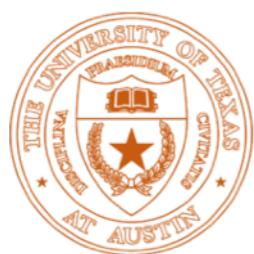


Formal Methods for Database Application Evolution

İşıl Dillig
University of Texas at Austin



THE UNIVERSITY OF
TEXAS
AT AUSTIN



VSTTE 2020

Database Applications



DB application: program that interacts with underlying database

Database applications are ubiquitous

- Enterprise software, web applications, CRM applications,

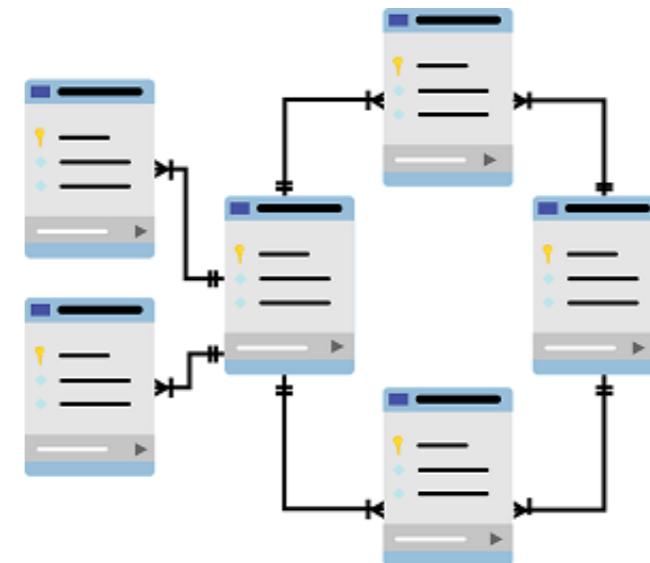


Schema Refactoring

Common theme during DB app evolution:
schema refactoring

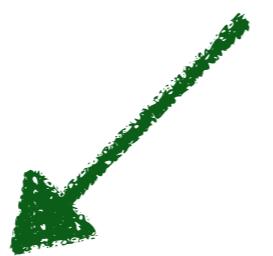
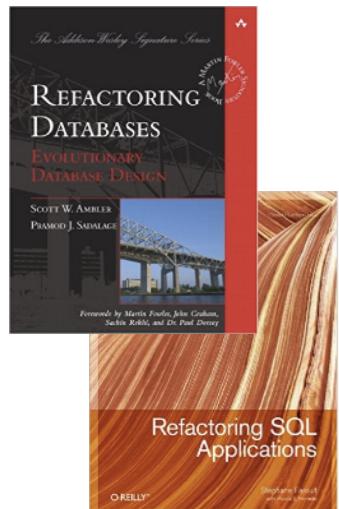
Examples

- Splitting/merging tables
- Denormalization
- Even more extreme: Switch from relational schema to non-relational DB



Implications of Schema Changes

Structural schema changes have significant implications!



Data migration:
Move data from source
to target schema

Code migration:
Must re-implement
parts of program

Attracted lots
of attention

Currently done
manually!

Our Recent Research

Program verification and synthesis techniques to help programmers with DB schema refactoring

Just a starting point, not a finished body of work!



My goal: Convince you that there are lots of open & interesting problems in this space!

Idealized Model: Parametrized SQL Programs

Program is a set of methods (transactions), where each method can take user input, but the body is straight-line SQL code

```
string getName(int id) {  
    SELECT name  
    FROM users  
    WHERE uid = id  
}
```

```
void insertUser  
(int id, string name) {  
    INSERT INTO users  
    VALUES (id, name)  
}
```

Each method is either an update transaction (i.e., modifies DB), or query transaction

Outline

Part I: Equivalence verification for parametrized SQL programs (POPL'18)



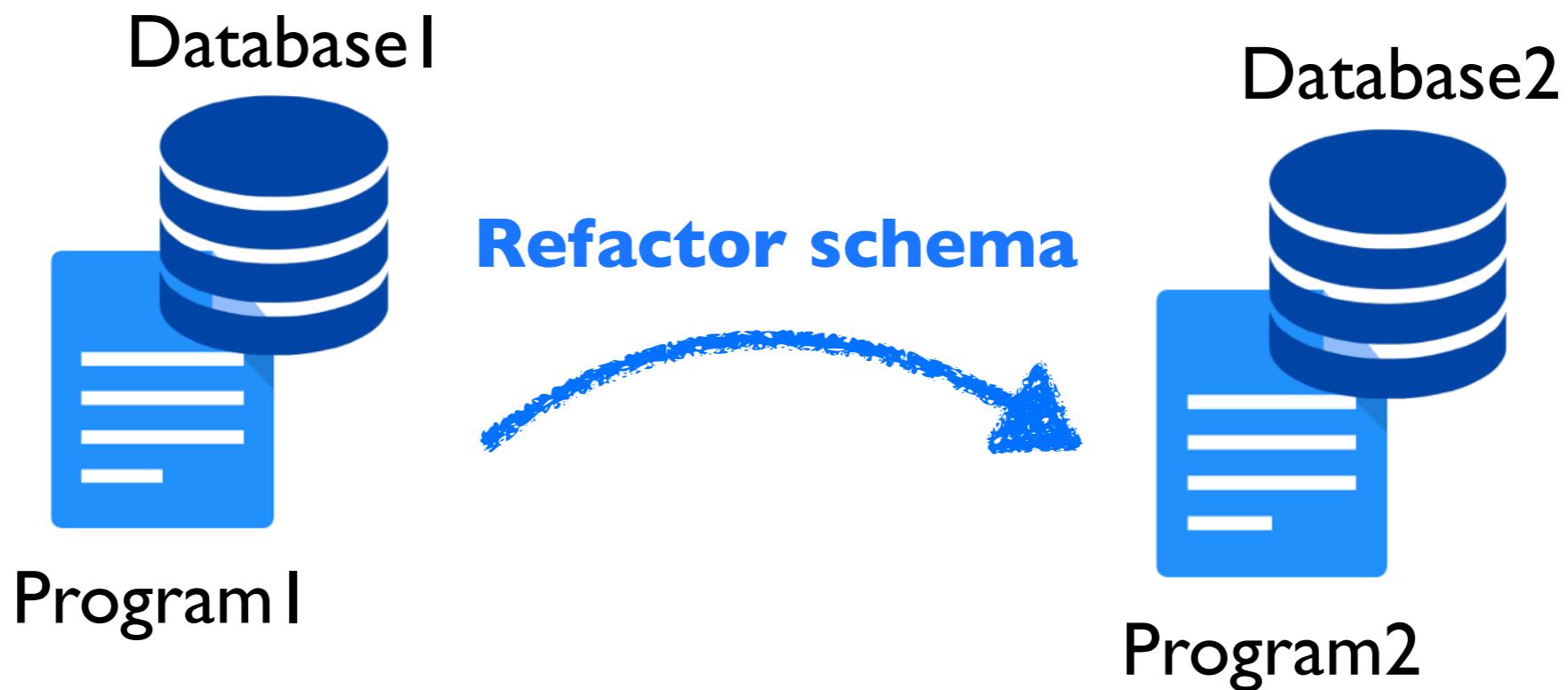
Part II: Synthesizing new version of parametrized SQL program for a given target schema (PLDI'19)



Part III: Open problems & challenges



Verification Problem



*Are the two programs equivalent
before and after schema change?*

Example

P

```
void createSub(int id, String name, String fltr)
    INSERT INTO Subscriber VALUES (id, name, fltr);

void updateSub(int id, String fltr)
    UPDATE Subscriber SET filter=fltr WHERE sid=id;

List<Tuple> getSubFilter(int id)
    SELECT filter FROM Subscriber WHERE sid=id;
```

P'

```
void createSub(int id, String name, String fltr)
    INSERT INTO Subscriber' VALUES (id, name, UID_x);
    INSERT INTO Filter VALUES (UID_x, fltr);

void updateSub(int id, String fltr)
    UPDATE Filter SET params=fltr WHERE fid IN
        (SELECT fid_fk FROM Subscriber' WHERE sid=id);

List<Tuple> getSubFilter(int id)
    SELECT params FROM Filter JOIN Subscriber'
        ON fid=fid_fk WHERE sid=id;
```

Subscriber

sid	sname	filter
...

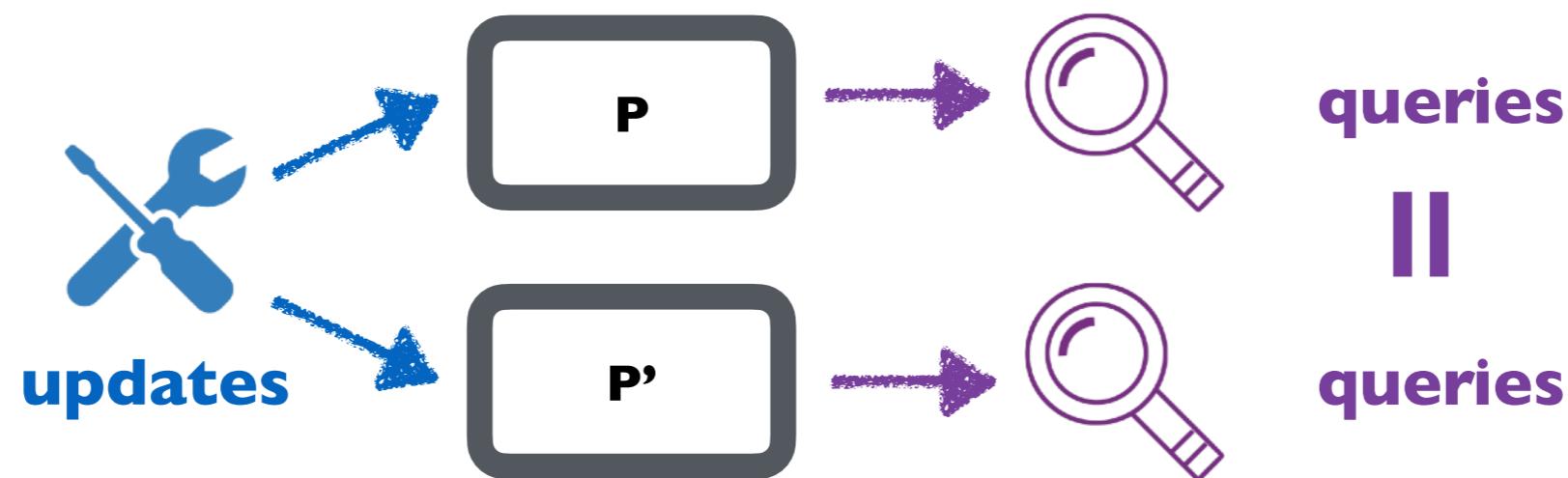
*Are these
implementations
equivalent?*

Filter

fid	params
...	...

Defining Equivalence

Consider two SQL programs P and P' that provide same interface but different implementations



$P \equiv P'$: Every query transaction yields the same result after invoking same sequence of update transactions

Example Revisited

P

```
void createSub(int id, String name, String fltr)
    INSERT INTO Subscriber VALUES (id, name, fltr);

void updateSub(int id, String fltr)
    UPDATE Subscriber SET filter=fltr WHERE sid=id;

List<Tuple> getSubFilter(int id)
    SELECT filter FROM Subscriber WHERE sid=id;
```

P'

```
void createSub(int id, String name, String fltr)
    INSERT INTO Subscriber' VALUES (id, name, UID_x);
    INSERT INTO Filter VALUES (UID_x, fltr);

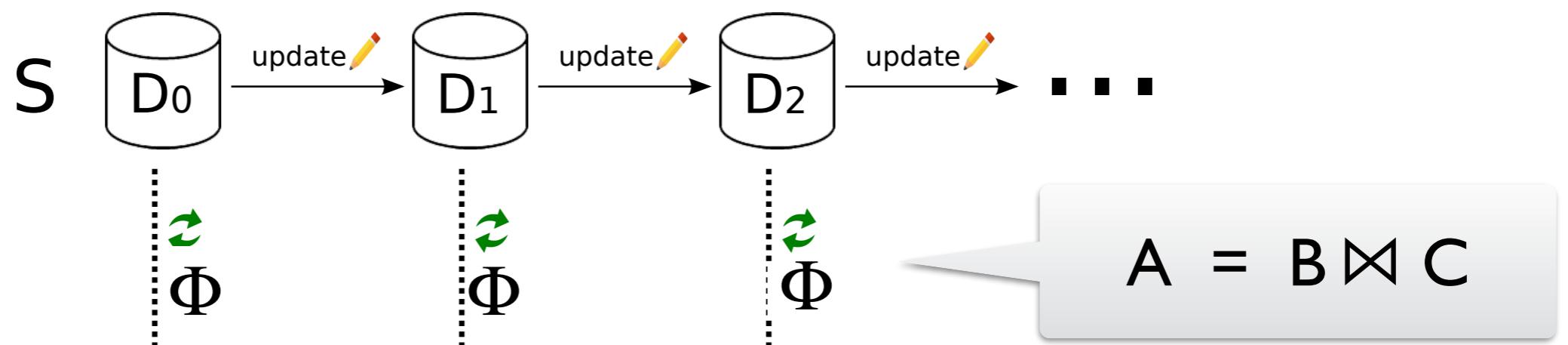
void updateSub(int id, String fltr)
    UPDATE Filter SET params=fltr WHERE fid IN
        (SELECT fid_fk FROM Subscriber' WHERE sid=id);

List<Tuple> getSubFilter(int id)
    SELECT params FROM Filter JOIN Subscriber'
        ON fid=fid_fk WHERE sid=id;
```

After an arbitrary sequence of `createSub` and `updateSub` transactions, `getSubFilter` should yield same answer

Proving Equivalence: Methodology

Find bisimulation invariant Φ that relates the two DB states



$$(\Phi \wedge \vec{x} = \vec{y}) \models \llbracket Q_i \rrbracket = \llbracket Q'_i \rrbracket$$

?

Inferring Bisimulation Invariants

*Automatically infer inductive bisimulation invariants
using a guess-and-check approach (Houdini)*

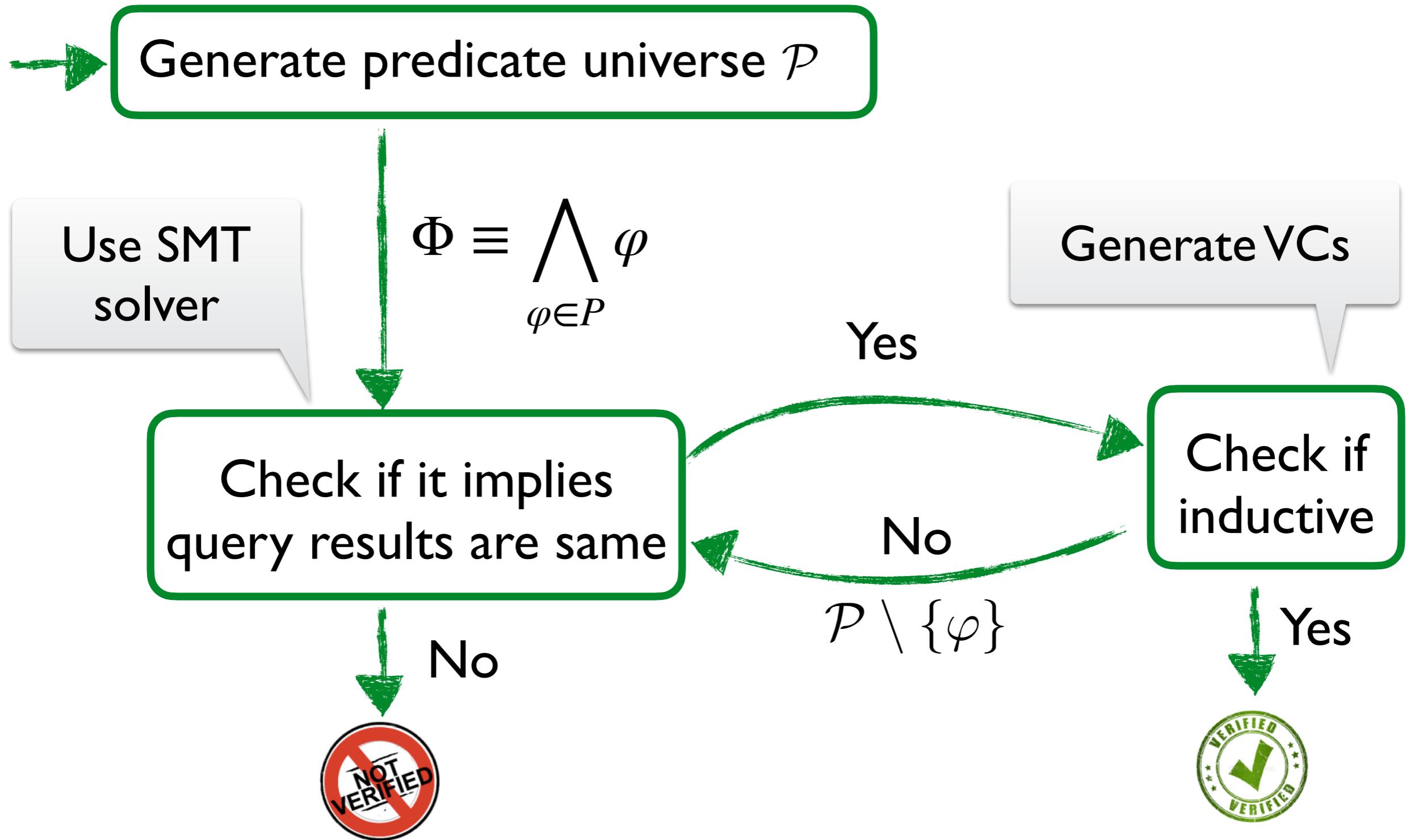
- Generate a **universe** of candidate predicates from a set of templates

$$\Pi_?(?) = \Pi_?(?) \quad \Pi_?(?) = \Pi_?(? \bowtie ?)$$



- Perform fixed-point computation to find strongest (conjunctive) bisimulation invariant over this universe

Verification Workflow



Outline

 **Part I:** Equivalence verification for parametrized SQL programs (POPL'18)



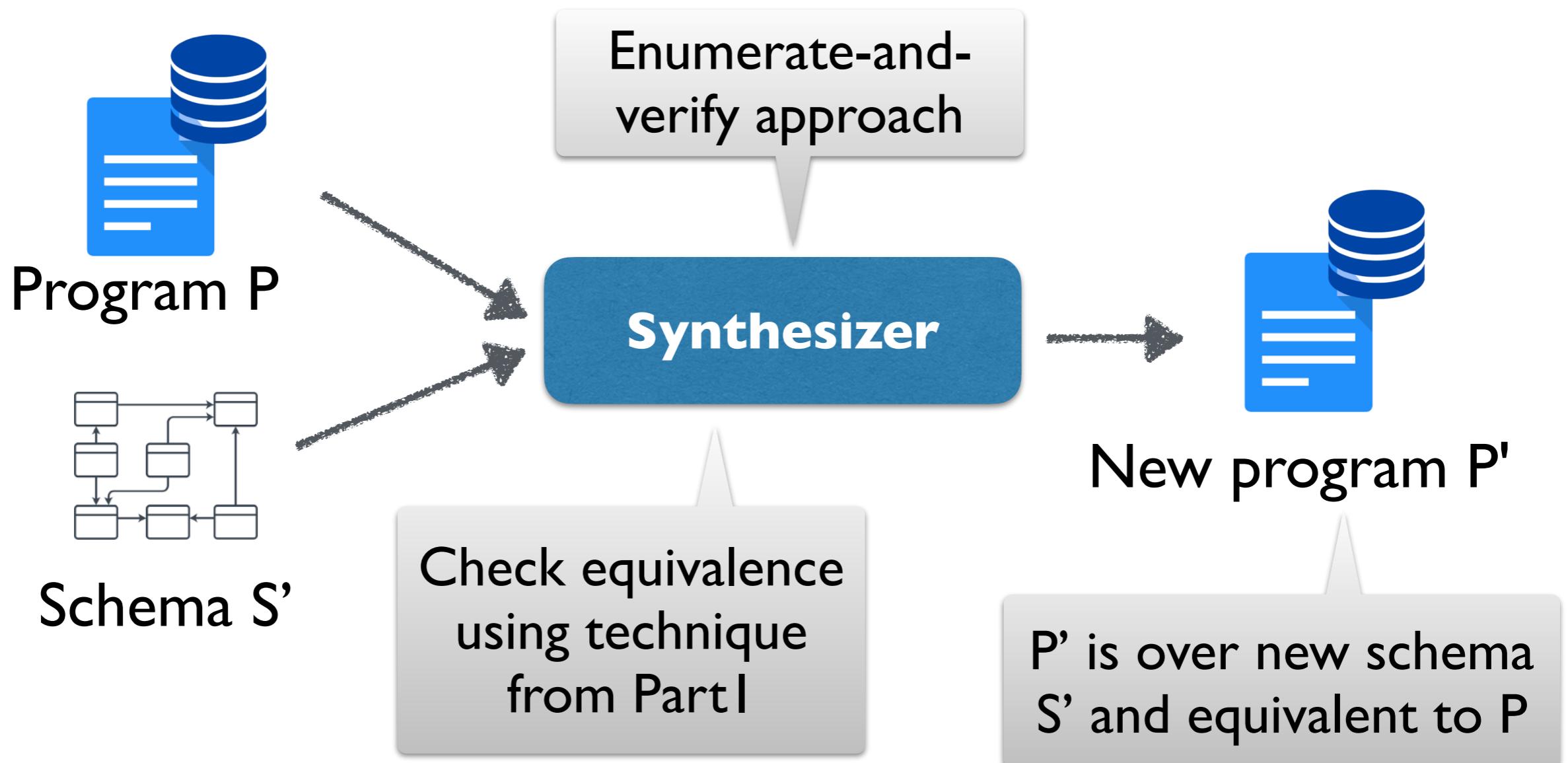
Part II: Synthesizing new version of parametrized SQL program for a given target schema (PLDI'19)



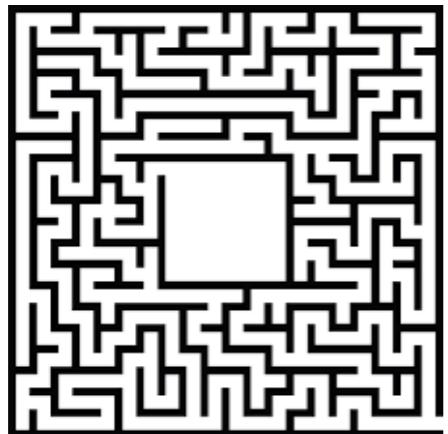
Part III: Open problems & challenges



Synthesis Problem



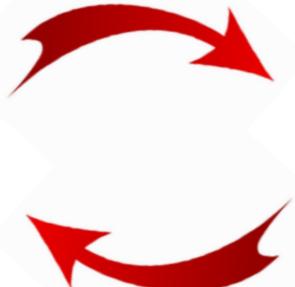
Challenge



Search space is very large!



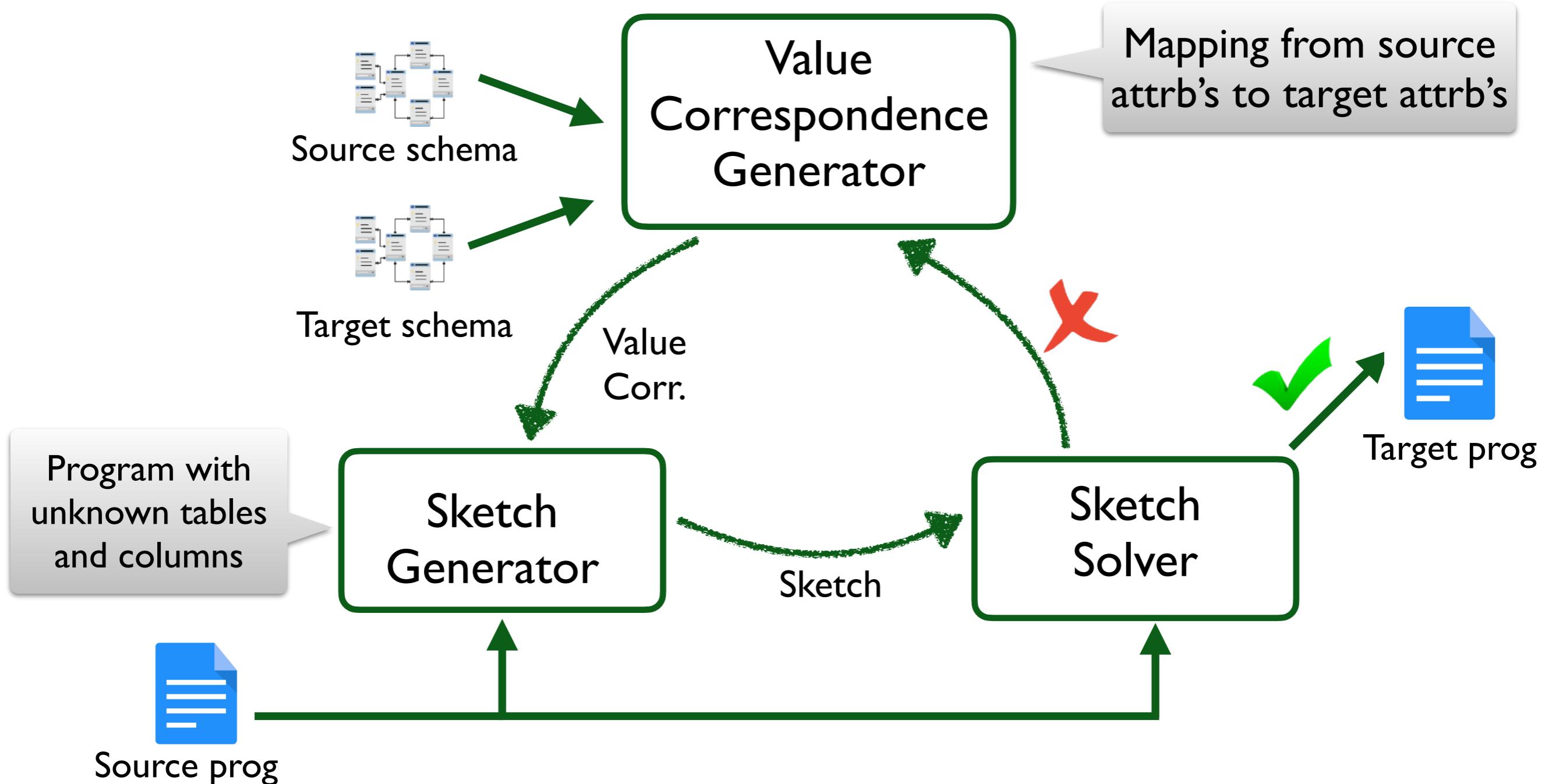
*Sketch
generation*



*Sketch
completion*



Synthesis Methodology



Motivating Example for Synthesis

```
update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);

update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;

query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;
```



Inst		
InstId	IName	IPic

TA		
Tald	TName	TPic

Picture		
PicId	Pic	InstId

Inst'		
InstId	IName	PicId

TA'		
Tald	TName	PicId

Motivating Example for Synthesis

```
update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);
```

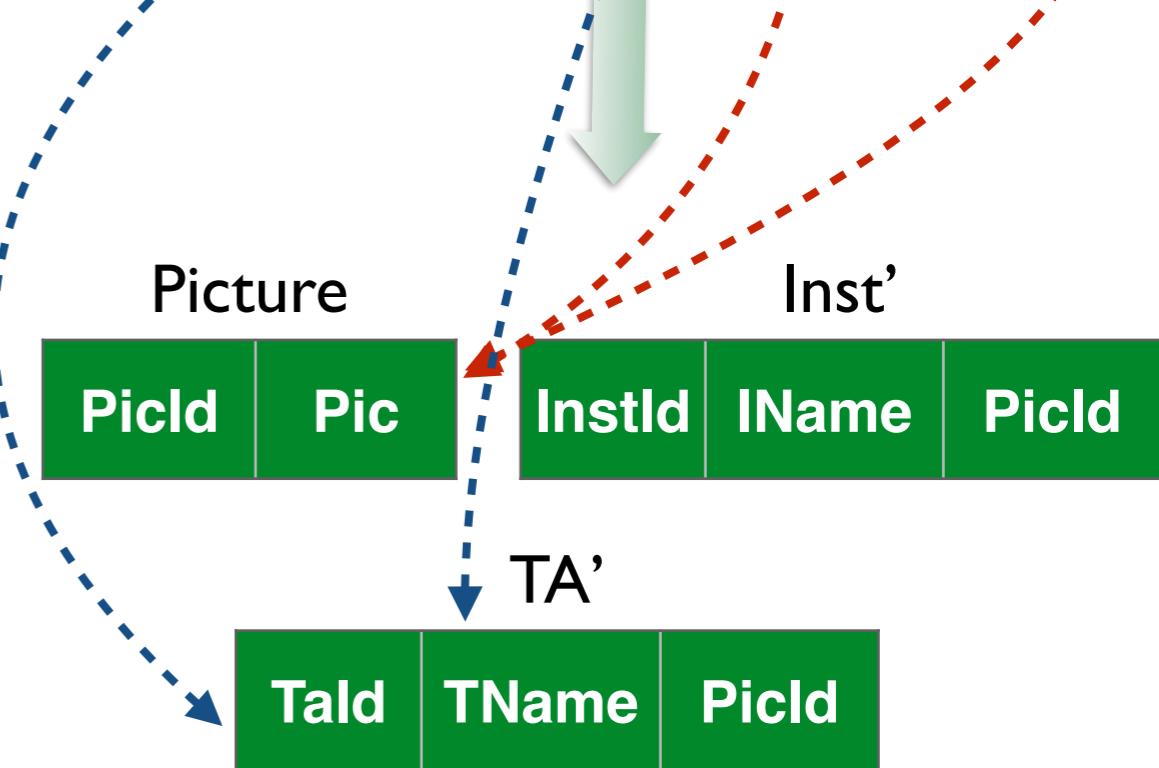
```
update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;
```

```
query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;
```



Inst		
InstId	IName	IPic

TA		
Tald	TName	TPic



Motivating Example for Synthesis

```
update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);
```

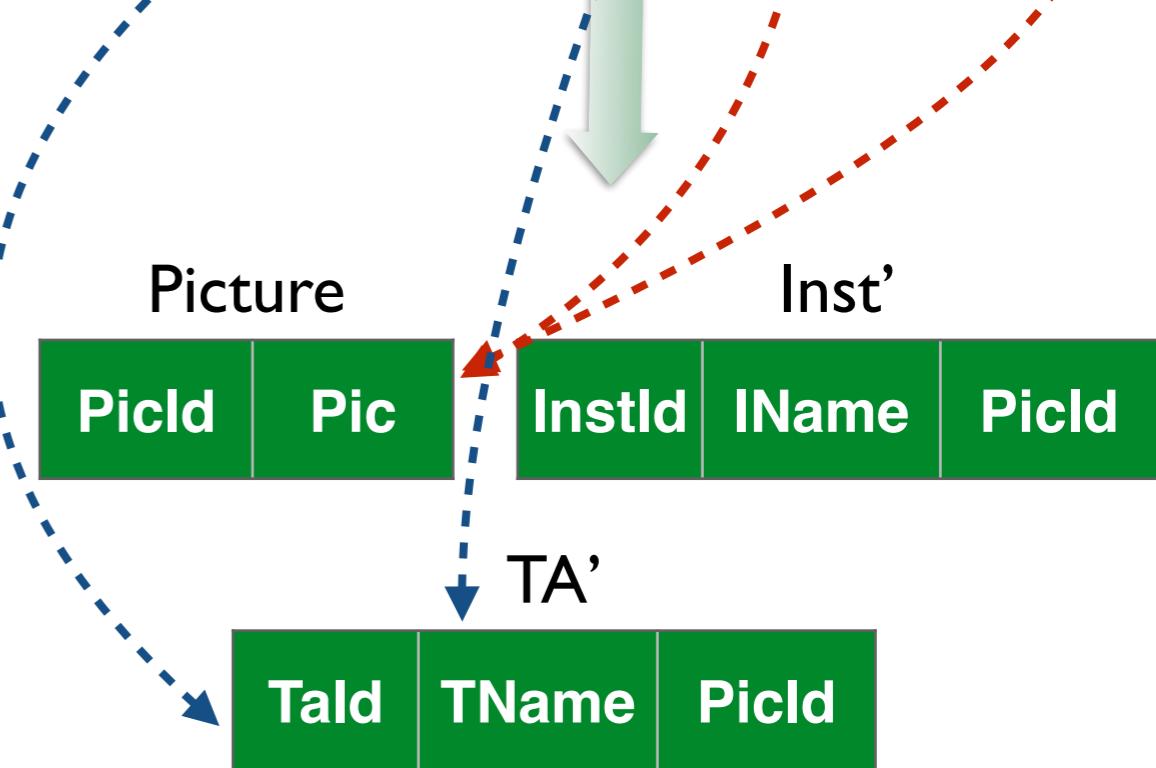
```
update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;
```

```
query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;
```



Inst		
InstId	IName	IPic

TA		
Tald	TName	TPic



Motivating Example for Synthesis

```
update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);
```

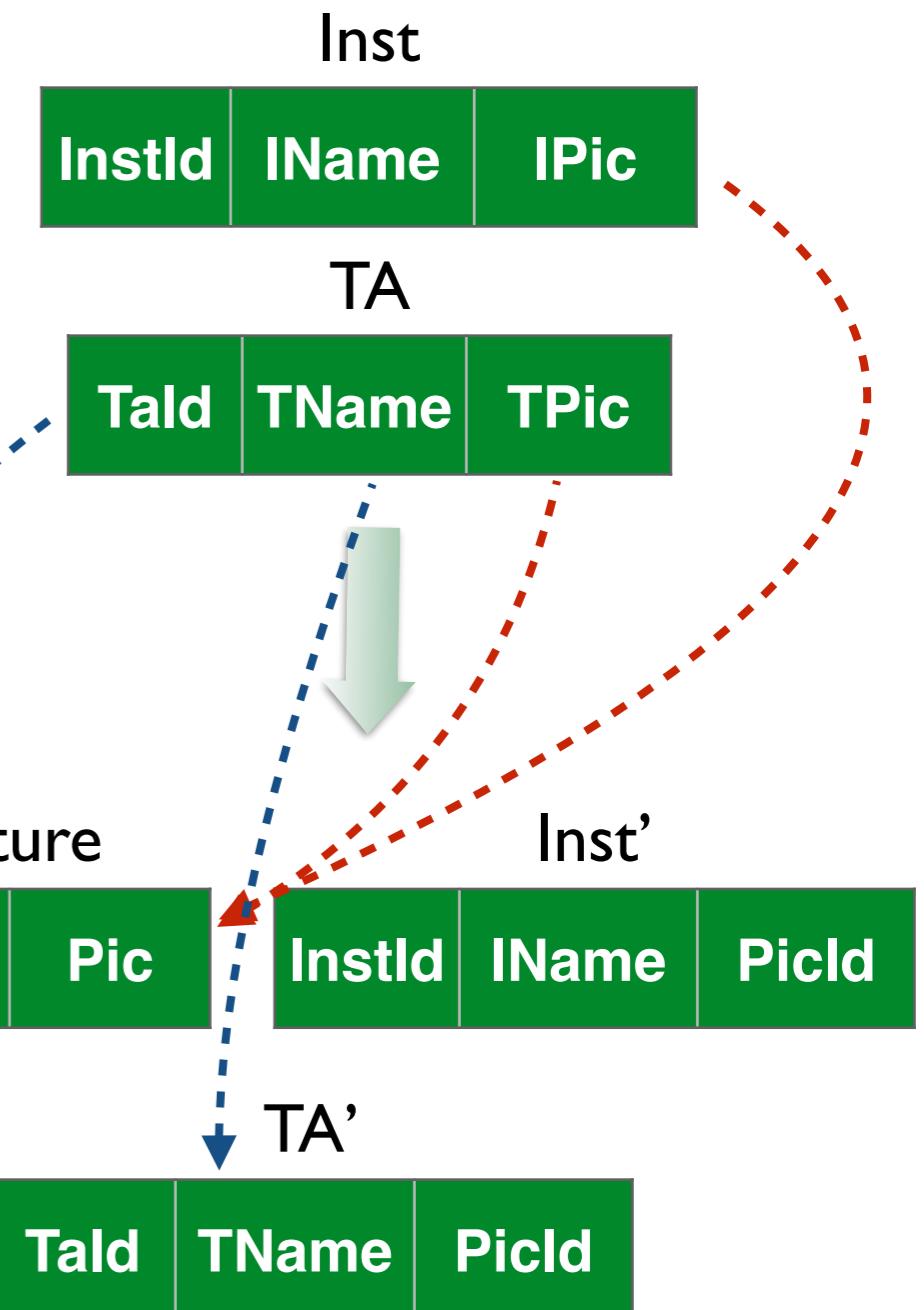
```
update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;
```

```
query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;
```

query getTAInfo(int id)
SELECT ??₁, ??₂ FROM ??₃ WHERE ??₄=id;

??₁=TName ??₂=Pic ??₄=Tald

??₃ ∈ { Picture ⊗ TA, Picture ⊗ TA ⊗ Inst }



Motivating Example for Synthesis

```
update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);
```

```
update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;
```

```
query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;
```

```
update addTA(int id, String name, Binary pic)
    INSERT INTO ??1 VALUES (id, name, pic);
```

```
update deleteTA(int id)
    DELETE ??2 FROM ??3 WHERE Tald = id;
```

```
query getTAInfo(int id)
    SELECT TName, Pic FROM ??4 WHERE Tald=id;
```

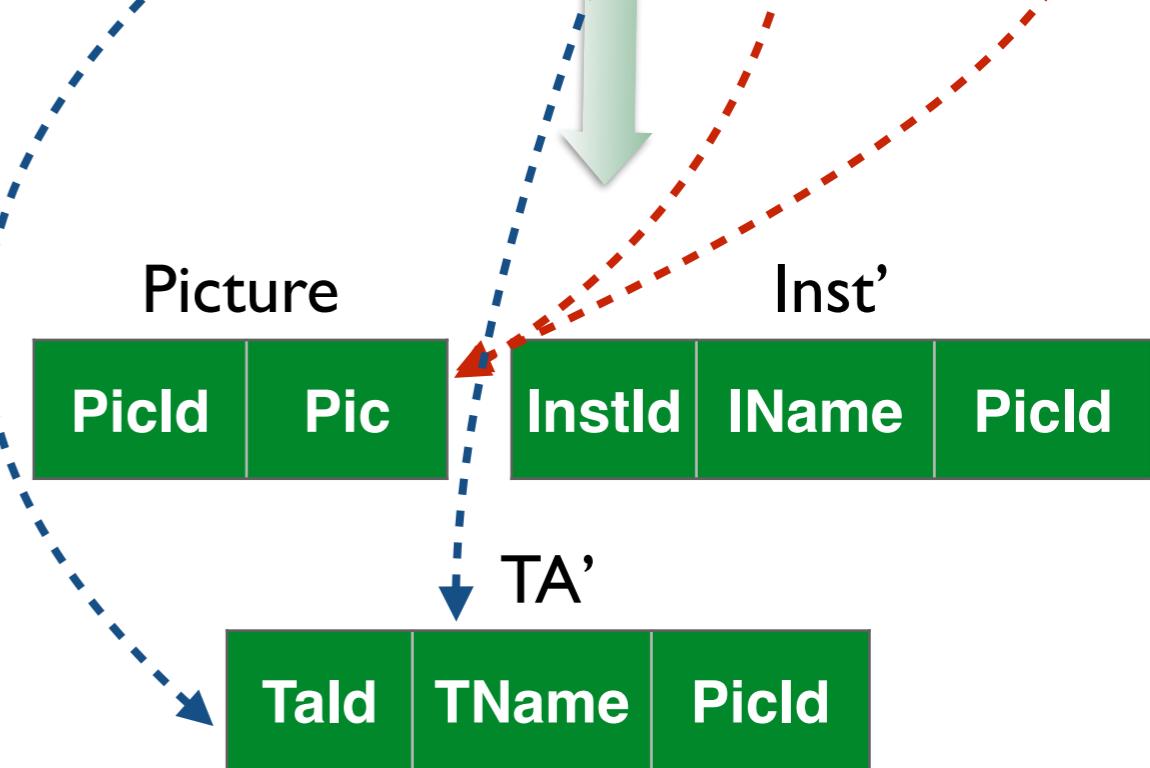
$$??_1, ??_4 \in \{ \text{Picture} \bowtie \text{TA}, \text{Picture} \bowtie \text{TA} \bowtie \text{Inst} \}$$

$$??_2 \in \{ [\text{Picture}], [\text{TA}], \dots, [\text{Picture}, \text{TA}, \text{Inst}] \}$$

$$??_3 \in \{ \text{Picture} \bowtie \text{TA}, \text{Picture} \bowtie \text{TA} \bowtie \text{Inst} \}$$

Inst		
InstId	IName	IPic

TA		
Tald	TName	TPic



Solving the Sketch

Basic idea: Enumerate all programs in search space

For each completion, check if it is equivalent to original program

Scalability?

15 holes, 3 instantiations:
>14 million programs!



Use **conflict-driven learning***
to prune search space

* Program Synthesis using Conflict-Driven Learning. Feng, Martins, Bastani, Dillig. PLDI'18

Learning from Conflicts

Whenever you enumerate an incorrect program,
infer a set of other provably-incorrect programs!

Prior work* shows how to do this for programming-by-example, but not applicable here



Leverage notion of
“minimum distinguishing inputs”
to learn from failed synthesis attempts!

* Program Synthesis using Conflict-Driven Learning. Feng, Martins, Bastani, Dillig. PLDI’18

Distinguishing Inputs

Recall: Input to DB program is a set of function invocations along with their arguments.

```
update addTA(int id, String name, Binary pic)
  INSERT INTO TA VALUES (id, name, UID1);
  INSERT INTO Picture VALUES (UID1, pic);
```

```
update deleteTA(int id)
  DELETE Picture FROM Picture
    JOIN TA ON Picture.PicId = TA.PicId
    JOIN Inst ON TA.PicId = Inst.PicId WHERE Tald = id;
```

```
query getTaInfo(int id)
  SELECT TName, Pic FROM Picture
    JOIN TA ON Picture.PicId=TA.PicId
    JOIN Inst ON TA.PicId = Inst.PicID WHERE Tald=id;
```

Input:

```
addTA(1, "A", null);
getTaInfo(1);
```

Input I is **distinguishing** for a pair of programs P, P' iff $P(I) \neq P'(I)$

Conflict-Driven Learning from Distinguishing Inputs

Consider distinguishing input I for original program P and synthesized (incorrect) program P'

Suppose P, P' contain N functions but I invokes only K functions where $K \ll N$



Any program P'' that agrees with P' on those K functions will also be incorrect!!

Rather than just rejecting a single program, we can reject a whole SET!

The smaller the input, the bigger the set, so want minimum distinguishing inputs!

Distinguishing Inputs in Action

Synthesized incorrect program:

```
update addTA(int id, String name, Binary pic)
  INSERT INTO TA VALUES (id, name, UID1);
  INSERT INTO Picture VALUES (UID1, pic);
```

```
update deleteTA(int id)
  DELETE Picture FROM Picture
    JOIN TA ON Picture.PicId = TA.PicId
    JOIN Inst ON TA.PicId = Inst.PicId WHERE Tald = id;
```

```
query getTaInfo(int id)
  SELECT TName, Pic FROM Picture
    JOIN TA ON Picture.PicId=TA.PicId
    JOIN Inst ON TA.PicId = Inst.PicID WHERE Tald=id;
```

Distinguishing Input:
addTA(1, "A", null);
getTAInfo(1);

Assignments to holes in
deleteTA are irrelevant!

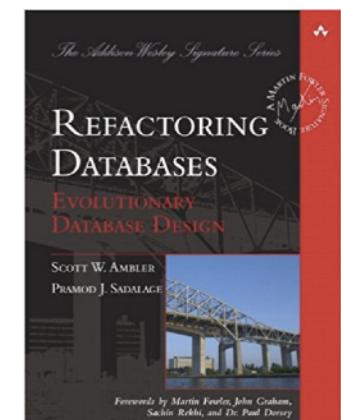


Can use this information to rule out 14
programs instead of a single one!

Evaluation

Implemented tool called **Migrator**:
<https://github.com/utopia-group/migrator>

Used Migrator to synthesize new versions of 20 different parametrized SQL programs



Statistics about Benchmarks

	Avg functions	Avg #tables (source)	Avg # tables (target)	Avg # cols (source)	Avg #cols (target)
Text book	10	2	2	8	9
Github	105	12	13	107	110

Key Results

**Successfully synthesized equivalent
versions of all 20 SQL programs!**

	# programs enumerated	Average synthesis time	Average total time
Textbook Benchmarks	3	0.4	5.6
Github benchmarks	19	138	155

**Migrator can synthesize new version of SQL
programs with over 100 transactions in 2.5 minutes!**

Outline

 **Part I:** Equivalence verification for
parametrized SQL programs (POPL'18)



 **Part II:** Synthesizing new version of
parametrized SQL program for a given
target schema (PLDI'19)



Part III: Open problems & challenges



Future Work



*Good starting point, but lots of
research remains to be done...*

Challenge #1

From parametrized SQL programs to real-world DB applications

Interaction
between SQL
and other
languages

SQL code
dynamically
generated

Computation
on query
inputs & result

Challenge #2

From relational DB applications to no-SQL applications

Verify equivalence between programs written in different languages

Larger search space: need to synthesize entire program

Challenge #3

Unified verification & falsification

POPL'18 work
cannot
disprove
equivalence

CEGAR-like
approaches that
search for both
proofs & cexs

Call for New Research



Lots of exciting research opportunities for FM research in automating DB application evolution!

Acknowledgements



Yuepeng Wang
PhD student @ UT Austin

Acknowledgements, cont.



James Dong
UT Austin undergrad



Rushi Shah
UT Austin undergrad

Acknowledgements, cont.



Shuvendu Lahiri
Principal Researcher @ MSR

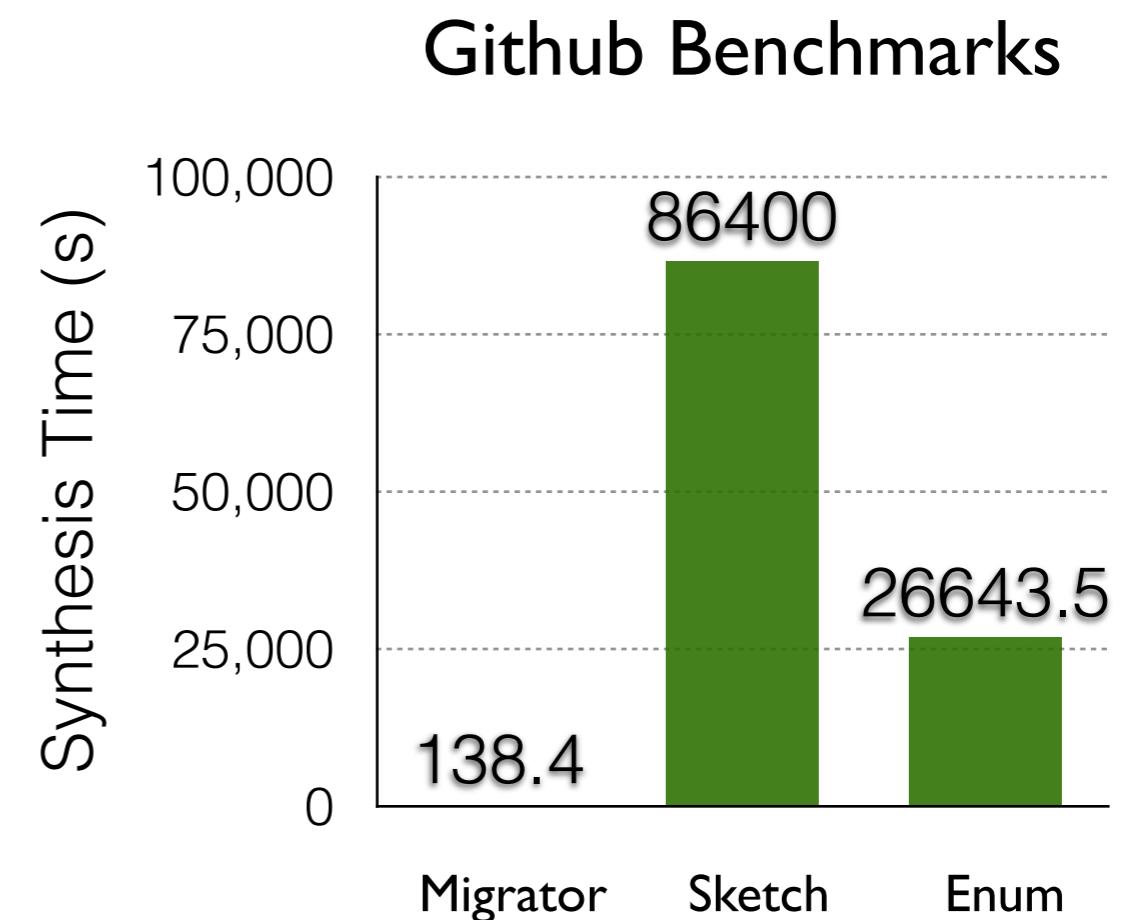
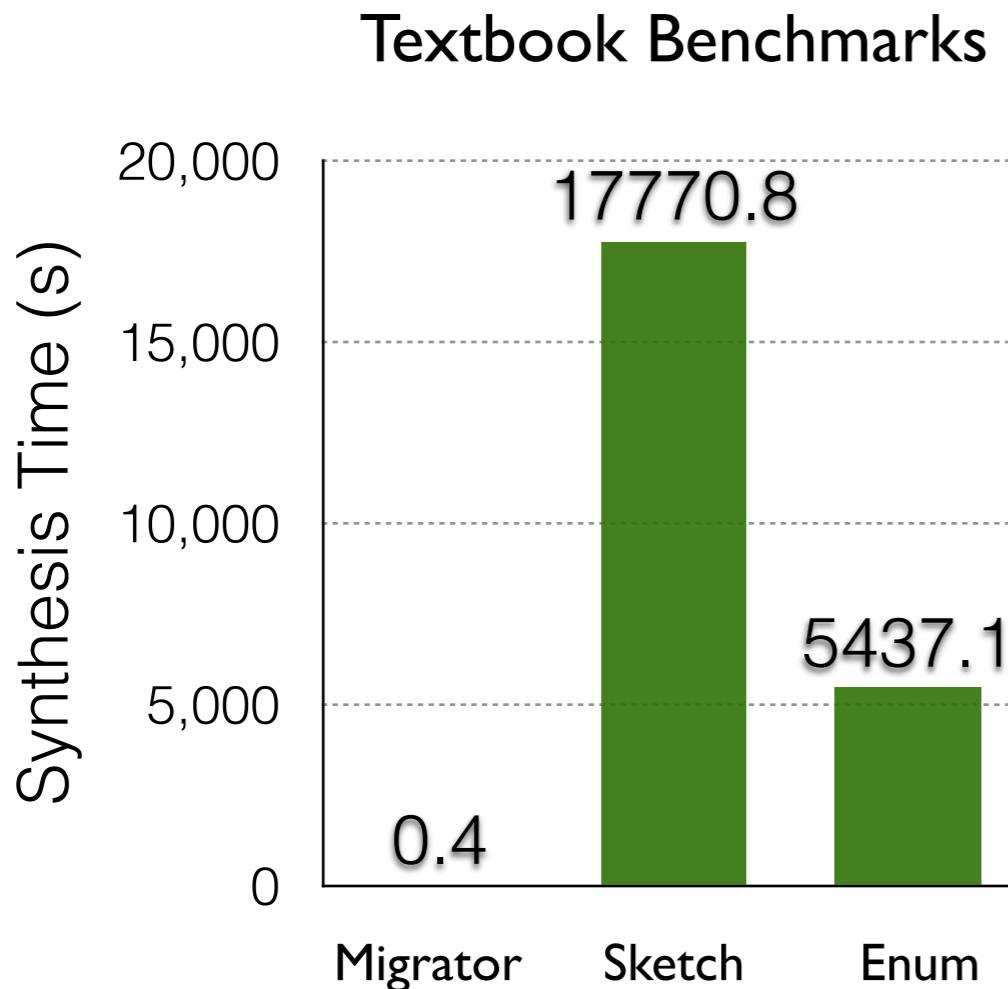


William Cook
Faculty @ UT Austin

Thank you!



Comparison Results



Take-away: >190x average speedup compared to enumeration and >750x speed up compared to Sketch!

Key Results

	# benchmarks verified	Avg verification time
Textbook Benchmarks	10/10	12 s
Github benchmarks	10/11	47 s

Mediator can verify all but one benchmark!

Cause of FP: bisimulation invariant not strong enough

Avg verification time is under 1 min!

Theory of Relational Algebra w/ Updates

Recall: Bisimulation invariants relate DB states

No existing first-order theory
for expressing such invariants

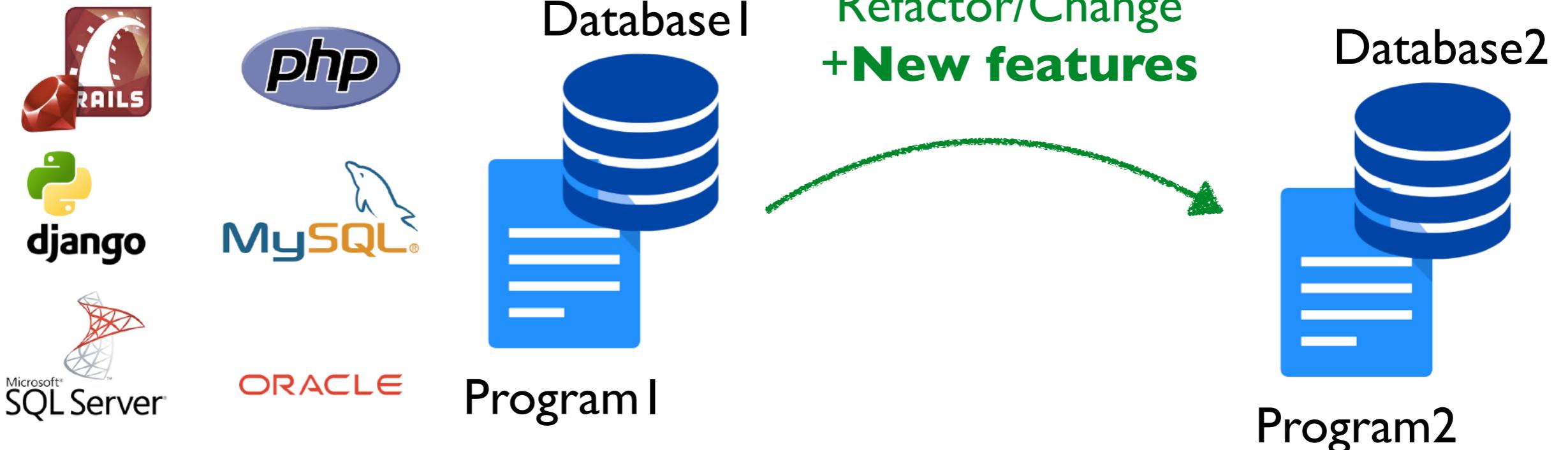
Axiomatized new theory \mathcal{T}_{RA}
(Relational Algebra with Updates)



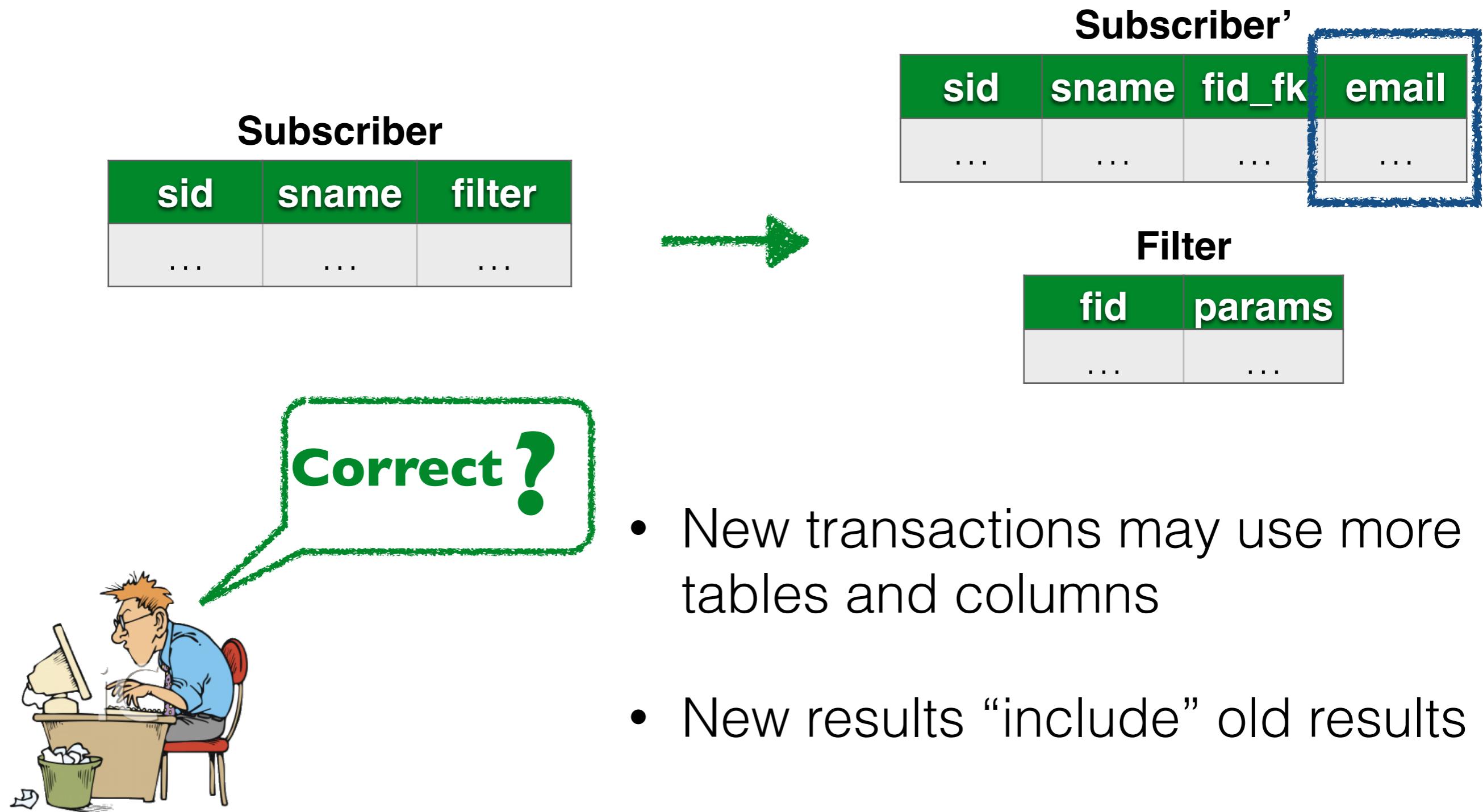
$$\begin{aligned}\Pi_{sid,sname}(Subscriber) &= \Pi_{sid',sname'}(Subscriber') \wedge \\ \Pi_{sid,sname,filter}(Subscriber) &= \Pi_{sid',sname',params'}(Subscriber' \bowtie Filter')\end{aligned}$$

See
POPL'18
paper!

Refinement



Refinement



- New transactions may use more tables and columns
- New results “include” old results

Refinement Checking

- Simulation invariant Φ holds with empty databases
 - Updates $\lambda \vec{x}.U_i$ and $\lambda \vec{y}.U'_i$
 - Queries $\lambda \vec{x}.Q_i$ and $\lambda \vec{y}.Q'_i$

Inductive

$$\left\{ \Phi \wedge \bigwedge_{x_j \in \vec{x}} x_j = y_j \right\} \quad U_i ; \quad U'_i \quad \left\{ \Phi \right\}$$

Sufficient

$$\left(\Phi \wedge \bigwedge_{x_j \in \vec{x}} x_j = y_j \right) \models \exists L. Q_i = \Pi_L(Q'_i)$$

New results include the old results

Generating Value Correspondence

To ensure completeness of synthesis algorithm, need to enumerate **all** possible value correspondences



can't do eagerly!

Need to lazily enumerate in decreasing order of likelihood

Weighted soft constraints for similarity between attr names

Use MaxSAT-based approach

Hard constraints for type compatibility requirements



Allows lazy enumeration of most likely value correspondences

Sketch Completion Algorithm

