



SYMBOLIC ANALYSIS OF WEAK CONCURRENCY SEMANTICS IN MODERN DATABASE PROGRAMS

Kia Rahmani

Aug 2, 2022

University of Texas at Austin
Austin, TX

INTRODUCTION

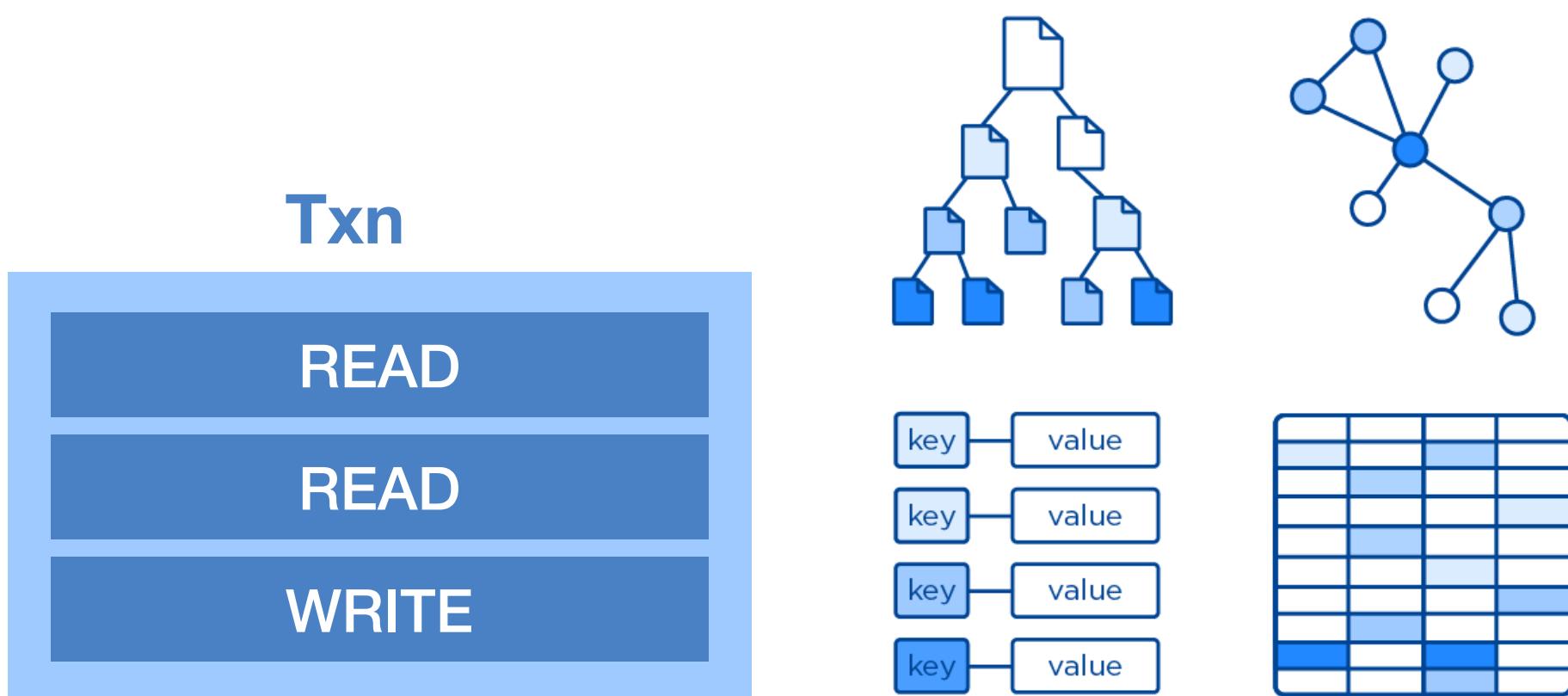
(MAIN SUBJECT: DATABASE SYSTEMS)

TRADITIONAL DATABASE SYSTEMS & PROGRAMS

- Data is ubiquitously managed by databases

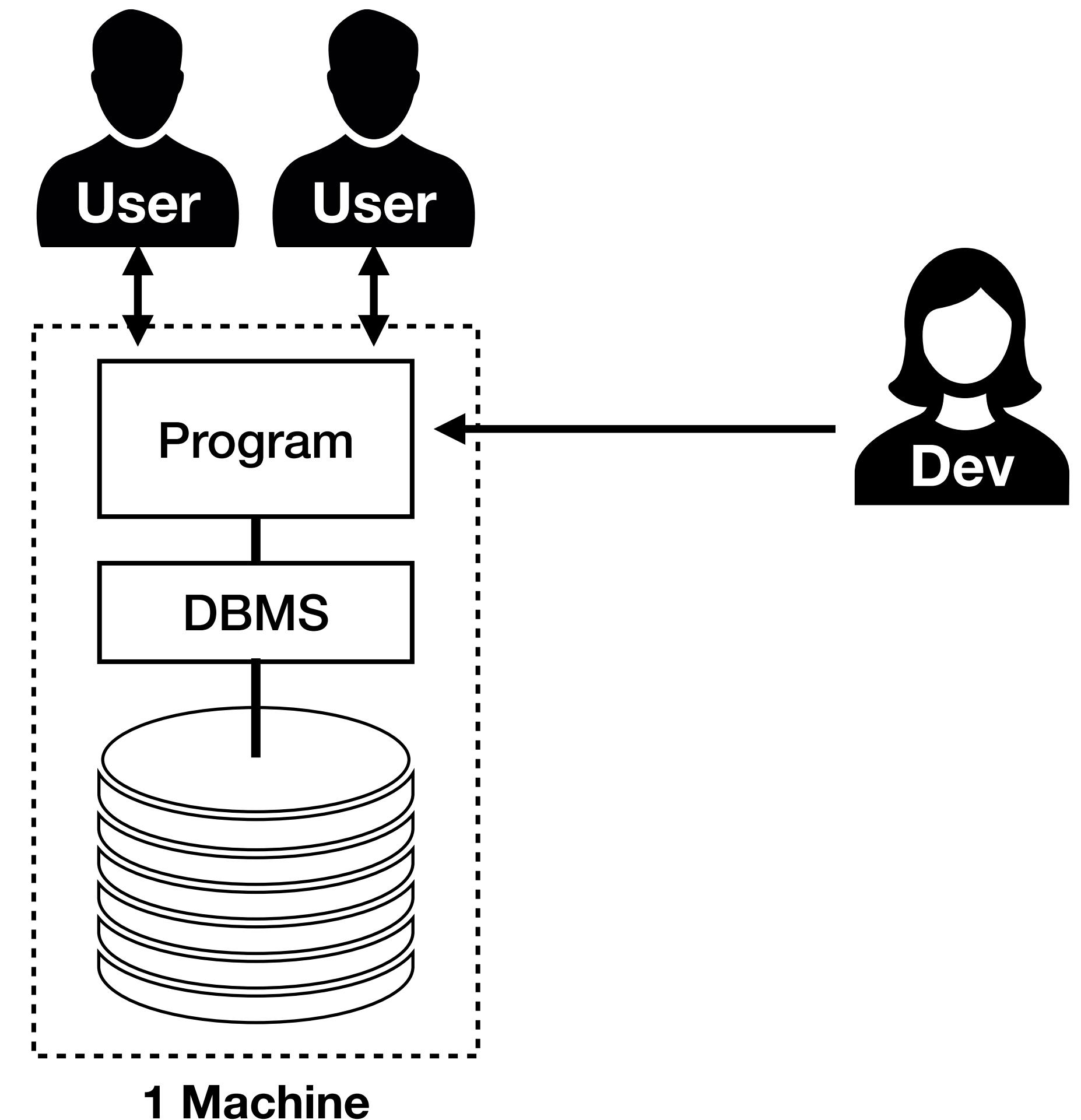
TRADITIONAL DATABASE SYSTEMS & PROGRAMS

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



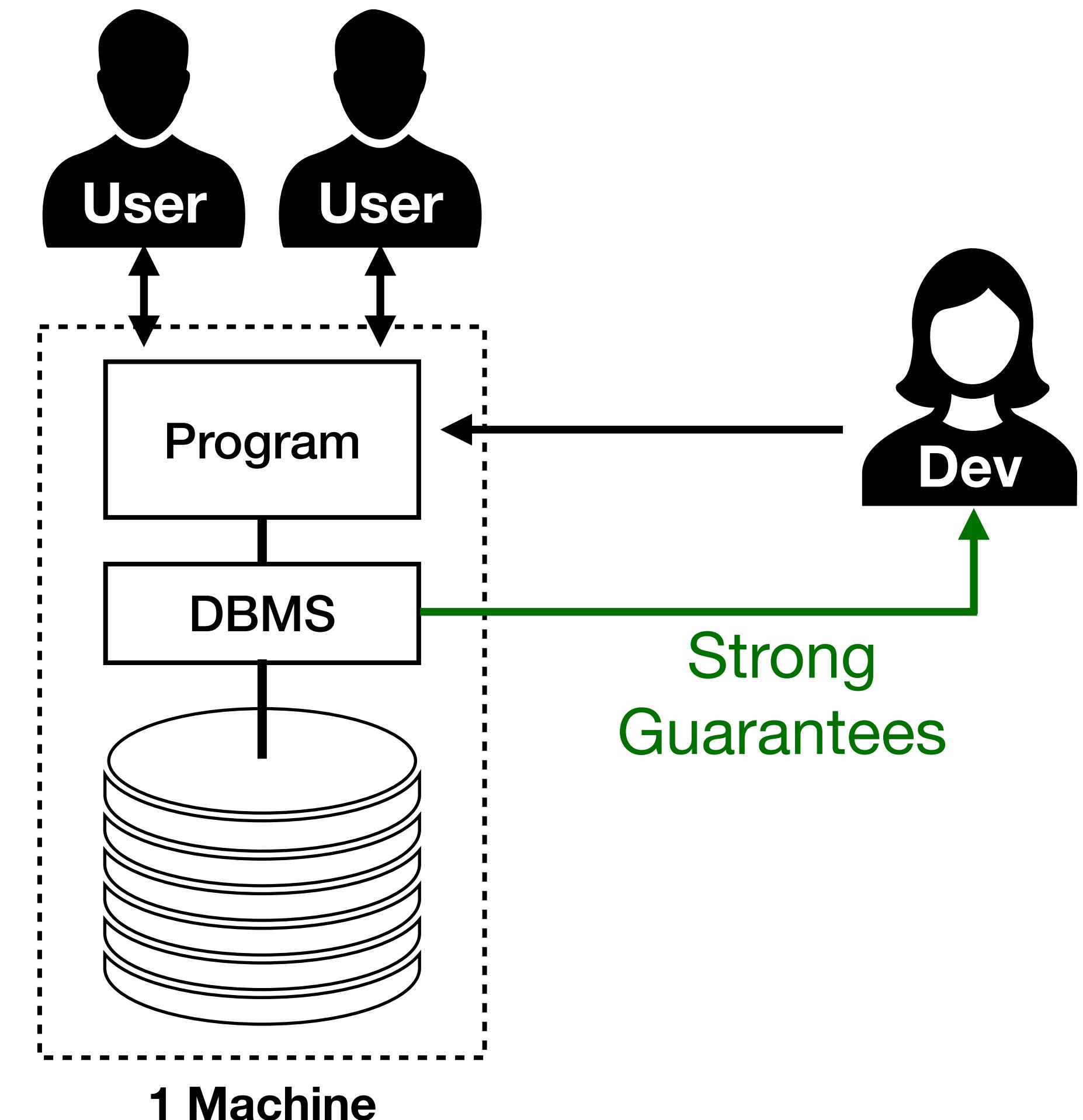
TRADITIONAL DATABASE SYSTEMS & PROGRAMS

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



TRADITIONAL DATABASE SYSTEMS & PROGRAMS

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



MODERN DATABASE SYSTEMS & PROGRAMS

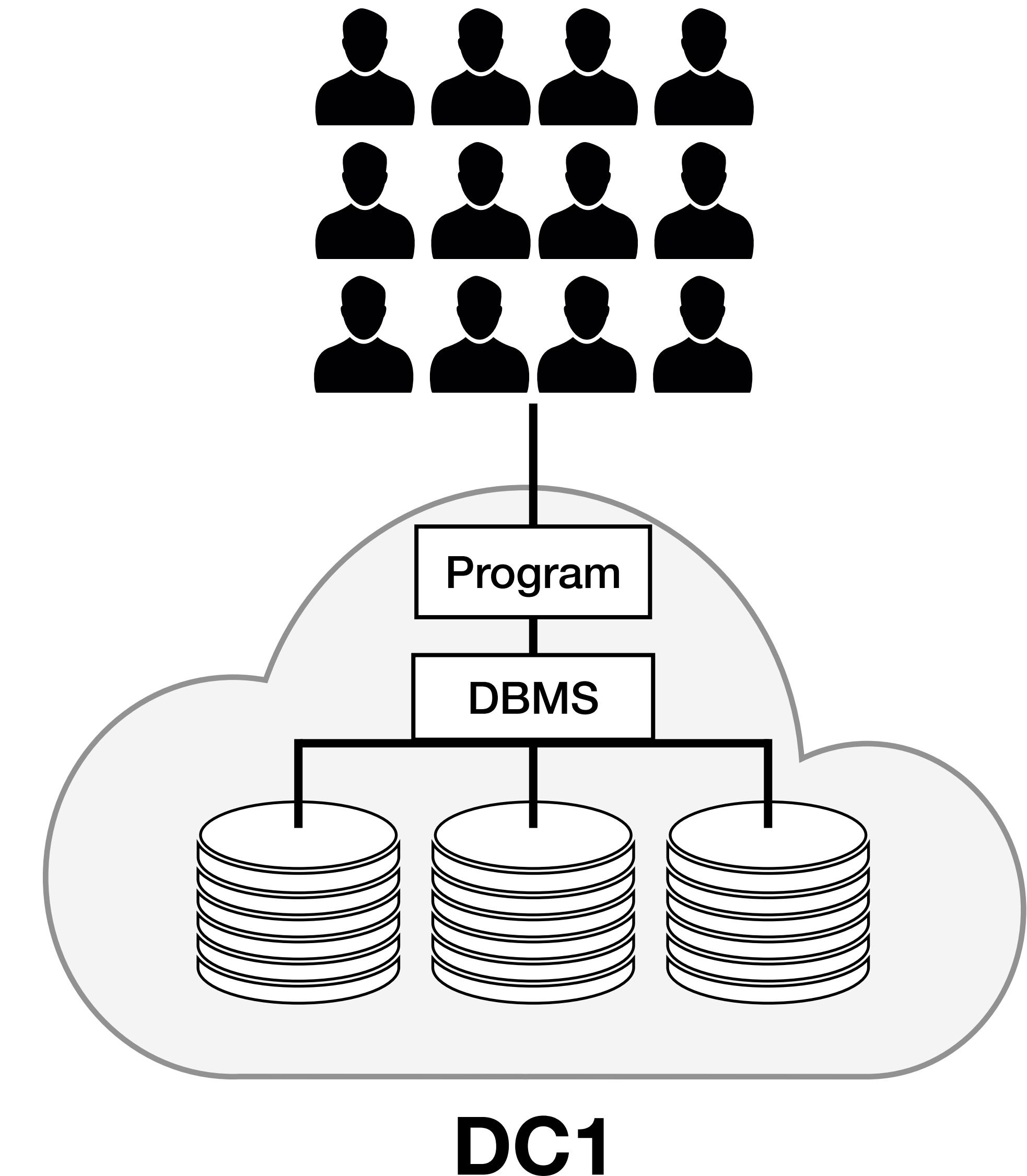
- Single machine = poor performance

MODERN DATABASE SYSTEMS & PROGRAMS

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

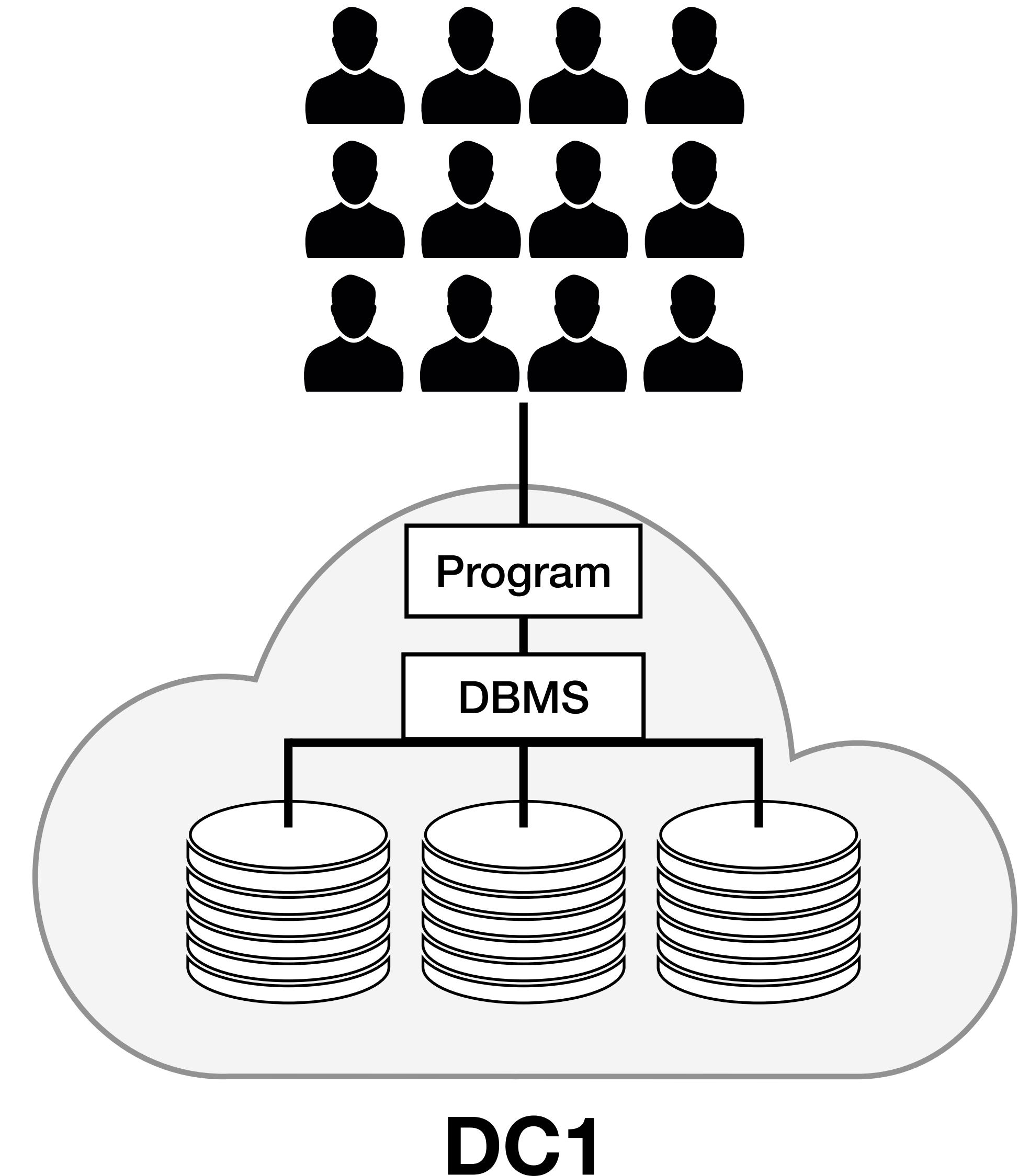


amazon
DynamoDB



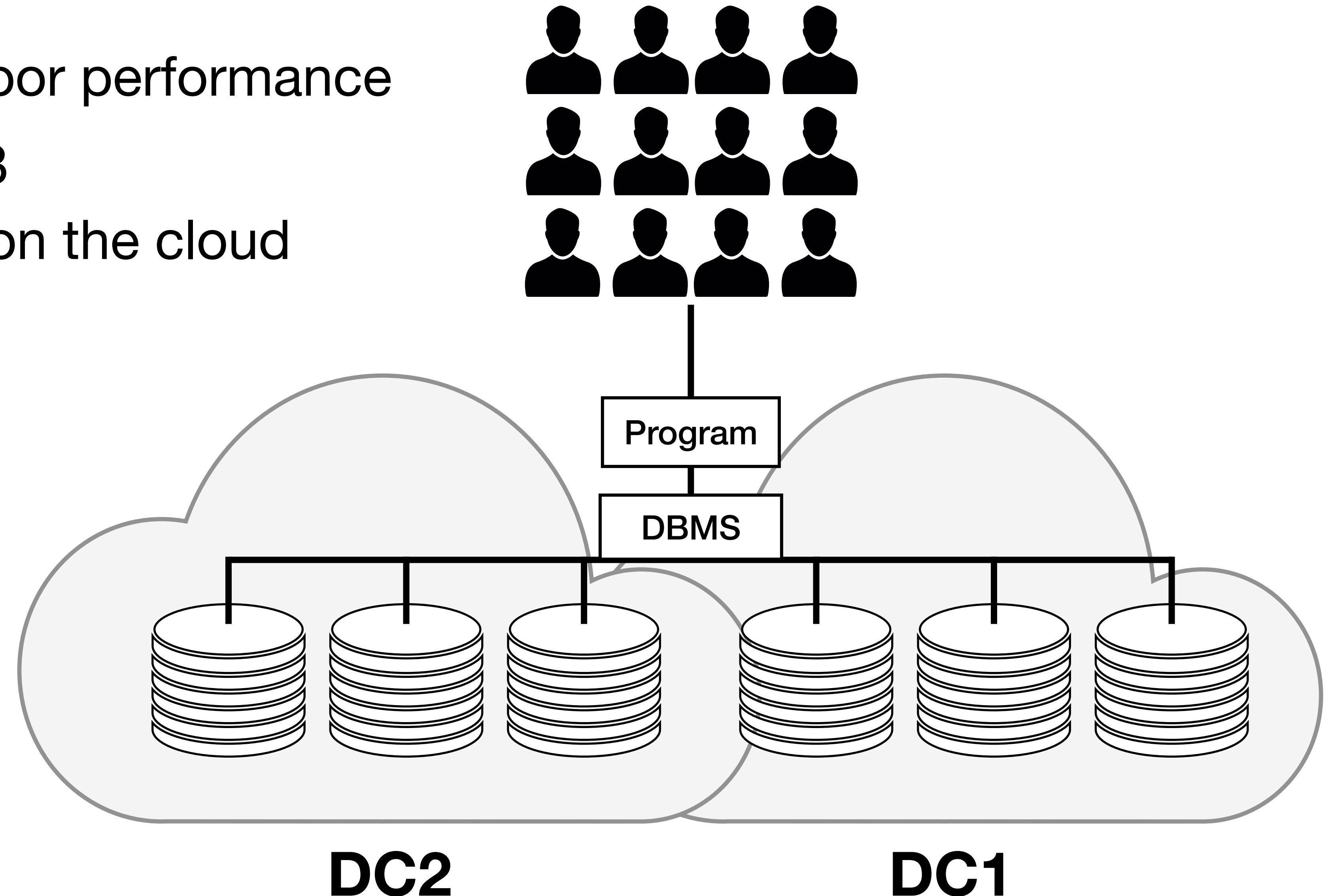
MODERN DATABASE SYSTEMS & PROGRAMS

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



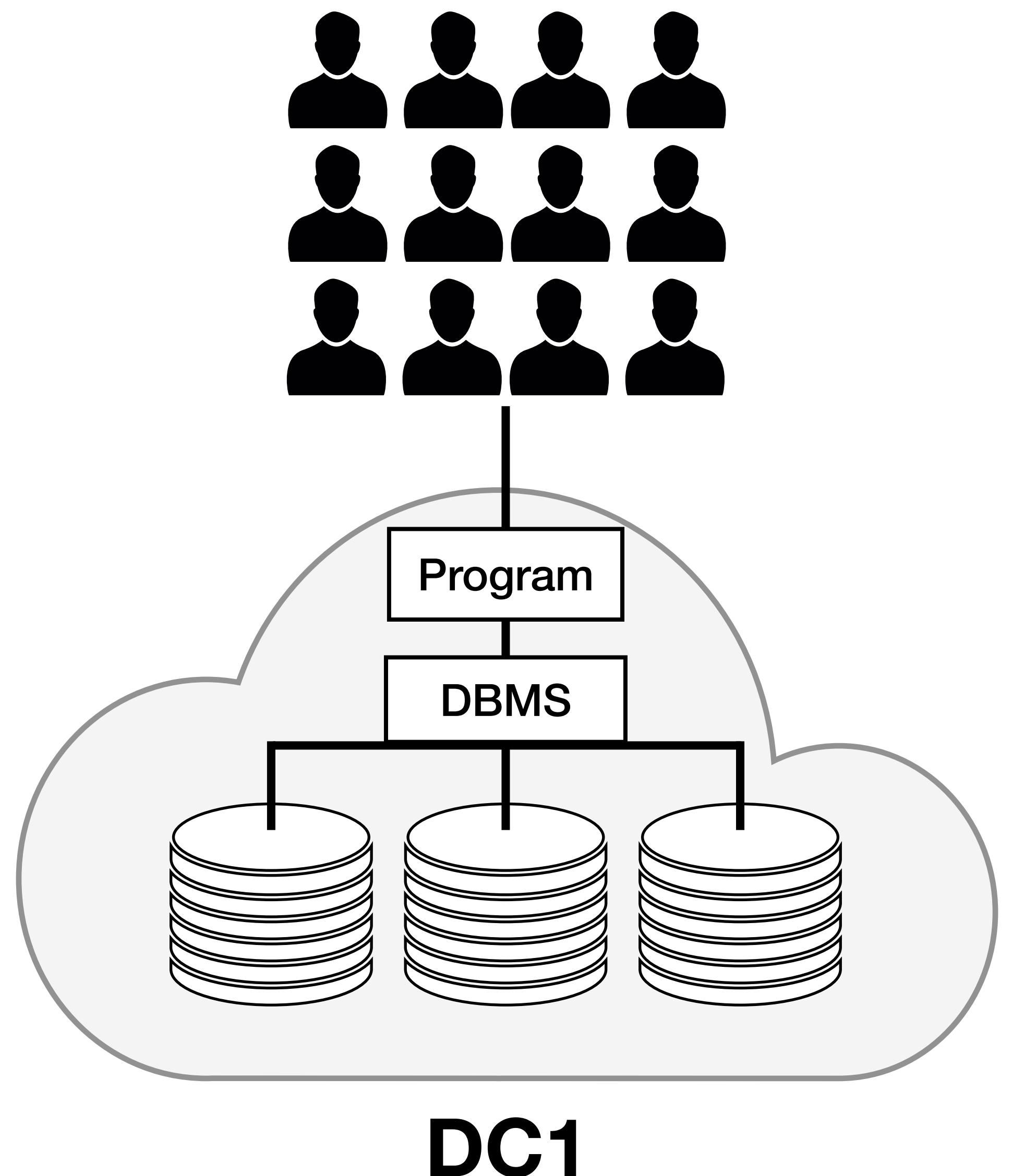
MODERN DATABASE SYSTEMS & PROGRAMS

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



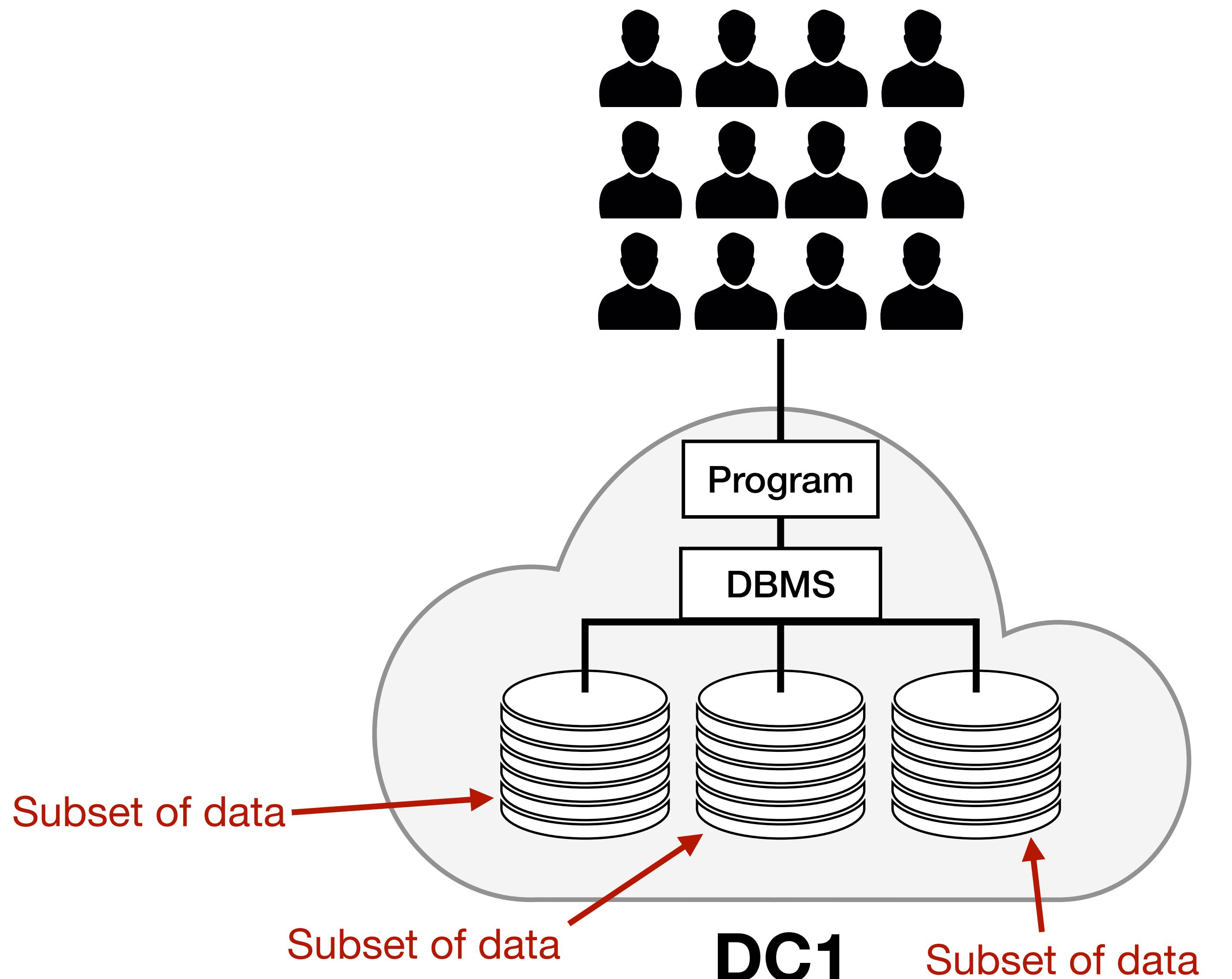
SCALING DATABASE CLUSTERS

- Clustering architectures
 - ▶ Partitioning
 - ▶ Replicating



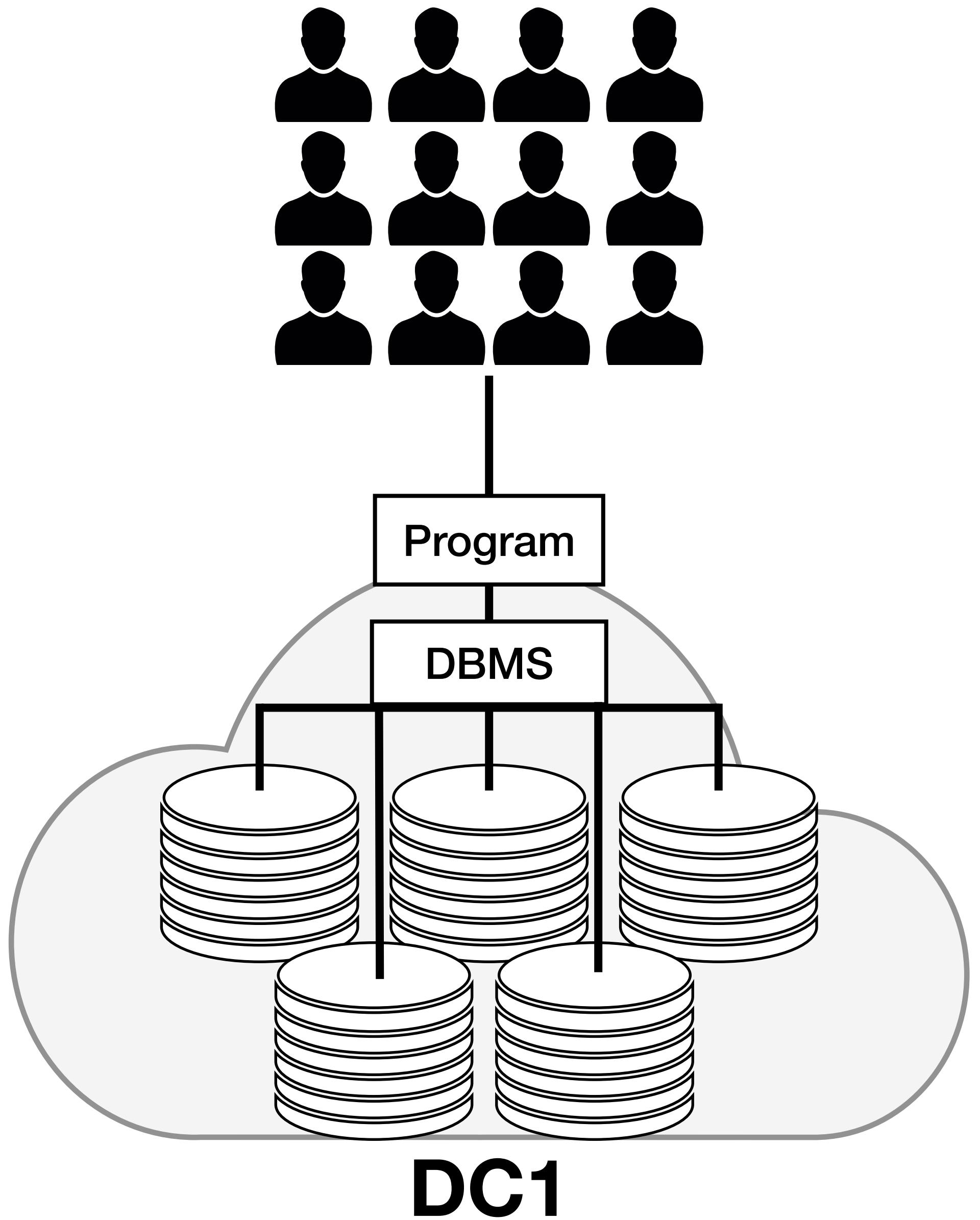
SCALING DATABASE CLUSTERS

- Clustering architectures
 - ▶ Partitioning
 - ▶ Replicating



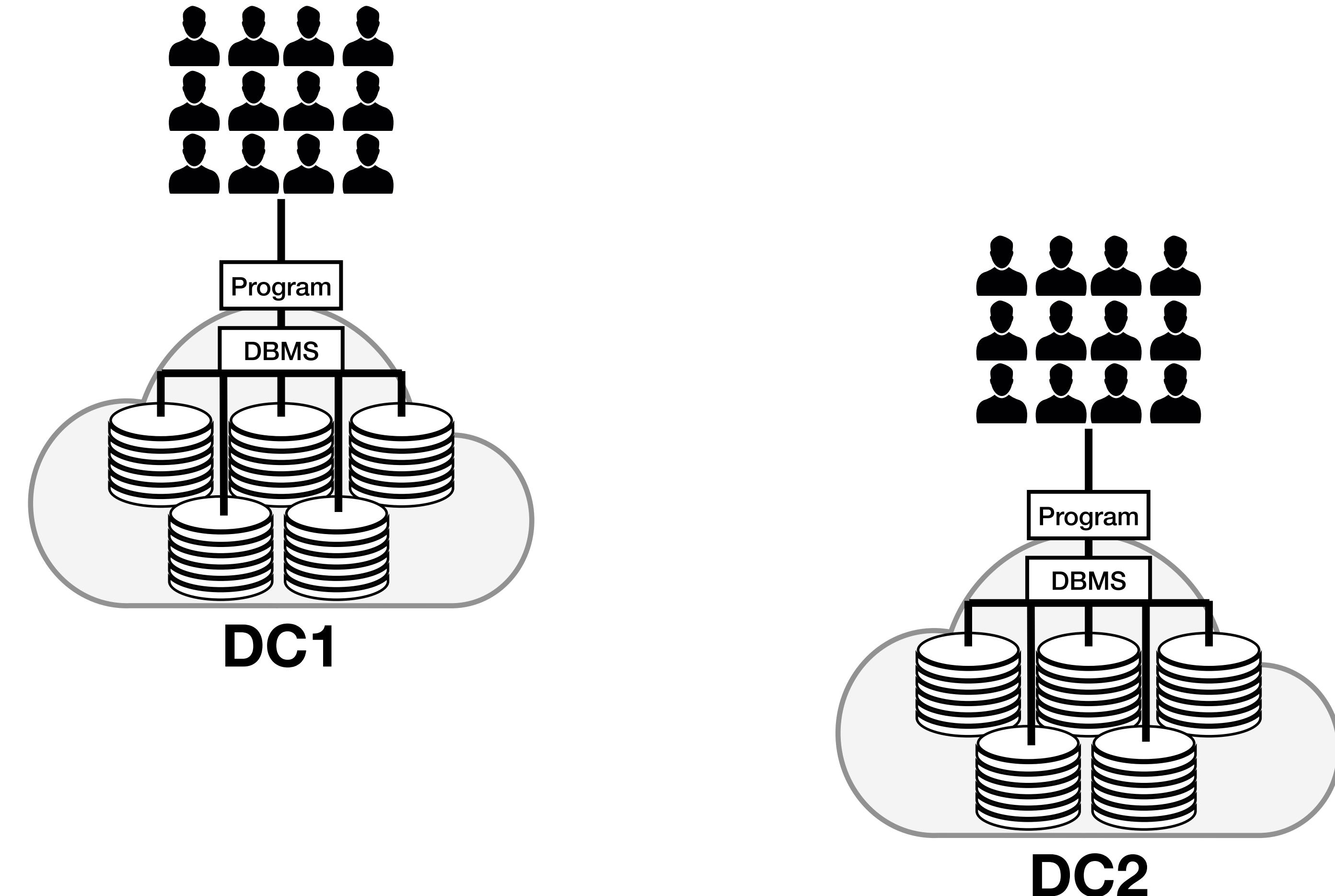
SCALING DATABASE CLUSTERS

- Clustering architectures
 - ▶ Partitioning
 - ▶ Replicating



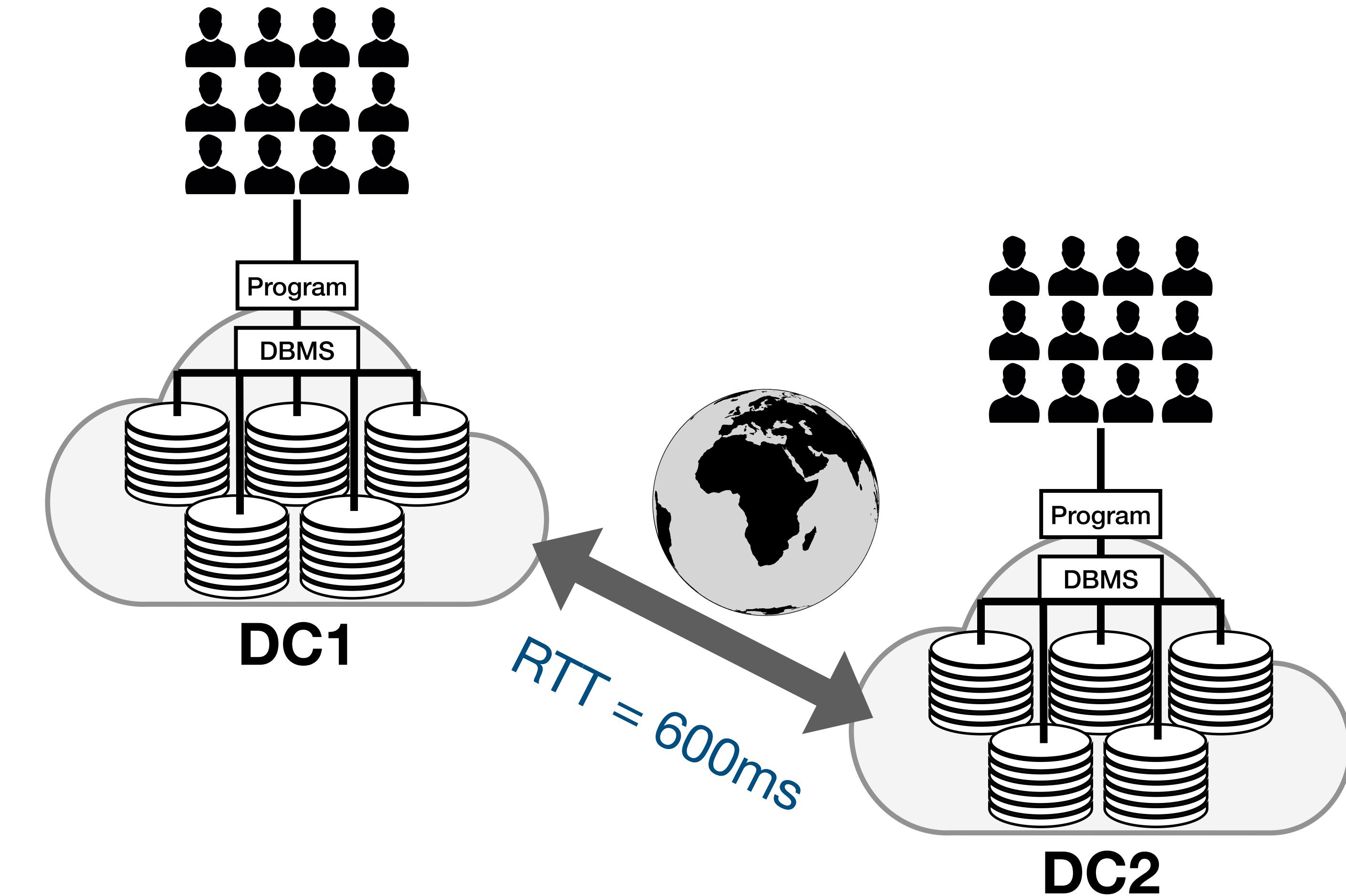
SCALING DATABASE CLUSTERS

- Clustering architectures
 - ▶ Partitioning
 - ▶ Replicating



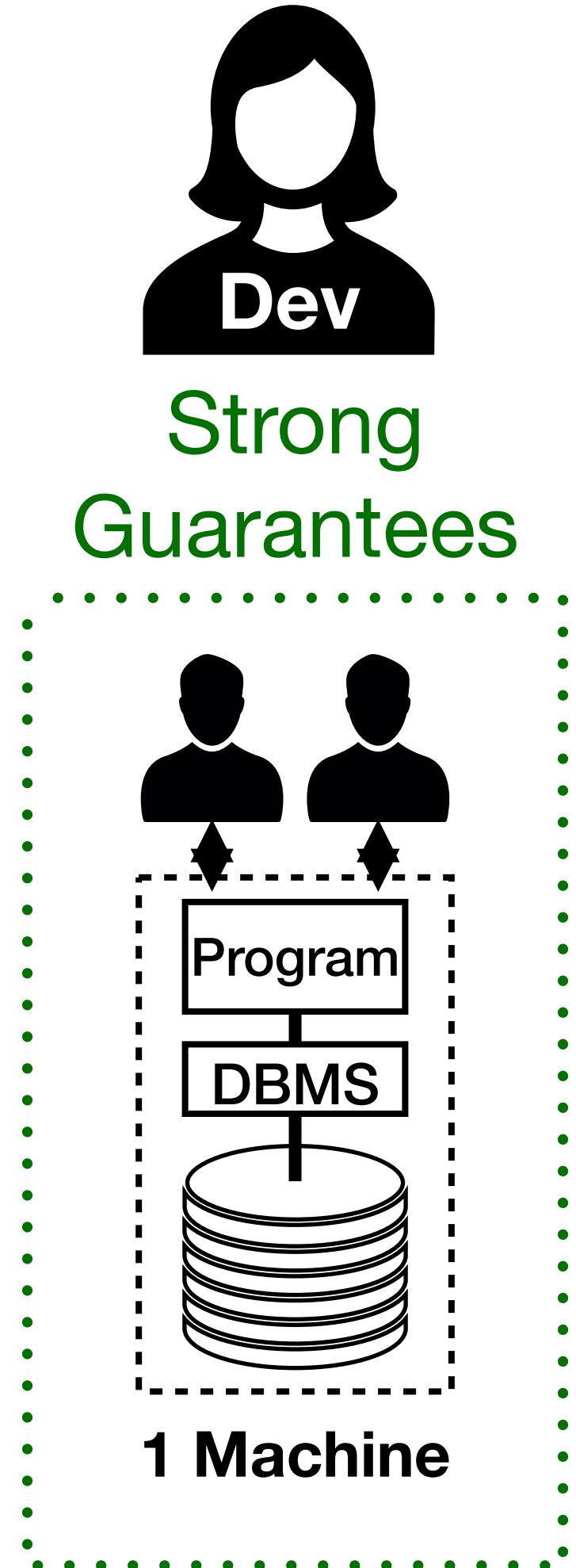
SCALING DATABASE CLUSTERS

- Clustering architectures
 - ▶ Partitioning
 - ▶ Replicating



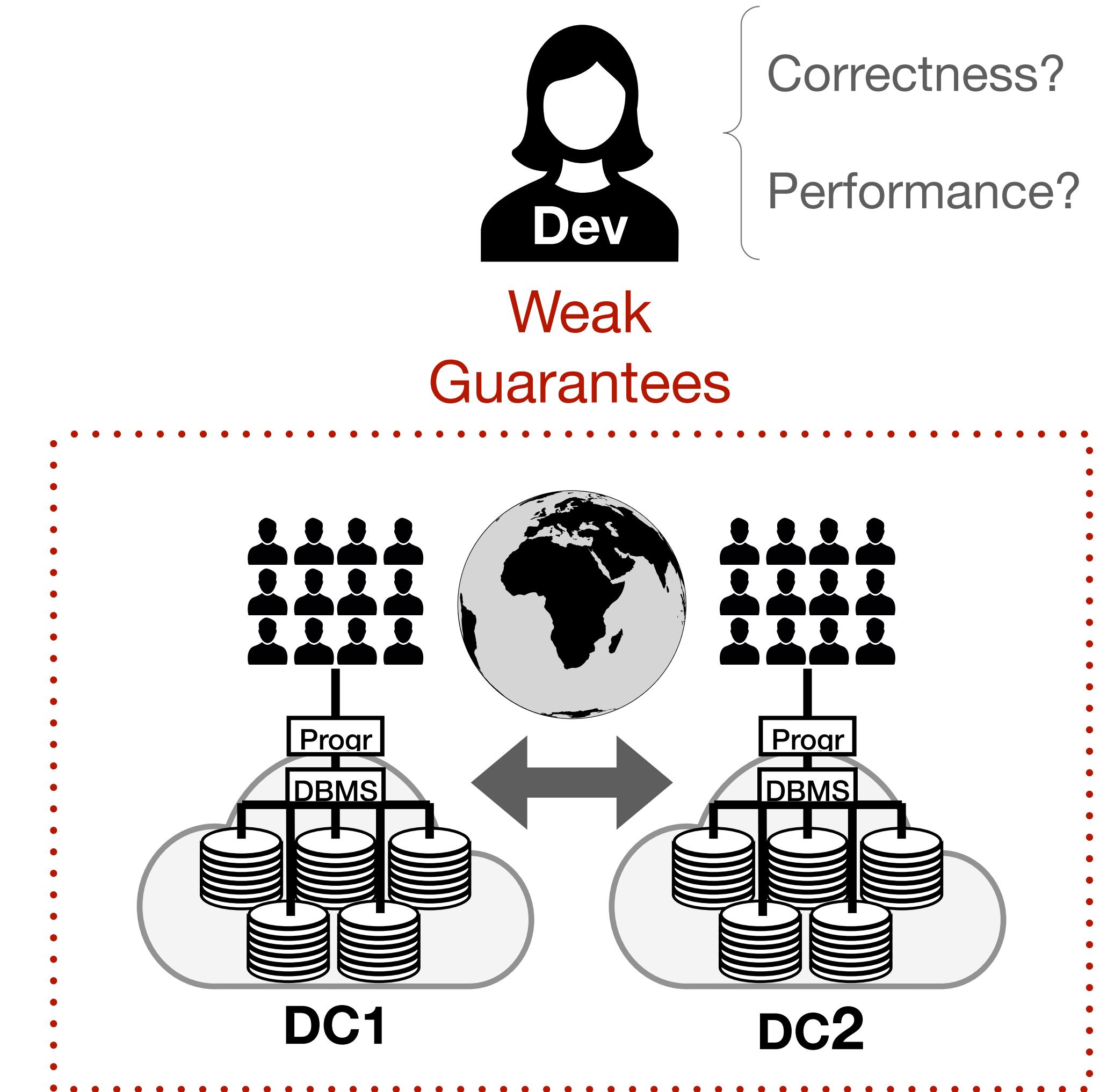
WEAK CONCURRENCY SEMANTICS

- Clustering architectures
 - ▶ Partitioning
 - ▶ Replicating
 - ▶ Weak concurrency guarantees



WEAK CONCURRENCY SEMANTICS

- 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

ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis
Stanford InfoLab

ABSTRACT

In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an *ACIDRain attack*, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation to analyze latent potential for non-serializable behavior under concurrent web API calls. We introduce a language-agnostic method for detecting potential isolation anomalies in web applications, called Abstract Anomaly Detection (2AD), that uses dynamic traces of database accesses to efficiently reason about the space of possible concurrent interleavings. We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in four languages and deployed on over 2M websites. We identify and verify 22 critical ACIDRain attacks that allow attackers to corrupt store inventory, over-spend gift cards, and steal inventory.

1. INTRODUCTION

For decades, database systems have been tasked with maintaining application integrity despite concurrent access to shared state [39]. The serializable transaction concept dictates that, if programmers correctly group their application operations into transactions, application integrity will be preserved [34]. This concept has formed the cornerstone of decades of database research and design and has led to at least one Turing award [2,40].

In practice, the picture is less clear-cut. Some databases, including Oracle's flagship offering and SAP HANA, do not offer serializability as an option at all. Other databases allow applications

to configure the database isolation level but often default to non-serializable levels [17, 19] that may corrupt application state [45]. Moreover, we are unaware of any systematic study that examines whether programmers correctly utilize transactions.

For many applications, this state of affairs is apparently satisfactory. That is, some applications do not require serializable transactions and are resilient to concurrency-related anomalies [18, 26, 48]. More prevalently, many applications do not experience concurrency-related data corruption because their typical workloads are not highly concurrent [21]. For example, for many businesses, even a few transactions per second may represent enormous sales volume.

However, the rise of the web-facing interface (i.e., API) leads

to the possibility of increased concurrency—and the deliberate ex-

plotation of concurrency-related errors. Specifically, given a public

API, a third party can programmatically trigger database-backed

behavior at a much higher rate than normal. This highly concur-

rent workload can trigger latent programming errors resulting from

incorrect transaction usage and/or incorrect use of weak isolation

levels. Subsequently, a determined adversary can systematically

exploit these errors, both to induce data corruption and induce un-

```
1 def withdraw(amt, user_id): (a)
2     bal = readBalance(user_id)
3     if (bal >= amt):
4         writeBalance(bal - amt, user_id)
```



```
1 def withdraw(amt, user_id): (b)
2     beginTxn()
3     bal = readBalance(user_id)
4     if (bal >= amt):
5         writeBalance(bal - amt, user_id)
6     commit()
```

Figure 1: (a) A simplified example of code that is vulnerable to an ACIDRain attack allowing overdraft under concurrent access. Two concurrent instances of the withdraw function could both read balance \$100, check that \$100 > \$99, and each allow \$99 to be withdrawn, resulting in \$198 total withdrawals. (b) Example of how transactions could be inserted to address this error. However, even this code is vulnerable to attack at isolation levels at or below Read Committed, unless explicit locking such as SELECT FOR UPDATE is used. While this scenario closely resembles textbook examples of improper transaction use, in this paper, we show that widely-deployed eCommerce applications are similarly vulnerable to such ACIDRain attacks, allowing corruption of application state and theft of assets.

What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

1. ACKNOWLEDGEMENTS

Most of the academic papers on concurrency control published in the last five years have assumed the following two design decisions: (1) applications execute transactions with serializable isolation and (2) applications execute most (if not all) of their transactions using stored procedures. I know this because I am guilty of writing these papers too. But results from a recent survey of database administrators indicates that these assumptions are not realistic. This survey includes both legacy deployments where the cost of changing the application to use either serializable isolation or stored procedures is not feasible, as well as new “greenfield” projects that not encumbered by prior constraints. As such, the research produced by our community is not helping people with their real-world systems and thus is essentially irrelevant.

In this talk/denunciation, I will descend from my ivory tower and argue that we need to rethink our agenda for concurrency control research. Recent trends focus on asking the wrong questions and solving the wrong problems. I contend that the real issues that will have the most impact are not easily solved by more “clever” algorithms. Instead, in many cases, they can only be solved by hardware improvements and artificial intelligence.

2. BIOGRAPHIES

Andrew Pavlo is an Assistant Professor of Databaseology in the Computer Science Department at Carnegie Mellon University. At CMU, he is a member of the Database Group and the Parallel Data Laboratory. His work is also in collaboration with the Intel Science and Technology Center for Big Data.

Permission to make digital or hard copies of part or all of this work for personal classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '17, May 14-19, 2017, Chicago, IL, USA

© 2017 Copyright held by the owner/authors. Publication rights licensed to ACM. ISBN 978-1-4503-4197-4/17/05... \$15.00

DOI: <http://dx.doi.org/10.1145/3035918.3064037>

SIGMOD '17 May 14-19, 2017, Chicago, IL, USA

© 2017 Copyright held by the owner/authors.

ACM ISBN 978-1-4503-4197-4/17/05...

DOI: <http://dx.doi.org/10.1145/3035918.3056096>

3



flexcoin
@flexcoin

Flexcoin will be shutting its doors.
Flexcoin.com

2:52 AM · Mar 4, 2014 · Twitter for iPhone

MY THESIS: THE OVERARCHING GOAL

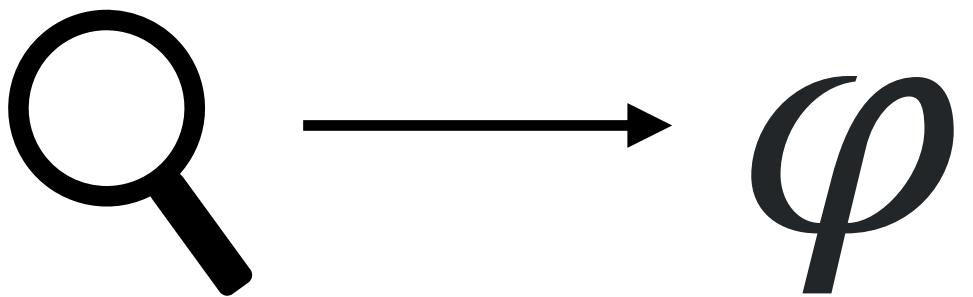
- Ease of strong semantics
- Performance of cloud-native

MY THESIS: THE OVERARCHING GOAL

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

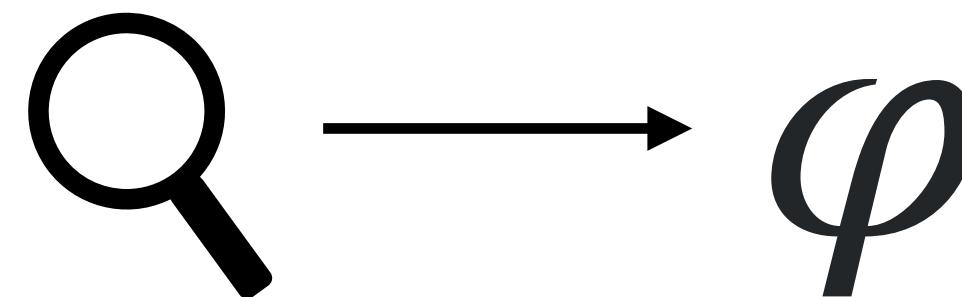
MY THESIS: THE OVERARCHING GOAL

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

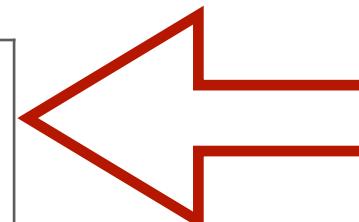


MY THESIS: THE OVERARCHING GOAL

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

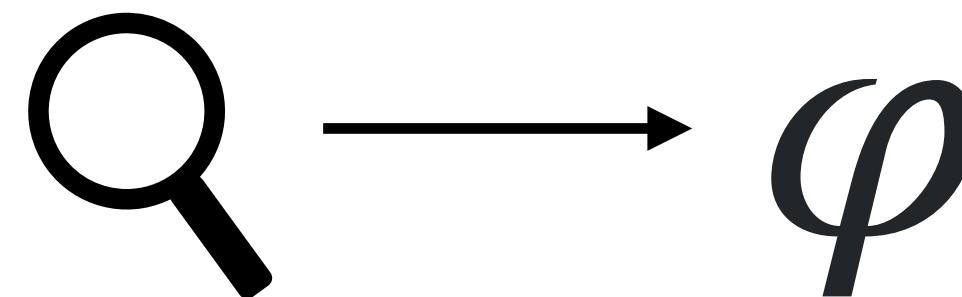


CLOTHO	Guided Testing Framework	OOPSLA 2019
ATROPOS	Automated repair of replication anomalies	PLDI 2021
LACHESIS	Automated repair of partitioning anomalies	under submission

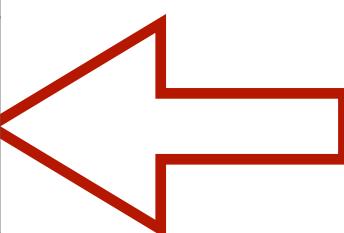


MY THESIS: THE OVERARCHING GOAL

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

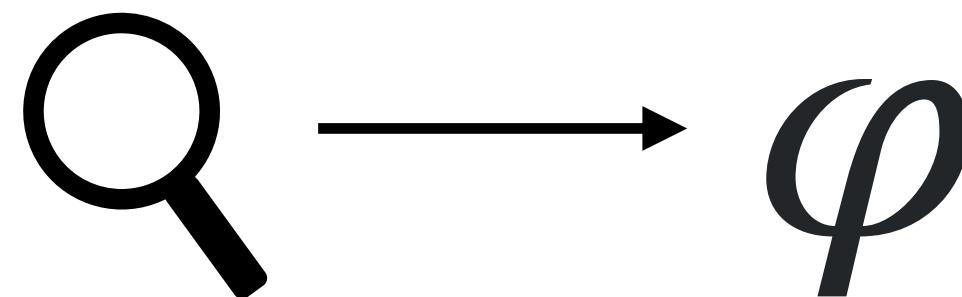


CLOTHO	Guided Testing Framework	OOPSLA 2019
ATROPOS	Automated repair of replication anomalies	PLDI 2021
LACHESIS	Automated repair of partitioning anomalies	under submission

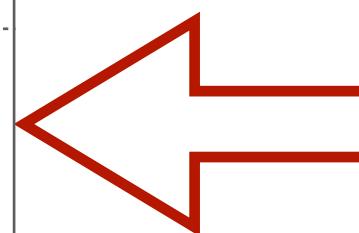


MY THESIS: THE OVERARCHING GOAL

- 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

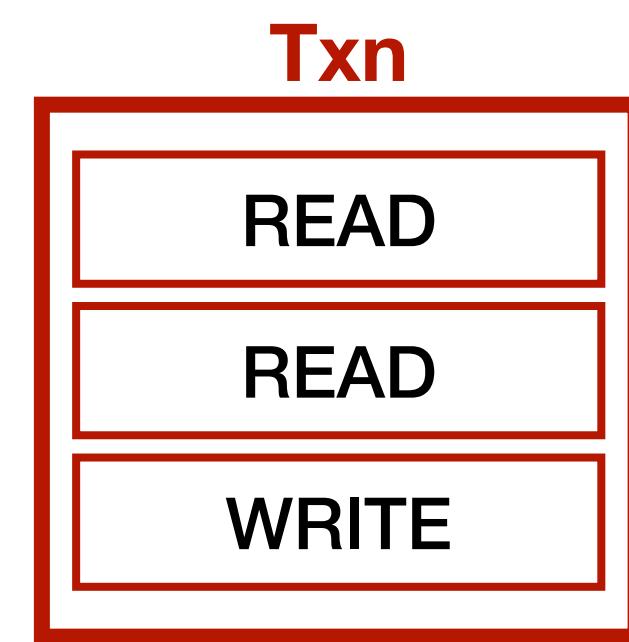


CLOTHO

(SYMBOLIC PROGRAM ANALYSIS & GUIDED TESTING)

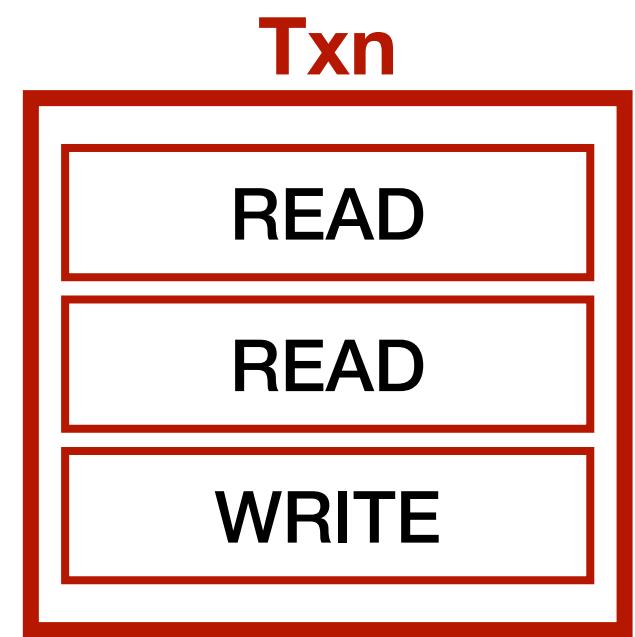
TRANSACTIONAL GUARANTEES

- ACID guarantees



TRANSACTIONAL GUARANTEES

- ACID guarantees
 - ▶ Atomicity
 - ▶ Consistency
 - ▶ Isolation
 - ▶ Durability



TRANSACTIONAL GUARANTEES

- ACID guarantees
 - ▶ Atomicity
 - ▶ Consistency
 - ▶ Isolation
 - ▶ Durability



TRANSACTIONAL GUARANTEES

- ACID guarantees
 - ▶ Atomicity
 - ▶ Consistency
 - ▶ Isolation
 - ▶ Durability



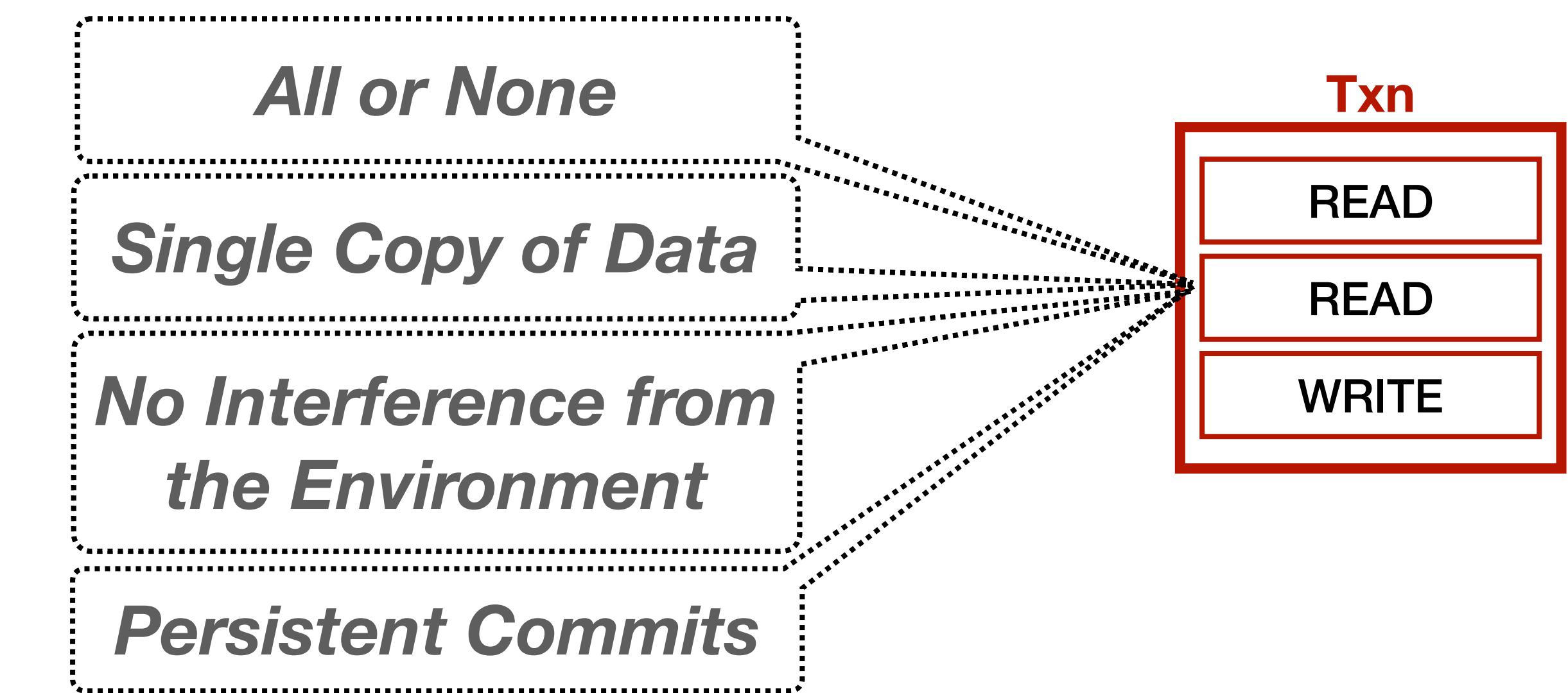
TRANSACTIONAL GUARANTEES

- ACID guarantees
 - ▶ Atomicity
 - ▶ Consistency
 - ▶ Isolation
 - ▶ Durability



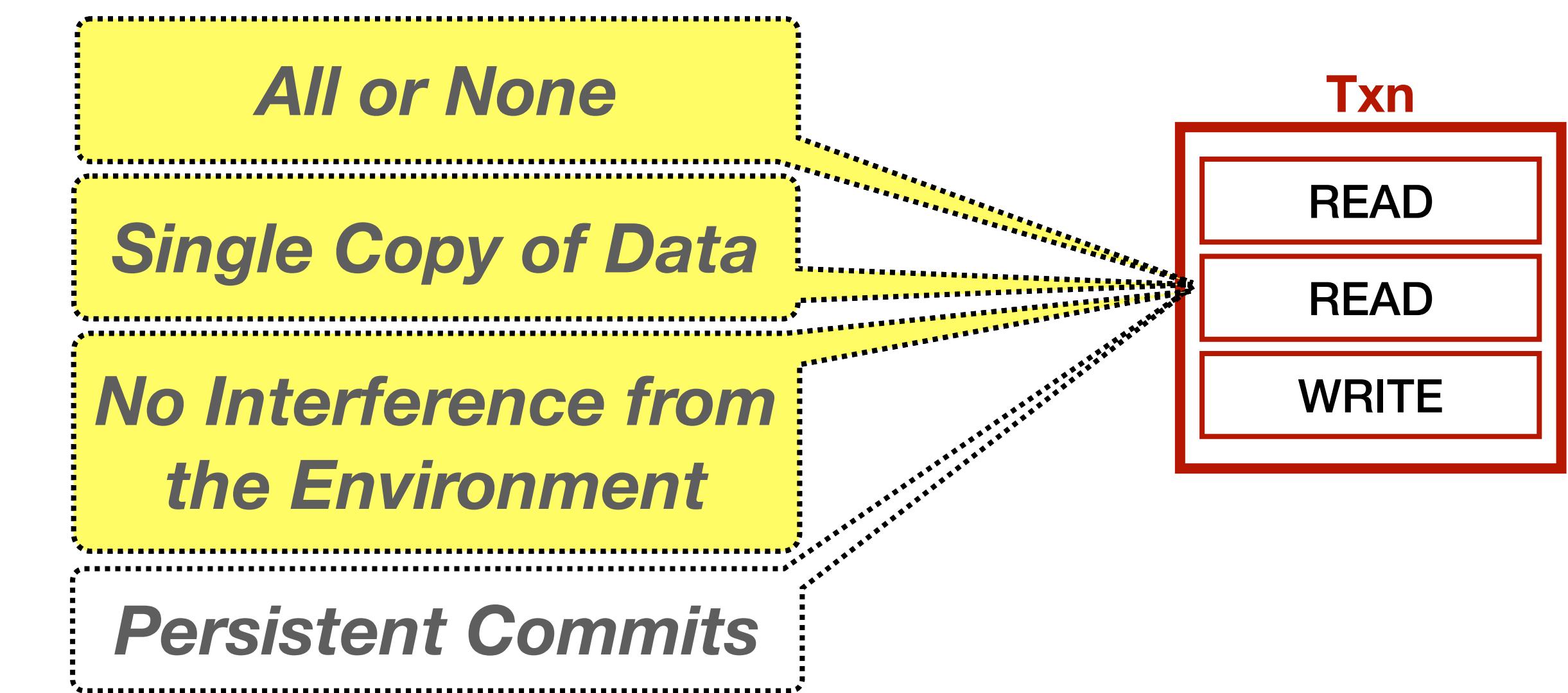
TRANSACTIONAL GUARANTEES

- ACID guarantees
 - ▶ Atomicity
 - ▶ Consistency
 - ▶ Isolation
 - ▶ Durability



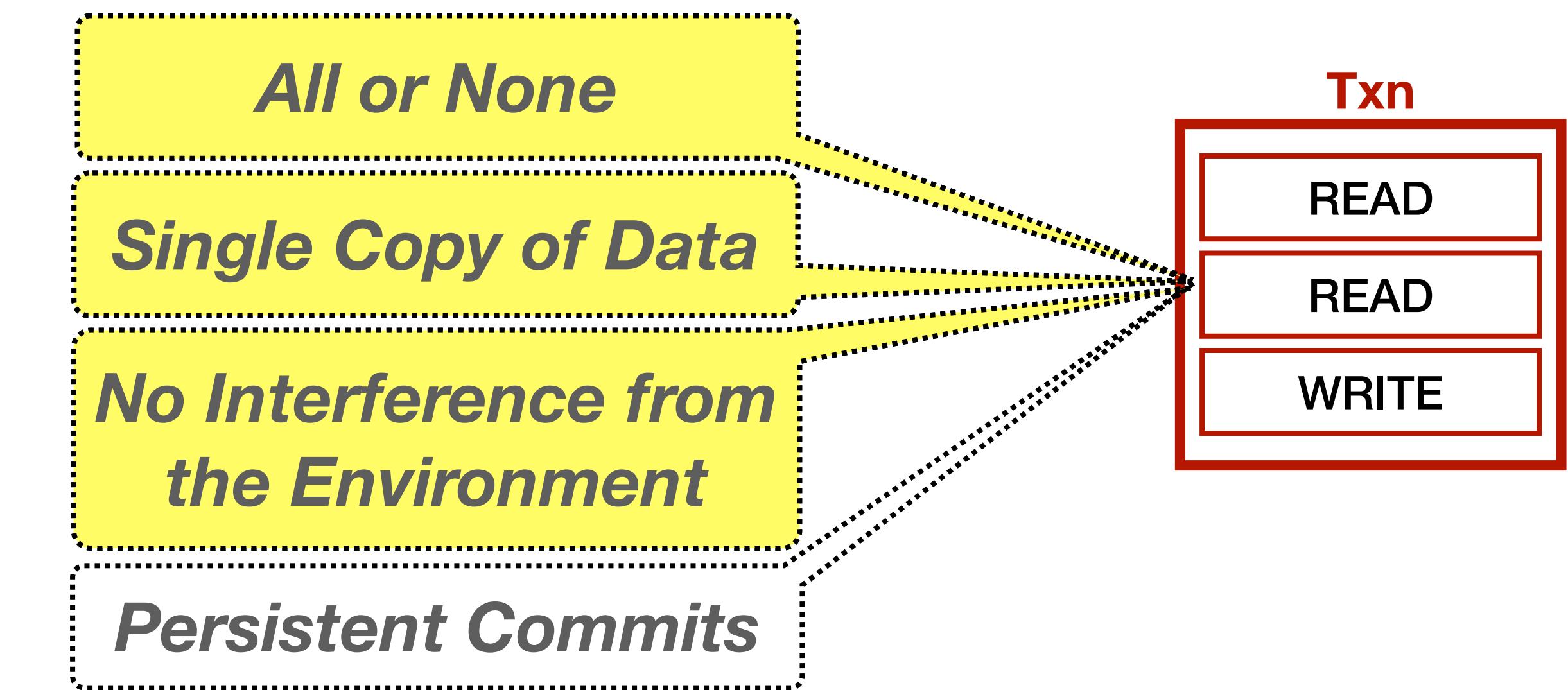
TRANSACTIONAL GUARANTEES

- ACID guarantees
 - ▶ Atomicity
 - ▶ Consistency
 - ▶ Isolation
 - ▶ Durability
- Serializability



TRANSACTIONAL GUARANTEES

- ACID guarantees
 - ▶ Atomicity
 - ▶ Consistency
 - ▶ Isolation
 - ▶ Durability
- Serializability
- Facilitates program design and reasoning

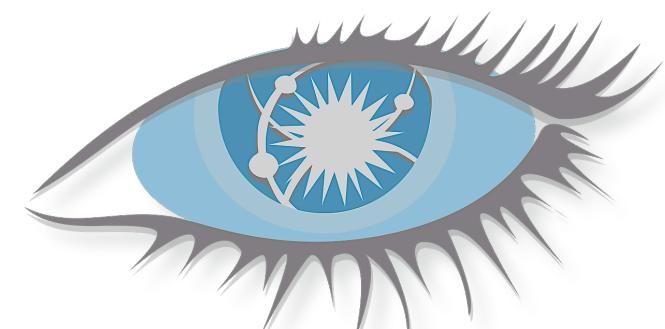


SERIALIZABILITY

- 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
------	--------	--------	--------

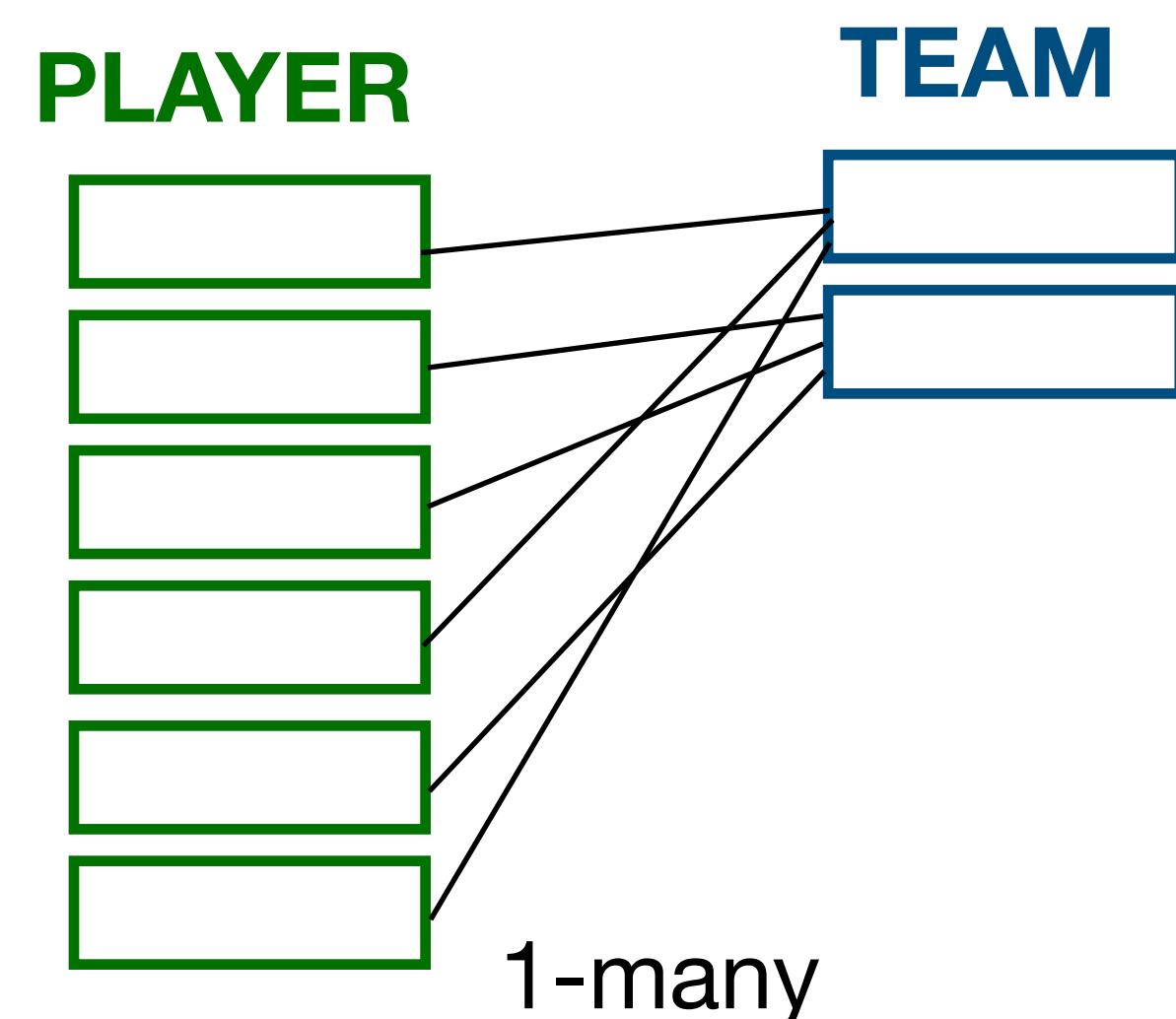
TEAM

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_cnt
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

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
```

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
```

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)

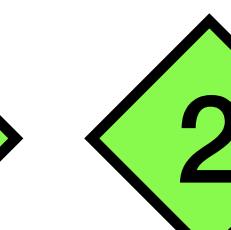
Select p_cnt // 2



1

...

Update p_cnt = 3

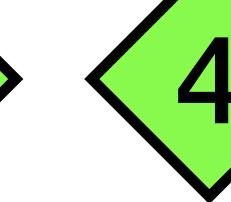


2

Select p_cnt // 2

...

Update p_cnt = 3



3

...



4

EXAMPLE (ANOMALY)

TEAM

t_id	t_name	p_cnt
T1	"A"	2

addPlayer(p=P0, t=T1)

Select p_cnt // 2



addPlayer(p=P0, t=T1)

Select p_cnt // 2

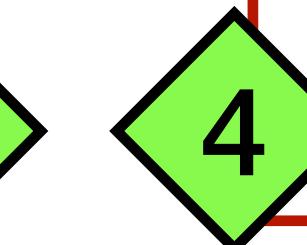


...

Update p_cnt = 3



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

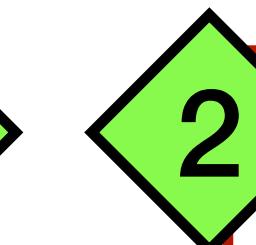
addPlayer(p=P0, t=T1)

Select p_cnt // 2



addPlayer(p=P0, t=T1)

Select p_cnt // 2

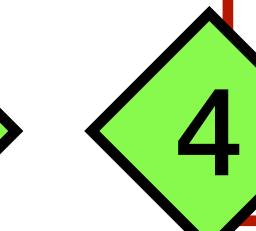


...

Update p_cnt = 3



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)

addPlayer(p=P0, t=T1)

Select p_cnt // 2

1

2

Select p_cnt // 2

...

Update p_cnt = 3

3

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)

addPlayer(p=P0, t=T1)

Select p_cnt // 2

1

2

Select p_cnt // 2

...

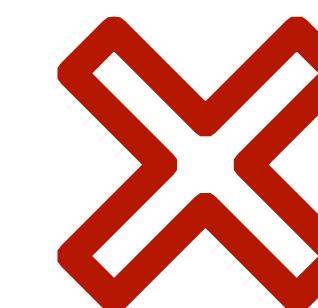
Update p_cnt = 3

3

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

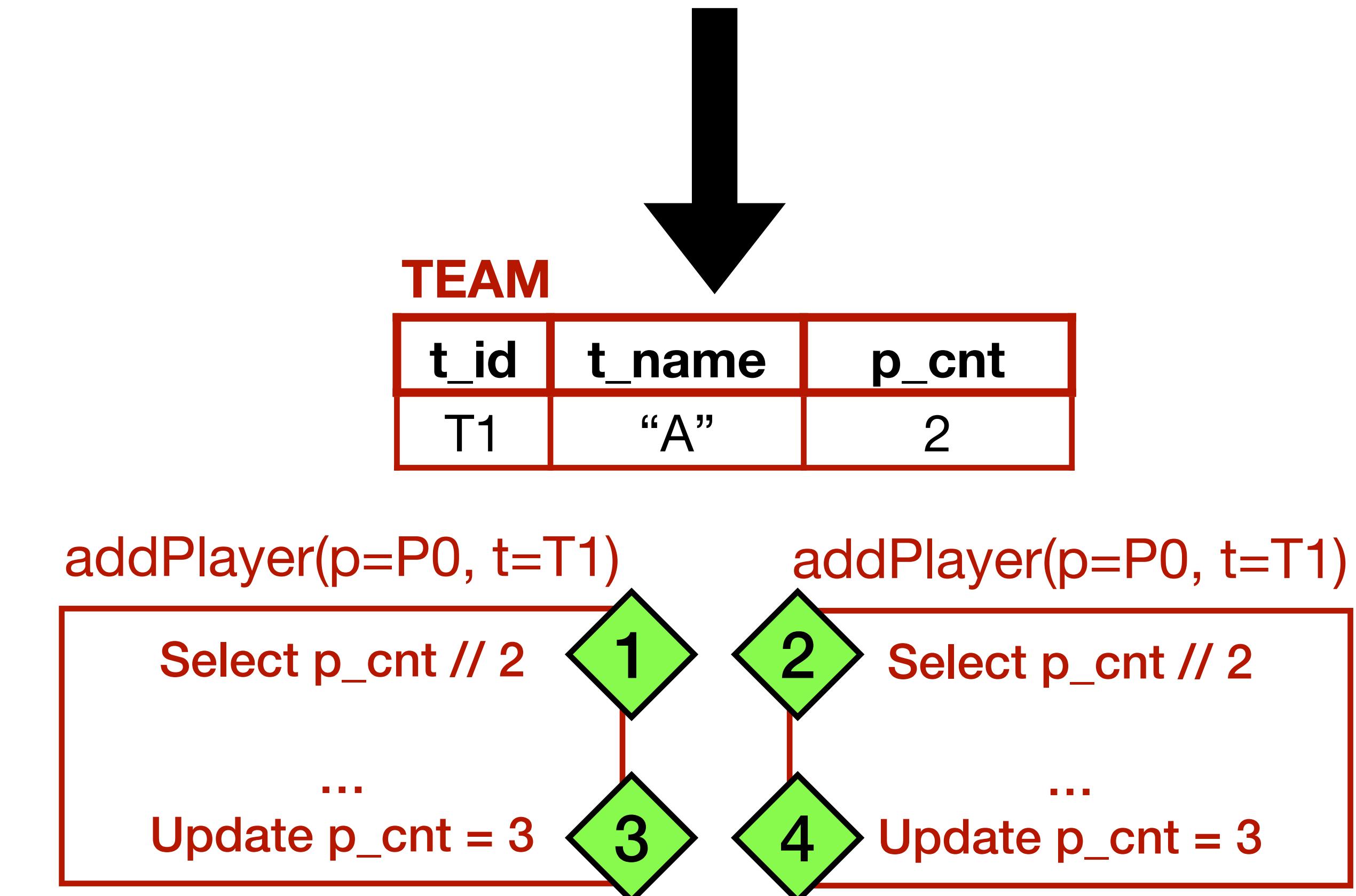
Lost Update

CHALLENGES OF TESTING

- Serializability anomalies are subtle

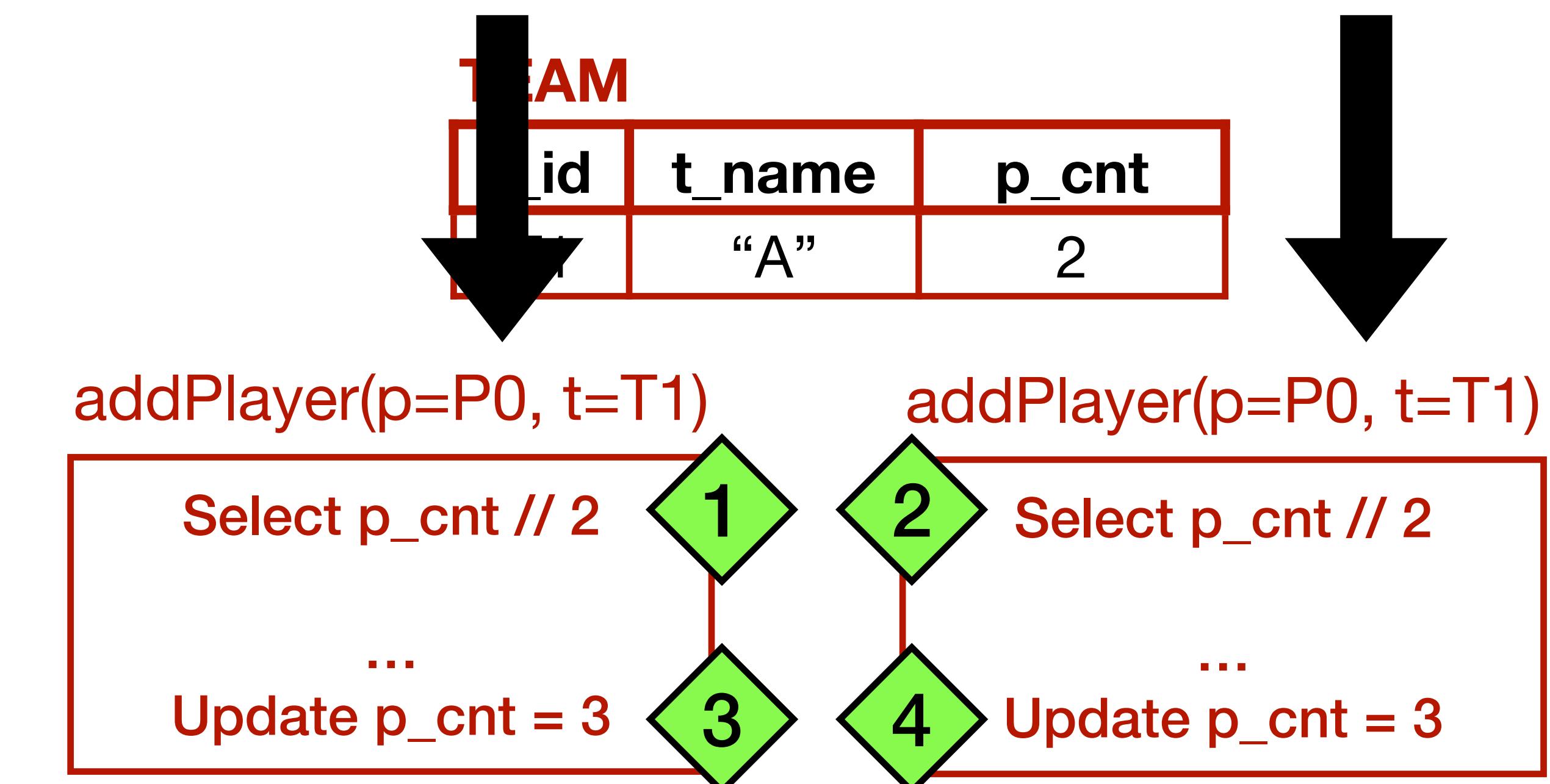
CHALLENGES OF TESTING

- Serializability anomalies are subtle
 - ▶ initial state



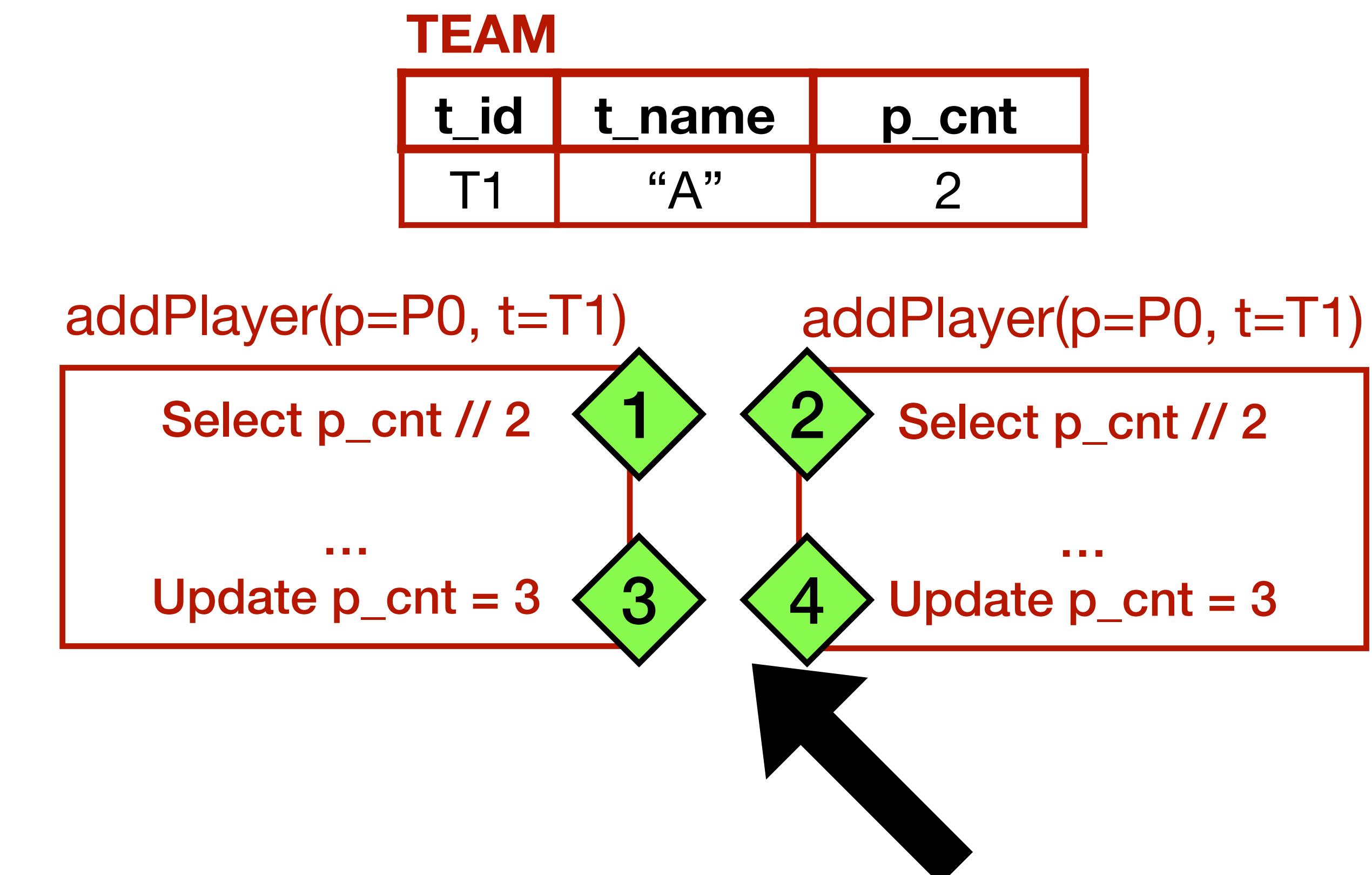
CHALLENGES OF TESTING

- Serializability anomalies are subtle
 - ▶ initial state
 - ▶ transaction arguments



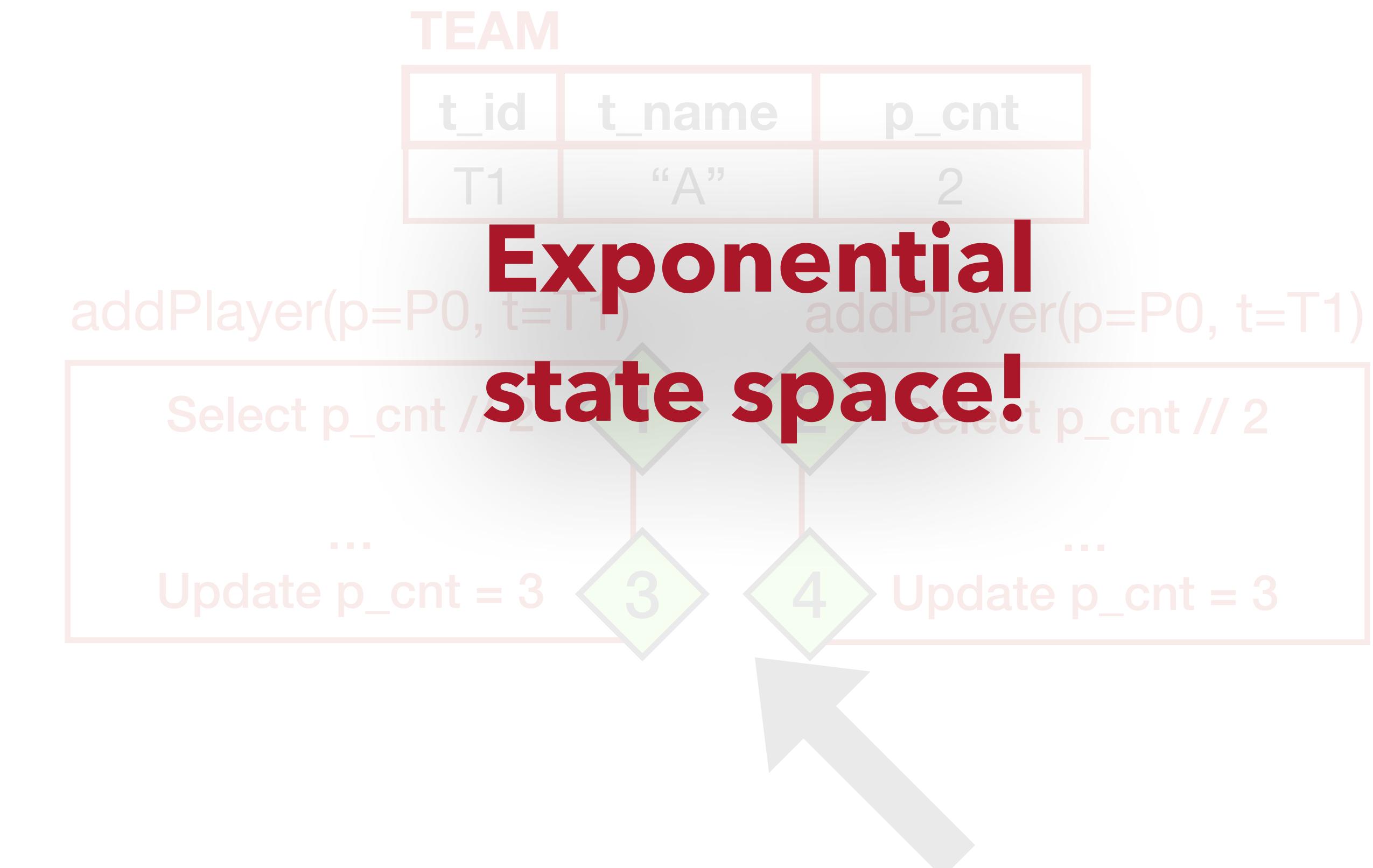
CHALLENGES OF TESTING

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

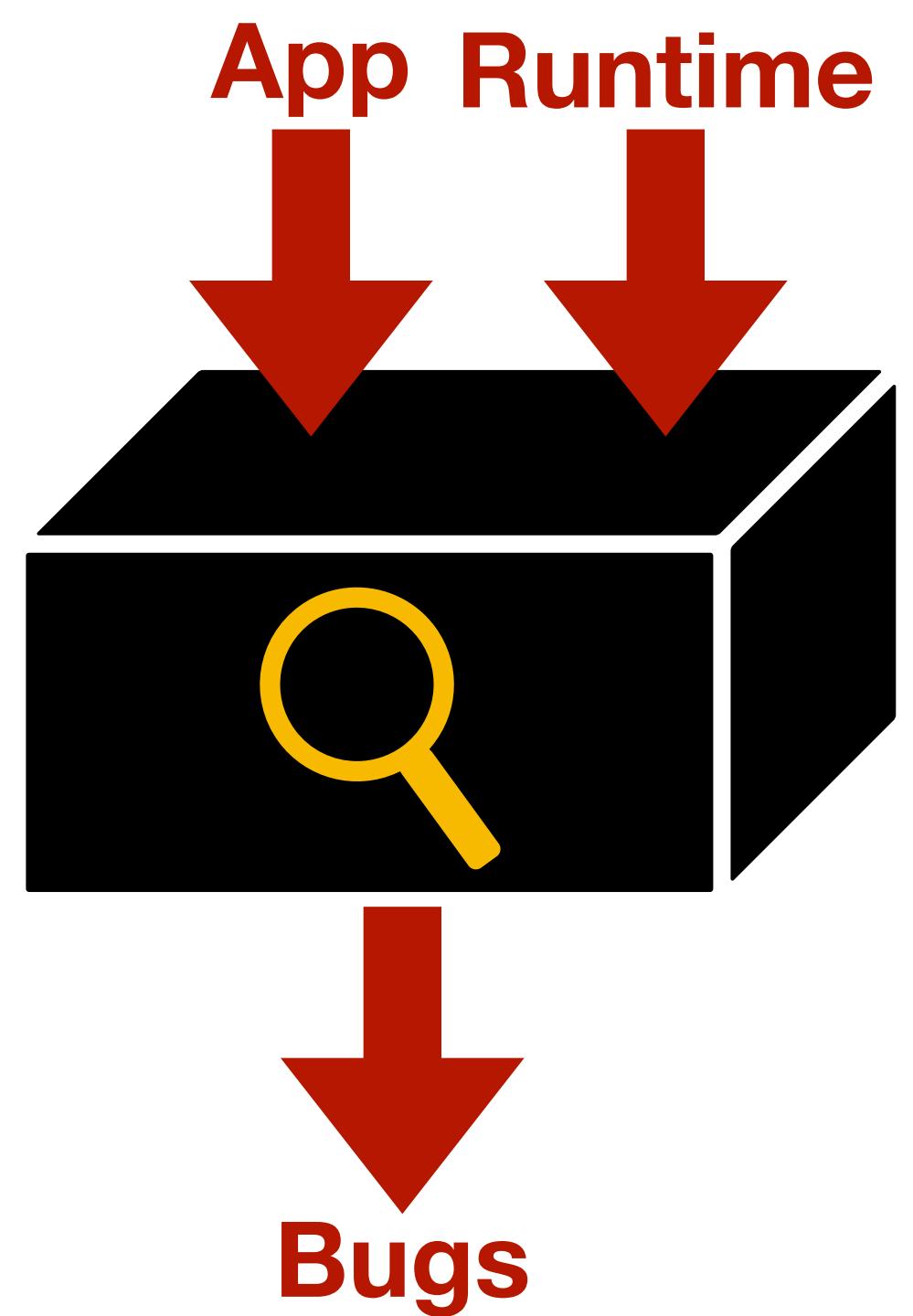


CHALLENGES OF TESTING

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

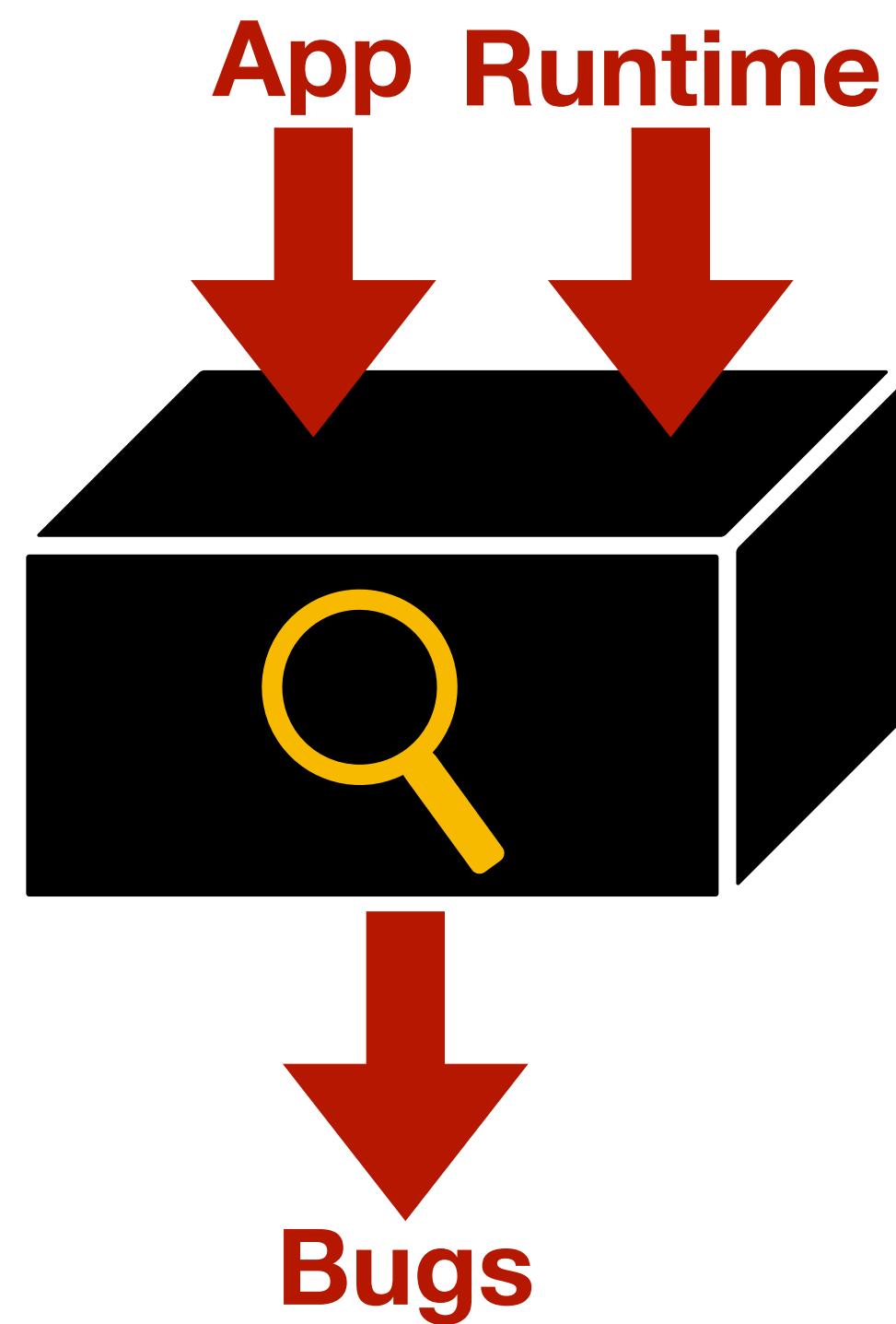


BLACKBOX TESTING



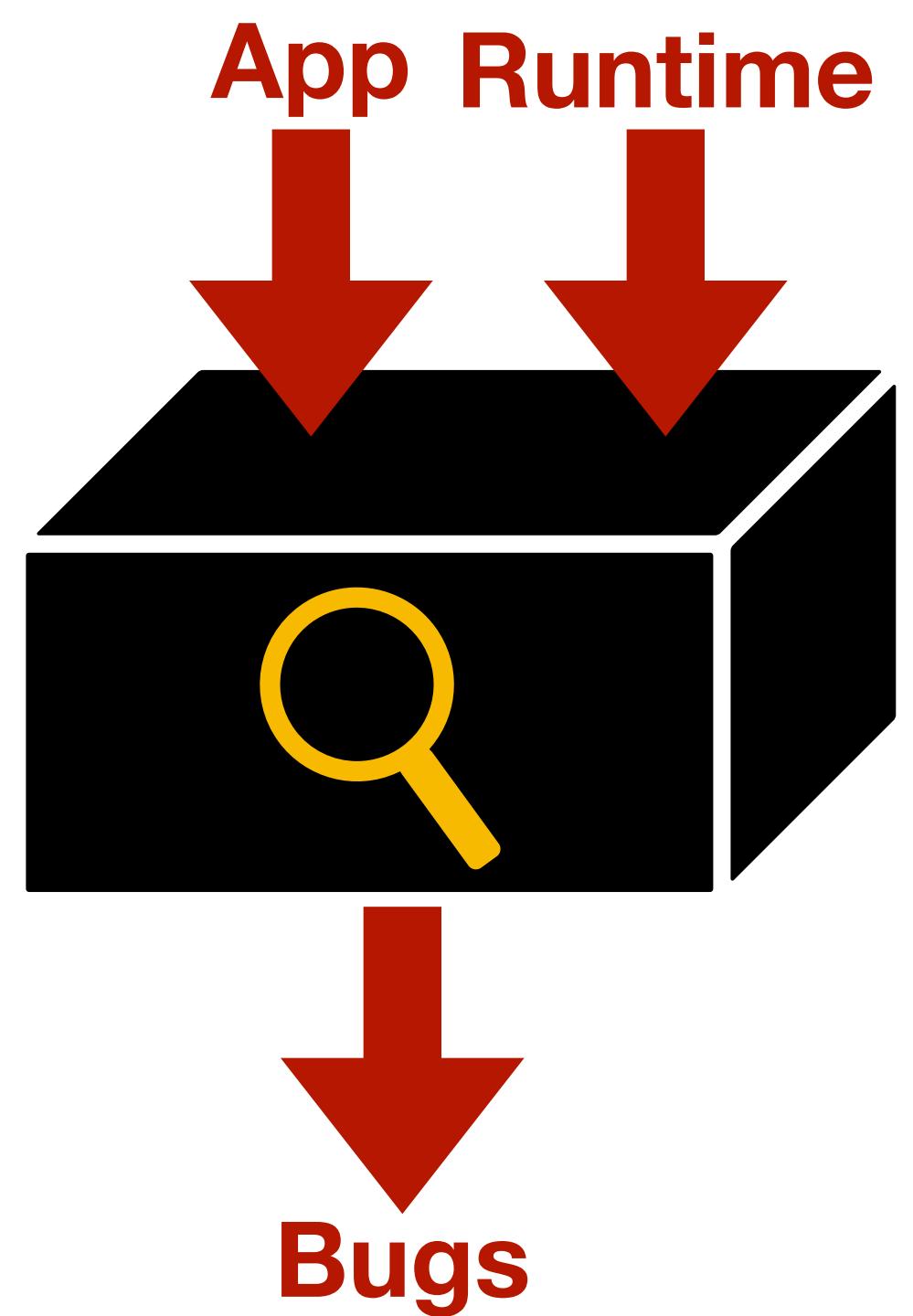
BLACKBOX TESTING

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



BLACKBOX TESTING

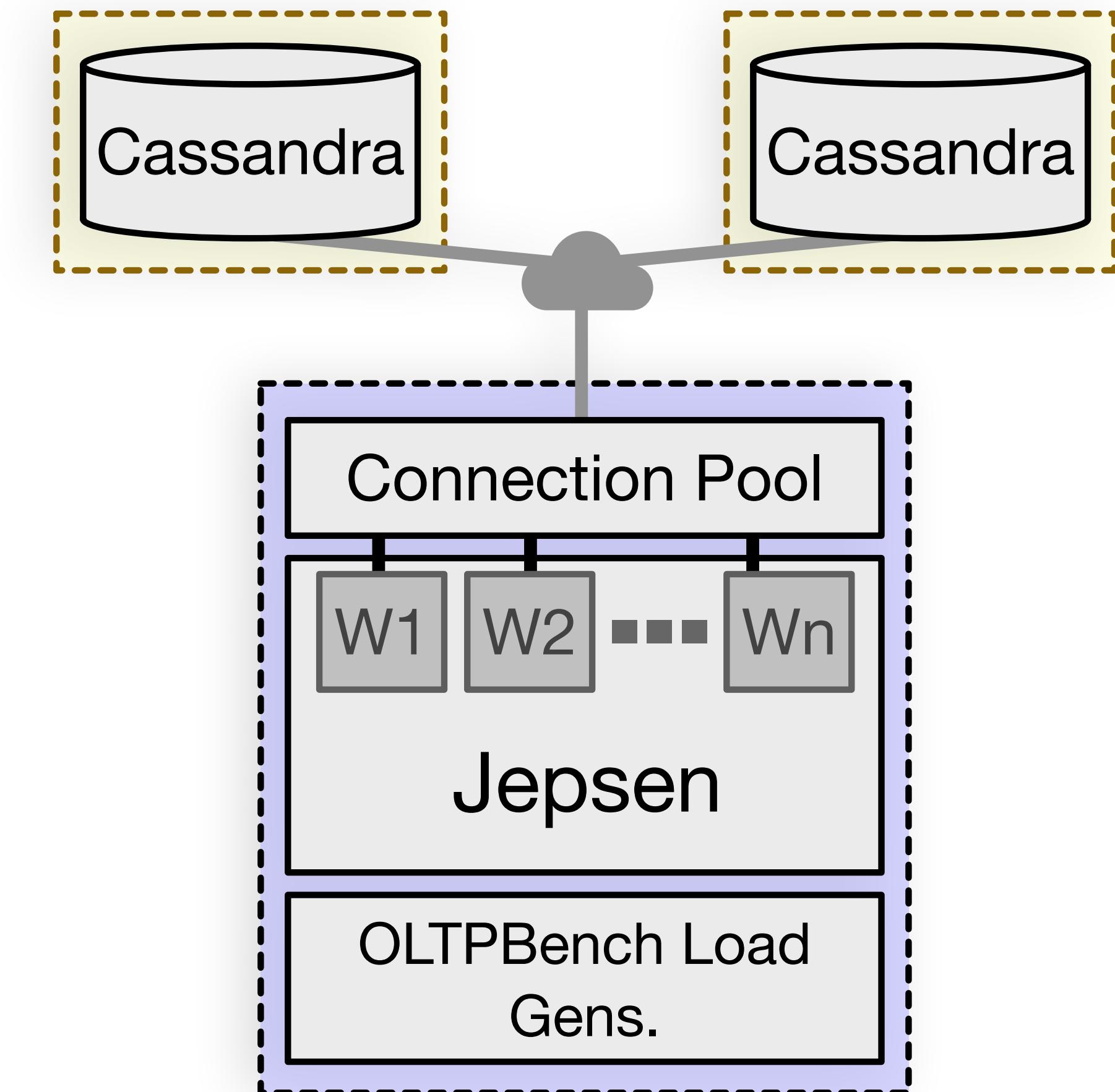
- Run and monitor apps in (semi-) production environment
 - + real bugs (no false positive)
- costly
- too specific
- no guarantee of coverage
- manual effort



BLACKBOX TESTING: STATE OF THE ART

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

PAYMENT	STOCK_LEVEL
ORDER_STATUS	NEW_ORDER
DELIVERY	



BLACKBOX TESTING: STATE OF THE ART

- 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+
▶ 2+
▶ On

Invariant
CR1
CR2
CR3
CR4
CR5A
CR5B

— KEY IDEA —

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

CR12
NCR1
NCR2
NCR3
NCR4
NCR5
NCR6
NCR7

WHITE BOX ANALYSIS

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

WHITE BOX ANALYSIS

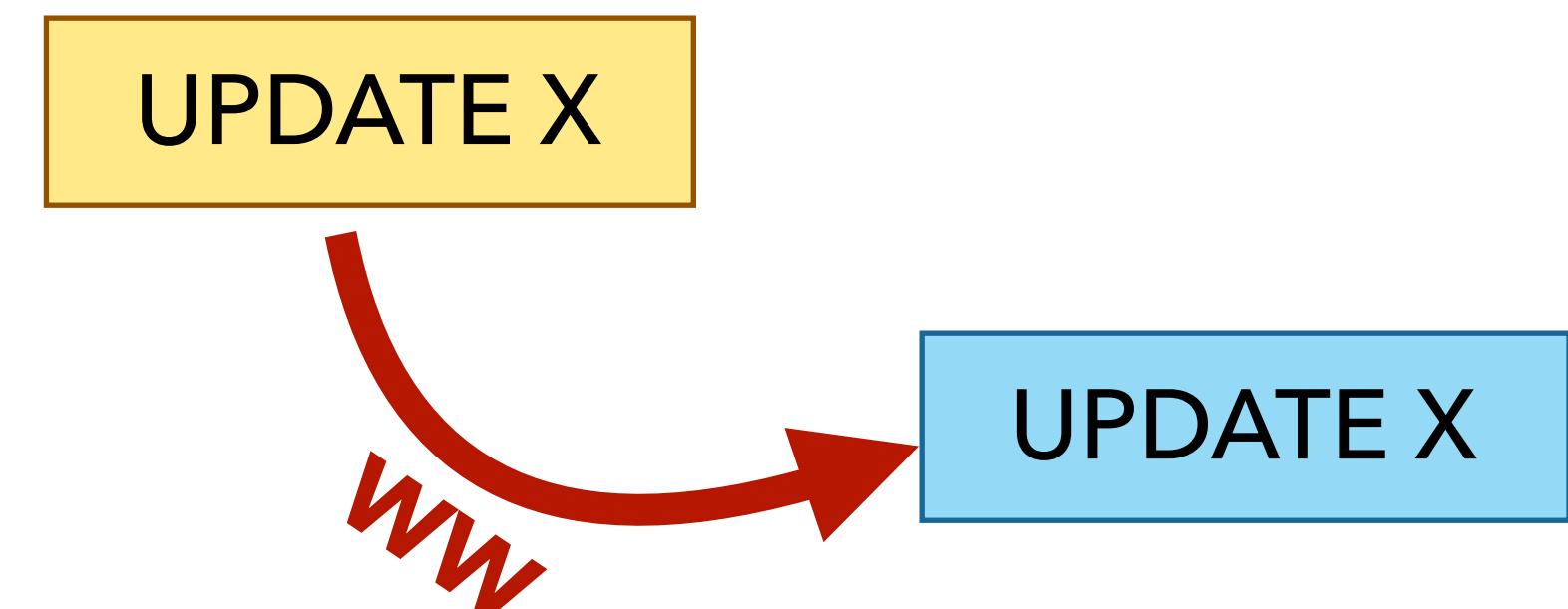
- Non-serializable executions in a program, iff
 - ▶ Cyclic dependencies between txns
- Executions abstracted by directed dependency graphs

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

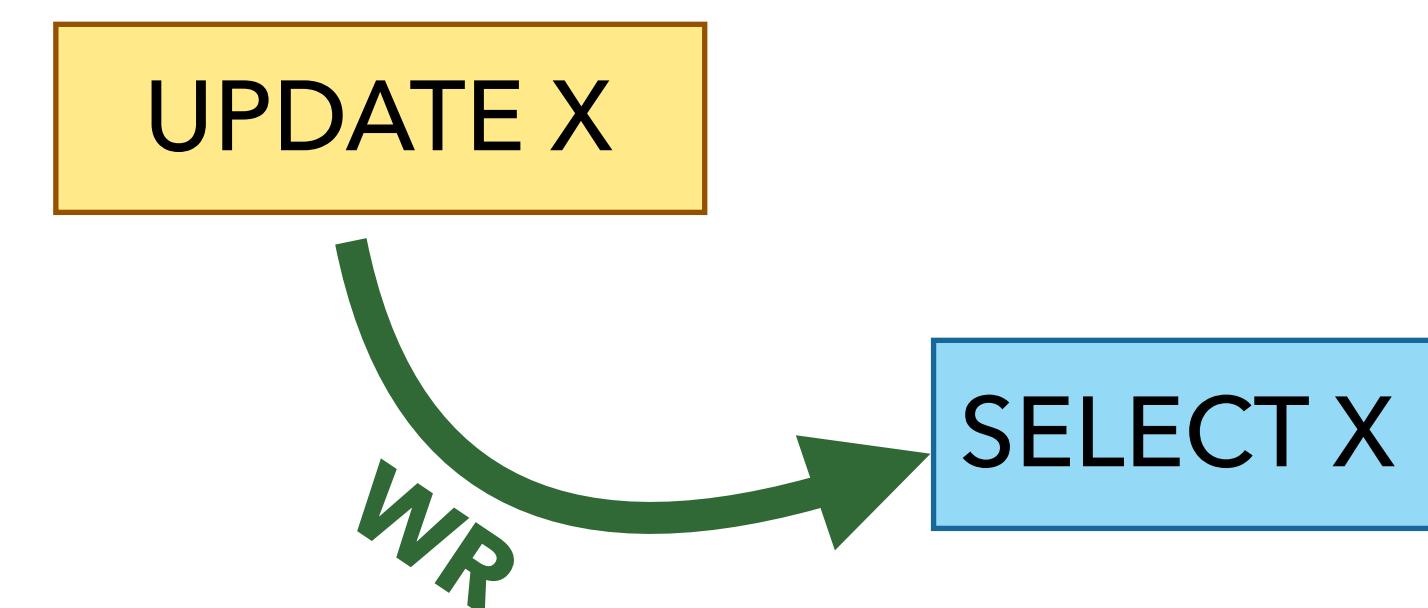
WHITE BOX ANALYSIS

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



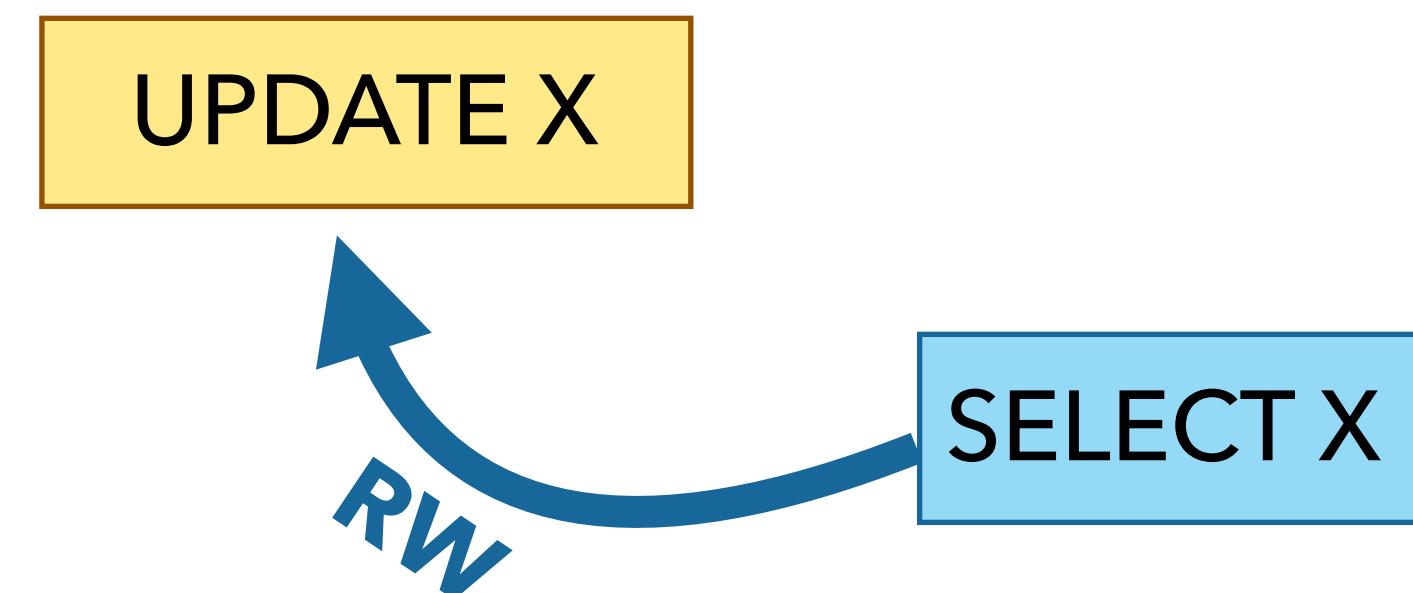
WHITE BOX ANALYSIS

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



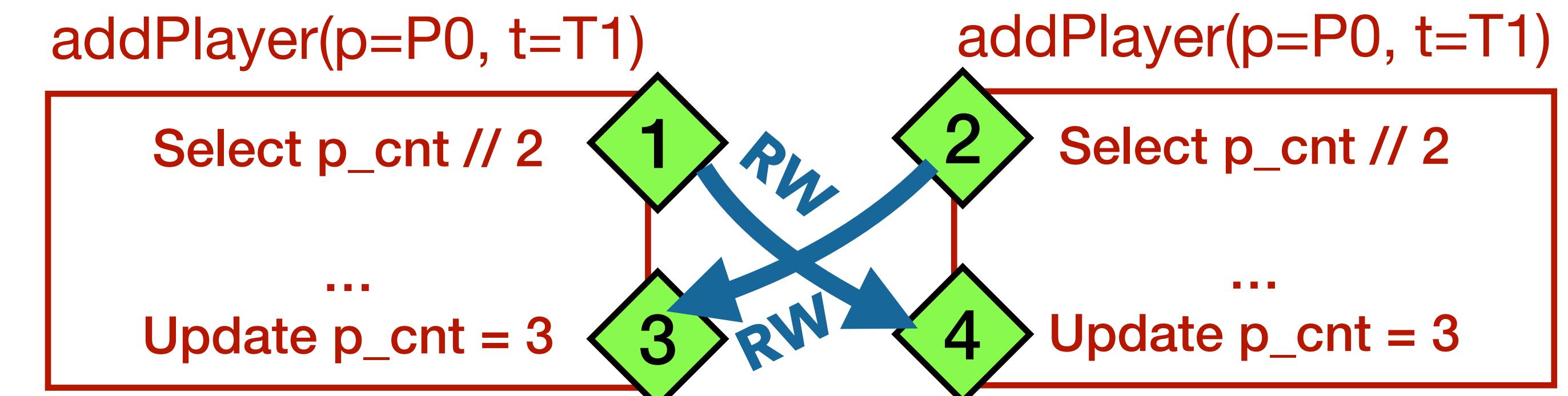
WHITE BOX ANALYSIS

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



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
 - ▶ write dependency (**WW**)
 - ▶ read dependency (**WR**)
 - ▶ read anti-dependency (**RW**)
- Example: lost update



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

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

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

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

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

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

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

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

conditions satisfied by any execution of any program

Necessary conditions to establish a dependency relation

database-specific consistency constraints

Sufficient conditions to establish a dependency relation

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

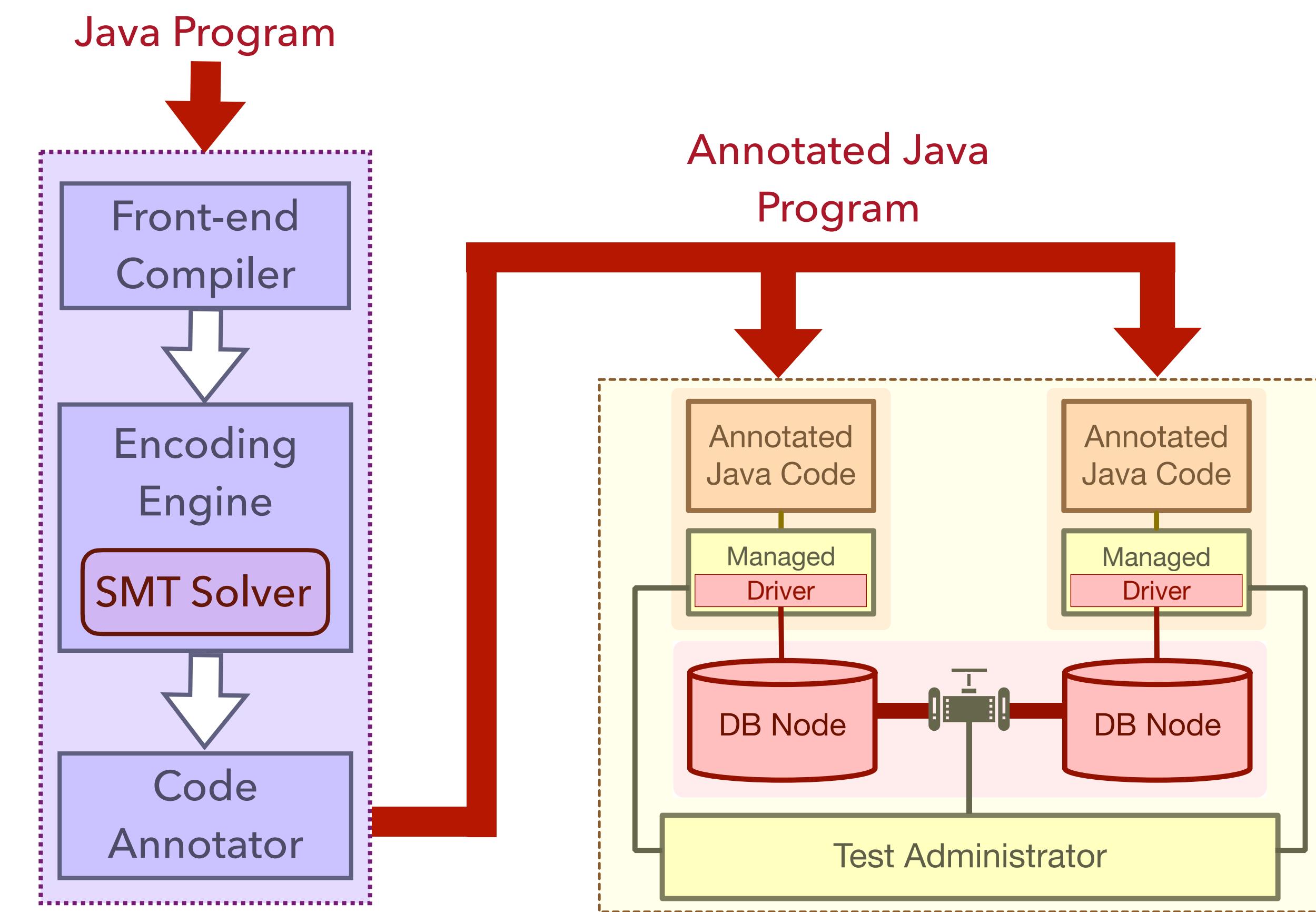
conditions satisfied by any
execution of any program

Necessary conditions to
establish a dependency relation

Enforces the existence of a
dependency cycle

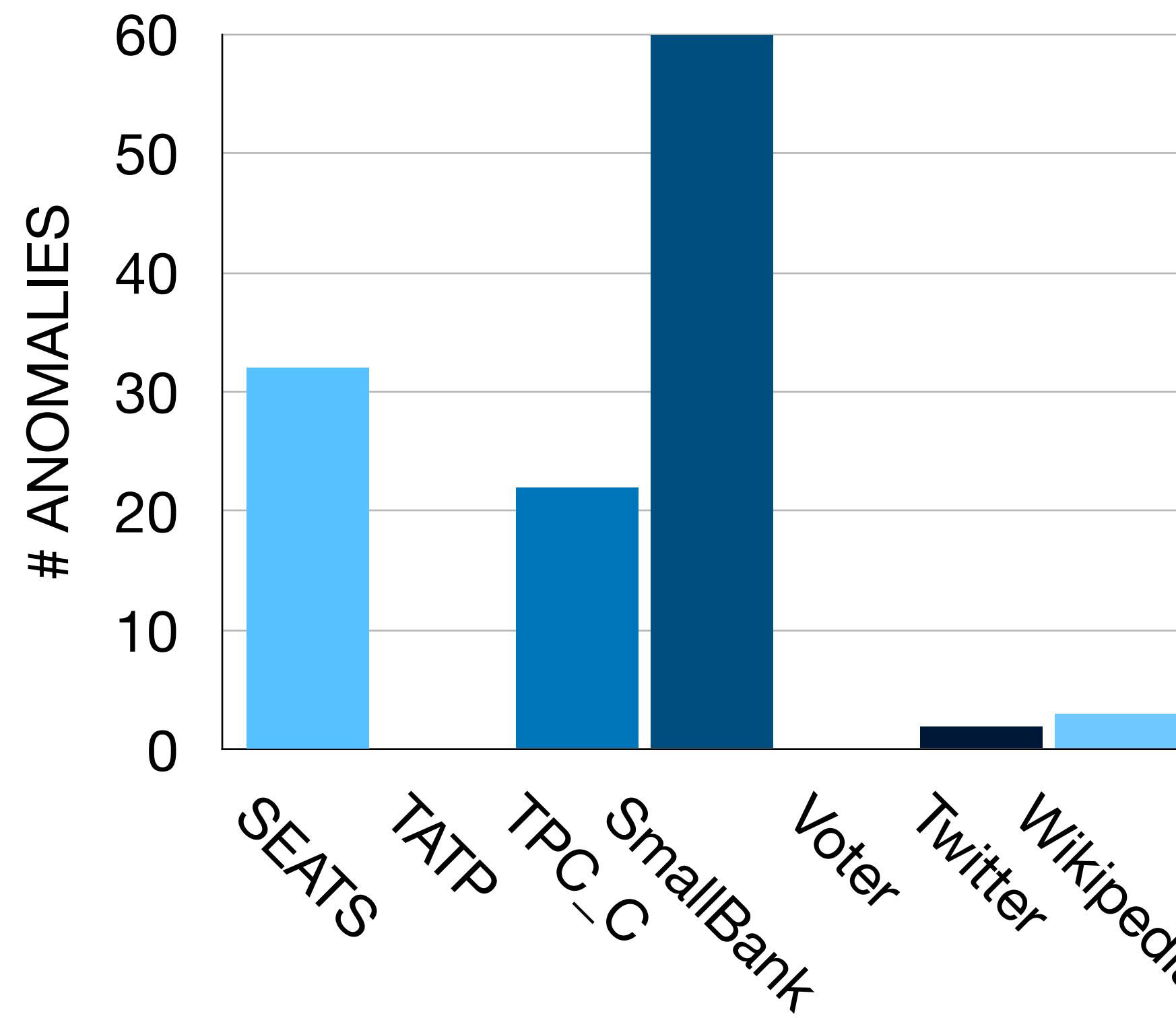
CLOTHO: DIRECTED TEST GENERATION AND REPLAY

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



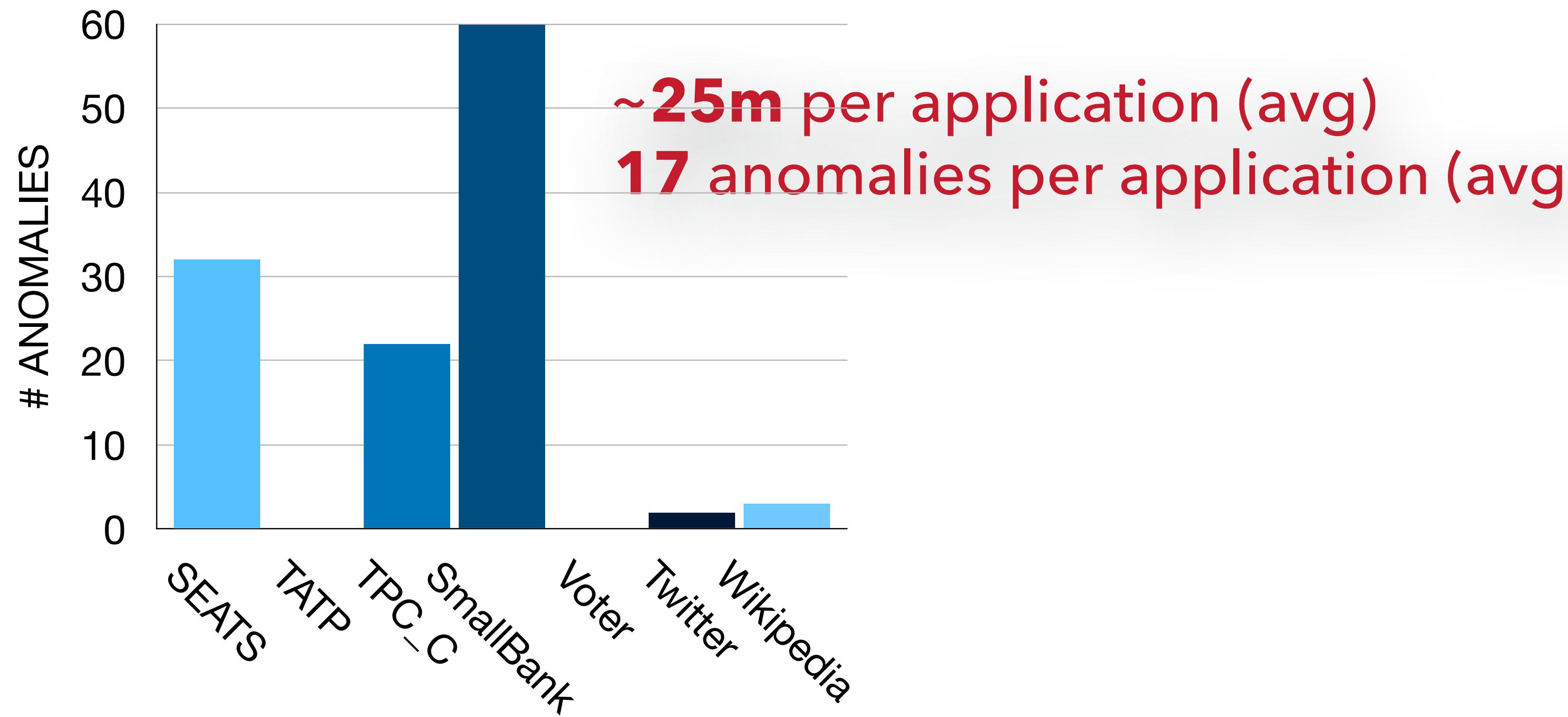
EXPERIMENTAL RESULTS

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



EXPERIMENTAL RESULTS

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



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)

REPLICATION ANOMALIES

- Serializability anomaly

addPlayer(p, t)

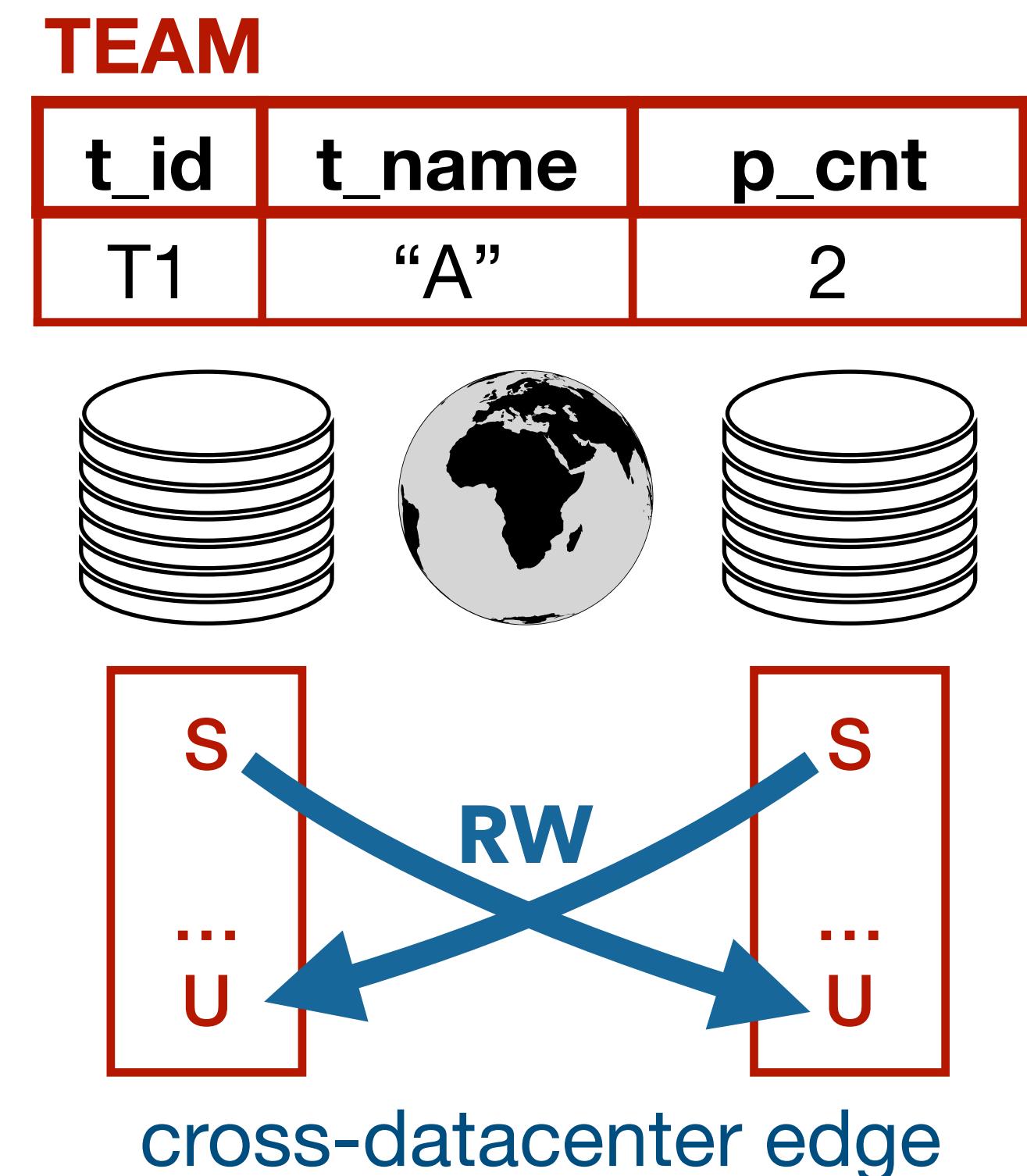
```
old_cnt := select p_cnt from TEAM where t_id=t
Insert (p,t,∅,∅) into PLAYER
update TEAM set p_cnt=old_cnt+1 where t_id=t
```

REPLICATION ANOMALIES

- Serializability anomaly

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
```



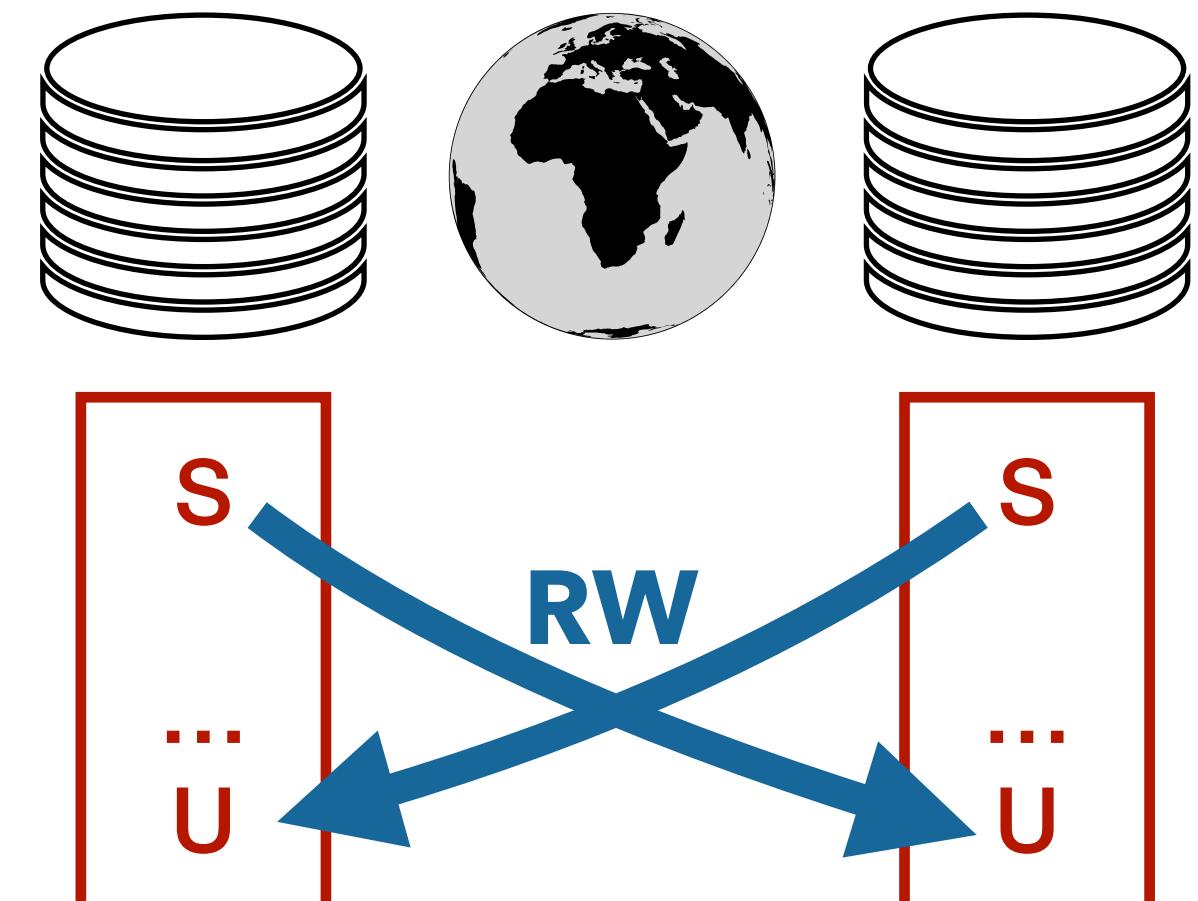
REPLICATION ANOMALIES

- Serializability anomaly
- 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



REFACTORING FOR ELIMINATING ANOMALIES

- 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,  $\emptyset$ ,  $\emptyset$ ) into PLAYER  
    update TEAM set p_cnt=old_cnt+1 where t_id=t  
}
```

PLAYER

<code>p_id</code>	<code>p_t_id</code>	<code>p_role</code>	<code>p_stat</code>
-------------------	---------------------	---------------------	---------------------

TEAM

<code>t_id</code>	<code>t_name</code>	<code>p_cnt</code>
-------------------	---------------------	--------------------

`addPlayer(p, t)`



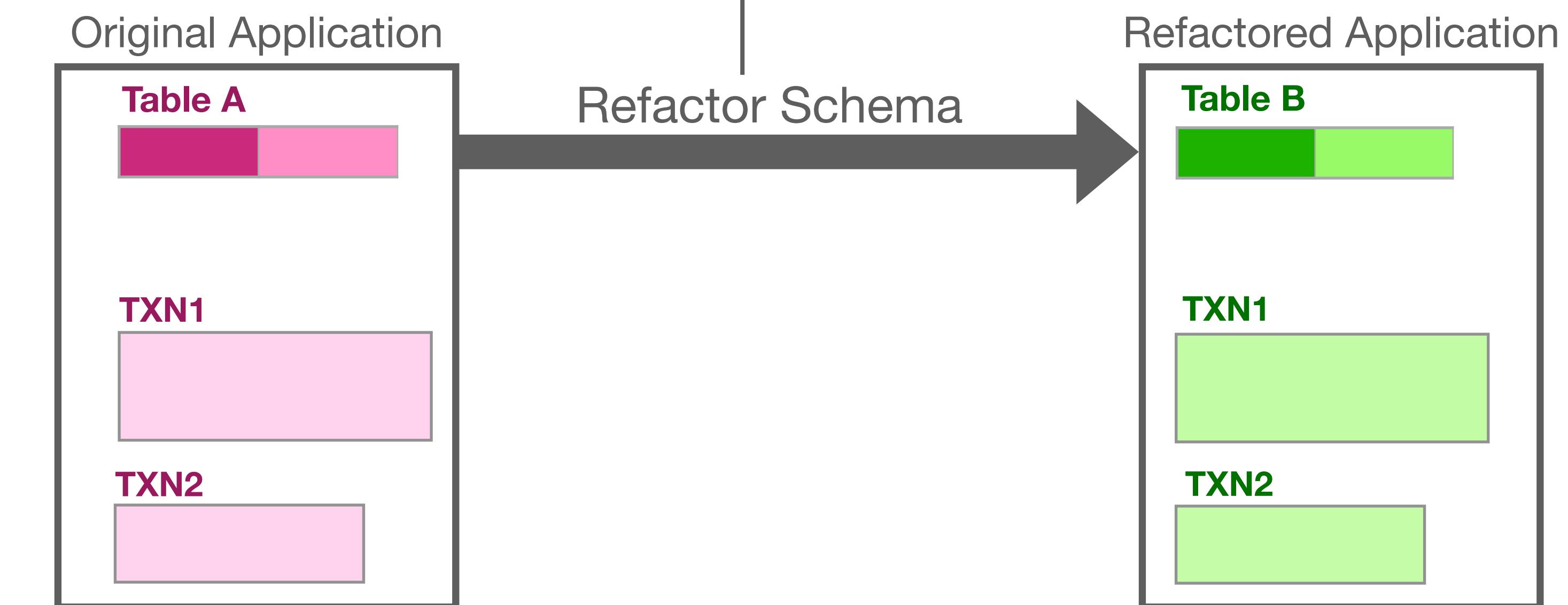
?

?	?	?	?
---	---	---	---

REFACTORING FOR ELIMINATING ANOMALIES

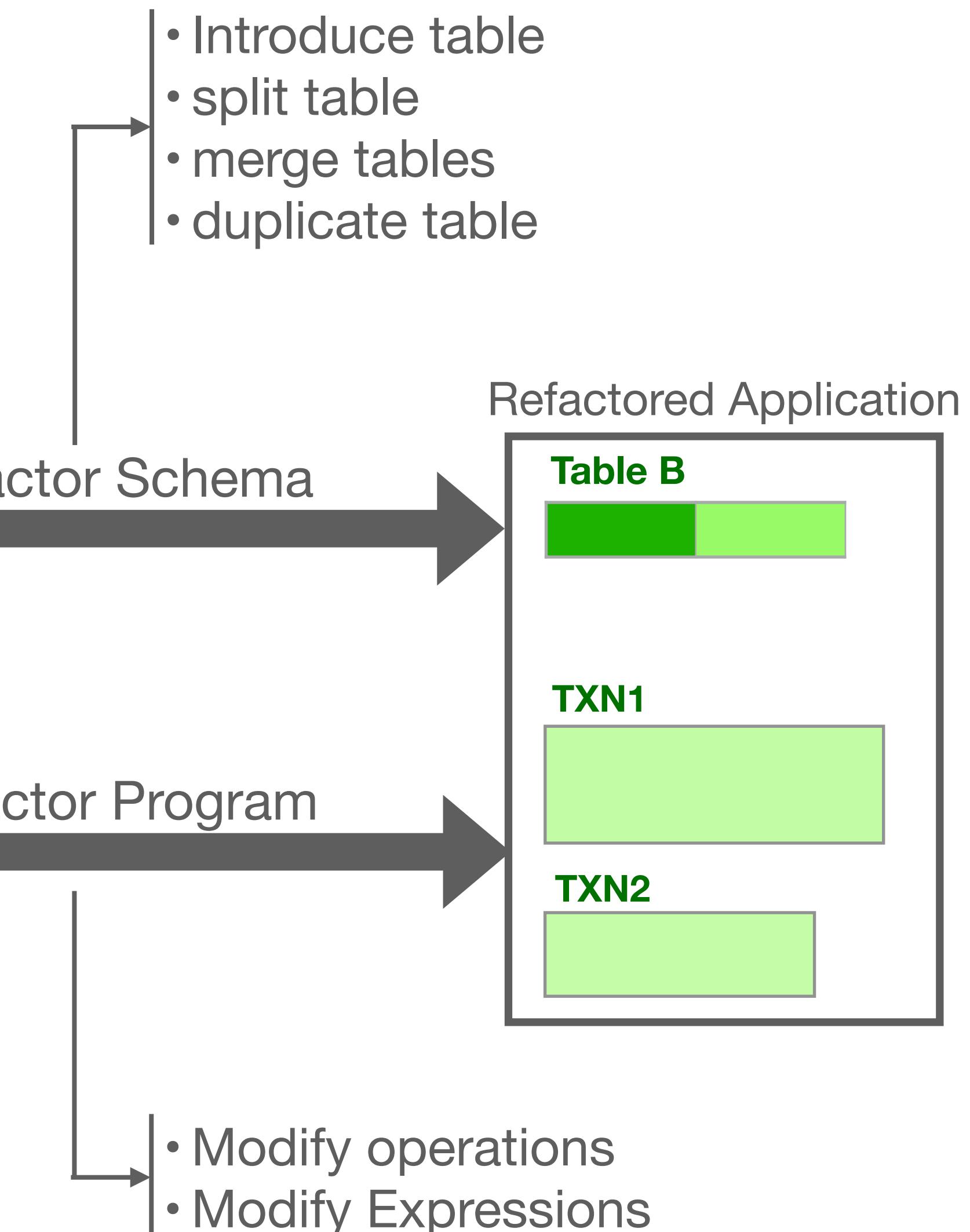
- Equivalent program without dependency cycles
 - Schema Refactoring

- Introduce table
- split table
- merge tables
- duplicate table



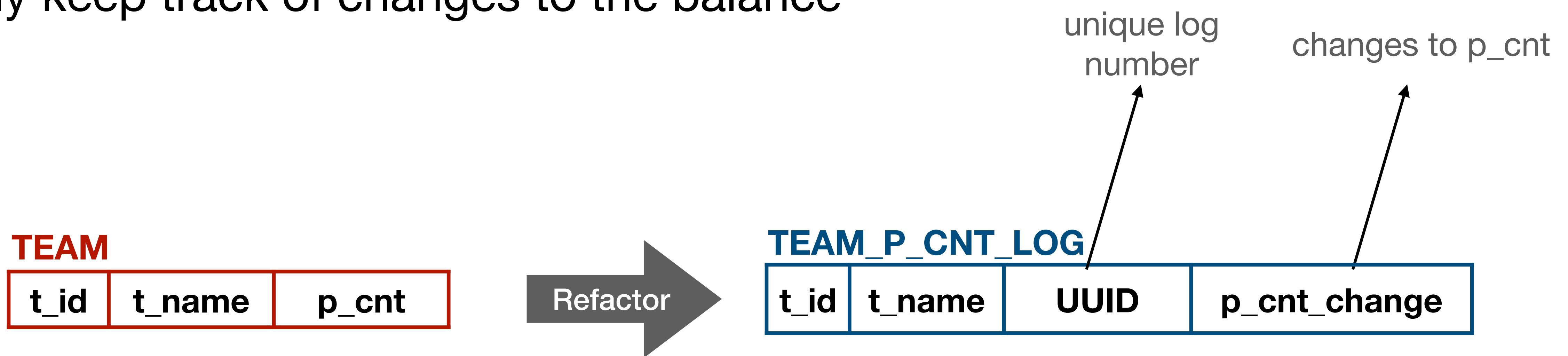
REFACTORING FOR ELIMINATING ANOMALIES

- 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



REFACTORING FOR ELIMINATING ANOMALIES

- 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
```

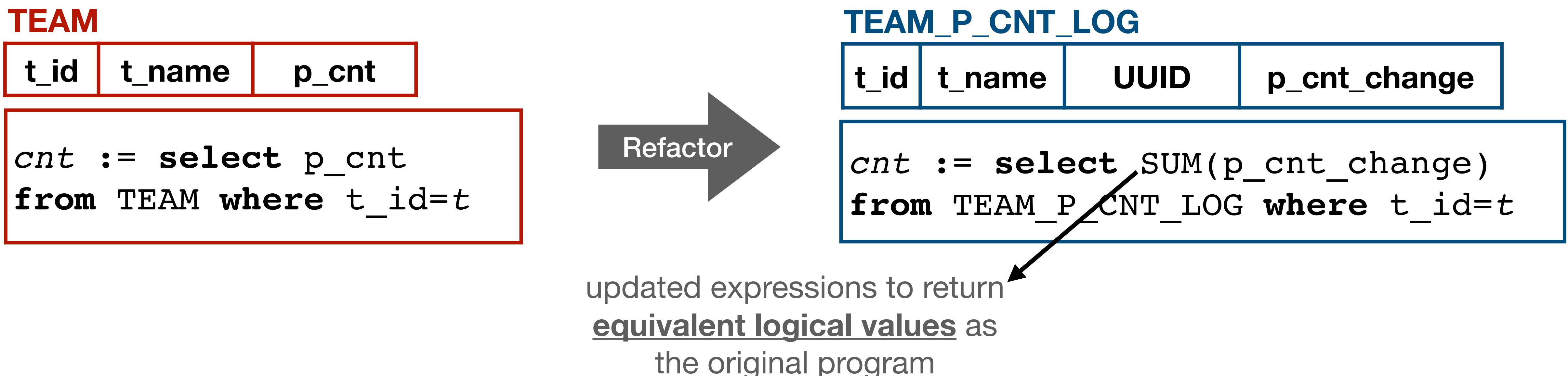
New record is inserted for each new player

TEAM_P_CNT_LOG

t_id	t_name	UUID	p_cnt_change

REFACTORING FOR ELIMINATING ANOMALIES

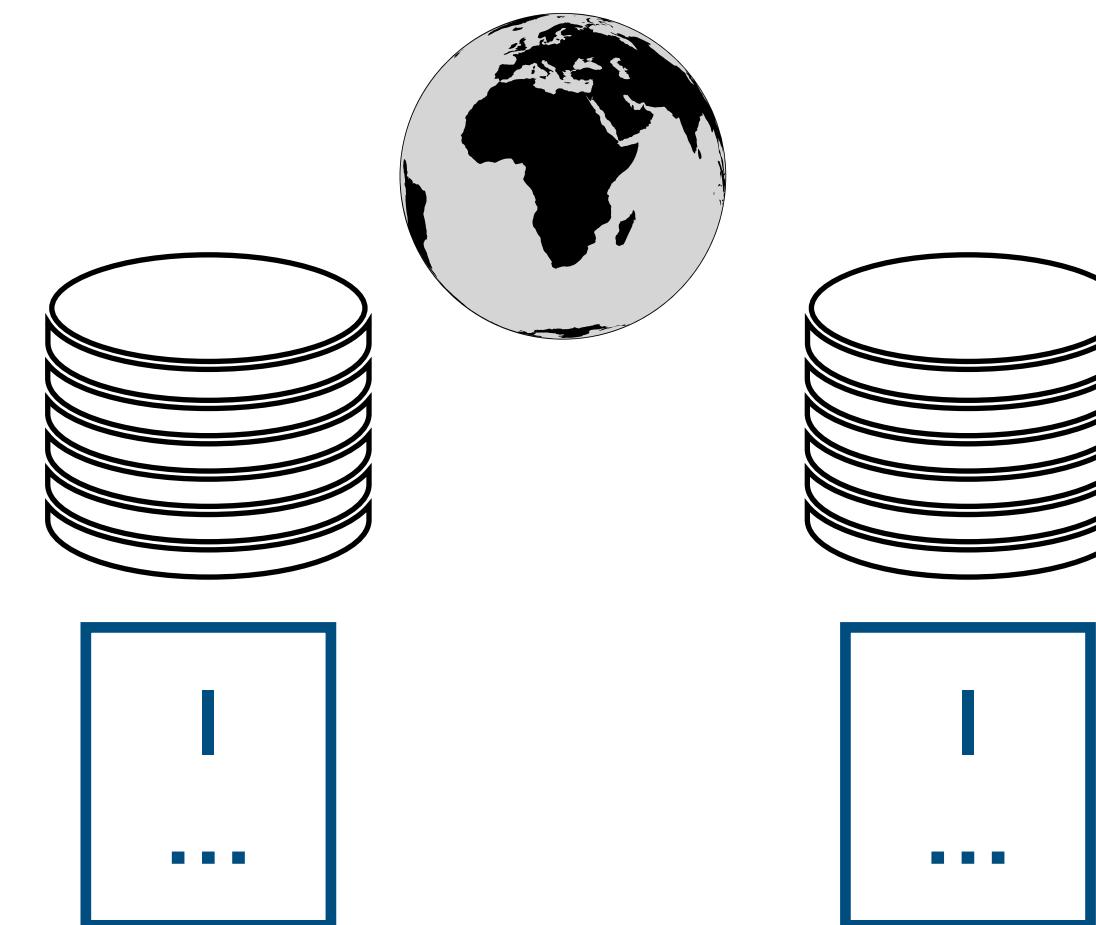
- Equivalent program without dependency cycles
 - ▶ Schema Refactoring
 - ▶ Program Refactoring
- Only keep track of changes to the balance
- No shared item → No dependency



REPAIRED PROGRAM'S EXECUTION

TEAM_P_CNT_LOG

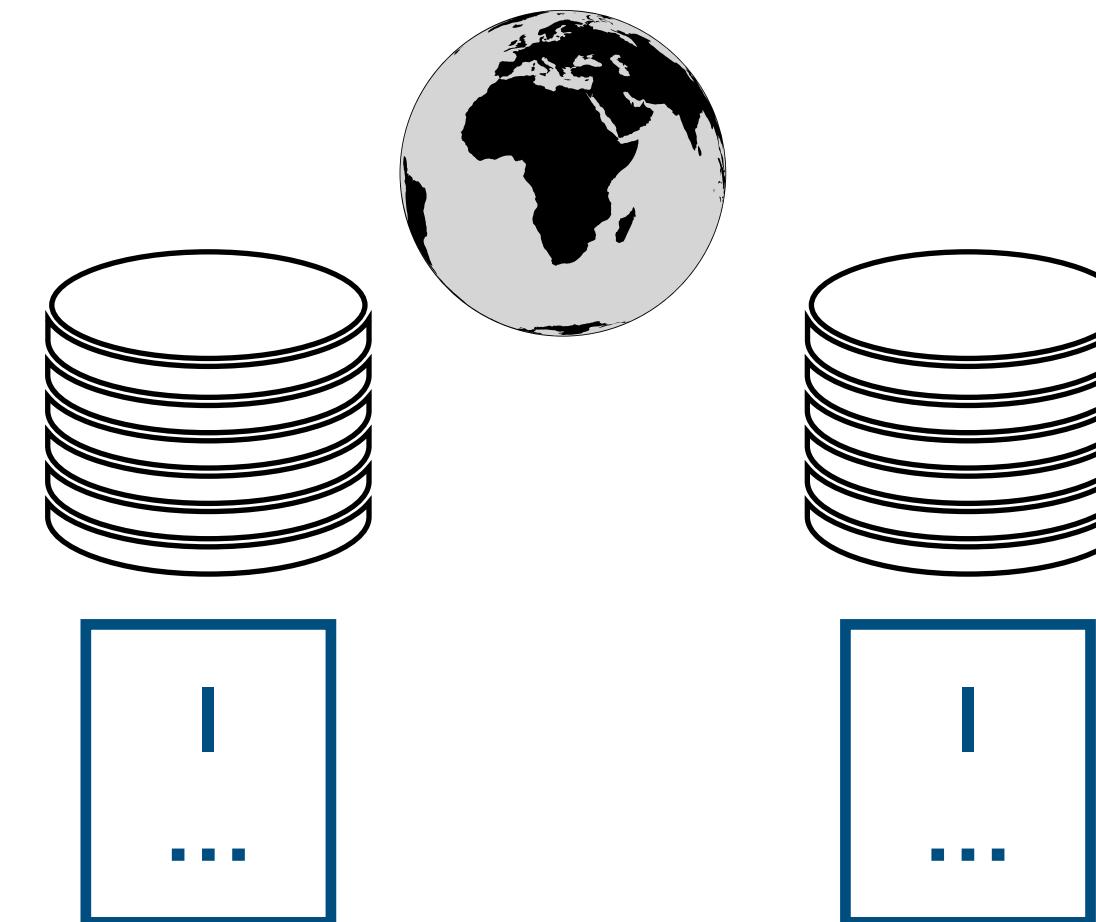
t_id	t_name	UUID	p_cnt_change
1	"A"	U3	2



REPAIRED PROGRAM'S EXECUTION

TEAM_P_CNT_LOG

t_id	t_name	UUID	p_cnt_change
1	"A"	U3	2



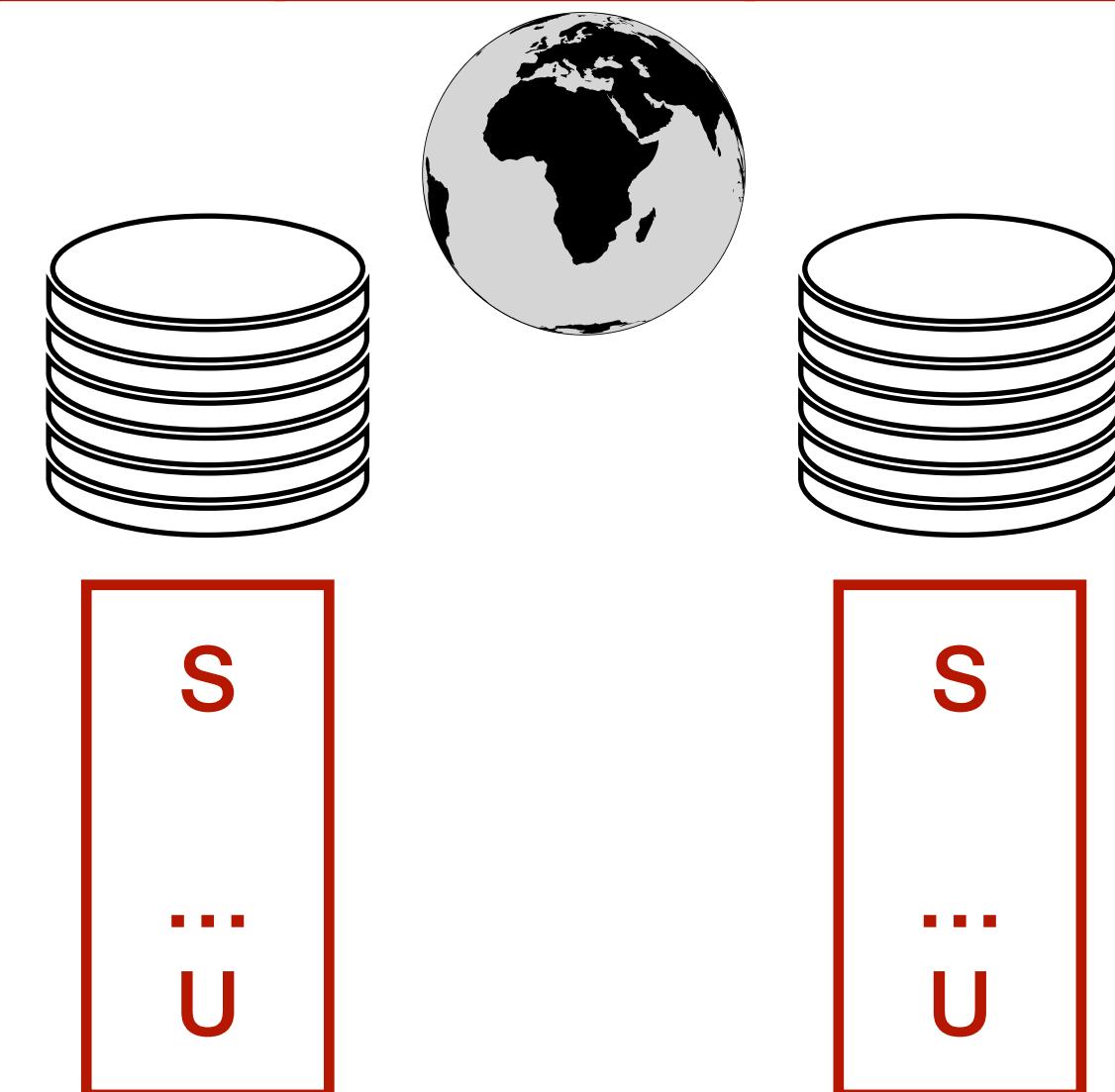
TEAM_P_CNT_LOG

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

REPAIRED PROGRAM'S EXECUTION

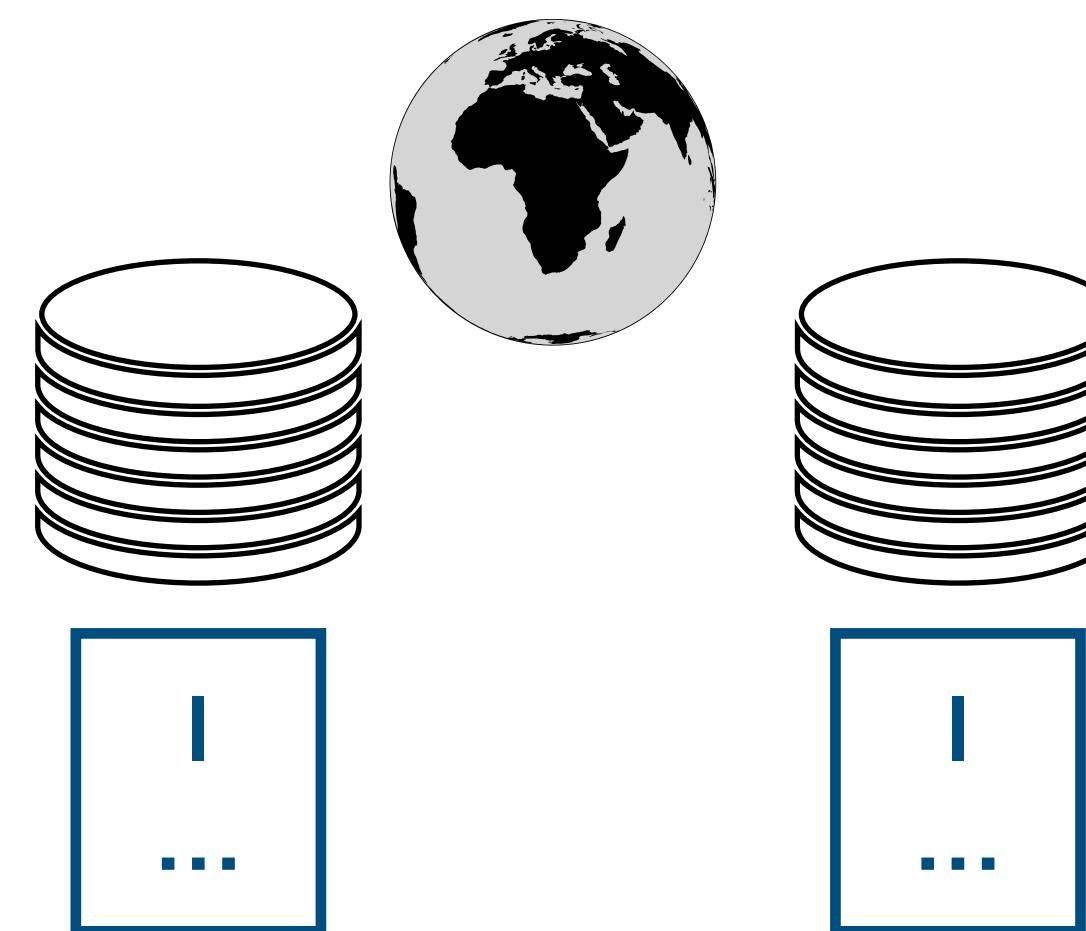
TEAM

t_id	t_name	p_cnt
T1	"A"	2



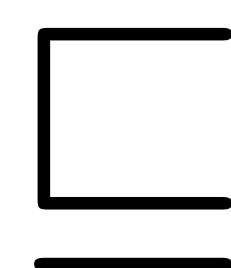
TEAM_P_CNT_LOG

t_id	t_name	UUID	p_cnt_change
1	"A"	U3	2



TEAM

t_id	t_name	p_cnt
T1	"A"	4



TEAM_P_CNT_LOG

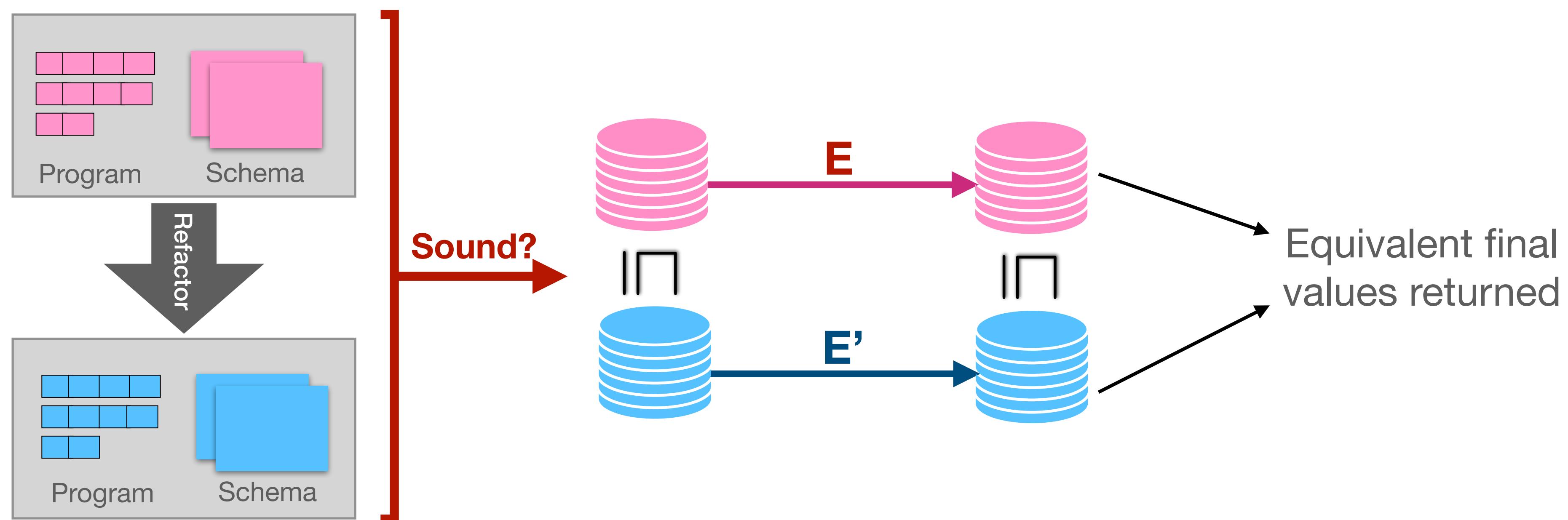
t_id	t_name	UUID	p_cnt_change
1	"A"	U1	1
1	"A"	U3	2
1	"A"	U4	1

SOUNDNESS

- Program P' is a sound refactoring of P iff:

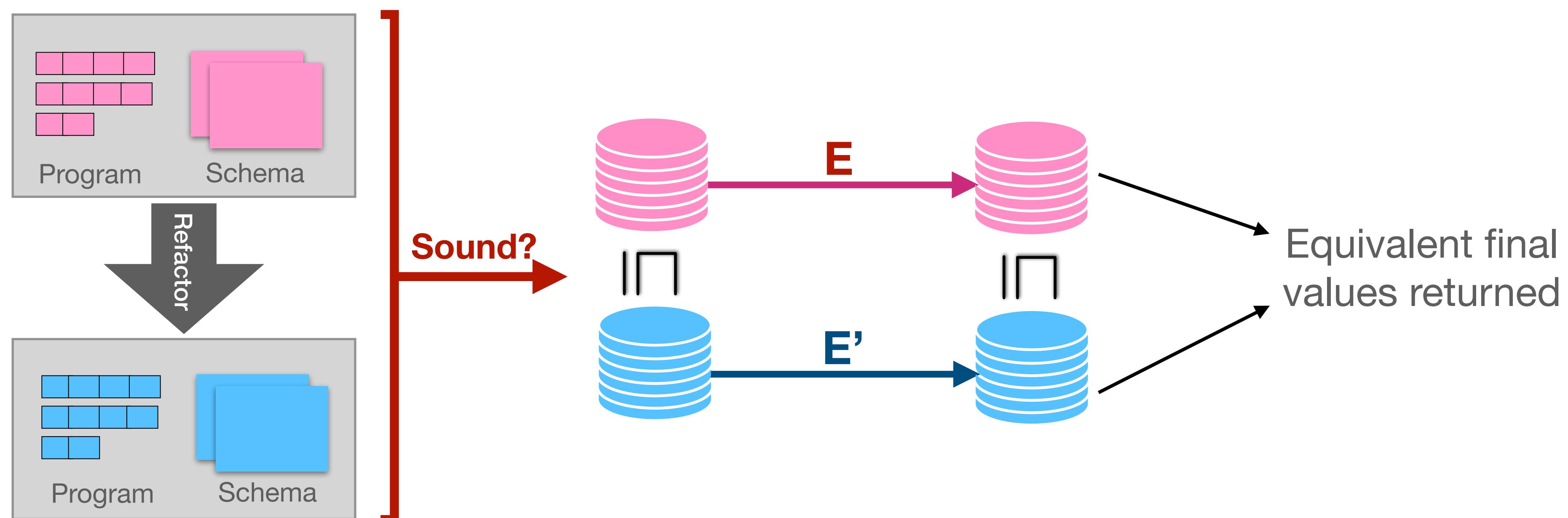
SOUNDNESS

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



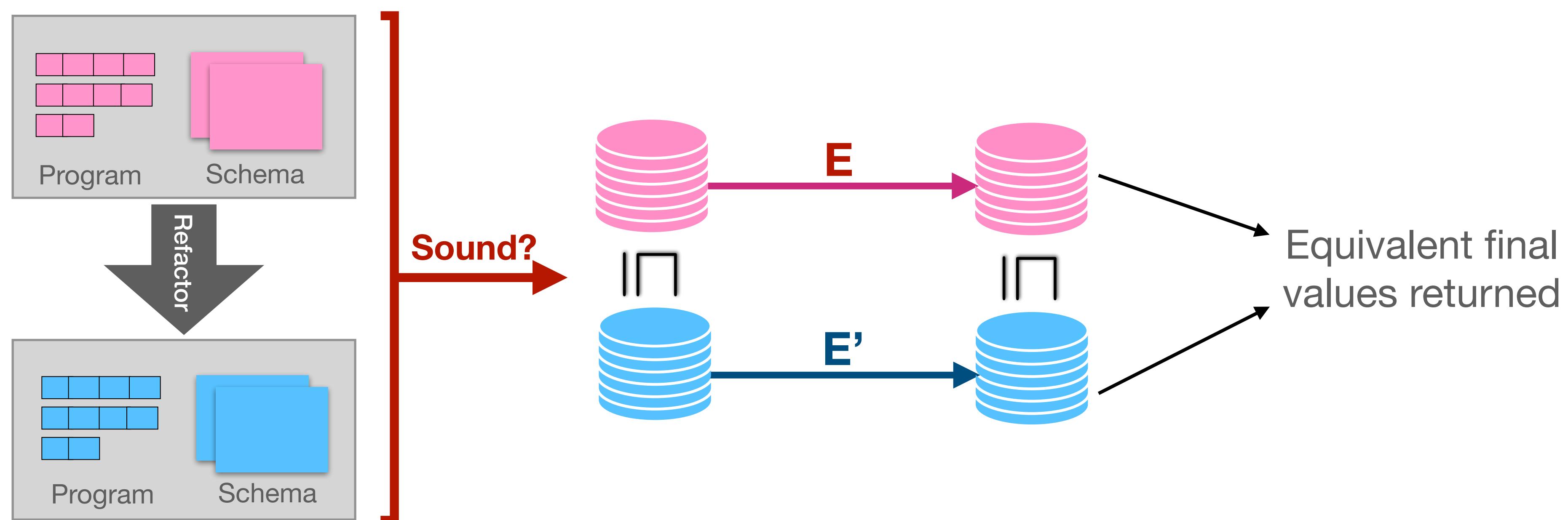
SOUNDNESS

- 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



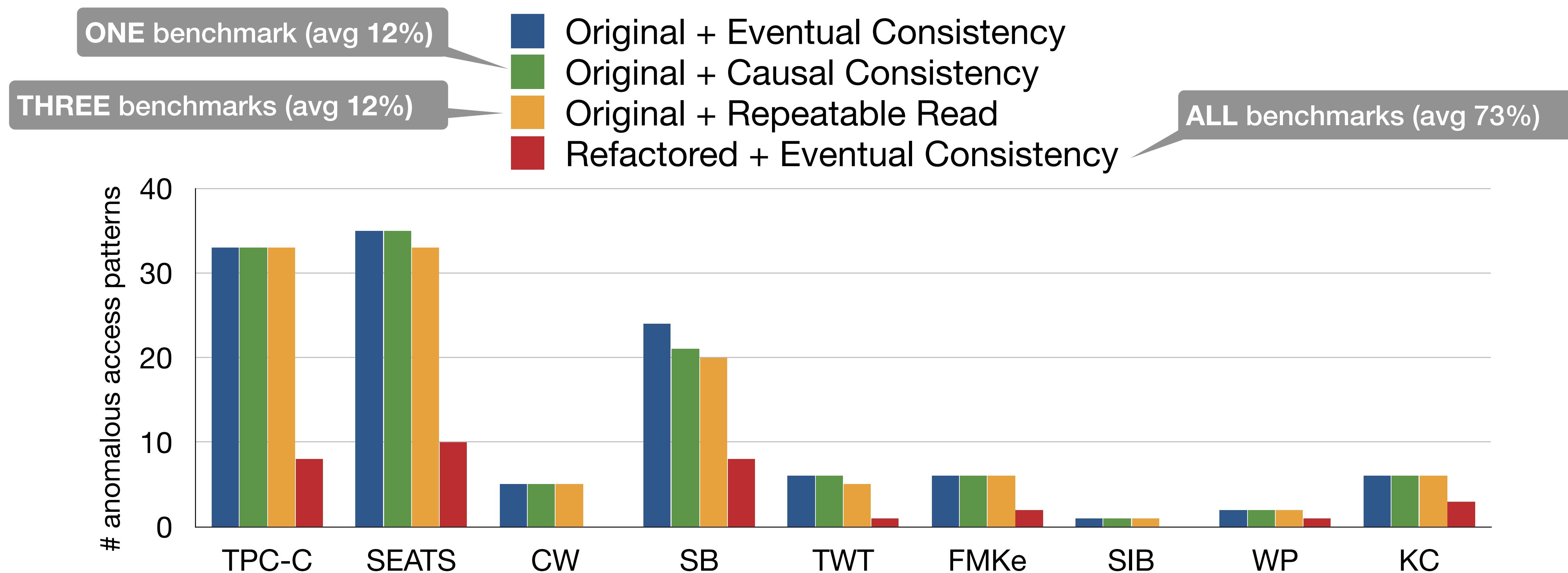
SOUNDNESS

- 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



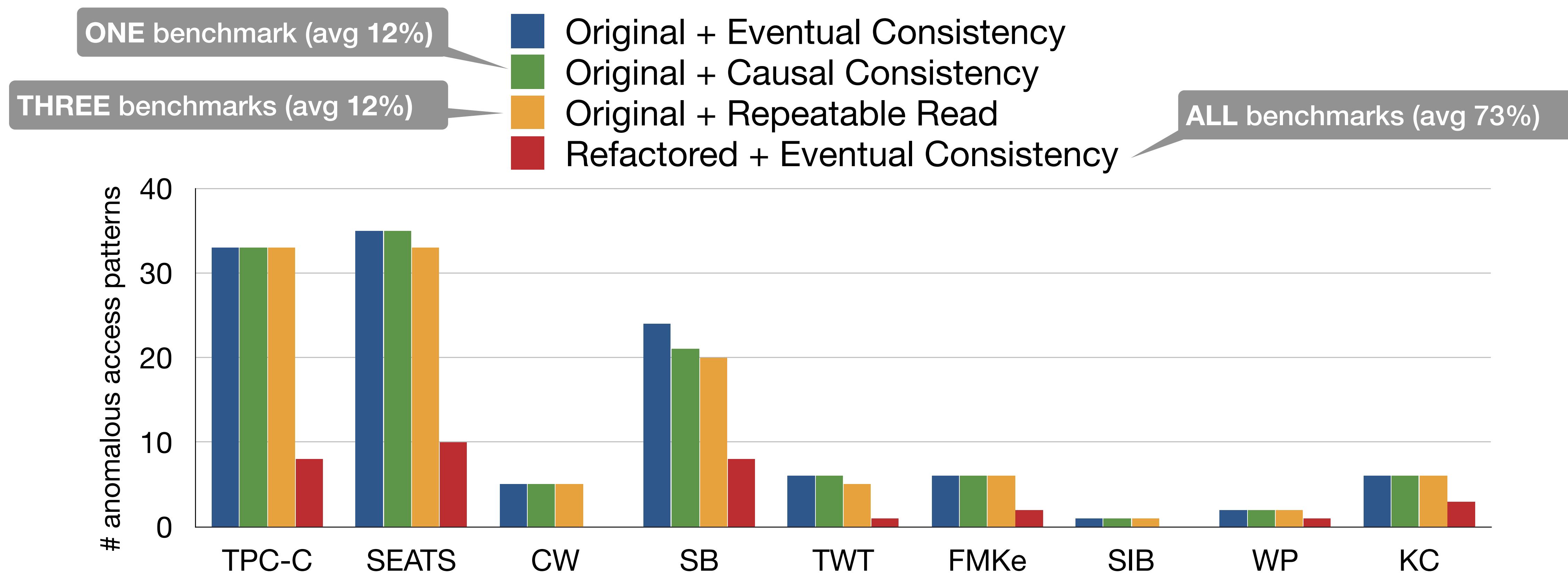
EVALUATION (APPLICABILITY + EFFECTIVENESS)

- 9 benchmarks
 - ▶ Number of statically identified anomalous access pairs



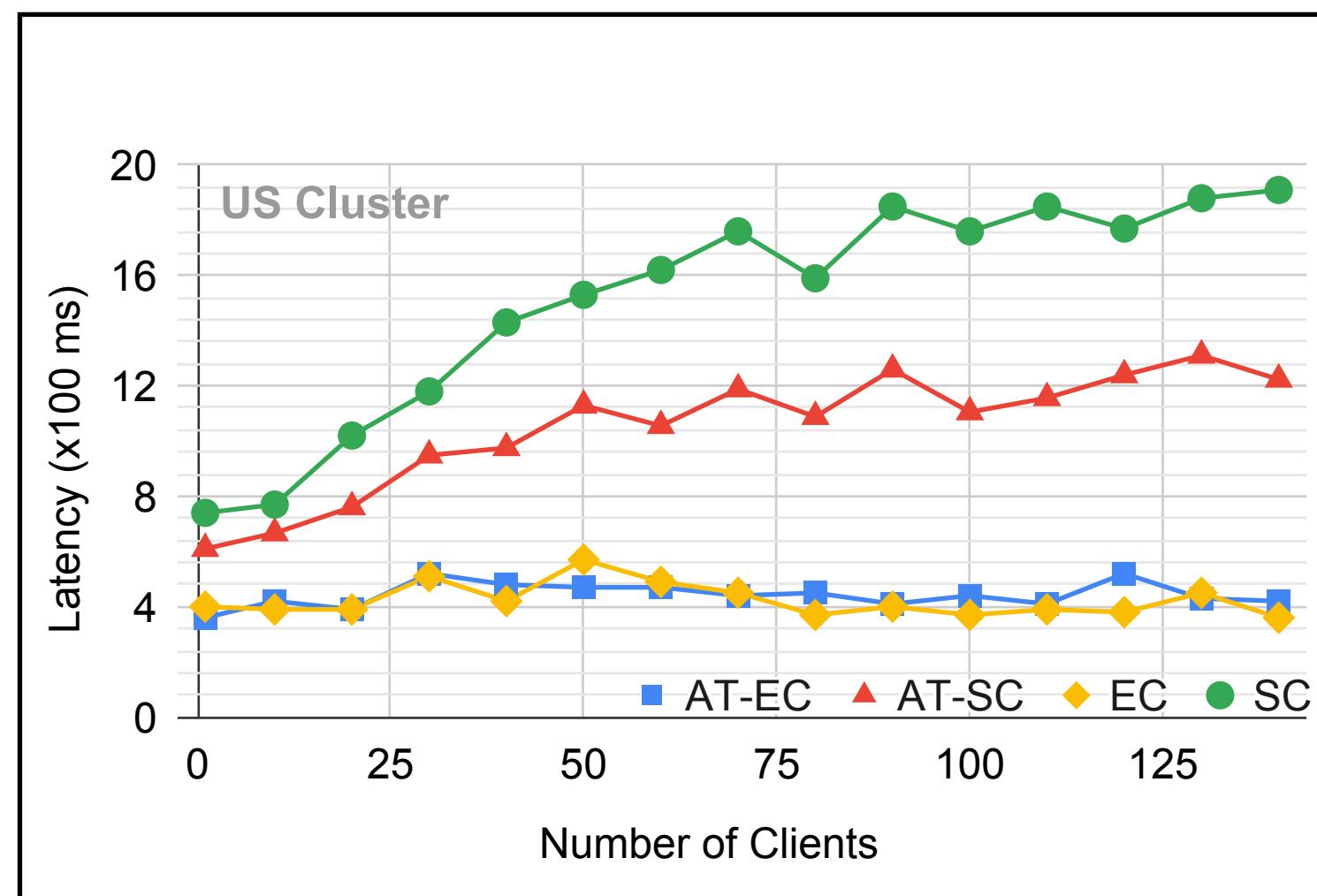
EVALUATION (APPLICABILITY + EFFECTIVENESS)

- 9 benchmarks
 - ▶ Number of statically identified anomalous access pairs

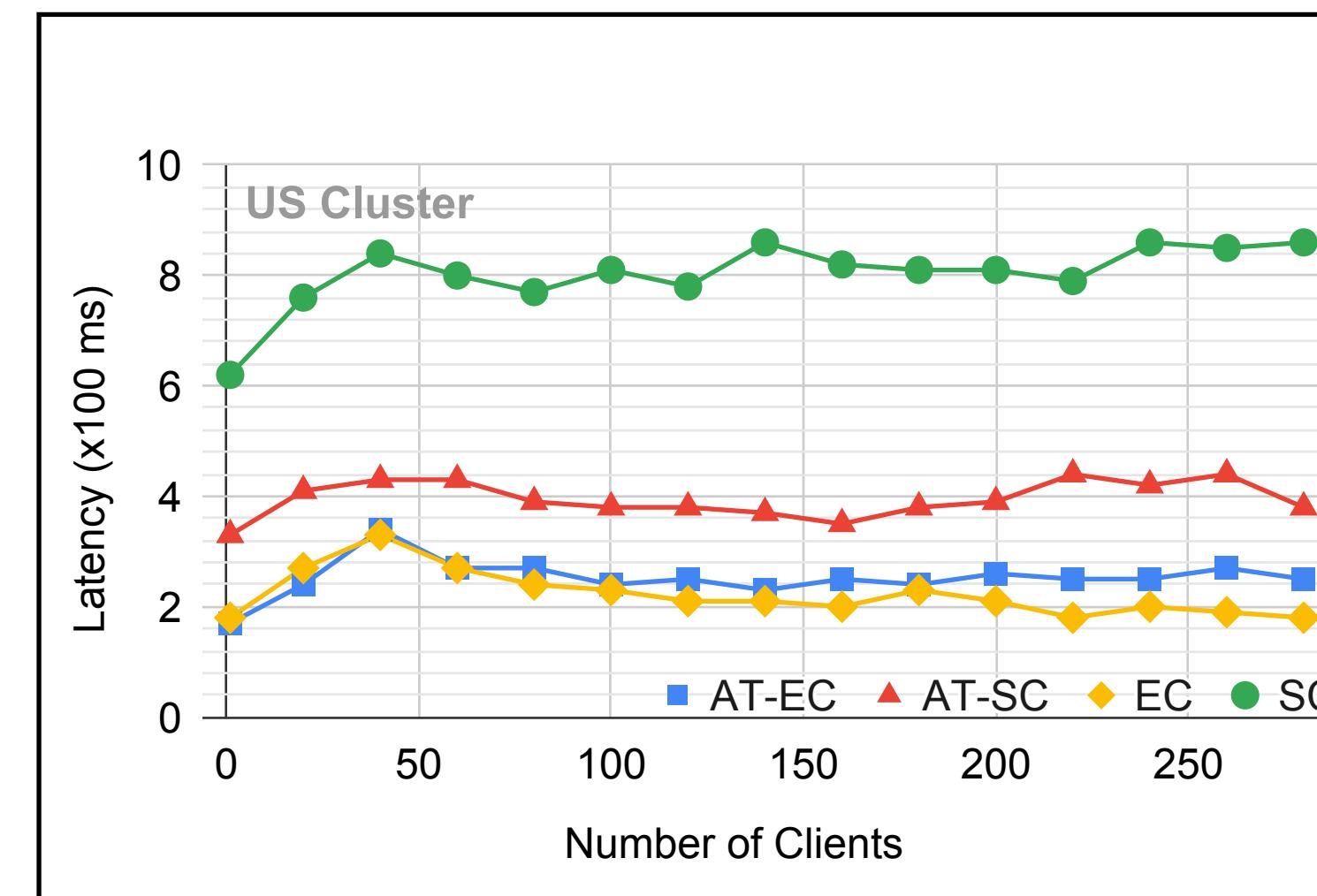


EVALUATION (PERFORMANCE)

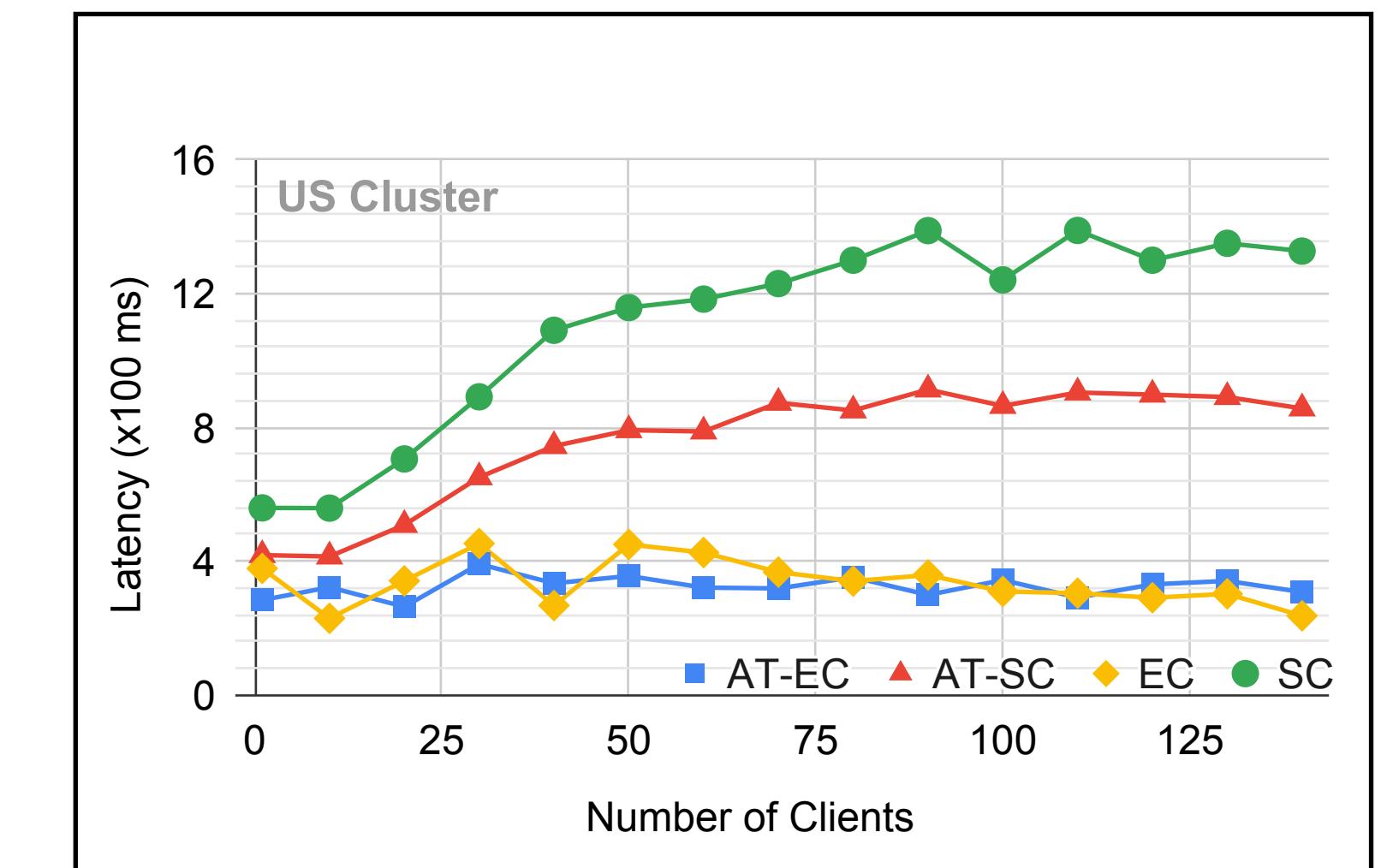
- 3 benchmarks
- Latency and throughput
 - ▶ EC, SC: original program (eventual consistency/serializability)
 - ▶ AT_EC, AT_SC: refactored program (eventual consistency/serializability)



TPC-C

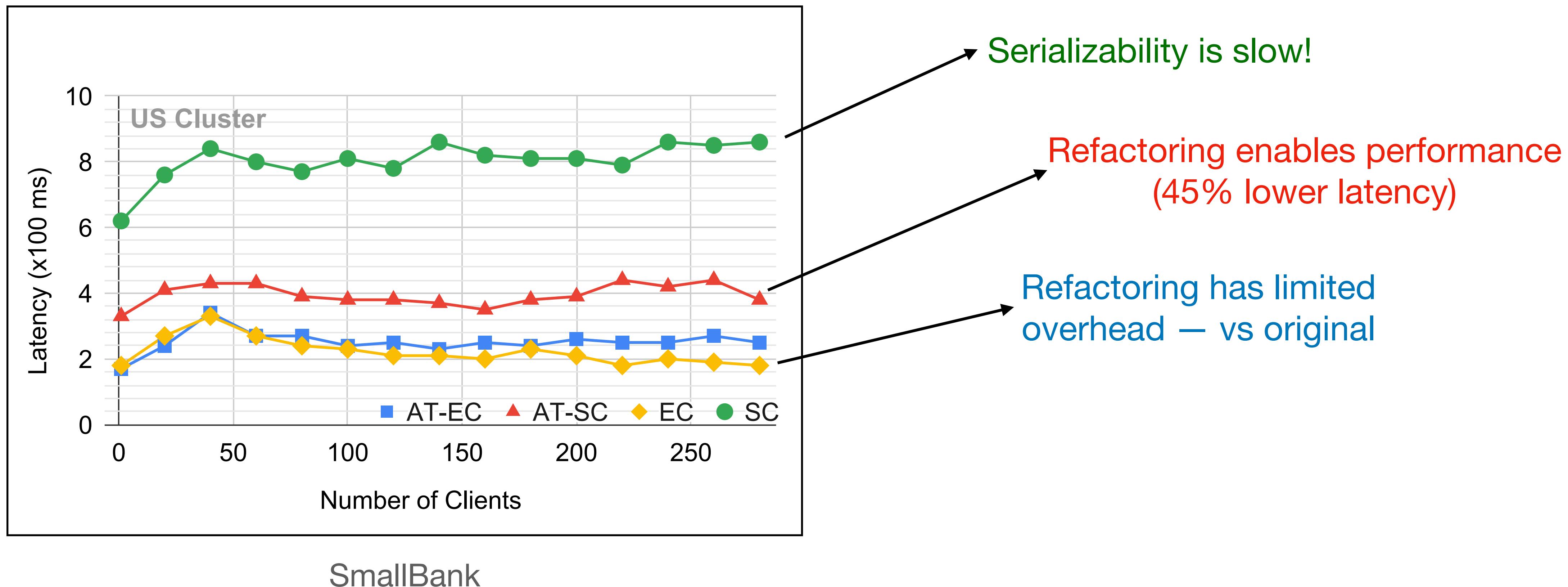


SmallBank



SEATS

EVALUATION (PERFORMANCE)



SmallBank

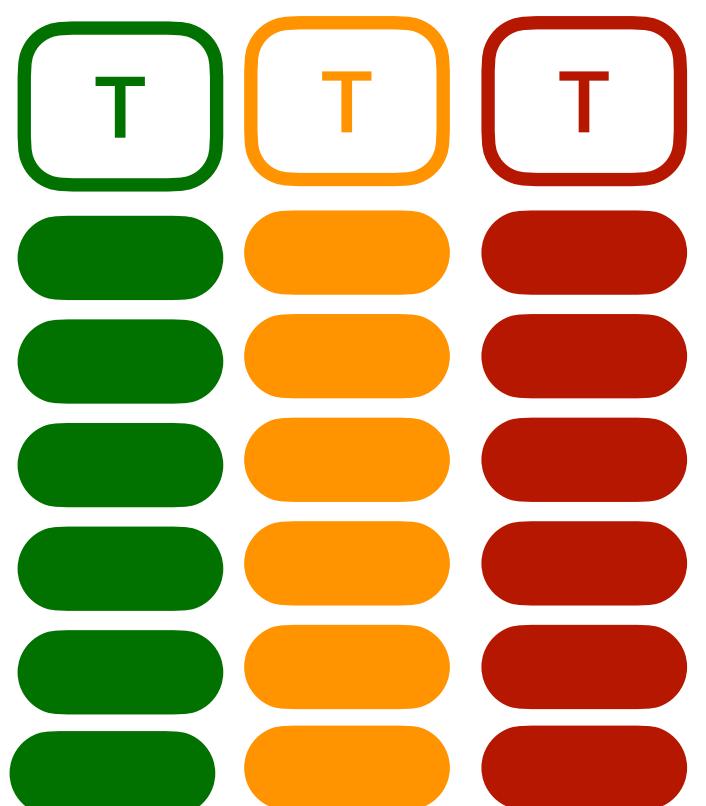
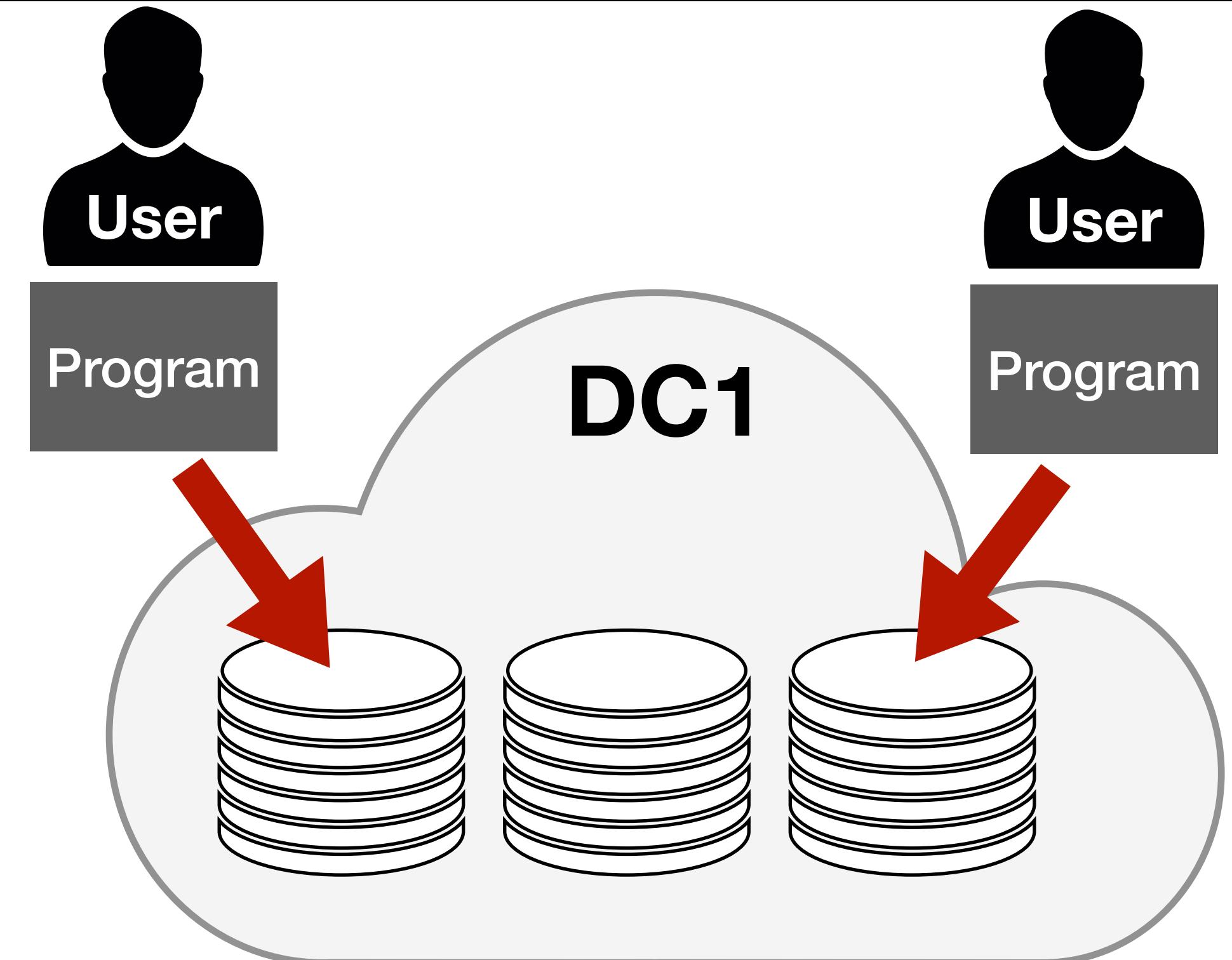
- EC, SC: original program + eventual consistency and serializability
- AT_EC, AT_SC: original program + eventual consistency and serializability

LACHESIS

(REPAIRING PARTITIONING ANOMALIES)

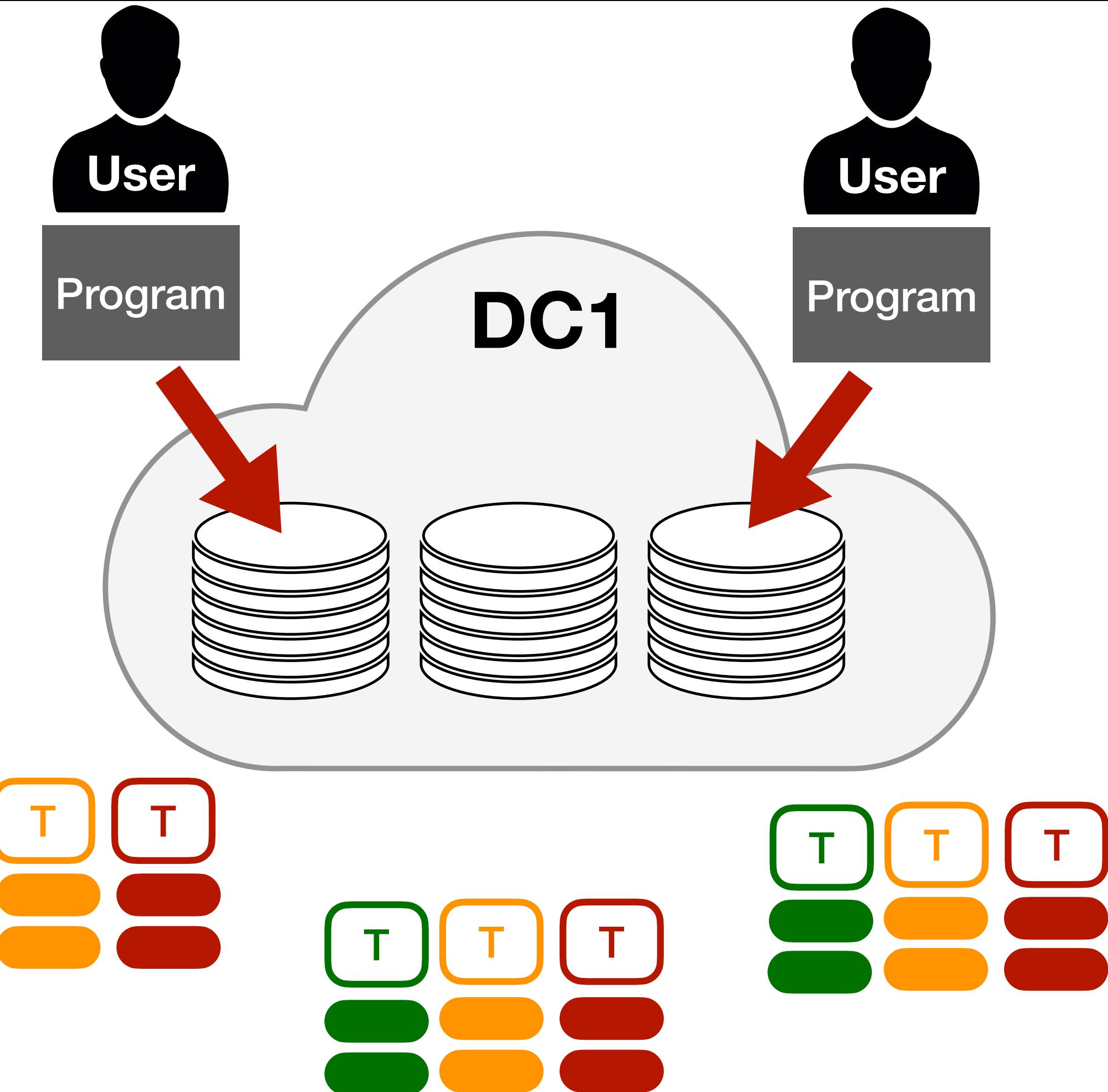
PARTITIONING BASICS

- Partitioned Database Clusters
 - Same data-center deployment



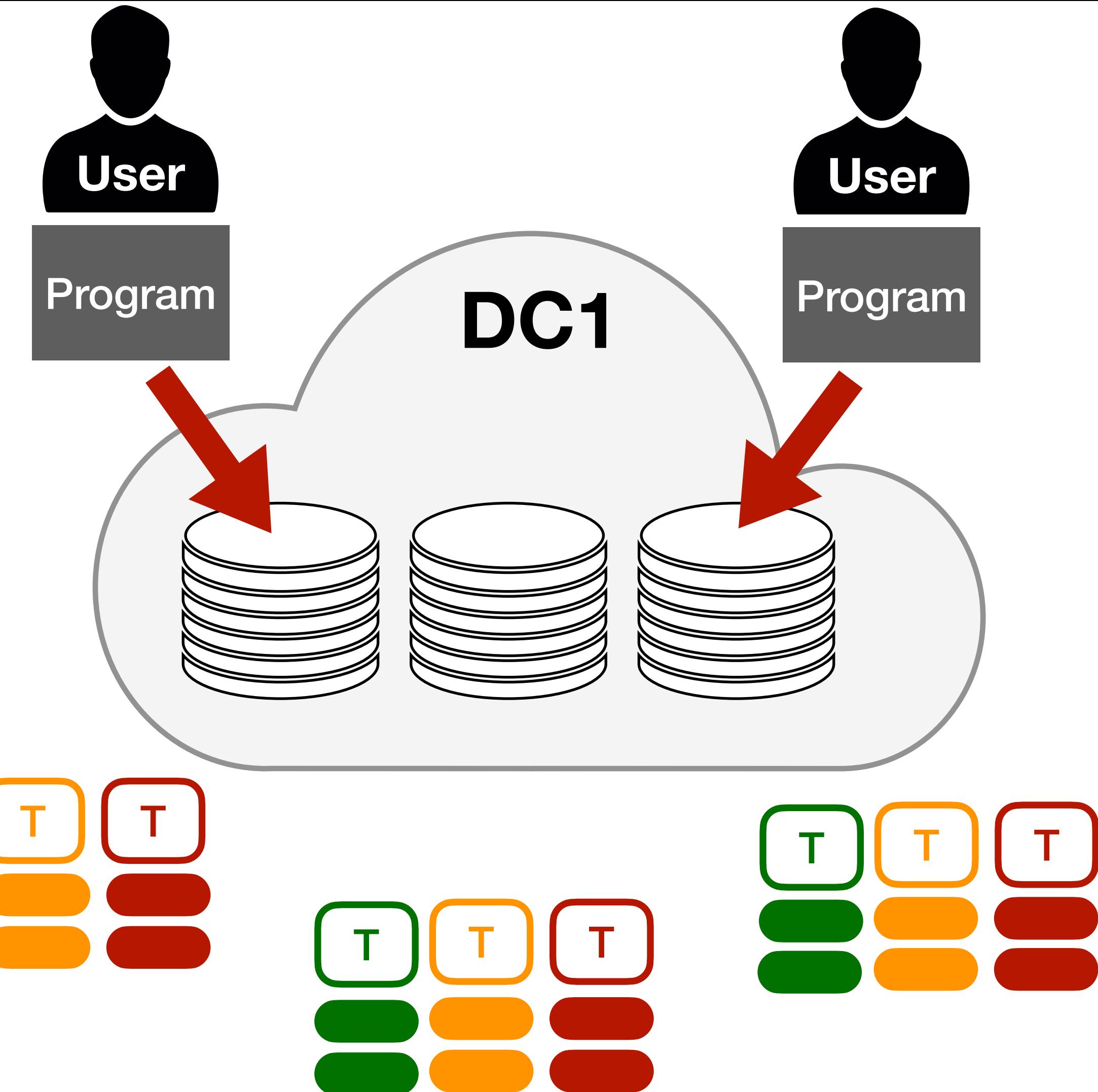
PARTITIONING BASICS

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



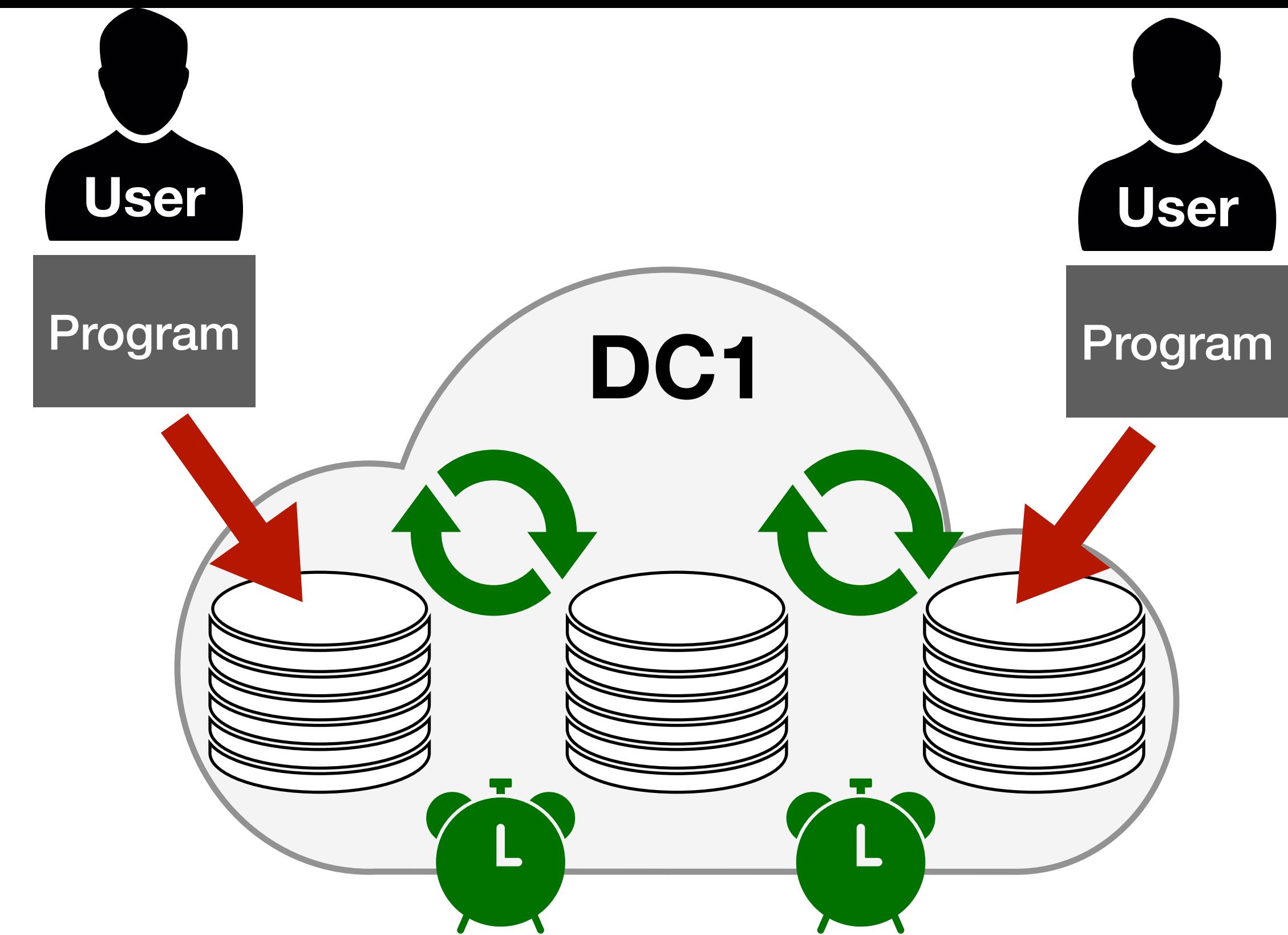
PARTITIONING BASICS

- Partitioned Database Clusters
 - ▶ Same data-center deployment
 - ▶ Subset of records at each node
- Partitioning Policy Π
 - ▶ Performance



PARTITIONING BASICS

- Partitioned Database Clusters
 - ▶ Same data-center deployment
 - ▶ Subset of records at each node
- Partitioning Policy Π
 - ▶ Performance



PARTITIONING BASICS

- Partitioned Database Clusters
 - ▶ Same data-center deployment
 - ▶ Subset of records at each node
- Partitioning Policy Π
 - ▶ 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

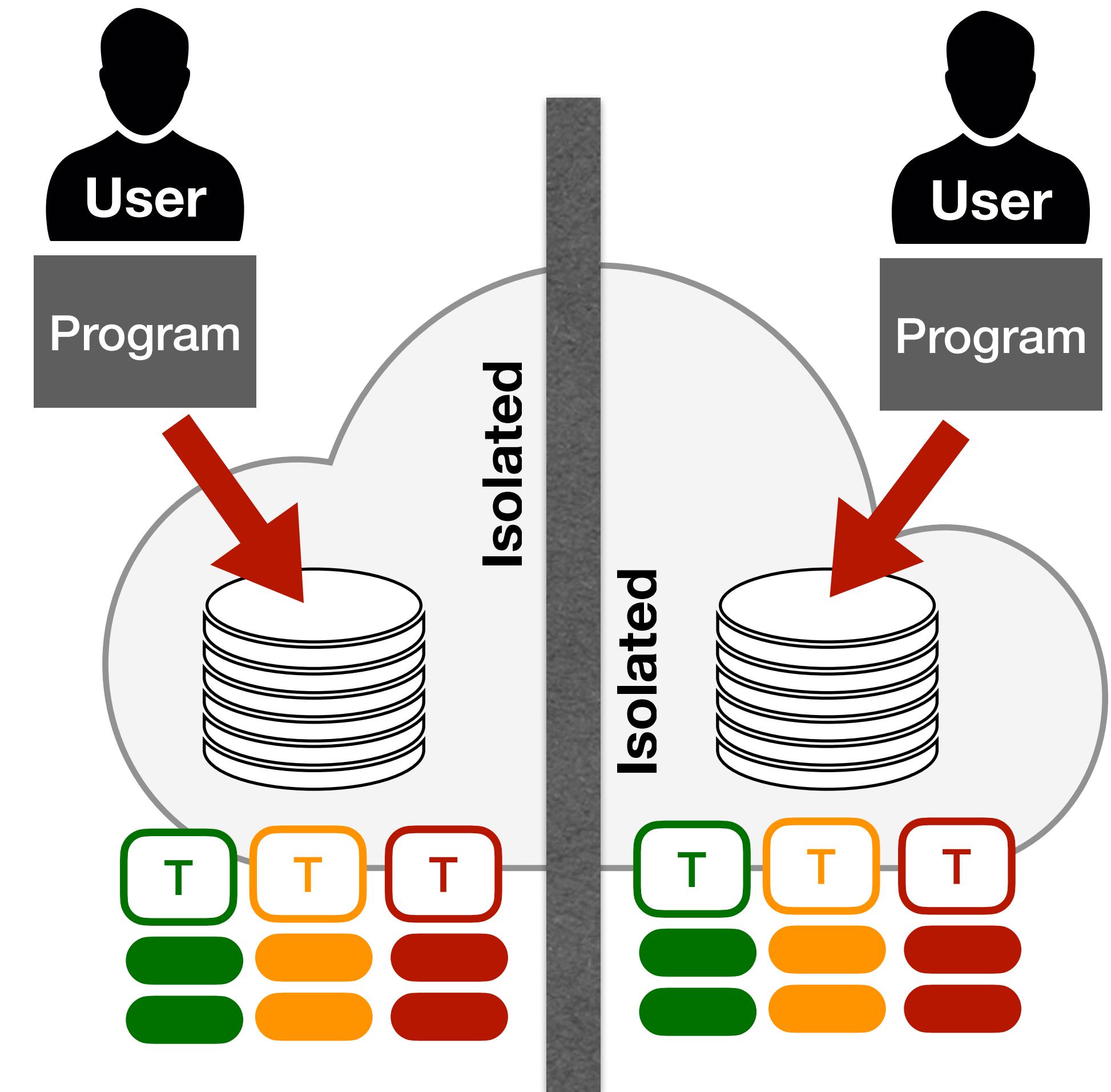
Sam Madden
madden@csail.mit.edu

CONCURRENCY IN PARTITIONING

- Correctness?

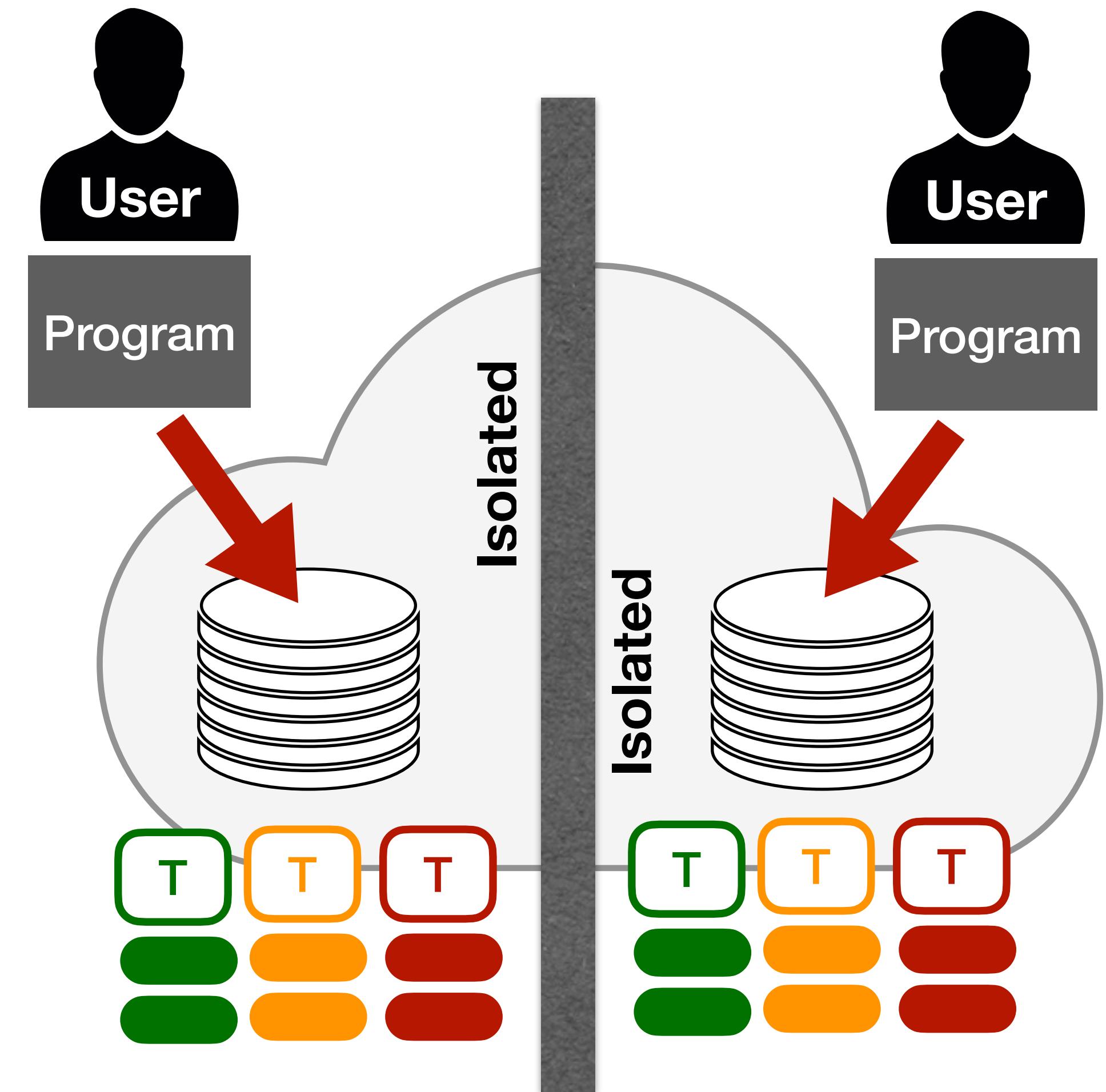
CONCURRENCY IN PARTITIONING

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



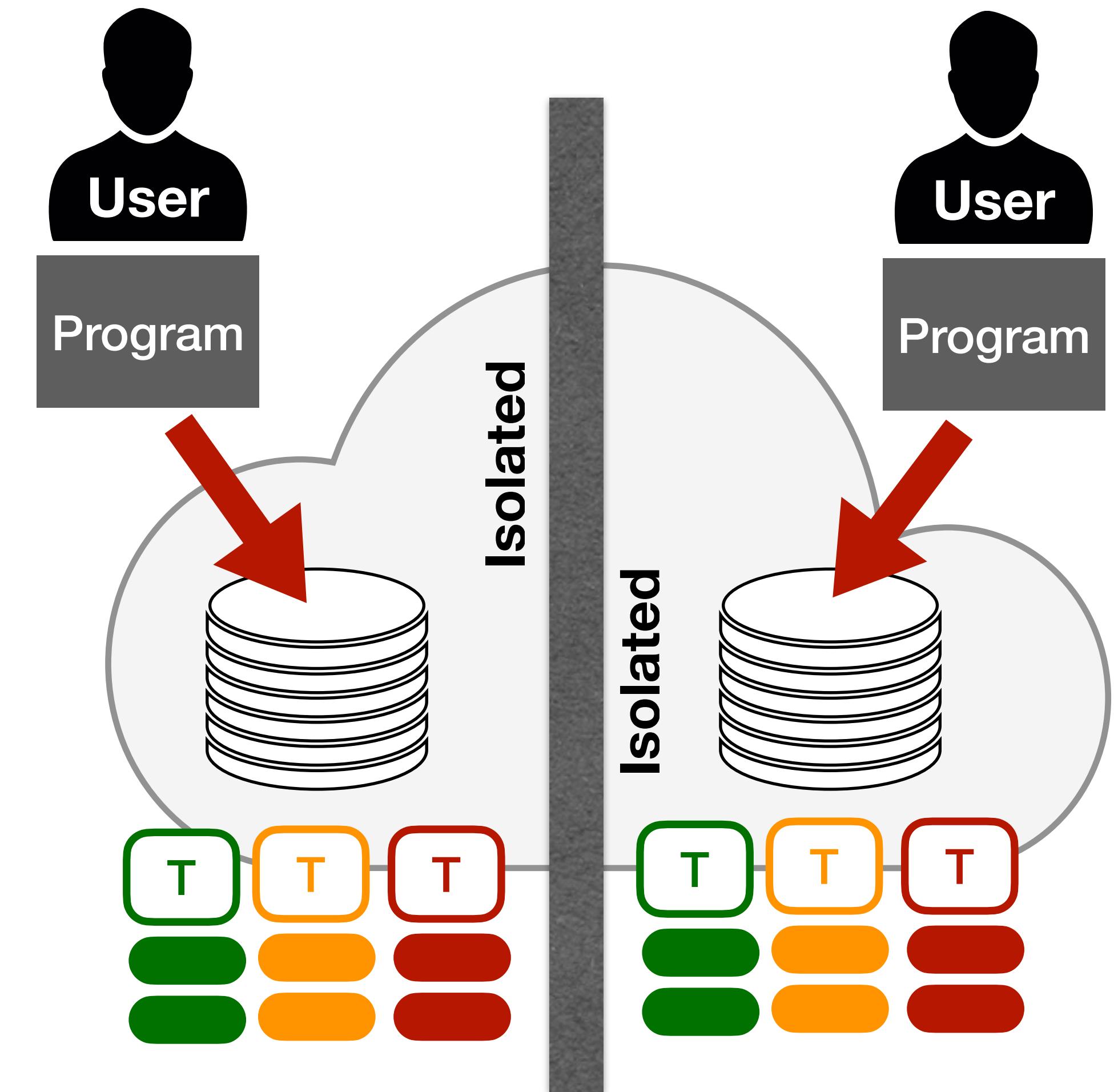
CONCURRENCY IN PARTITIONING

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



CONCURRENCY IN PARTITIONING

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



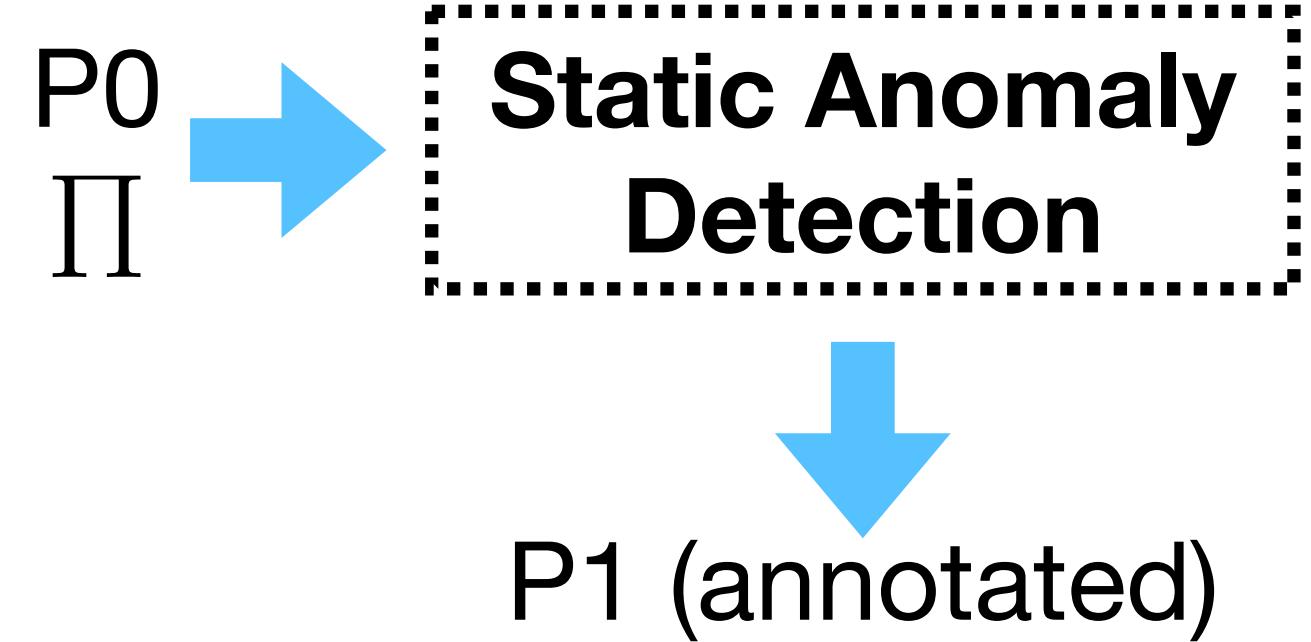
INTRODUCTION TO LACHESIS

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



INTRODUCTION TO LACHESIS

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



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.*)
```

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.* )
```

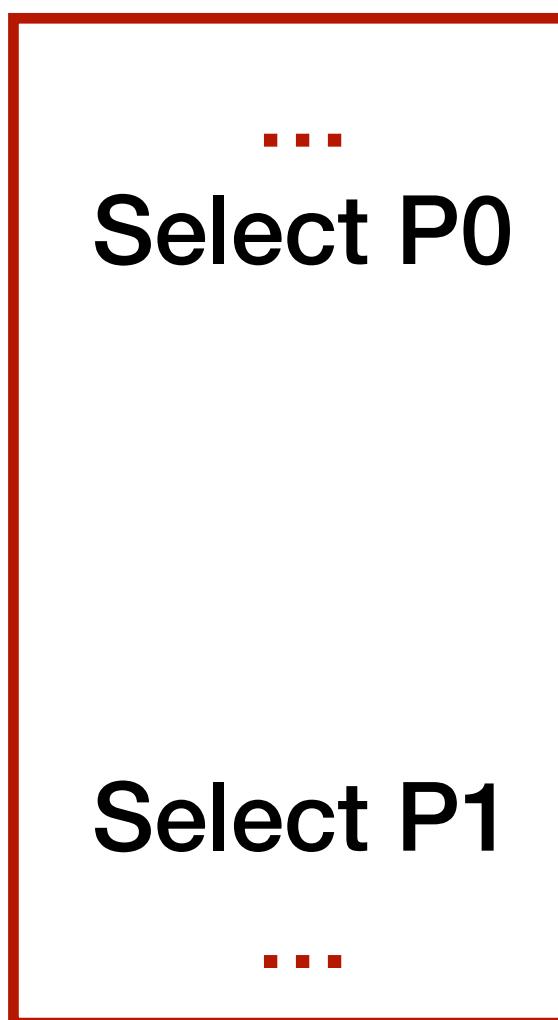
PLAYER			
p_id	p_t_id	p_role	p_stat
P0	T1	captain	on
P1	T1	soldier	on
P2	T2	soldier	off

||| →

INTRODUCTION TO LACHESIS

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

playerByTeam (t=T1)



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
P0	T1	captain	on
P1	T1	soldier	on
P2	T2	soldier	off

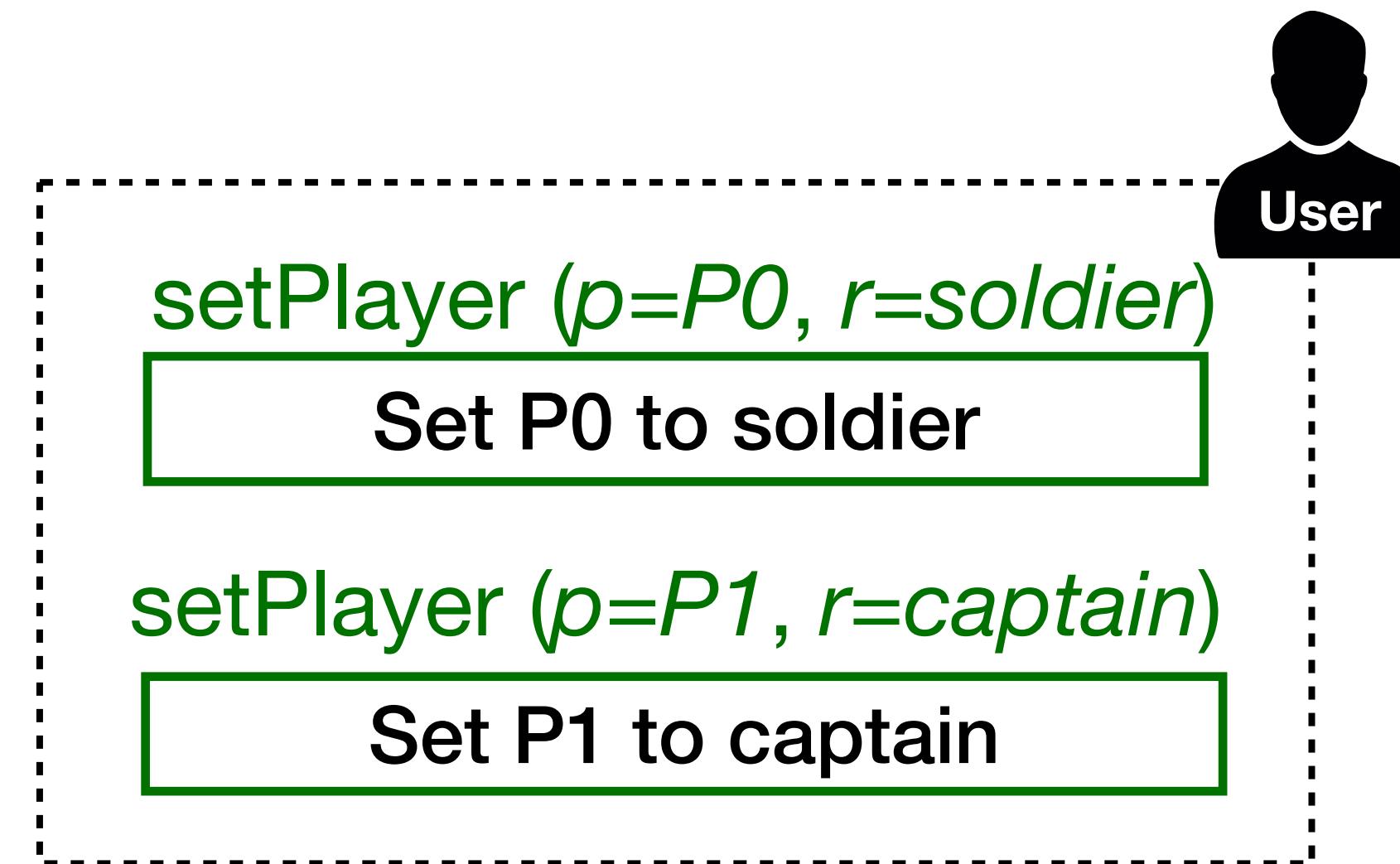
Three arrows point to the rows in the table: a blue arrow points to row P0, a pink arrow points to row P1, and a green arrow points to row P2.

INTRODUCTION TO LACHESIS

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

playerByTeam (t=T1)

...
Select P0
...
Select P1
...



PlayerByTeam (t)

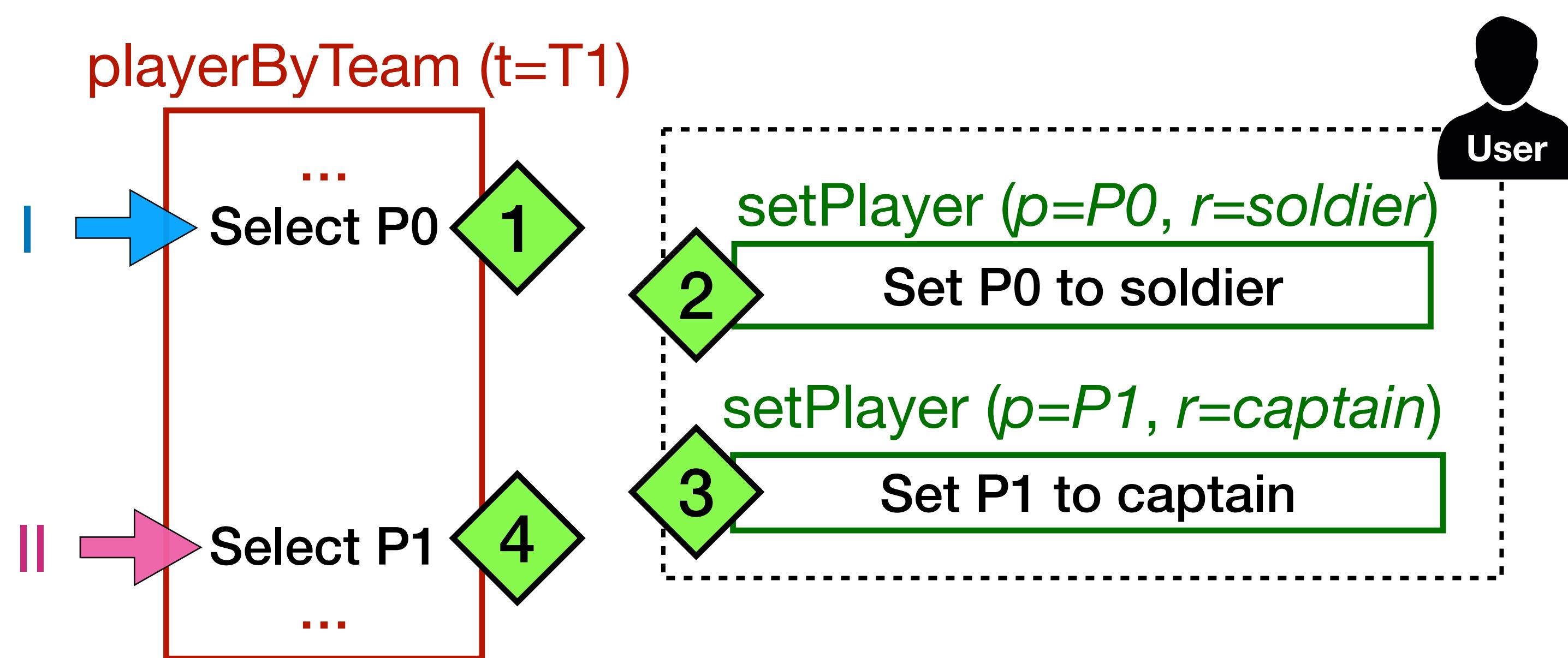
```
ps := select * from PLAYER
      where p_t_id=t
foreach p in ps do
    print(p.*)
```

PLAYER

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

INTRODUCTION TO LACHESIS

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



PlayerByTeam (t)

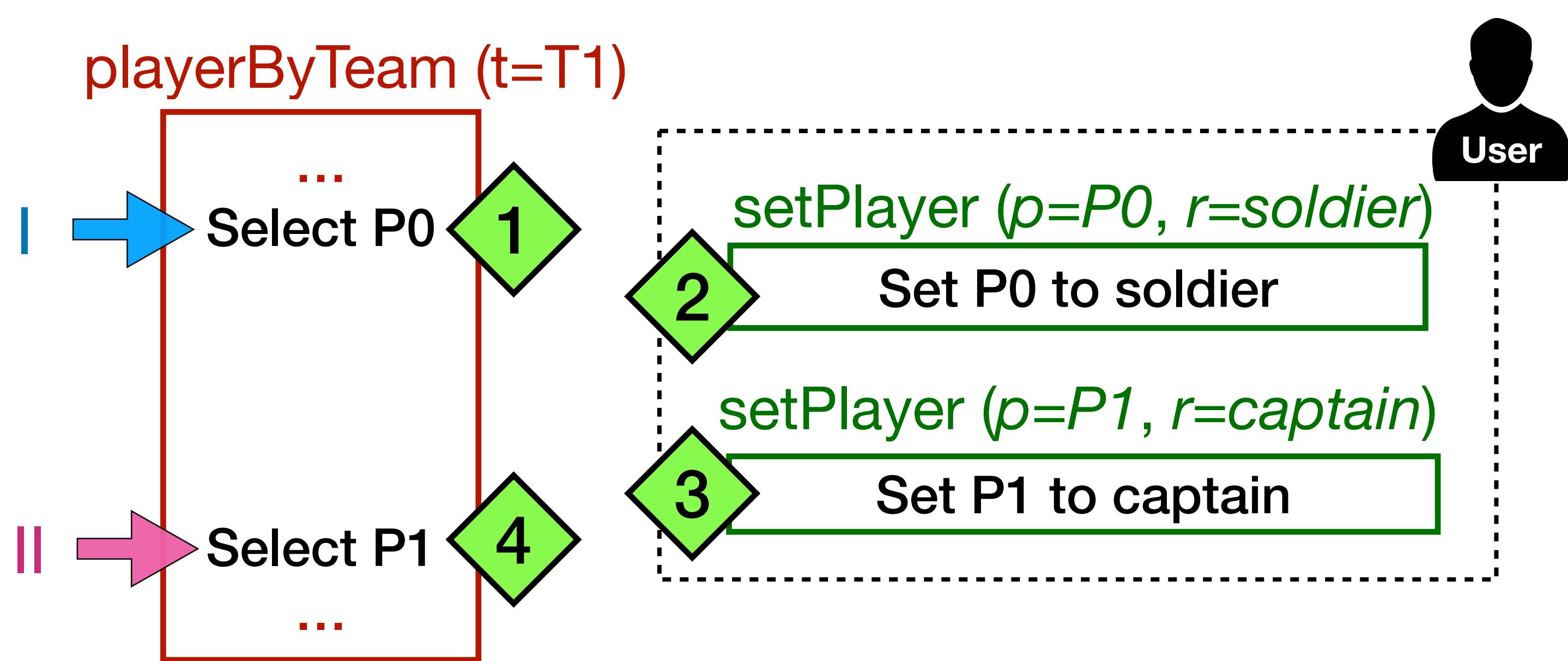
```
ps := select * from PLAYER
      where p_t_id=t
foreach p in ps do
  print(p.* )
```

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

I → II → III →

INTRODUCTION TO LACHESIS

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

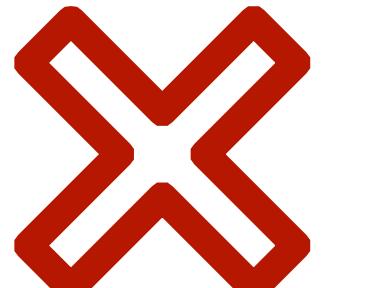


PlayerByTeam (t)

```
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
P0	T1	captain	on
P1	T1	soldier	on
P2	T2	soldier	off

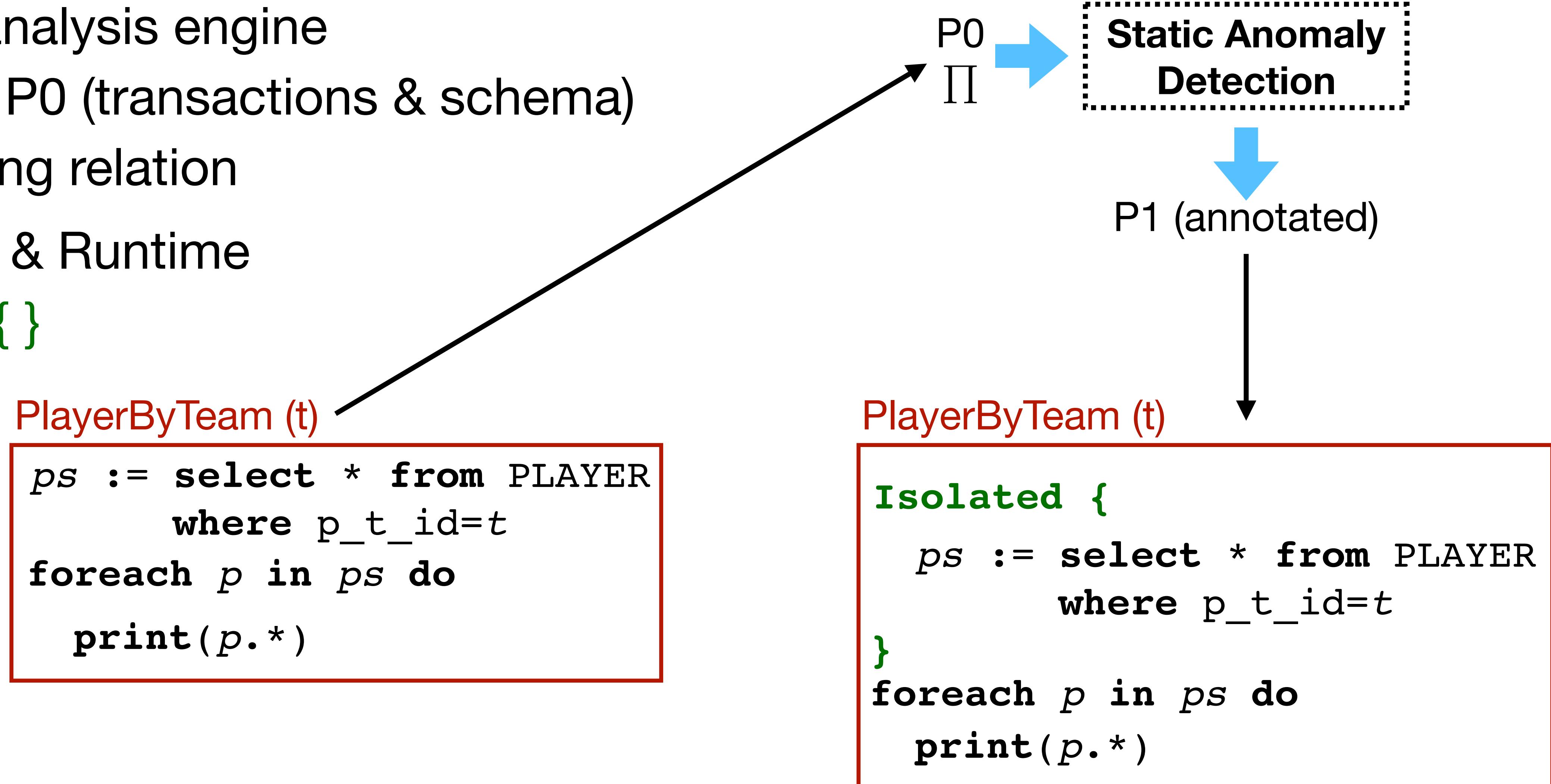
// P0, 1, captain, on
// P1, 1, captain, on



Invariant
Violated!

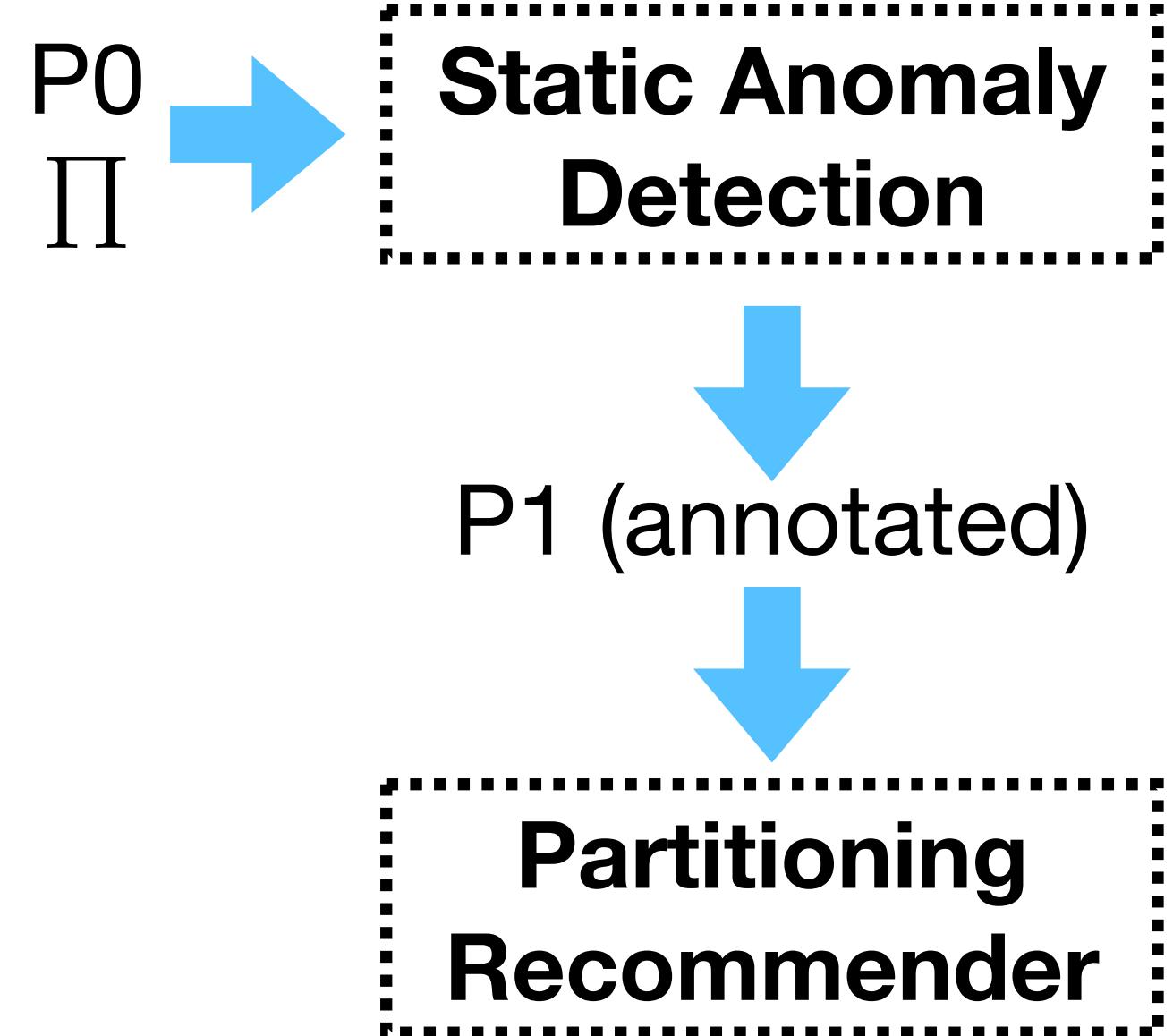
LACHESIS PIPELINE: CONTINUED

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



PARTITIONING RECOMMENDATION

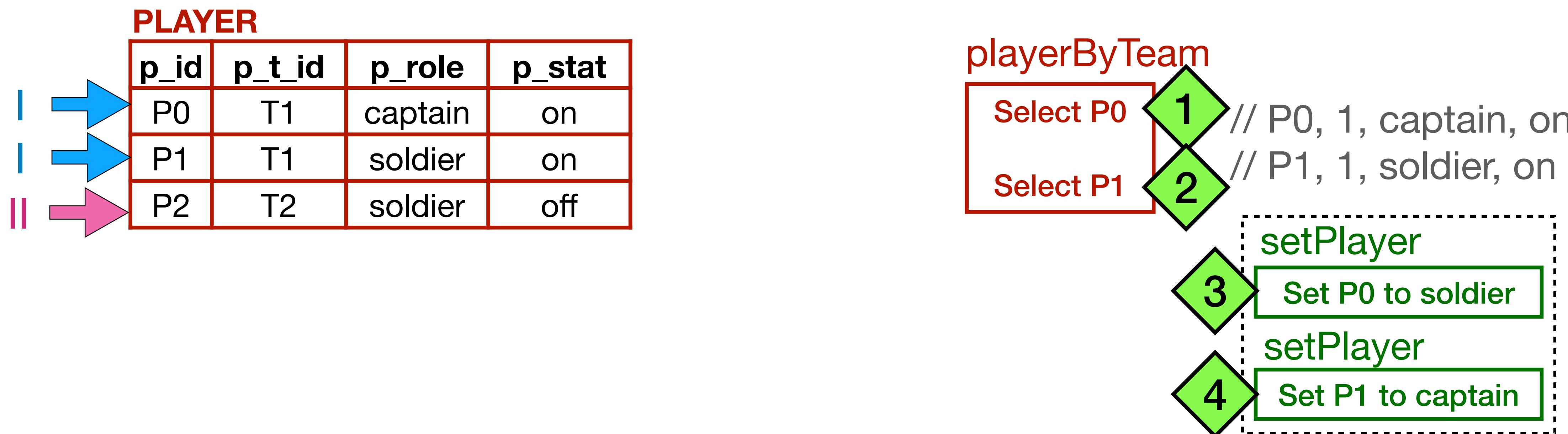
- Symbolic analysis engine
 - ▶ program P0 (transactions & schema)
 - ▶ partitioning relation
- Annotation & Runtime
 - ▶ Isolated { }
- Partitioning Recommendation
 - ▶ Static & efficient
 - ▶ Comparable to Schism



PARTITIONING RECOMMENDATION

- Recommended partitioning policy:

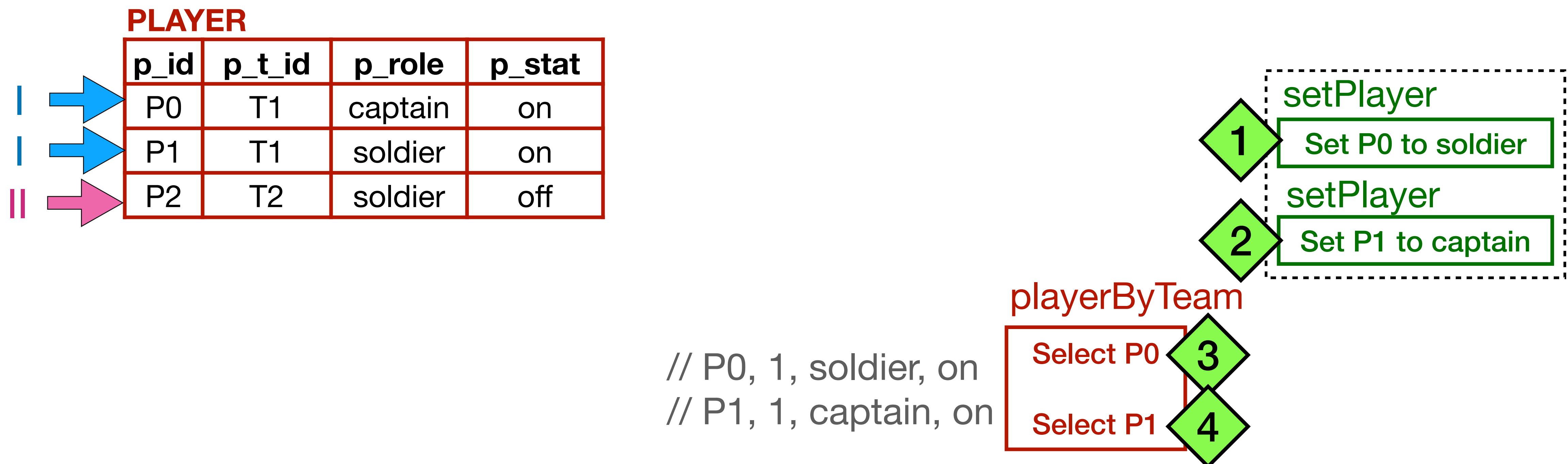
$$\forall r, r'. r.\text{p_t_id} = r'.\text{p_t_id} \Rightarrow (r, r') \in \Pi$$



PARTITIONING RECOMMENDATION

- Recommended partitioning policy:

$$\forall r, r'. r.\text{p_t_id} = r'.\text{p_t_id} \Rightarrow (r, r') \in \Pi$$



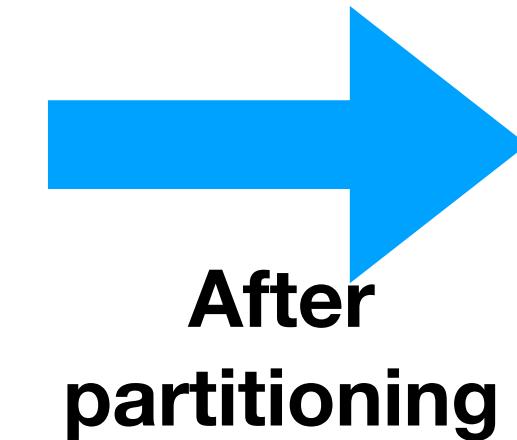
PARTITIONING RECOMMENDATION

- Recommended partitioning policy:

$$\forall r, r'. r.\text{p_t_id} = r'.\text{p_t_id} \Rightarrow (r, r') \in \Pi$$

PlayerByTeam (t)

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



PlayerByTeam (t)

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

PARTITIONING RECOMMENDATION

- Recommended partitioning policy:
 $\forall r, r'. r.p_t_id = r'.p_t_id \Rightarrow (r, r') \in \Pi$
- 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?
- $\forall r, r'. r.p_t_id = r'.p_t_id \Rightarrow (r, r') \in \Pi$

LIMITATION OF PARTITIONING

- Each table is
 - ▶ partitioned for one access pattern
- Conflicting access patterns?
- $\forall r, r'. r.p_t_id = r'.p_t_id \Rightarrow (r, r') \in \Pi$

PLAYER

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

A1 {

||

PLAYER

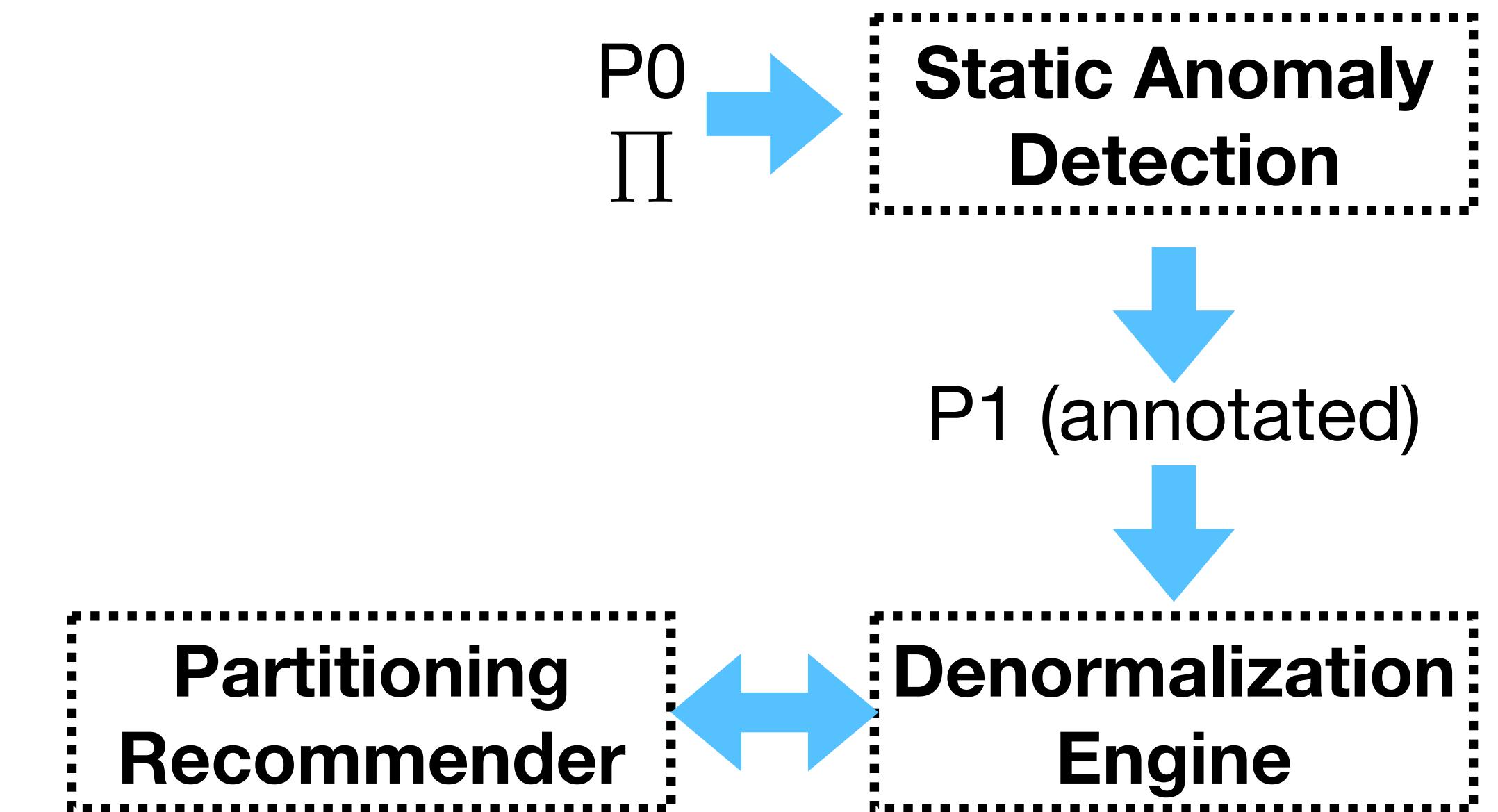
p_id	p_t_id	p_role	p_stat
P0	T1	captain	on
P1	T1	soldier	on
P2	T2	soldier	off

A2 {

||

LIMITATION OF PARTITIONING

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



EXAMPLE

- Running example:
 - ▶ additional three tables

GAME \bowtie **PLAYER**

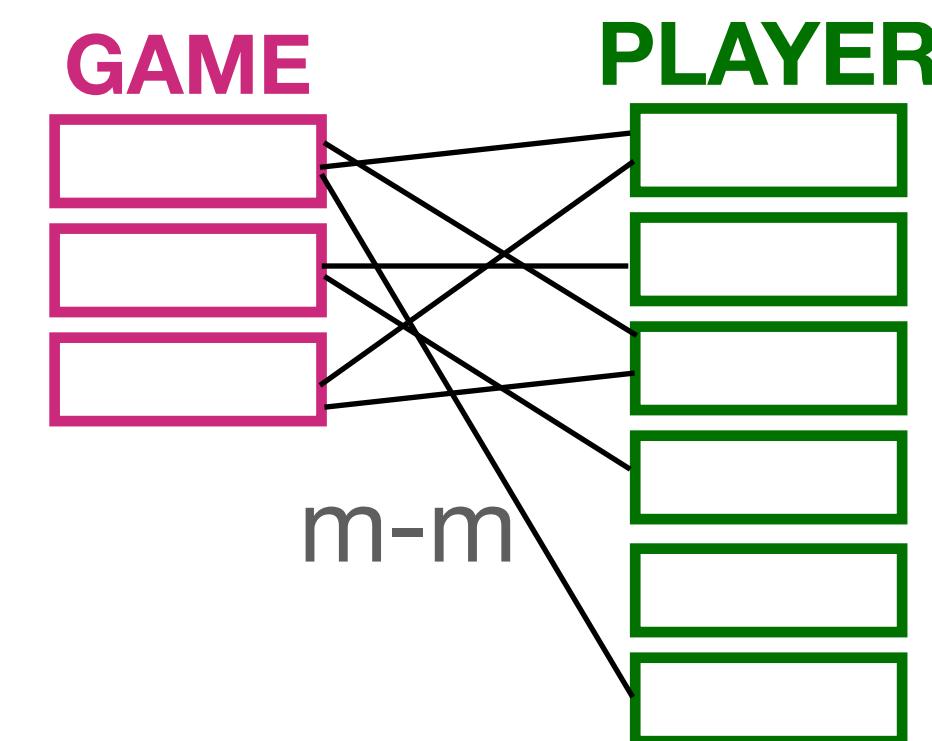
g_id	p_id
------	------

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

GAME \bowtie PLAYER

g_id	p_id
1	1

GAME

g_id	g_stat	g_min_player
1	1	1

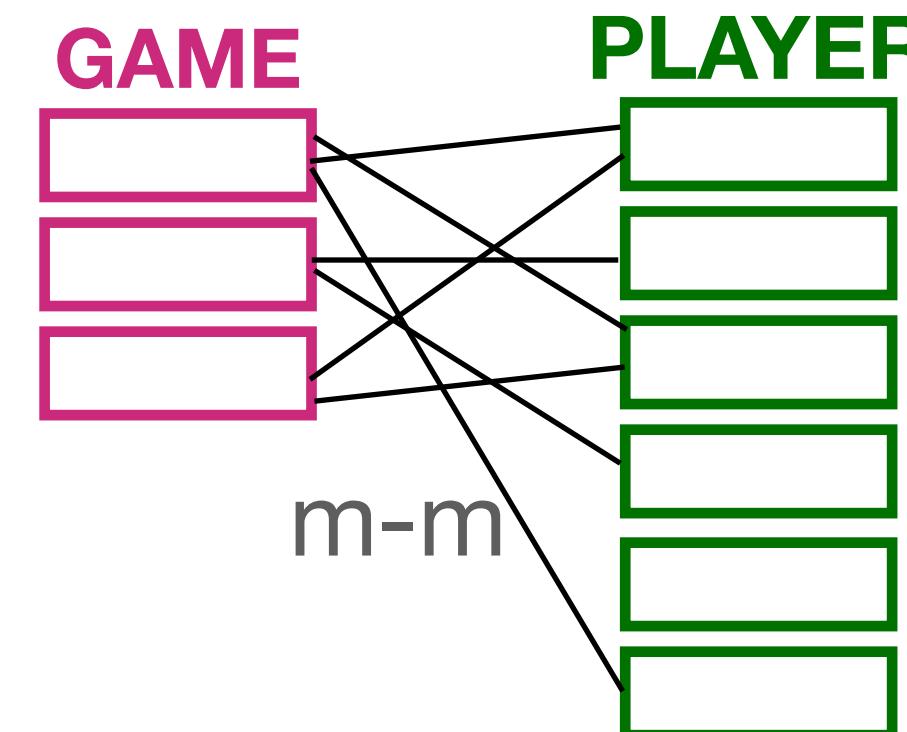
PLAYER

p_id	p_t_id	p_role	p_stat
1	1	1	1

playerByGame (g)

```
Game := select * from GAME where g_id=g
print(game.*)

ids := select p_id from GXP where g_id=g
foreach id in ids do
    p := select * from PLAYER where p_id=id
    print(p.*)
```



EXAMPLE

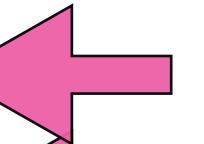
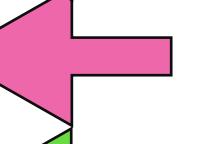
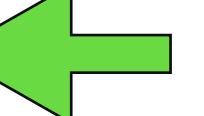
- Running example:
 - additional three tables
 - additional one transaction
- Anomaly is not fixed by partitioning

playerByGame (g)

```
Game := select * from GAME where g_id=g
print(game.*)

ids := select p_id from GXP 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

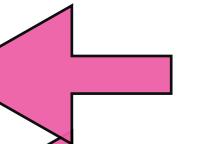
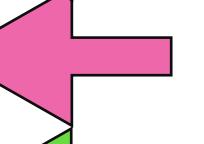
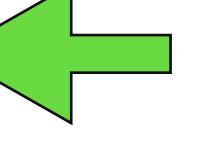
- Running example:
 - additional three tables
 - additional one transaction
- Anomaly is not fixed by partitioning

playerByGame (g)

```
Game := select * from GAME where g_id=g
print(game.*)

ids := select p_id from GXP 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

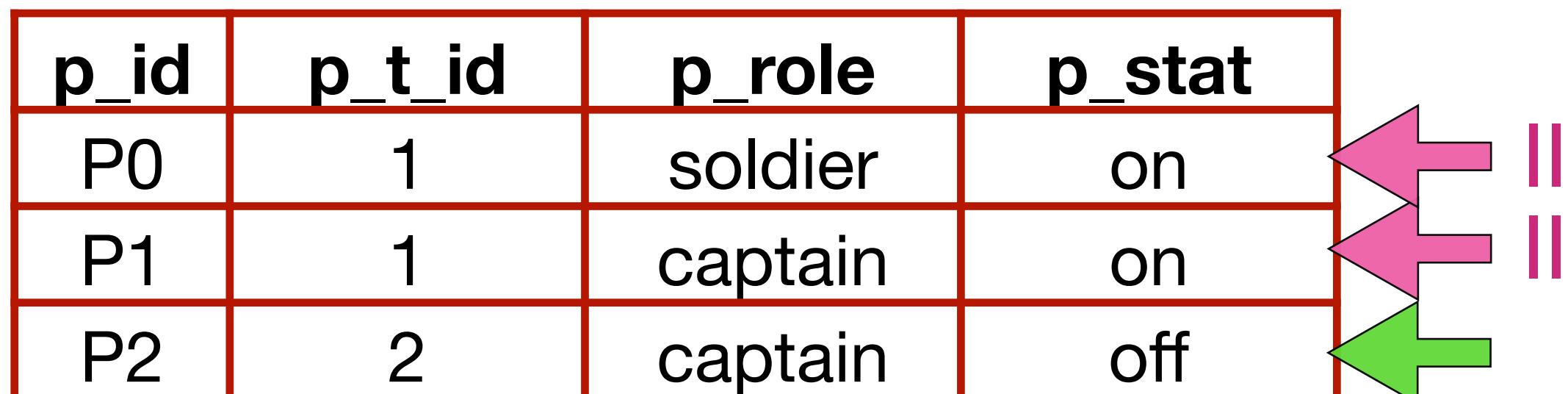
playerByGame (g)

```
Game := select * from GAME where g_id=g
print(game.*)

ids := select p_id from GXP 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

PLAYER

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

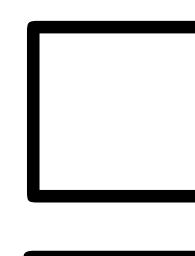
GAME

g_id	g_stat	g_min_player
G1	active	2

playerByGame (g)

```
Game := select * from GAME where g_id=g
print(game.*)

ids := select p_id from GXP where g_id=g
foreach id in ids do
    p := select * from PLAYER_BY_GAME
        where p_id=id
    print(p.*)
```

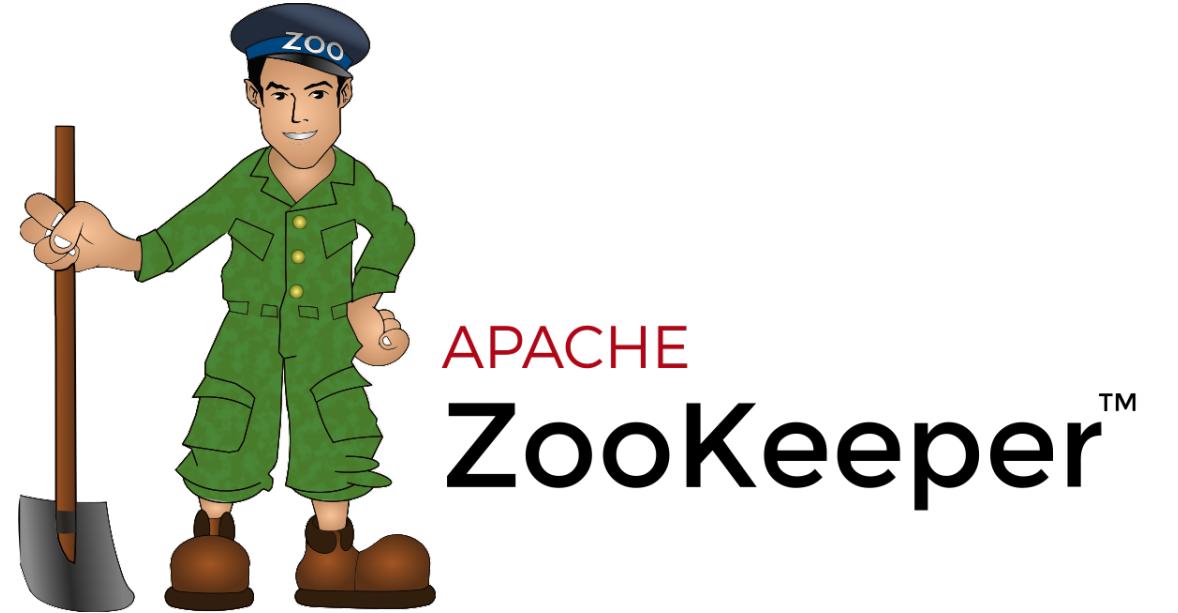


PLAYER_BY_GAME

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

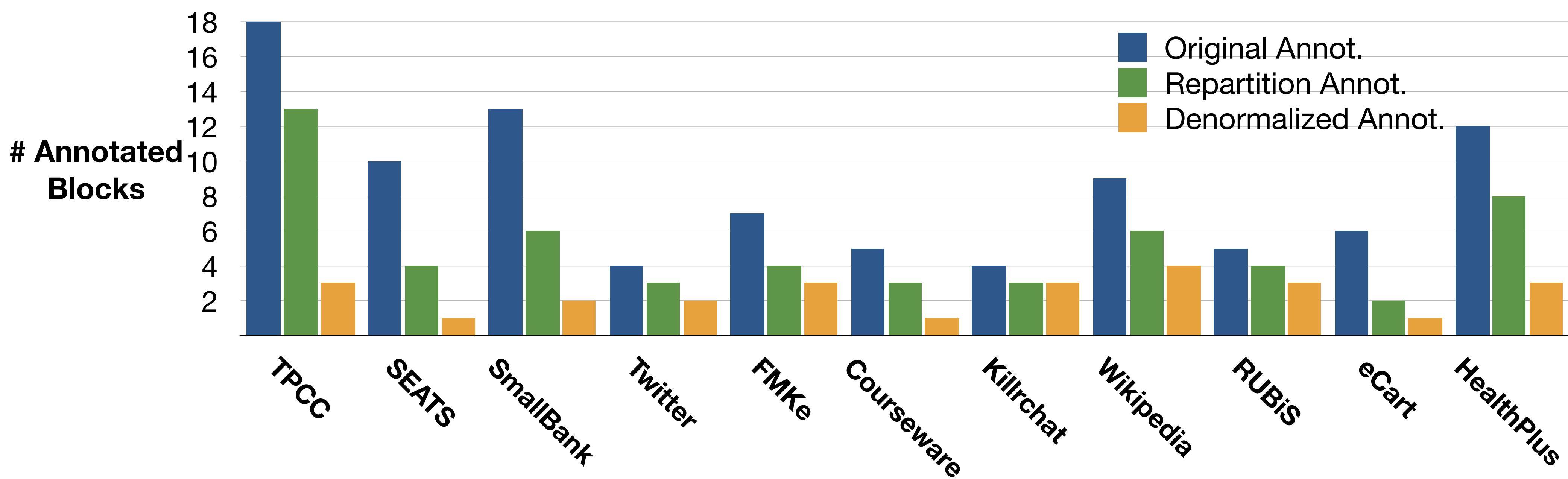
IMPLEMENTATION & EMPIRICAL RESULTS

- ZooKeeper runtime
- 11 Benchmarks



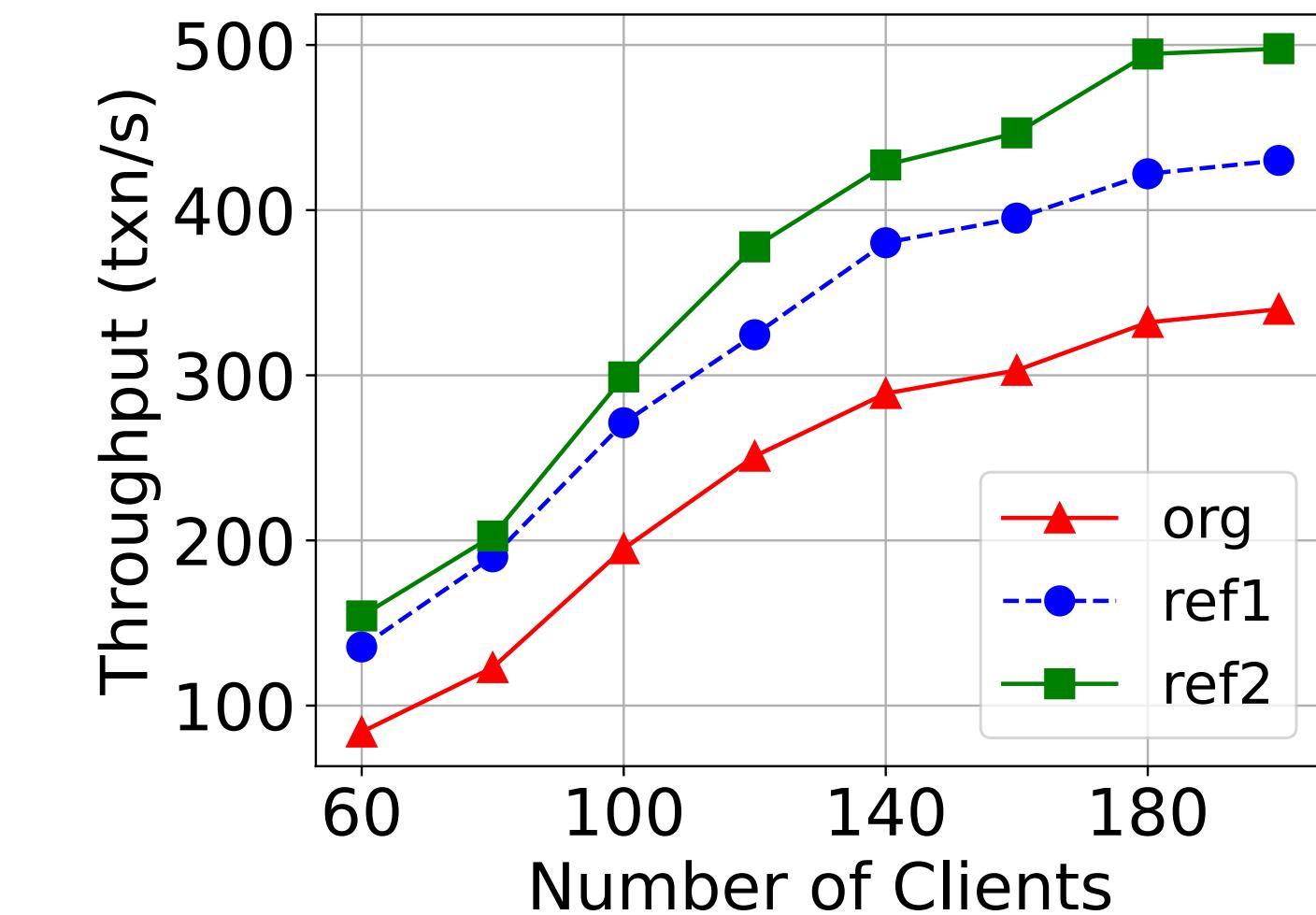
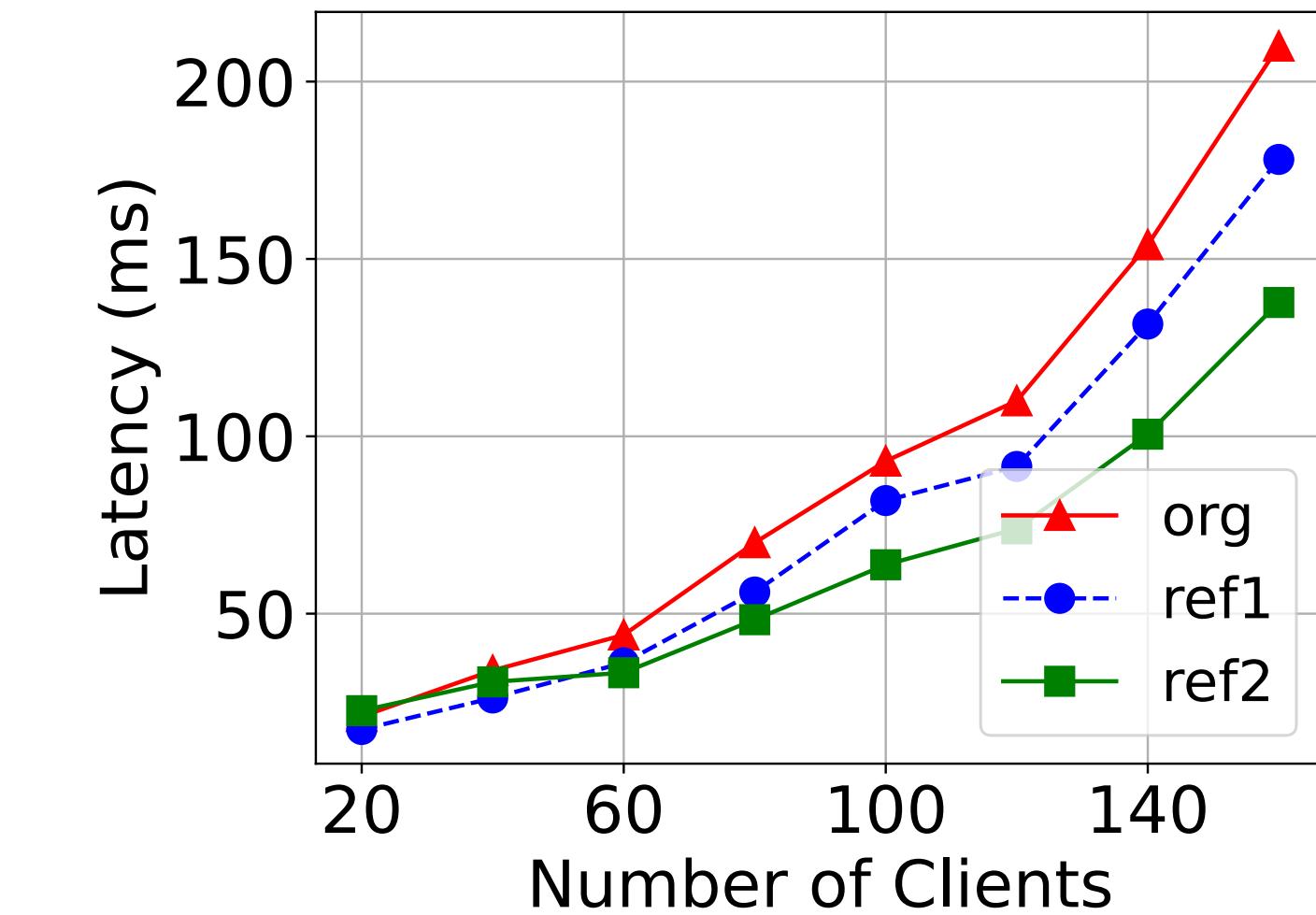
IMPLEMENTATION & EMPIRICAL RESULTS

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



IMPLEMENTATION & EMPIRICAL RESULTS

- 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