A client-centric approach to transactional datastores

NATACHA CROOKS

Laboratory for Advanced Systems Research

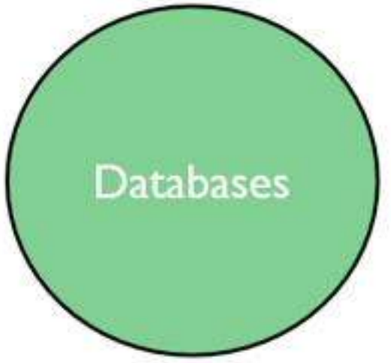
Department of Computer Science at The University of Texas at Austin

Cornell University
Computer Science

My Research

[PODC'17]

[OSDI'18, NSDI'17, OOPSLA'17, NSDI'16 Eurosys'15]



[SIGMOD'16, SIGMOD'17]

Improve semantics, performance, and security of transactional datastores

Applications are data-driven

Electronic
Health Records

Sensor Data

Consumer Purchase
Information

The promise of the cloud

Application designers domain-experts

Cloud promises simplicity

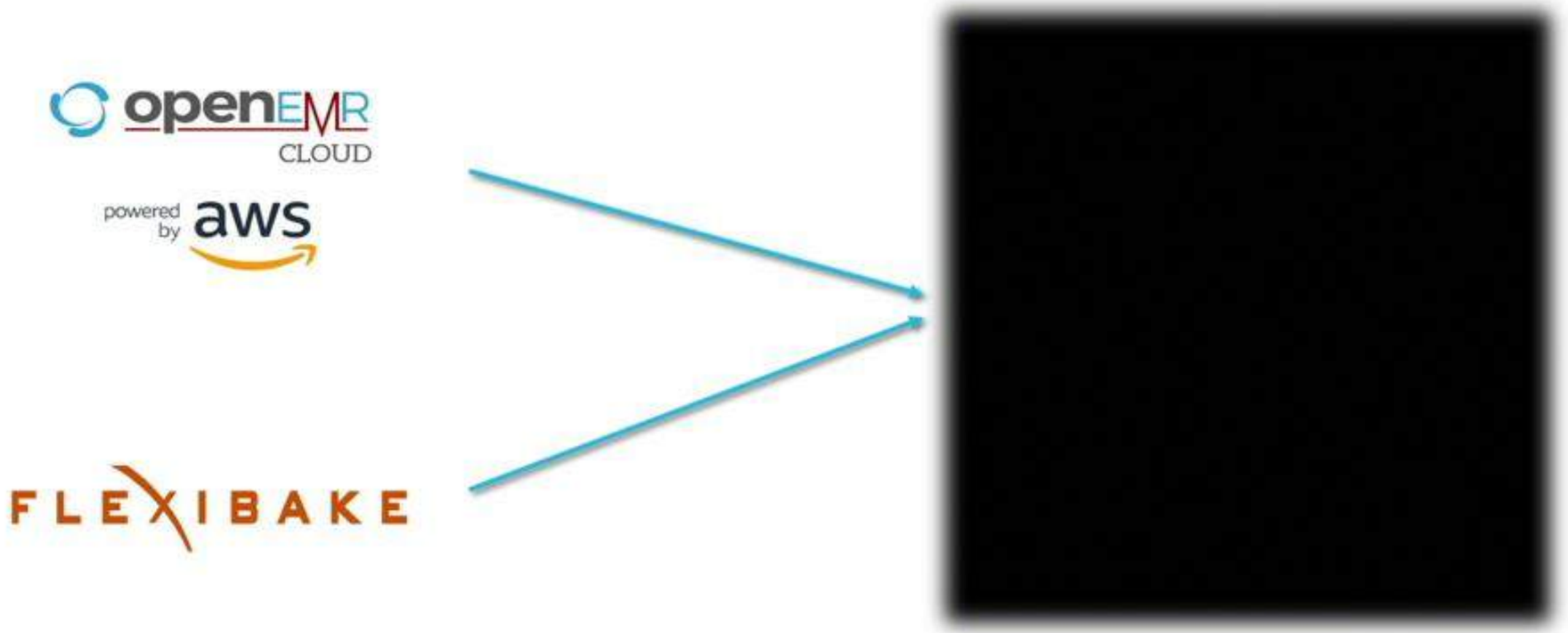
The promise of the cloud

Application designers domain-experts

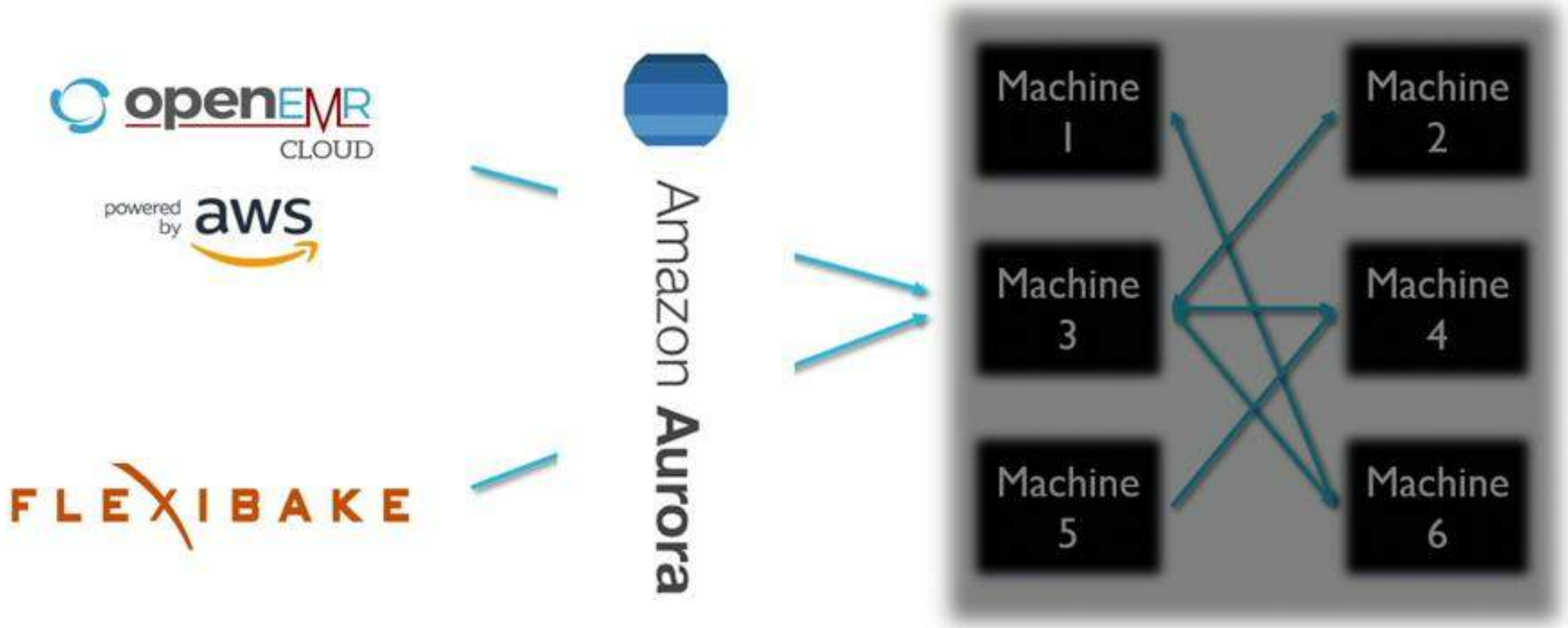
Cloud promises simplicity

Offers managed solutions to store data

The cloud as a black-box



The cloud as a black-box



The cloud as a black-box

Applications cannot observe system internals, only external state

Challenges

Correctness defined with low-level
operations that are invisible to applications

The cloud as a black-box

Applications cannot observe system internals, only external state

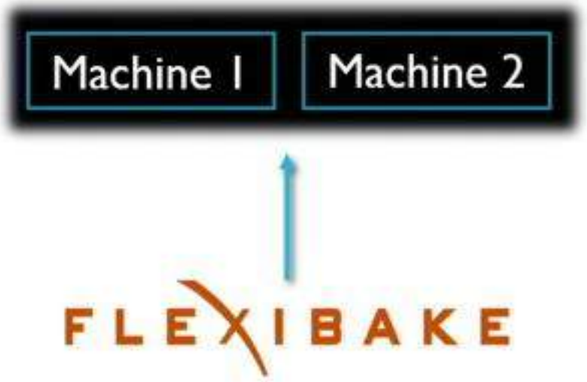
Opportunities

System internals do not need be correct, as long as external state appears correct to applications

My approach

Client-centric view
of system development

My approach



Correctness expressed from
system's view point

Should be expressed from
application's view point

The power of a client-centric view

Addressing challenges

1. Cleanly reformulate correctness guarantees
   [PODC'17]

Seizing opportunities

2. Improve performance and conflict handling of causal consistency
   [SIGMOD'16, NSDI'17]

3. Support private transactions with reasonable overheads
   [OSDI'18]

Outline

1) The promise of the cloud
2) Addressing challenges: a client-centric view of correctness
3) Seizing opportunities: Efficient oblivious transactions
4) Looking forward

A client-centric view of correctness

The problem

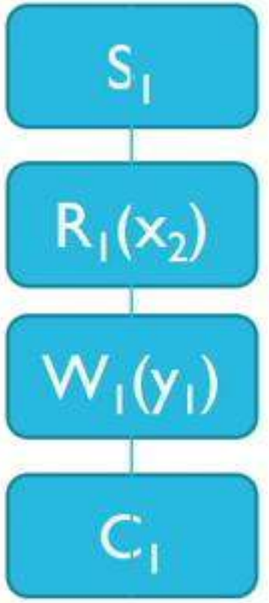
Correctness definitions are system-centric

The solution

Express correctness in terms of what clients observe

Transactions

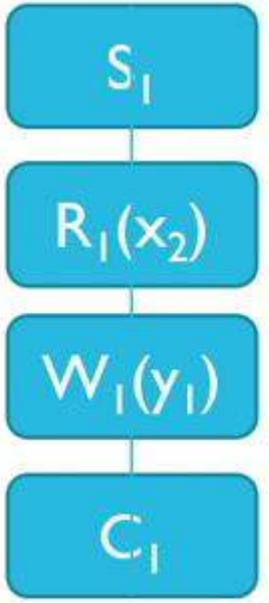
Operations in a transaction take effect atomically



Transactions

Operations in a transaction take effect atomically

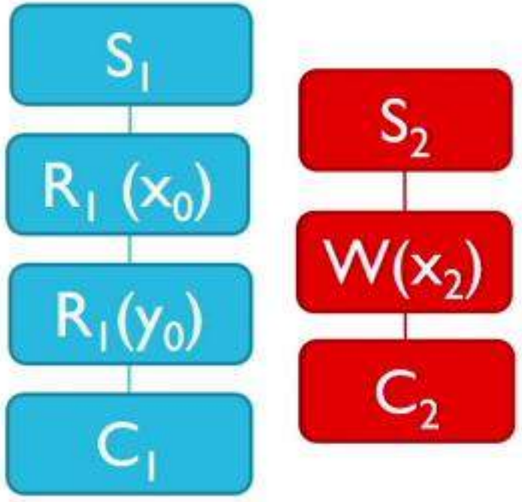
Support for transactions increases ease of programming



Correctness

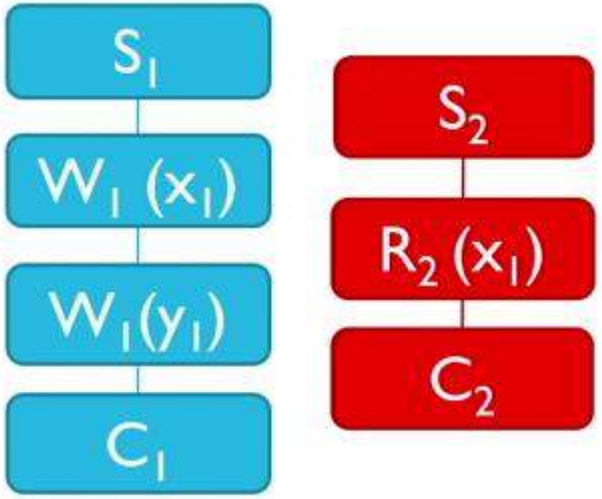
Isolation

Contract that regulates the interaction between concurrent transactions



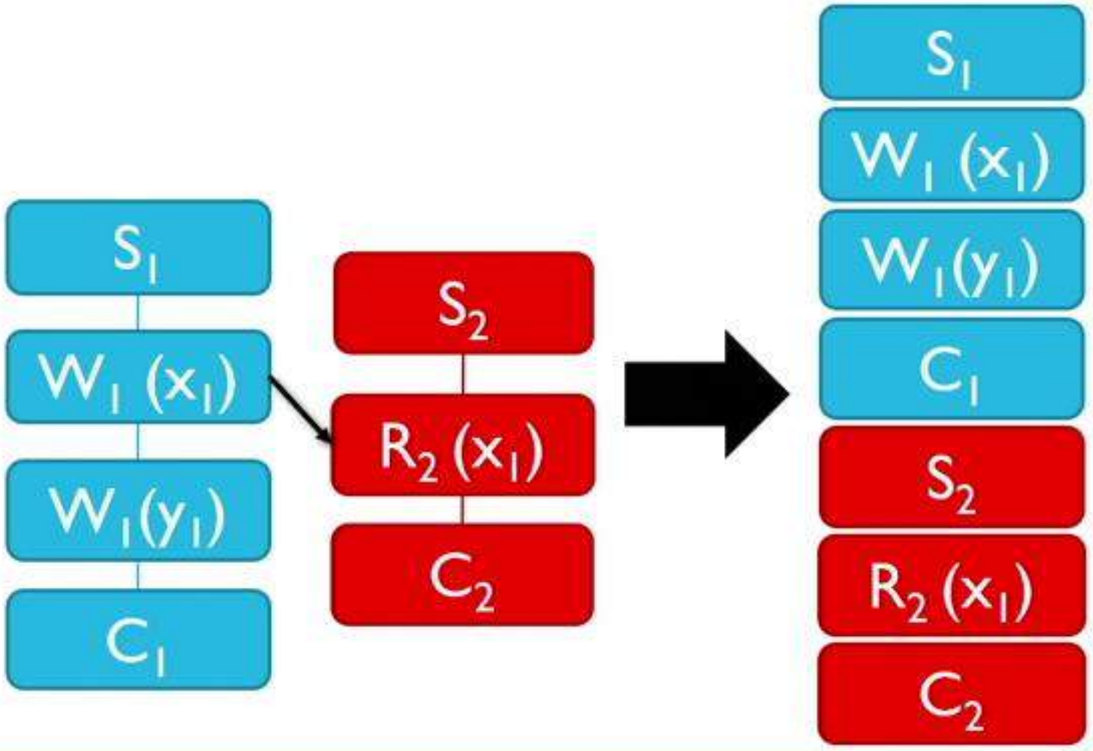
Gold standard: serializability

Execution should be equivalent to a serial schedule



Gold standard: serializability

Execution should be equivalent to a serial schedule



Easy to program

Performs poorly

Weak isolation is tricky

Relax isolation guarantees

The Good

Better scalability and performance

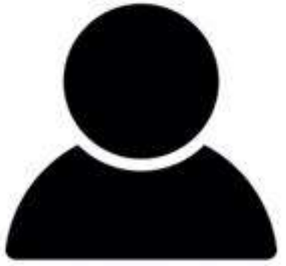
The Bad

Anomalies break application logic

The dangers of write-skew

Checkings
30

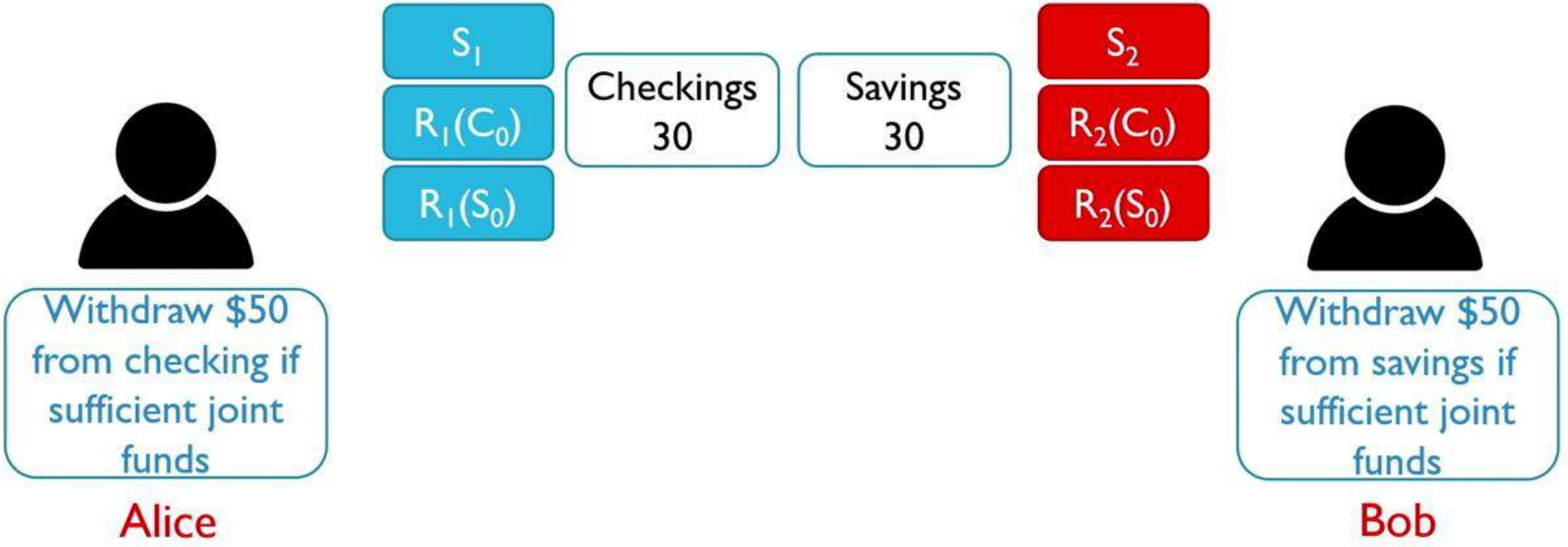
Savings
30



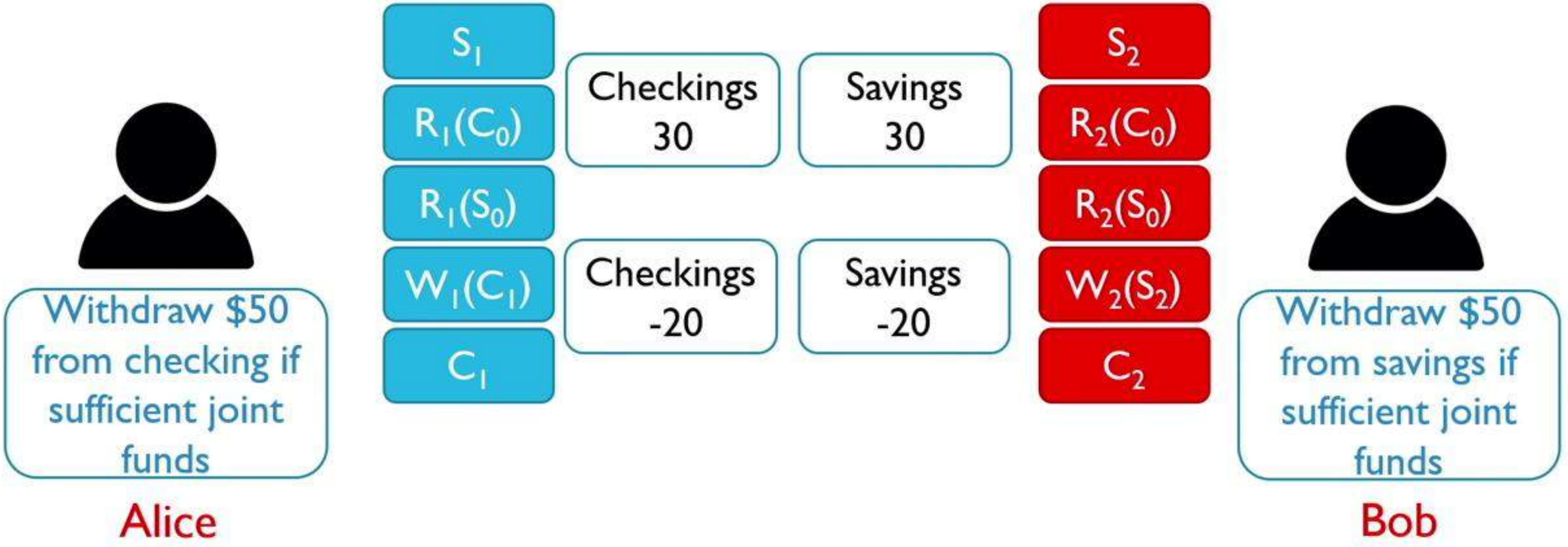
Alice

Bob

The dangers of write-skew



The dangers of write-skew



Weak isolation is still popular

Weak isolation is still popular

The quest of formalisation

1977

Degrees of Consistency
[Gray77]

Lock-based implementations

1992

ANSI SQL
[Ansi92]

More general implementations

1995

A critique of ANSI SQL
[Berenson95]

Formal treatment of ANSI SQL

2000

Cycle-based Isolation
[Adya00]

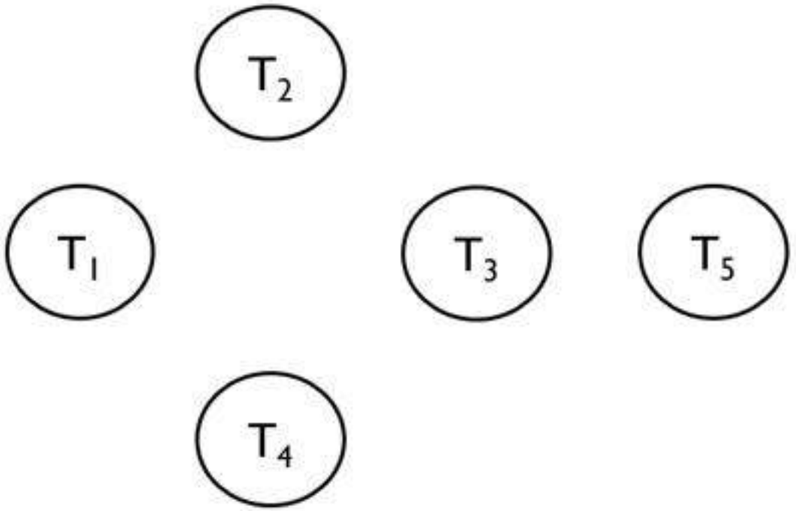
Supports optimistic implementations

Cycle-based isolation

Uses version order and timestamps to construct serialization graph

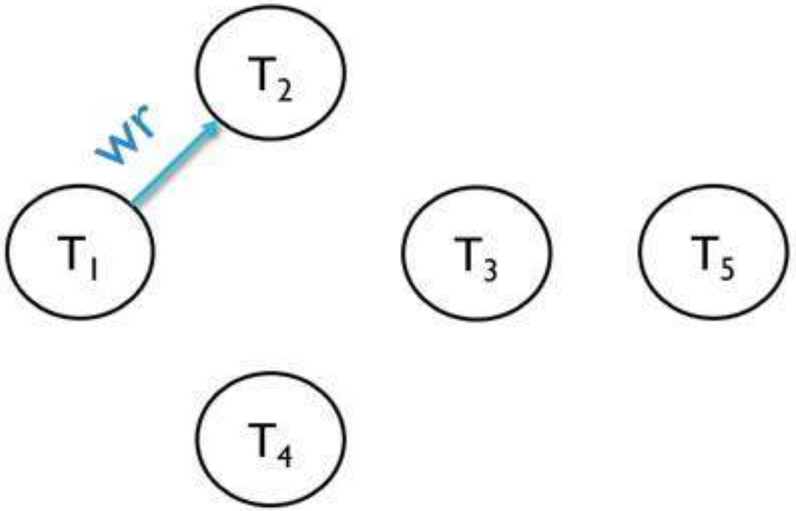
Cycle-based isolation

Uses version order and timestamps to construct serialization graph



Cycle-based isolation

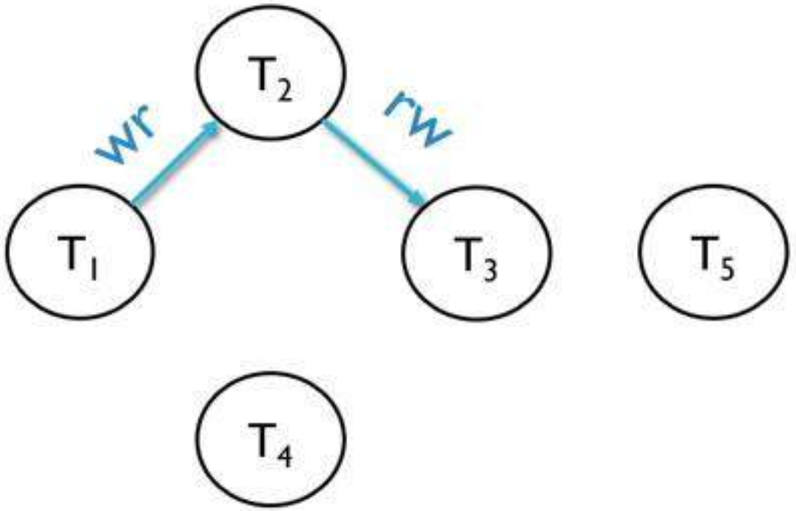
Uses version order and timestamps to construct serialization graph



- write-read (wr): T2 reads T1's write

Cycle-based isolation

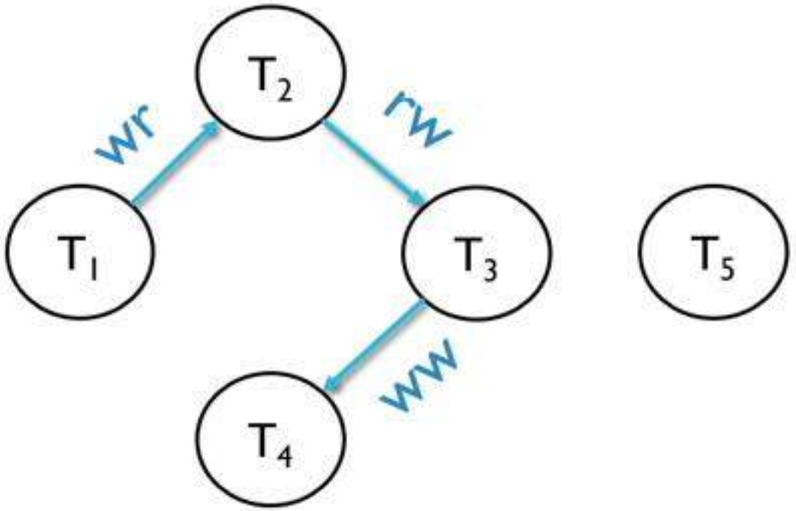
Uses version order and timestamps to construct serialization graph



- write-read (wr): T2 reads T1's write
- read-write (rw): T2 misses T3's write

Cycle-based isolation

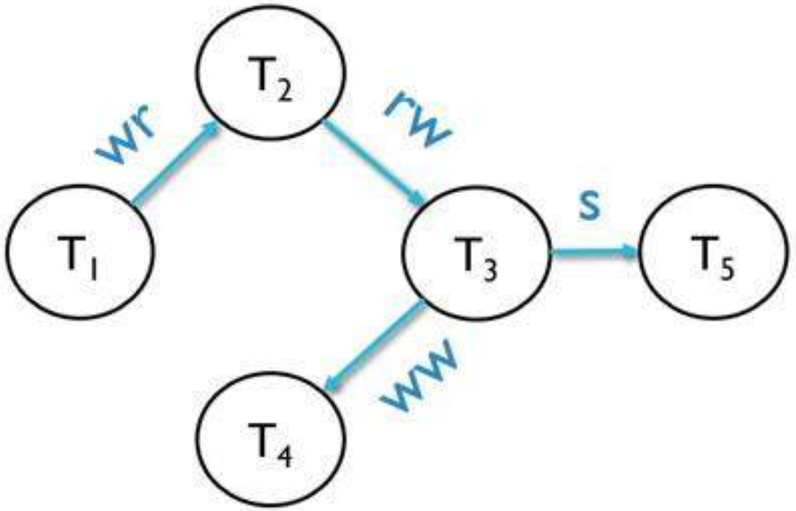
Uses version order and timestamps to construct serialization graph



- write-read (wr): T2 reads T1's write
- read-write (rw): T2 misses T3's write
- write-write (ww): T4 overwrites T3's write

Cycle-based isolation

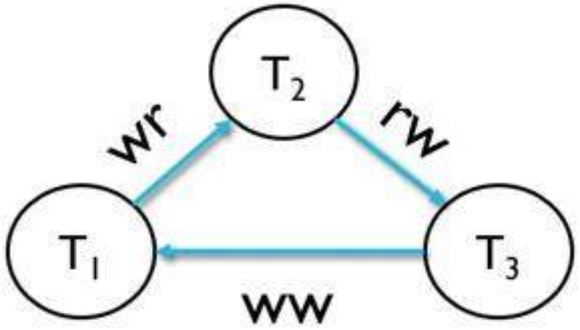
Uses version order and timestamps to construct serialization graph



- write-read (wr): T2 reads T1's write
- read-write (rw): T2 misses T3's write
- write-write (ww): T4 overwrites T3's write
- start (s): T5 starts after T3

Isolation guarantee prevents specific cycles

Serializability

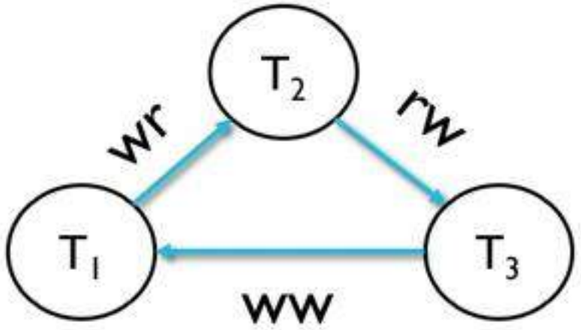


Prevents all cycles

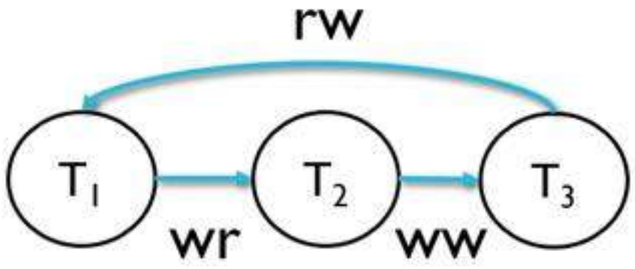
Snapshot isolation, informally

1. Transactions read from committed snapshot of system
2. Concurrent transactions do not update same object

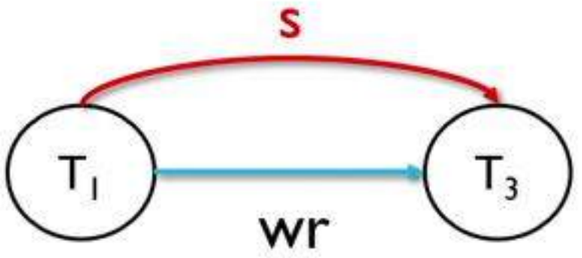
Snapshot isolation, formally



cycles of ww,wr edges



cycles of ww,wr edges
with single rw edge



wr or ww edge
without an s edge

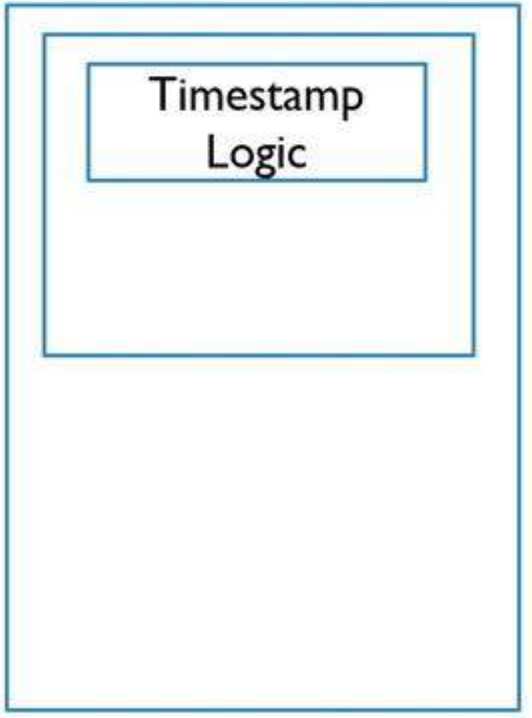
Weak isolation – The ugly

Isolation formalised using low-mechanisms

Weak isolation – The ugly

Isolation formalised using low-mechanisms

Encourages definitions that depend on system properties

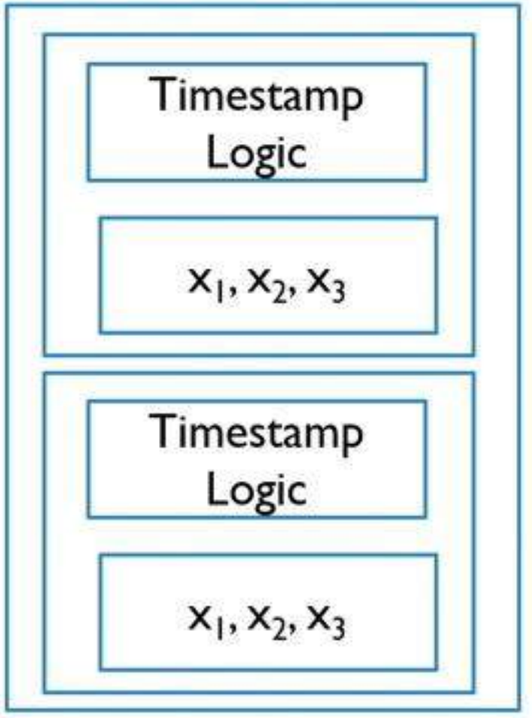


Weak isolation – The ugly

Isolation formalised using low-mechanisms

Encourages definitions that depend on system properties

18c ORACLE Database



Weak isolation – The ugly

Isolation formalised using low-mechanisms

Encourages definitions that depend on system properties 18c ORACLE Database

But system properties are hidden behind cloud-services!

The problem

Guarantees are formalised

Guarantees are observed

Using low-level mechanisms

Contract specifying what to read

Our solution: trust clients' reads

Applications only observe values returned by **reads**

Associate transactions with **read states**

State consistent with what the application observed

Introduce a per-transaction **commit test**.

Narrows down set of acceptable reads states

Read States

State consistent with what the application observed

Read States

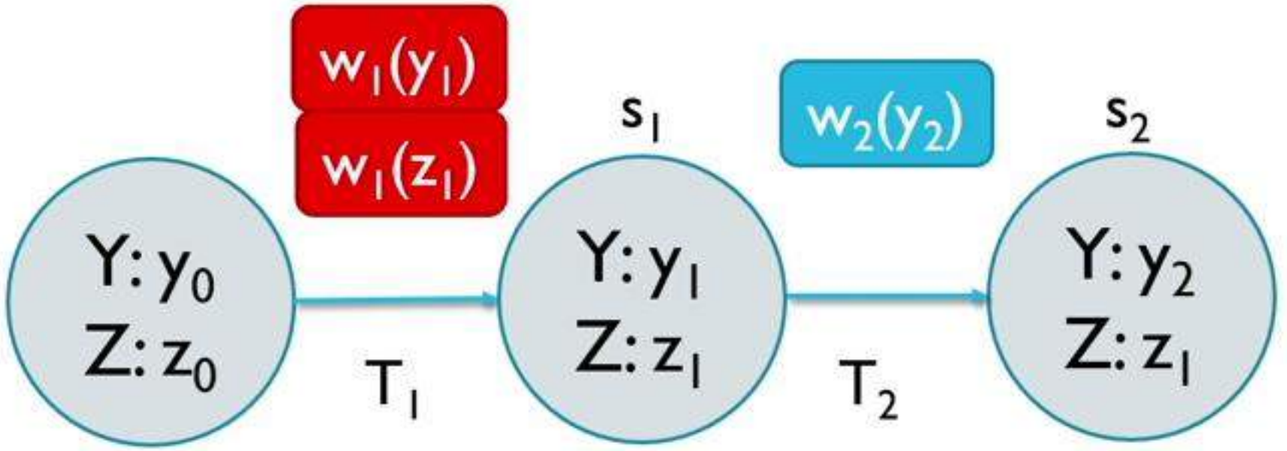
State consistent with what the application observed

Read States

State consistent with what the application observed

Our model

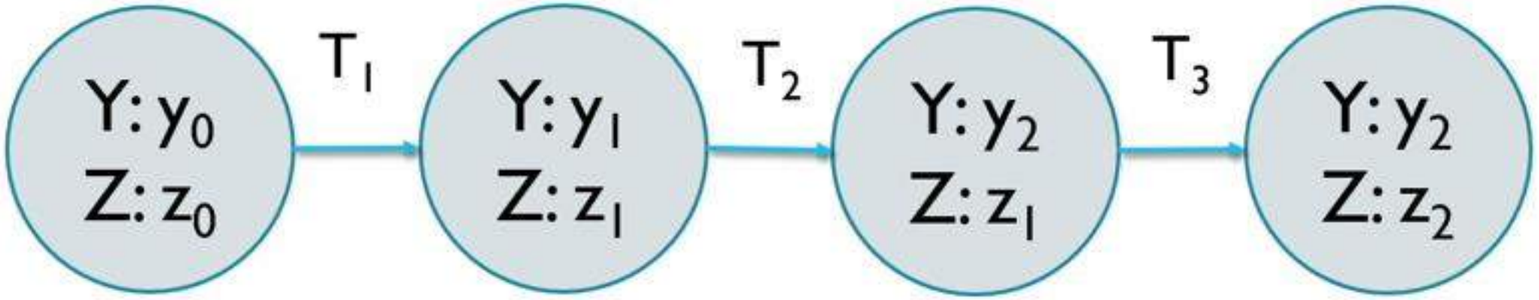
A storage system guarantees isolation level *I* if it can produce an execution that



satisfies the commit test of *I* for every transaction

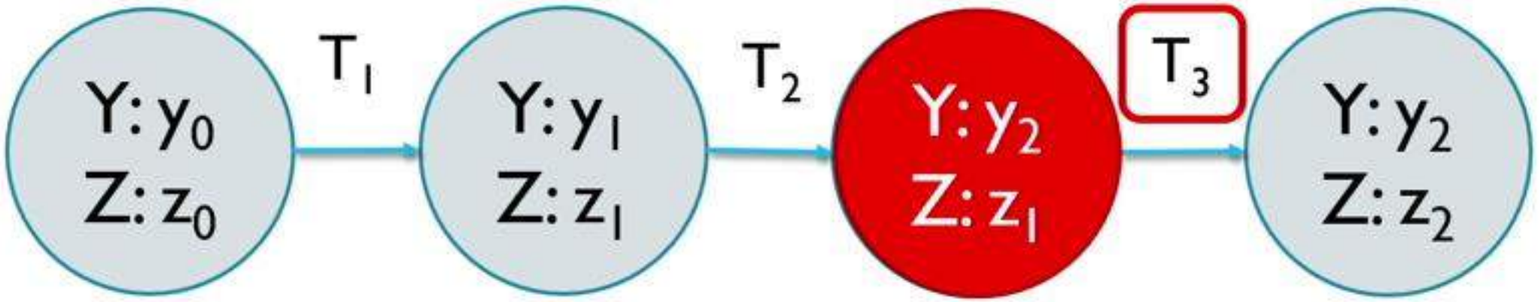
Parent State

State that precedes the state that T creates



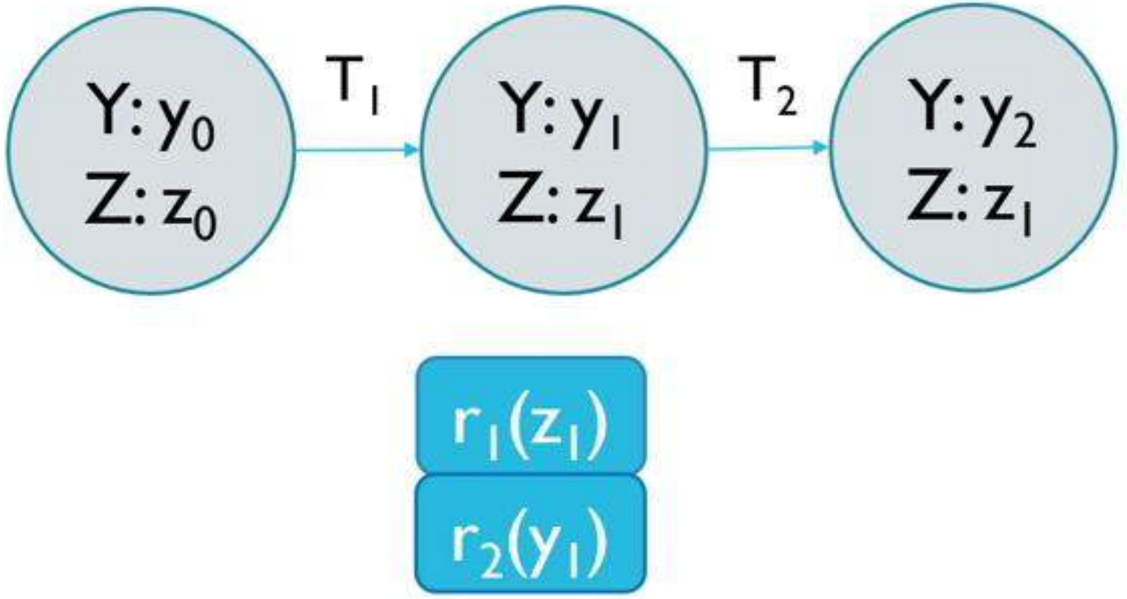
Parent State

State that precedes the state that T creates



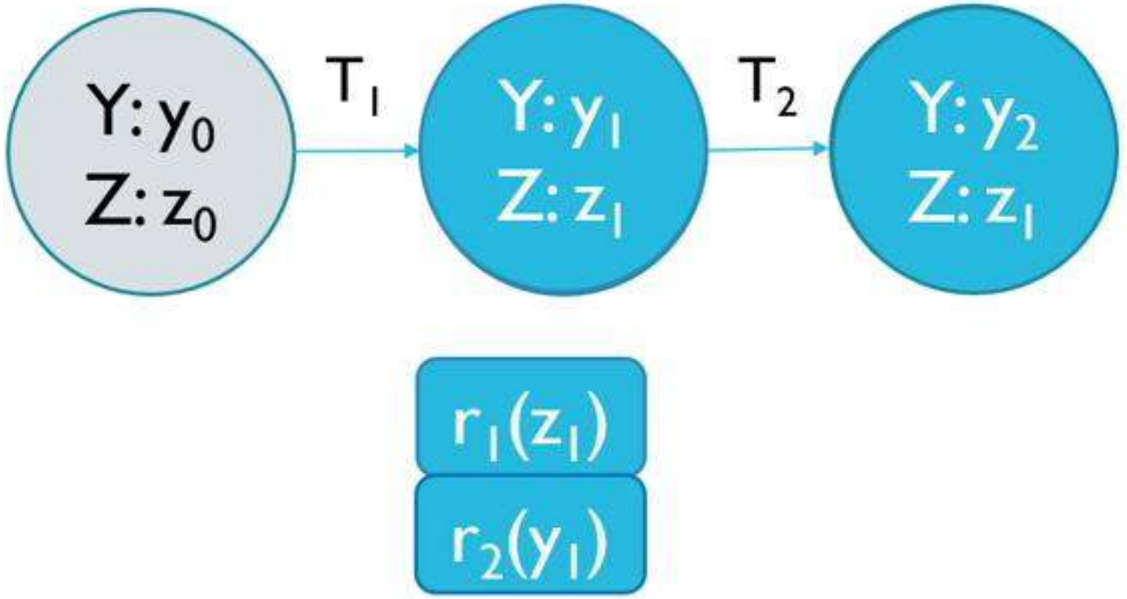
Complete State

A read state for all operations in T



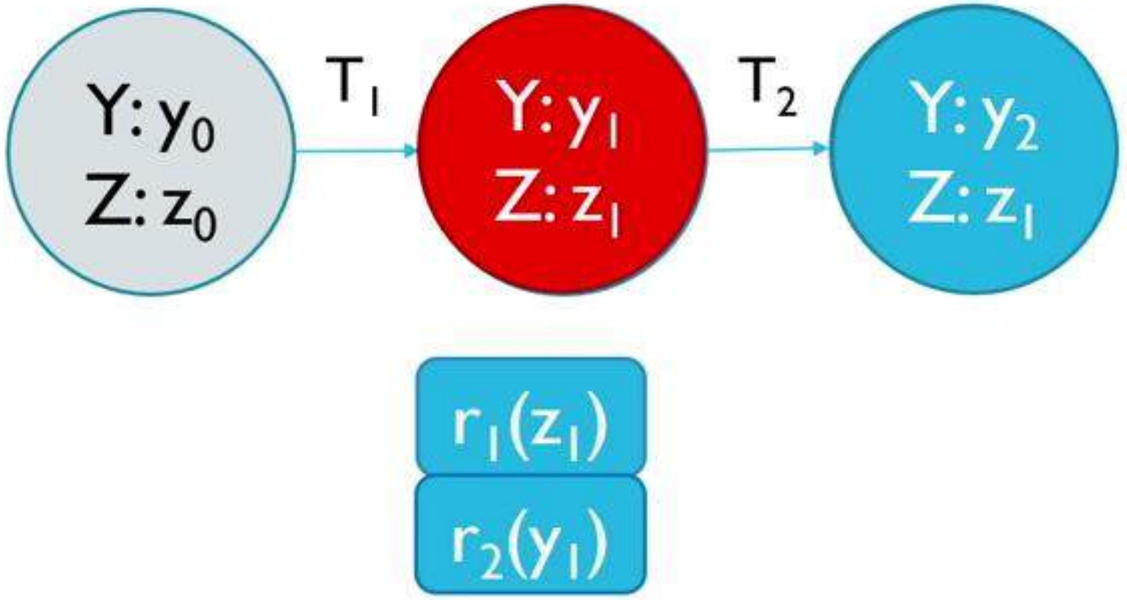
Complete State

A read state for all operations in T



Complete State

A read state for all operations in T



Commit Tests

Serializability

Parent state $\mathbf{s_p}$ of T is complete

COMPLETE(s_p)

Snapshot Isolation

Commit Tests

Serializability

Parent state $\mathbf{s_p}$ of T is complete

COMPLETE(s_p)

Snapshot Isolation

Exists complete state **s** for T

$\exists$s : COMPLETE(s))

Commit Tests

Serializability

Parent state $\mathbf{s_p}$ of T is complete

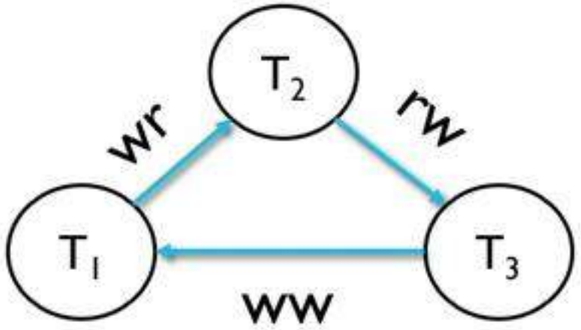
COMPLETE(s_p)

Snapshot Isolation

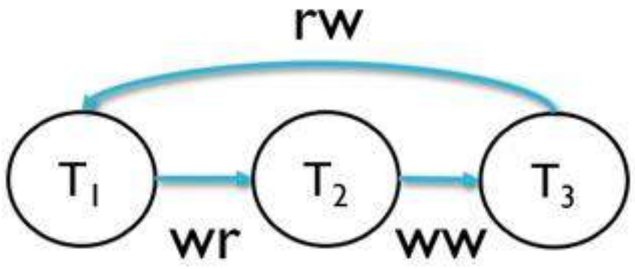
Exists complete state **s** for T
No conflicting writes since **s**

$\exists$s : COMPLETE(s)) & NO-CONF(s)

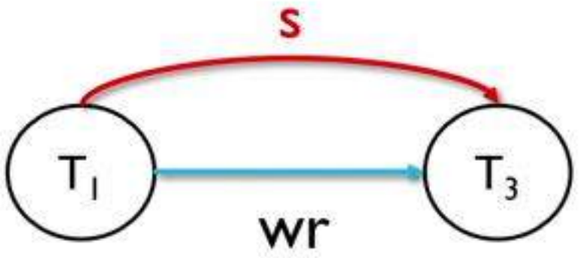
System-centric Approach [Adya00]



cycles of ww,wr edges

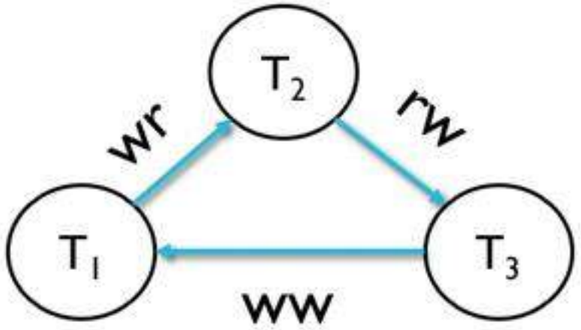


cycles of ww,wr edges with single rw edge

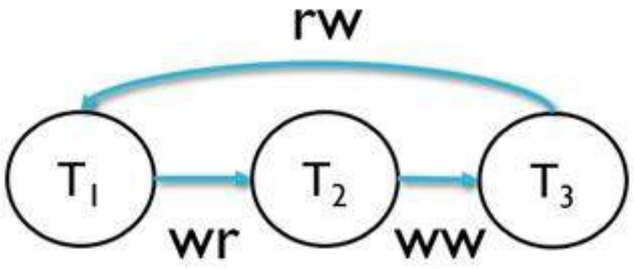


wr or ww edge without an s edge

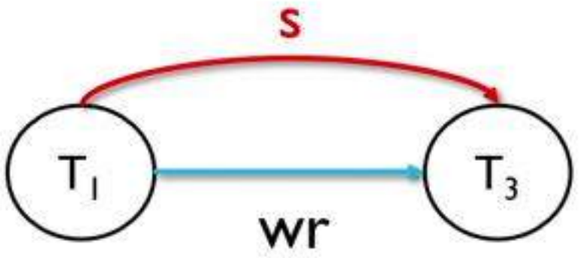
System-centric Approach [Adya00]



cycles of ww,wr edges



cycles of ww,wr edges with single rw edge



wr or ww edge without an s edge

The many snapshot isolations

PC-SI

Strong SI

ANSI SI

Adya SI

GSI

PL-2+

Strong Session SI

PSI

Clarity for snapshot isolation

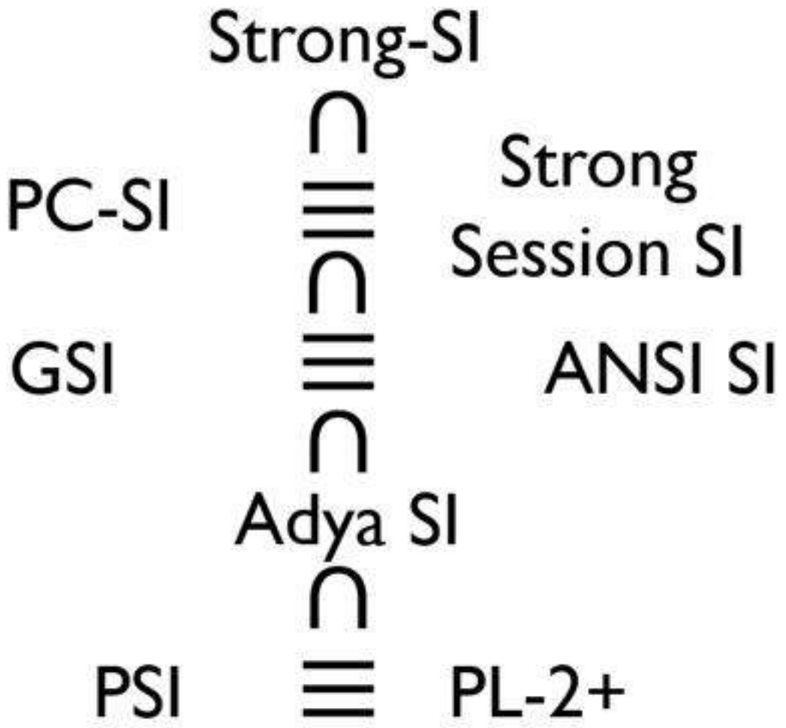
PC-SI ≡ Strong Session SI

GSI ≡ ANSI SI

PSI ≡ PL-2+

Highlight similarities when differences caused by implementation artefacts

Clarity for snapshot isolation



Highlight similarities when differences caused by implementation artefacts

Possible to express differences when they exist

Clean hierarchy of snapshot-based guarantees

Outline

1) The promise of the cloud
2) Addressing challenges: a client-centric view of correctness
3) Seizing opportunities: Efficient oblivious transactions
4) What's next?

Performance opportunities

Client-centric view allows implementations to "cheat"

System-centric definitions require implementing a correct datastore

Client-centric definitions require observing a correct datastore

[SIGMOD'16, NSDI'17, OSDI'18]

Performance opportunities

Client-centric view allows implementations to "cheat"

System-centric definitions require implementing a correct datastore

Client-centric definitions require observing a correct datastore

[SIGMOD'16, NSDI'17, OSDI'18]

Obladi: efficient oblivious transactions

Supports serializable transactions
while hiding access patterns from the cloud
with reasonable performance

(hiding what, when and how data is accessed)

Our secret sauce

System-centric view

Enforce serializability when transaction commits

Client-centric view

Enforce serializability when clients observe that transaction committed

Efficient oblivious transactions

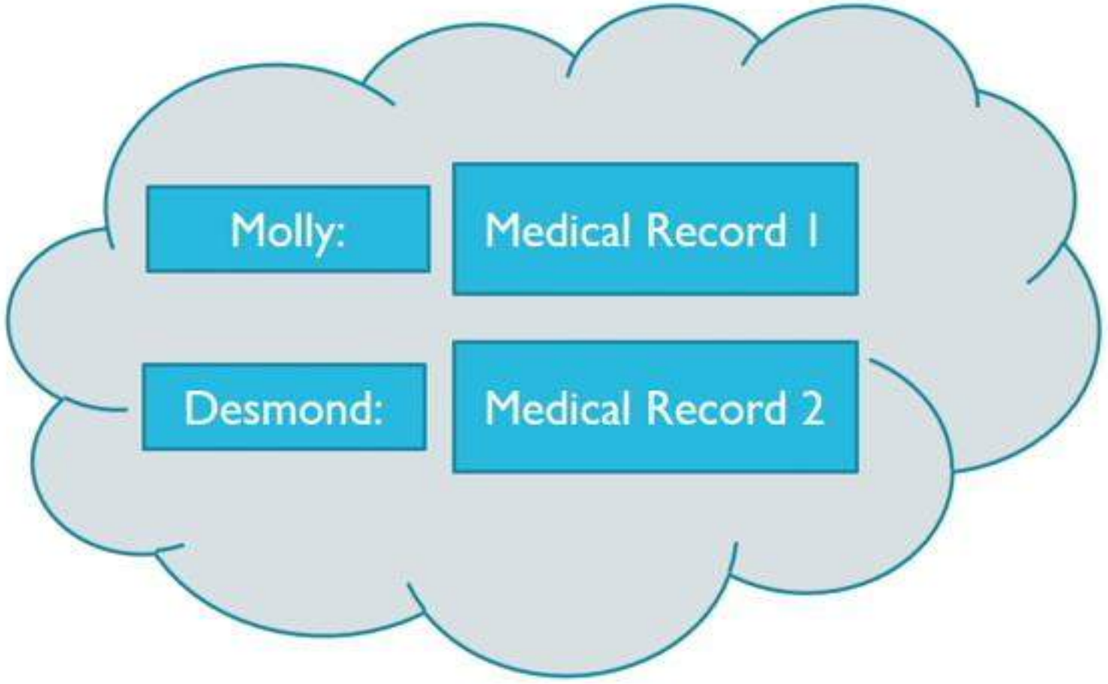
Datastore enforces serializability only
when clients observe that a transaction has committed

Serializable transactions

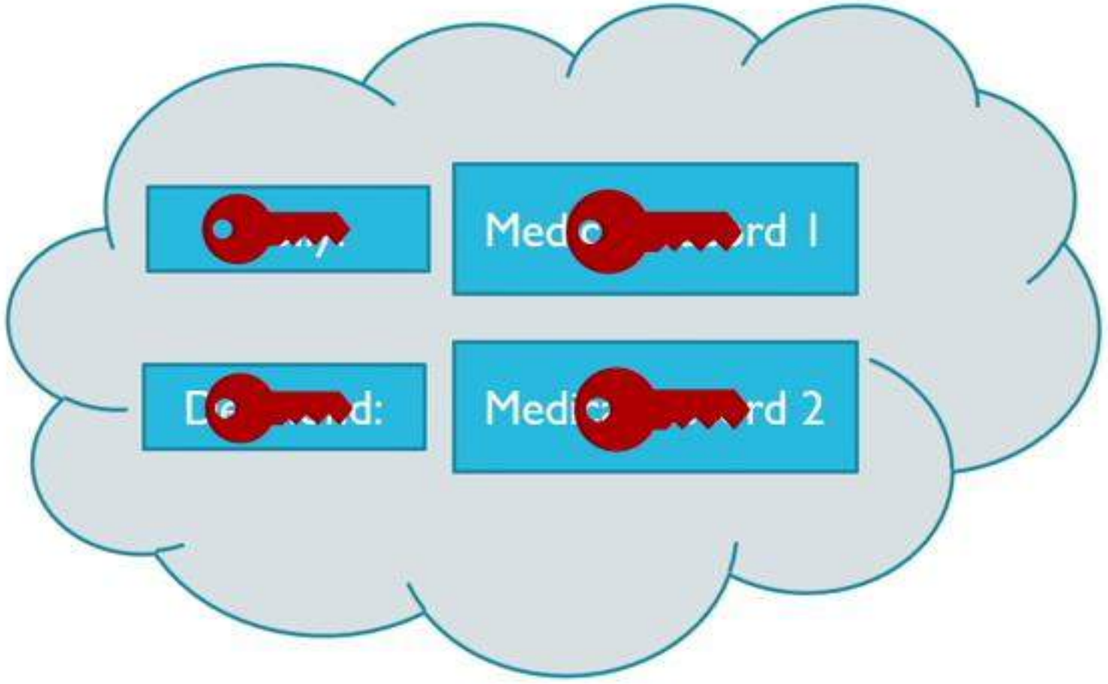
Guarantee durability

Amortise overheads

Encryption is not sufficient

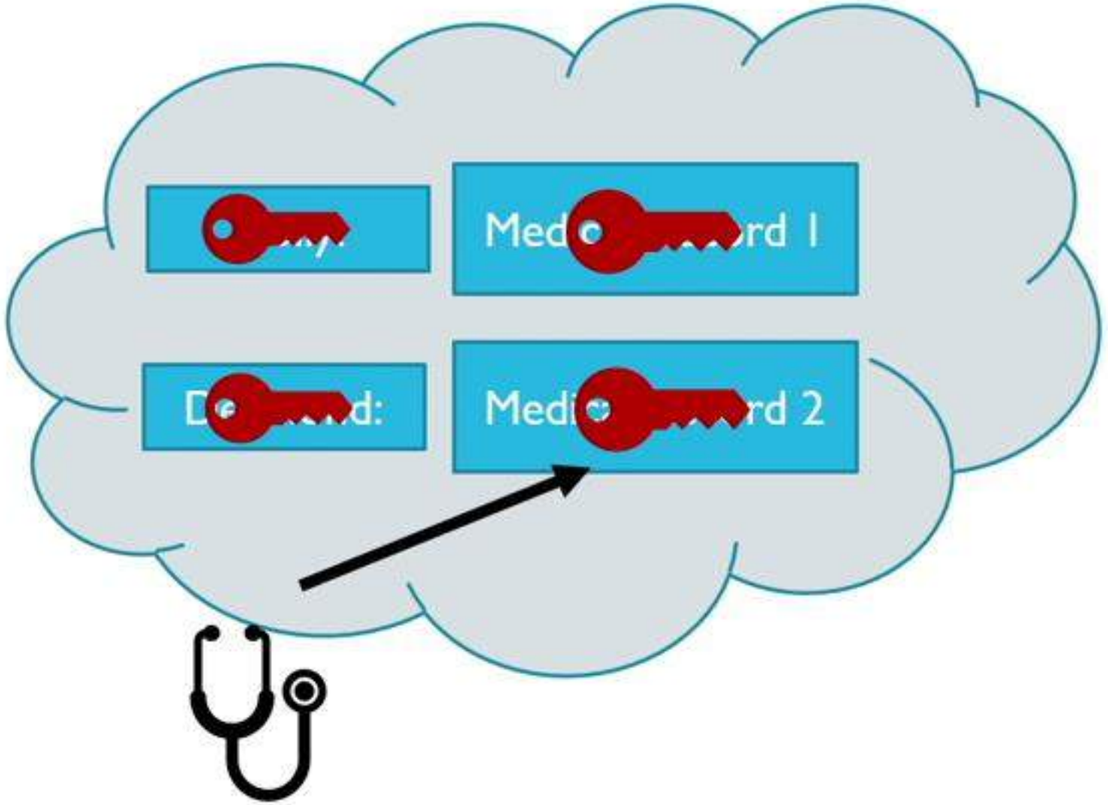


Encryption is not sufficient



Encryption only hides
contents of data

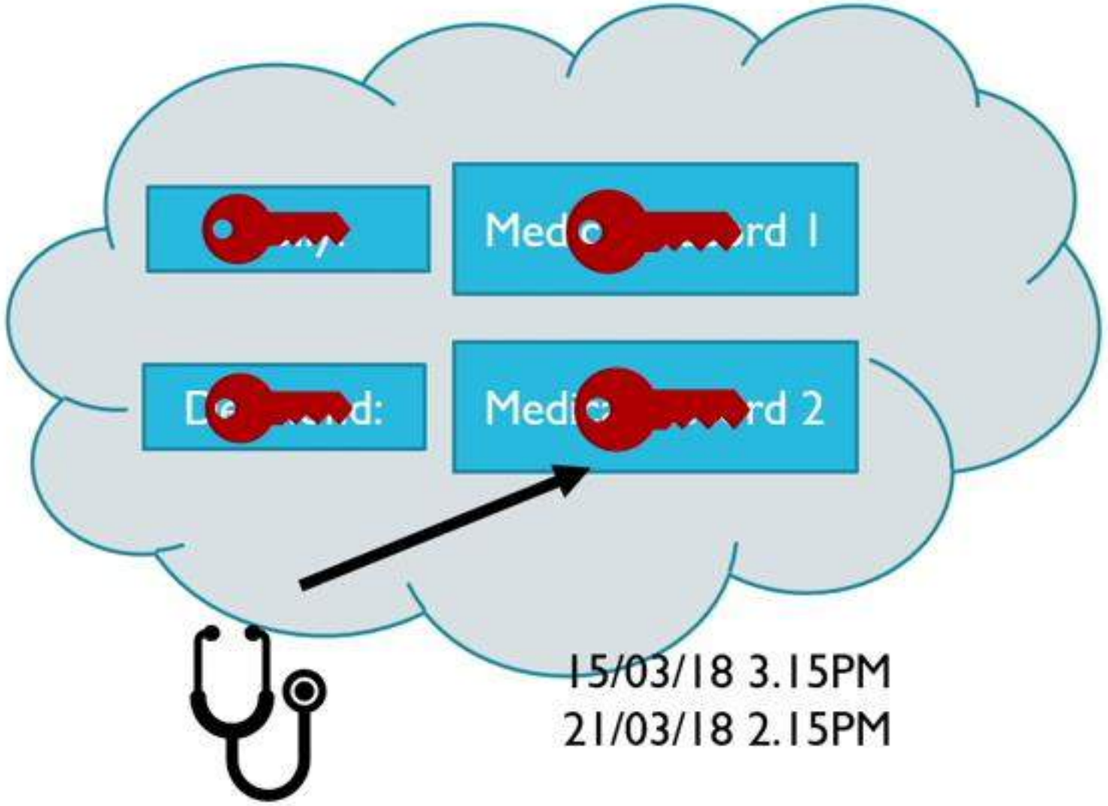
Encryption is not sufficient



Encryption only hides
contents of data

Leak information about
what data is accessed

Encryption is not sufficient

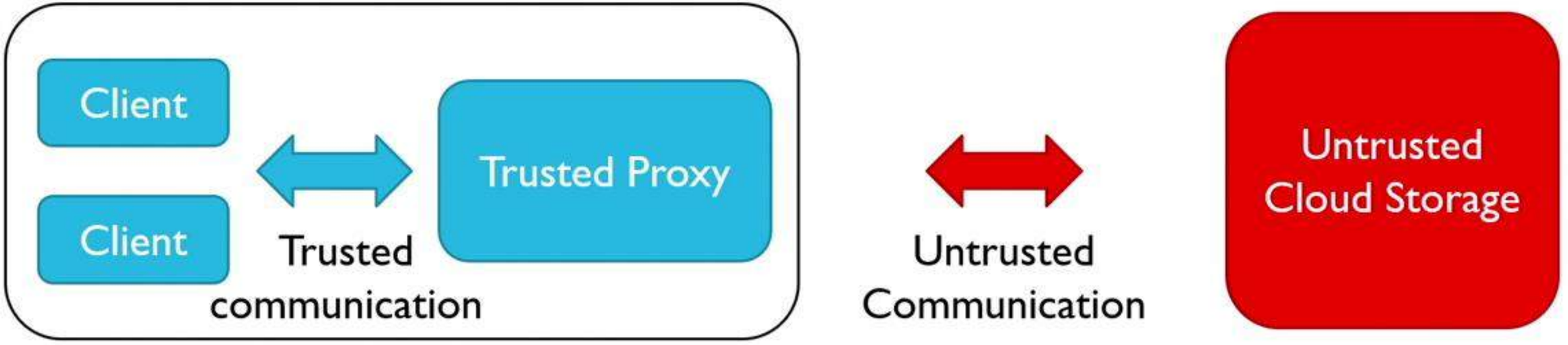


Encryption only hides contents of data

Leak information about what data is accessed

Leak information about when data is accessed

Threat Model: Trusted Proxy

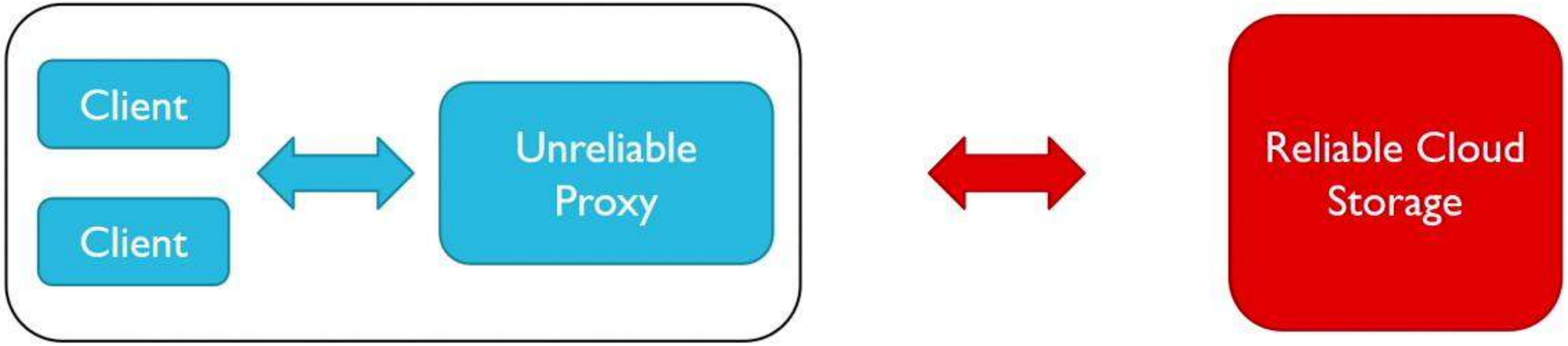


Doctors communicating over hospital LAN

Cloud storage accessed over WAN

Failure Model

Assume clients and proxy can fail



But that cloud storage is reliable

Workload Independence

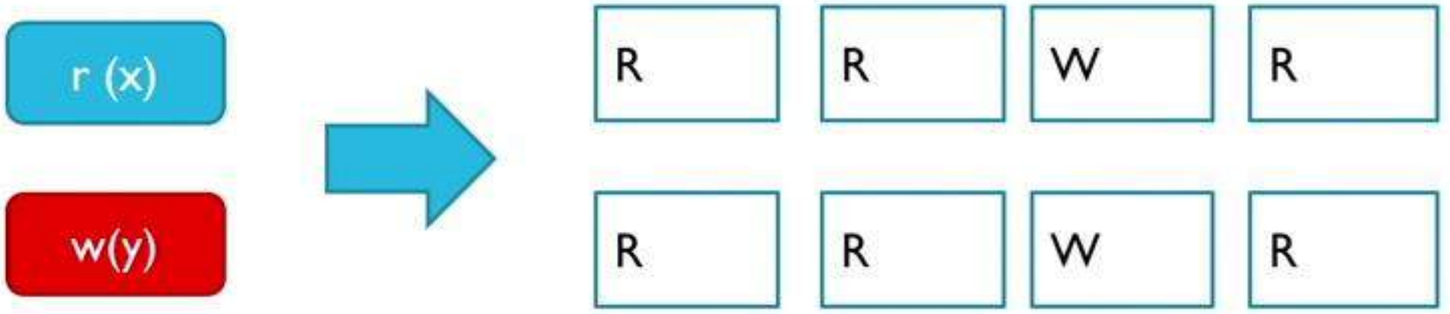
The request pattern sent to the untrusted cloud should be independent of ongoing transactions

Oblivious RAM [Goldreich1996]

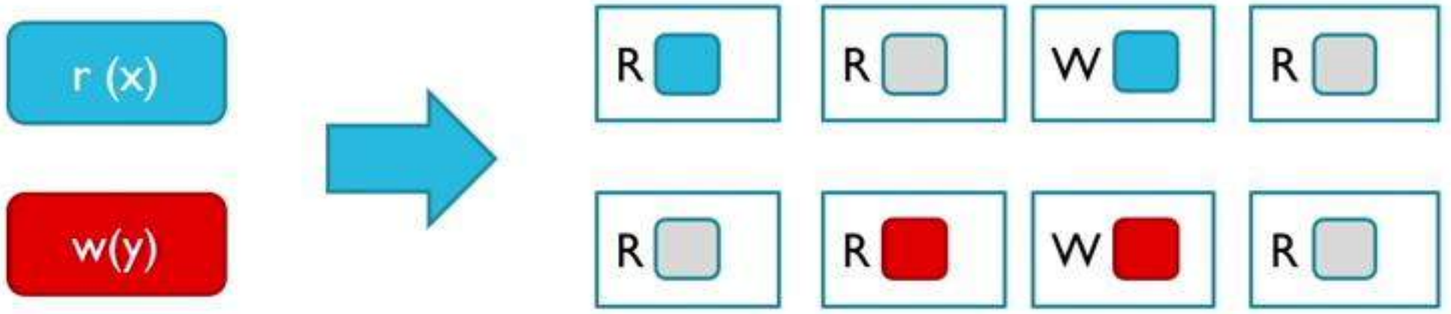
ORAM hides access patterns for
read and write operations

Ensures requests to untrusted storage are
independent of workload

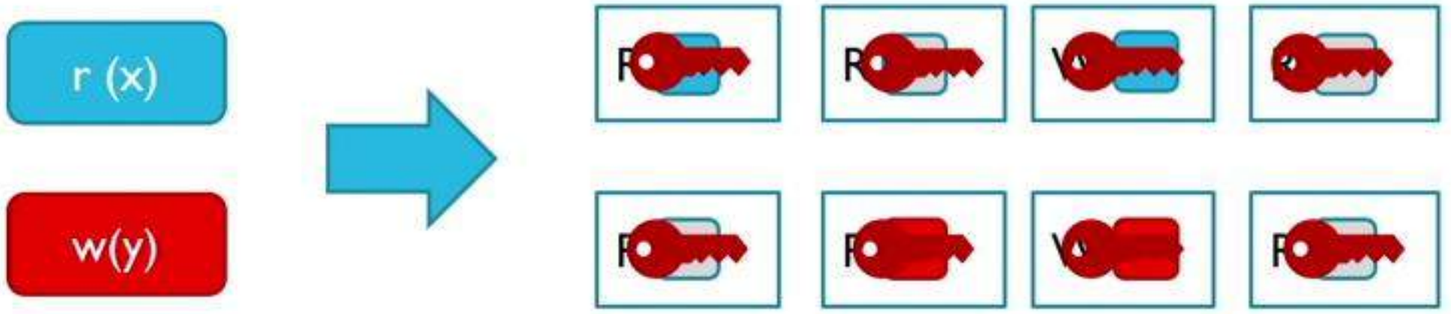
ORAM from 1000 feet



ORAM from 1000 feet



ORAM from 1000 feet



Challenges of Transactional ORAM

How can we preserve workload independence while guaranteeing

Isolation

Durability

Performance

Client-centric delayed visibility

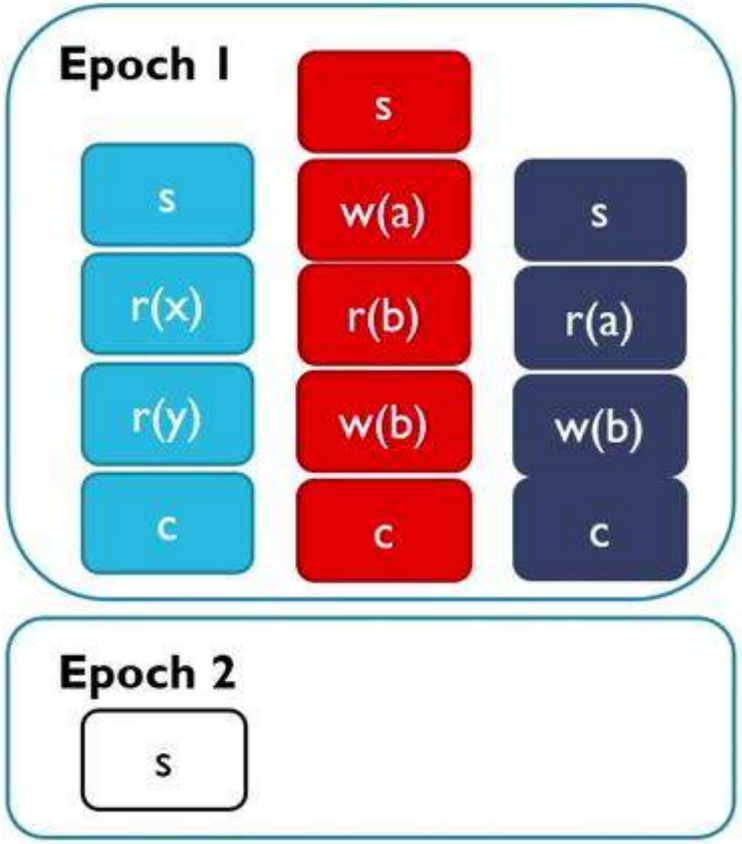
Serializability guarantees apply when transactions observed as committed

Commit notifications to clients can be delayed

The secret sauce: epochs

Use delayed visibility to partition transaction into fixed-sized epochs

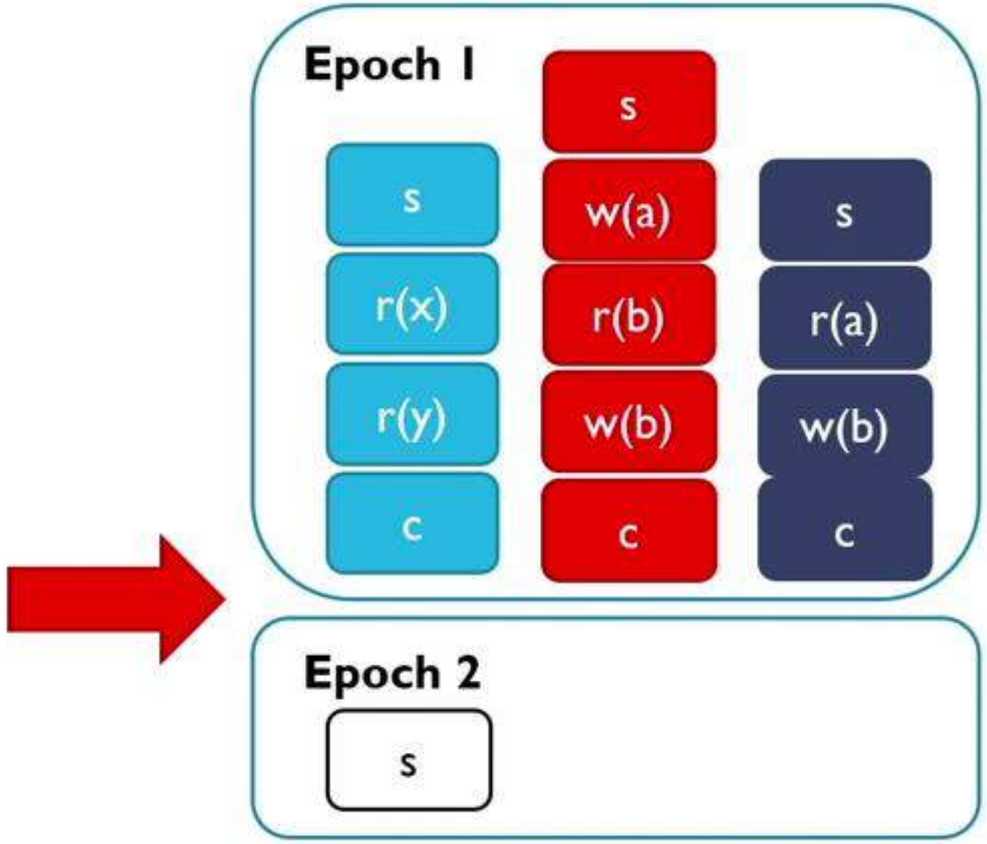
Delay commit notifications until epoch ends



Epochs offer flexibility

Serializability only holds for committed transactions

Enforce durability and serializability at epoch boundaries only

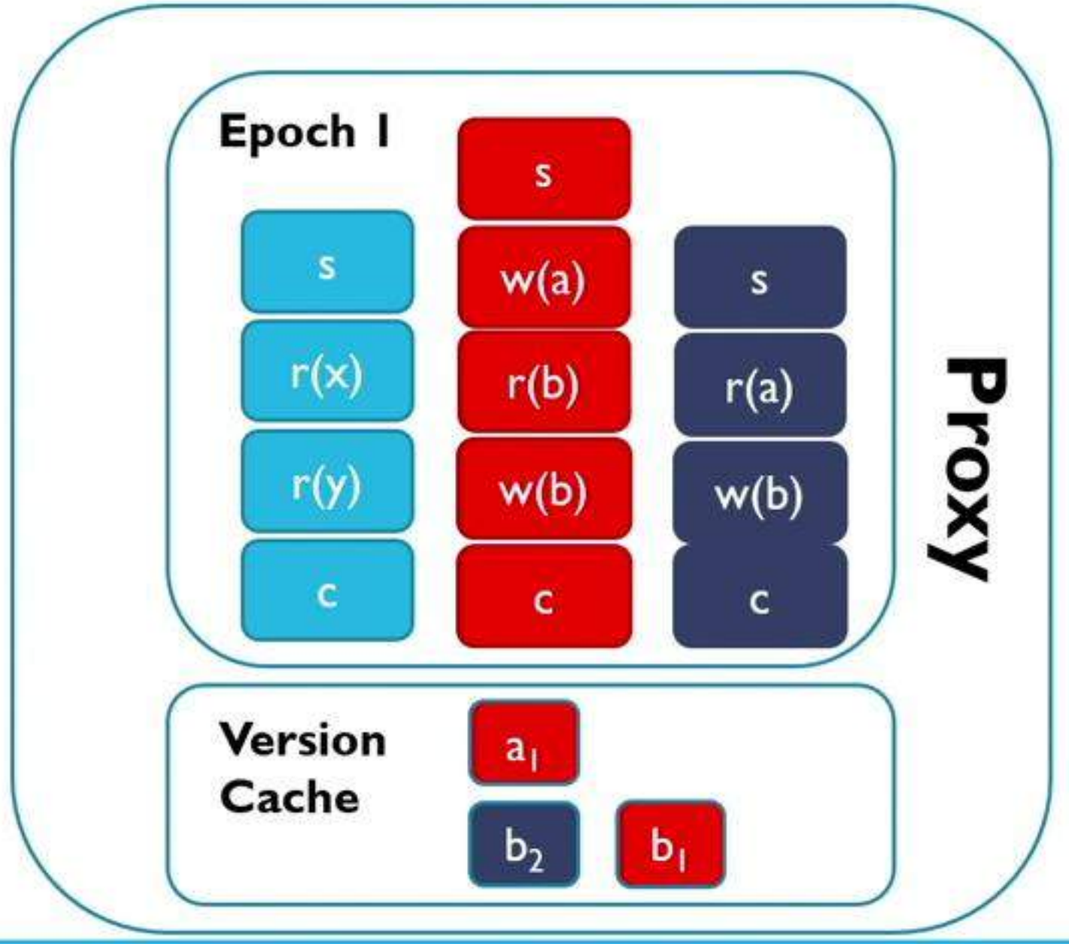


Epochs offer flexibility

During an epoch:

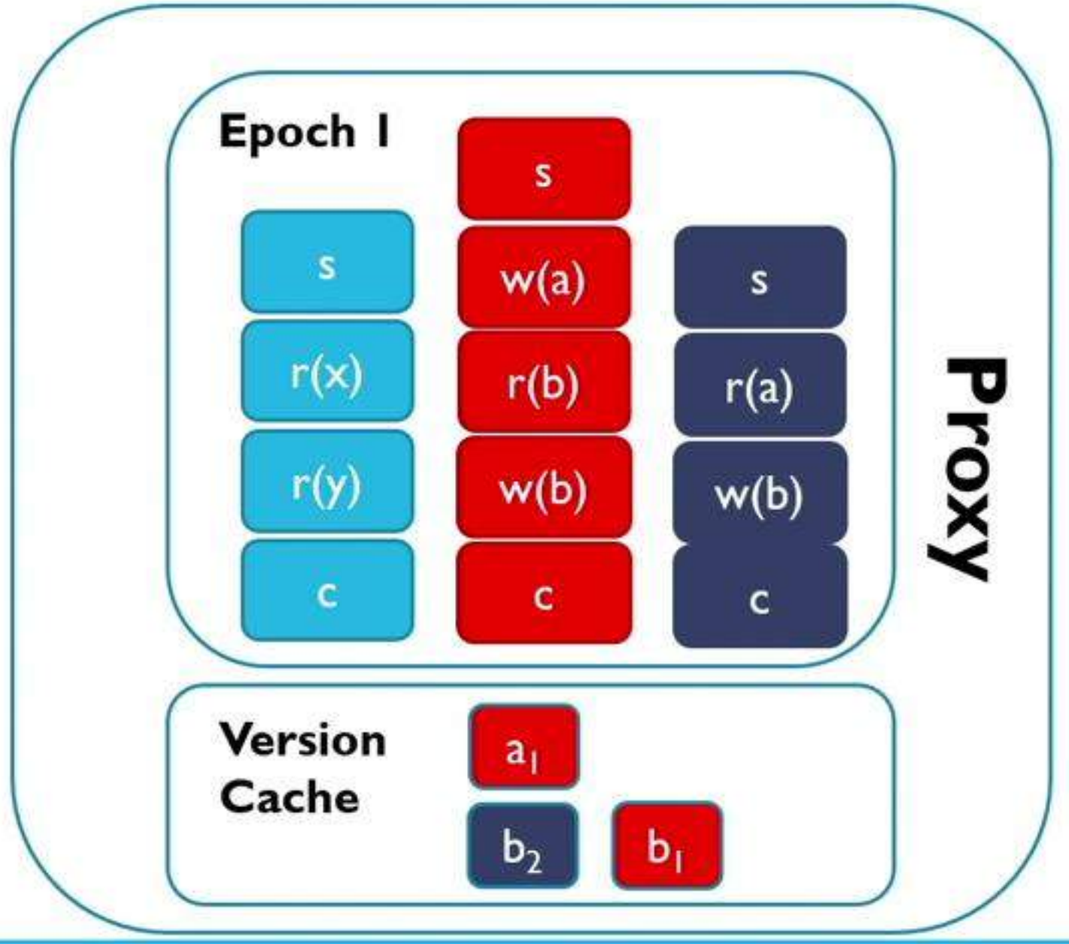
Execute transaction at trusted proxy

Buffer writes until epoch ends



Performance

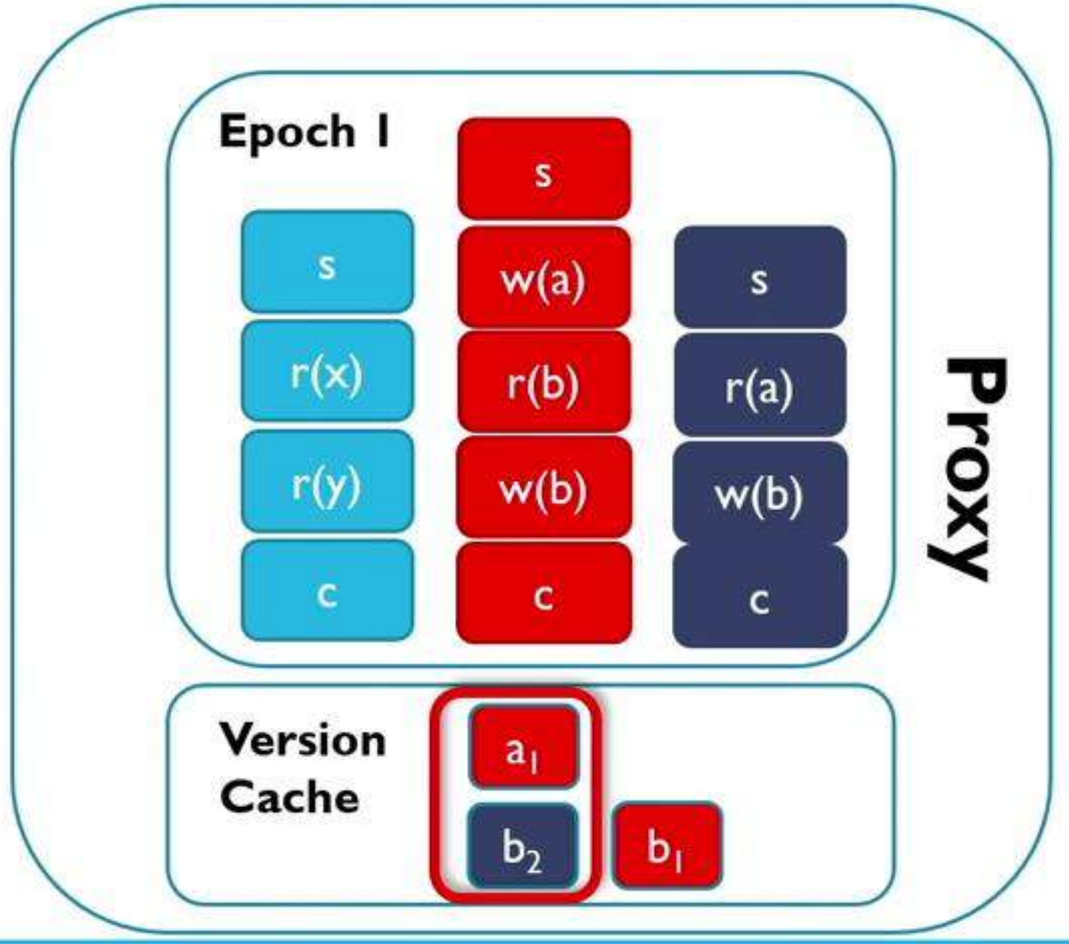
Reduces number of requests sent to ORAM



Performance

Reduces number of requests sent to ORAM

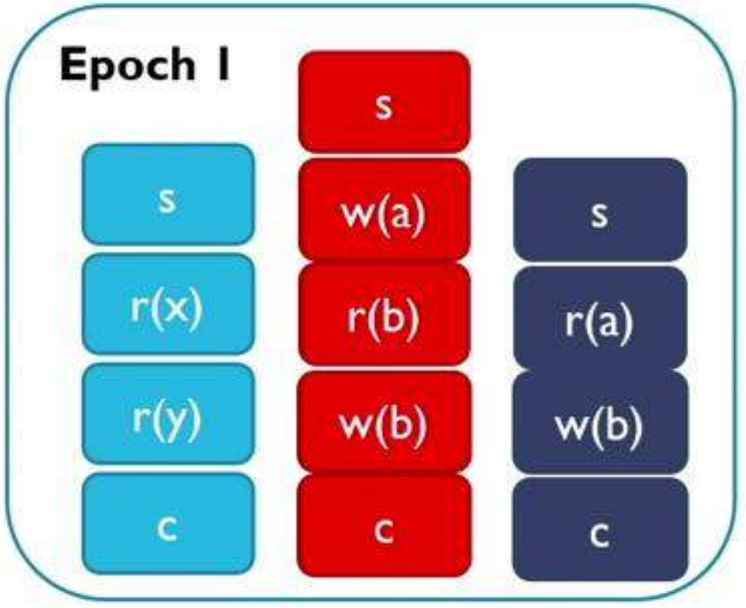
Implement multi-versioned concurrency control on top of single-versioned ORAM



Concurrency control

No increase in contention

Allow transactions to see each other's effects

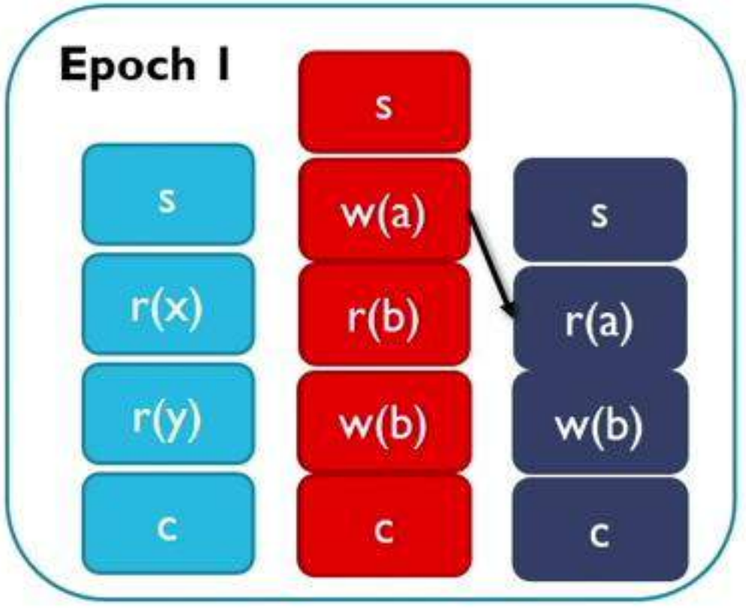


Concurrency control

No increase in contention

Allow transactions to see each other's effects

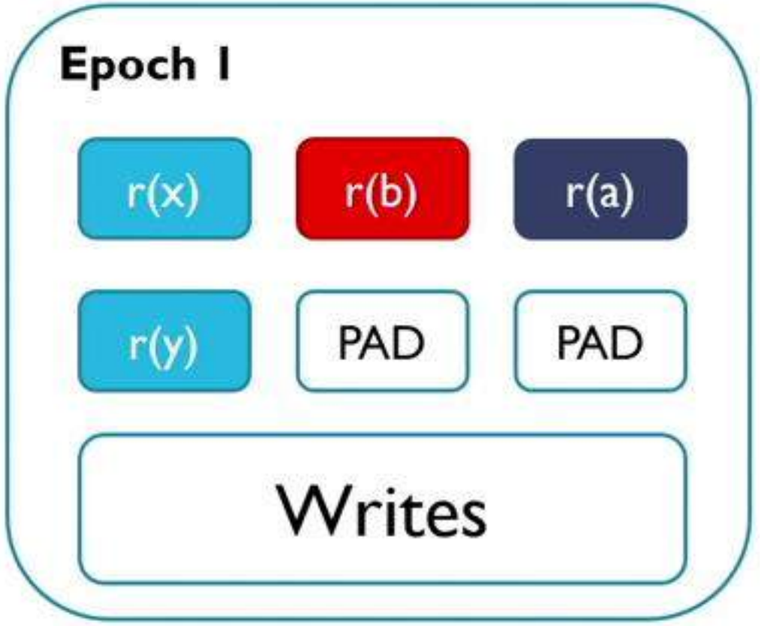
Choose concurrency control that optimistically exposes uncommitted writes



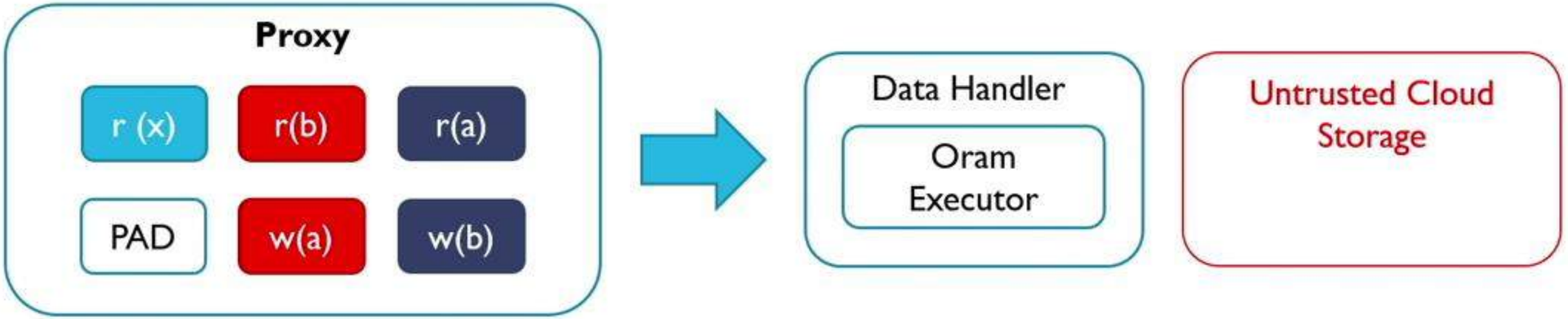
Delayed visibility for security

Fixed structure of epochs guarantees workload independence

ORAM observes the same sequence of reads followed by the buffered writes



ORAM Performance



ORAM constructions are largely sequential

Parallelising ORAM

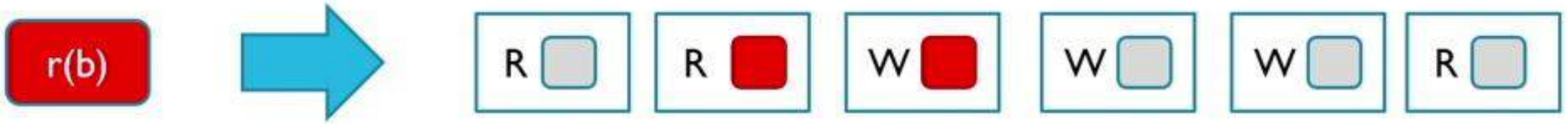
For correctness

parallelization should be linearizable

For security

parallelization should be workload independent

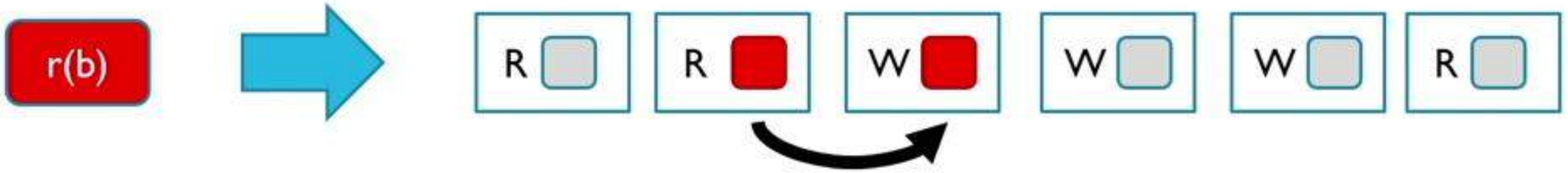
Guaranteeing linearizability



Execute operations that do not have data dependencies in parallel

Data-dependent operations must be executed sequentially

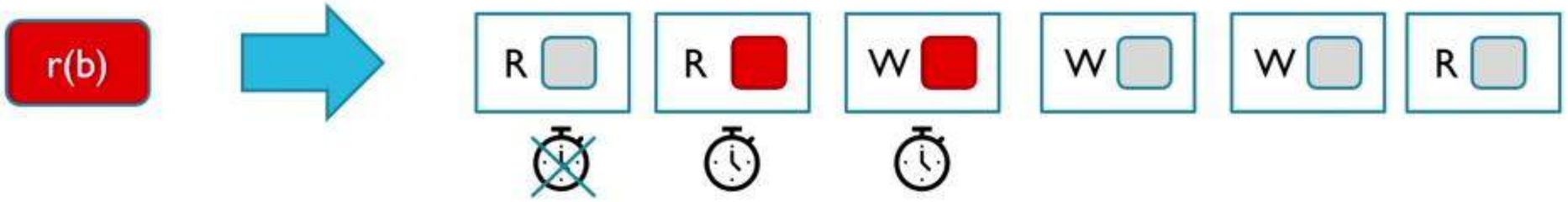
Guaranteeing linearizability



Execute operations that do not have data dependencies in parallel

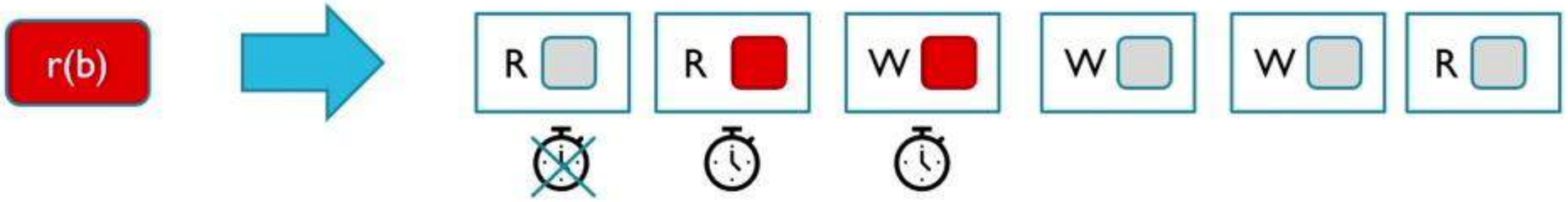
Data-dependent operations must be executed sequentially

Dependencies violate independence



Wait for data dependencies to be satisfied introduces timing channels

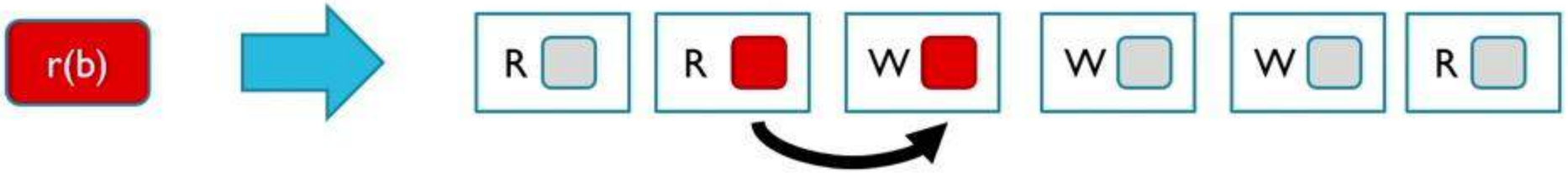
Potential data dependencies



Must wait for all potential data dependencies

Never secure to execute reads and writes in parallel

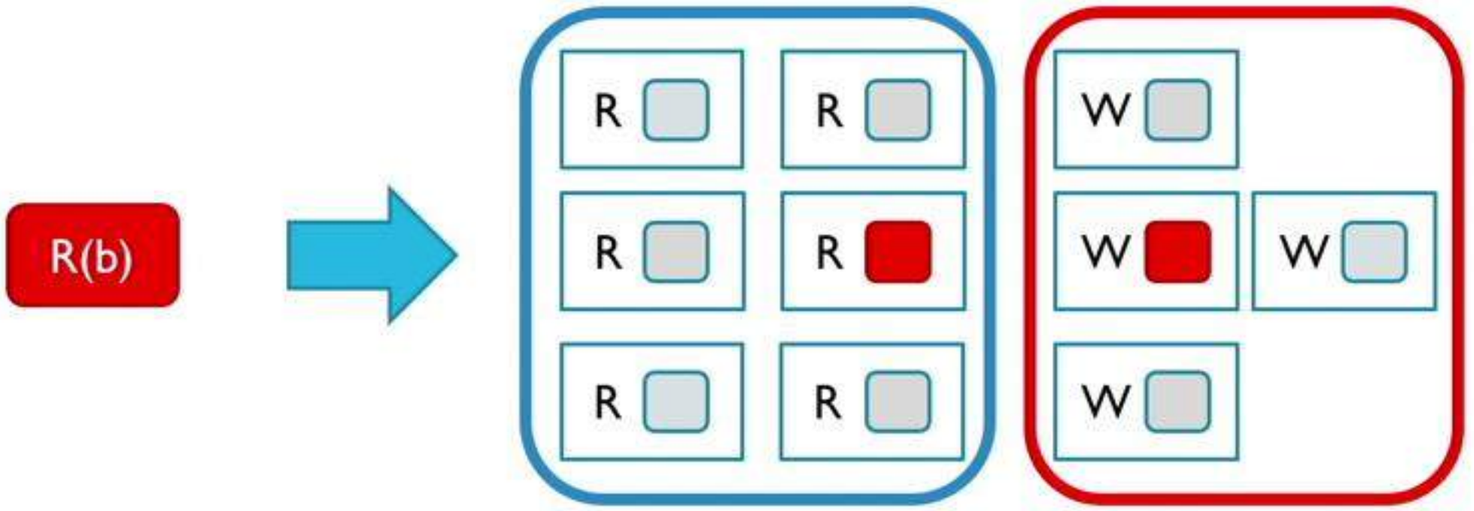
Delayed visibility to the rescue



Delayed visibility allows ORAM to be consistent at epoch boundaries only

Writes can be safely delayed to epoch end

Phase decomposition



Separate ORAM execution into a read phase and a write phase

Evaluation

TPC-C

(10 Warehouses)

SmallBank

(1 million records)

FreeHealth

(7000 patients, 10 hospitals)

NoPriv

(Shares concurrency logic with Obladi)

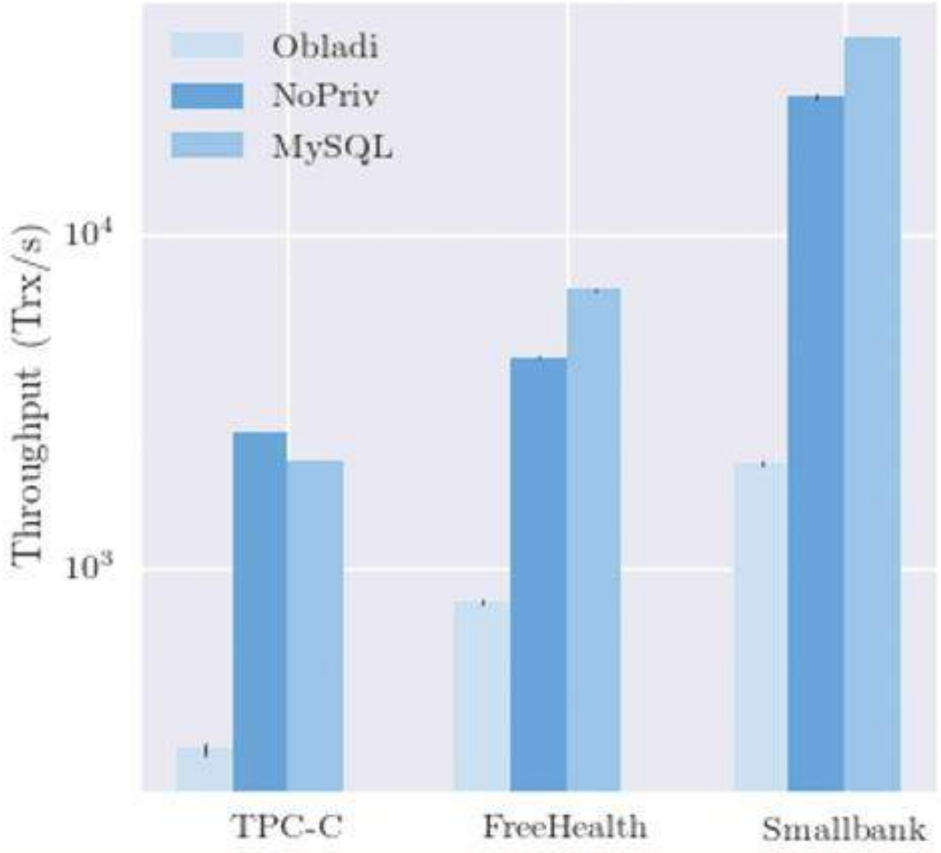
MySQL 5.7 InnoDB

(Multiversioned database)

c5.4xlarge AWS instances

10 ms latency between proxy and storage

Reasonable throughput

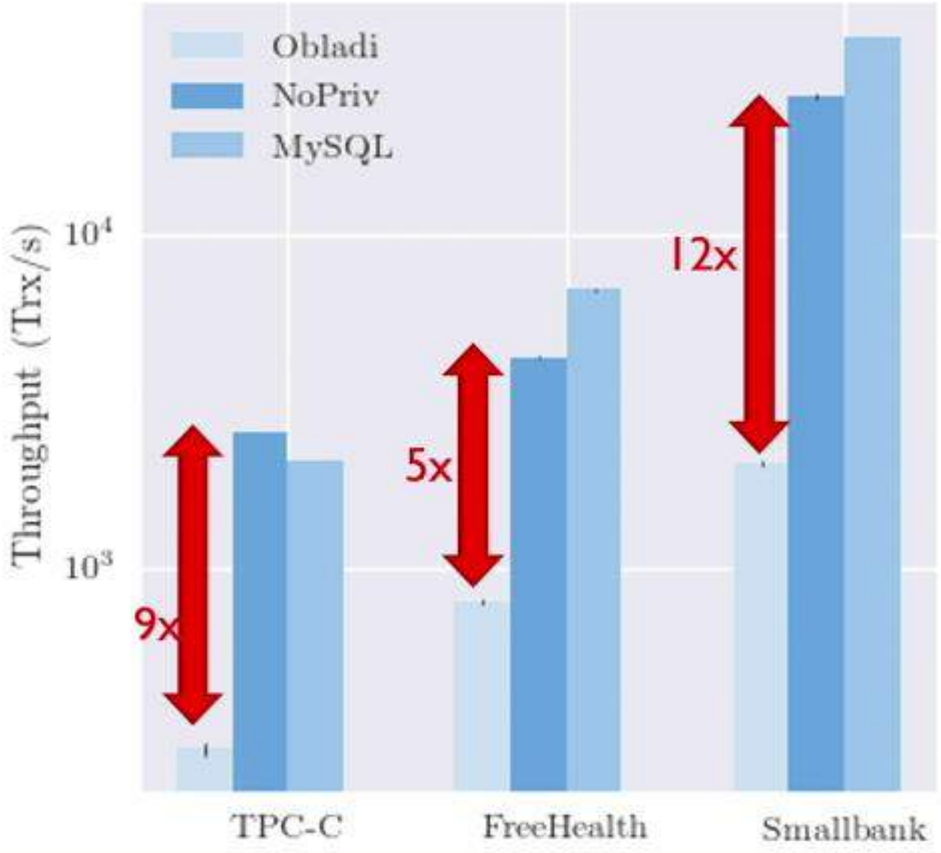


Obladi is slow, but not too slow

5x and 9x lower for
TPC-C and FreeHealth

12x lower throughput for SmallBank

Reasonable throughput

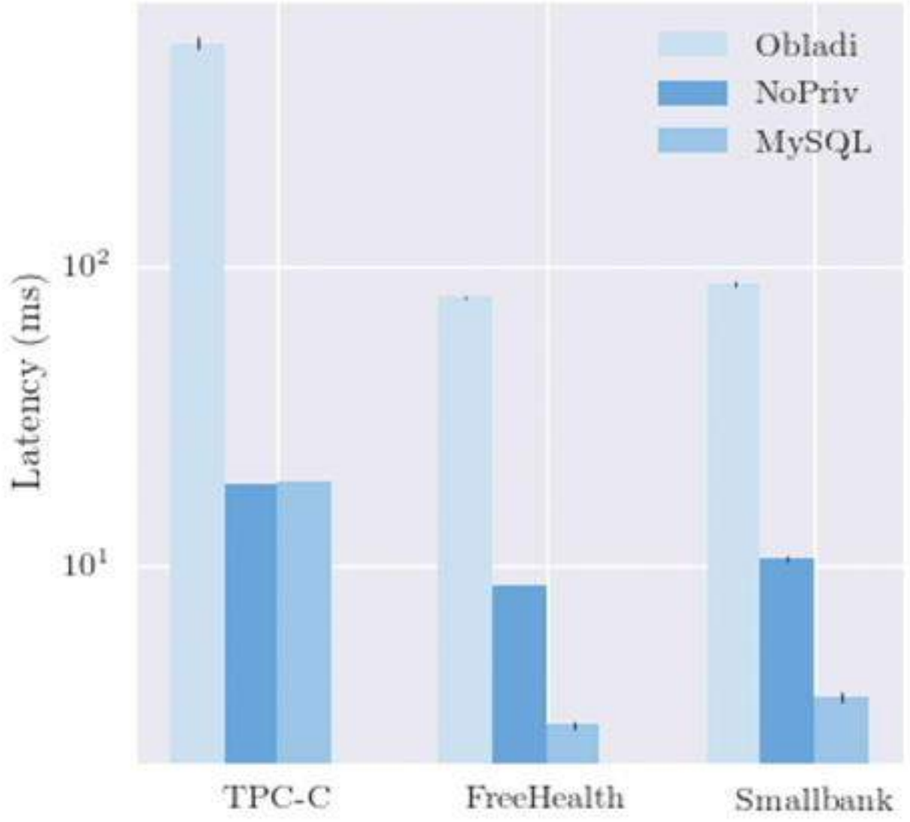


Obladi is slow, but not too slow

5x and 9x lower for
TPC-C and FreeHealth

12x lower throughput for SmallBank

High latency

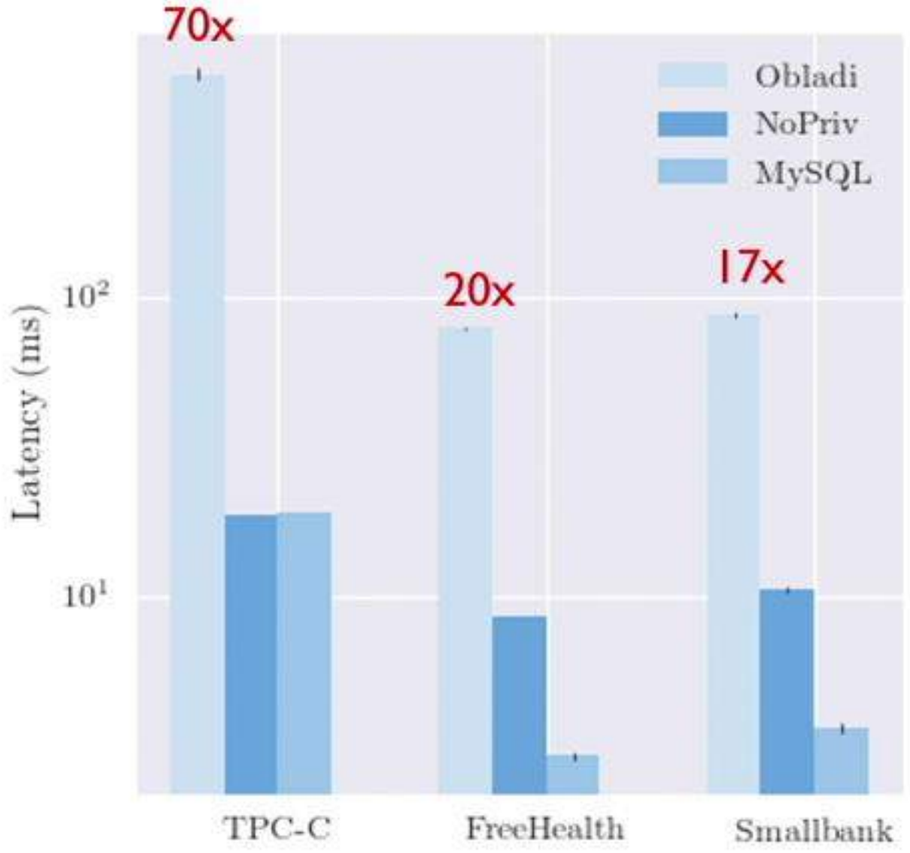


Batching significantly increases latency

Up to 70x on TPC-C

Better on other applications because of smaller write batches

High latency

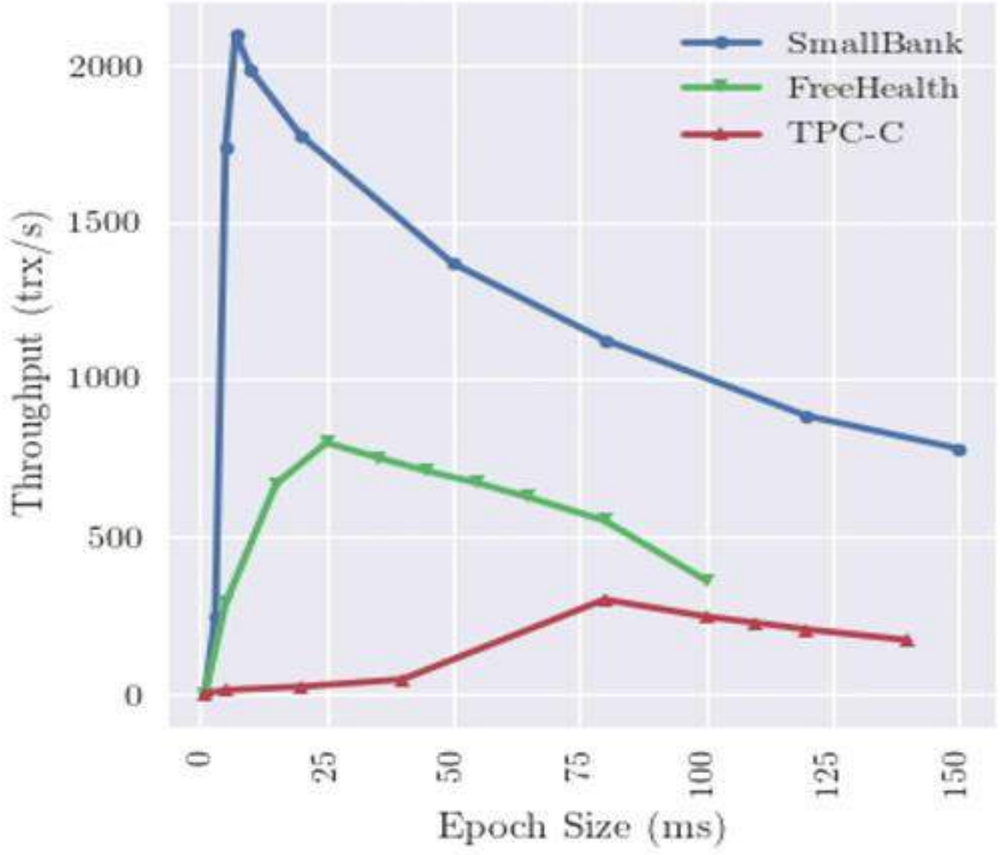


Batching significantly increases latency

Up to 70x on TPC-C

Better on other applications because of smaller write batches

Sensitive to batch size



Performance is sensitive to tuning of epoch size

If too low, transactions cannot finish

If too high, idle time

Delayed visibility for security

Client-centric notion of delayed visibility

Offered reasonable performance

Expanded ORAM abstraction to support parallel accesses and tolerate failures.

In summary

Correctness should be expressed from the application's view point

Addressing challenges

New client-centric model of isolation

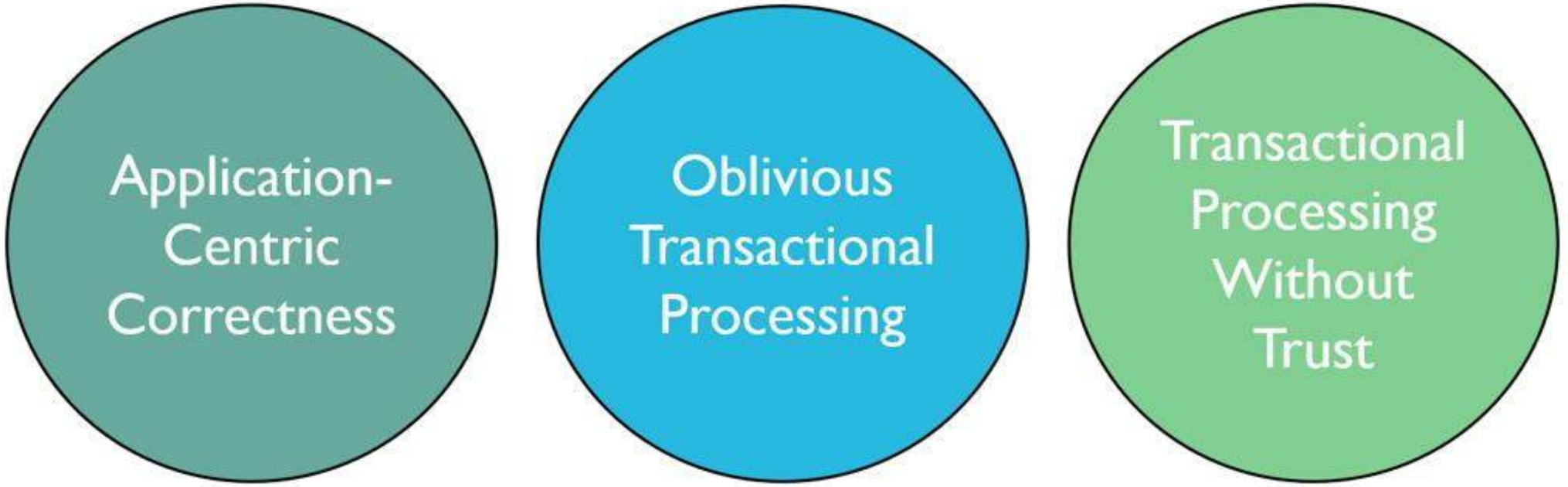
Seizing opportunities

Support private transactions with reasonable overheads

Outline

1) The promise of the cloud
2) Addressing challenges: a client-centric view of correctness
3) Seizing opportunities: Efficient oblivious transactions
4) What's next?

Looking forward



Variable Consistency

Long line of research on variable consistency models

(error bounds, staleness bounds)

Variable Consistency

Long line of research on variable consistency models (error bounds, staleness bounds)

Machine-learning algorithms tolerate some amount of asynchrony

Can we design systems that leverage weak consistency to speed-up model training?

Oblivious Transactional Processing

Overheads of obliviousness depend on data size

Oblivious Transactional Processing

Overheads of obliviousness depend on data size

Log-structured databases are already oblivious

Can we solve the client-playback problem obliviously?

Trustless Serializable Transactions

Replicated databases: scalable, geo-distributed transactions among trusted parties

Partially Ordered Log

Byzantine-fault tolerance: shared computation among non-trusted parties

Totally Ordered Log

Can we combine the two?

Acknowledgements

Rachit Agarwal

Lorenzo Alvisi

Phil Bernstein

Sebastian Burckhardt

Nathan Bronson

Matthew Burke

Sergey Bykov

Ethan Cecchetti

Allen Clement

Cong Ding

Nancy Estrada

Jose Faleiro

Ionel Gog

Trinabh Gupta

Matt Grosvenor

Steven Hand

Gabriel Kliot

Alok Kumbhare

Cody Littley

Wyatt Lloyd

Akbar Mehdi

Whitney Mulhern

Youer Pu

Muntasir Raihan Rahman

Vivek Shah

Malte Schwarzkopf

Srinath Setty

Chunzhi Su

Adriana Szekeres

Michael Walfish

Chao Xie

Jorgen Thelin

Conclusion

"It's not a lie if you don't get caught."

Client-centric approach improves
understanding and performance
of transactional datastores