Kaituo Li
U. Massachusetts, Amherst

Christoph Reichenbach
Goethe U. Frankfurt

Yannis Smaragdakis
U. Athens

Yanlei Diao
U. Massachusetts, Amherst

Christoph Csallner
U. Texas Arlington

SEDGE: Symbolic Example Data Generation for Dataflow Programs

Motivation


support
cloud
storage
Database
analize
information
tools
statistics
Big Data
mobile
processing
terabytes
NoSQL
compression

SEDGE: Symbolic Example Data GEnerator

- **Problem**: Generate **fewest** test cases to exercise **all** key behaviors of operators in a dataflow program
  - e.g. both passing and failing a filter
- **Our approach**: First dynamic-symbolic (aka “concolic”) testing engine for a dataflow language
- **Results**:
  - Improved coverage and running time over industrial state-of-the-art
    - e.g. Pig Latin “illustrate”, SIGMOD ’09 best paper award

Overview of Talk

- Background: Dataflow languages
- Metrics
- Our algorithm in action
- Comparison with state-of-the-art

Example Dataflow Program

- Input: Galaxy profiles, Star profiles
- Find stars with a surface brightness less than 100 from galaxies with a squared ellipticity > 0.25
  - $ellipticity^2 = g\_r^2 + u\_r^2$

In Pig Latin

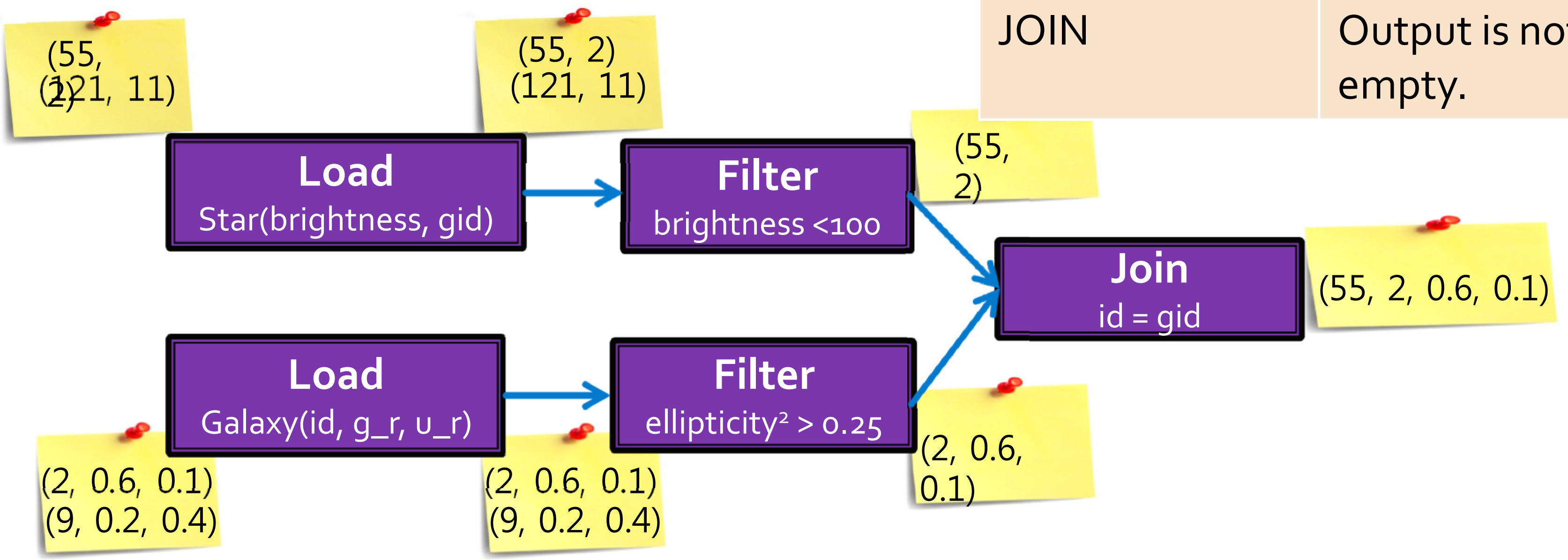
```
A=LOAD 'Star' using PigStorage  AS (brightness:int, gid:int);
B=LOAD 'Galaxy' using PigStorage AS (id: int, g_r, u_r:double);
C=FILTER A BY brightness < 100;
D=FILTER B BY power(g_r, 2) + power(u_r, 2) > 0.25;
E=JOIN C ON gid, D ON id;
```

Goal: Coverage with Fewest Tests

- *Completeness*
  - similar to “branch coverage” in imperative languages
- *Conciseness*
  - with as few tests as possible

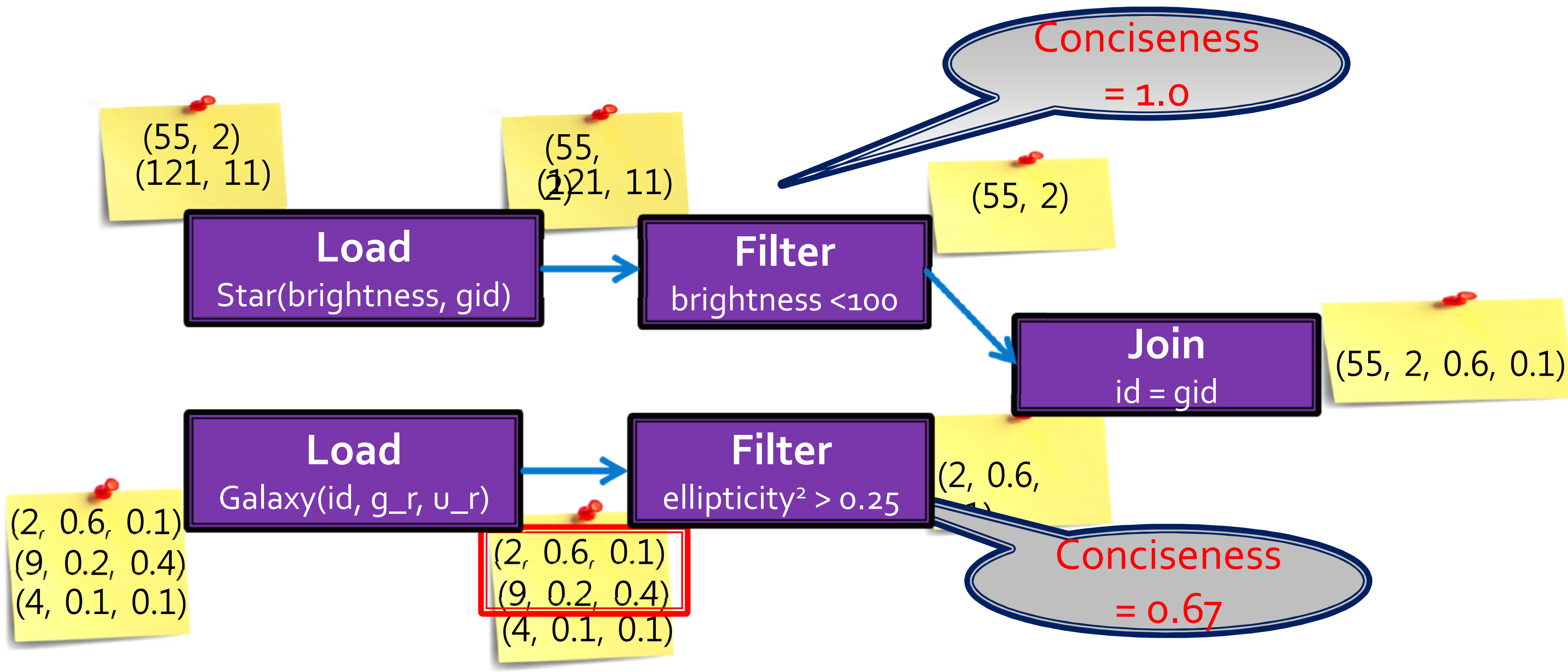
But What Is Coverage?

Operator	Coverage
LOAD	Input is not empty.
FILTER	**(a)** one record that passes the filter; **(b)** one that does not pass.
JOIN	Output is not empty.



Fewest tests

- Conciseness = $\frac{\text{\# coverage requirements}}{\text{\# test cases}}$



Our Algorithm

- **Dynamic execution**
- **Pruning**
- **Symbolic synthesis**

Our Algorithm

Dynamic execution

Pruning

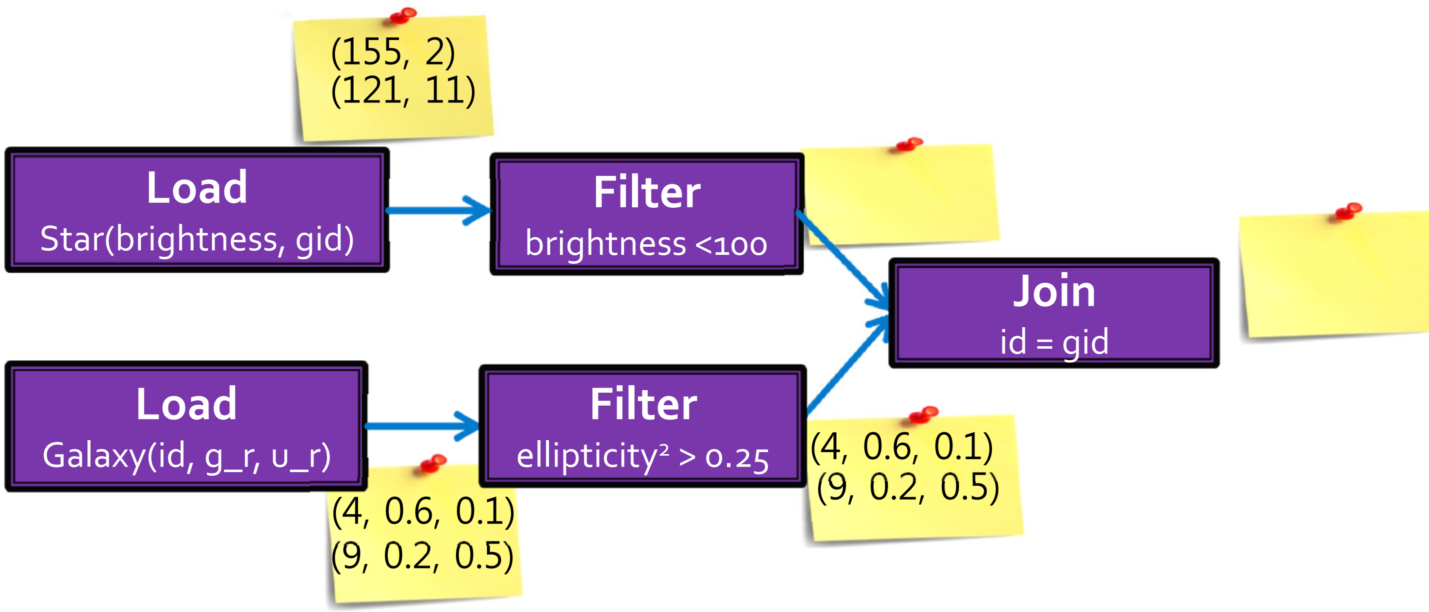
Symbolic synthesis

Run sampled existing data

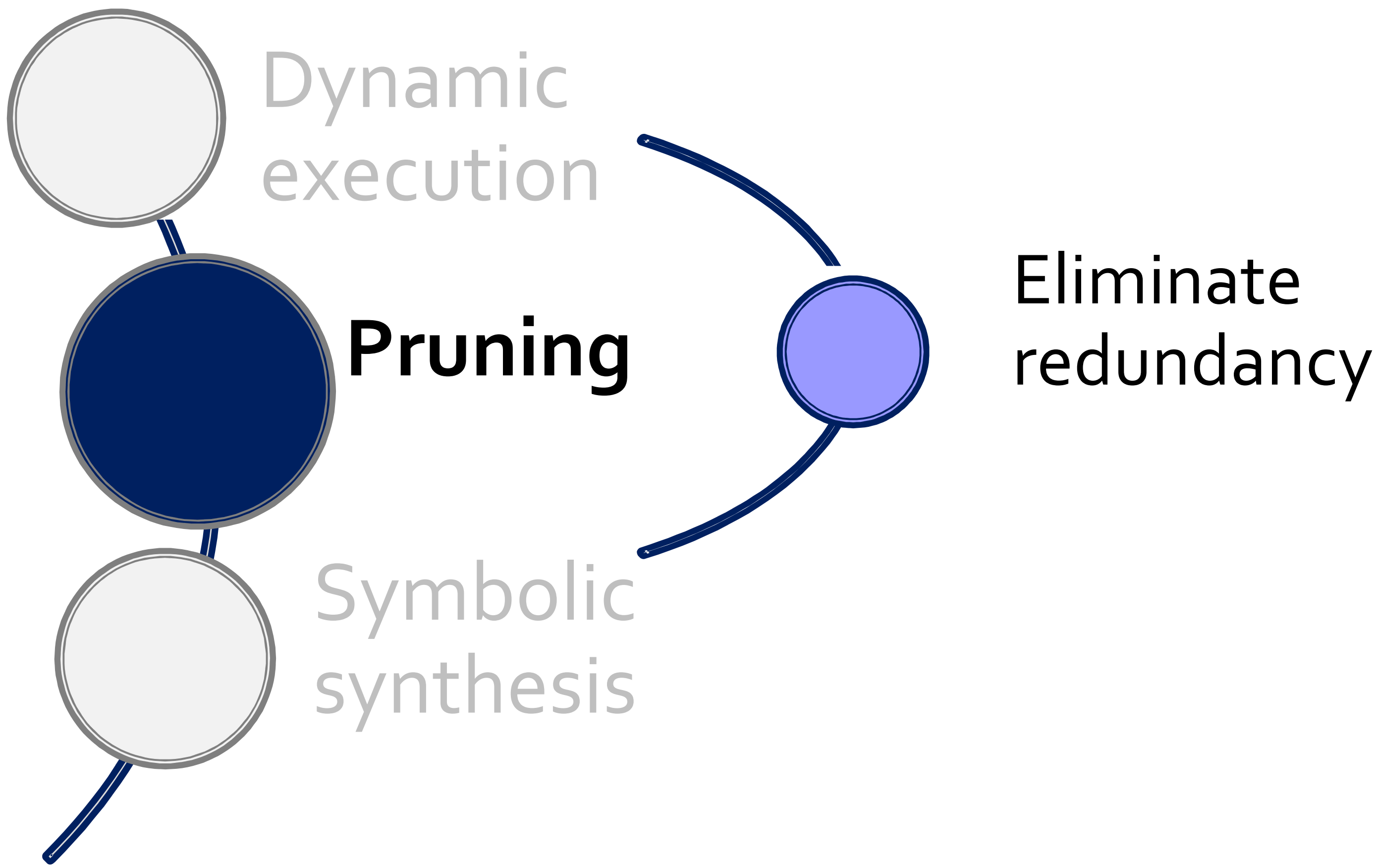
Cache concrete results of user-defined functions (UDF) across runs (if any)

Dynamic Execution

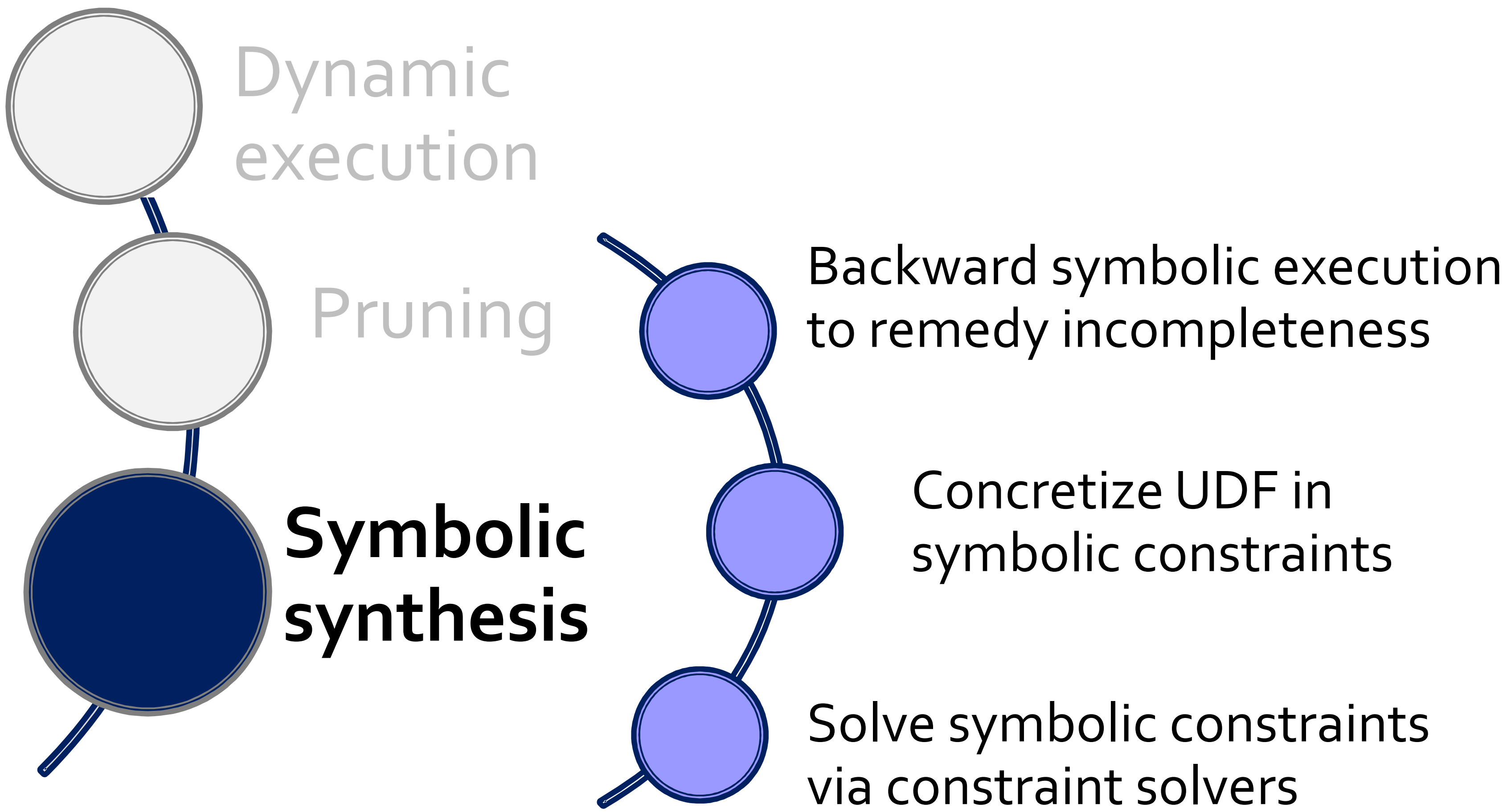
UDF	Parameters	Return
power	(0.6, 2)	0.36
power	(0.1, 2)	0.01
power	(0.2, 2)	0.04
power	(0.5, 2)	0.25



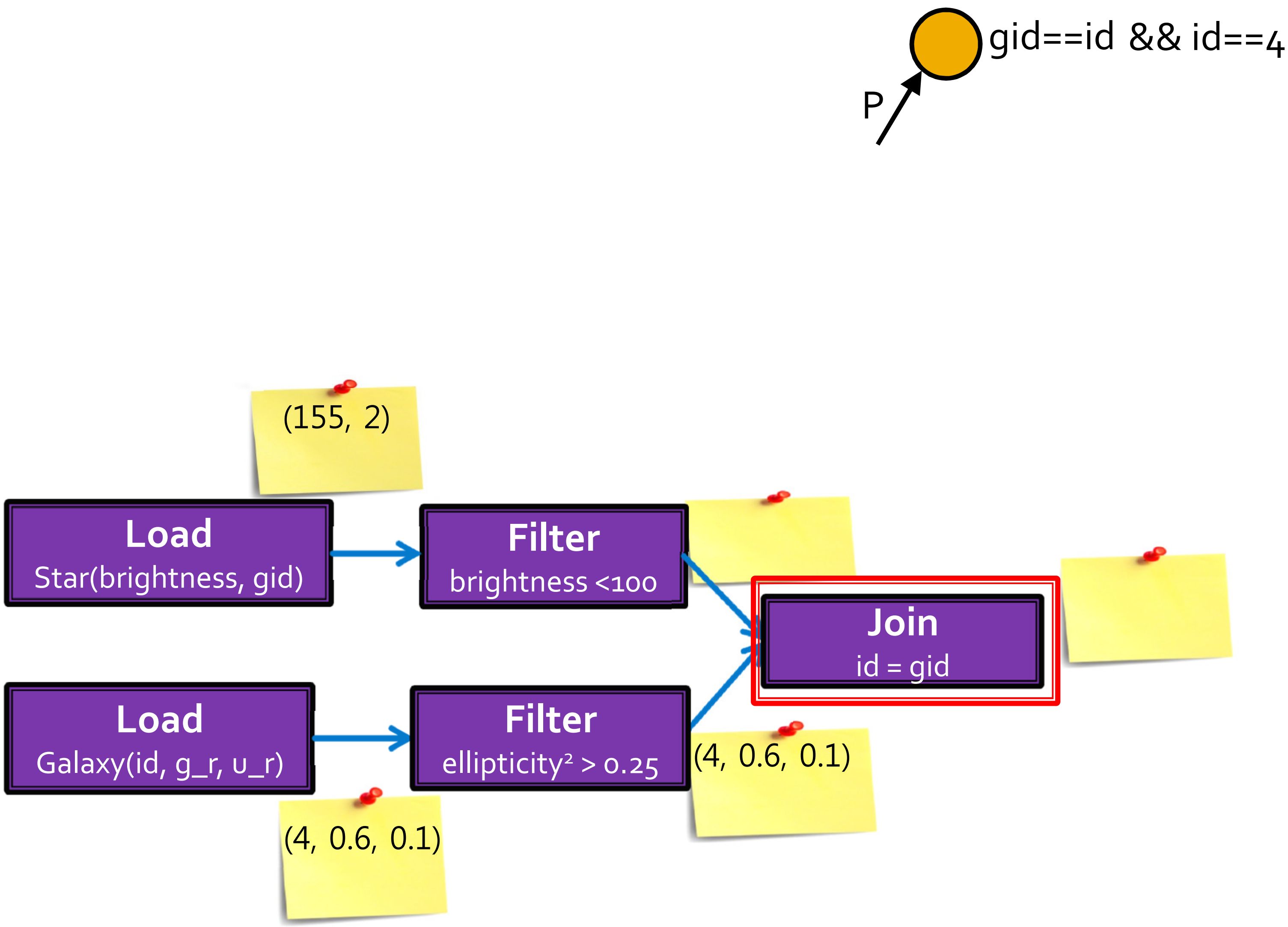
Our Algorithm



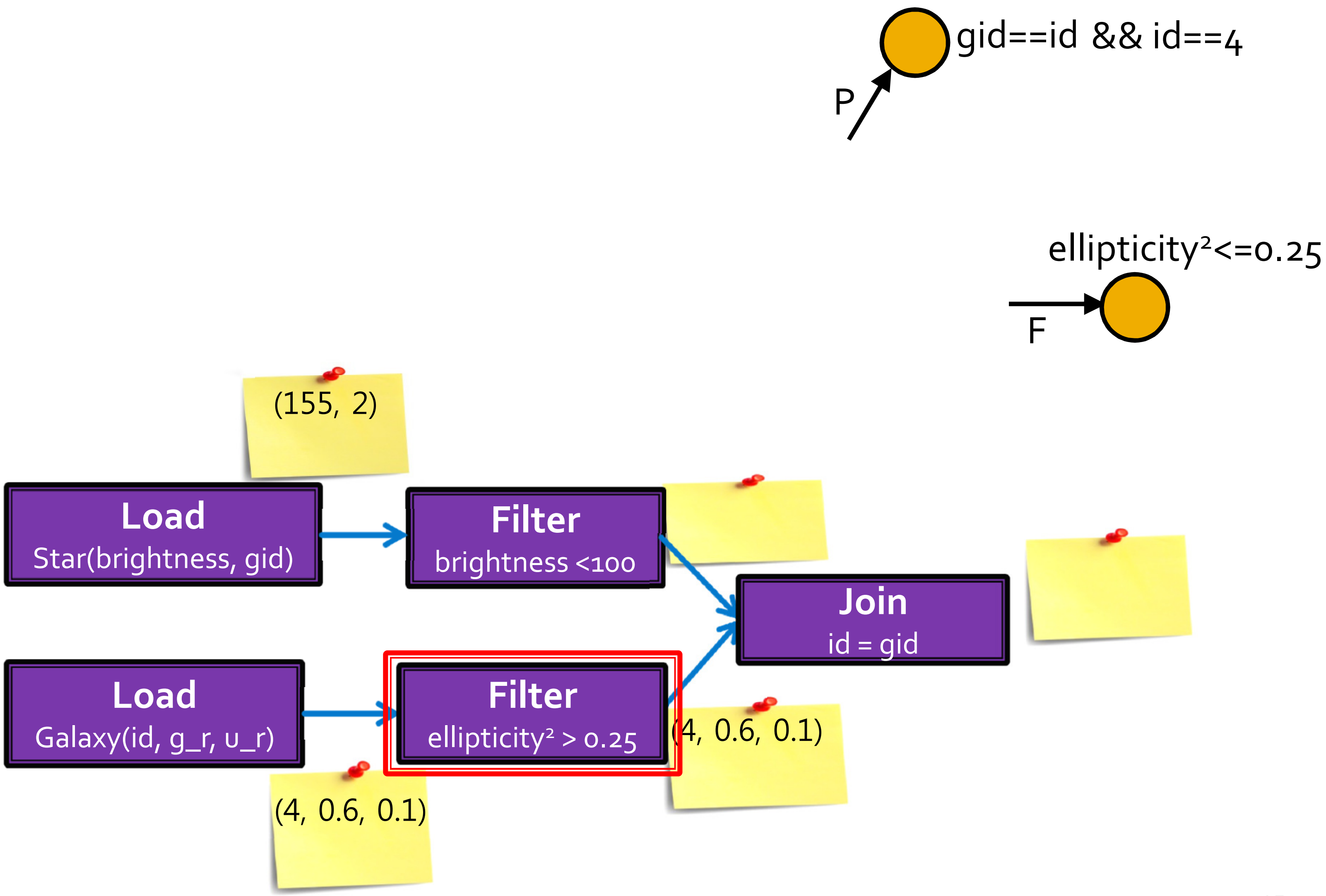
Our Algorithm



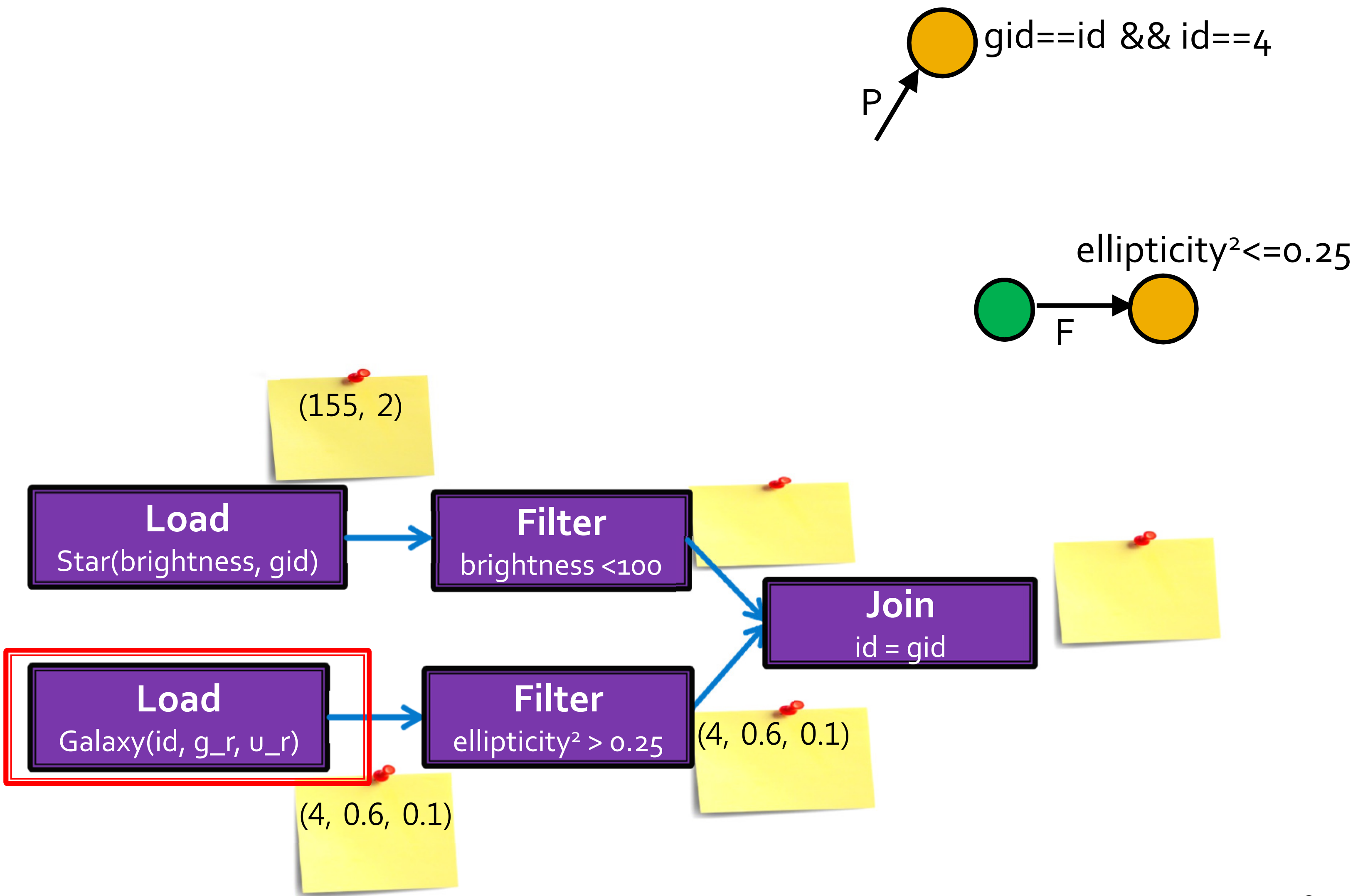
Backward Symbolic Execution



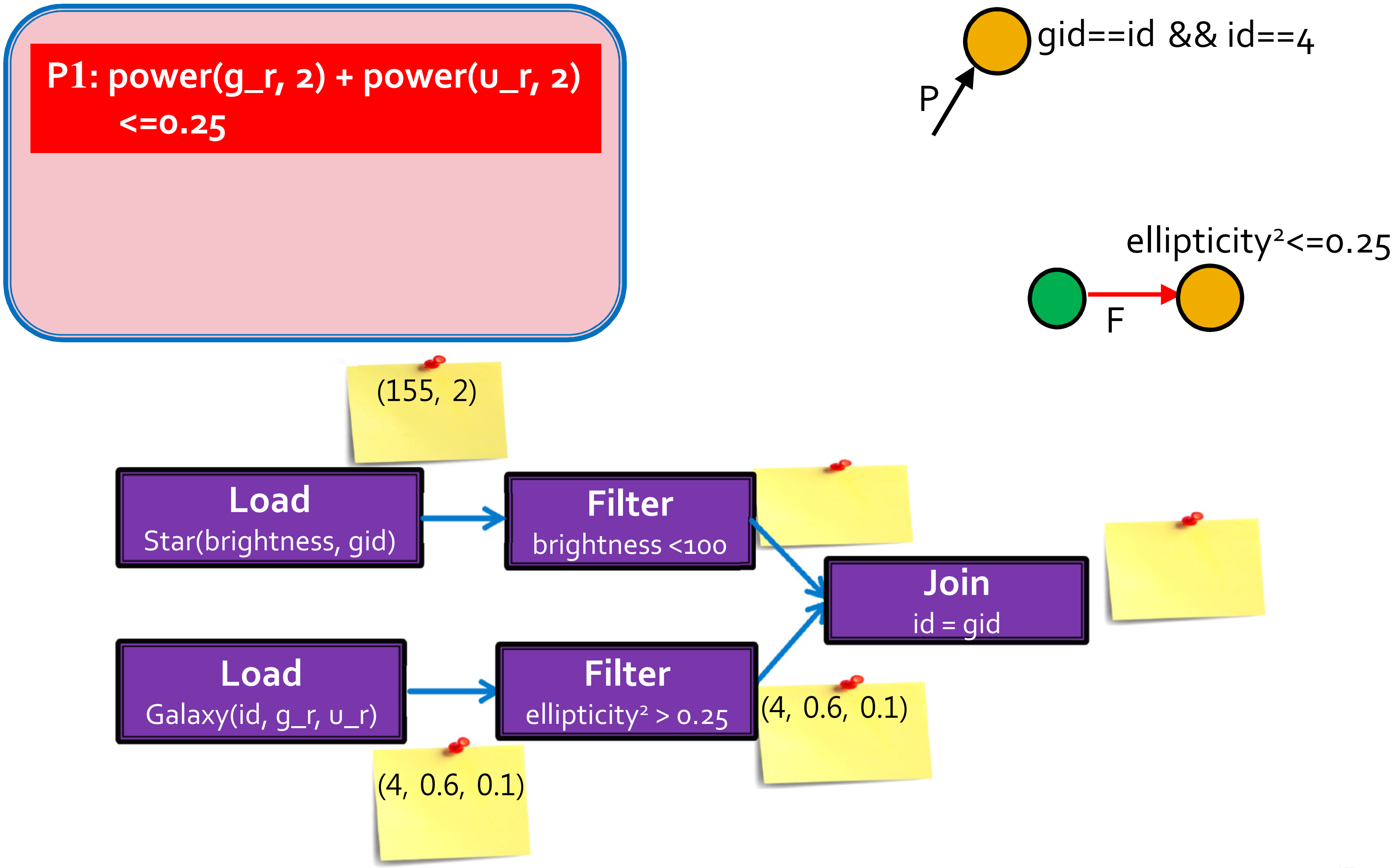
Backward Symbolic Execution



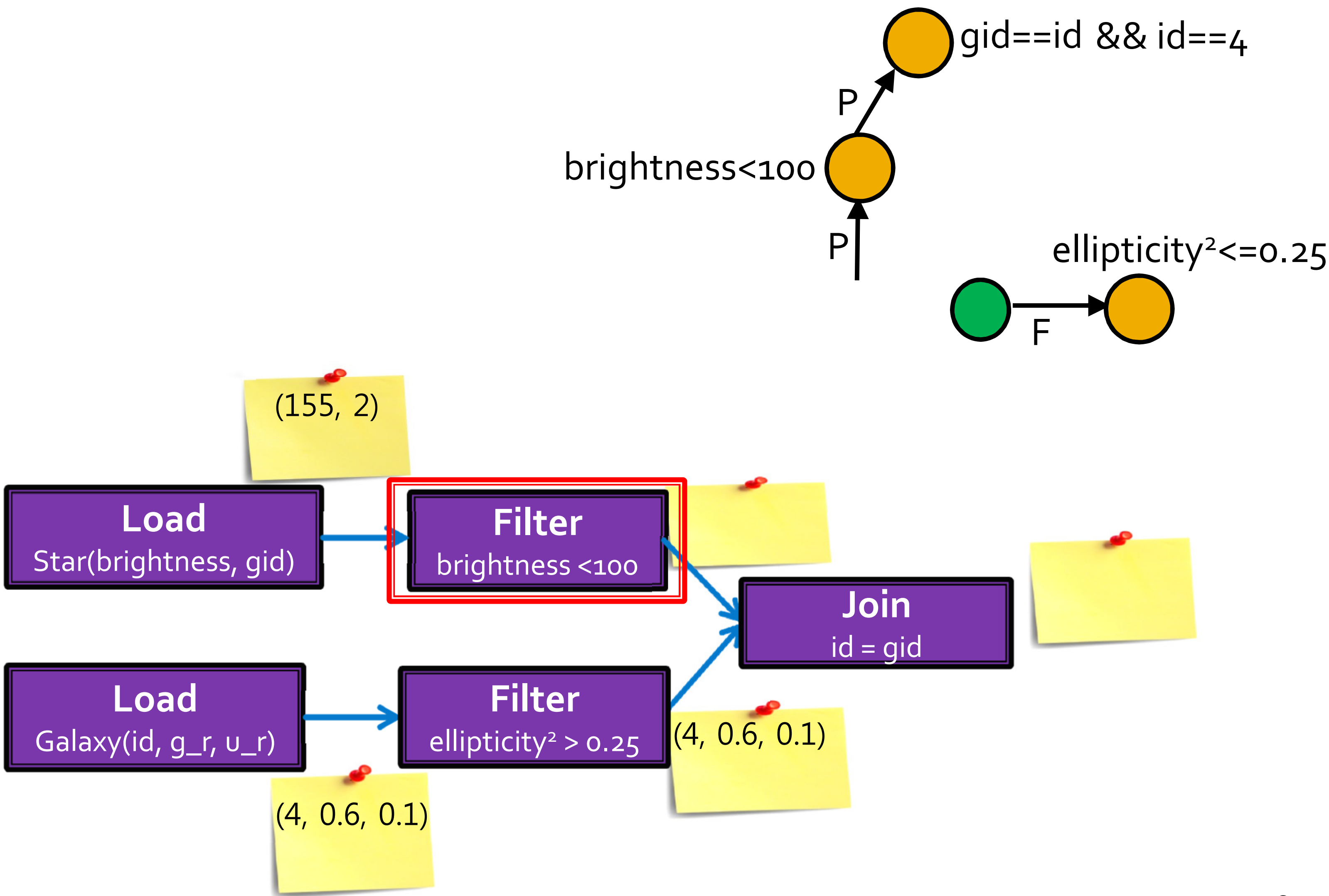
Backward Symbolic Execution



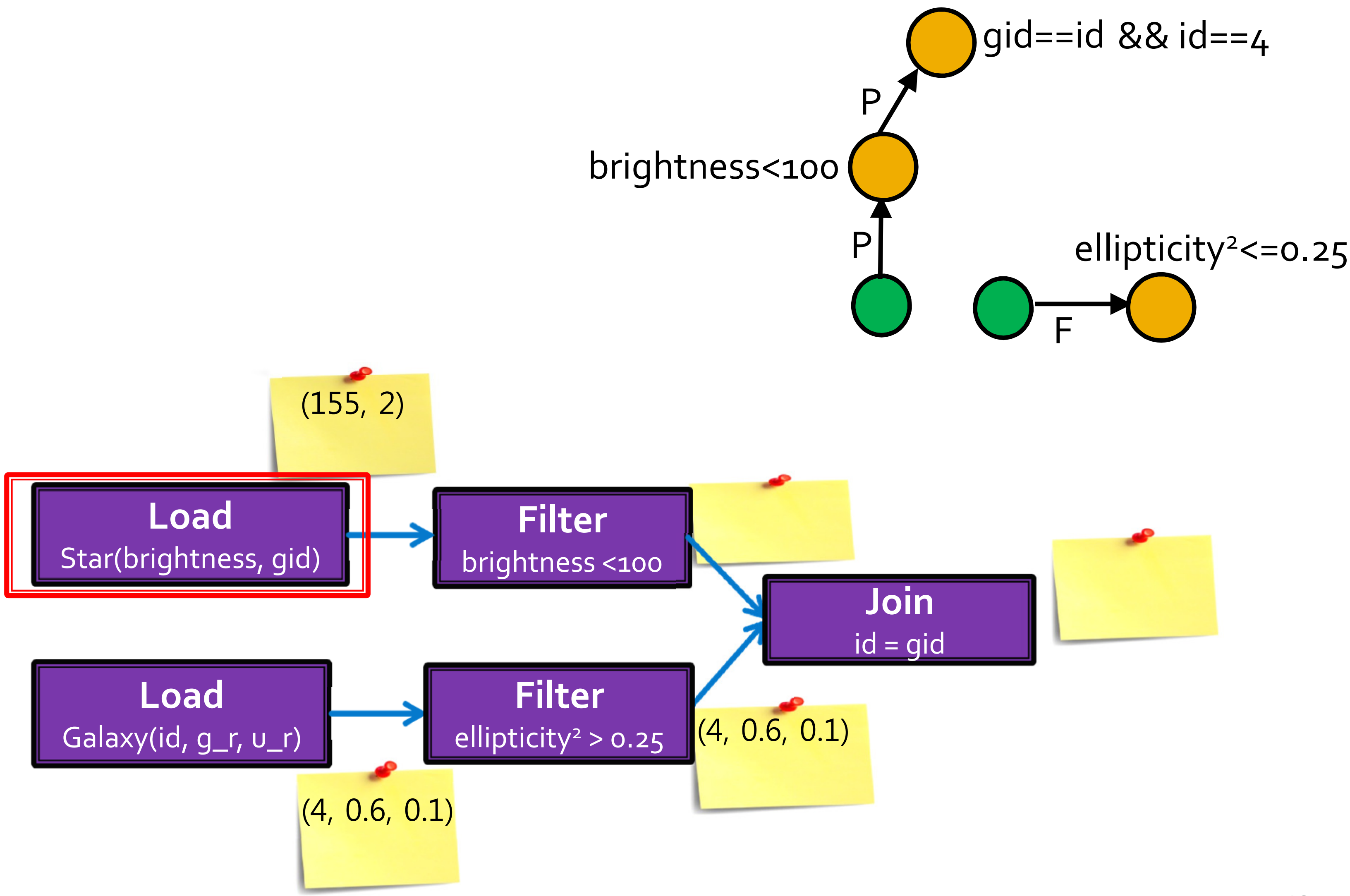
Constraint Generation



Backward Symbolic Execution

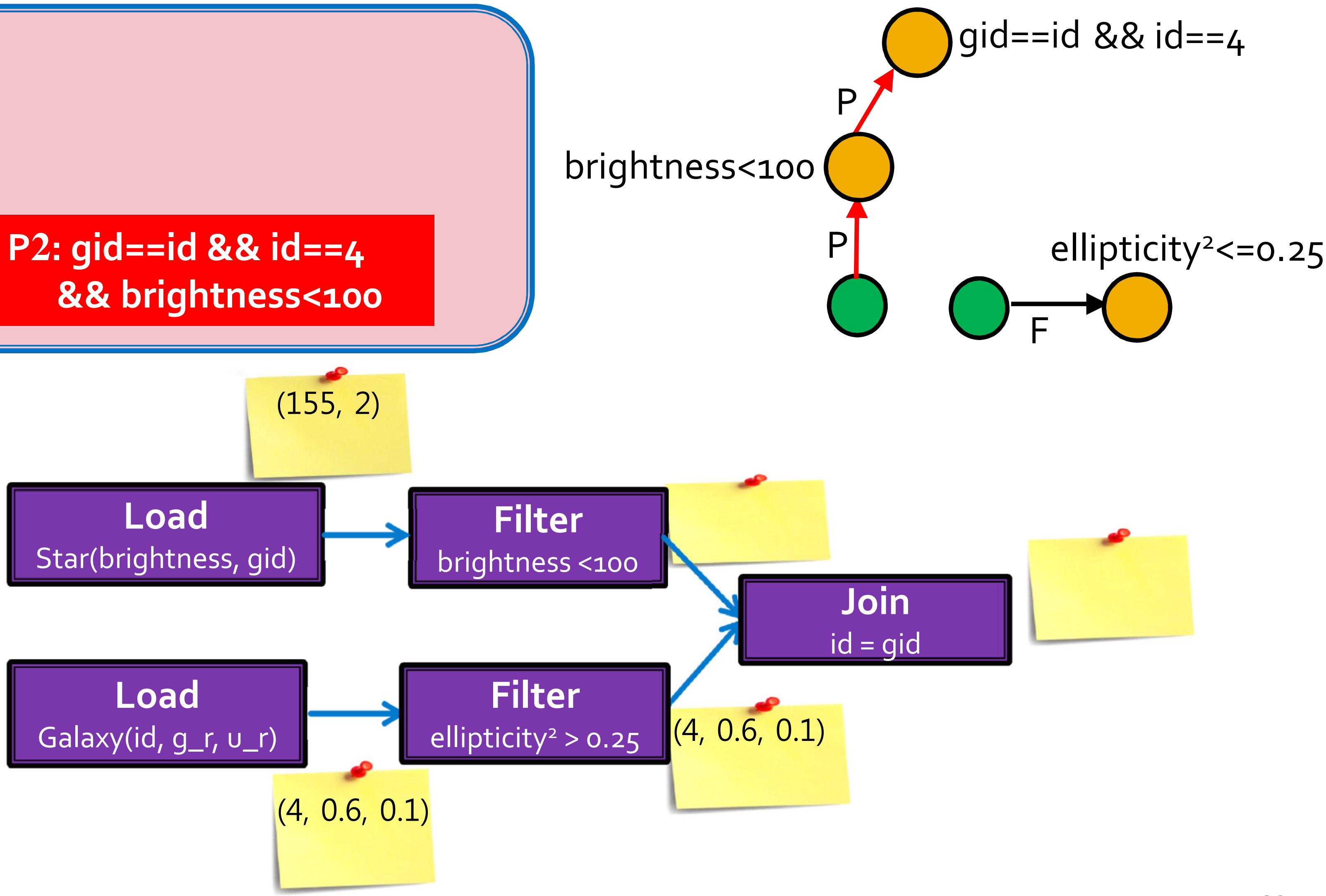


Backward Symbolic Execution



Constraint Generation

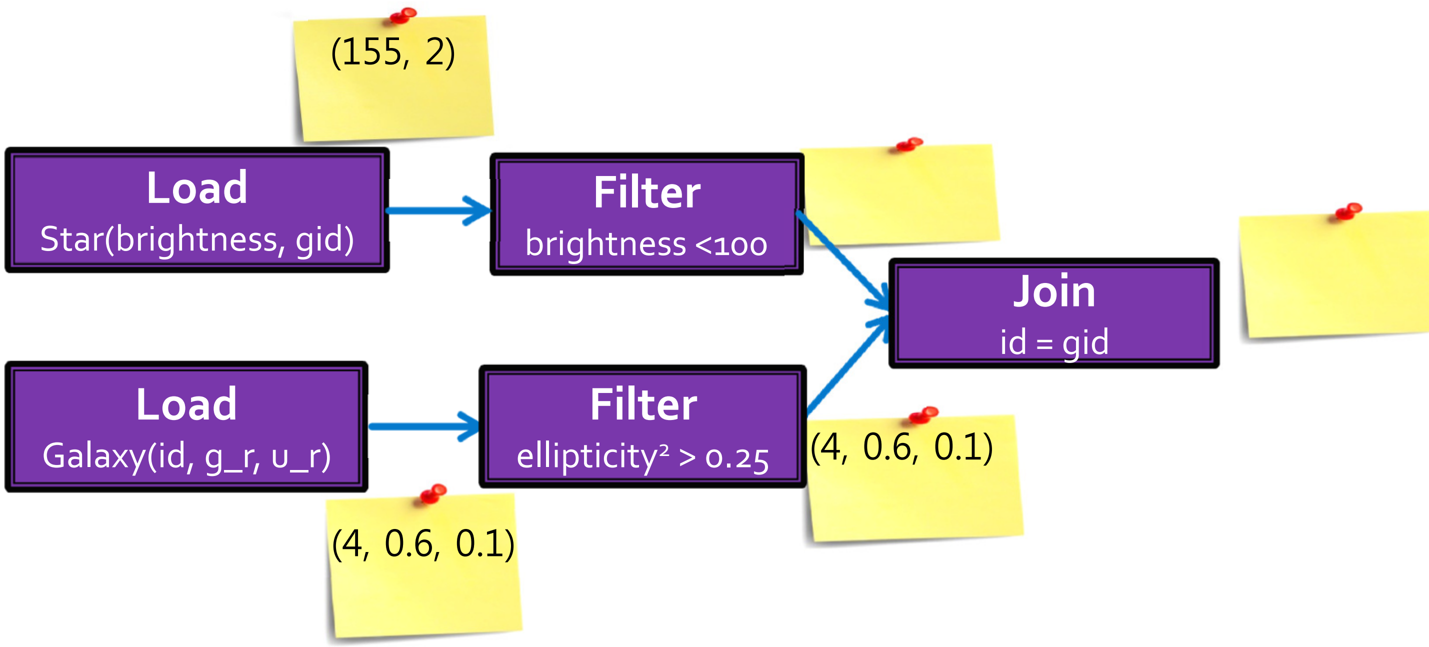
P2: gid==id && id==4
&& brightness<100



Concretization

P1: power(g_r, 2) + power(u_r, 2) <=0.25

P2: gid==id && id==4 && brightness<100



Concretization

Model the behavior of a user-defined function as its input-output behavior across all observed test cases.

UDF	Parameters	Return
power	(0.6, 2)	0.36
power	(0.1, 2)	0.01
power	(0.2, 2)	0.04
power	(0.5, 2)	0.25

Uninterpreted function

```
(define-fun power ((u Double) (v Int)) Double
 (ite (and (= u 0.6) (= v 2) 0.36
 (ite (and (= u 0.1) (= v 2) 0.01
 (ite (and (= u 0.2) (= v 2)  0.04
 (ite (and (= u 0.5) (= v 2) 0.25
  0)))
    (declare-const x Double)
    (declare-const y Int)
    (assert (not (= (power x y) 0)))
```

Assumption

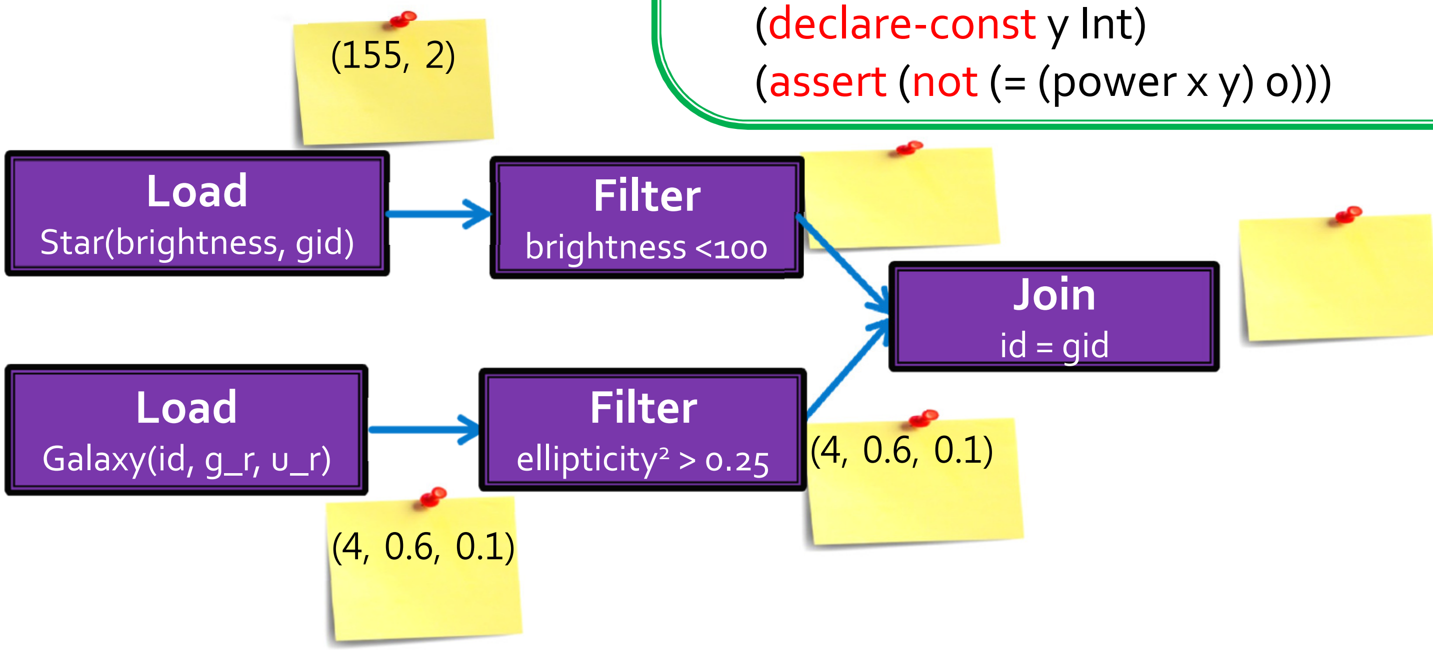
UDFs return values only depend on argument value(s) in dataflow programs.

Concretization

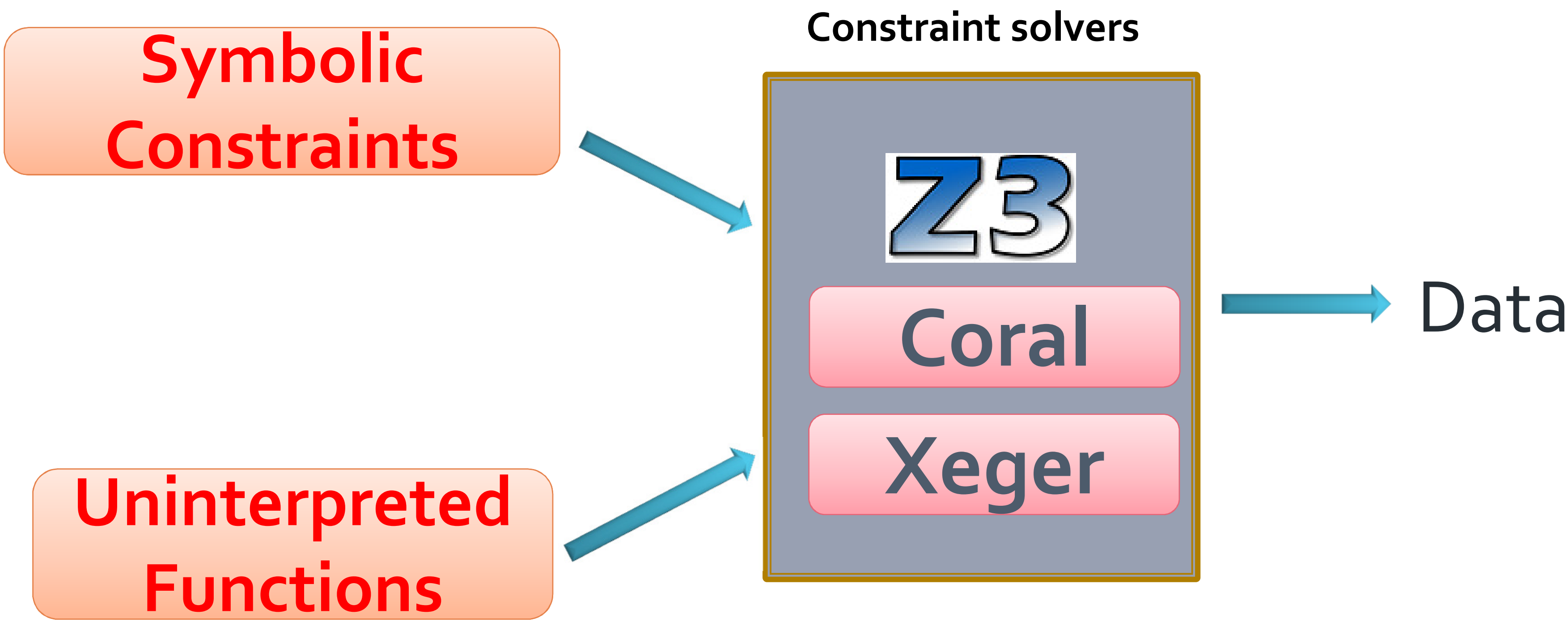
P1: gid==id && id==4
&& brightness<100

P2: power(g_r, 2) + power(u_r, 2)
<=0.25

```
(define-fun power ((u Double) (v Int)) Double
  (ite (and (= u 0.6) (= v 2) 0.36
  (ite (and (= u 0.1) (= v 2) 0.01
  (ite (and (= u 0.2) (= v 2)  0.04
  (ite (and (= u 0.5) (= v 2) 0.25
    0)))
      (declare-const x Double)
      (declare-const y Int)
      (assert (not (= (power x y) 0)))
```



Constraint Solving

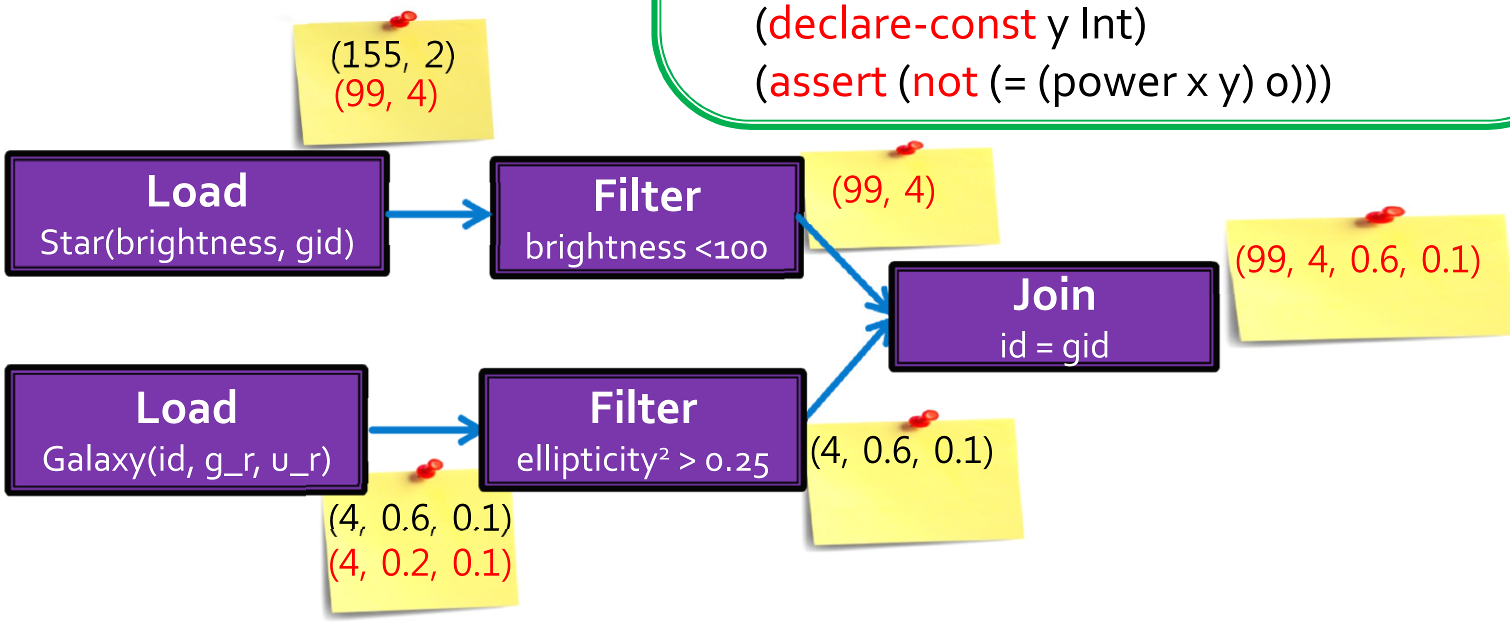


Constraint Solving

P1: gid==id && id==4 && brightness<100

P2: power(g_r, 2) + power(u_r, 2) <=0.25

```
(define-fun power ((u Double) (v Int)) Double
 (ite (and (= u 0.6) (= v 2) 0.36
 (ite (and (= u 0.1) (= v 2) 0.01
 (ite (and (= u 0.2) (= v 2)  0.04
 (ite (and (= u 0.5) (= v 2) 0.25
  0)))
   (declare-const x Double)
   (declare-const y Int)
   (assert (not (= (power x y) 0)))
```



Experiments

- Two benchmark suites
  - PigMix: 20 representative Pig programs from Pig community
  - SDSS: 11 Pig programs hand-translated from sample SQL queries
    - from the Sloan Digital Sky Survey (astronomy DB)

SEDGE vs. Pig Latin Illustrate

Pig Latin Illustrate

- Cannot handle UDFs
- Poor constraint solving ability
- Generate local constraint without looking ahead

SEDGE

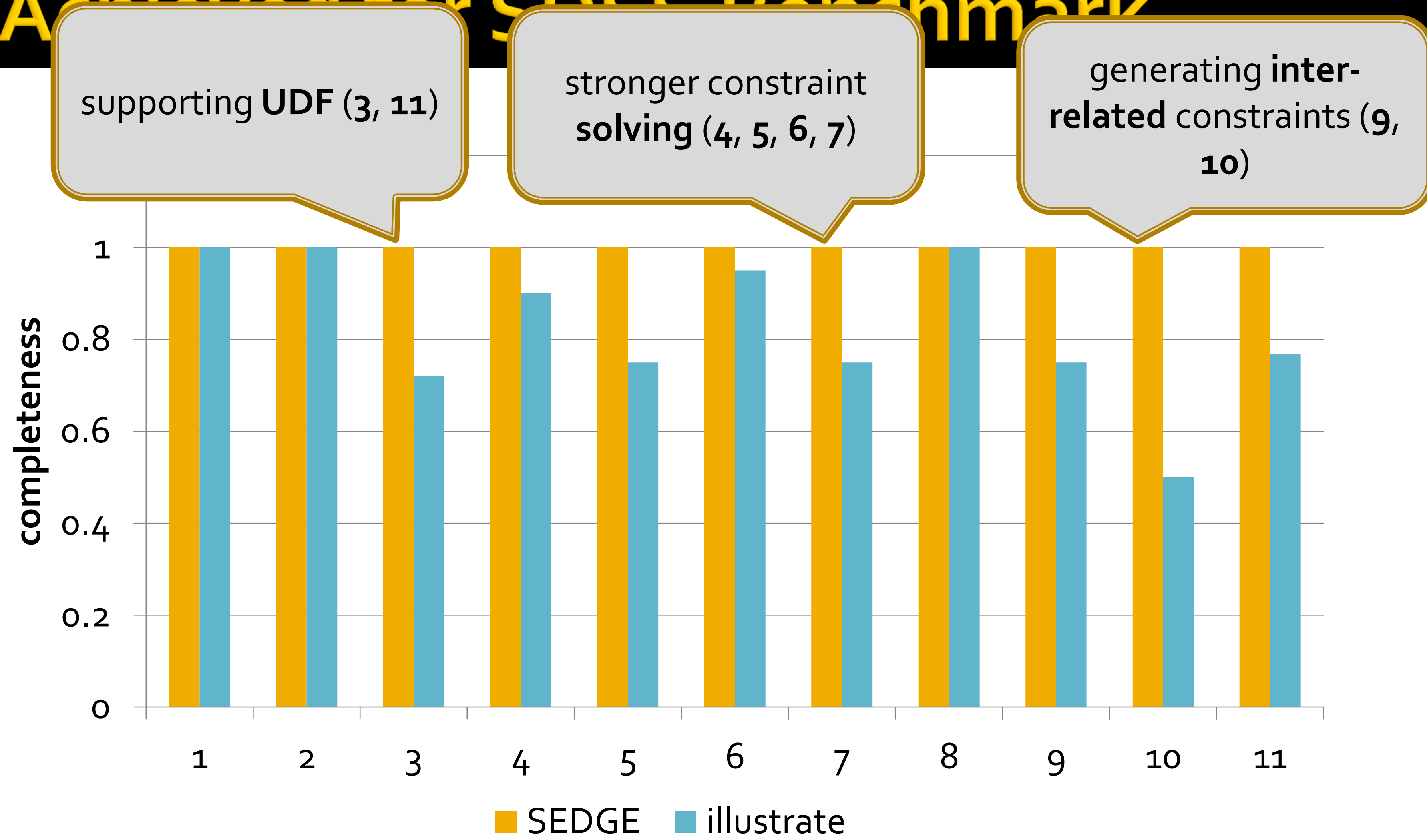
- Support UDFs
- Stronger constraint solving
- Generate inter-related constraints

Pig Latin Illustrate is the current state-of-the-art in example data generation for dataflow programs .

- Industrial tool ("illustrate" functionality in Pig, by Yahoo)
- SIGMOD '09 best paper

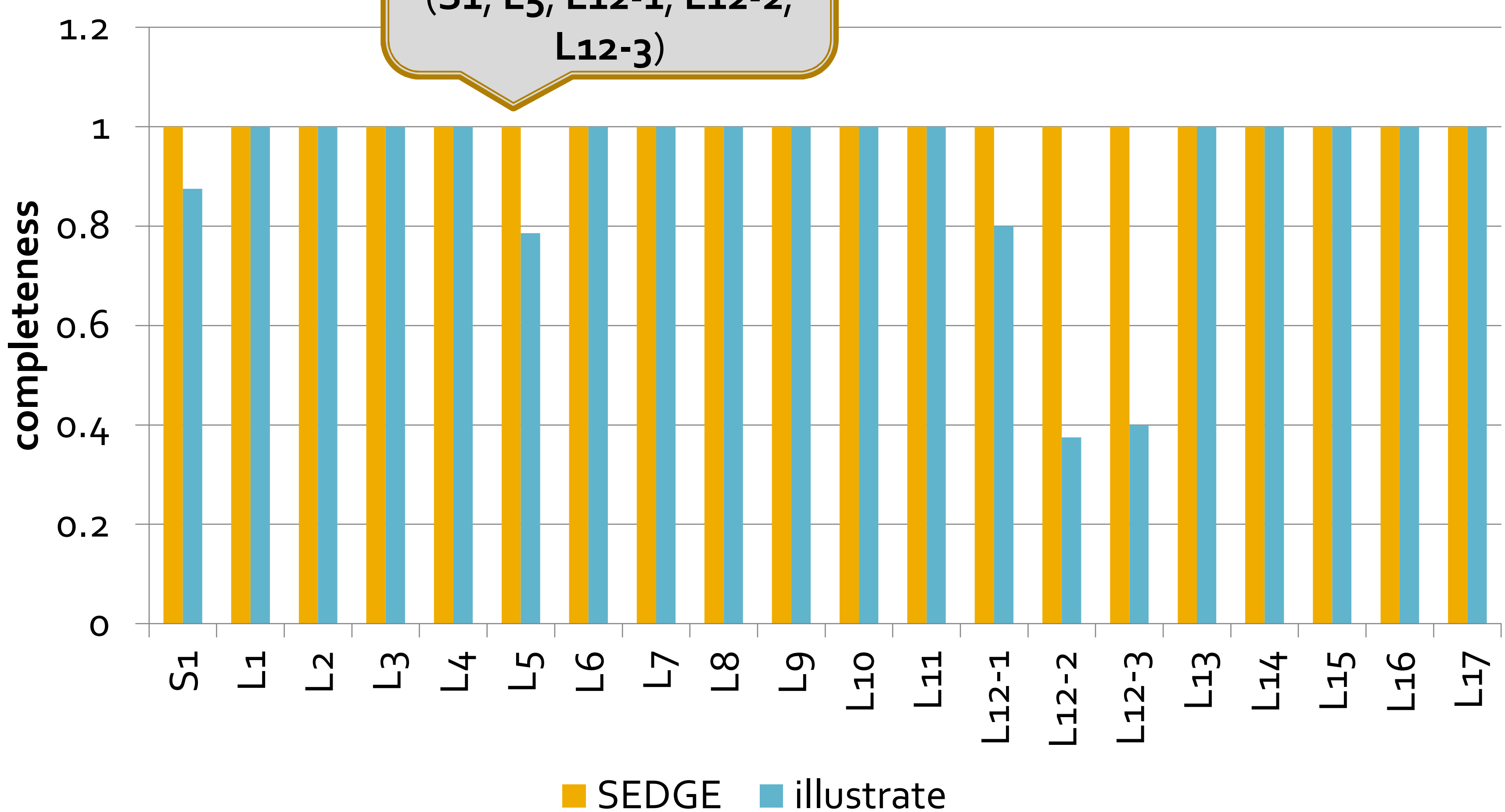
SEDGE can generate example data for dataflow programs better (completeness) and cheaper (running time)

Completeness Comparison Achieved for SDSS Benchmark

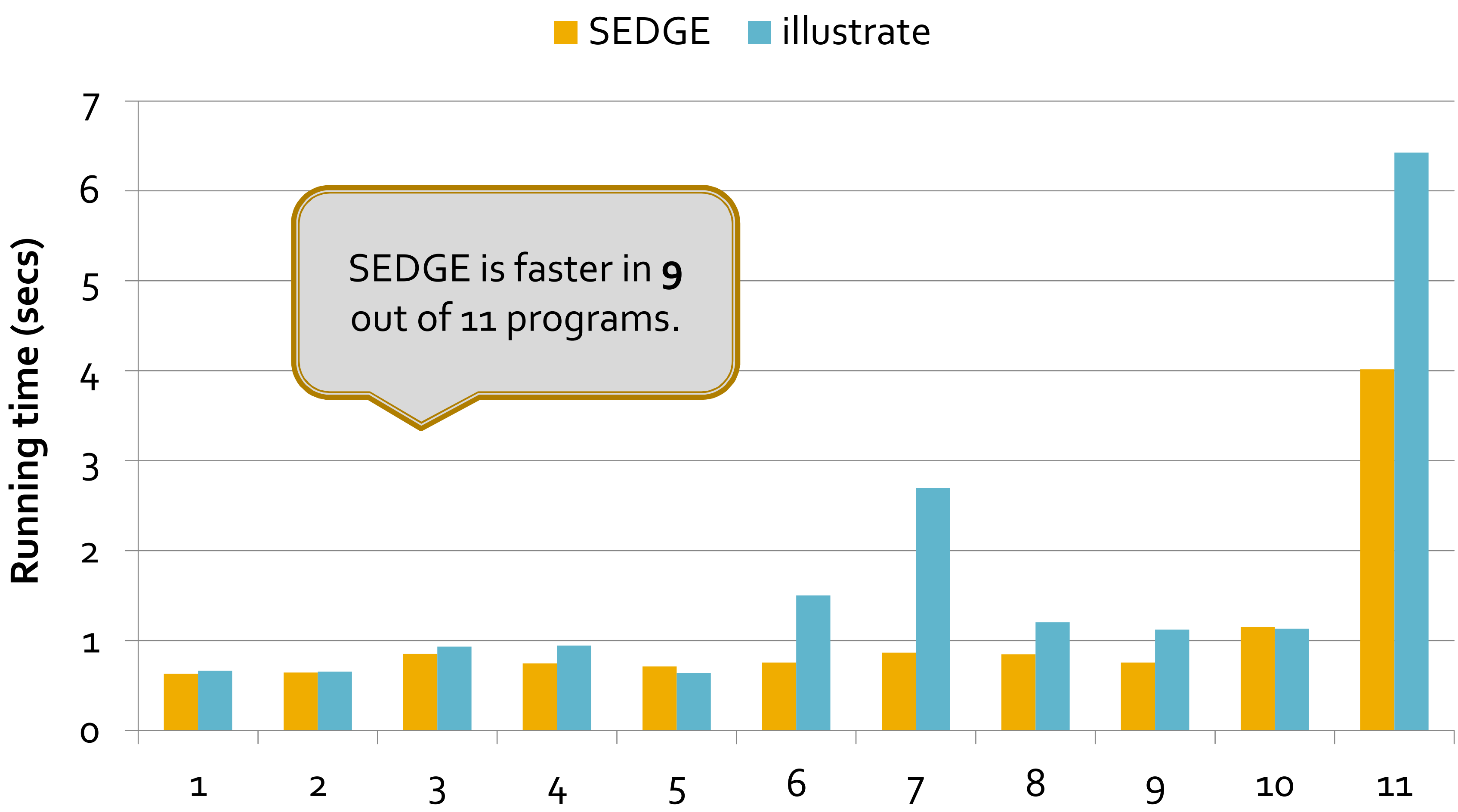


Completeness Comparison Achieved for BigMix Benchmark

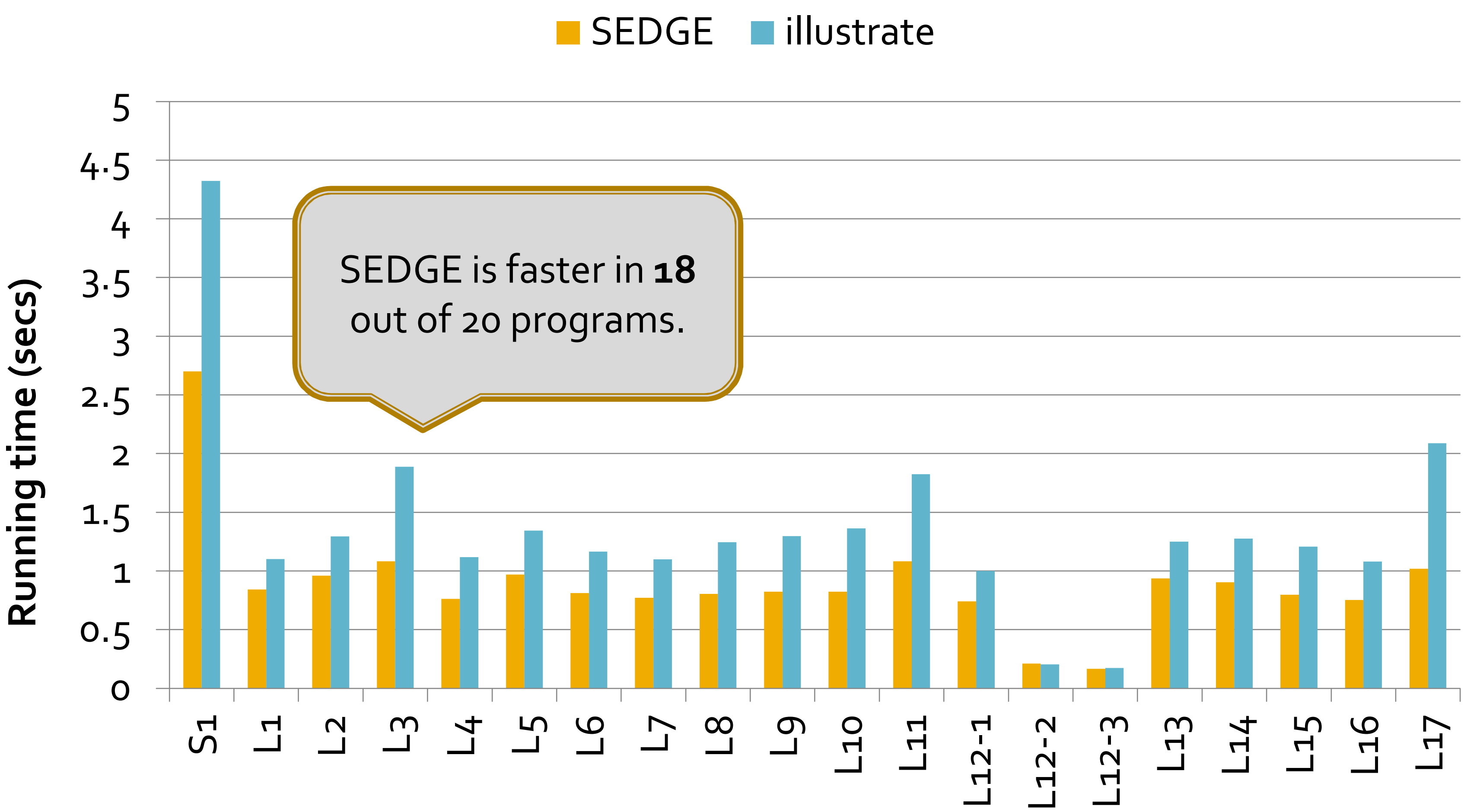
generating **inter-related** constraints (**S1**, **L5**, **L12-1**, **L12-2**, **L12-3**)



Timing Comparison for SDSS Benchmark



Timing Comparison for PigMix Benchmark



Summary

- Created the first dynamic-symbolic (aka "concolic") testing engine for dataflow languages
- Suggested the use of concrete results across runs of a UDF to represent the UDF
- Proposed an approach that balances completeness, conciseness, and running time