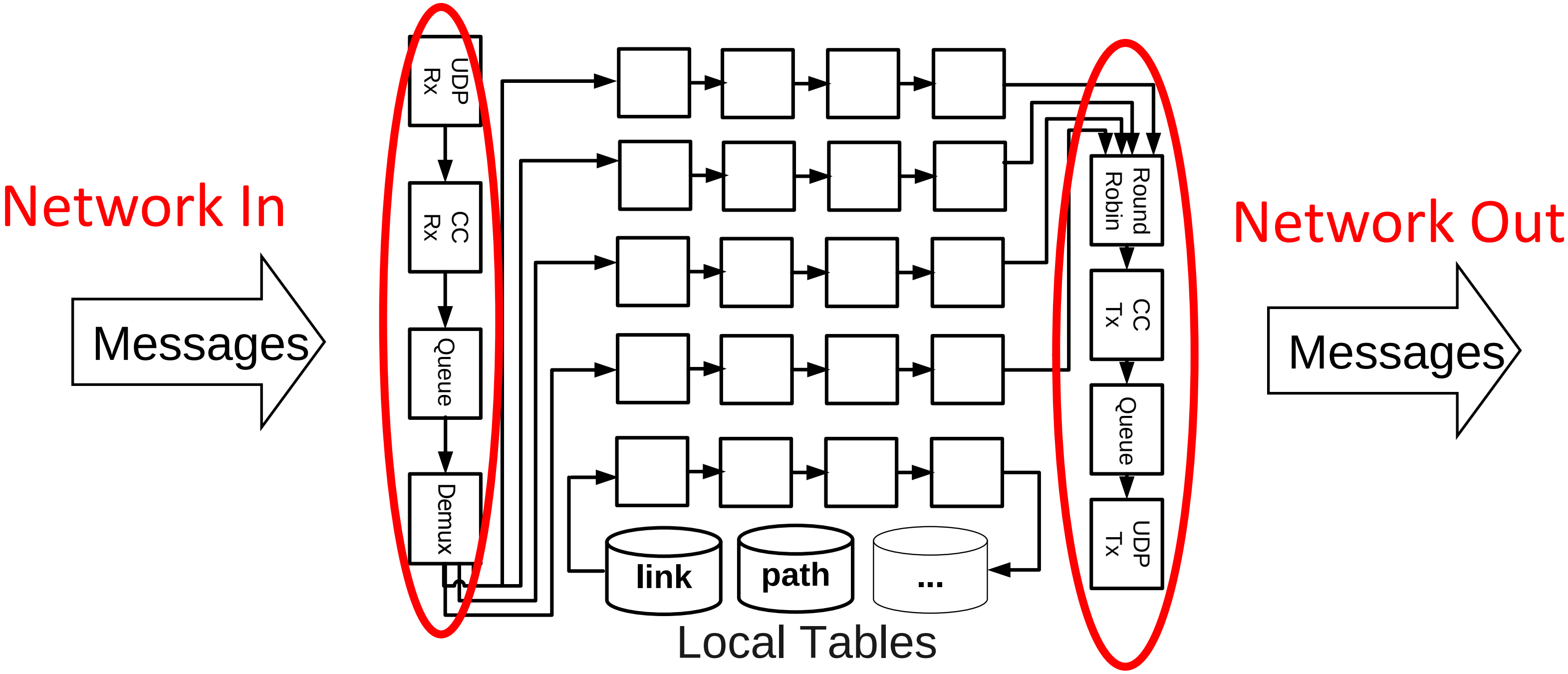
Dataflow Graph



Single Node

- Nodes in dataflow graph ("elements"):
  - Network elements (send/recv, rate limitation, jitter)
  - Flow elements (mux, demux, queues)
  - Relational operators (selects, projects, joins, aggregates)

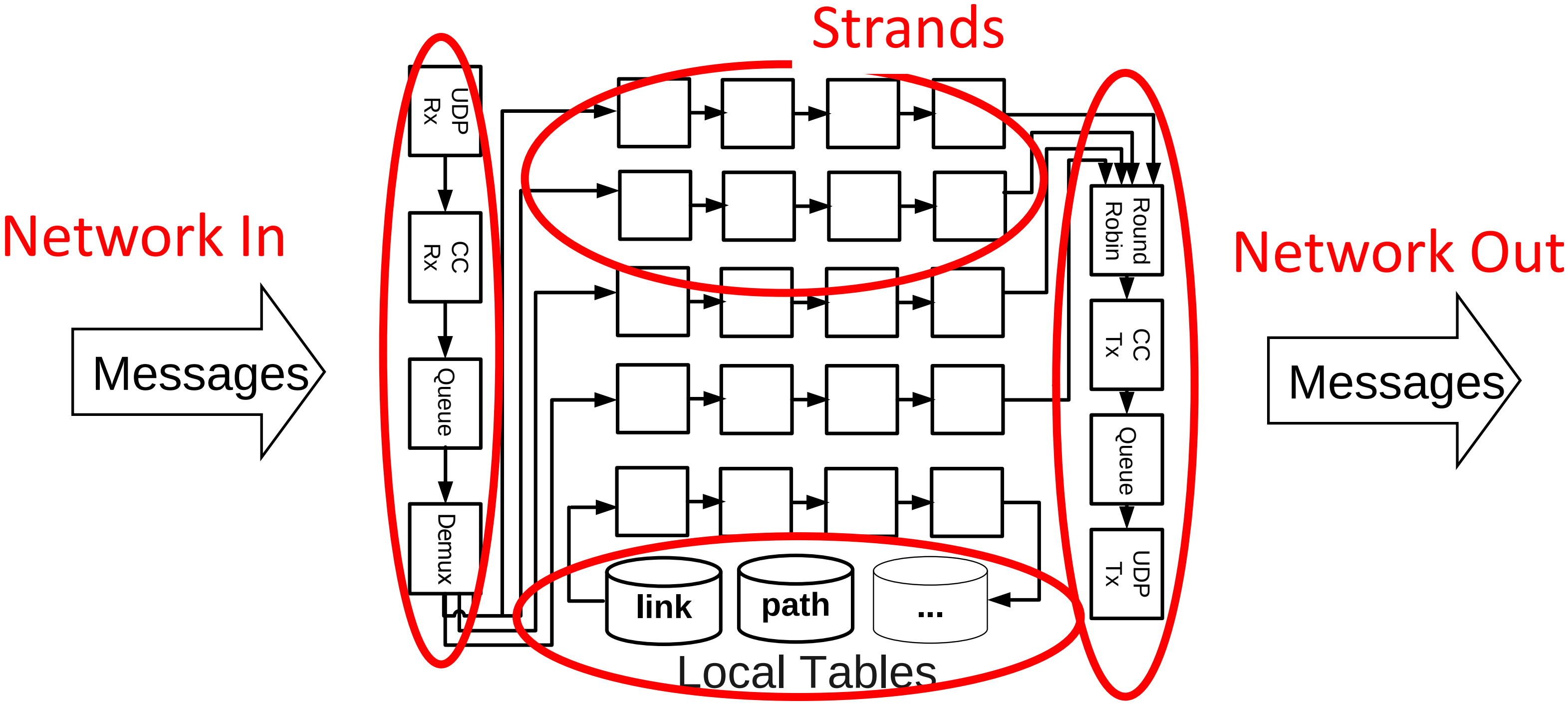
Dataflow Graph



Single Node

- Nodes in dataflow graph ("elements"):
  - Network elements (send/recv, rate limitation, jitter)
  - Flow elements (mux, demux, queues)
  - Relational operators (selects, projects, joins, aggregates)

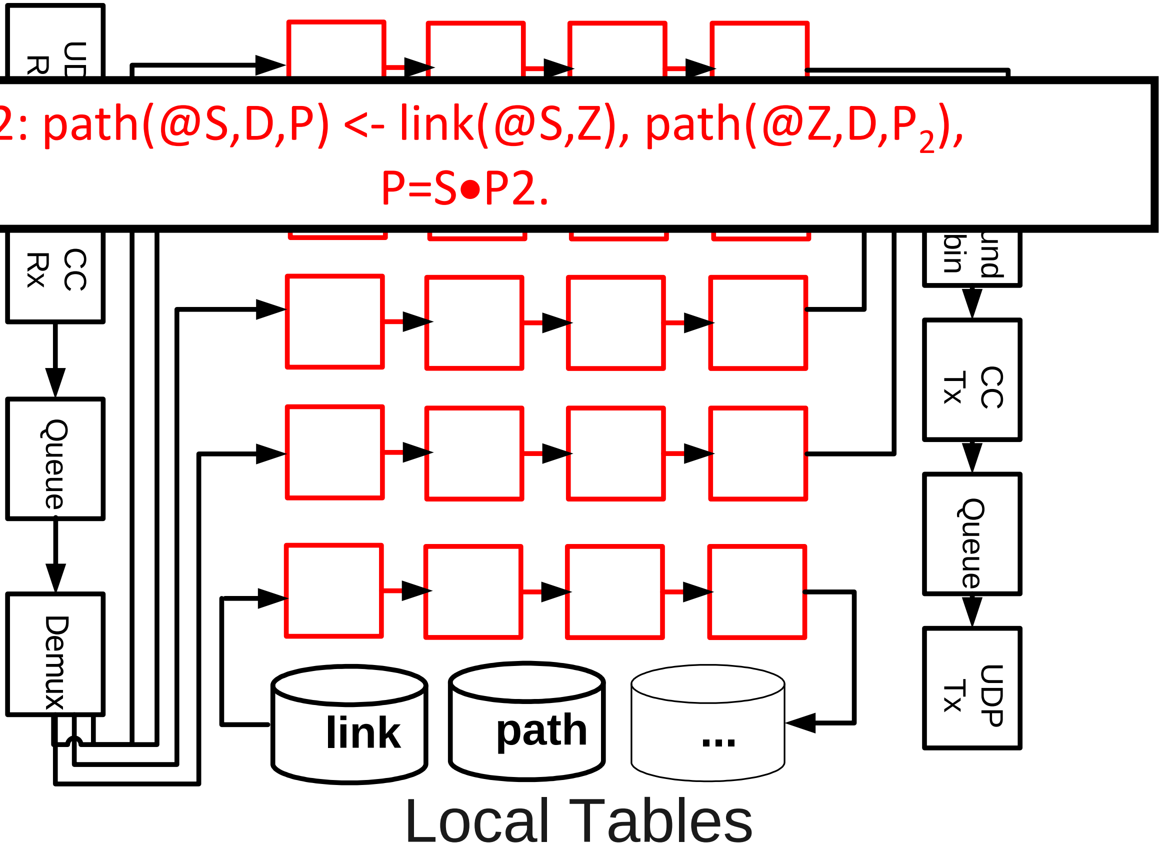
Dataflow Graph



Single Node

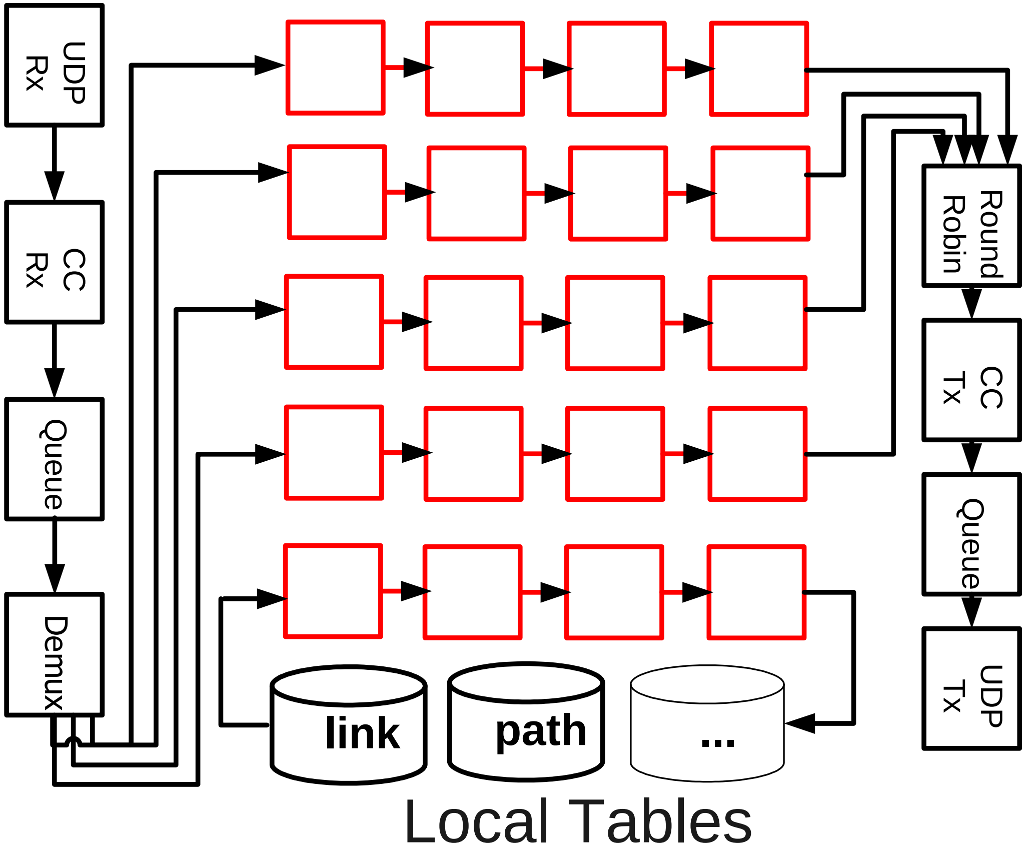
- Nodes in dataflow graph ("elements"):
  - Network elements (send/recv, rate limitation, jitter)
  - Flow elements (mux, demux, queues)
  - Relational operators (selects, projects, joins, aggregates)

Rule → Dataflow “Strands”



R2: path(@S,D,P) <- link(@S,Z), path(@Z,D,P_2),
P=S•P2.

Rule → Dataflow "Strands"



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” ⋈

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,P_2), P=S•P_2.

Matching variable Z = “Join” $\bowtie$

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,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.

Matching variable Z = “Join” ⋈

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:

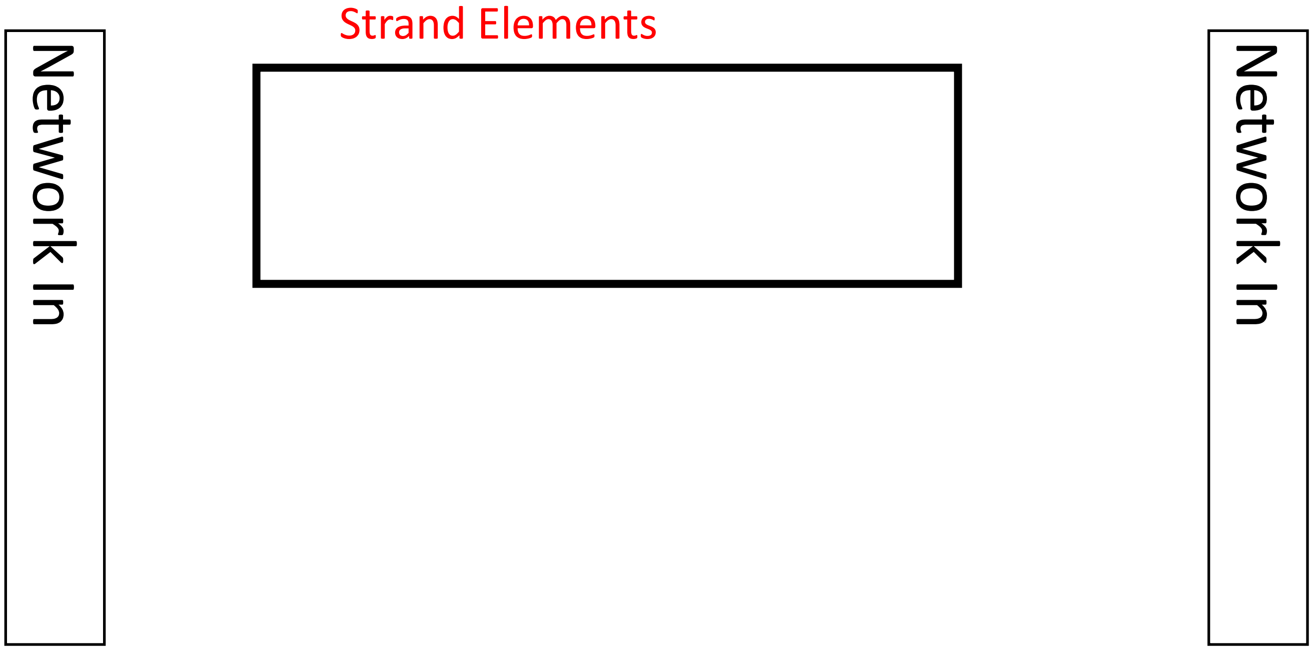
R2a: linkD(S,@D) ← link(@S,D)

R2b: path(@S,D,P) ← linkD(S,@Z), path(@Z,D,P_2), P=S•P_2.

Matching variable Z = “Join” ⋈

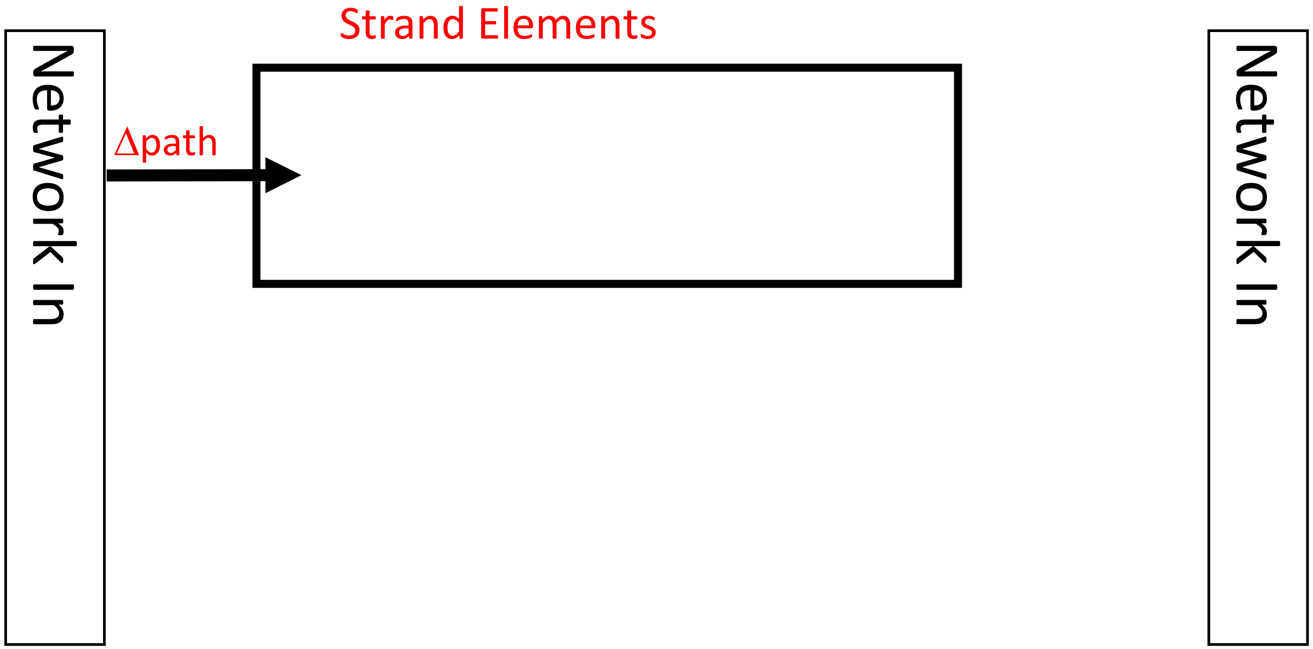
Physical Execution Plan

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P_2), P=S•P_2.



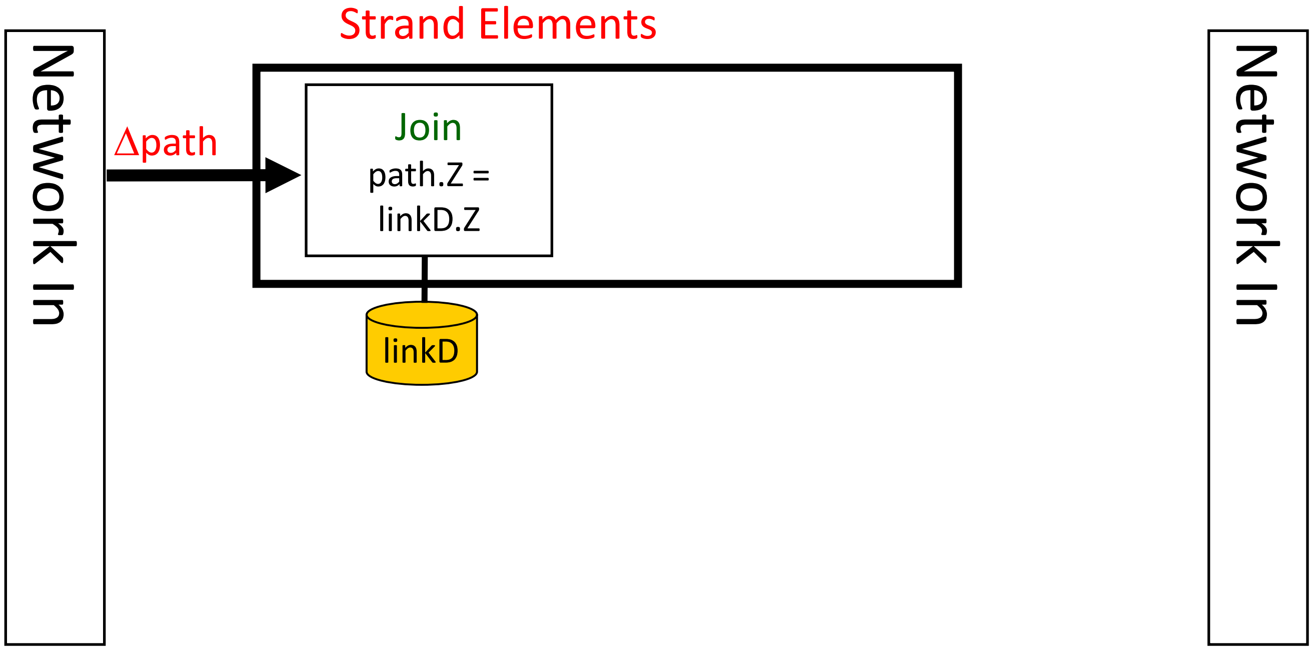
Physical Execution Plan

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P_2), P=S•P_2.



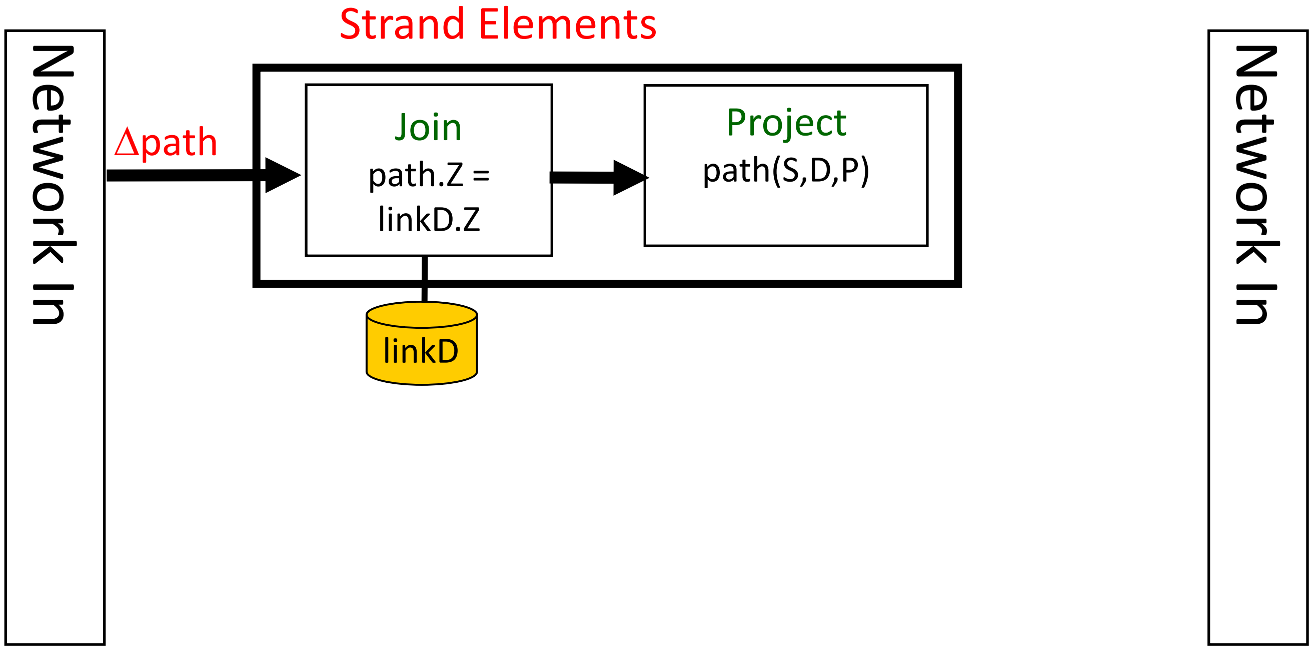
Physical Execution Plan

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P_2), P=S•P_2.



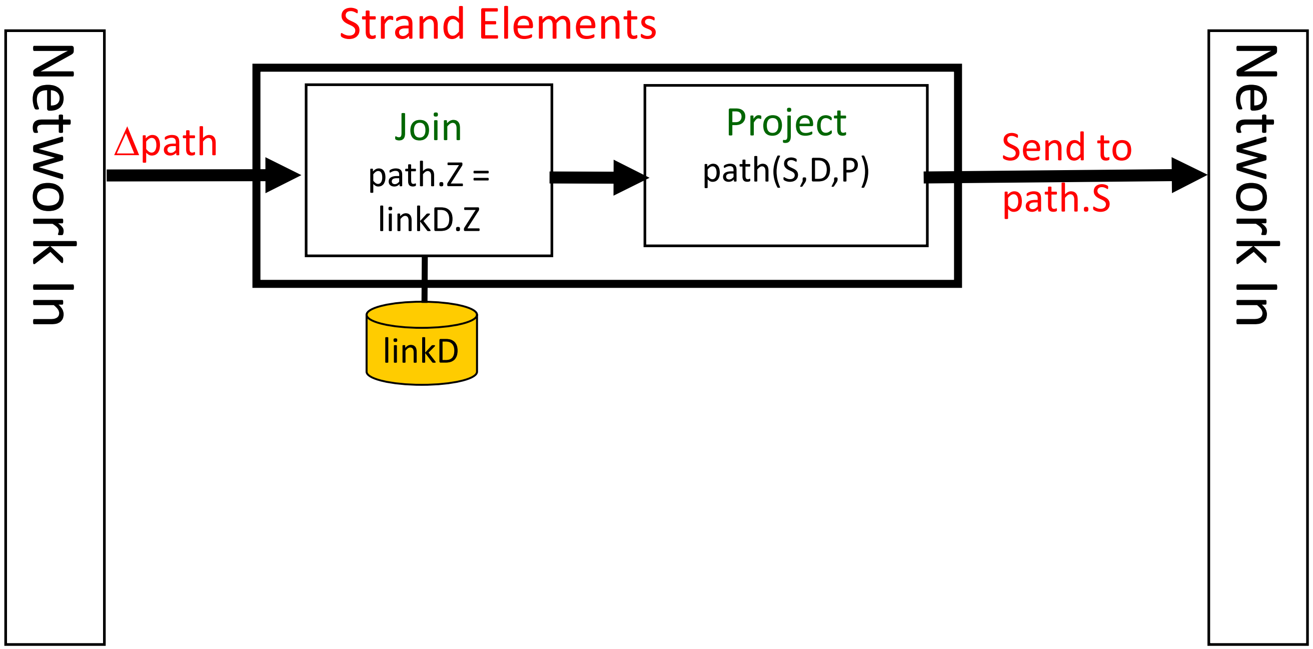
Physical Execution Plan

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P_2), P=S•P_2.



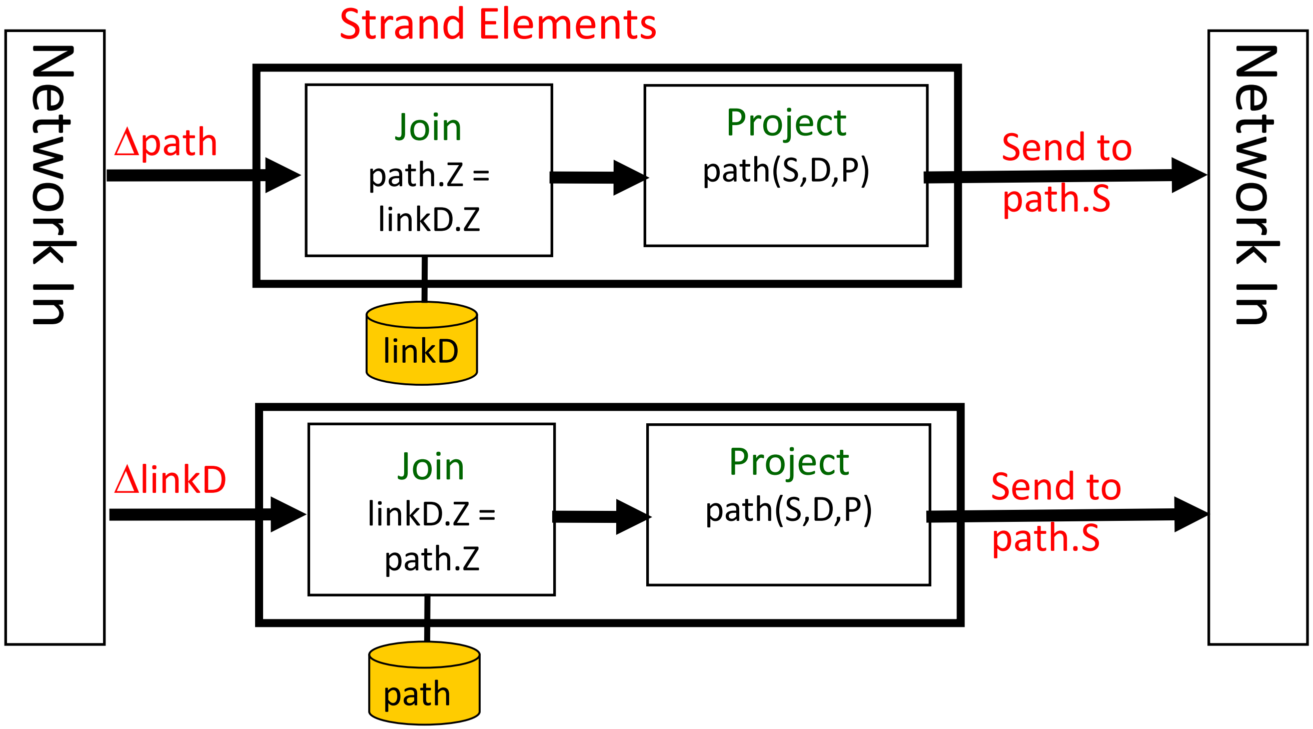
Physical Execution Plan

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P_2), P=S•P_2.



Physical Execution Plan

R2b: path(@S,D,P) <- linkD(S,@Z), path(@Z,D,P_2), P=S•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.

Maintaining Views Incrementally. Gupta, Mumick, Ramakrishnan, Subrahmanian. SIGMOD 1993.

Recent PSN Enhancements

- Provenance-based approach
  - Condensed form of provenance piggy-backed with each tuple for derivability test.
  - **Recursive Computation of Regions and Connectivity in Networks.** Liu, Taylor, Zhou, Ives, and Loo. ICDE 2009.

- Relaxation of FIFO requirements:
  - **Maintaining Distributed Logic Programs Incrementally.**
    Vivek Nigam, Limin Jia, Boon Thau Loo and Andre Scedrov.
    13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP), 2011.

Optimizations

- Traditional:
  - Aggregate Selections
  - Magic Sets rewrite
  - Predicate Reordering

Optimizations

- Traditional:
  - Aggregate Selections
  - Magic Sets rewrite
  - Predicate Reordering

PV/DV → DSR

Optimizations

- Traditional:
  - Aggregate Selections
  - Magic Sets rewrite
  - Predicate Reordering

PV/DV → DSR

- 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

Suggested Readings

- Networking use cases:
  - **Declarative Routing: Extensible Routing with Declarative Queries.** Loo, Hellerstein, Stoica, and Ramakrishnan. SIGCOMM 2005.
  - **Implementing Declarative Overlays.** Loo, Condie, Hellerstein, Maniatis, Roscoe, and Stoica. SOSP 2005.

- Distributed recursive query processing:
  - ***Declarative Networking: Language, Execution and Optimization**. Loo, Condie, Garofalakis, Gay, Hellerstein, Maniatis, Ramakrishnan, Roscoe, and Stoica, SIGMOD 06.
  - **Recursive Computation of Regions and Connectivity in Networks.** Liu, Taylor, Zhou, Ives, and Loo. ICDE 2009.

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

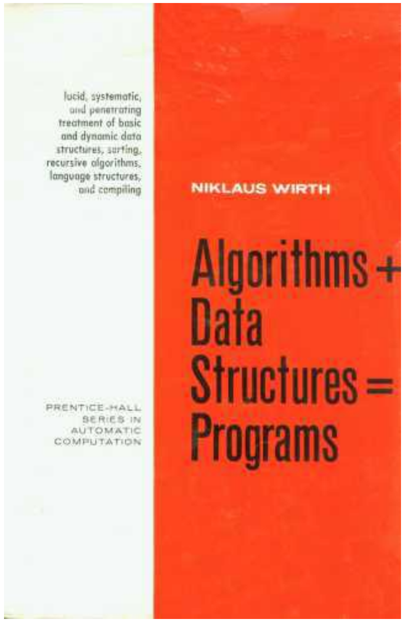
Outline of Tutorial

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



lucid, systematic,
and penetrating
treatment of basic
and dynamic data
structures, sorting,
recursive algorithms,
language structures,
and compiling

PRENTICE-HALL
SERIES IN
AUTOMATIC
COMPUTATION

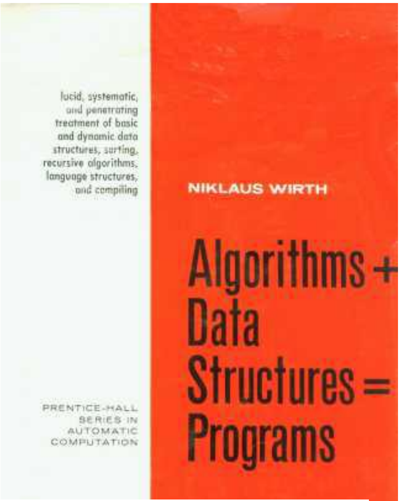
NIKLAUS WIRTH

Algorithms +
Data
Structures =
Programs

What Is A Program?

program = algorithms + data structures

algorithm = logic + control

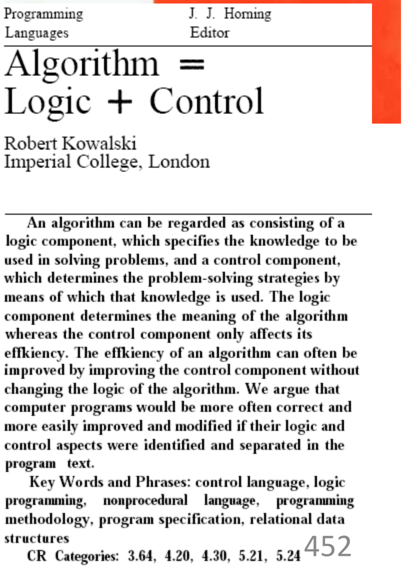


lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling

NIKLAUS WIRTH

Algorithms + Data Structures = Programs

PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION



Programming Languages | J. J. Horning Editor

Algorithm = Logic + Control

Robert Kowalski
Imperial College, London

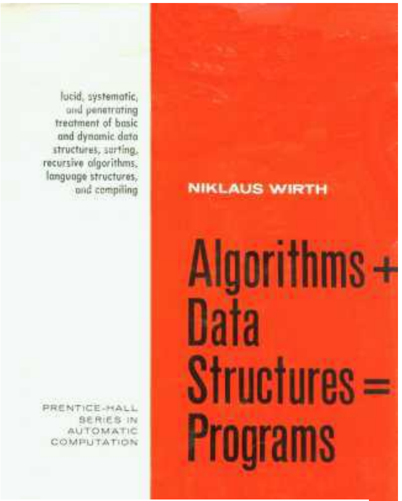
An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used. The logic component determines the meaning of the algorithm whereas the control component only affects its effkiency. The effkiency 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

CR Categories: 3.64, 4.20, 4.30, 5.21, 5.24

What Is A Program?

program = logic + control + data structures

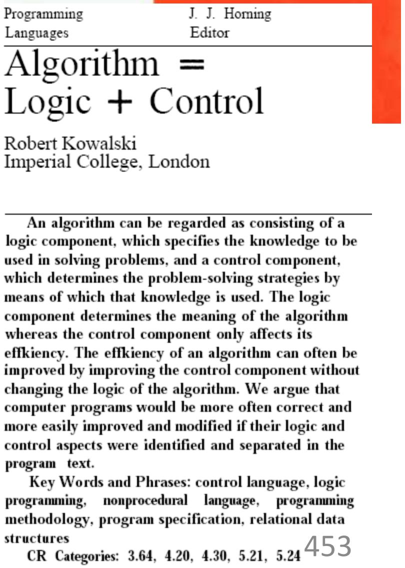


lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling

NIKLAUS WIRTH

Algorithms + Data Structures = Programs

PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION



Programming Languages

J. J. Horning Editor

Algorithm = Logic + Control

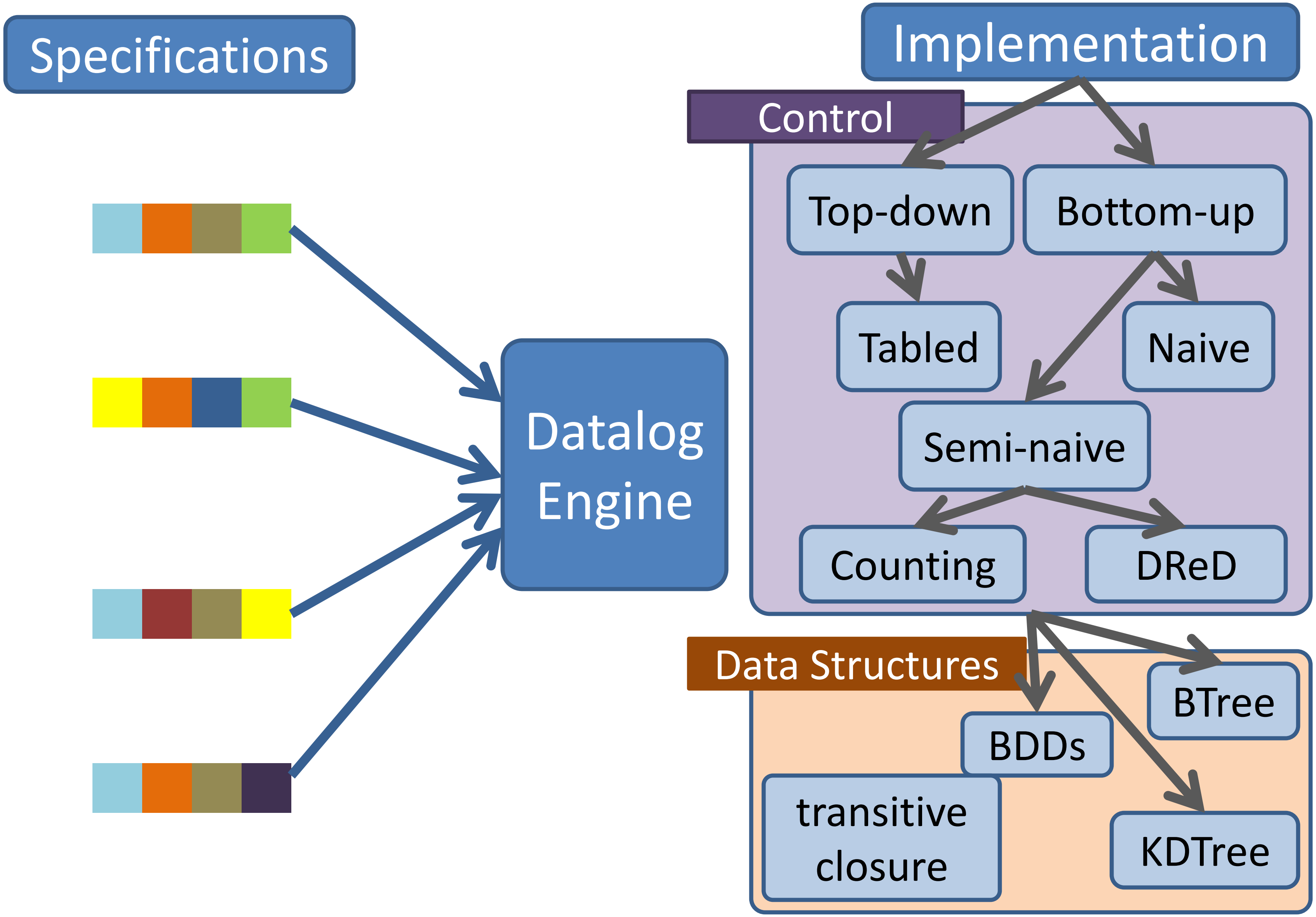
Robert Kowalski
Imperial College, London

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 effkiency. The effkiency 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

CR Categories: 3.64, 4.20, 4.30, 5.21, 5.24

Logic + Control + Data Structures



THE END… OR IS IT THE BEGINNING?