SYMBOLIC ANALYSIS OF WEAK CONCURRENCY SEMANTICS IN MODERN DATABASE PROGRAMS

Kia Rahmani

Aug 2, 2022
University of Texas at Austin
Austin, TX

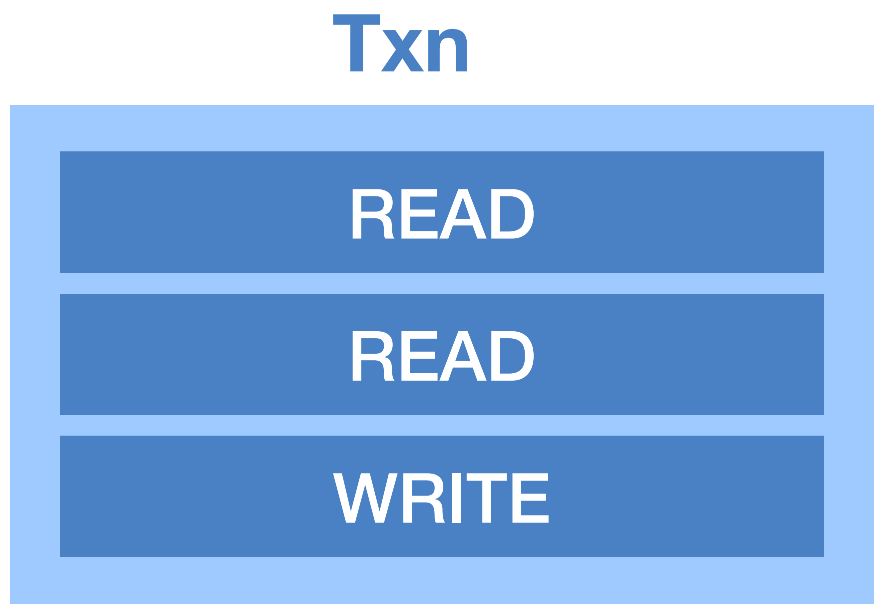
INTRODUCTION

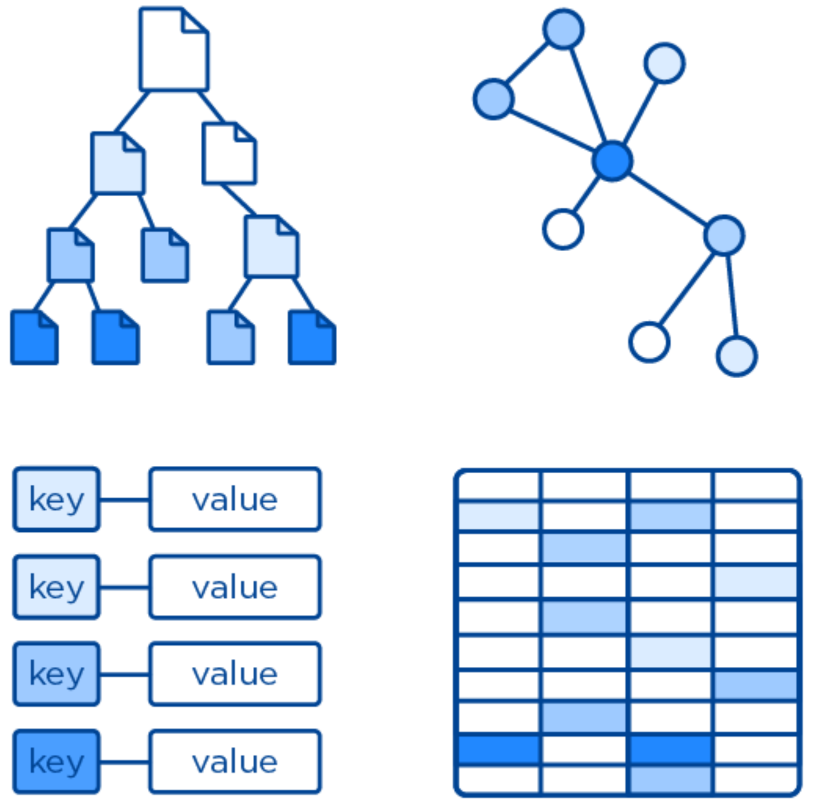
(MAIN SUBJECT: DATABASE SYSTEMS)

TRADITIONAL DATABASE SYSTEMS & PROGRAMS

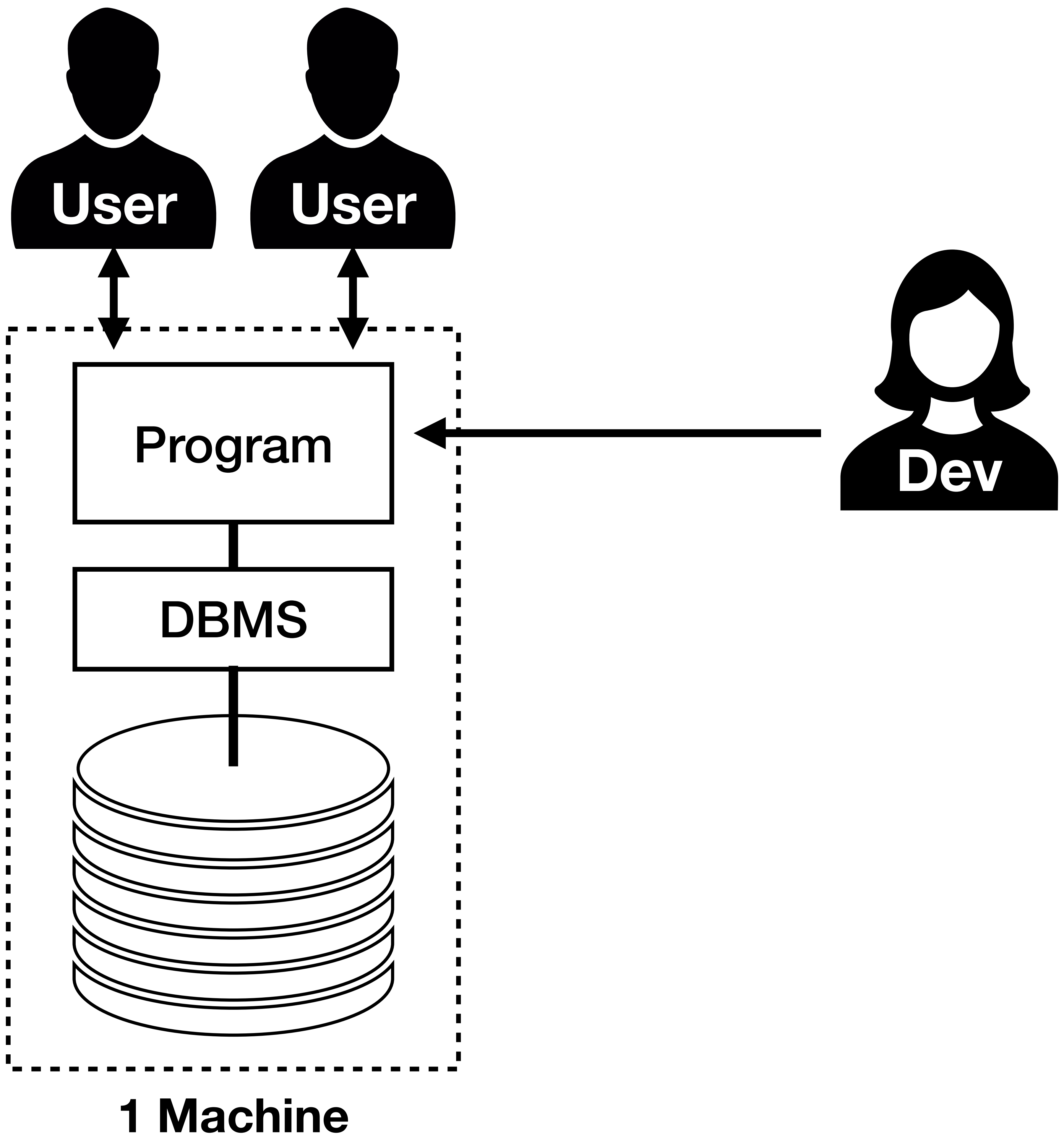
- Data is ubiquitously managed by databases

- Data is ubiquitously managed by databases
- Well-defined abstractions
  - data definition language (**DDL**)
  - data manipulation language (**DML**)

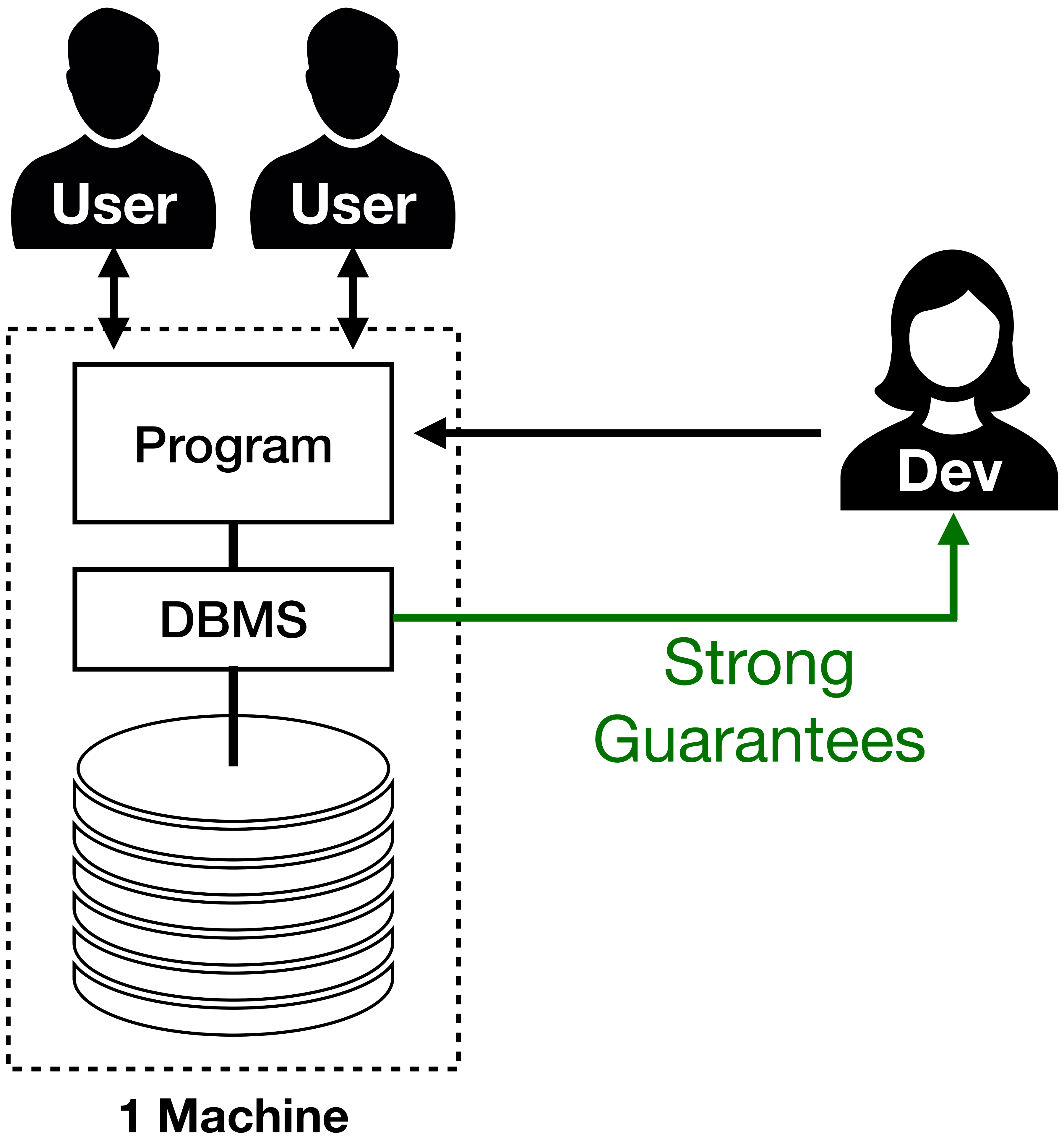




- Data is ubiquitously managed by databases
- Well-defined abstractions
  - data definition language (**DDL**)
  - data manipulation language (**DML**)
- Single machine

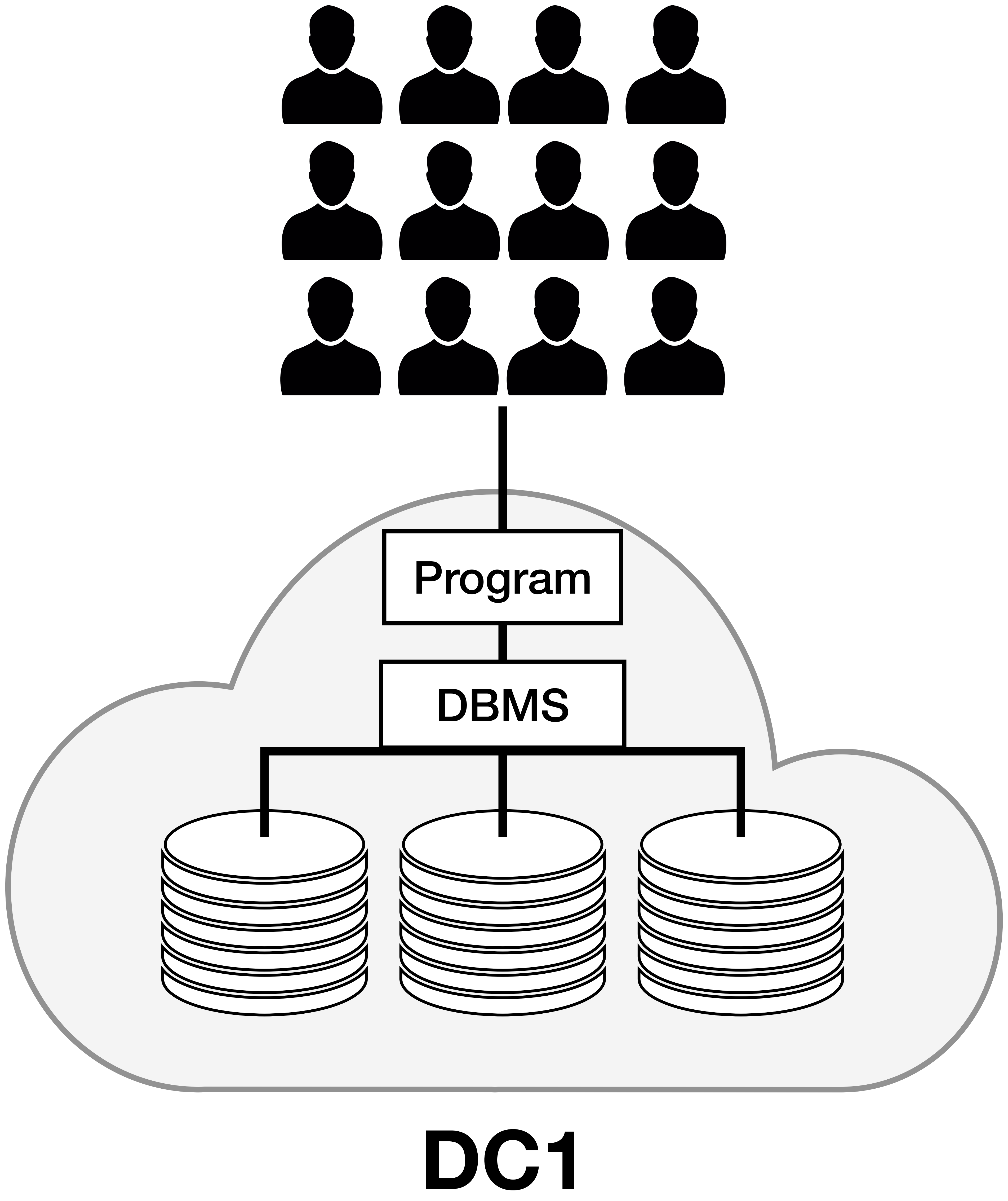


- Data is ubiquitously managed by databases
- Well-defined abstractions
  - data definition language (**DDL**)
  - data manipulation language (**DML**)
- Single machine
- Strong concurrency guarantees

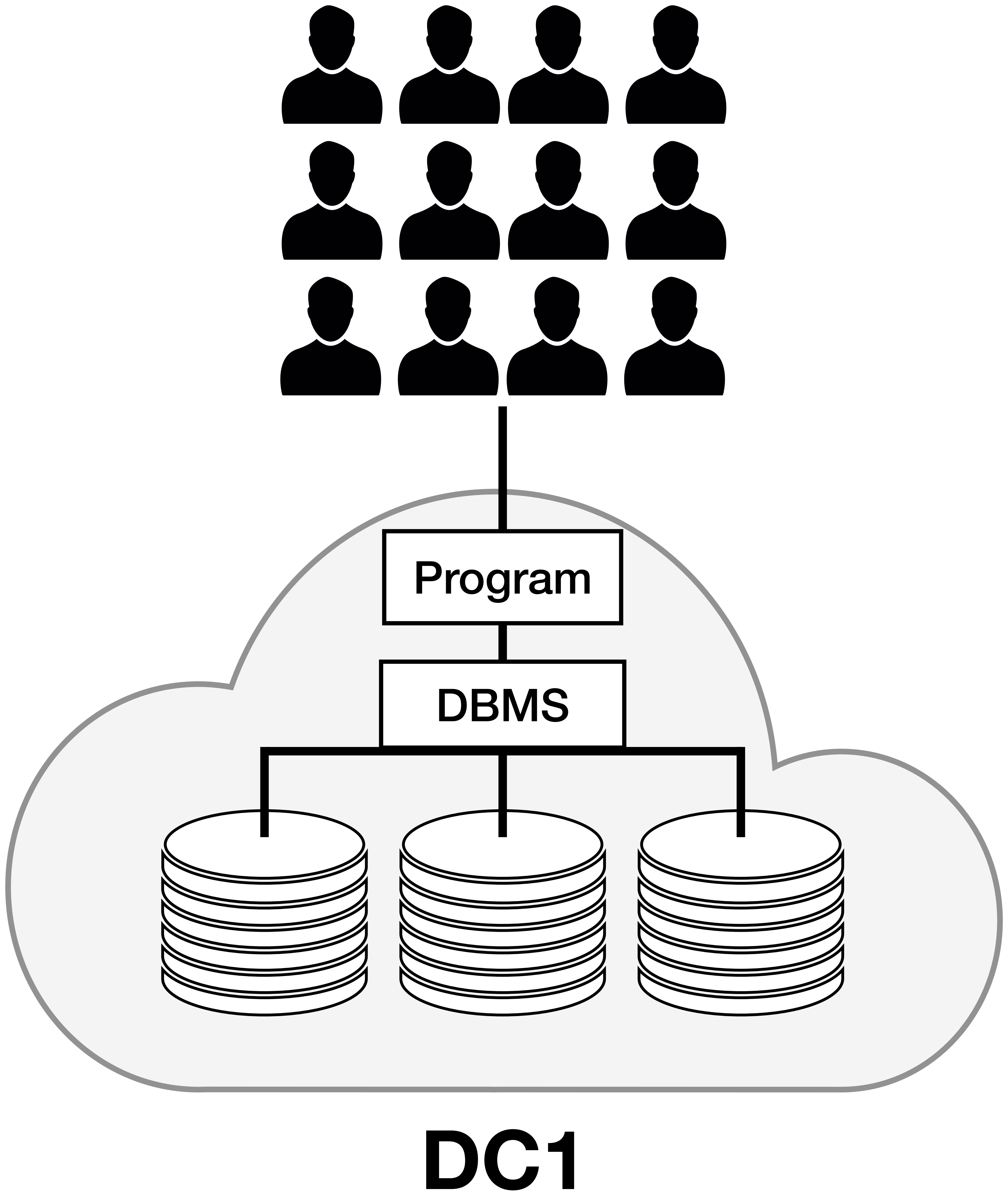


- Single machine = poor performance

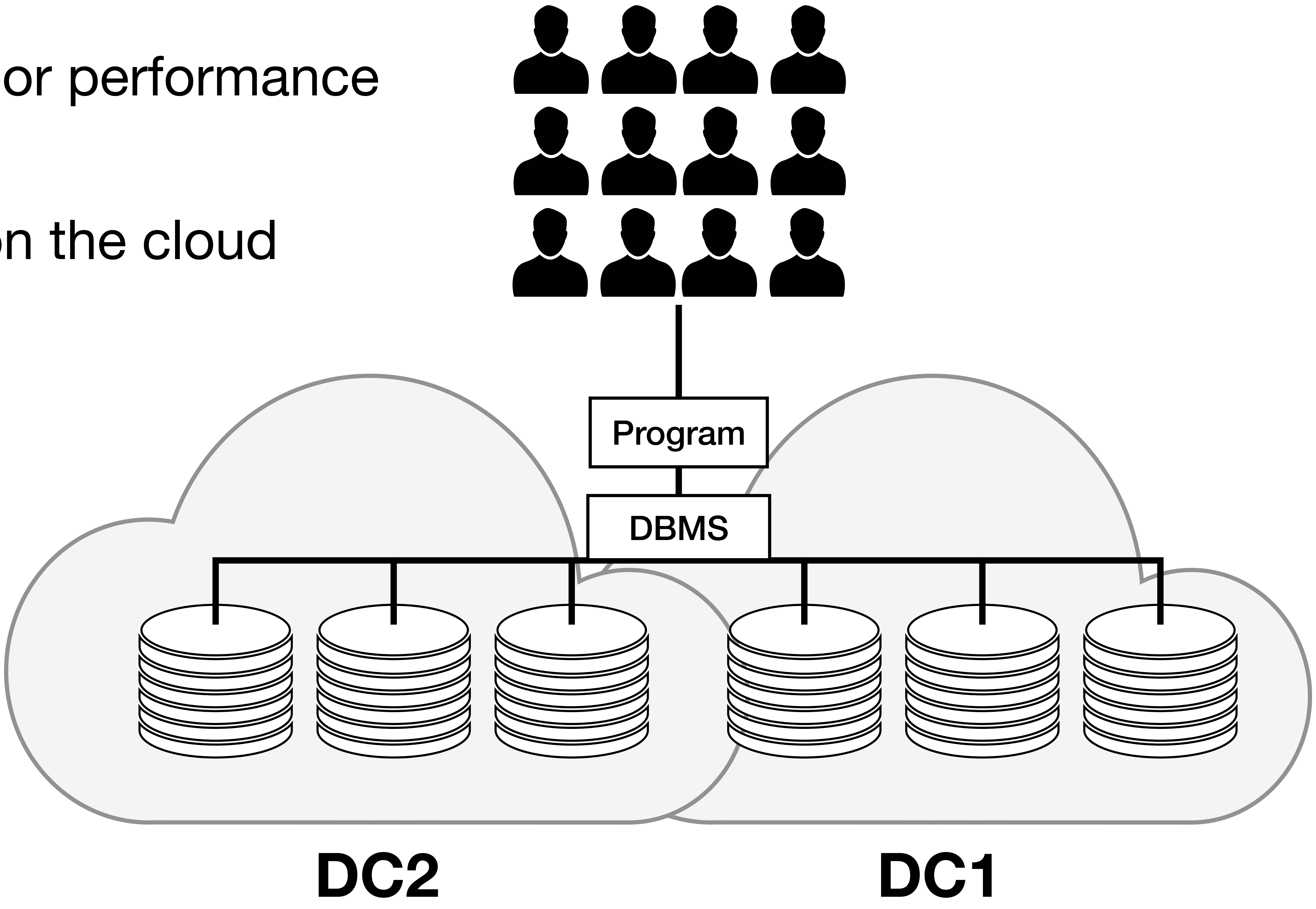
- Single machine = poor performance
- Amazon DynamoDB
  - Cluster of nodes on the cloud



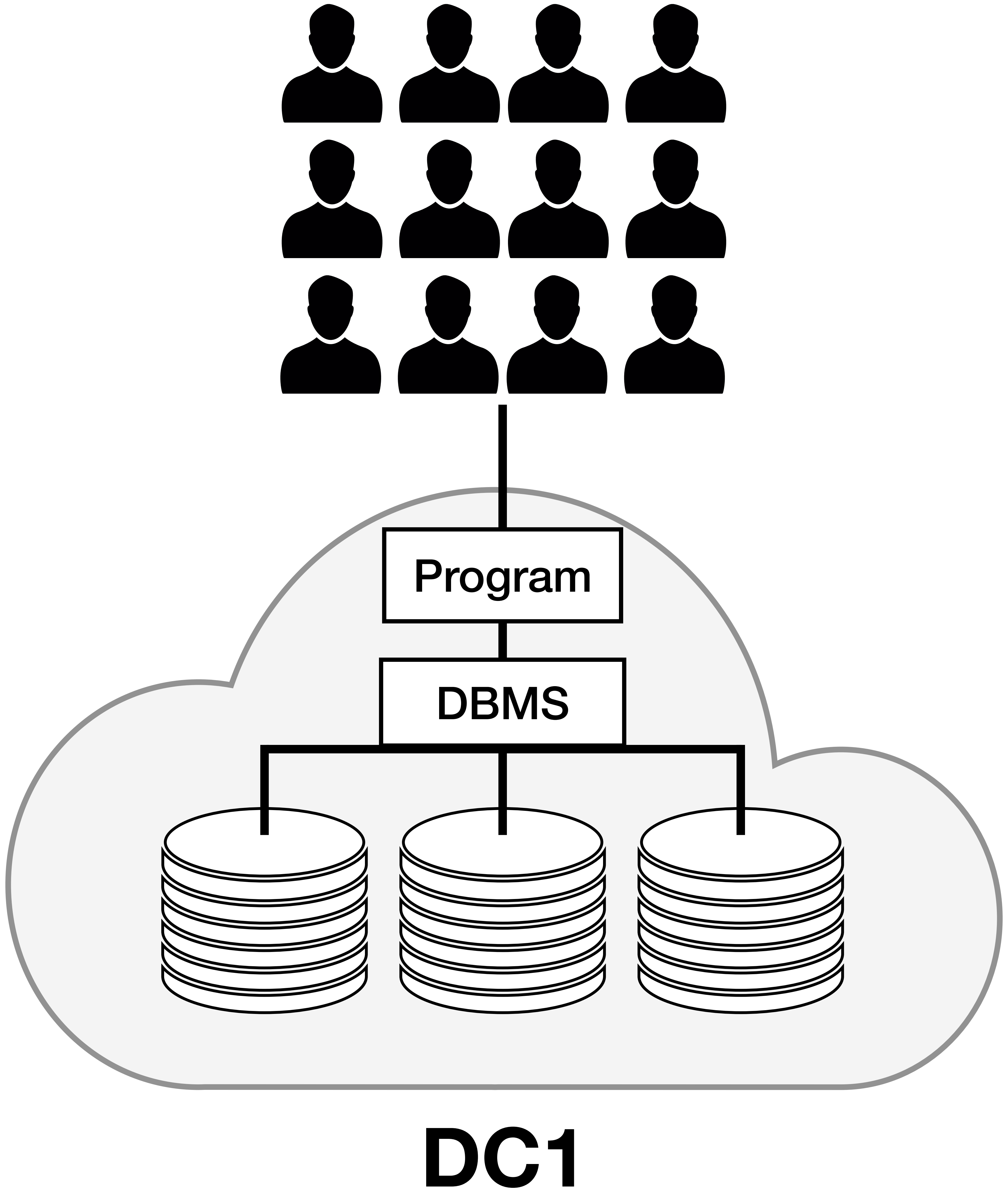
- Single machine = poor performance
- Amazon DynamoDB
  - Cluster of nodes on the cloud
- Designed to scale



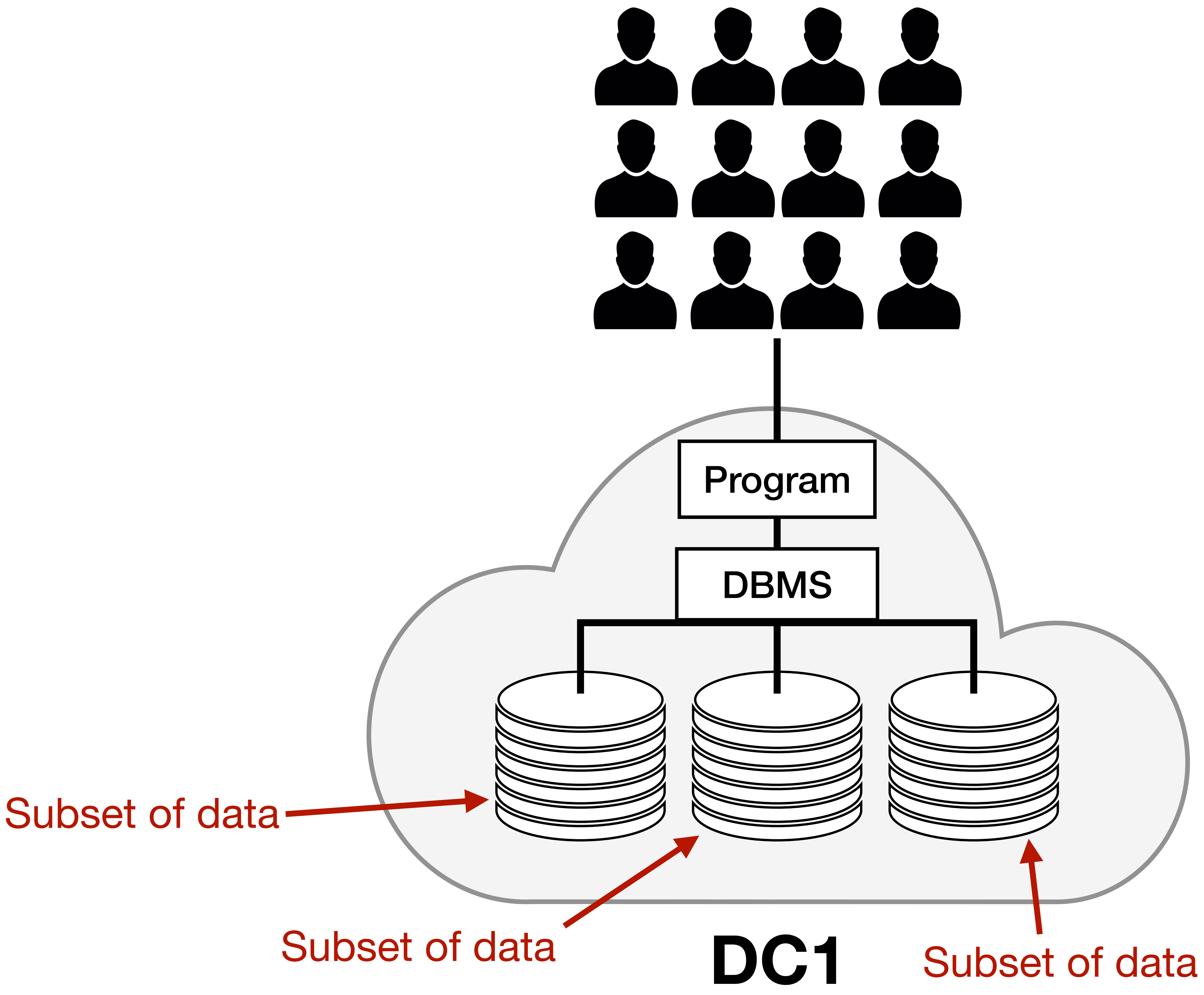
- Single machine = poor performance
- Amazon DynamoDB
  - Cluster of nodes on the cloud
- Designed to scale



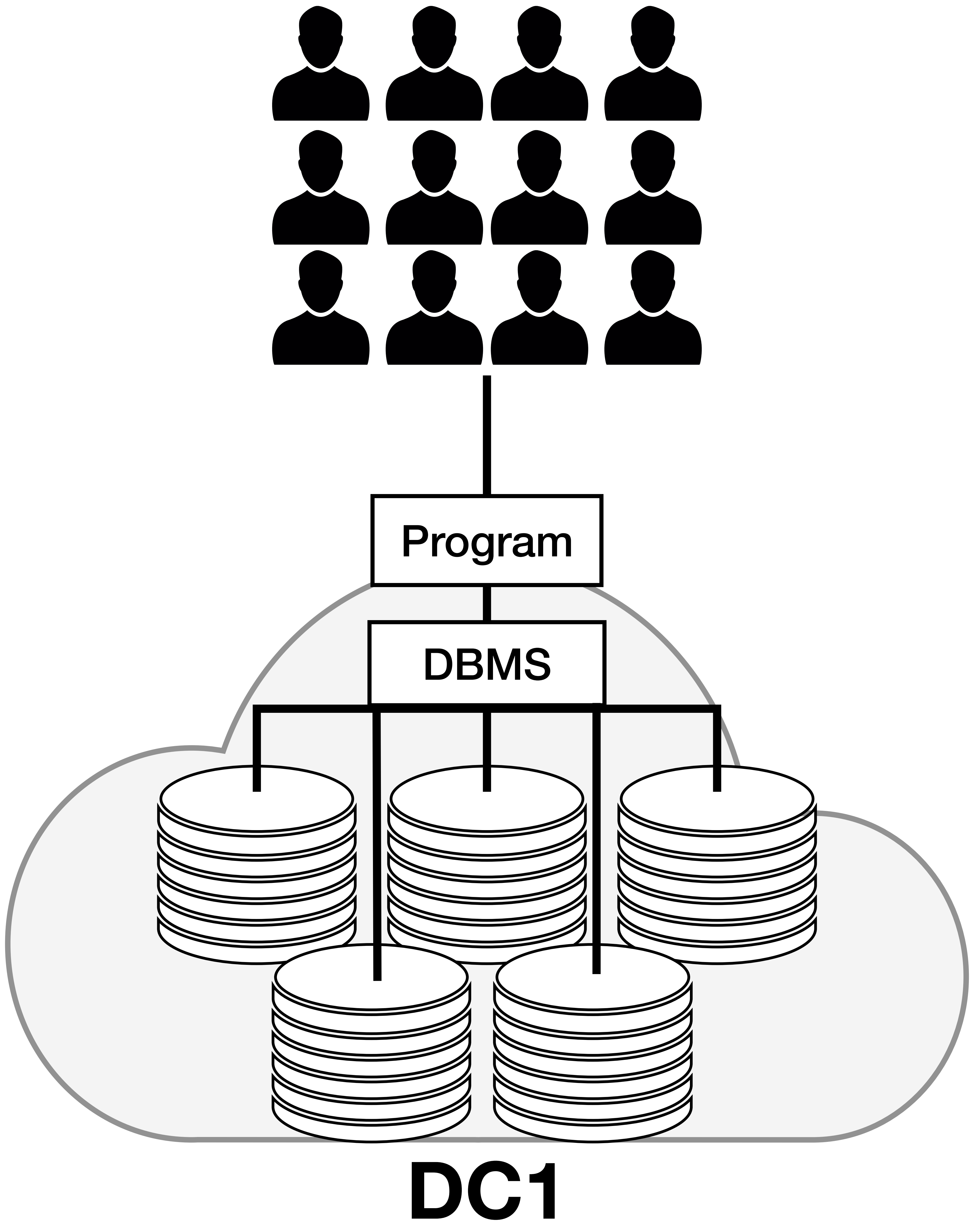
- Clustering architectures
  - Partitioning
  - Replicating



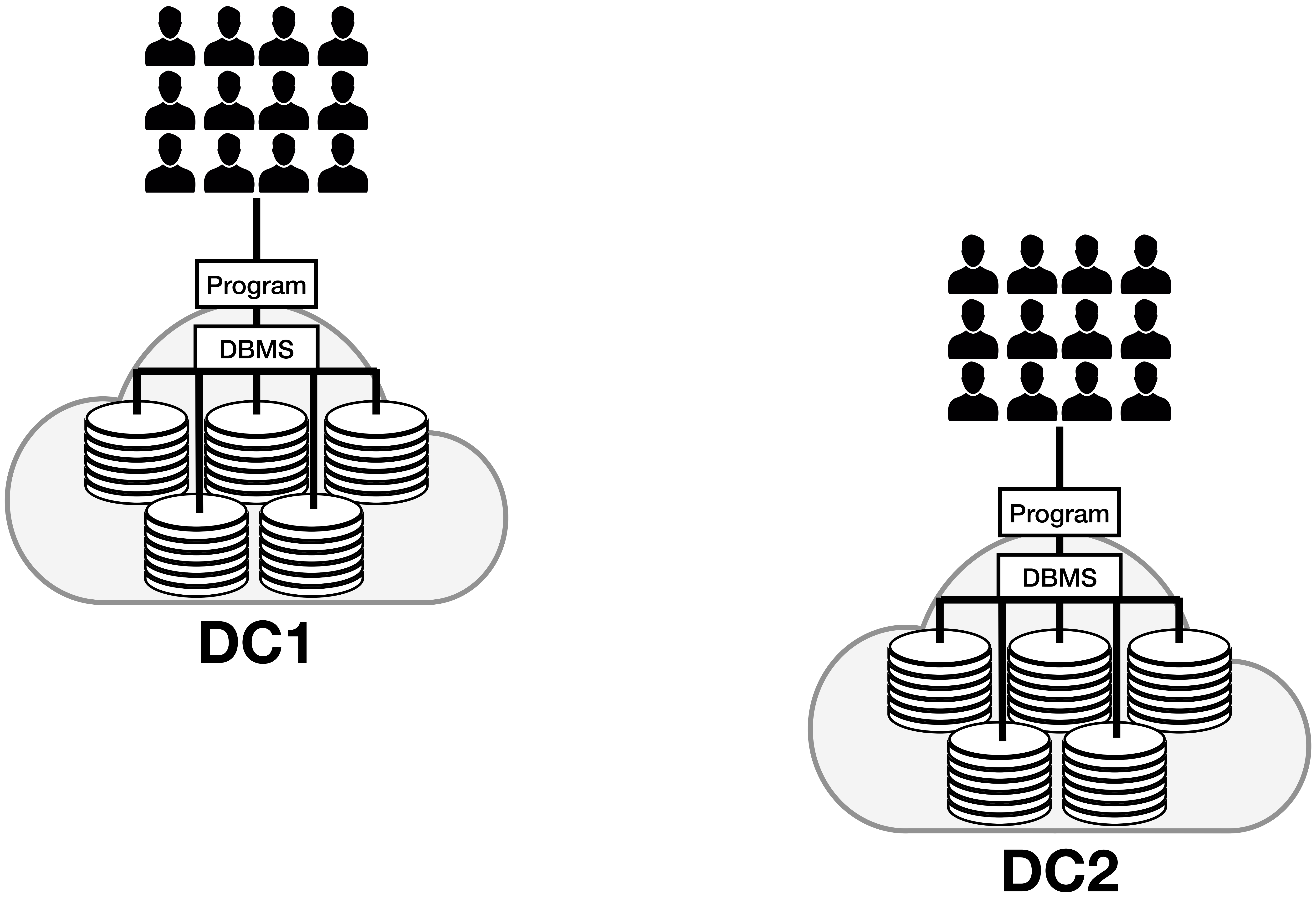
- Clustering architectures
  - Partitioning
  - Replicating



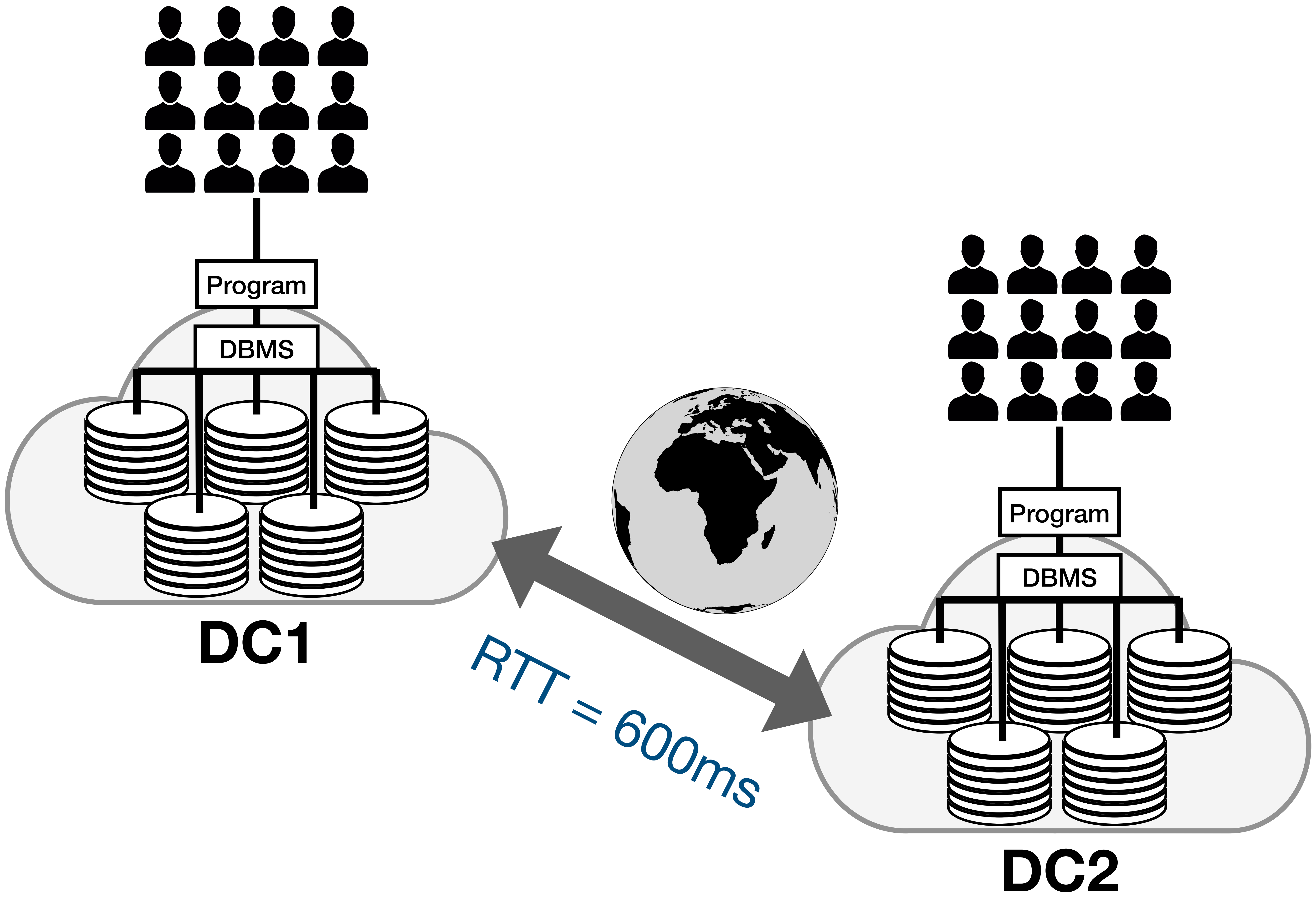
- Clustering architectures
  - Partitioning
  - Replicating



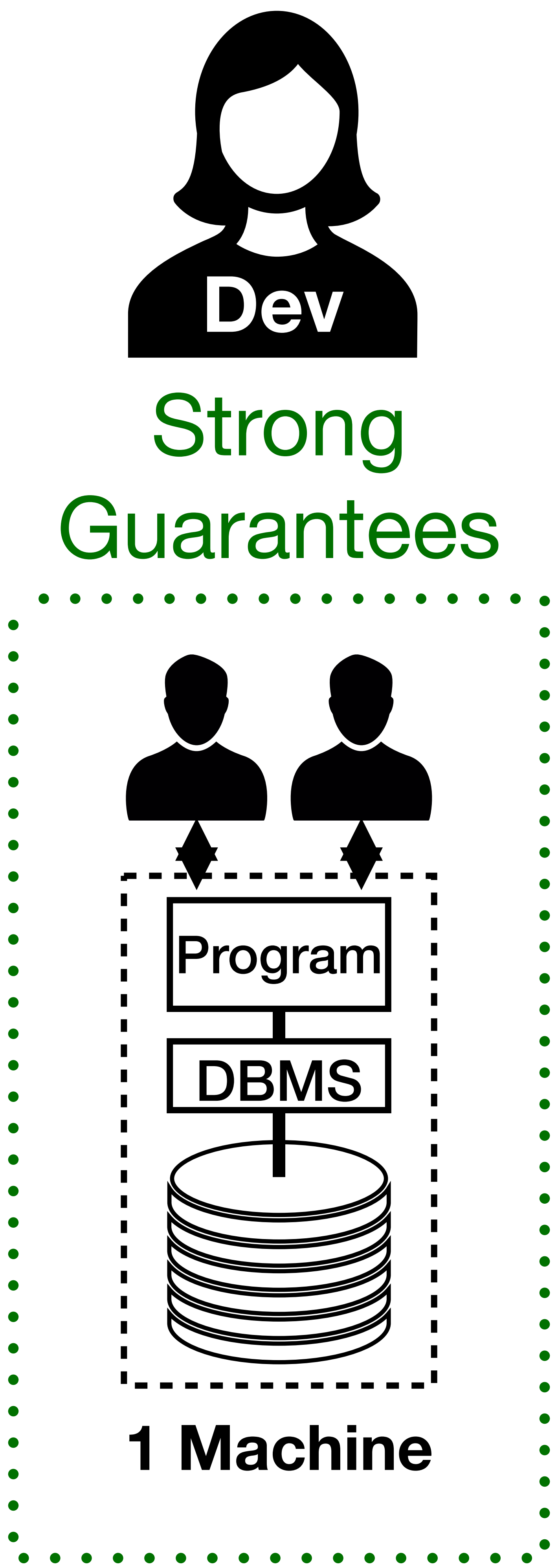
- Clustering architectures
  - Partitioning
  - Replicating



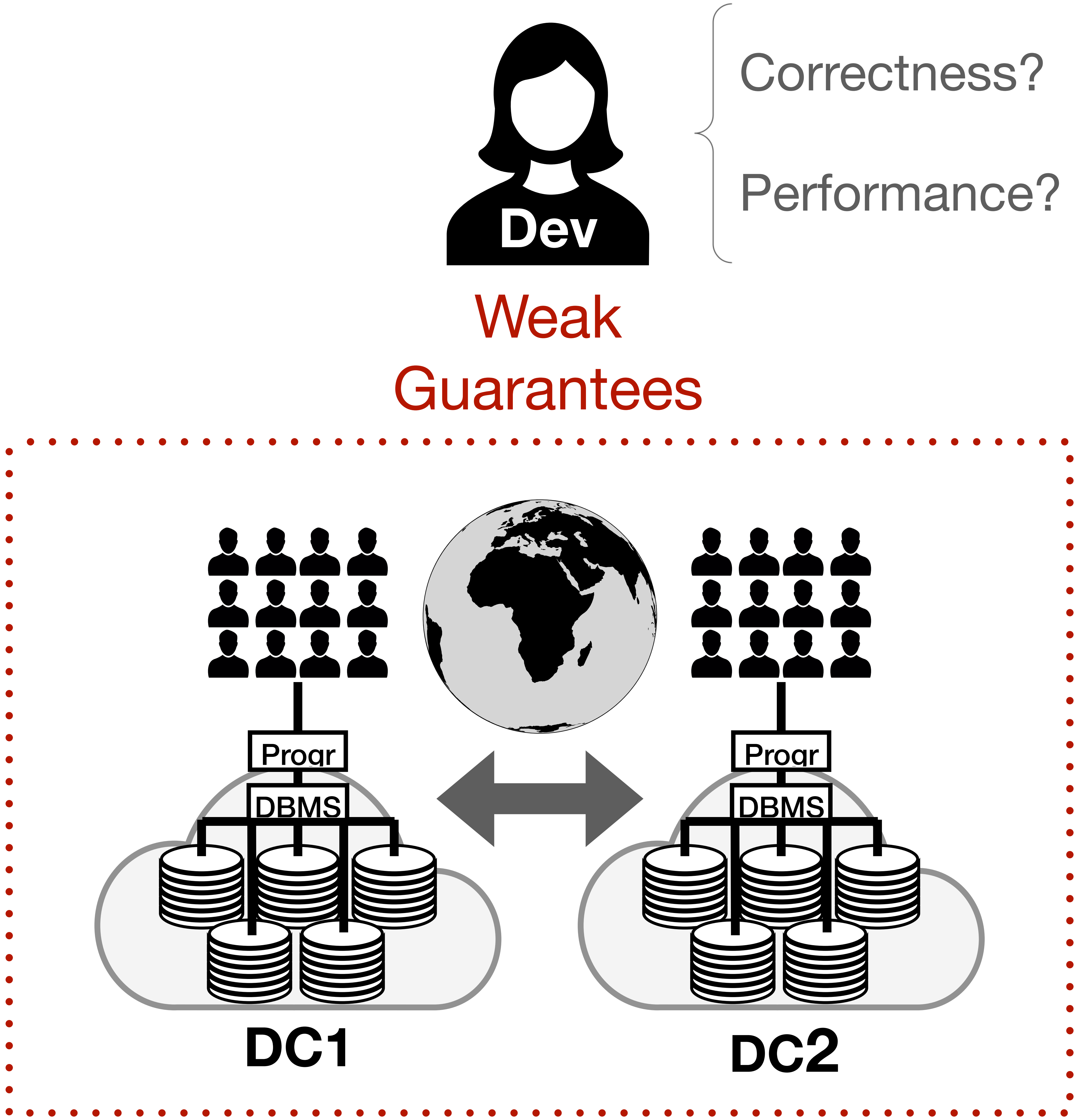
- Clustering architectures
  - Partitioning
  - Replicating



- Clustering architectures
  - Partitioning
  - Replicating
- Weak concurrency guarantees

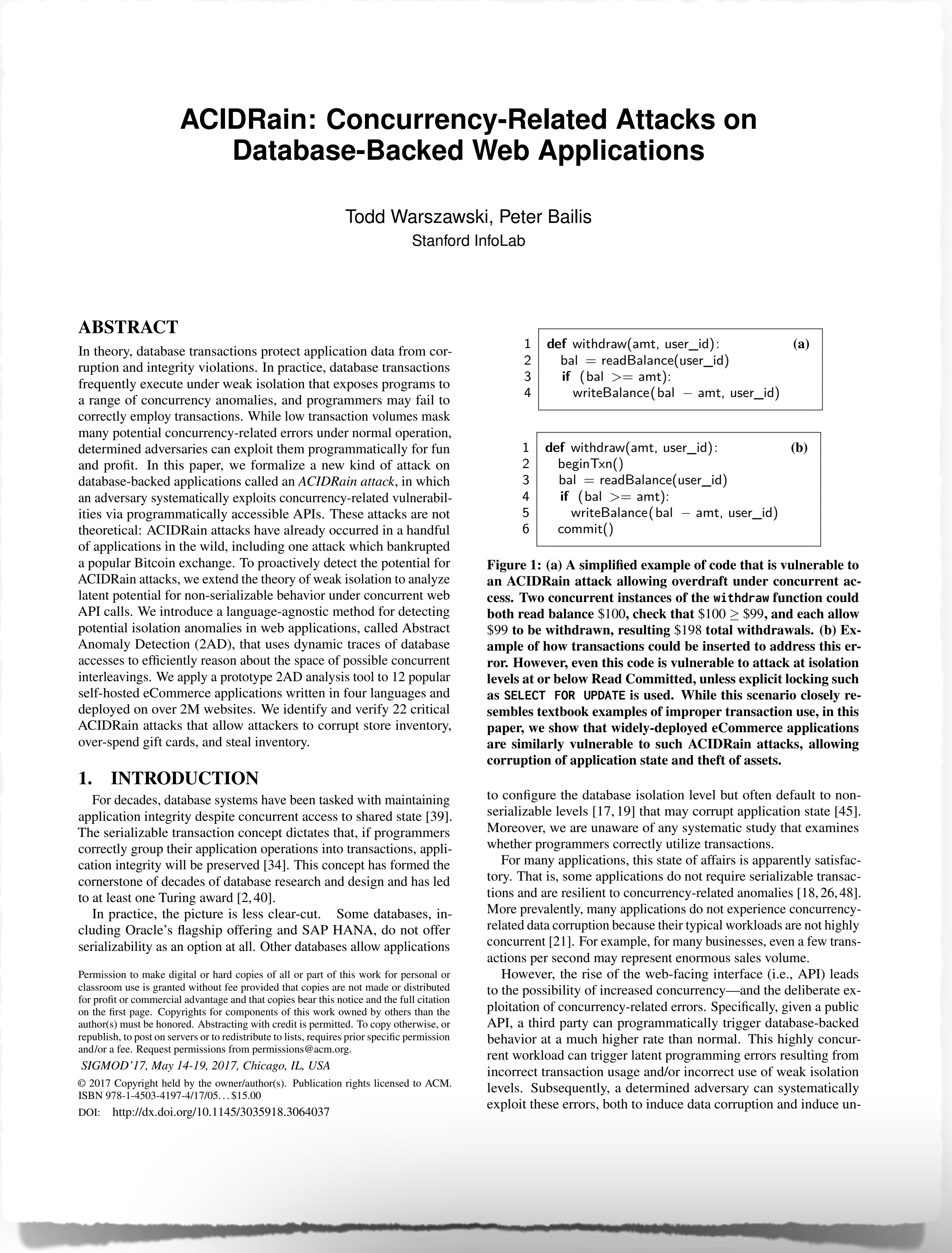


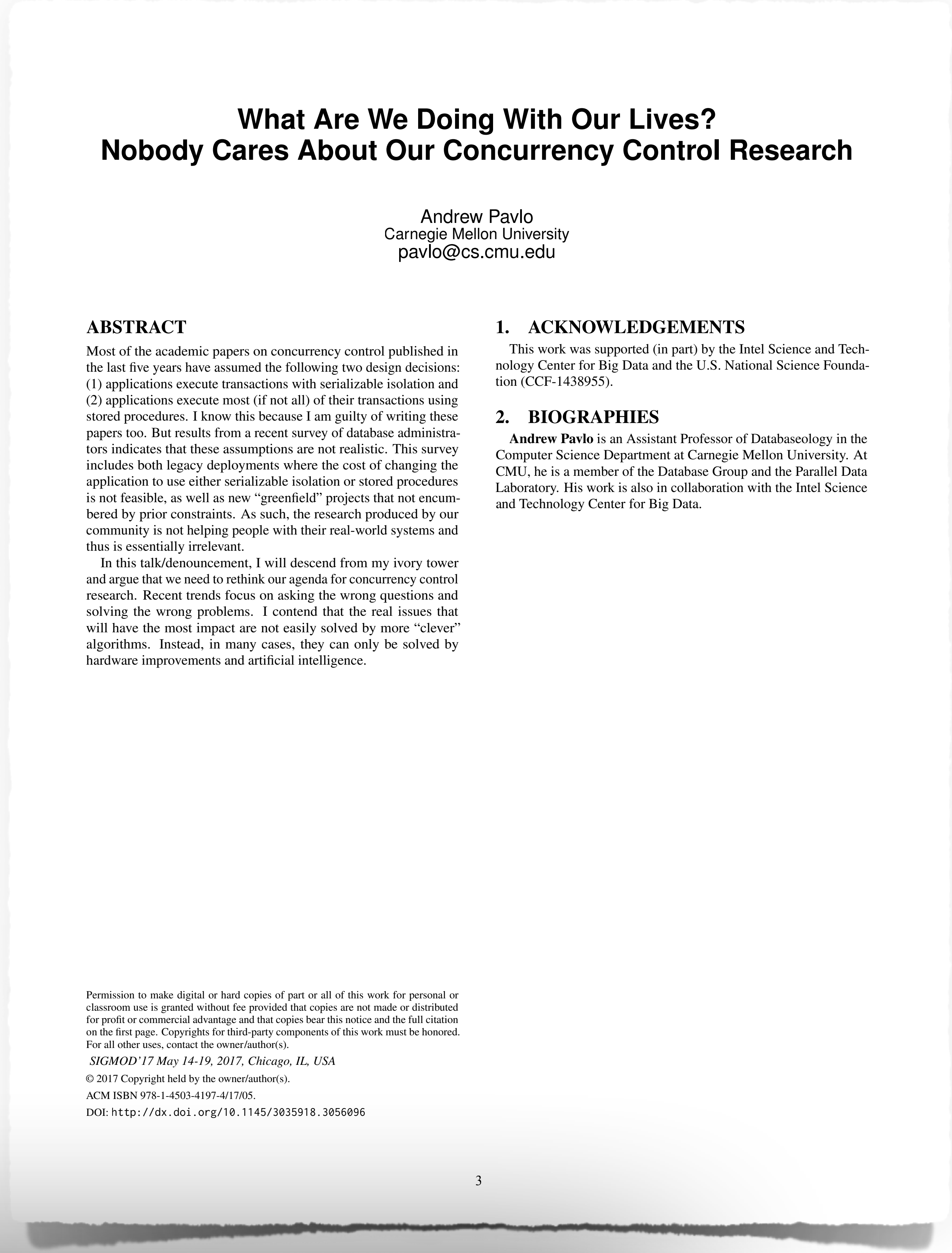
- Clustering architectures
  - Partitioning
  - Replicating
- Weak concurrency guarantees
  - Programmer is exposed to concurrency



WEAK CONCURRENCY SEMANTICS

- Clustering architectures
  - Partitioning
  - Replicating
- Weak concurrency guarantees
  - Programmer is exposed to concurrency
- Concurrency bugs
  - prevalent
  - dangerous
  - alerted database community





MY THESIS: THE OVERARCHING GOAL

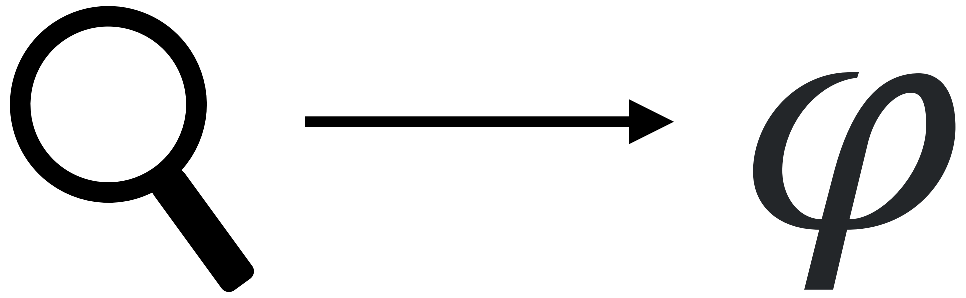
- Ease of strong semantics
- Performance of cloud-native

MY THESIS: THE OVERARCHING GOAL

- Ease of strong semantics
- Performance of cloud-native
- Language-oriented solutions
    - SMT Solvers
    - Model Checking

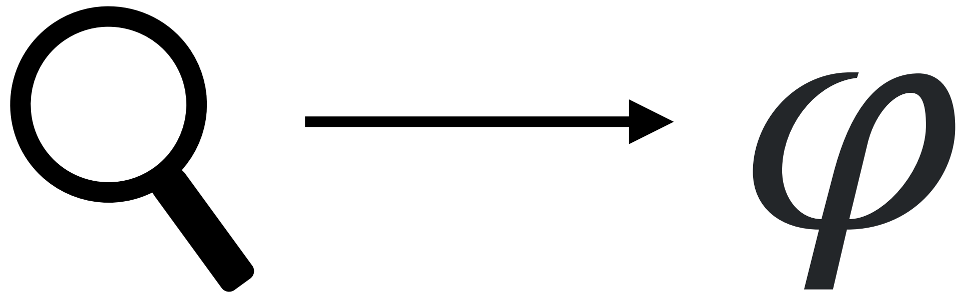
MY THESIS: THE OVERARCHING GOAL

- Ease of strong semantics
- Performance of cloud-native
- Language-oriented solutions
  - SMT Solvers
  - Model Checking



MY THESIS: THE OVERARCHING GOAL

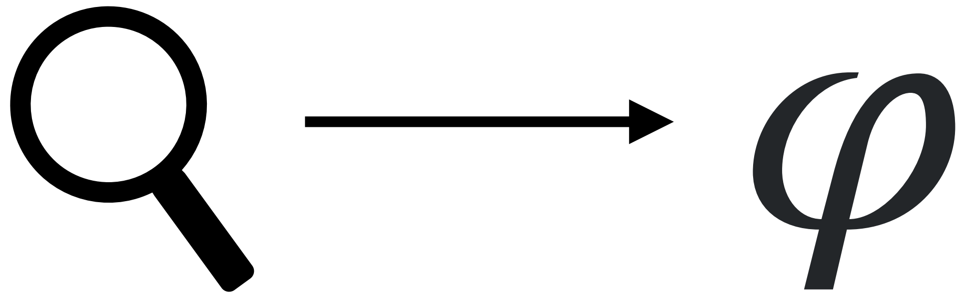
- Ease of strong semantics
- Performance of cloud-native
- Language-oriented solutions
  - SMT Solvers
  - Model Checking



CLOTHO	Guided Testing Framework	OOPSLA 2019
ATROPOS	Automated repair of replication anomalies	PLDI 2021
LACHESIS	Automated repair of partitioning anomalies	under submission

MY THESIS: THE OVERARCHING GOAL

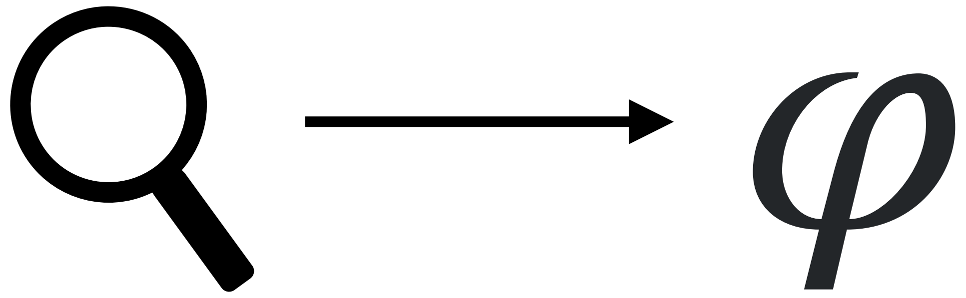
- Ease of strong semantics
- Performance of cloud-native
- Language-oriented solutions
  - SMT Solvers
  - Model Checking



CLOTHO	Guided Testing Framework	OOPSLA 2019
ATROPOS	Automated repair of replication anomalies	PLDI 2021
LACHESIS	Automated repair of partitioning anomalies	under submission

MY THESIS: THE OVERARCHING GOAL

- Ease of strong semantics
- Performance of cloud-native
- Language-oriented solutions
    - SMT Solvers
    - Model Checking

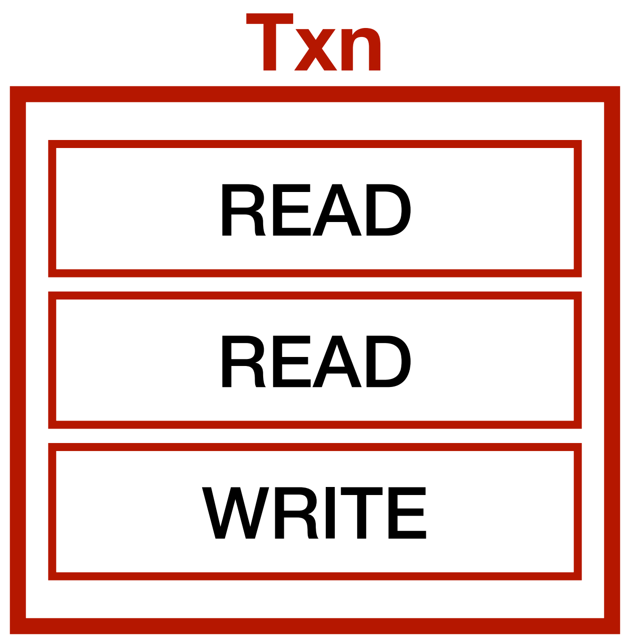


CLOTHO	Guided Testing Framework	OOPSLA 2019
ATROPOS	Automated repair of replication anomalies	PLDI 2021
LACHESIS	Automated repair of partitioning anomalies	under submission

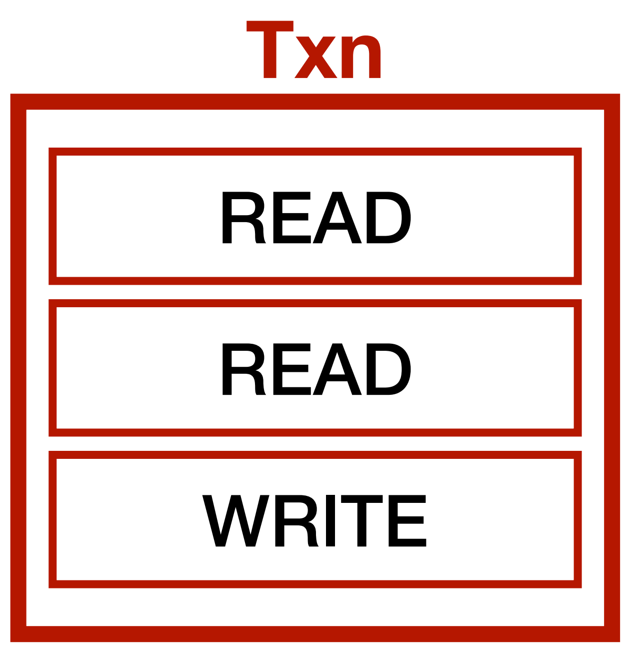
CLOTHO

(SYMBOLIC PROGRAM ANALYSIS & GUIDED TESTING)

- **ACID** guarantees



- **ACID** guarantees
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

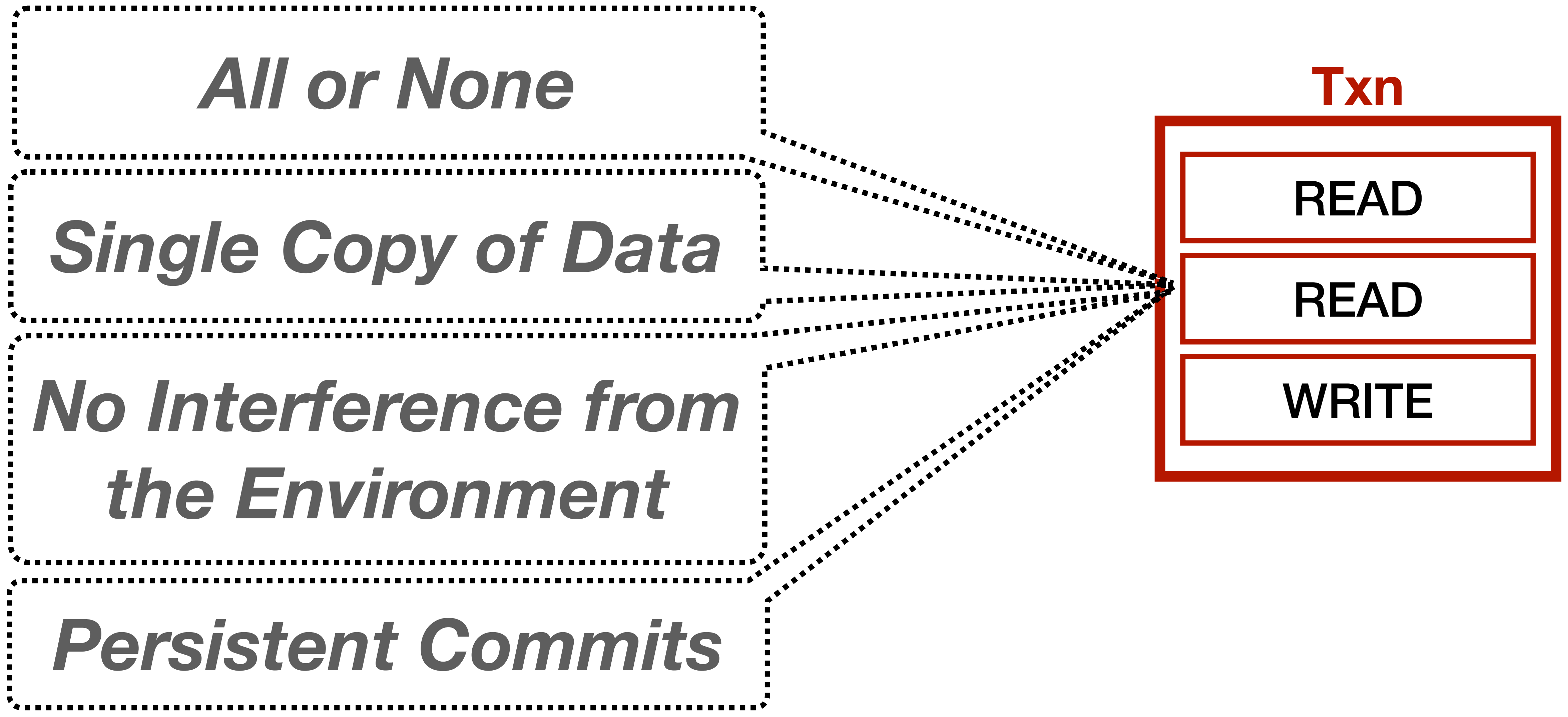


- **ACID** guarantees
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

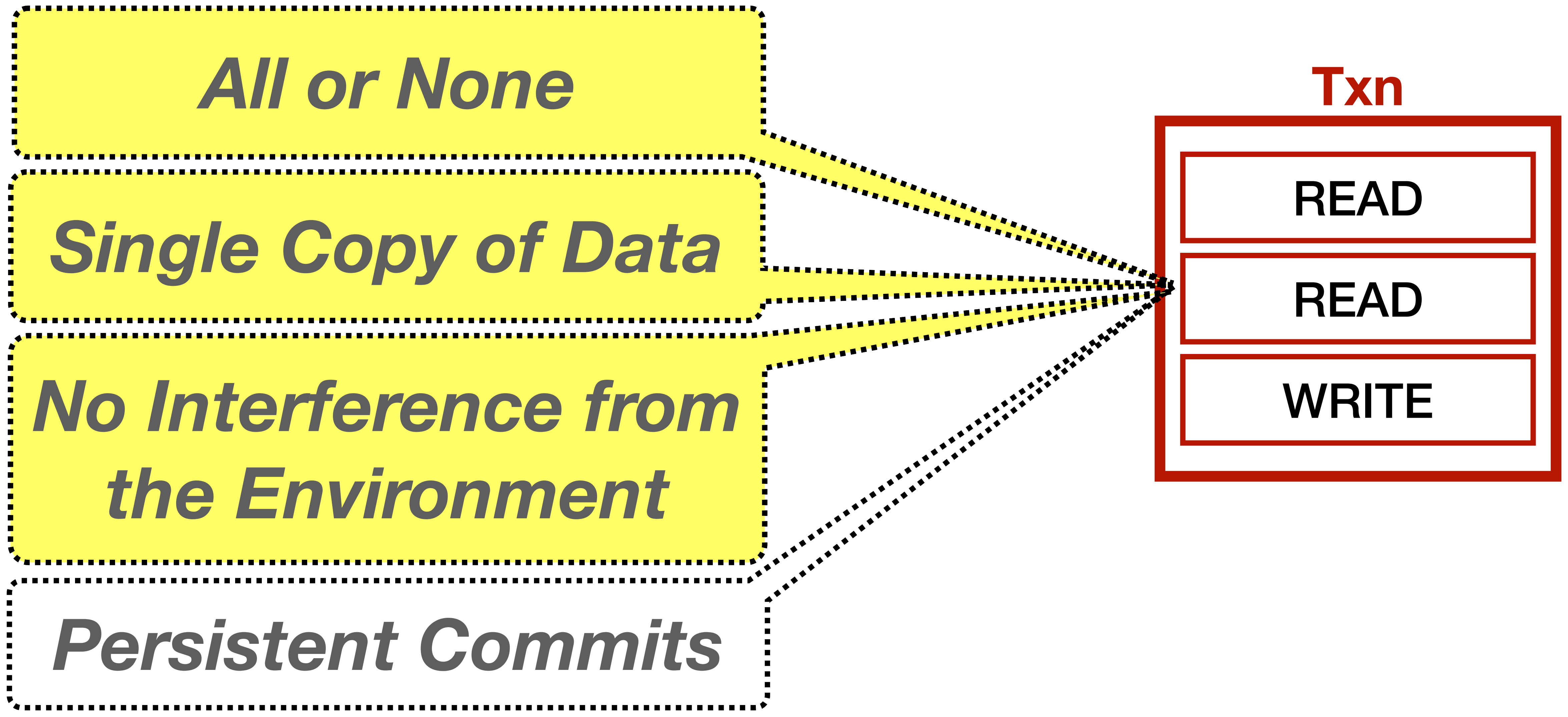
- **ACID** guarantees
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

- **ACID** guarantees
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

- **ACID** guarantees
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

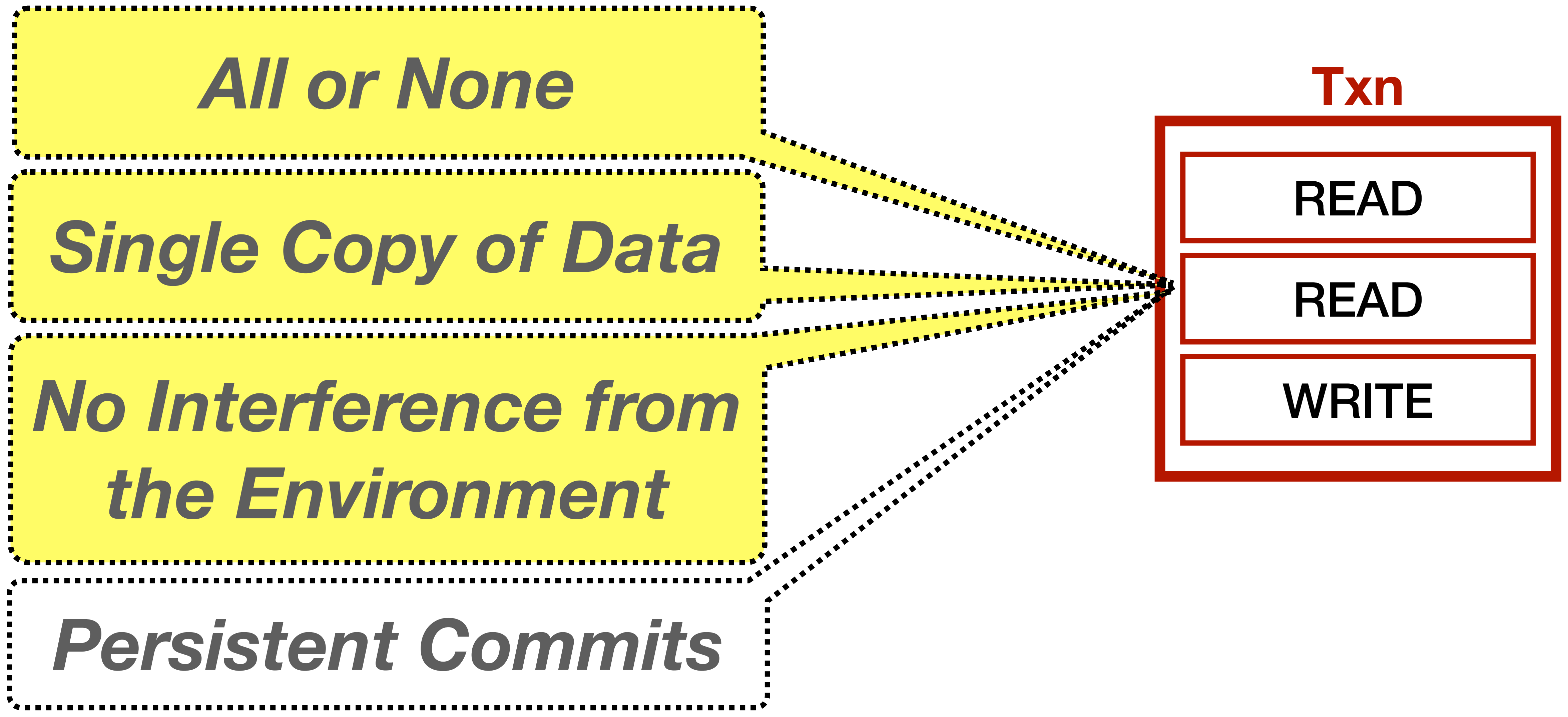


- **ACID** guarantees
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability
- Serializability



TRANSACTIONAL GUARANTEES

- **ACID** guarantees
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability
- Serializability
- Facilitates program design and reasoning



- Serializability is costly

SERIALIZABILITY

- Serializability is costly
- Developers forced to use weaker semantics
  - Transactions are interleaved
  - Asynchronous replication of data

SERIALIZABILITY

- Serializability is costly
- Developers forced to use weaker semantics
  - Transactions are interleaved
  - Asynchronous replication of data
- Less intuitive behaviors

SERIALIZABILITY

- Serializability is costly
- Developers forced to use weaker semantics
  - Transactions are interleaved
  - Asynchronous replication of data
- Less intuitive behaviors

Serializability Anomalies

EXAMPLE

- Online gaming platform

EXAMPLE (TABLES)

- Online gaming platform
  - 2 tables

PLAYER

p_id	p_t_id	p_role	p_stat

TEAM

t_id	t_name	p_cnt

EXAMPLE (TABLES)

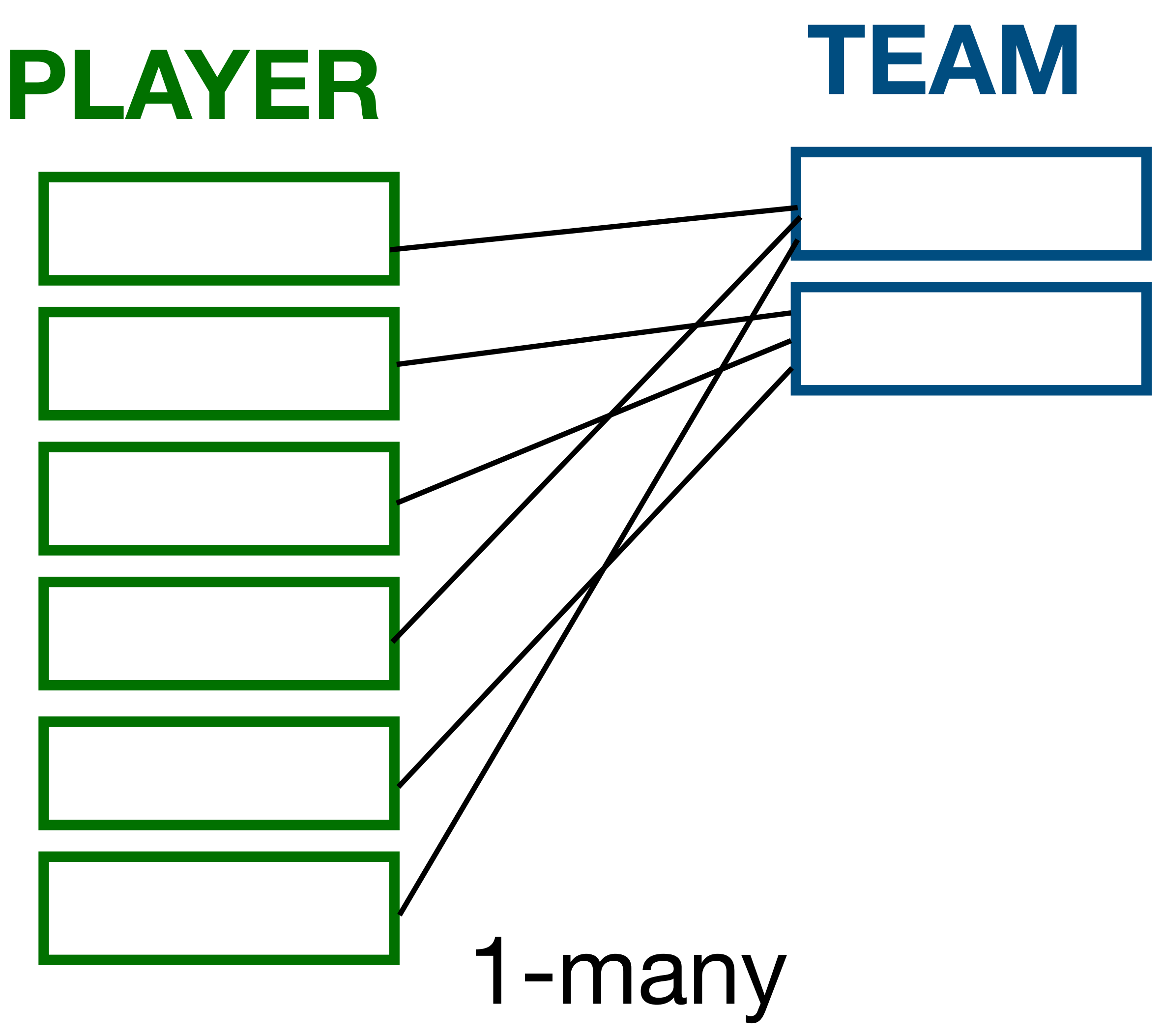
- Online gaming platform
  - 2 tables

PLAYER

p_id	p_t_id	p_role	p_stat

TEAM

t_id	t_name	p_cnt



EXAMPLE (TRANSACTIONS)

- Online gaming platform
  - 2 tables
  - 3 transactions

PLAYER

p_id	p_t_id	p_role	p_stat

TEAM

t_id	t_name	p_cnt

EXAMPLE (TRANSACTIONS)

- Online gaming platform
  - 2 tables
  - 3 transactions

PLAYER

p_id	p_t_id	p_role	p_stat

TEAM

t_id	t_name	p_cnt

addPlayer(*p*, *t*)

```
old_cnt := select p_cnt from TEAM where t_id=t
Insert (p,t,∅,∅) into PLAYER
update TEAM set p_cnt=old_cnt+1 where t_id=t
```

EXAMPLE (TRANSACTIONS)

- Online gaming platform
  - 2 tables
  - 3 transactions

PLAYER

p_id	p_t_id	p_role	p_stat

TEAM

t_id	t_name	p_cnt

addPlayer(*p*, *t*)

```
old_cnt := select p_cnt from TEAM where t_id=t
Insert (p,t,∅,∅) into PLAYER
update TEAM set p_cnt=old_cnt+1 where t_id=t
```

PlayerByTeam (t)

```
ps := select * from PLAYER where p_t_id=t
foreach p in ps do
  print(p.*)
```

EXAMPLE (TRANSACTIONS)

- Online gaming platform
  - 2 tables
  - 3 transactions

PLAYER

p_id	p_t_id	p_role	p_stat

TEAM

t_id	t_name	p_cnt

addPlayer(*p*, *t*)

```
old_cnt := select p_cnt from TEAM where t_id=t
Insert (p,t,∅,∅) into PLAYER
update TEAM set p_cnt=old_cnt+1 where t_id=t
```

PlayerByTeam (t)

```
ps := select * from PLAYER where p_t_id=t
foreach p in ps do
  print(p.*)
```

setPlayer (*p*, *r*, *s*)

```
update PLAYER set p_role=r and p_stat=p where p_id=p
```

EXAMPLE (ANOMALY)

TEAM

t_id	t_name	p_cnt
T1	"A"	2

addPlayer(p=P0, t=T1)

Select p_cnt // 2

...

Update p_cnt = 3

addPlayer(p=P0, t=T1)

Select p_cnt // 2

...

Update p_cnt = 3

EXAMPLE (ANOMALY)

TEAM

t_id	t_name	p_cnt
T1	“A”	2

addPlayer(p=P0, t=T1)

1 Select p_cnt // 2

...

3 Update p_cnt = 3

addPlayer(p=P0, t=T1)

2 Select p_cnt // 2

...

4 Update p_cnt = 3

EXAMPLE (ANOMALY)

TEAM

t_id	t_name	p_cnt
T1	“A”	2

addPlayer(p=P0, t=T1)

1 **Select p_cnt // 2**

...

3 **Update p_cnt = 3**

addPlayer(p=P0, t=T1)

2 **Select p_cnt // 2**

...

4 **Update p_cnt = 3**

TEAM

t_id	t_name	p_cnt
T1	“A”	3

EXAMPLE (ANOMALY)

TEAM

t_id	t_name	p_cnt
T1	“A”	2

addPlayer(p=P0, t=T1)

1 Select p_cnt // 2

...

3 Update p_cnt = 3

addPlayer(p=P0, t=T1)

2 Select p_cnt // 2

...

4 Update p_cnt = 3

TEAM

t_id	t_name	p_cnt
T1	“A”	3

Lost Update

EXAMPLE (ANOMALY)

TEAM

t_id	t_name	p_cnt
T1	“A”	2

addPlayer(p=P0, t=T1)

1 **Select p_cnt // 2**

...

3 **Update p_cnt = 3**

addPlayer(p=P0, t=T1)

2 **Select p_cnt // 2**

...

4 **Update p_cnt = 3**

“The column ‘p_cnt’ reflects the number of players in each team”

TEAM

t_id	t_name	p_cnt
T1	“A”	3

Lost Update

EXAMPLE (ANOMALY)

TEAM

t_id	t_name	p_cnt
T1	“A”	2

addPlayer(p=P0, t=T1)

1 **Select p_cnt // 2**

...

3 **Update p_cnt = 3**

addPlayer(p=P0, t=T1)

2 **Select p_cnt // 2**

...

4 **Update p_cnt = 3**

“The column ‘p_cnt’ reflects the number of players in each team”

Invariant Violated!

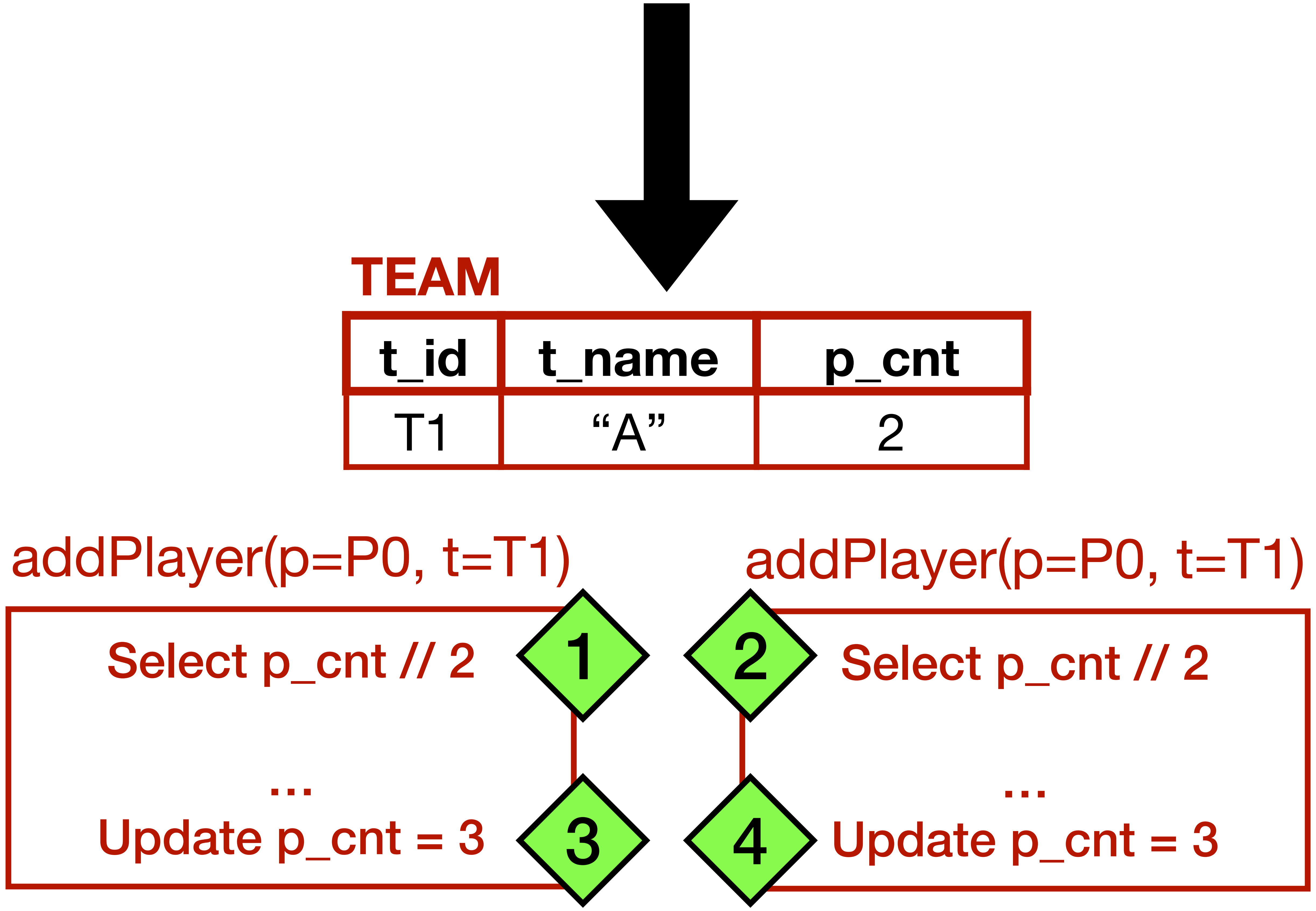
TEAM

t_id	t_name	p_cnt
T1	“A”	3

Lost Update

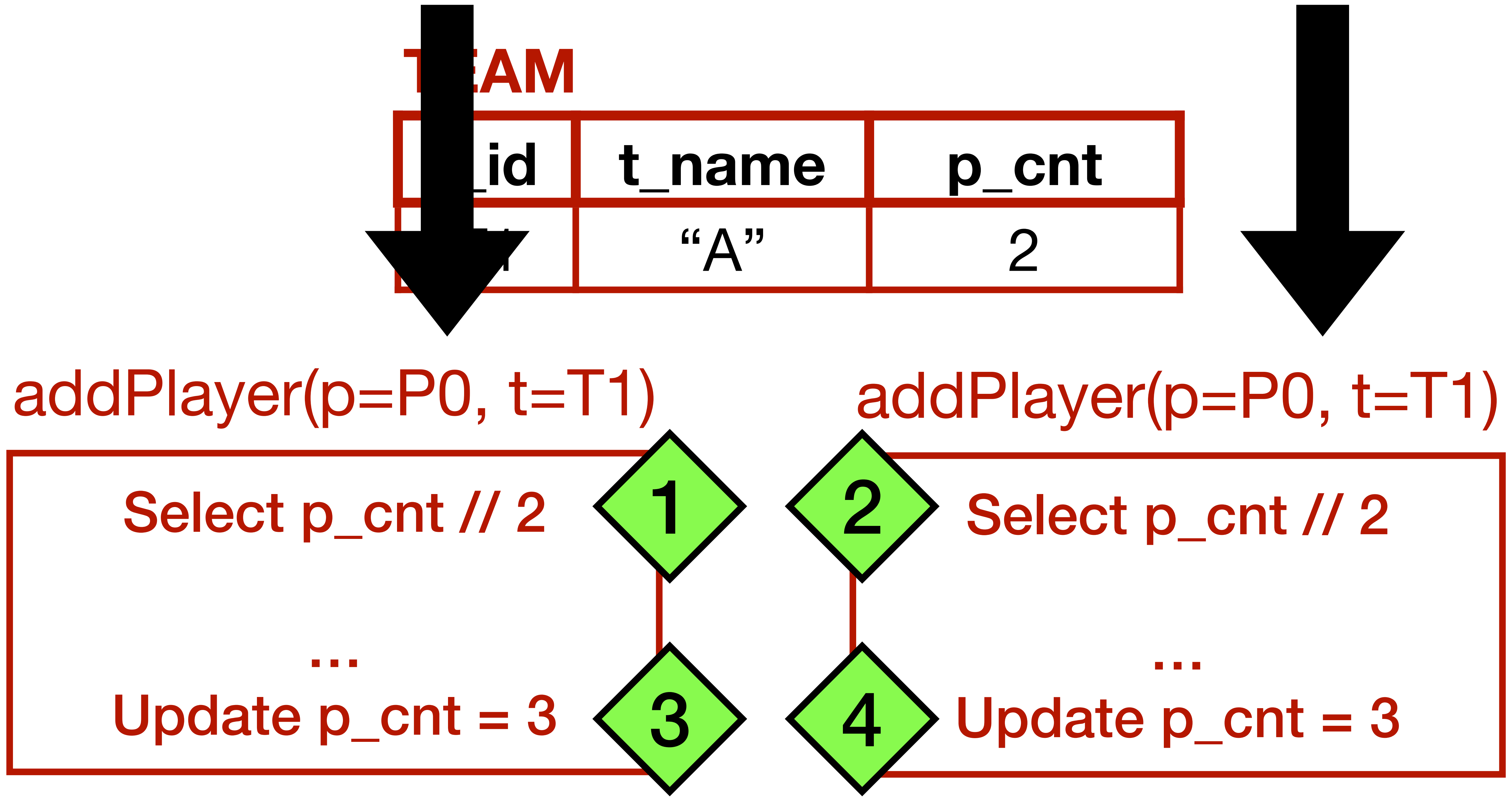
- Serializability anomalies are subtle

- Serializability anomalies are subtle
  - initial state



CHALLENGES OF TESTING

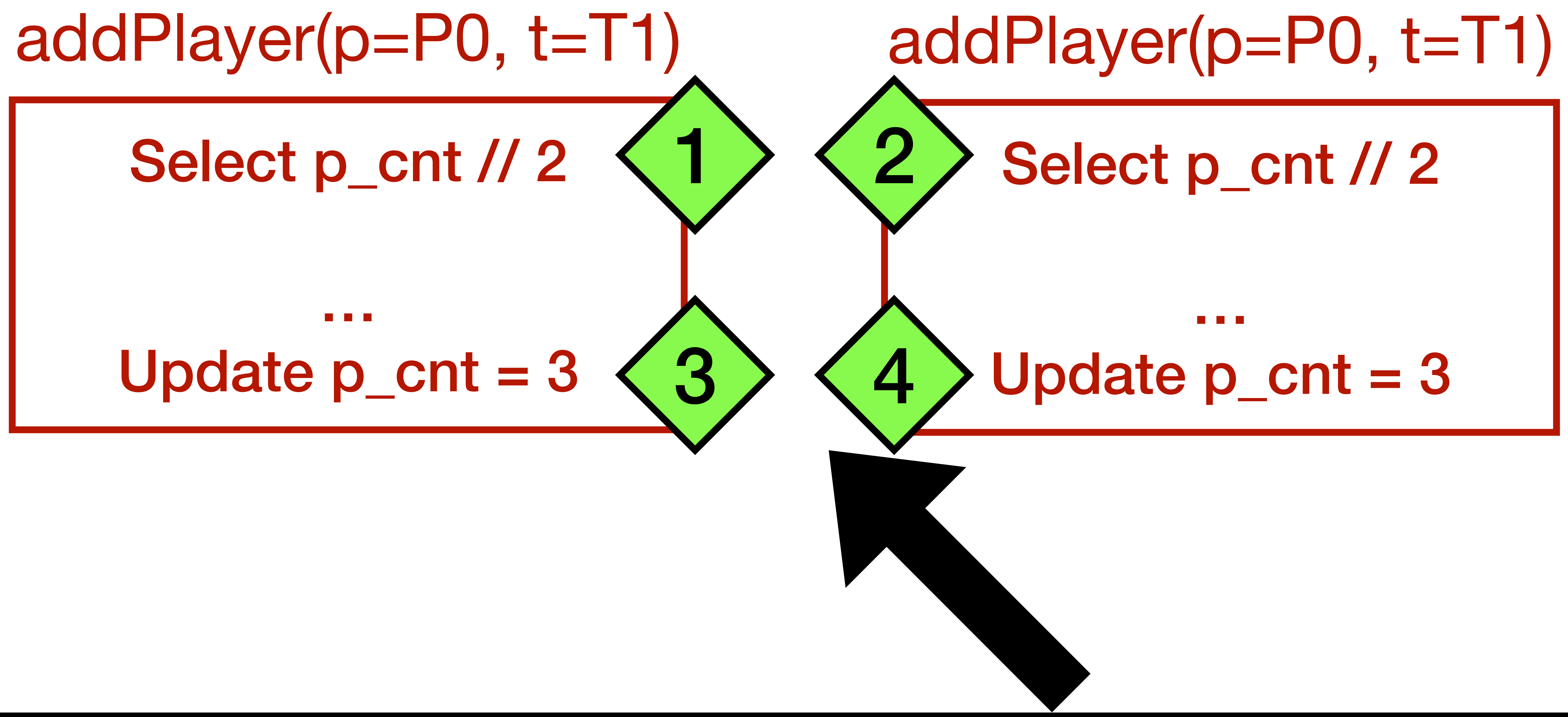
- Serializability anomalies are subtle
  - initial state
  - transaction arguments



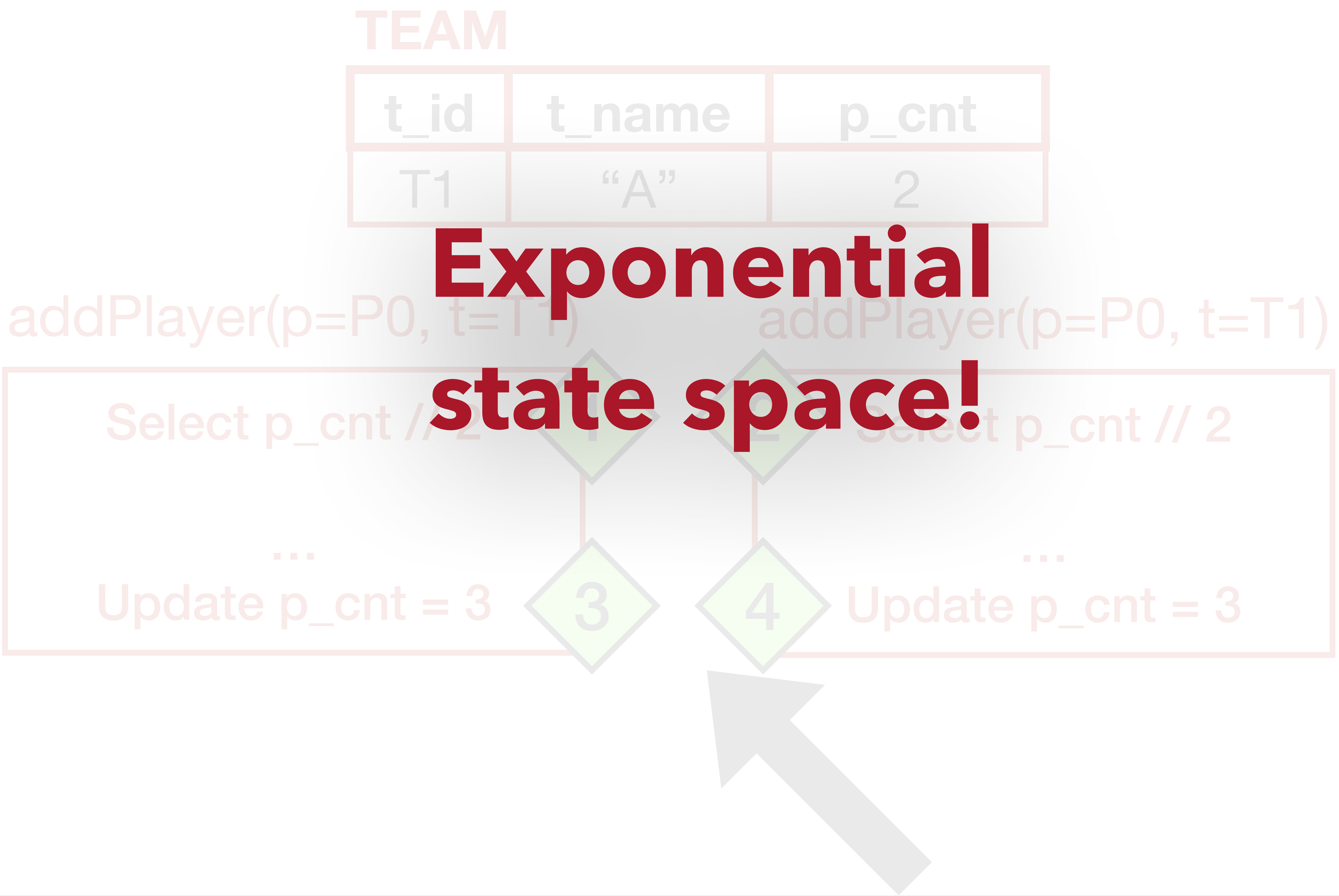
- Serializability anomalies are subtle
  - initial state
  - transaction arguments
  - interleaved order

TEAM

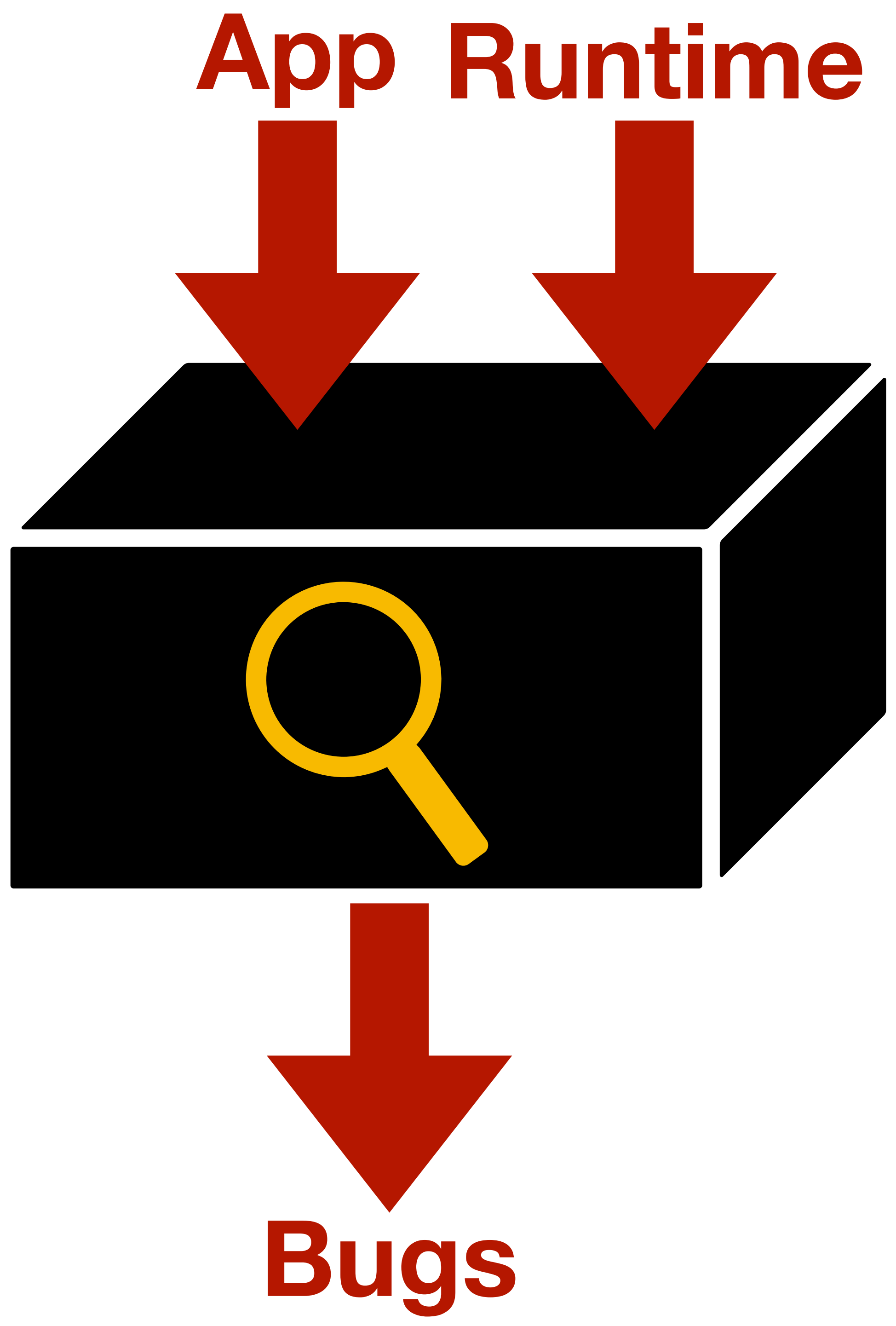
t_id	t_name	p_cnt
T1	“A”	2



- Serializability anomalies are subtle
  - initial state
  - transaction arguments
  - interleaved order

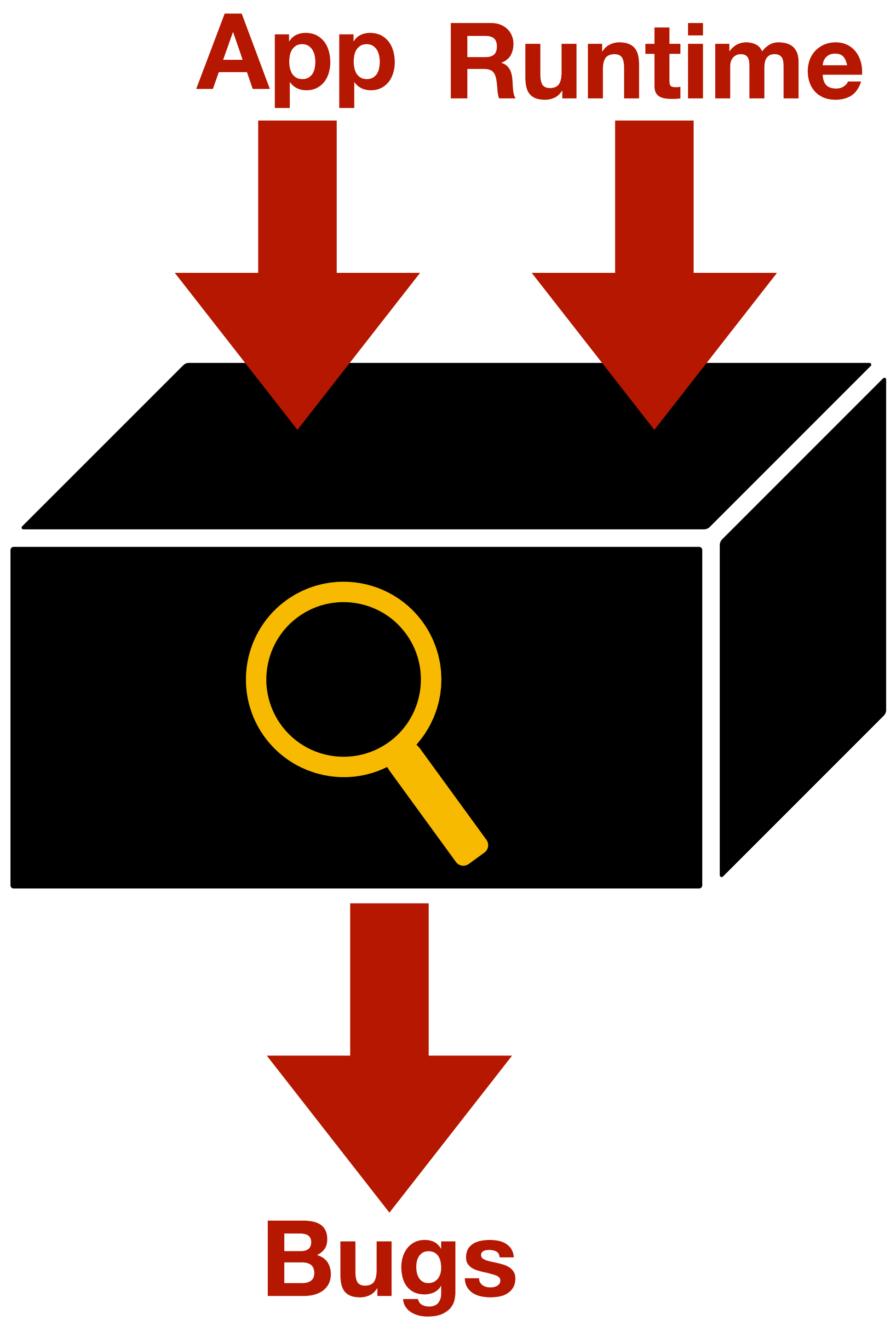


BLACKBOX TESTING

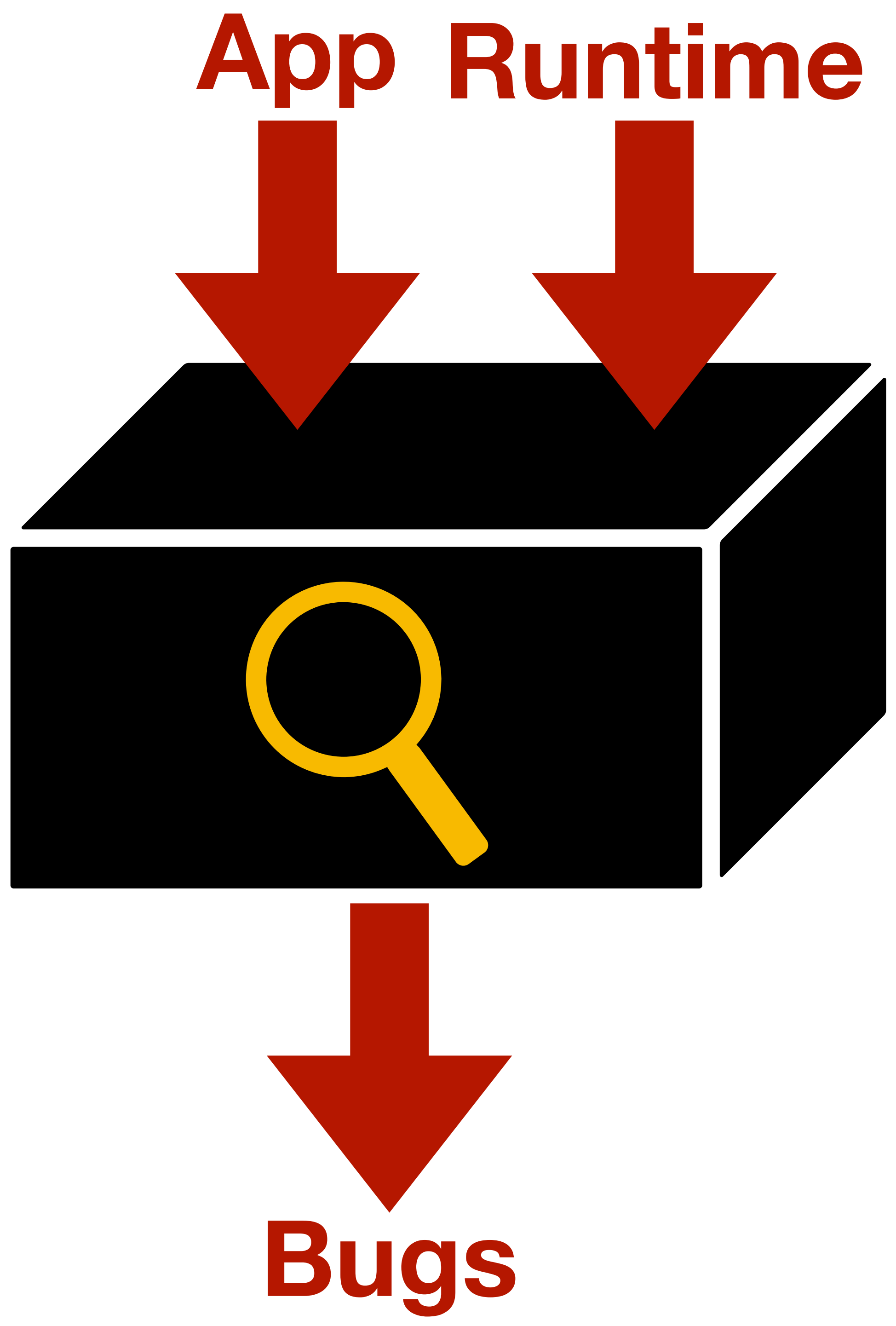


BLACKBOX TESTING

- Run and monitor apps in (semi-) production environment

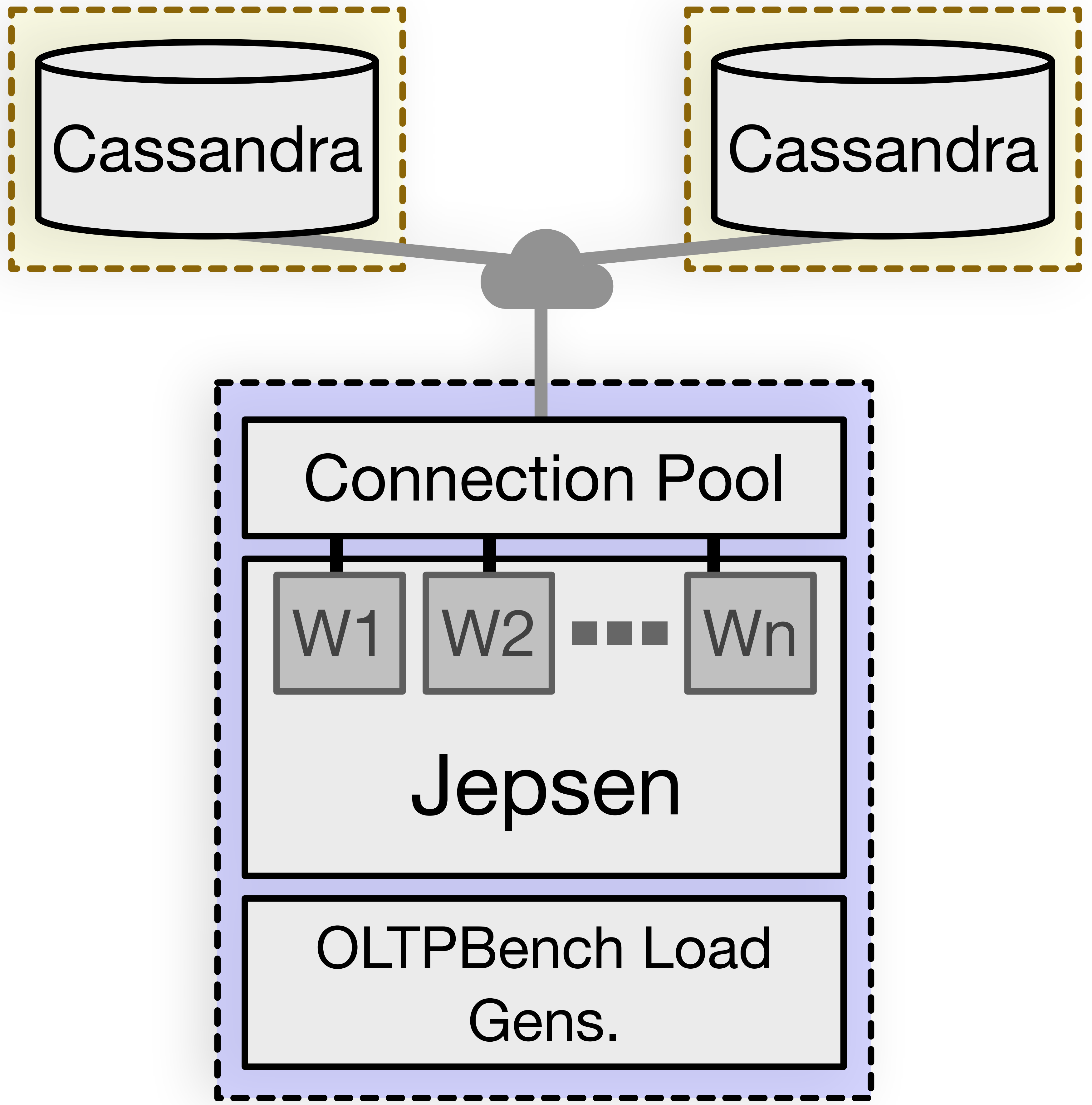


- Run and monitor apps in (semi-) production environment
  + real bugs (no false positive)
  - costly
  - too specific
  - no guarantee of coverage
  - manual effort



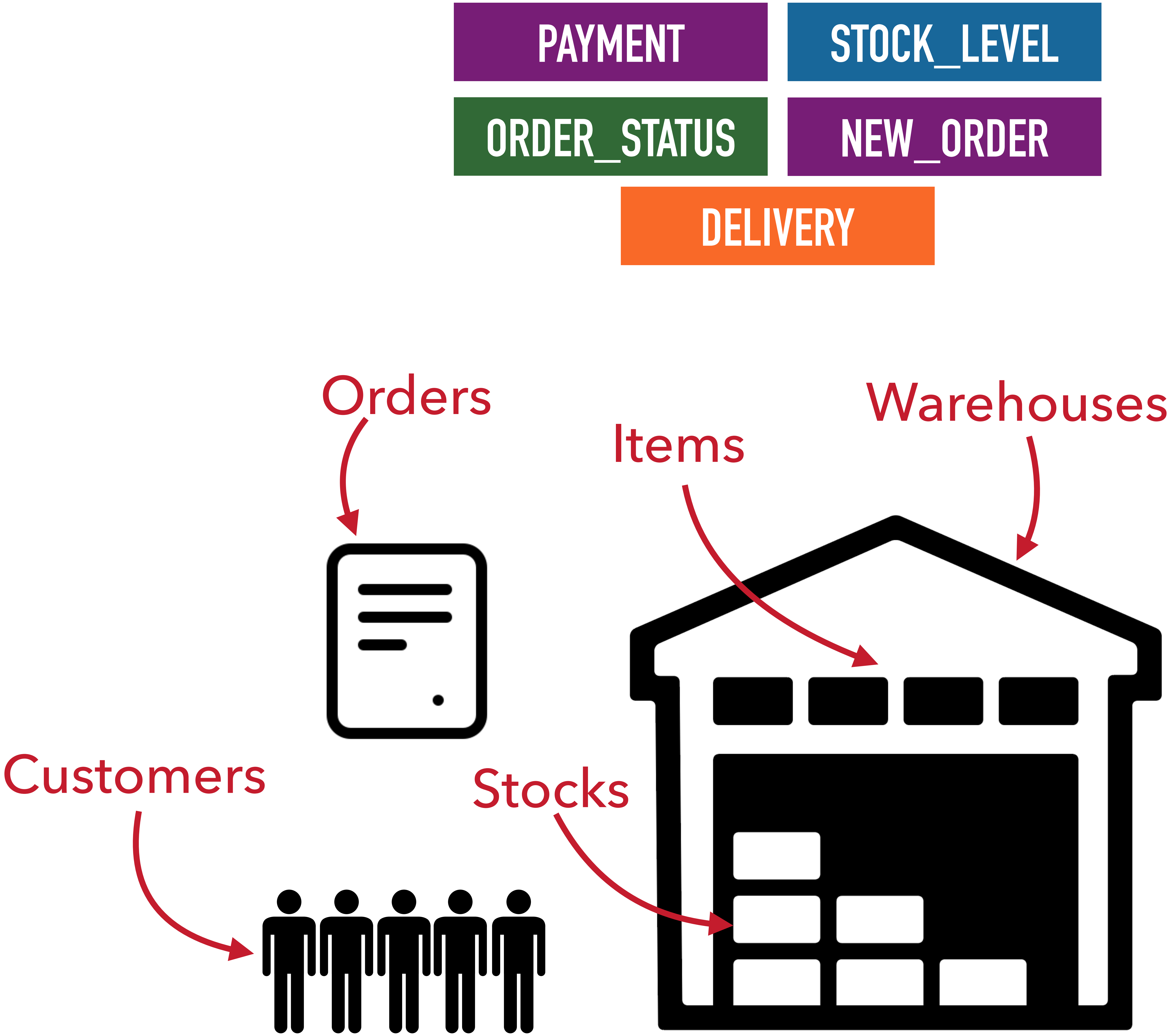
BLACKBOX TESTING: STATE OF THE ART

- Cloud-based testing framework
  - OLTPBench
  - Jepsen



BLACKBOX TESTING: STATE OF THE ART

- Cloud-based testing framework
  - OLTPBench
  - Jepsen
- TPC-C Benchmark
  - 5 txn and 9 tables



PURDUE UNIVERSITY

- Cloud-based testing framework
  - OLTPBench
  - Jepsen
- TPC-C Benchmark
  - 5 txn and 9 tables
  - 21 application-level invariants

Invariant
CR1
CR2
CR3
CR4
CR5A
CR5B
CR6
CR7A
CR7B
CR8
CR9
CR10
CR11
CR12
NCR1
NCR2
NCR3
NCR4
NCR5
NCR6
NCR7

BLACKBOX TESTING: STATE OF THE ART

- Cloud-based testing framework
  - OLTPBench
  - Jepsen
- TPC-C Benchmark
  - 5 txn and 9 tables
  - 21 application-level invariants
  - only 14 invariants broken

Invariant	Broken?
CR1	Y
CR2	Y
CR3	Y
CR4	Y
CR5A	N
CR5B	N
CR6	Y
CR7A	N
CR7B	N
CR8	Y
CR9	Y
CR10	Y
CR11	Y
CR12	Y
NCR1	Y
NCR2	Y
NCR3	N
NCR4	N
NCR5	Y
NCR6	Y
NCR7	N

ANOTHER SOLUTION?

- Cloud-based testing framework
  - OLTPBench
  - Jepsen
- TPC-C Benchmark
  - 5 t
  - 21
  - or

Invariant
CR1
CR2
CR3
CR4
CR5A
CR5B
CR12
NCR1
NCR2
NCR3
NCR4
NCR5
NCR6
NCR7

— KEY IDEA —

1. Statically analyze programs and find abstract anomalies (white-box testing)
2. Construct and run anomalous executions to the devs

- Non-serializable executions in a program, iff
    - Cyclic dependencies between txns

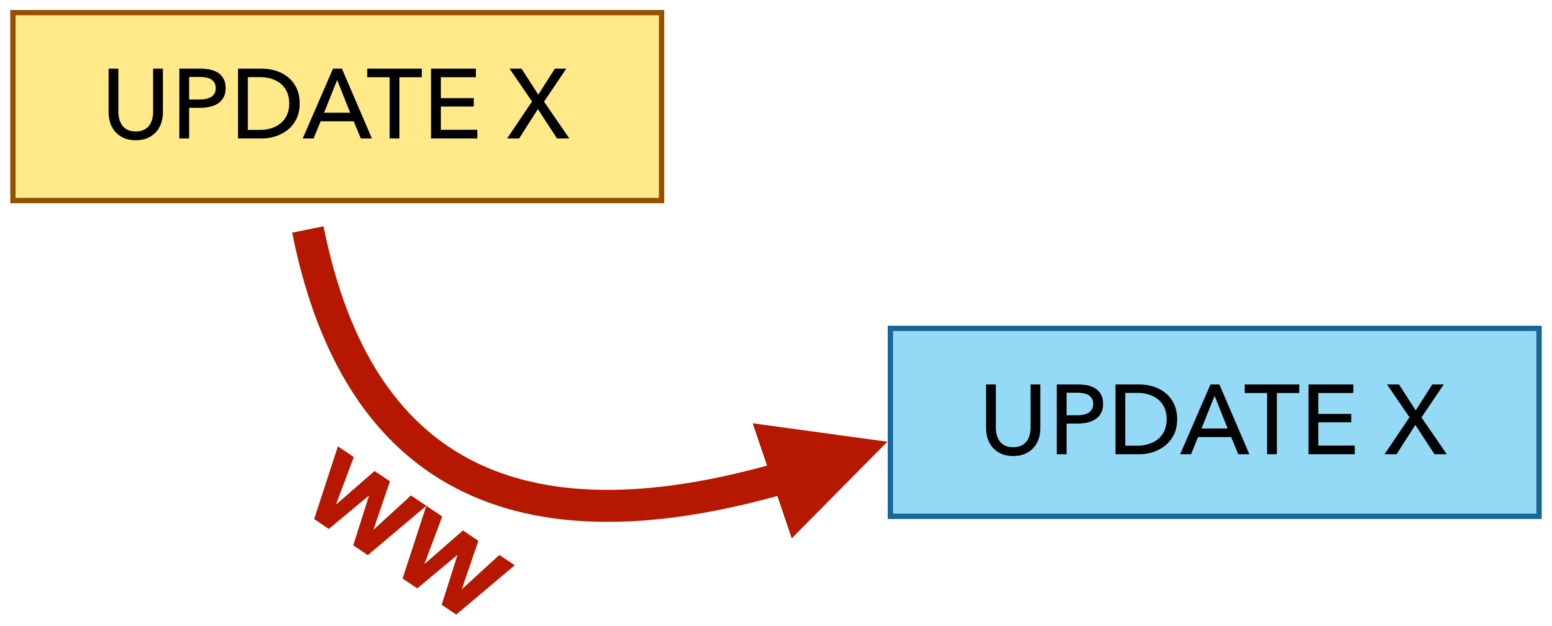
WHITE BOX ANALYSIS

- Non-serializable executions in a program, iff
  - Cyclic dependencies between txns
- Executions abstracted by directed dependency graphs

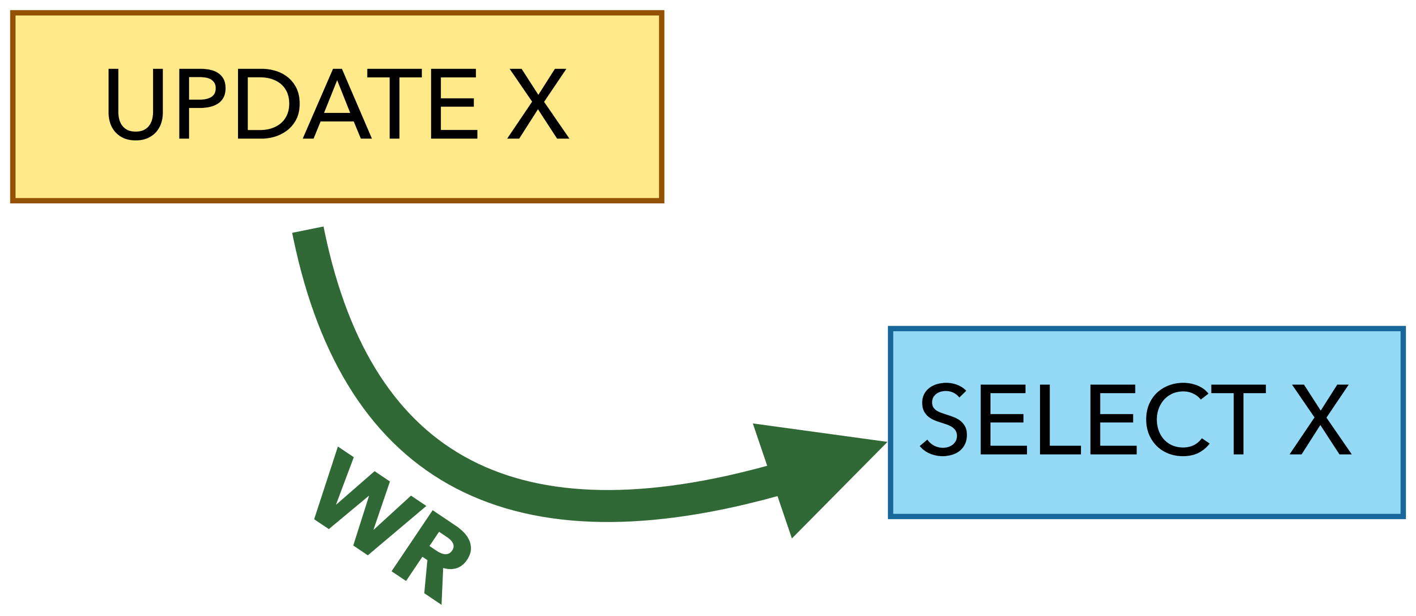
WHITE BOX ANALYSIS

- Non-serializable executions in a program, iff
  - Cyclic dependencies between txns
- Executions abstracted by directed dependency graphs
- Three types of dependency edges

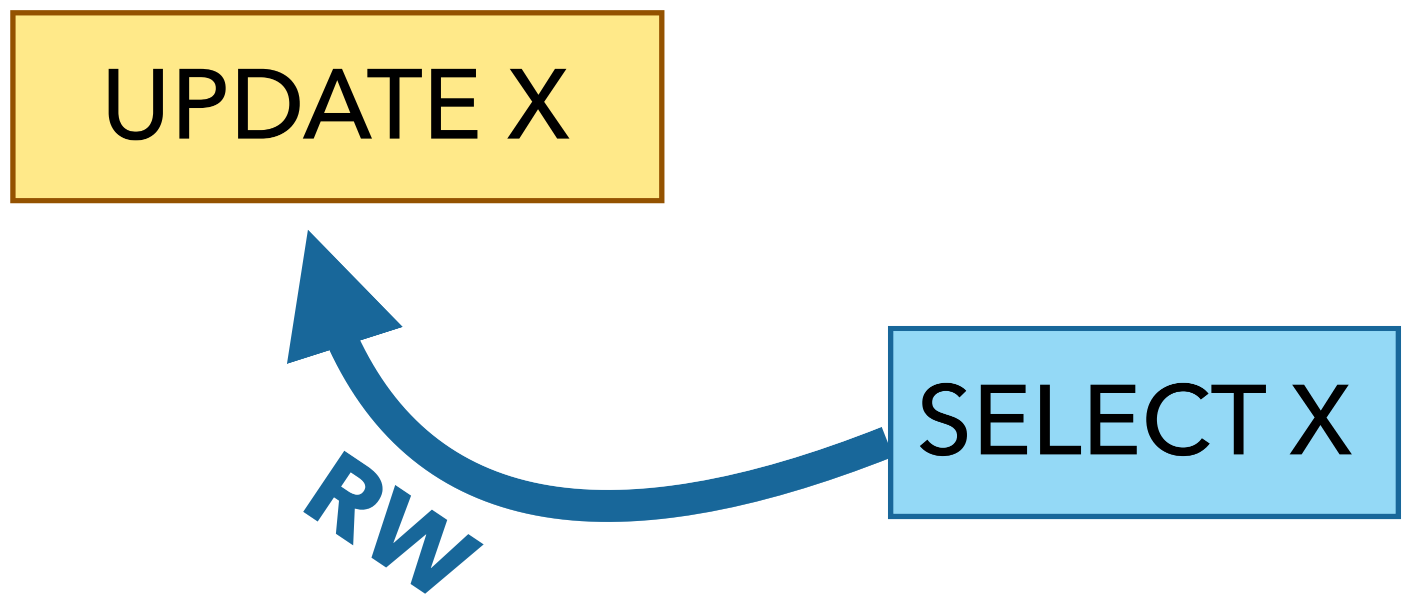
- Non-serializable executions in a program, iff
  - Cyclic dependencies between txns
- Executions abstracted by directed dependency graphs
- Three types of dependency edges
  - write dependency (WW)



- Non-serializable executions in a program, iff
  - Cyclic dependencies between txns
- Executions abstracted by directed dependency graphs
- Three types of dependency edges
  - write dependency (WW)
  - read dependency (WR)

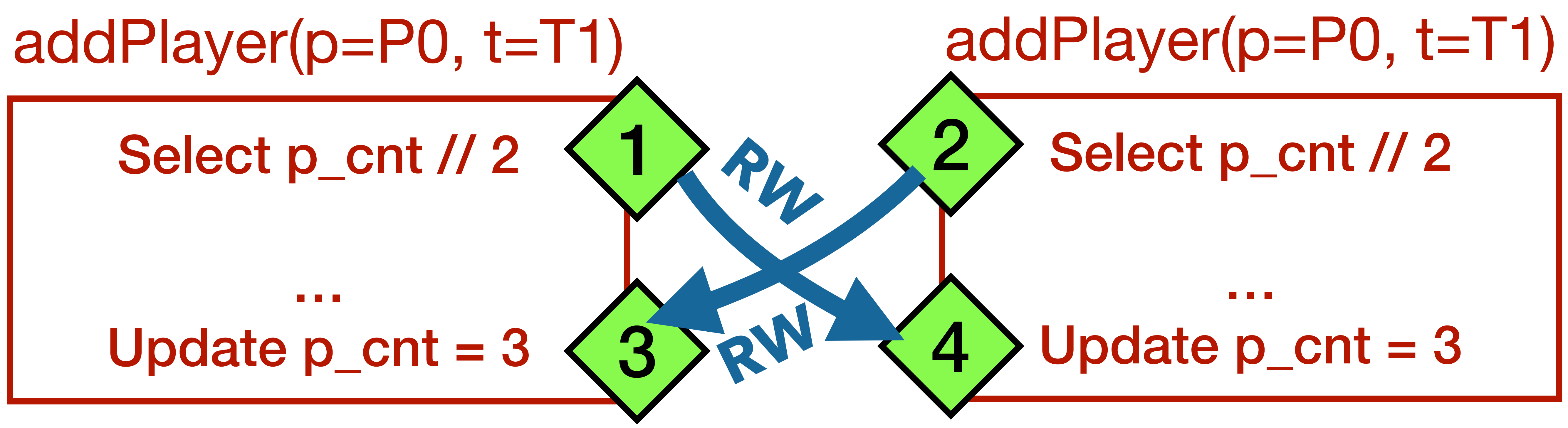


- Non-serializable executions in a program, iff
  - Cyclic dependencies between txns
- Executions abstracted by directed dependency graphs
- Three types of dependency edges
  - write dependency (WW)
  - read dependency (WR)
  - read anti-dependency (RW)



PURDUE UNIVERSITY

- Non-serializable executions in a program, iff
    - Cyclic dependencies between txns
- Executions abstracted by directed dependency graphs
- Three types of dependency edges
    - write dependency (WW)
    - read dependency (WR)
    - read anti-dependency (RW)
- Example: lost update



PURDUE UNIVERSITY

FINDING CYCLES STATICALLY

- Key idea: reduction to SMT

FINDING CYCLES STATICALLY

- Key idea: reduction to SMT
- Use efficient SMT-solvers, e.g. Z3

FINDING CYCLES STATICALLY

- Key idea: reduction to SMT
- Use efficient SMT-solvers, e.g. Z3
- Axiomatic relations encoded within a decidable fragment of FOL

FINDING CYCLES STATICALLY

- Key idea: reduction to SMT
- Use efficient SMT-solvers, e.g. Z3
- Axiomatic relations encoded within a decidable fragment of FOL
- Finding bounded anomalies against a database abstraction reduced to finding satisfying assignments to a formula φ

- Key idea: reduction to SMT
- Use efficient SMT-solvers, e.g. Z3
- Axiomatic relations encoded within a decidable fragment of FOL
- Finding bounded anomalies against a database abstraction reduced to finding satisfying assignments to a formula φ
- Components of the encoding

$$\varphi \equiv \varphi_{\text{CONTEXT}} \wedge \varphi_{\text{DB}} \wedge \varphi_{\text{DEP}\rightarrow} \wedge \varphi_{\rightarrow\text{DEP}} \wedge \varphi_{\text{ANOMALY}}$$

- Key idea: reduction to SMT
- Use efficient SMT-solvers, e.g. Z3
- Axiomatic relations encoded within a decidable fragment of FOL
- Finding bounded anomalies against a database abstraction reduced to finding satisfying assignments to a formula φ
- Components of the encoding

$$\varphi \equiv \varphi_{\text{CONTEXT}} \wedge \varphi_{\text{DB}} \wedge \varphi_{\text{DEP}\rightarrow} \wedge \varphi_{\rightarrow\text{DEP}} \wedge \varphi_{\text{ANOMALY}}$$

conditions satisfied by any execution of any program

FINDING CYCLES STATICALLY

- Key idea: reduction to SMT
- Use efficient SMT-solvers, e.g. Z3
- Axiomatic relations encoded within a decidable fragment of FOL
- Finding bounded anomalies against a database abstraction reduced to finding satisfying assignments to a formula φ
- Components of the encoding

database-specific consistency constraints

$$\varphi \equiv \varphi_{\text{CONTEXT}} \wedge \varphi_{\text{DB}} \wedge \varphi_{\text{DEP}\rightarrow} \wedge \varphi_{\rightarrow\text{DEP}} \wedge \varphi_{\text{ANOMALY}}$$

conditions satisfied by any execution of any program

FINDING CYCLES STATICALLY

- Key idea: reduction to SMT
- Use efficient SMT-solvers, e.g. Z3
- Axiomatic relations encoded within a decidable fragment of FOL
- Finding bounded anomalies against a database abstraction reduced to finding satisfying assignments to a formula φ
- Components of the encoding

$$\varphi \equiv \varphi_{\text{CONTEXT}} \wedge \varphi_{\text{DB}} \wedge \varphi_{\text{DEP}\rightarrow} \wedge \varphi_{\rightarrow\text{DEP}} \wedge \varphi_{\text{ANOMALY}}$$

database-specific consistency constraints

Sufficient conditions to establish a dependency relation

conditions satisfied by **any** execution of **any** program

Necessary conditions to establish a dependency relation

- Key idea: reduction to SMT
- Use efficient SMT-solvers, e.g. Z3
- Axiomatic relations encoded within a decidable fragment of FOL
- Finding bounded anomalies against a database abstraction reduced to finding satisfying assignments to a formula φ
- Components of the encoding

database-specific consistency constraints

Sufficient conditions to establish a dependency relation

$$\varphi \equiv \varphi_{\mathrm{CONTEXT}} \wedge \varphi_{\mathrm{DB}} \wedge \varphi_{\mathrm{DEP}\rightarrow} \wedge \varphi_{\rightarrow\mathrm{DEP}} \wedge \varphi_{\mathrm{ANOMALY}}$$

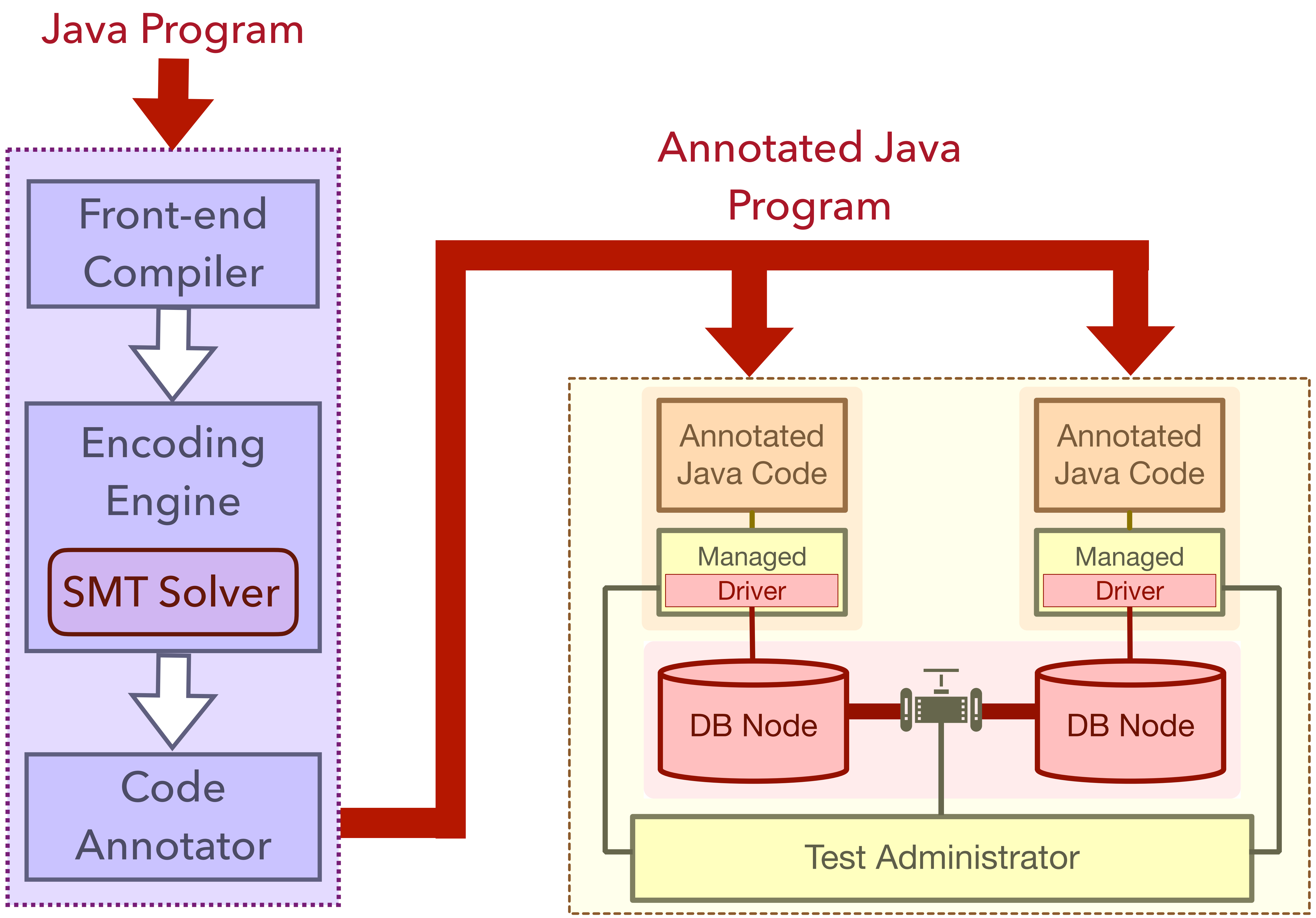
conditions satisfied by **any execution of any program**

Necessary conditions to establish a dependency relation

Enforces the existence of a dependency cycle

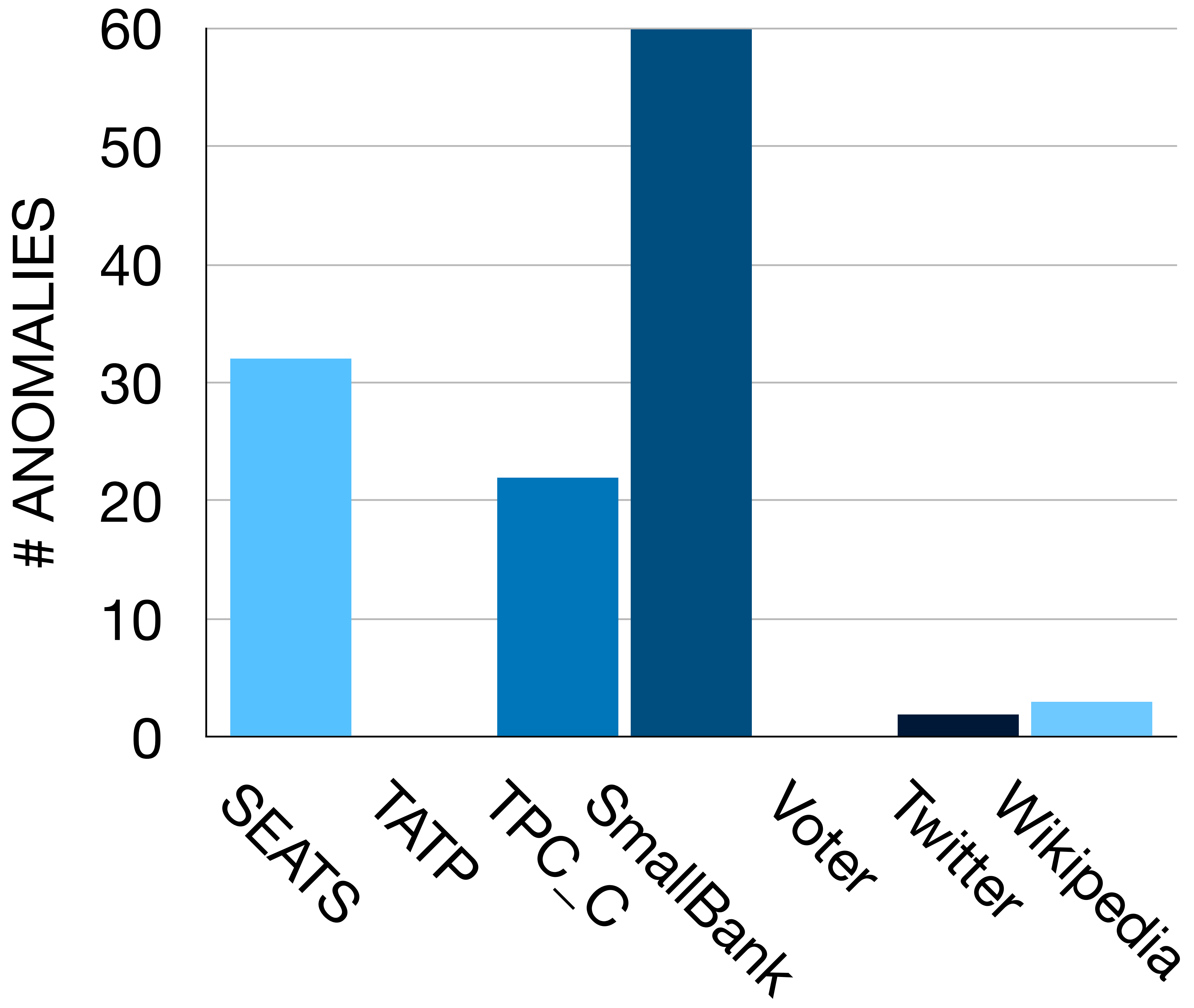
CLOTHO: DIRECTED TEST GENERATION AND REPLAY

- Static analysis engine for java programs
- Automatic replay of anomalous executions

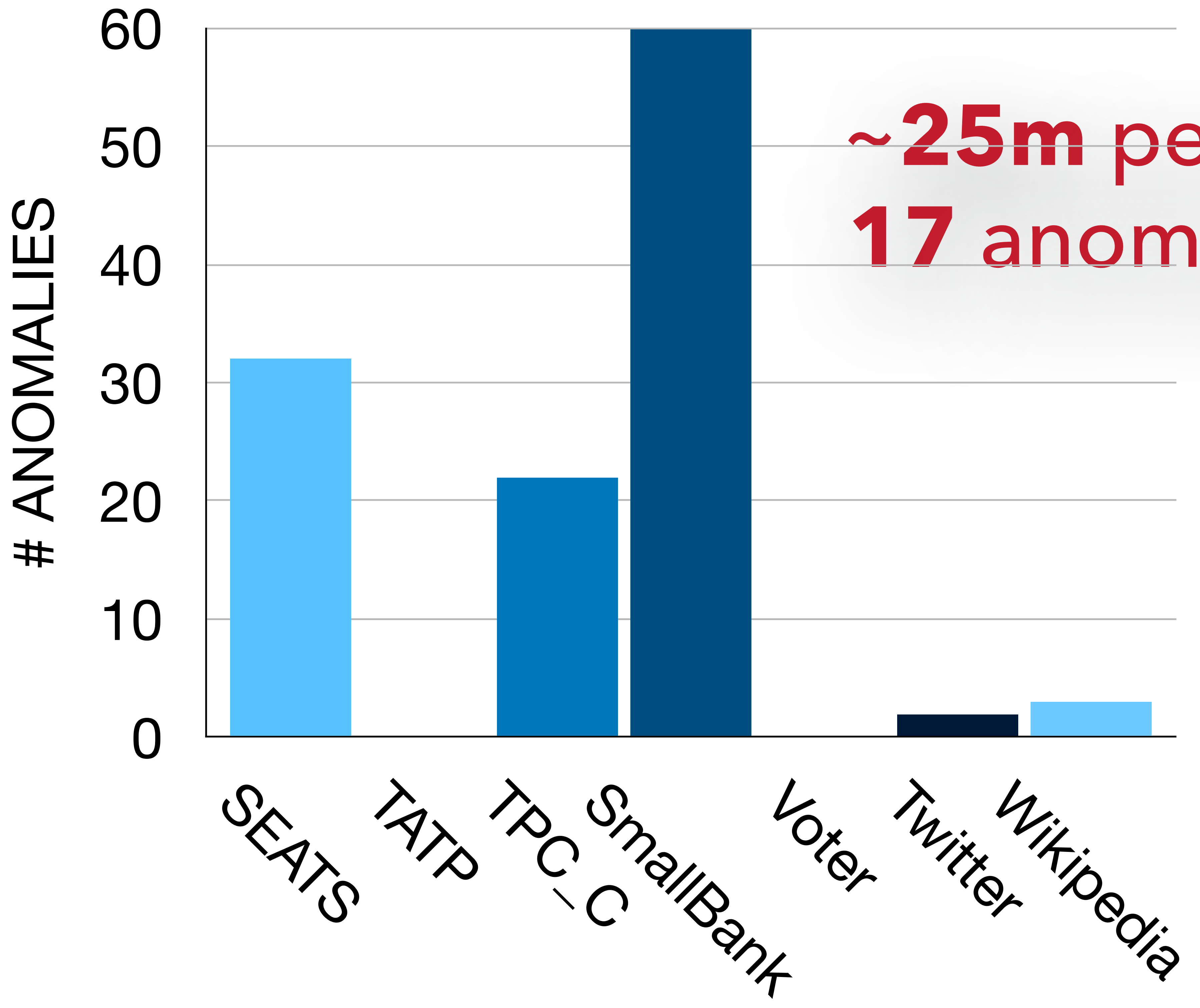


EXPERIMENTAL RESULTS

- **7 benchmarks** of various complexity analyzed
- Serializability anomalies found and replayed in 5 benchmark



- **7 benchmarks** of various complexity analyzed
- Serializability anomalies found and replayed in 5 benchmark



~**25m** per application (avg)

17 anomalies per application (avg)

CASE STUDY: TPC-C

- 22 anomalies mapped to invariant violations
  - All invariants were broken
  - Only 3 anomalies did NOT violate any invariant

Invariant	Blackbox	CLOTHO
CR1	Y	Y
CR2	Y	Y
CR3	Y	Y
CR4	Y	Y
CR5A	N	Y
CR5B	N	Y
CR6	Y	Y
CR7A	N	Y
CR7B	N	Y
CR8	Y	Y
CR9	Y	Y
CR10	Y	Y
CR11	Y	Y
CR12	Y	Y
NCR1	Y	Y
NCR2	Y	Y
NCR3	N	Y
NCR4	N	Y
NCR5	Y	Y
NCR6	Y	Y
NCR7	N	Y

ATROPOS

(REPAIRING REPLICATION ANOMALIES)

- Serializability anomaly

addPlayer(*p*, *t*)

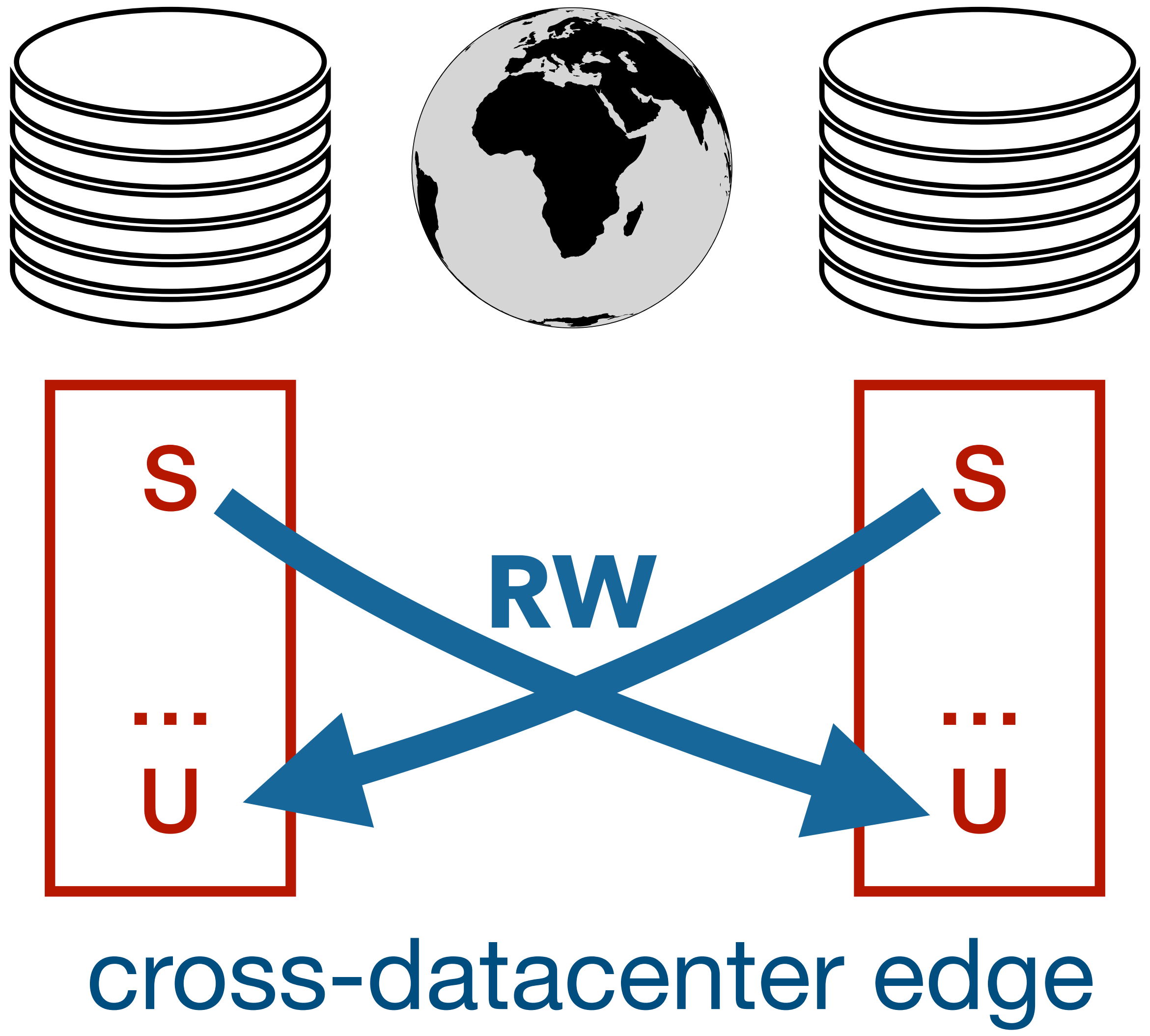
```
old_cnt := select p_cnt from TEAM where t_id=t
Insert (p,t,∅,∅) into PLAYER
update TEAM set p_cnt=old_cnt+1 where t_id=t
```

REPLICATION ANOMALIES

- Serializability anomaly

TEAM

t_id	t_name	p_cnt
T1	“A”	2



addPlayer(*p*, *t*)

```
old_cnt := select p_cnt from TEAM where t_id=t
Insert (p,t,∅,∅) into PLAYER
update TEAM set p_cnt=old_cnt+1 where t_id=t
```

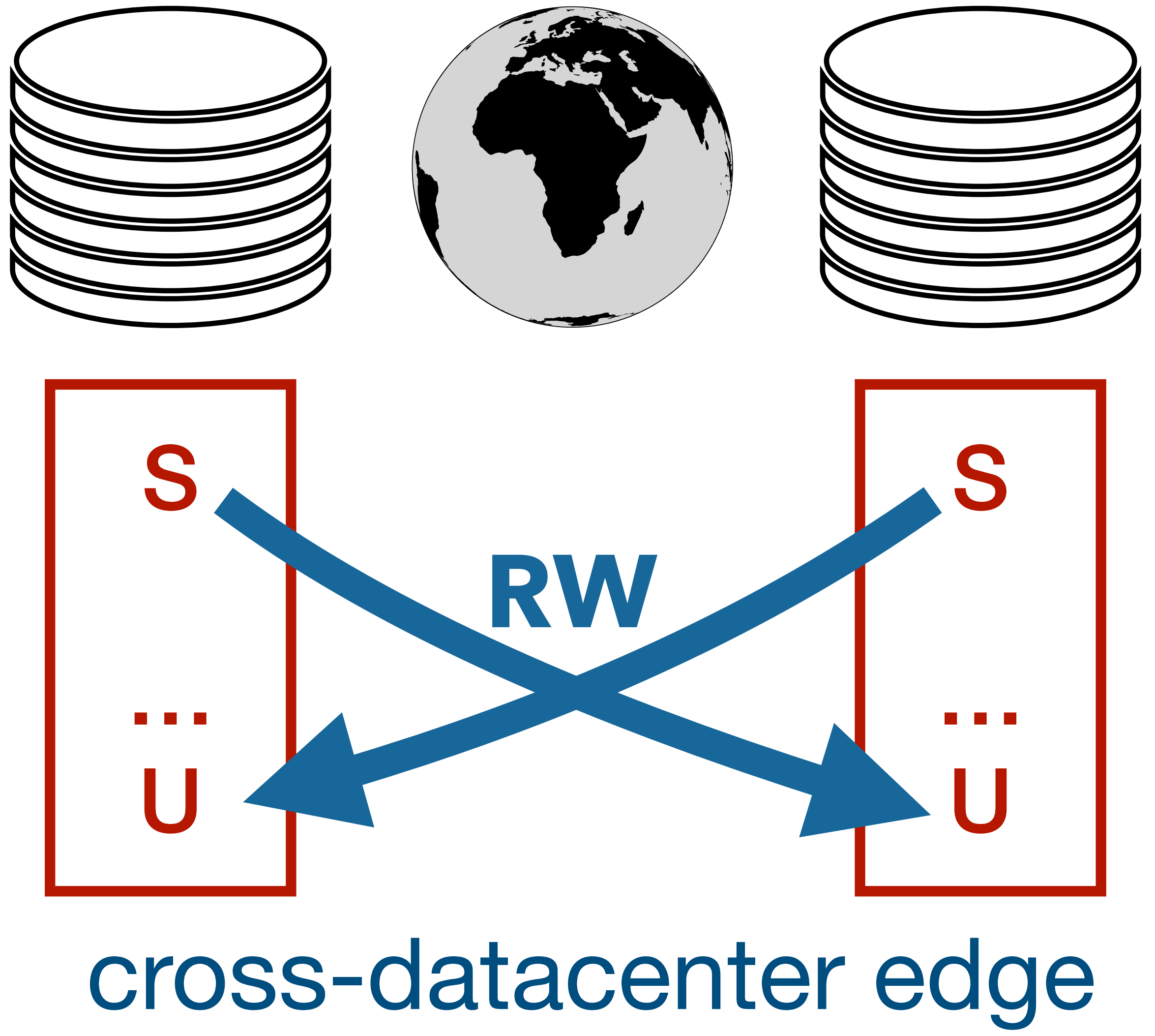
- Serializability anomaly
- Replication anomalies can be eliminated using consistency annotations

addPlayer(*p*, *t*)

```
Consistent {
   old_cnt := select * from TEAM where t_id=t
   Insert (p,t,∅,∅) into PLAYER
   update TEAM set p_cnt=old_cnt+1 where t_id=t
}
```

TEAM

t_id	t_name	p_cnt
T1	“A”	2



- Serializability anomaly
- Replication anomalies can be eliminated using consistency annotations
- **Equivalent program** without dependency cycles

addPlayer(*p*, *t*)

```
Consistent {
   old_cnt := select * from TEAM where t_id=t
   Insert (p,t,∅,∅) into PLAYER
   update TEAM set p_cnt=old_cnt+1 where t_id=t
}
```

≡

addPlayer(*p*, *t*)

?

PLAYER

p_id	p_t_id	p_role	p_stat

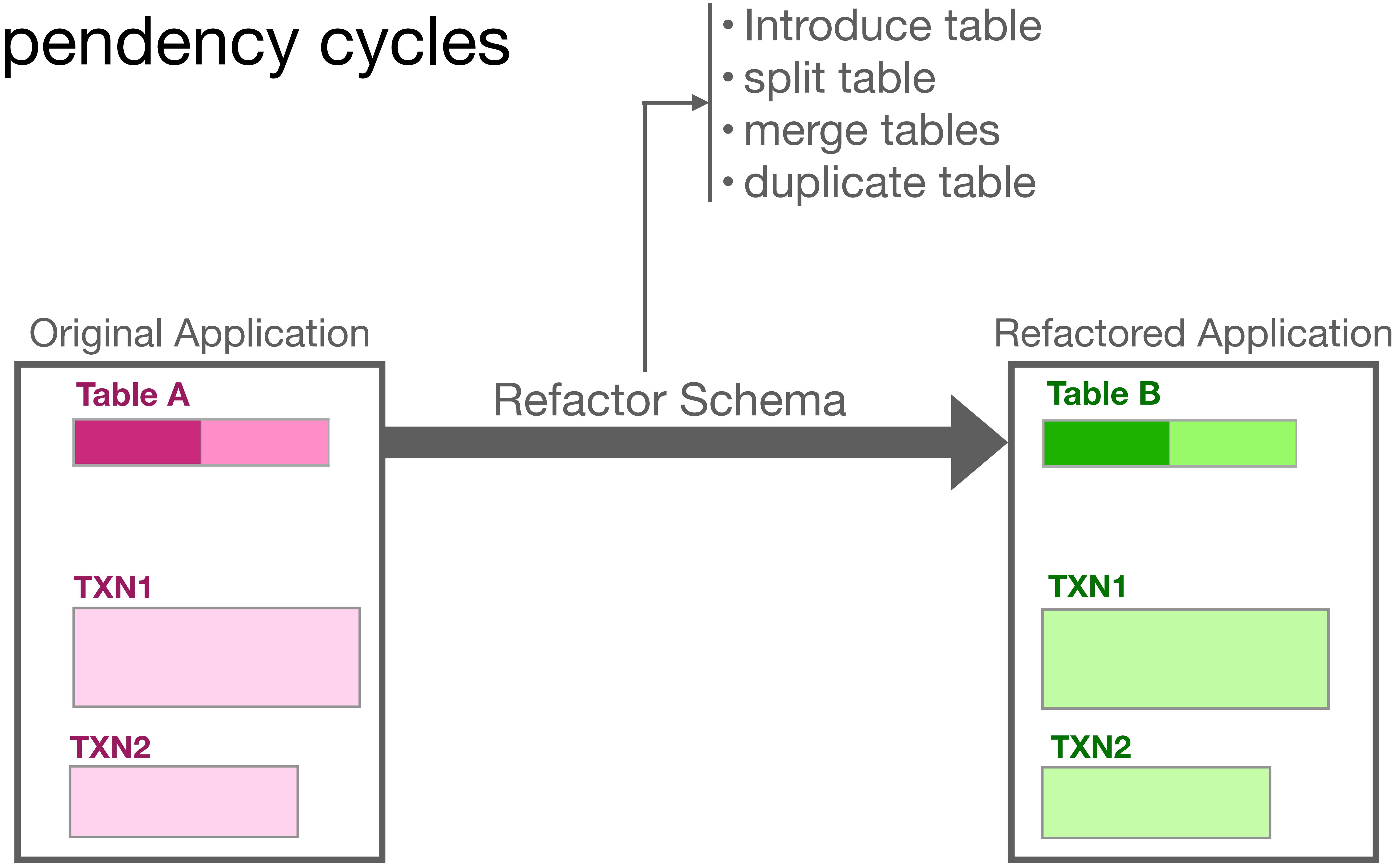
TEAM

t_id	t_name	p_cnt

?

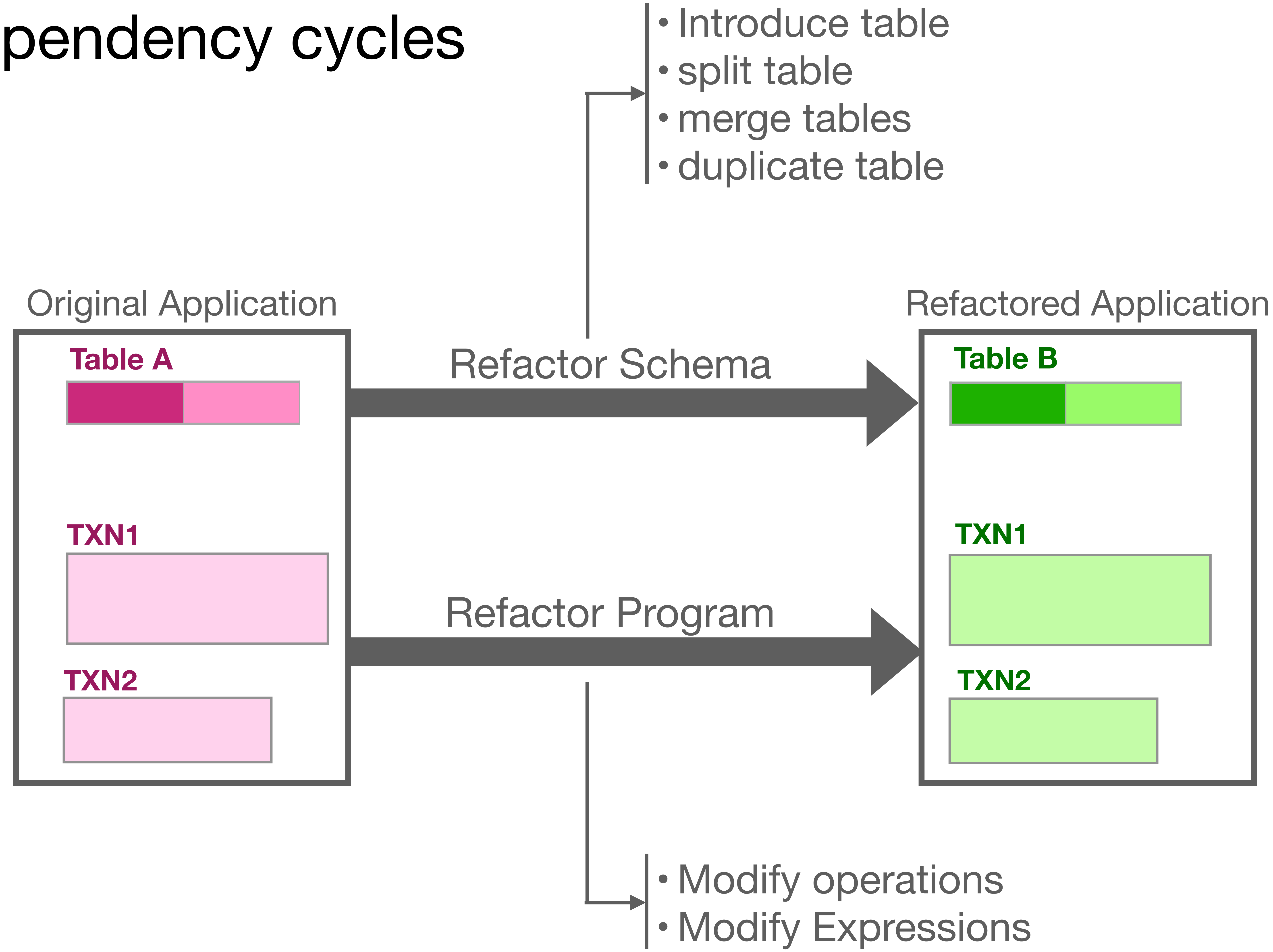
?	?	?	?

- Equivalent program without dependency cycles
  - Schema Refactoring



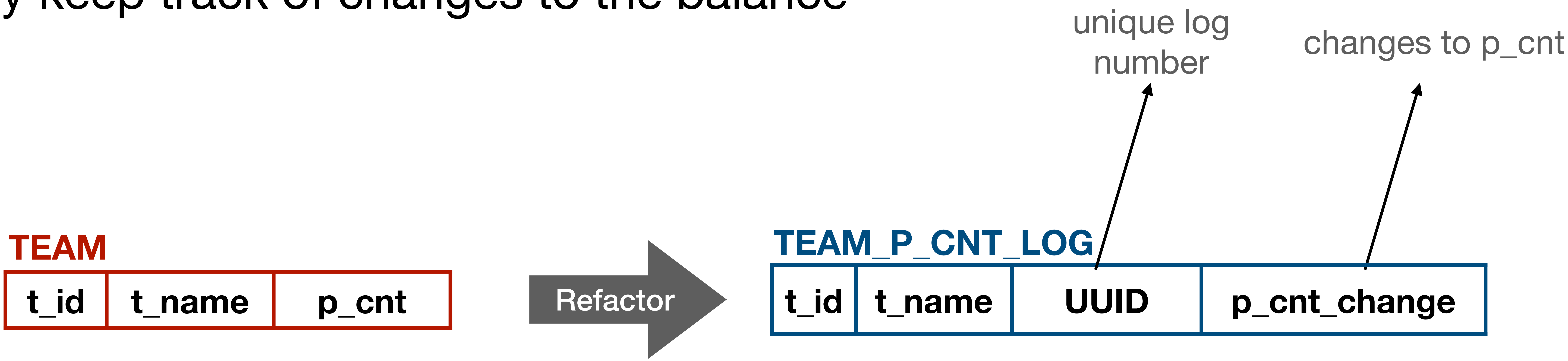
REFACTORING FOR ELIMINATING ANOMALIES

- Equivalent program without dependency cycles
  - Schema Refactoring
  - Program Refactoring



REFACTORING FOR ELIMINATING ANOMALIES

- Equivalent program without dependency cycles
  - Schema Refactoring
  - Program Refactoring
- Only keep track of changes to the balance



- Equivalent program without dependency cycles
  - Schema Refactoring
  - Program Refactoring
- Only keep track of changes to the balance
- No shared item ⇨ No dependency

addPlayer(*p*, *t*)

```
old_cnt := select p_cnt from TEAM where t_id=t
Insert (p,t,∅,∅) into PLAYER
Insert (t,∅,uuid(),+1) into TEAM_P_CNT_LOG
```

New record is inserted for each new player

TEAM_P_CNT_LOG

t_id	t_name	UUID	p_cnt_change

- Equivalent program without dependency cycles
  - Schema Refactoring
  - Program Refactoring
- Only keep track of changes to the balance
- No shared item ⇨ No dependency

TEAM

t_id	t_name	p_cnt

```
cnt := select p_cnt
from TEAM where t_id=t
```

Refactor →

TEAM_P_CNT_LOG

t_id	t_name	UUID	p_cnt_change

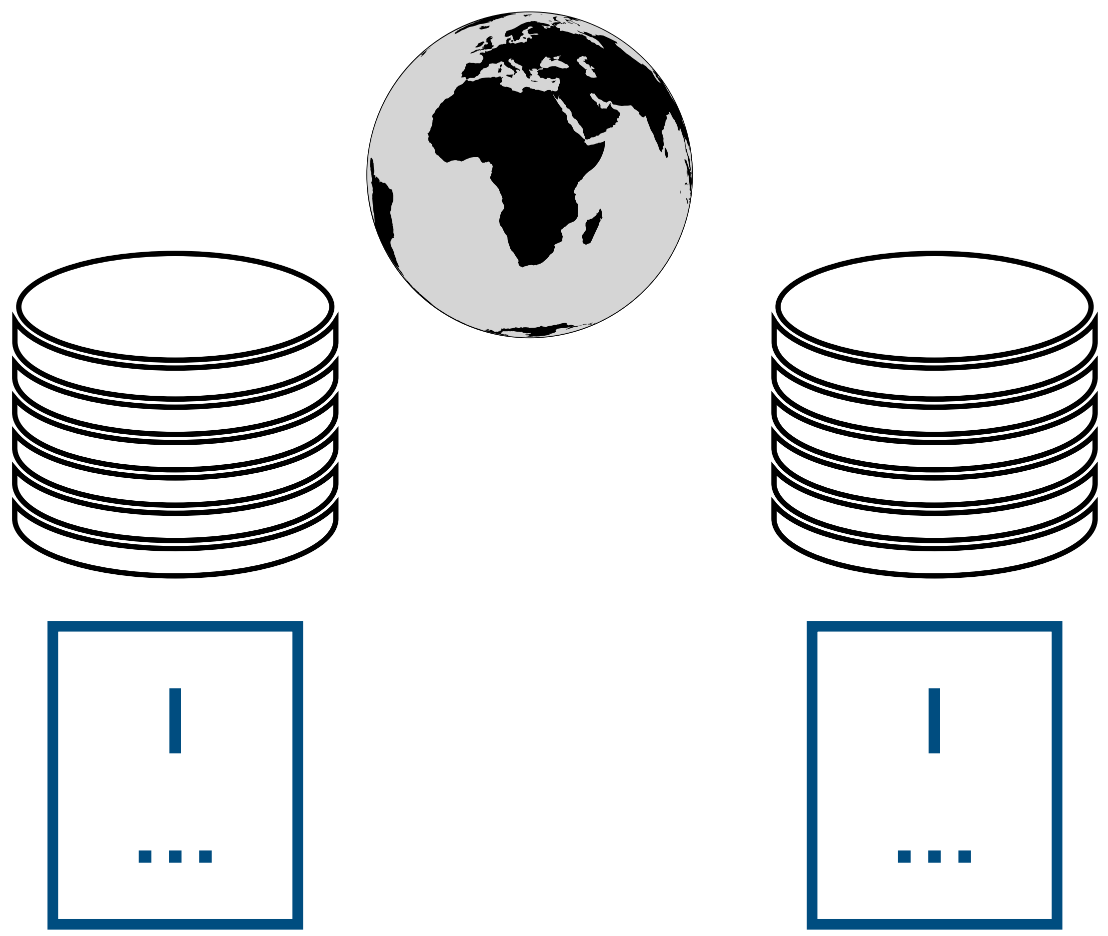
```
cnt := select SUM(p_cnt_change)
from TEAM_P_CNT_LOG where t_id=t
```

updated expressions to return **equivalent logical values** as the original program

REPAIRED PROGRAM’S EXECUTION

TEAM_P_CNT_LOG

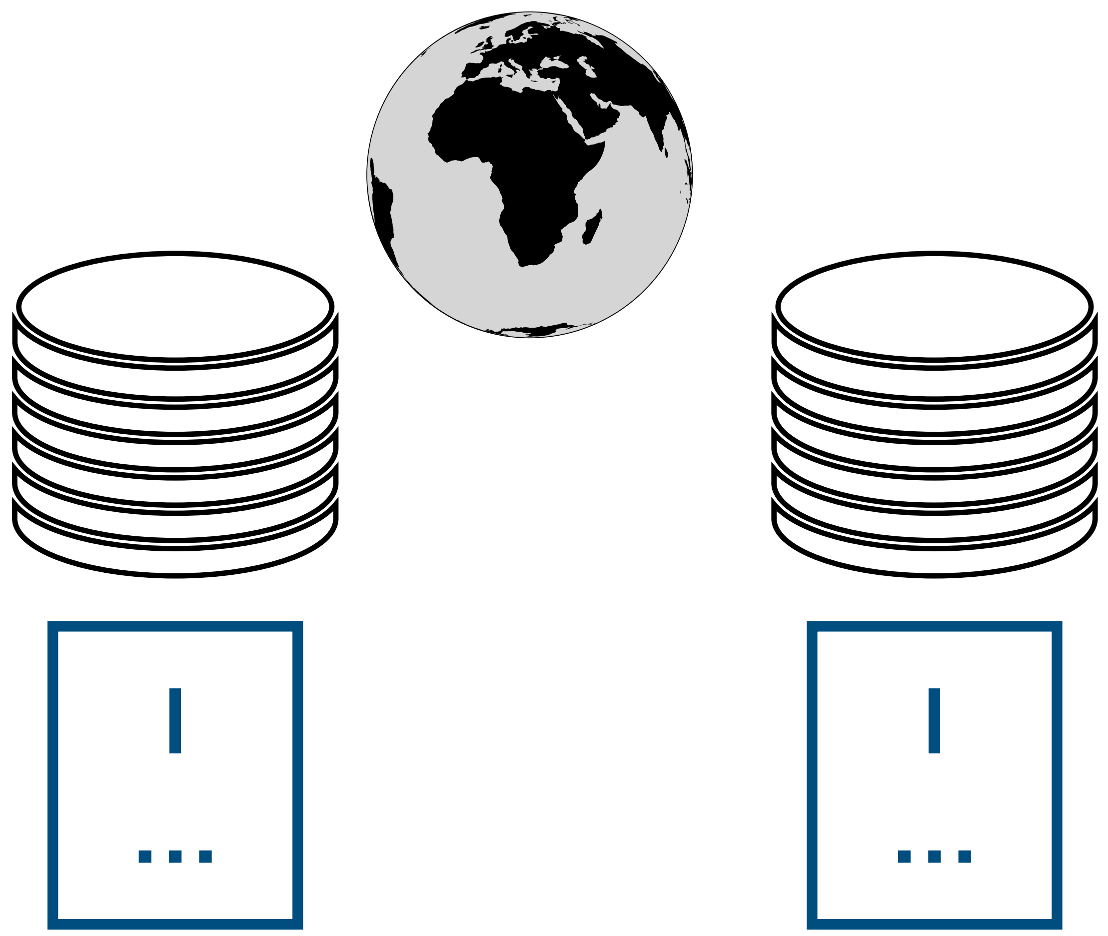
t_id	t_name	UUID	p_cnt_change
1	“A”	U3	2



REPAIRED PROGRAM'S EXECUTION

TEAM_P_CNT_LOG

t_id	t_name	UUID	p_cnt_change
1	"A"	U3	2



TEAM_P_CNT_LOG

t_id	t_name	UUID	p_cnt_change
1	"A"	U1	1
1	"A"	U3	2
1	"A"	U4	1

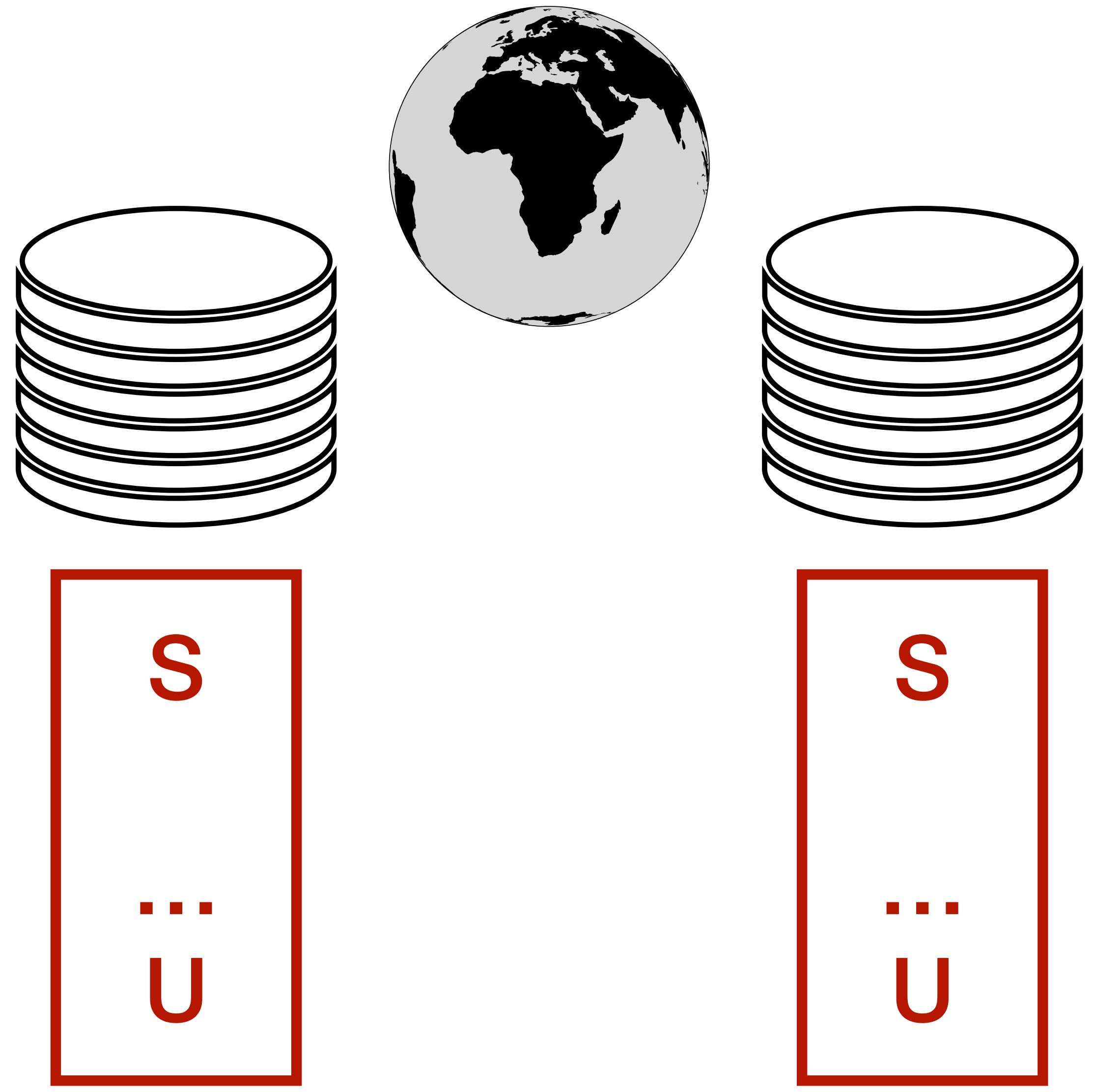
REPAIRED PROGRAM'S EXECUTION

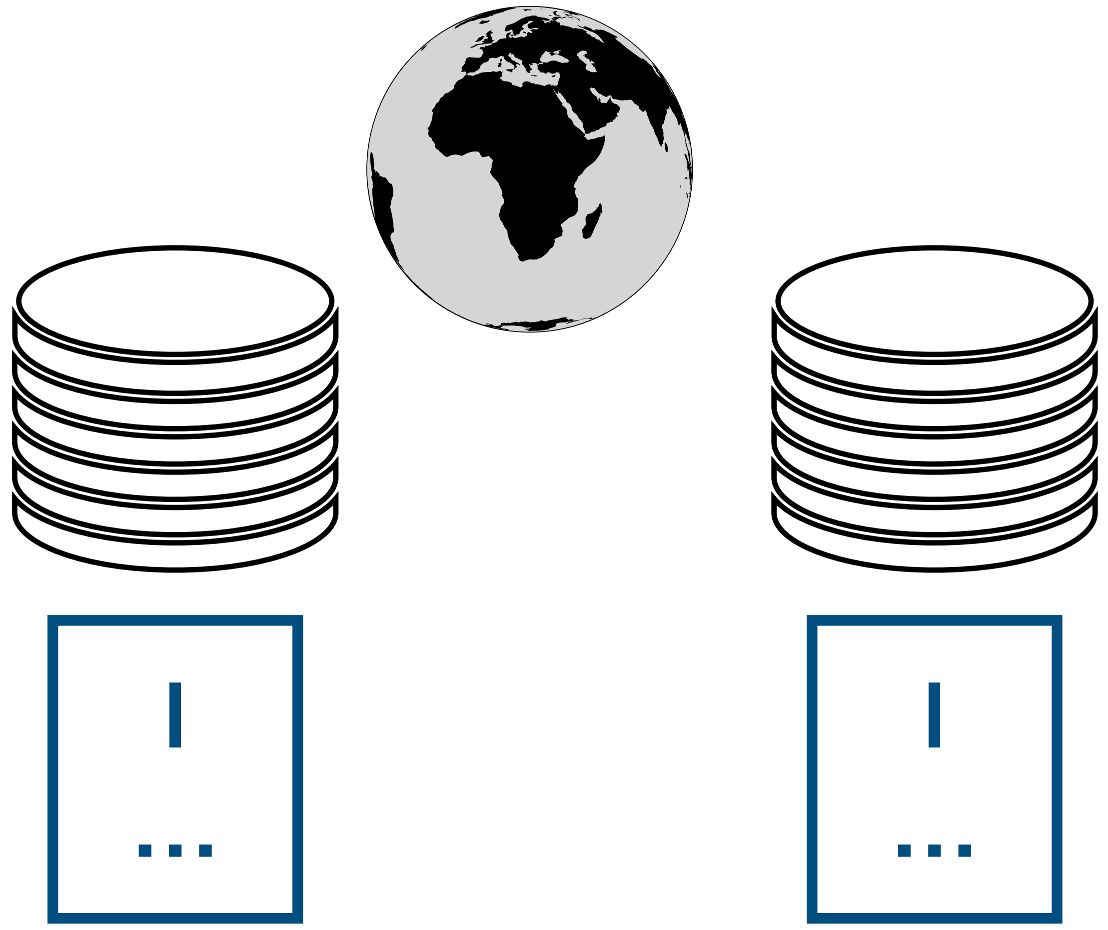
TEAM

t_id	t_name	p_cnt
T1	“A”	2

TEAM_P_CNT_LOG

t_id	t_name	UUID	p_cnt_change
1	“A”	U3	2





TEAM

t_id	t_name	p_cnt
T1	“A”	4

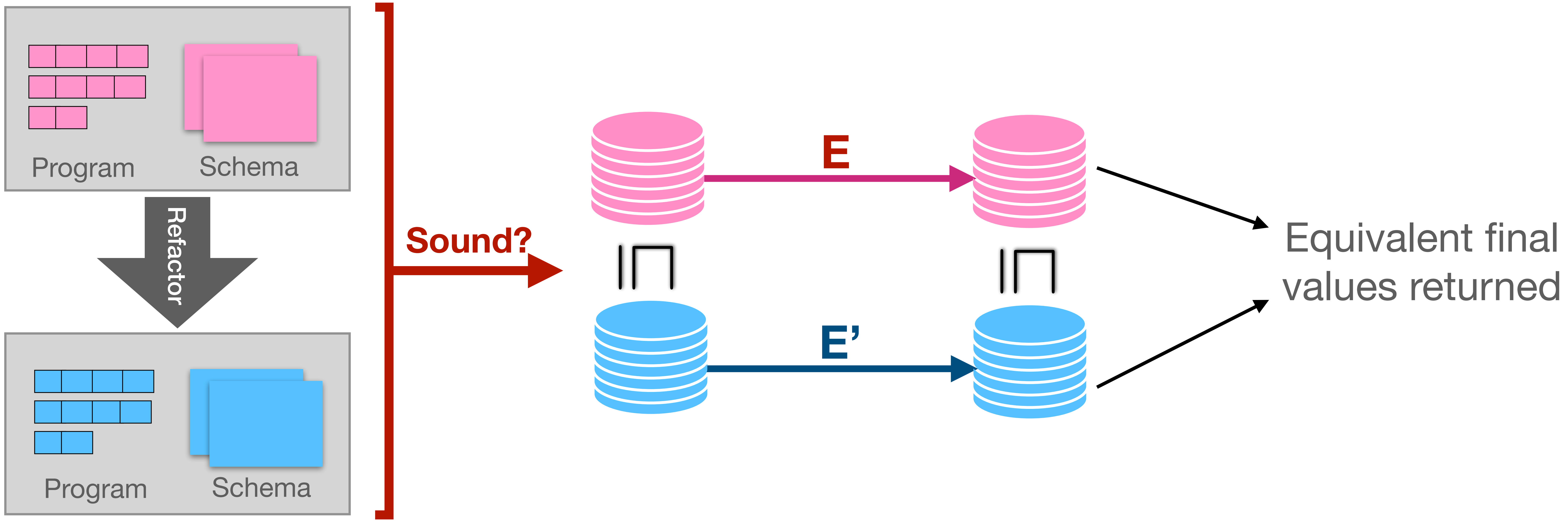
$\sqsubseteq$

TEAM_P_CNT_LOG

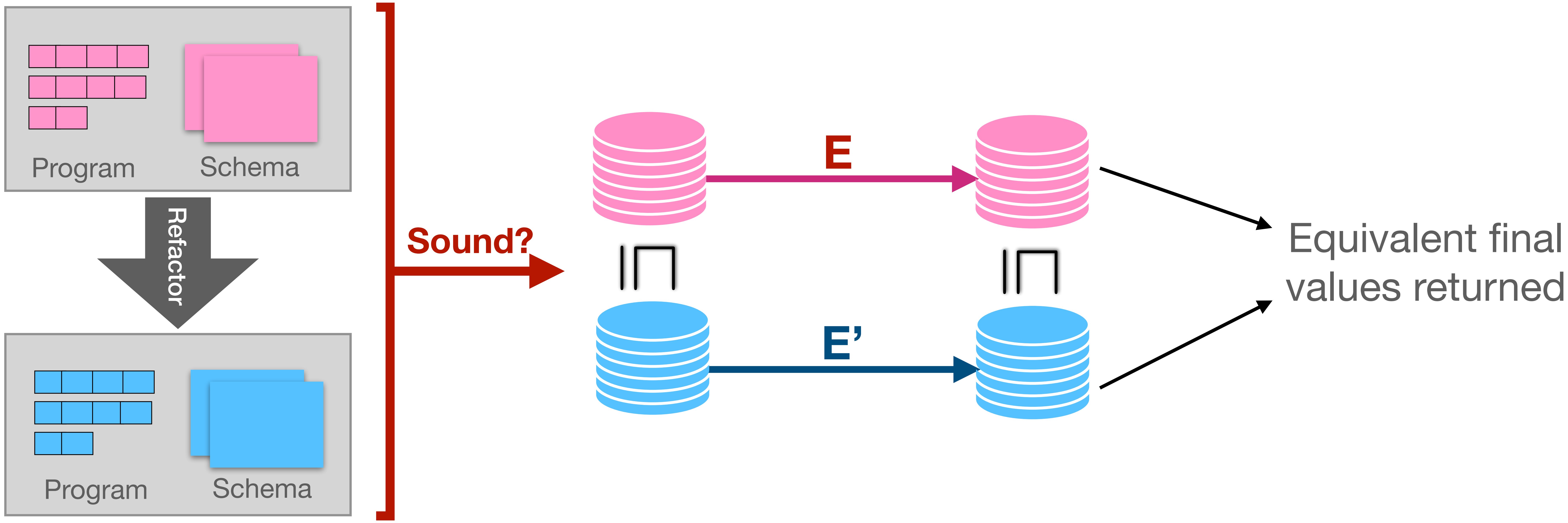
t_id	t_name	UUID	p_cnt_change
1	“A”	U1	1
1	“A”	U3	2
1	“A”	U4	1

- Program **P’** is a sound refactoring of **P** iff:

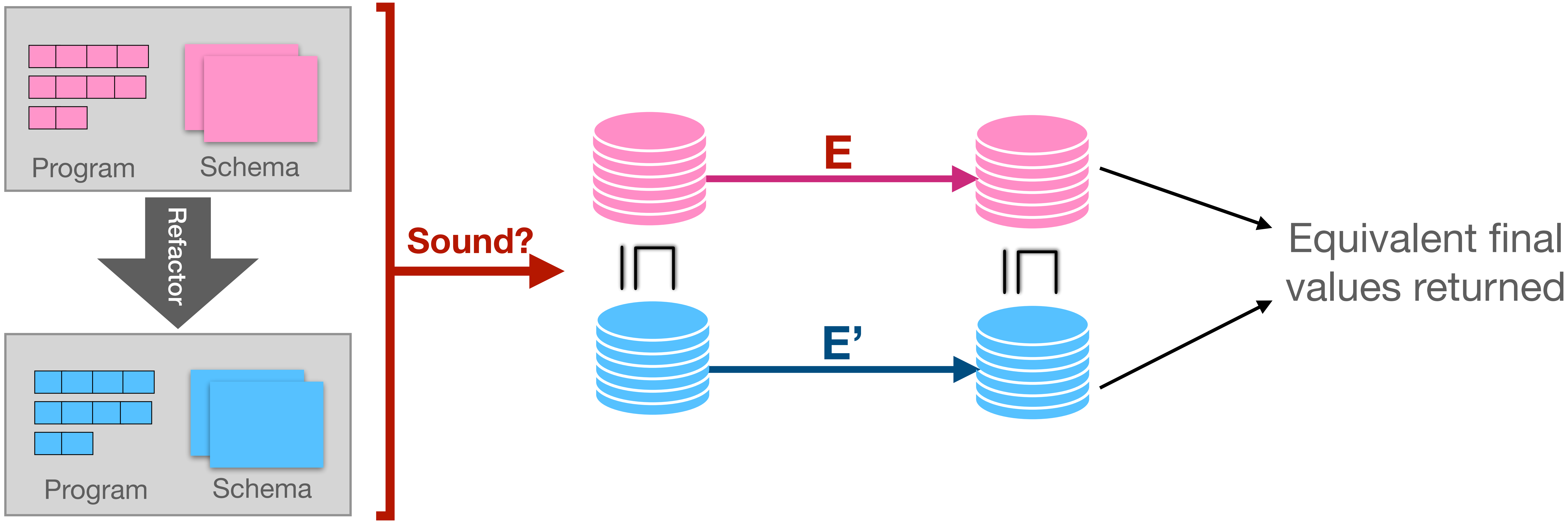
- Program **P'** is a sound refactoring of **P** iff:
  - for any execution **E'** of **P'** there is an execution E of P s.t.



- Program **P’** is a sound refactoring of **P** iff:
    - for any execution **E’** of **P’** there is an execution E of P s.t.
    - **E** and **E’** preserve the **containment relation** ($\sqsubseteq$) between initial and final DB states

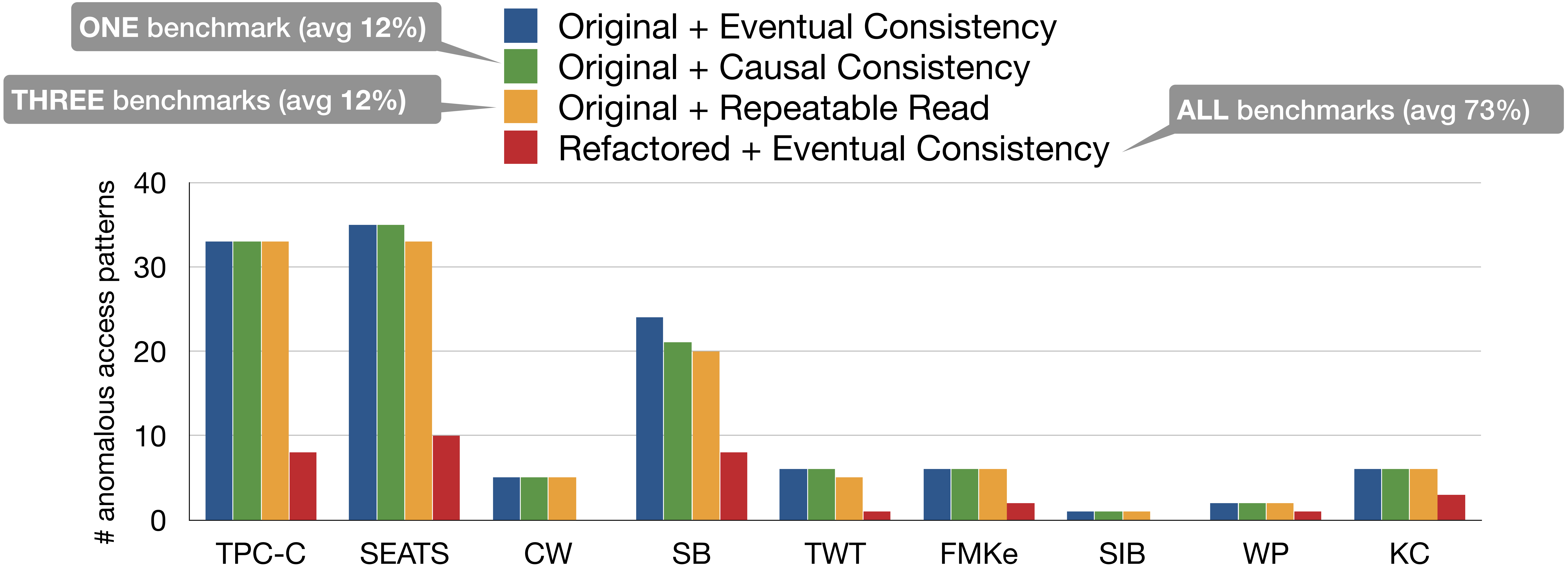


- Program **P'** is a sound refactoring of **P** iff:
  - for any execution **E'** of **P'** there is an execution E of P s.t.
  - **E** and **E'** preserve the **containment relation** ($\sqsubseteq$) between initial and final DB states
  - equivalent values returned



EVALUATION (APPLICABILITY + EFFECTIVENESS)

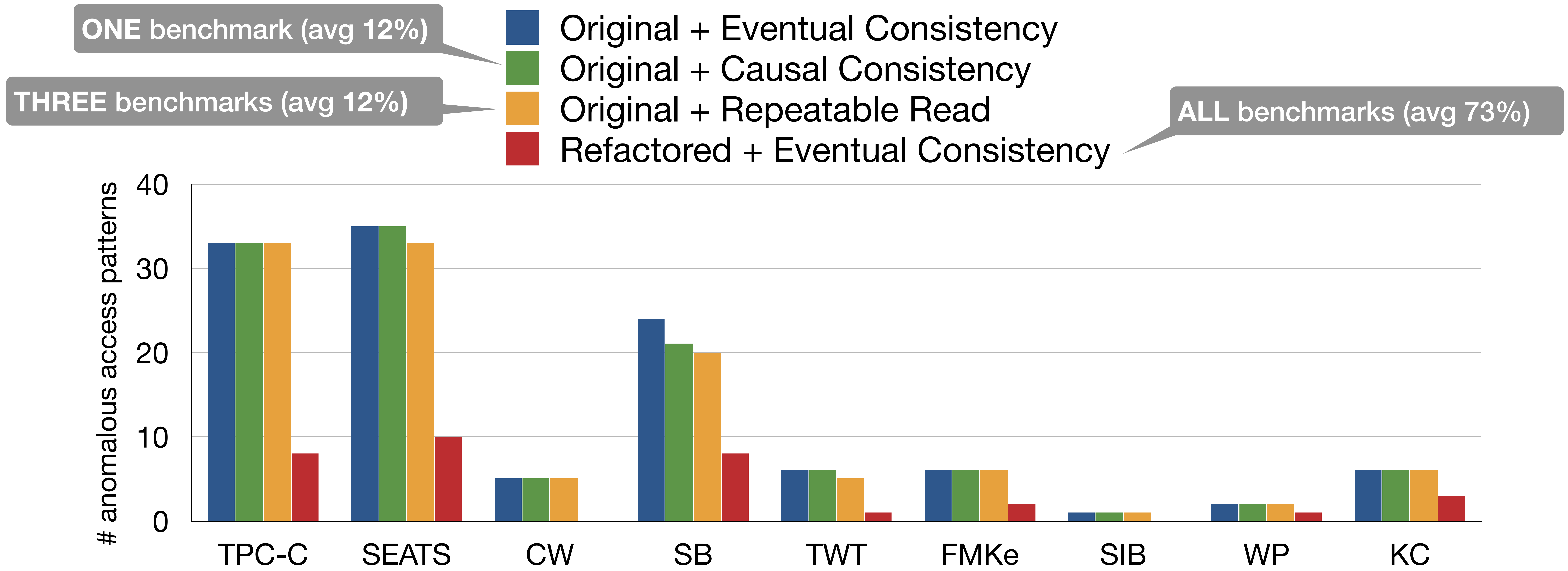
- 9 benchmarks
  - Number of statically identified anomalous access pairs



PURDUE UNIVERSITY

EVALUATION (APPLICABILITY + EFFECTIVENESS)

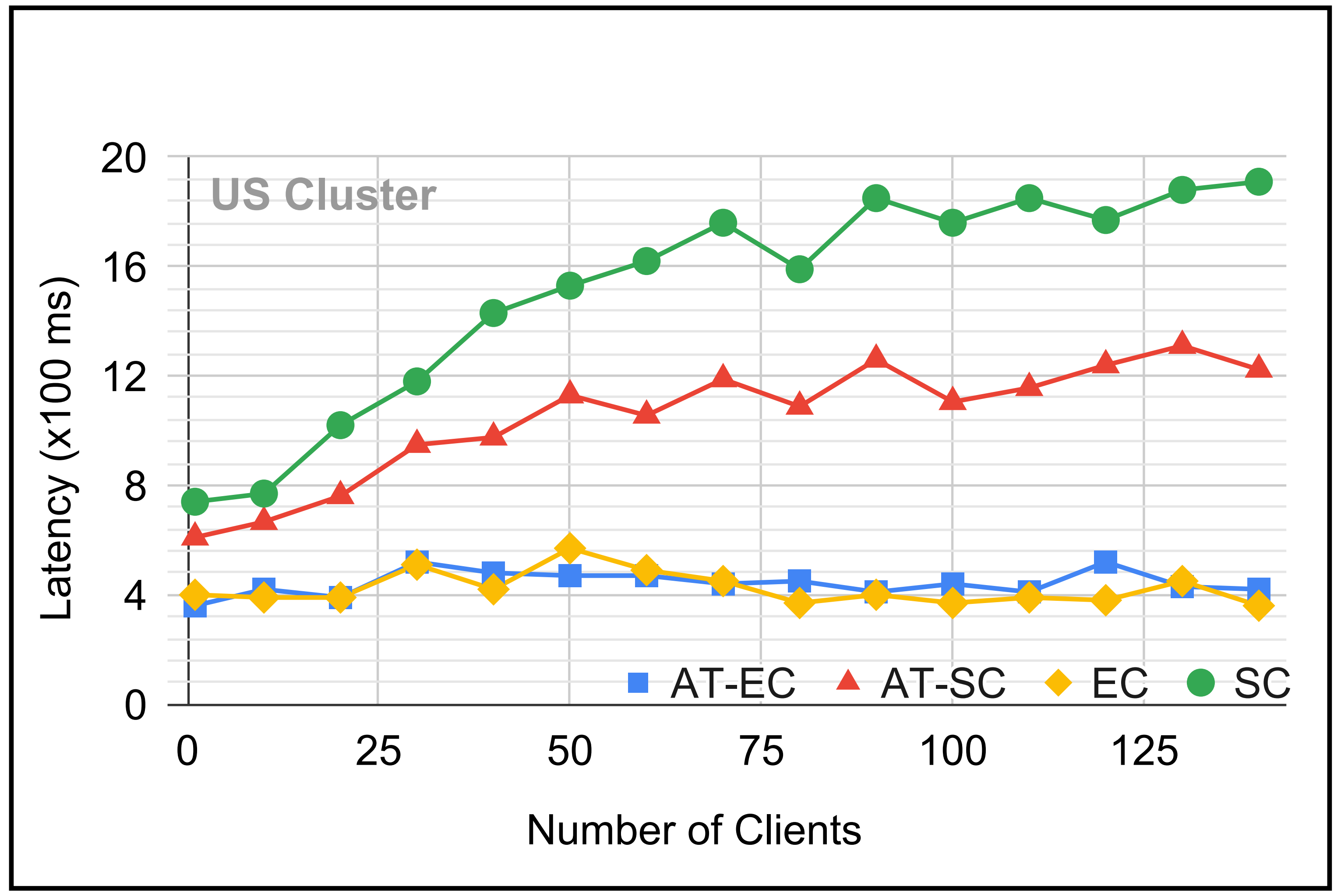
- 9 benchmarks
  - Number of statically identified anomalous access pairs



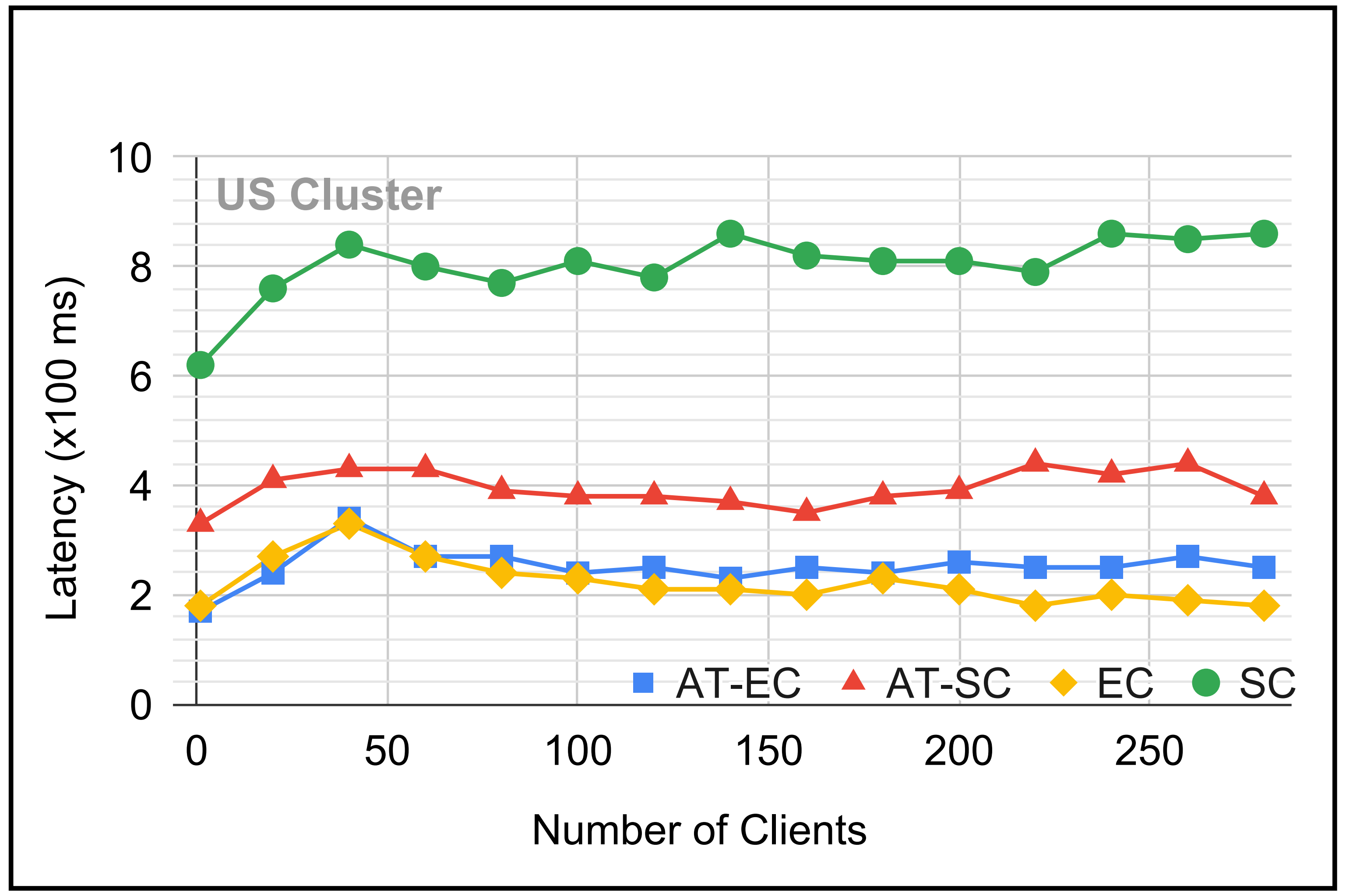
PURDUE UNIVERSITY

EVALUATION (PERFORMANCE)

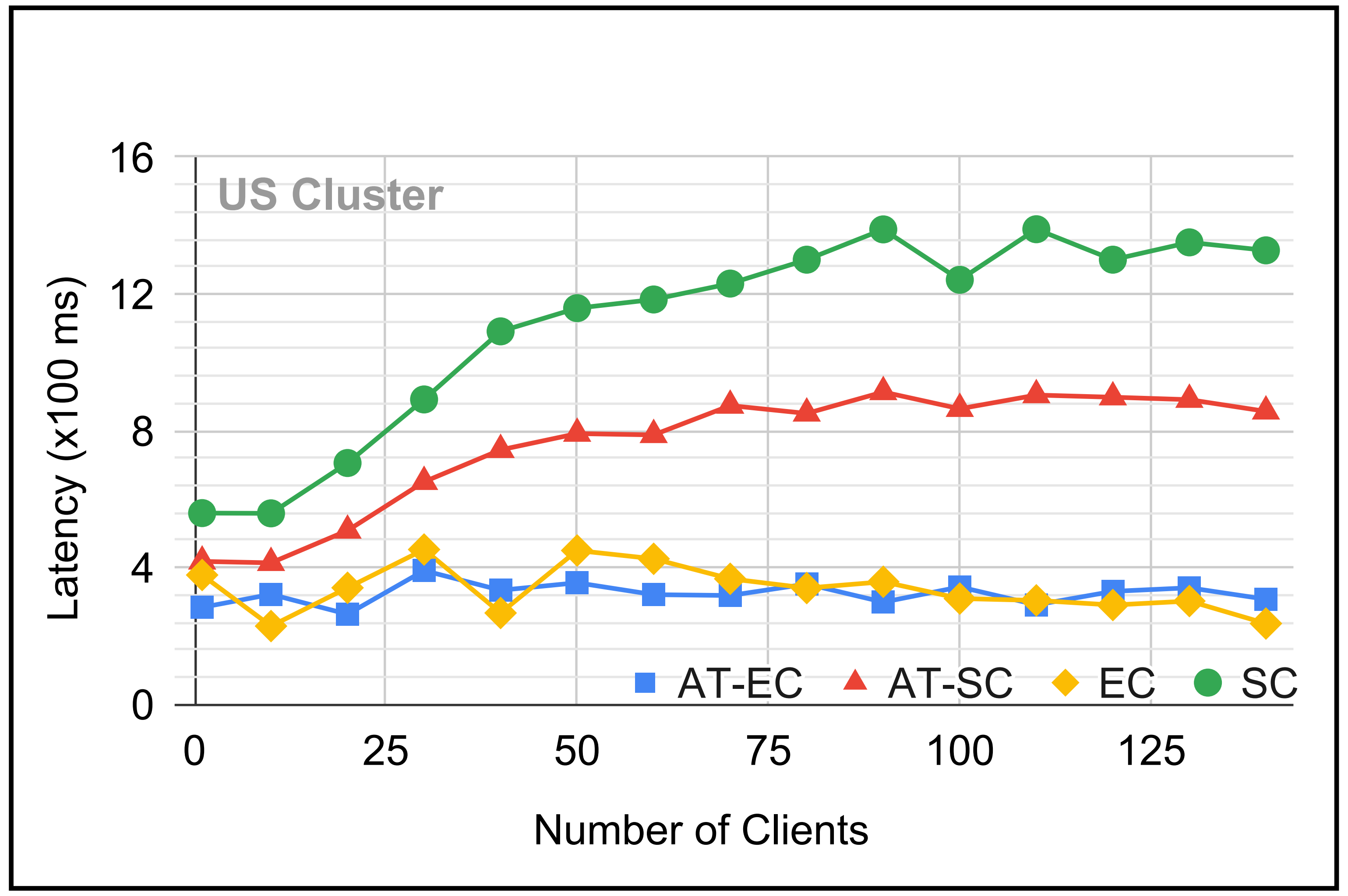
- 3 benchmarks
- Latency and throughput
  - EC, SC: original program (eventual consistency/serializability)
  - AT_EC, AT_SC: refactored program (eventual consistency/serializability)



TPC-C

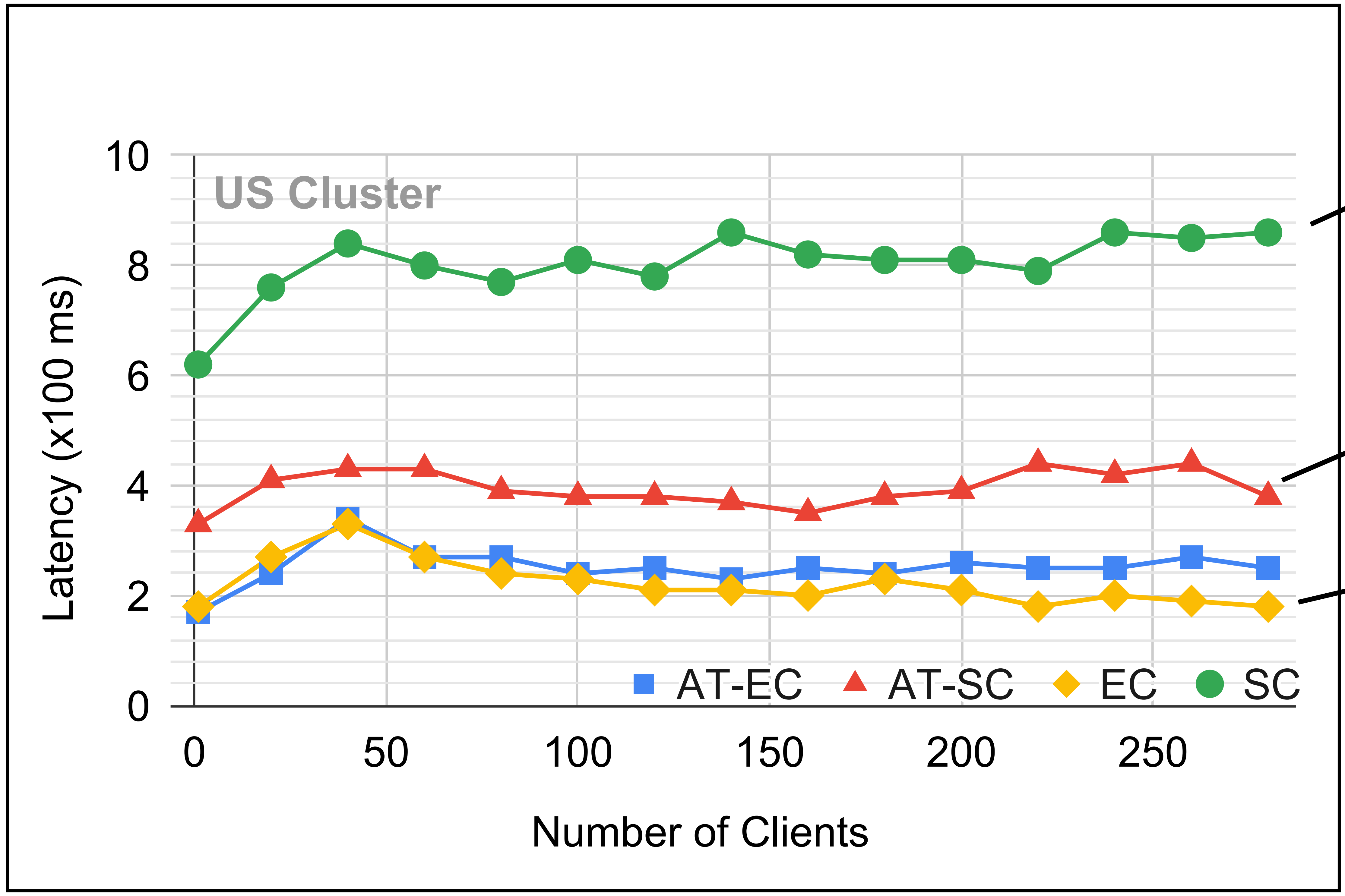


SmallBank



SEATS

EVALUATION (PERFORMANCE)



SmallBank

Serializability is slow!

Refactoring enables performance (45% lower latency)

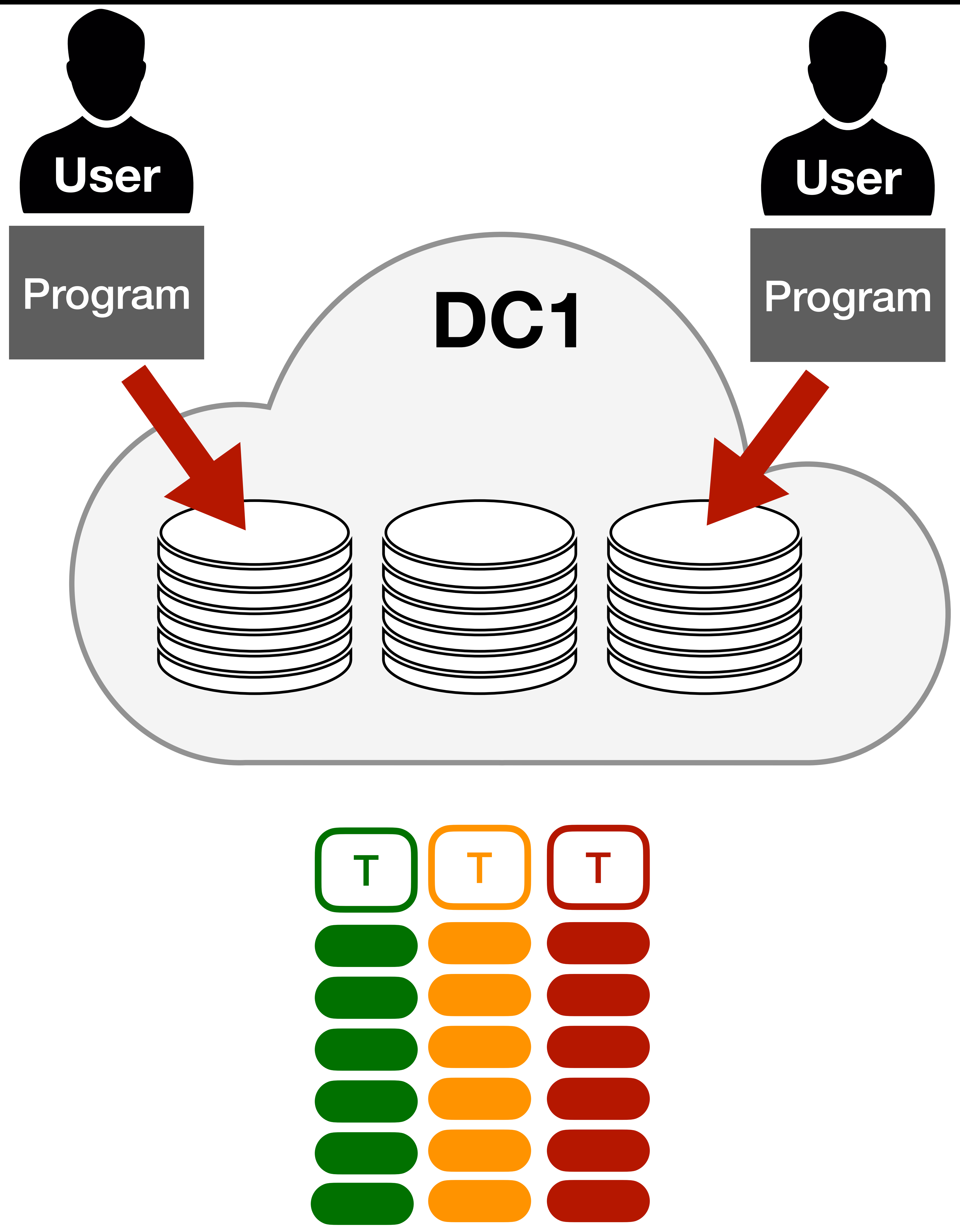
Refactoring has limited overhead — vs original

- EC, SC: original program + eventual consistency and serializability
- AT_EC, AT_SC: original program + eventual consistency and serializability

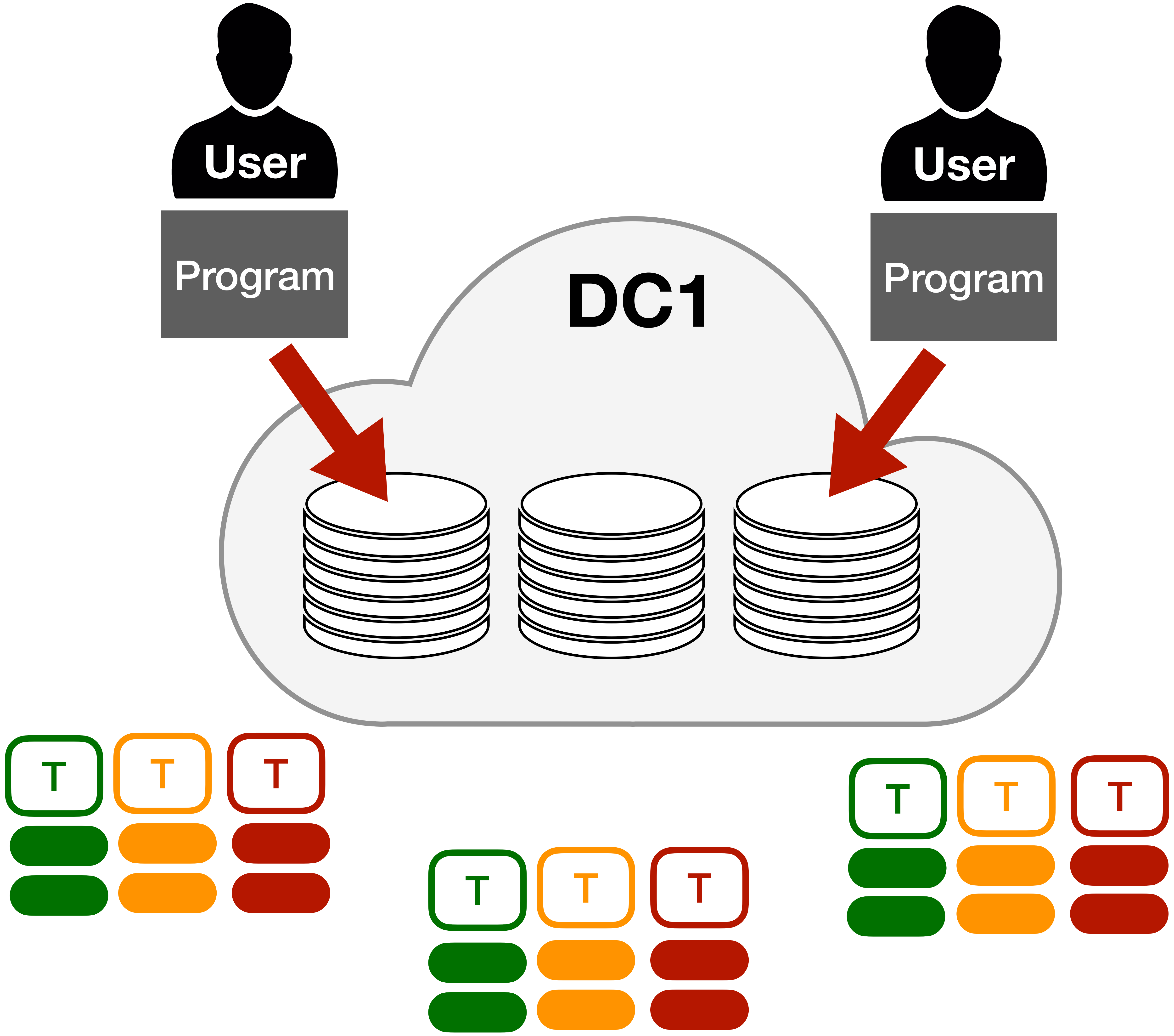
LACHESIS

(REPAIRING PARTITIONING ANOMALIES)

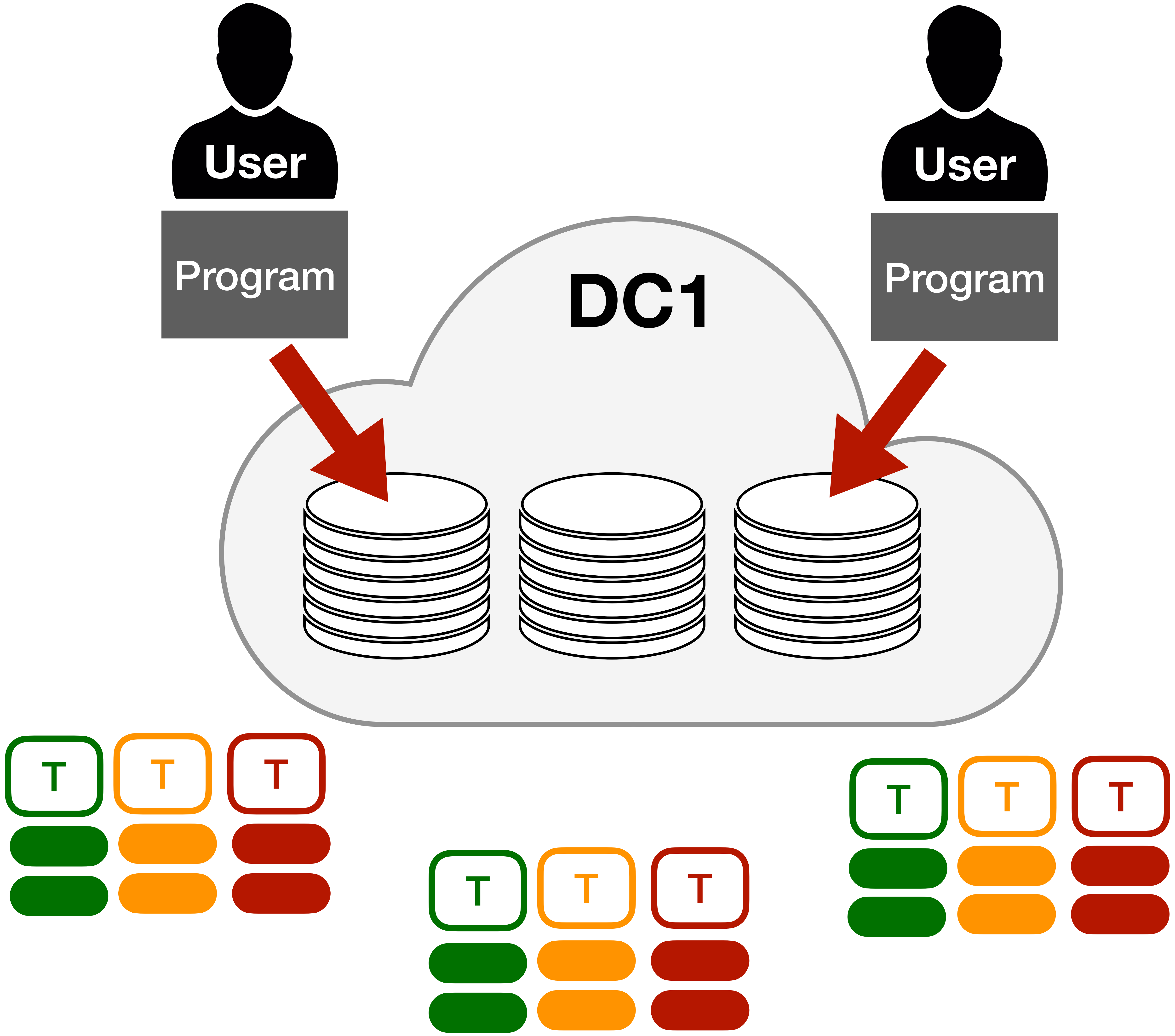
- Partitioned Database Clusters
  - Same data-center deployment



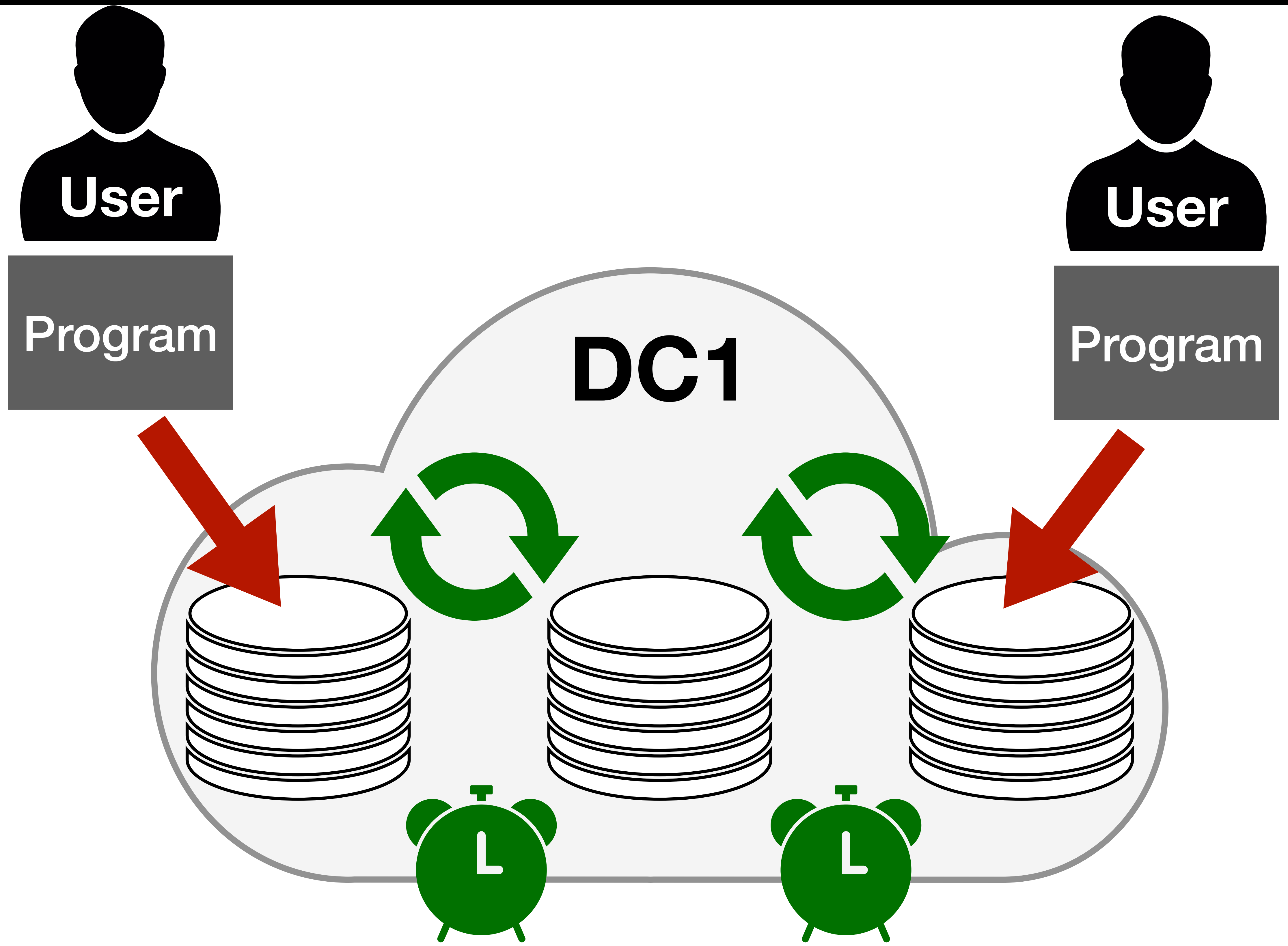
- Partitioned Database Clusters
  - Same data-center deployment
  - Subset of records at each node



- Partitioned Database Clusters
    - Same data-center deployment
    - Subset of records at each node
- Partitioning Policy $\prod$
    - Performance



- Partitioned Database Clusters
  - Same data-center deployment
  - Subset of records at each node
- Partitioning Policy $\prod$
  - Performance



- Partitioned Database Clusters
  - Same data-center deployment
  - Subset of records at each node
- Partitioning Policy $\prod$
  - Performance
- Schism: [Curino et.al]

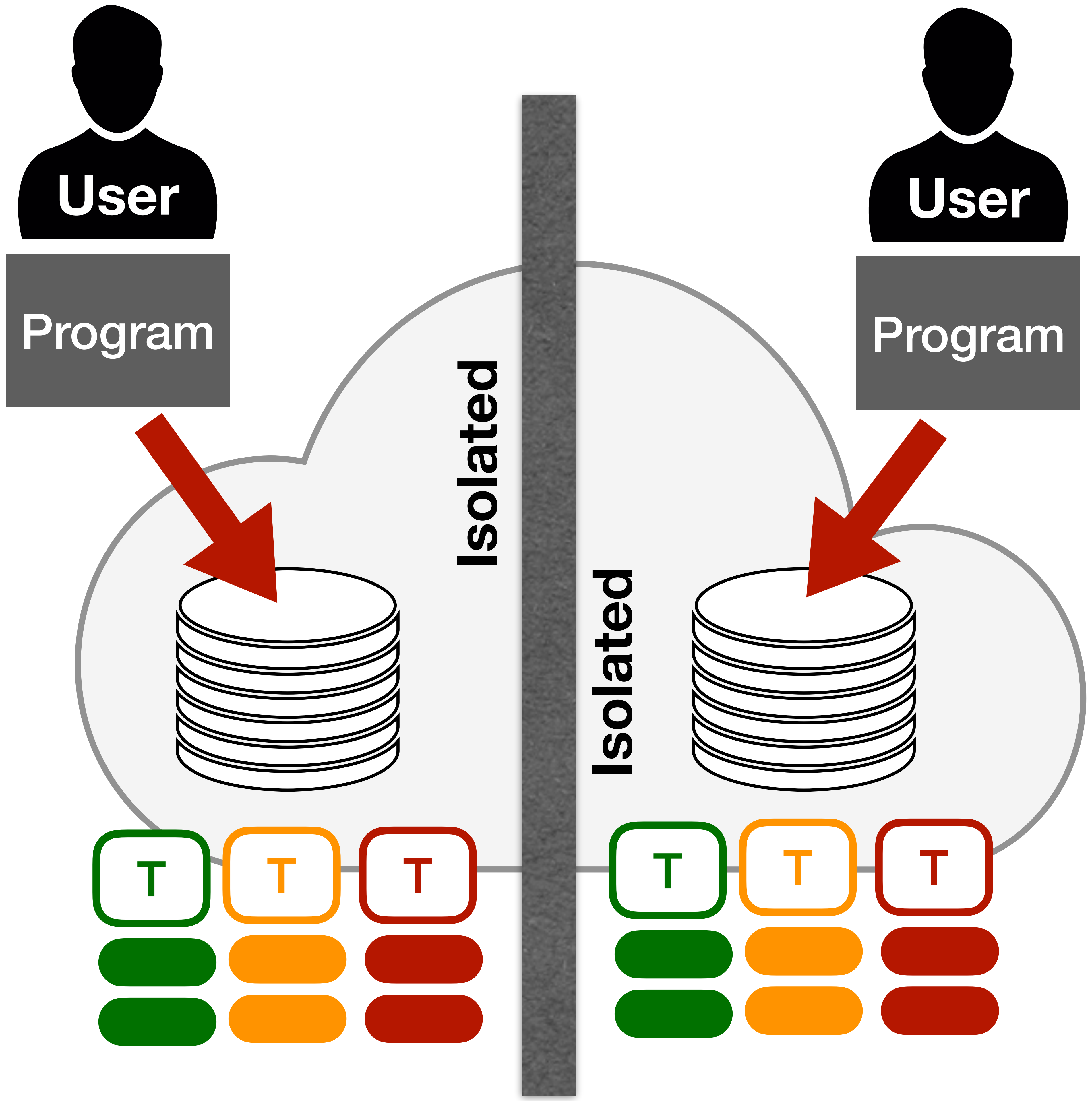
Schism: a Workload-Driven Approach to Database Replication and Partitioning

Carlo Curino curino@mit.edu
Evan Jones evanj@mit.edu
Yang Zhang yang@csail.mit.edu
Sam Madden madden@csail.mit.edu

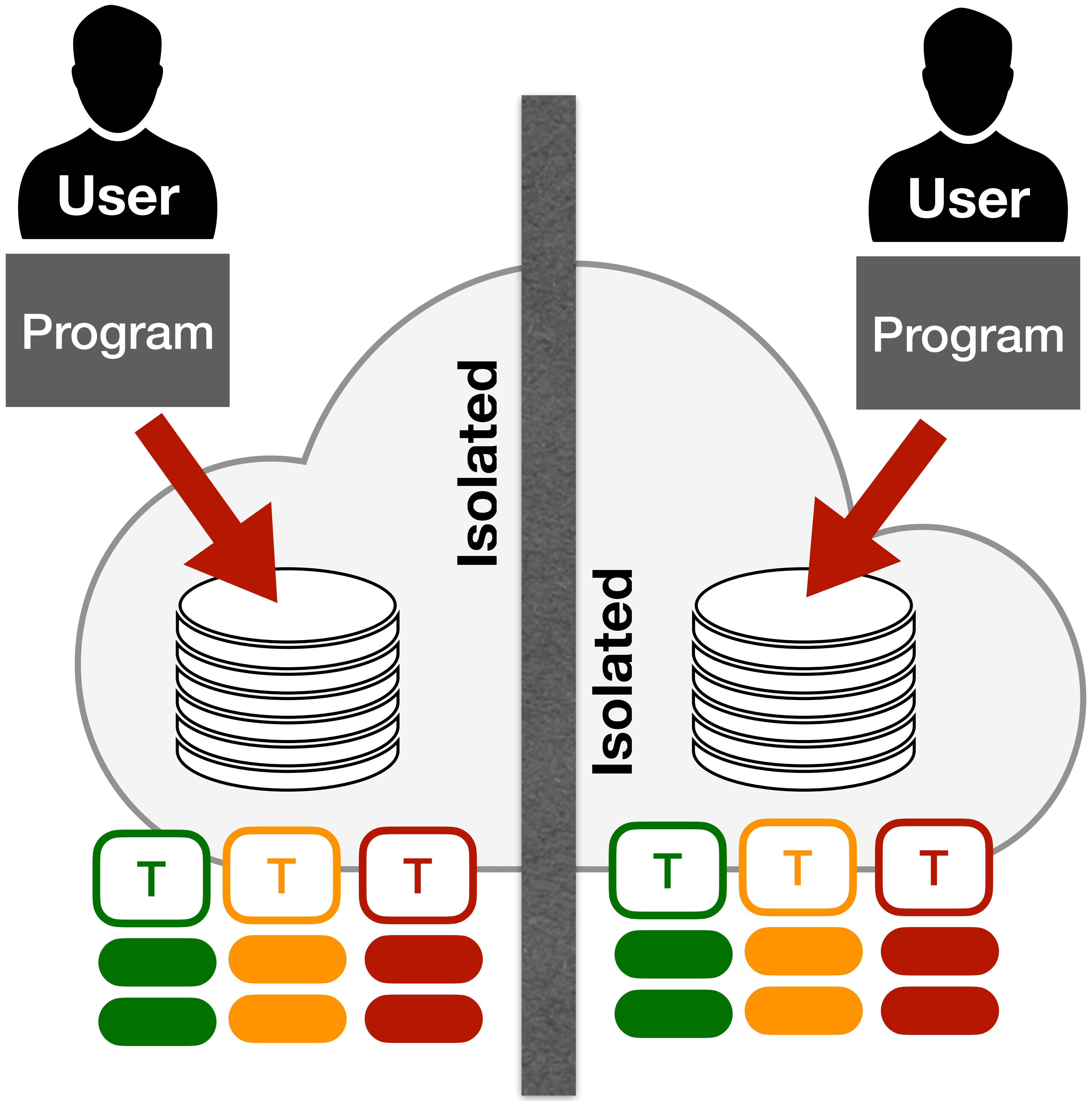
CONCURRENCY IN PARTITIONING

- Correctness?

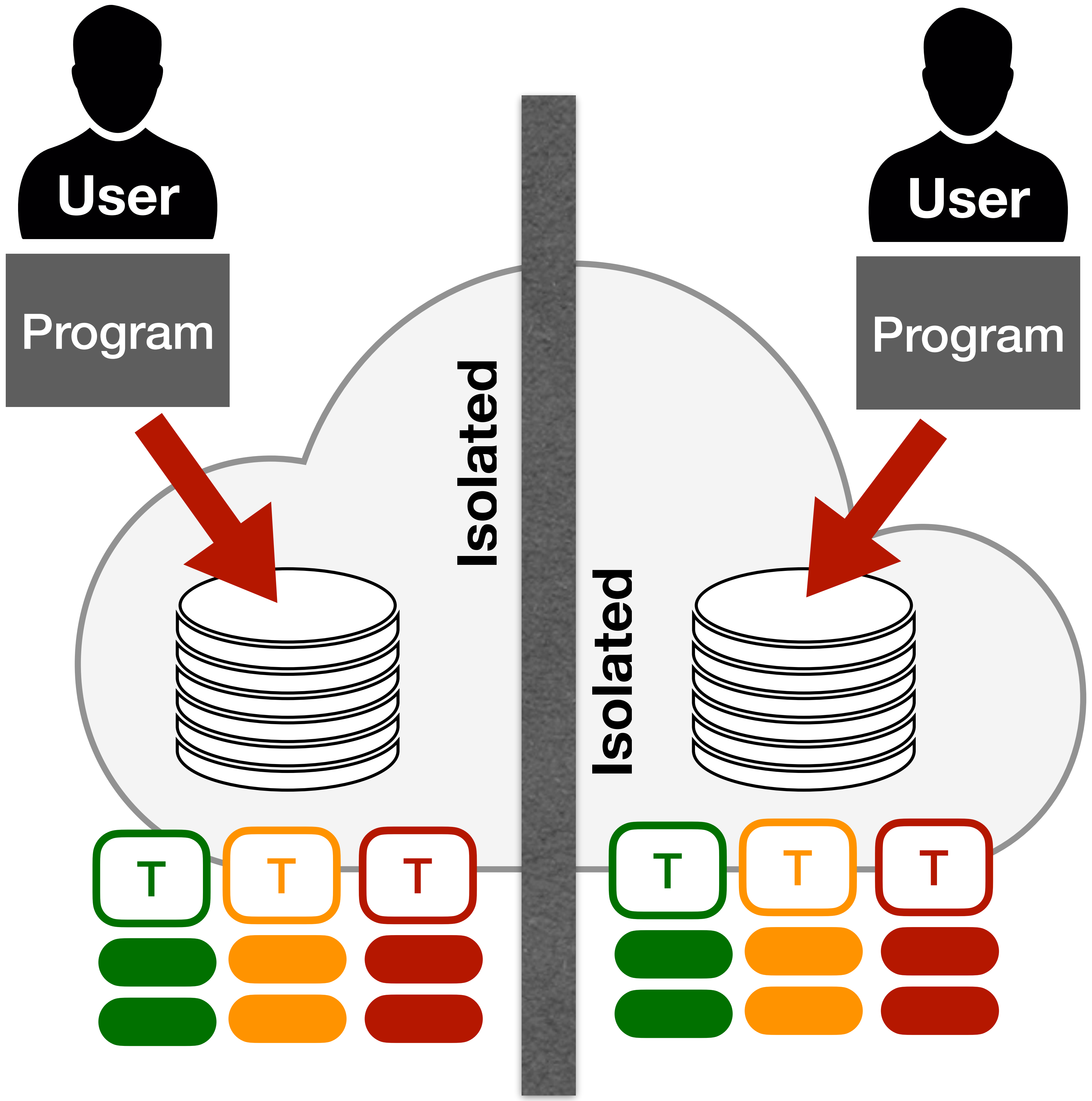
- Correctness?
- single-partition access = isolated & atomic



- Correctness?
- Single-partition access = Isolated & Atomic
- Eliminate concurrency anomalies by program refactoring



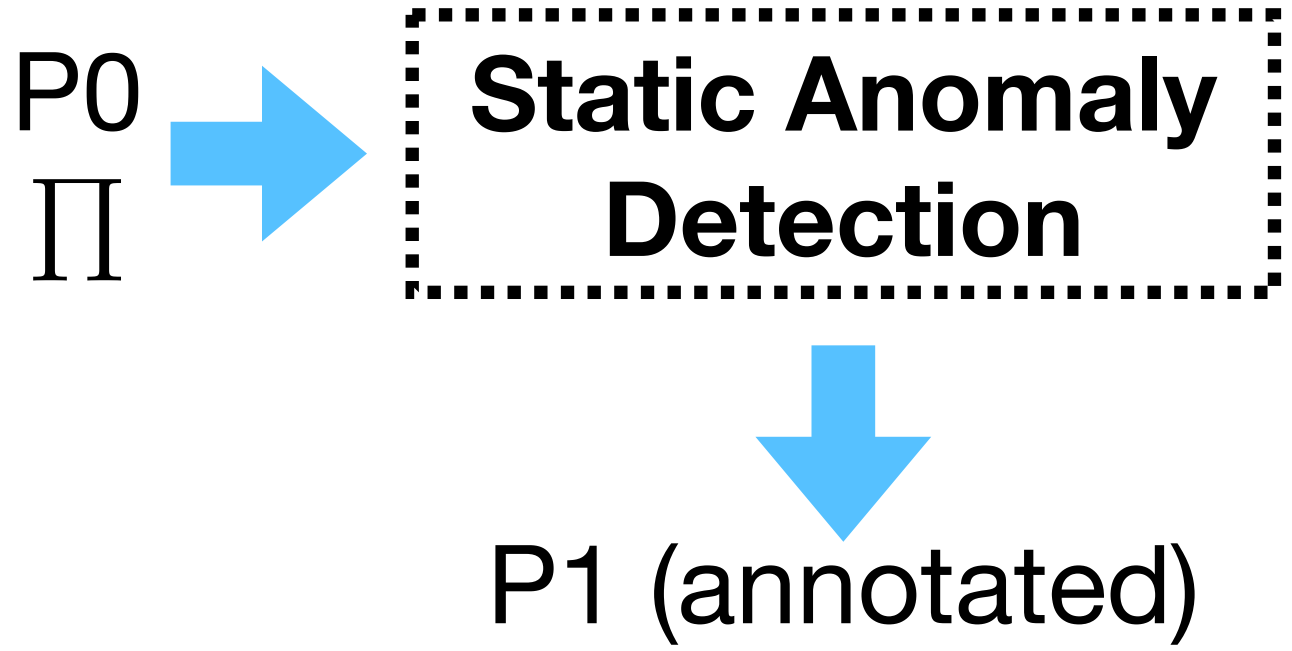
- Correctness?
- Single-partition access = Isolated & Atomic
- Eliminate concurrency anomalies by program refactoring
- Partitioning-aware symbolic execution



- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation

P0
Π
→ **Static Anomaly Detection**

- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation
- Annotation & Runtime
  - Isolated { }



- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation
- Annotation & Runtime
  - Isolated { }

PlayerByTeam (t)

```
ps := select * from PLAYER
       where p_t_id=t
foreach p in ps do
  print(p.*)
```

- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation
- Annotation & Runtime
  - Isolated { }

PlayerByTeam (t)

```
ps := select * from PLAYER
        where p_t_id=t
foreach p in ps do
  print(p.*)
```

PLAYER

	p_id	p_t_id	p_role	p_stat
I	P0	T1	captain	on
II	P1	T1	soldier	on
III	P2	T2	soldier	off

- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation
- Annotation & Runtime
  - Isolated { }

playerByTeam (t=T1)

...
Select P0

Select P1
...

PlayerByTeam (t)

```
ps := select * from PLAYER
        where p_t_id=t
foreach p in ps do
  print(p.*)
```

PLAYER

	p_id	p_t_id	p_role	p_stat
I	P0	T1	captain	on
II	P1	T1	soldier	on
III	P2	T2	soldier	off

- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation
- Annotation & Runtime
  - Isolated { }

PlayerByTeam (t)

```
ps := select * from PLAYER
        where p_t_id=t
foreach p in ps do
  print(p.*)
```

PLAYER

	p_id	p_t_id	p_role	p_stat
I	P0	T1	captain	on
II	P1	T1	soldier	on
III	P2	T2	soldier	off

playerByTeam (t=T1)

...
Select P0

Select P1
...

User

setPlayer (*p=P0*, *r=soldier*)

Set P0 to soldier

setPlayer (*p=P1*, *r=captain*)

Set P1 to captain

INTRODUCTION TO LACHESIS

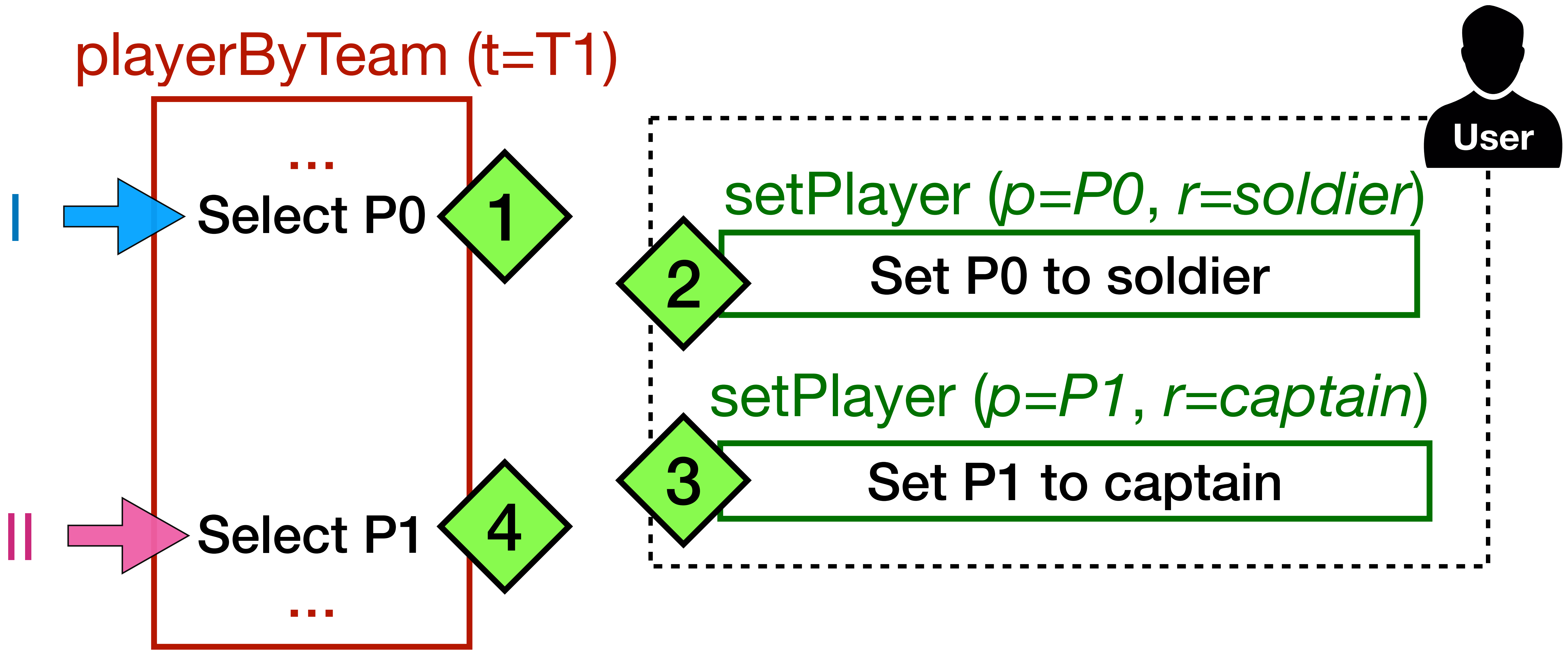
- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation
- Annotation & Runtime
  - Isolated { }

PlayerByTeam (t)

```
ps := select * from PLAYER
       where p_t_id=t
foreach p in ps do
  print(p.*)
```

PLAYER

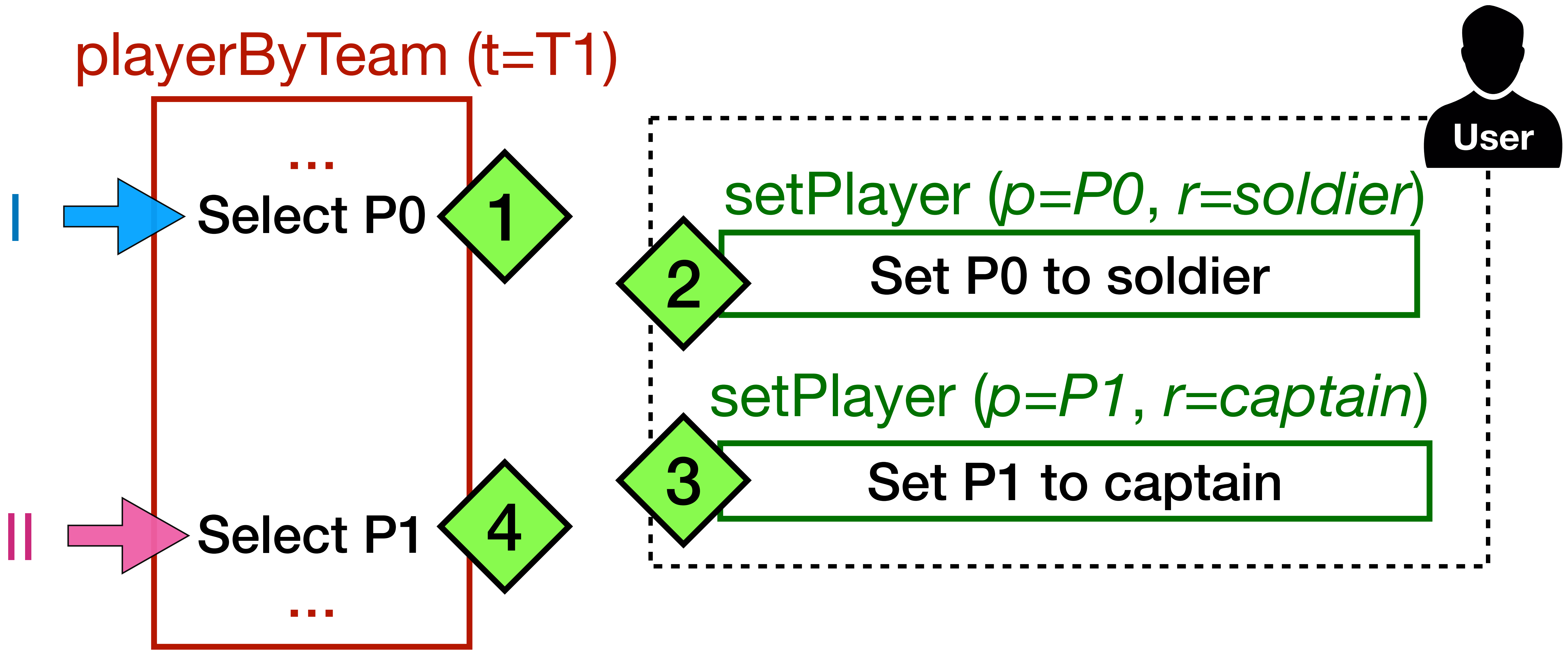
	p_id	p_t_id	p_role	p_stat
I	P0	T1	captain	on
II	P1	T1	soldier	on
III	P2	T2	soldier	off



- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation
- Annotation & Runtime
  - Isolated { }

PlayerByTeam (t)

```
ps := select * from PLAYER
      where p_t_id=t
foreach p in ps do
  print(p.*)
```



PLAYER

	p_id	p_t_id	p_role	p_stat
I	P0	T1	captain	on
II	P1	T1	soldier	on
III	P2	T2	soldier	off

// P0, 1, captain, on
// P1, 1, captain, on

Invariant Violated!

LACHESIS PIPELINE: CONTINUED

- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation
- Annotation & Runtime
  - Isolated { }

P0 Π → **Static Anomaly Detection** → P1 (annotated)

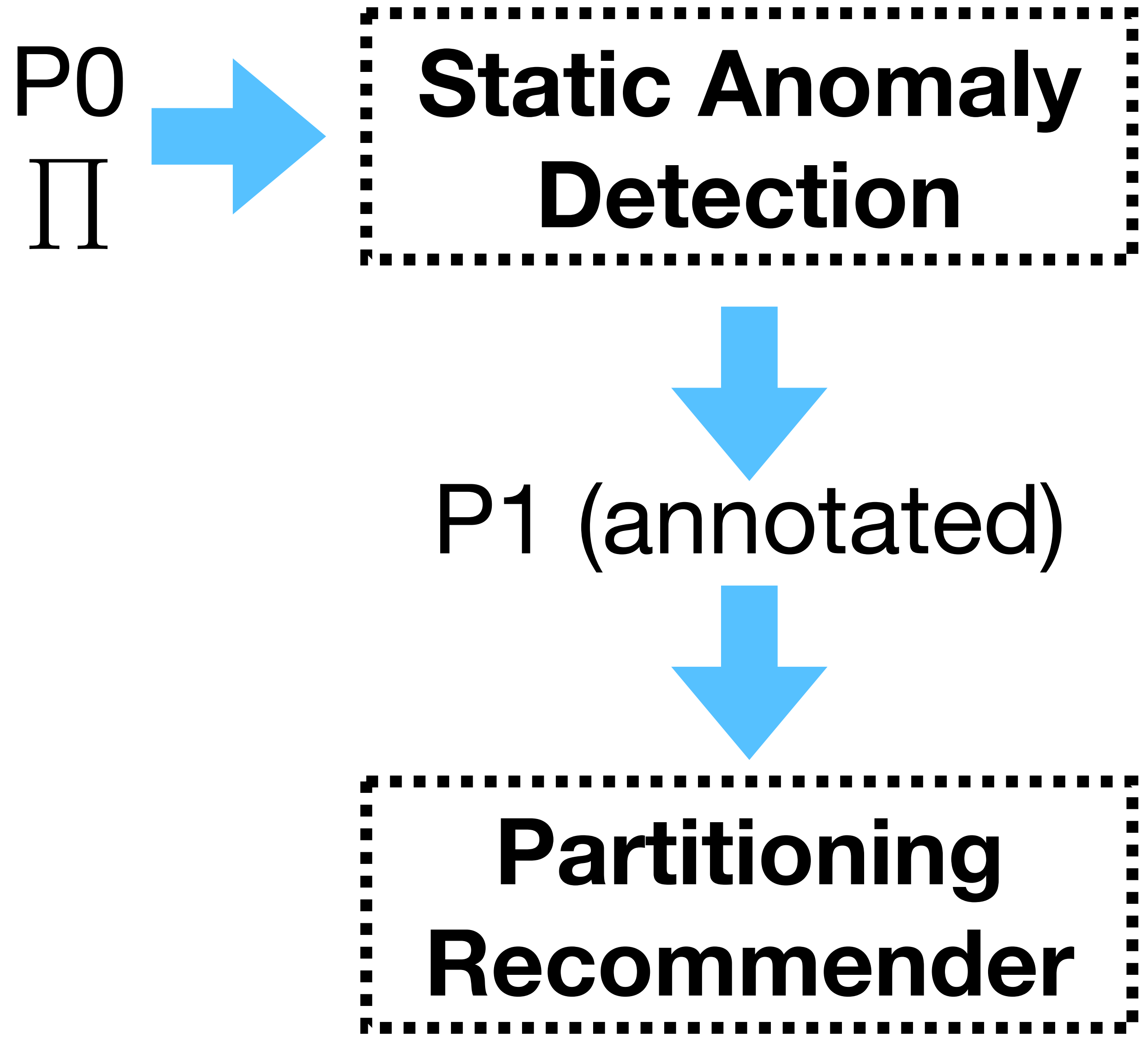
PlayerByTeam (t)

```
ps := select * from PLAYER
        where p_t_id=t
foreach p in ps do
  print(p.*)
```

PlayerByTeam (t)

```
Isolated {
  ps := select * from PLAYER
          where p_t_id=t
}
foreach p in ps do
  print(p.*)
```

- Symbolic analysis engine
  - program P0 (transactions & schema)
  - partitioning relation
- Annotation & Runtime
  - Isolated { }
- Partitioning Recommendation
  - Static & efficient
  - Comparable to Schism



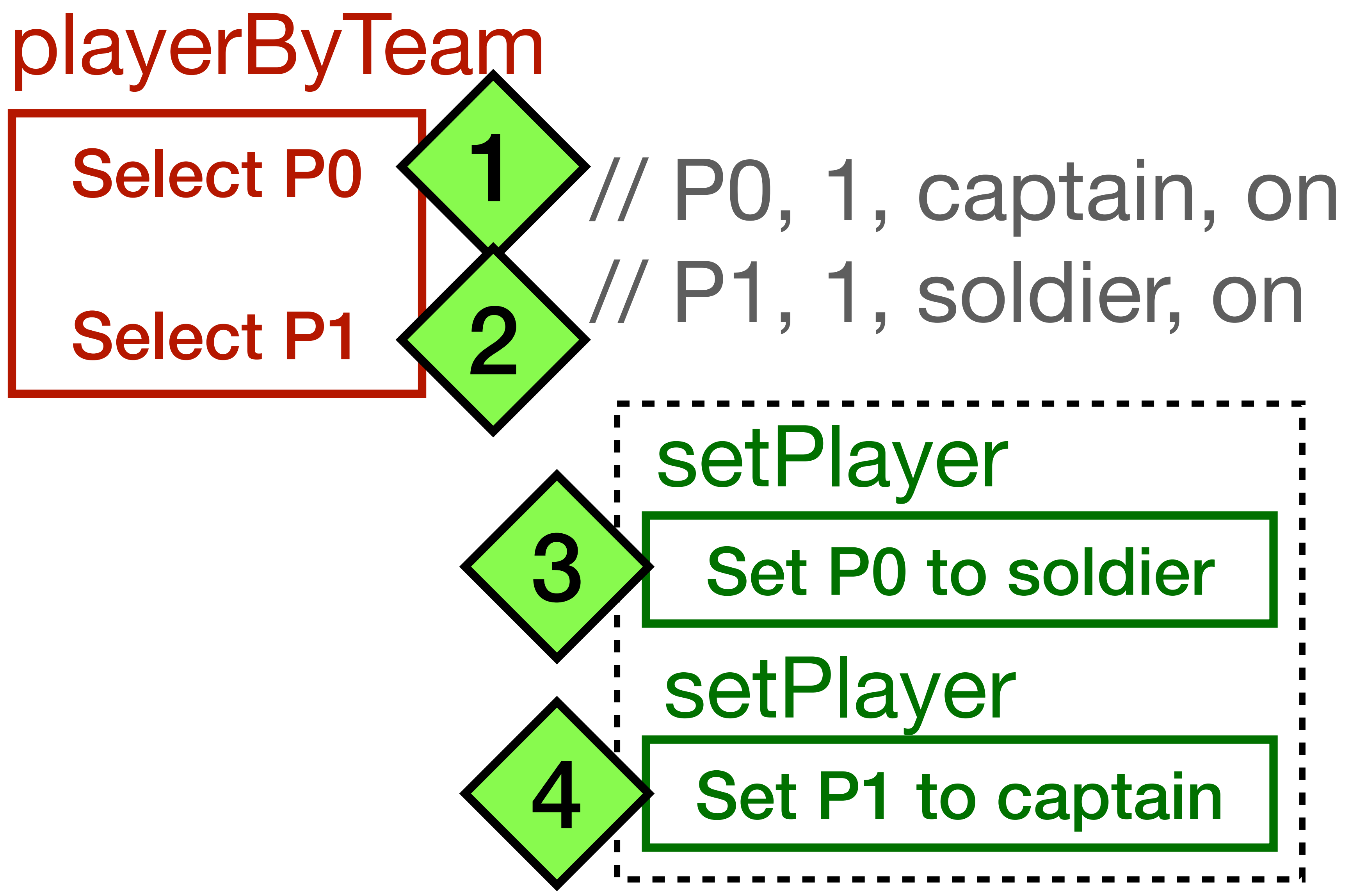
PARTITIONING RECOMMENDATION

- Recommended partitioning policy:

$$\forall r, r'.\ r.\texttt{p\_t\_id} = r'.\texttt{p\_t\_id} \Rightarrow (r, r') \in \Pi$$

PLAYER

	p_id	p_t_id	p_role	p_stat
I	P0	T1	captain	on
I	P1	T1	soldier	on
II	P2	T2	soldier	off



- Recommended partitioning policy:

$$\forall r, r'.\ r.\mathtt{p\_t\_id} = r'.\mathtt{p\_t\_id} \Rightarrow (r, r') \in \Pi$$

PLAYER

	p_id	p_t_id	p_role	p_stat
I	P0	T1	captain	on
I	P1	T1	soldier	on
II	P2	T2	soldier	off

setPlayer

1 Set P0 to soldier

setPlayer

2 Set P1 to captain

playerByTeam

3 Select P0

4 Select P1

// P0, 1, soldier, on

// P1, 1, captain, on

- Recommended partitioning policy:

$$\forall r, r'.\ r.\texttt{p\_t\_id} = r'.\texttt{p\_t\_id} \Rightarrow (r, r') \in \Pi$$

PlayerByTeam (t)

```
Isolated {
  ps := select * from PLAYER
        where p_t_id=t

foreach p in ps do
  print(p.*)
```

After partitioning

PlayerByTeam (t)

```
ps := select * from PLAYER
      where p_t_id=t
foreach p in ps do
  print(p.*)
```

PARTITIONING RECOMMENDATION

- Recommended partitioning policy:

  $$\forall r, r'.\ r.\texttt{p\_t\_id} = r'.\texttt{p\_t\_id} \Rightarrow (r, r') \in \Pi$$

- Partitioning eliminates annotations
- Annotations are costly
- 40% fewer annotations:
  - average of 23% higher throughput & 14% lower latency

LIMITATION OF PARTITIONING

- Each table is
  - partitioned for one access pattern
- Conflicting access patterns?
- $\forall r, r'.\ r.\texttt{p\_t\_id} = r'.\texttt{p\_t\_id} \Rightarrow (r, r') \in \Pi$

LIMITATION OF PARTITIONING

- Each table is
  - partitioned for one access pattern
- Conflicting access patterns?
- $\forall r, r'.\ r.\texttt{p\_t\_id} = r'.\texttt{p\_t\_id} \Rightarrow (r, r') \in \Pi$

A1

PLAYER

	p_id	p_t_id	p_role	p_stat
I	P0	T1	captain	on
I	P1	T1	soldier	on
II	P2	T2	soldier	off

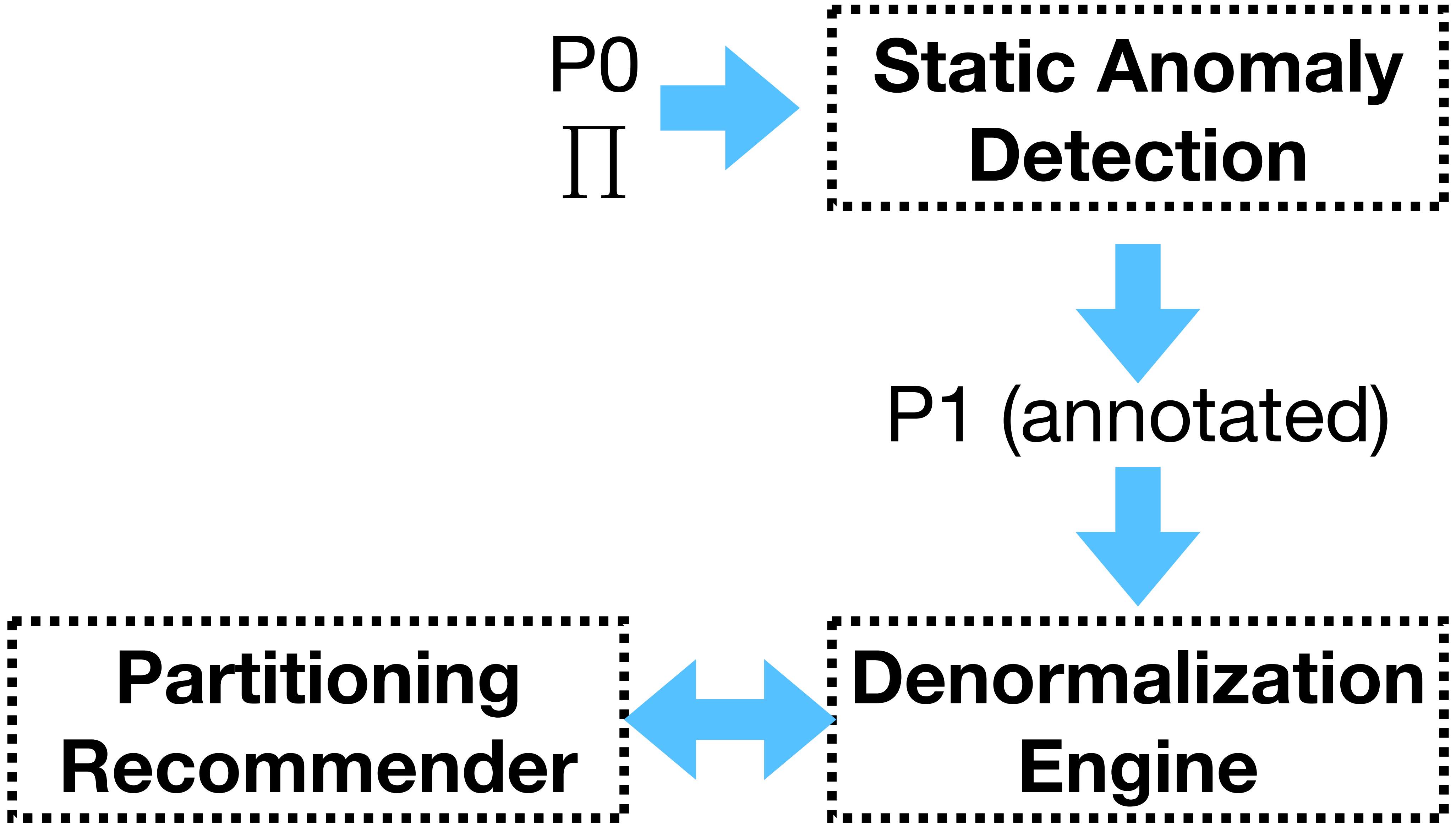
A2

PLAYER

	p_id	p_t_id	p_role	p_stat
II	P0	T1	captain	on
I	P1	T1	soldier	on
II	P2	T2	soldier	off

LIMITATION OF PARTITIONING

- Refactor the schema:
  - A new table for each anomalous access pattern
  - Rewrite program
  - Repartition



- Running example:
  - additional three tables

GAME ⋈ PLAYER

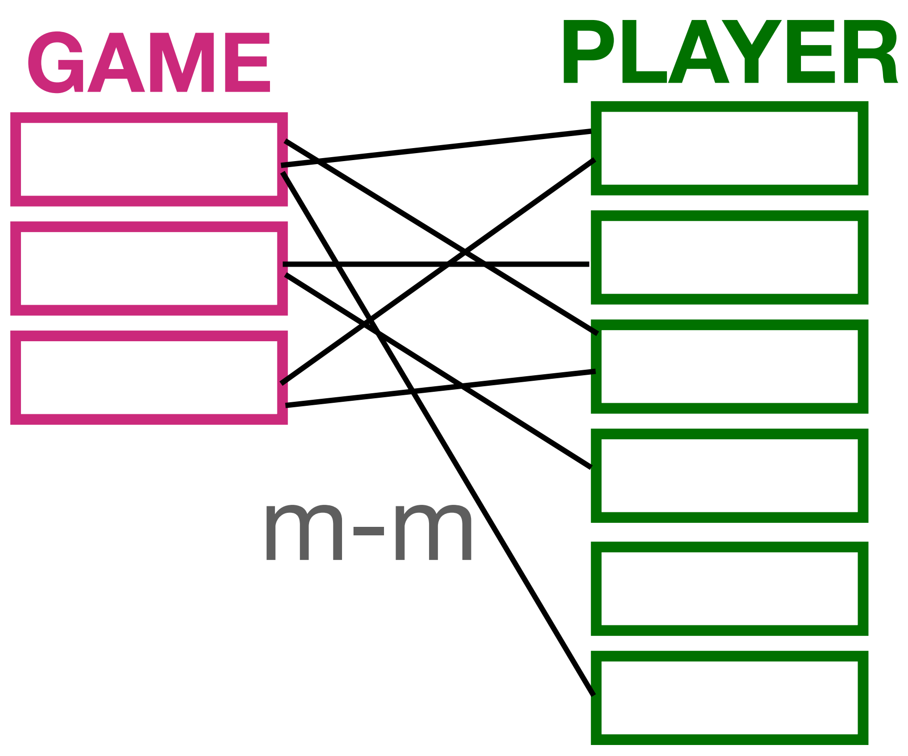
g_id	p_id

GAME

g_id	g_stat	g_min_player

PLAYER

p_id	p_t_id	p_role	p_stat



EXAMPLE

- Running example:
  - additional three tables
  - additional one transaction

playerByGame (*g*)

```
Game := select * from GAME where g_id=g
print(game.*)
ids := select p_id from G⋈P where g_id=g
foreach id in ids do
  p := select * from PLAYER where p_id=id
  print(p.*)
```

GAME ⋈ PLAYER

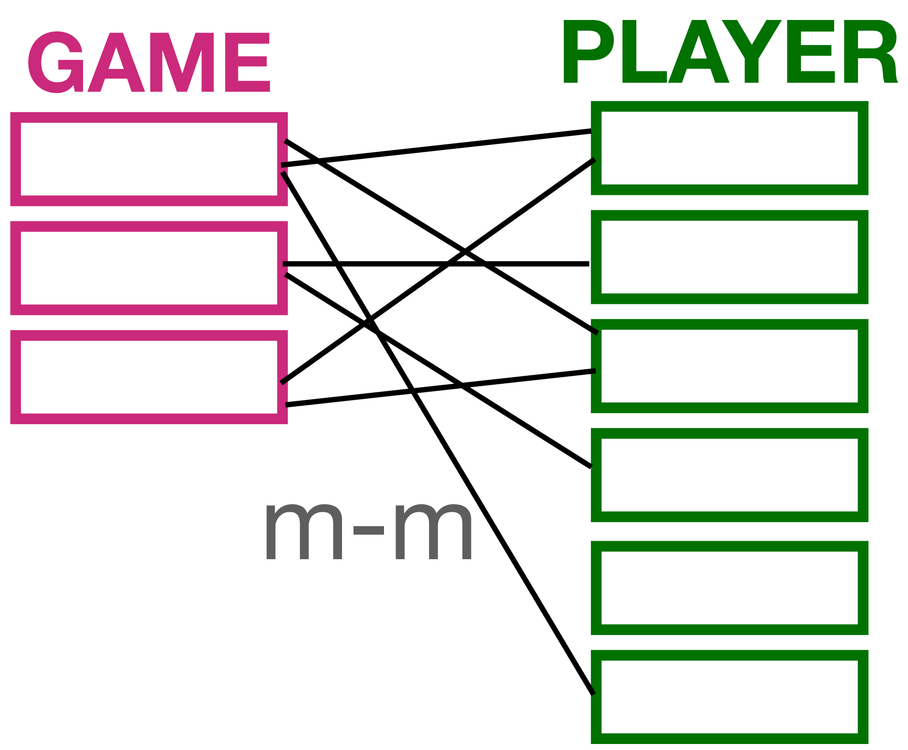
g_id	p_id

GAME

g_id	g_stat	g_min_player

PLAYER

p_id	p_t_id	p_role	p_stat



- Running example:
  - additional three tables
  - additional one transaction
- Anomaly is not fixed by partitioning

playerByGame (*g*)

```
Game := select * from GAME where g_id=g
print(game.*)
ids := select p_id from G⋈P where g_id=g
foreach id in ids do
  p := select * from PLAYER where p_id=id
  print(p.*)
```

PLAYER

p_id	p_t_id	p_role	p_stat
P0	1	soldier	on
P1	1	captain	on
P2	2	captain	off

EXAMPLE

- Running example:
  - additional three tables
  - additional one transaction
- Anomaly is not fixed by partitioning

playerByGame (*g*)

```
Game := select * from GAME where g_id=g
print(game.*)
ids := select p_id from G⋈P where g_id=g
foreach id in ids do
  p := select * from PLAYER where p_id=id
  print(p.*)
```

PLAYER

p_id	p_t_id	p_role	p_stat
P0	1	soldier	on
P1	1	captain	on
P2	2	captain	off

EXAMPLE

- Running example:
  - additional three tables
  - additional one transaction
- Anomaly is not fixed by partitioning

playerByGame (*g*)

```
Game := select * from GAME where g_id=g
print(game.*)
ids := select p_id from G⋈P where g_id=g
foreach id in ids do
  p := select * from PLAYER where p_id=id
  print(p.*)
```

PLAYER

p_id	p_t_id	p_role	p_stat
P0	1	soldier	on
P1	1	captain	on
P2	2	captain	off

PLAYER_BY_GAME

g_id	p_id	p_t_id	p_role	p_stat
G1	P0	1	soldier	on
G1	P2	2	captain	off

EXAMPLE

- Running example:
  - additional three tables
  - additional one transaction
- Anomaly is not fixed by partitioning

playerByGame (*g*)

```
Game := select * from GAME where g_id=g
print(game.*)
ids := select p_id from G⋈P where g_id=g
foreach id in ids do
   p := select * from PLAYER_BY_GAME
               where p_id=id
   print(p.*)
```

PLAYER

p_id	p_t_id	p_role	p_stat
P0	1	soldier	on
P1	1	captain	on
P2	2	captain	off

$\sqsubseteq$

PLAYER_BY_GAME

g_id	p_id	p_t_id	p_role	p_stat
G1	P0	1	soldier	on
G1	P2	2	captain	off

GAME

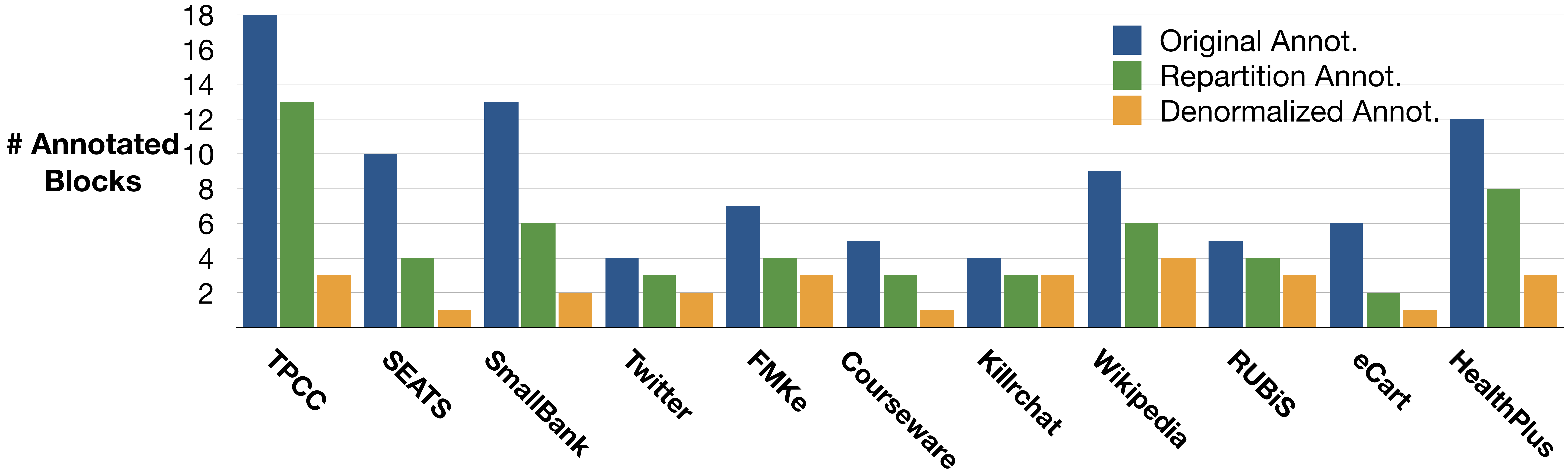
g_id	g_stat	g_min_player
G1	active	2

IMPLEMENTATION & EMPIRICAL RESULTS

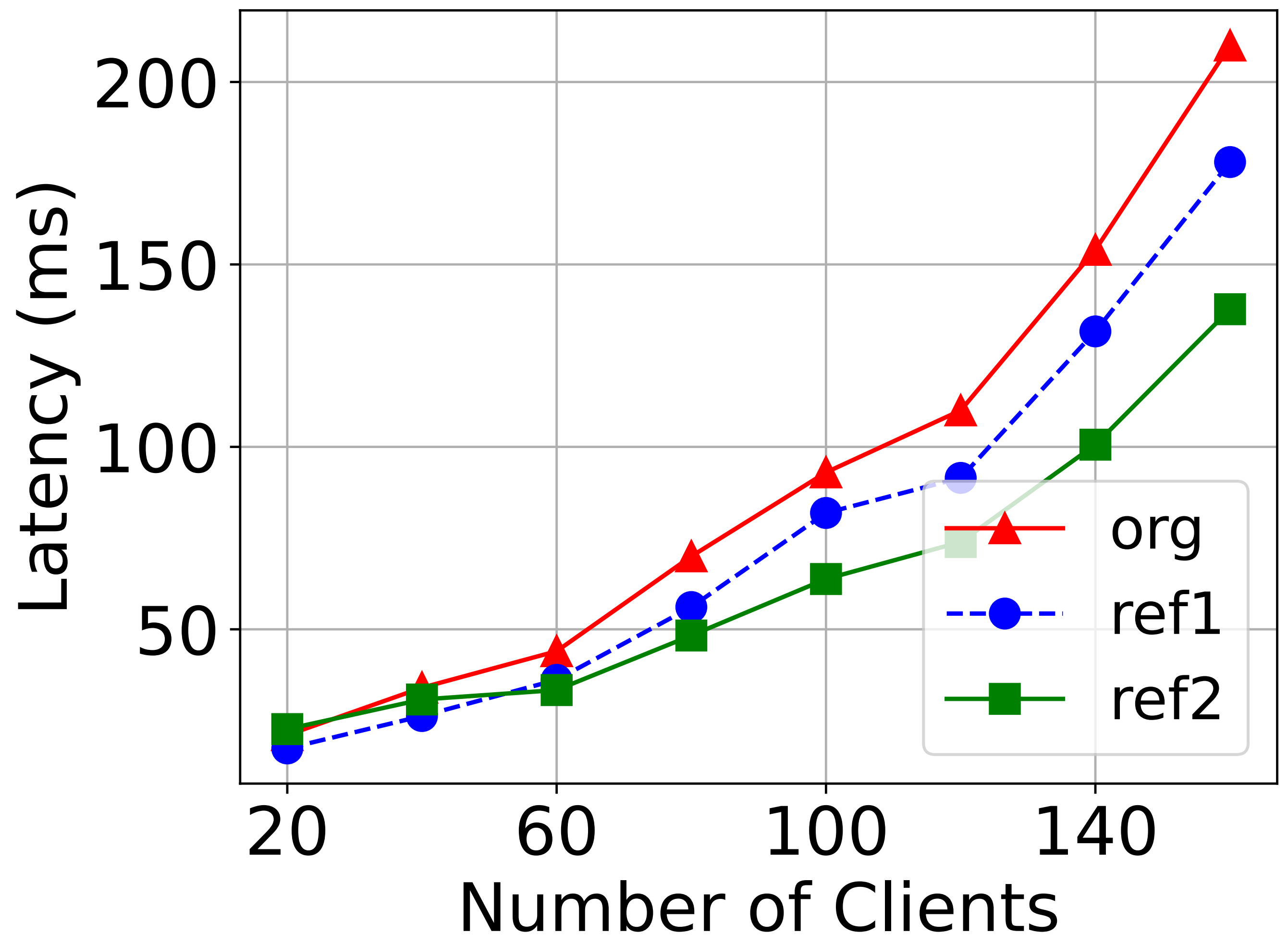
- ZooKeeper runtime
- 11 Benchmarks

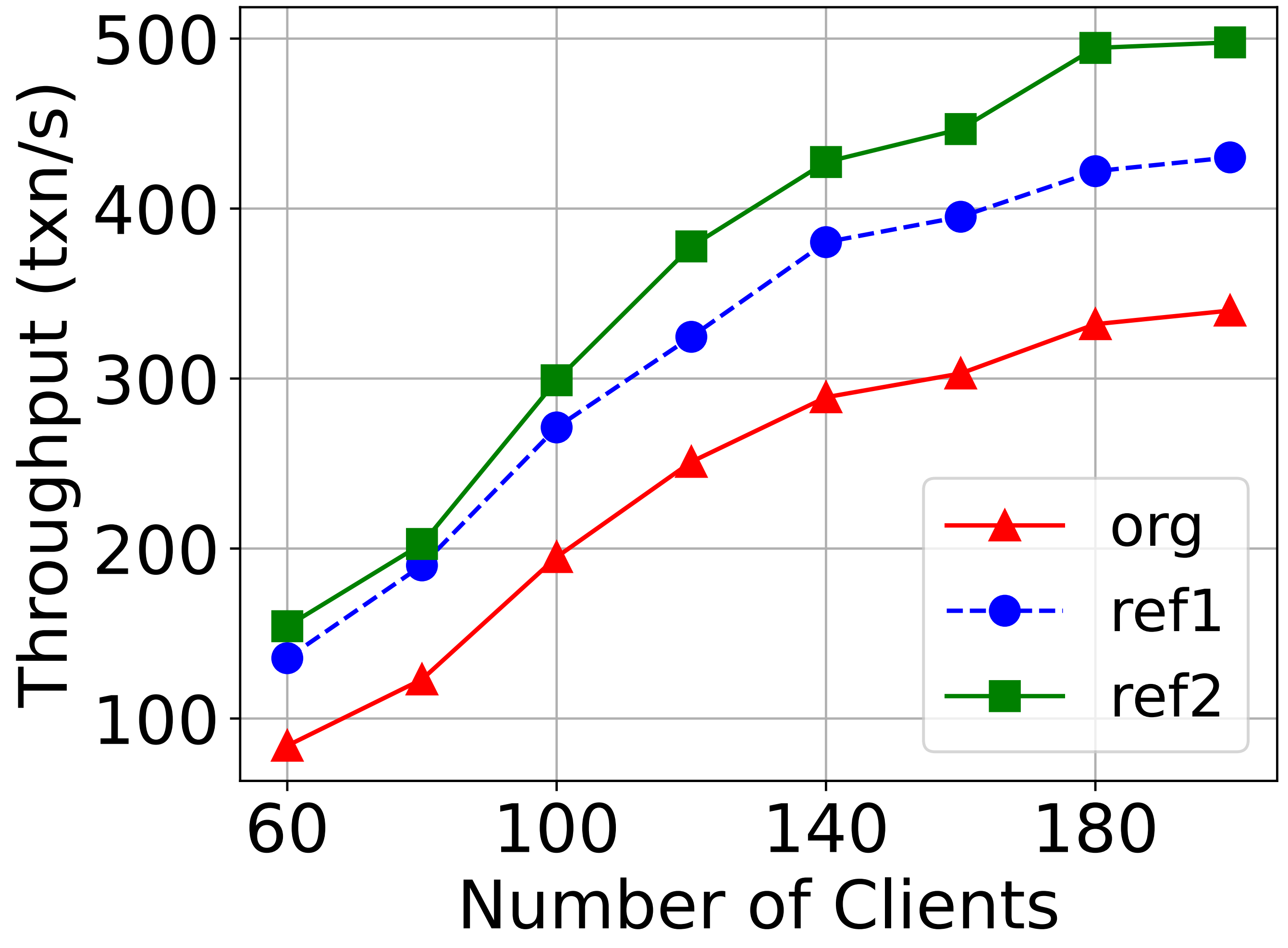
APACHE ZooKeeper™

- ZooKeeper runtime
- 11 Benchmarks
- Number of annotations
  - 40% reduction
  - Additional 32% reduction



- ZooKeeper runtime
- 11 Benchmarks
- Number of annotations
  - 40% reduction
  - Additional 32% reduction
- Lower latency + Higher throughput





TPC-C benchmark

CONCLUSION

CONCLUSION

- Modern database systems come with a plethora of complex and subtle concurrency semantics.

CONCLUSION

- Modern database systems come with a plethora of complex and subtle concurrency semantics.
- The developers have a hard time reasoning about programs.

CONCLUSION

- Modern database systems come with a plethora of complex and subtle concurrency semantics.
- The developers have a hard time reasoning about programs.
- A symbolic program analysis engine to identify and report undesirable behaviors from a given database program

CONCLUSION

- Modern database systems come with a plethora of complex and subtle concurrency semantics.
- The developers have a hard time reasoning about programs.
- A symbolic program analysis engine to identify and report undesirable behaviors from a given database program
- By analyzing the access patterns of the database program, the program can be rewritten to be optimized for deployment on a particular clustering architecture.

CONCLUSION

- Modern database systems come with a plethora of complex and subtle concurrency semantics.
- The developers have a hard time reasoning about programs.
- A symbolic program analysis engine to identify and report undesirable behaviors from a given database program
- By analyzing the access patterns of the database program, the program can be rewritten to be optimized for deployment on a particular clustering architecture.
- We report results from empirical experiments to support our claims.

ACKNOWLEDGEMENT

Suresh
Jagannathan

Benjamin
Delaware

Kartik Nagar