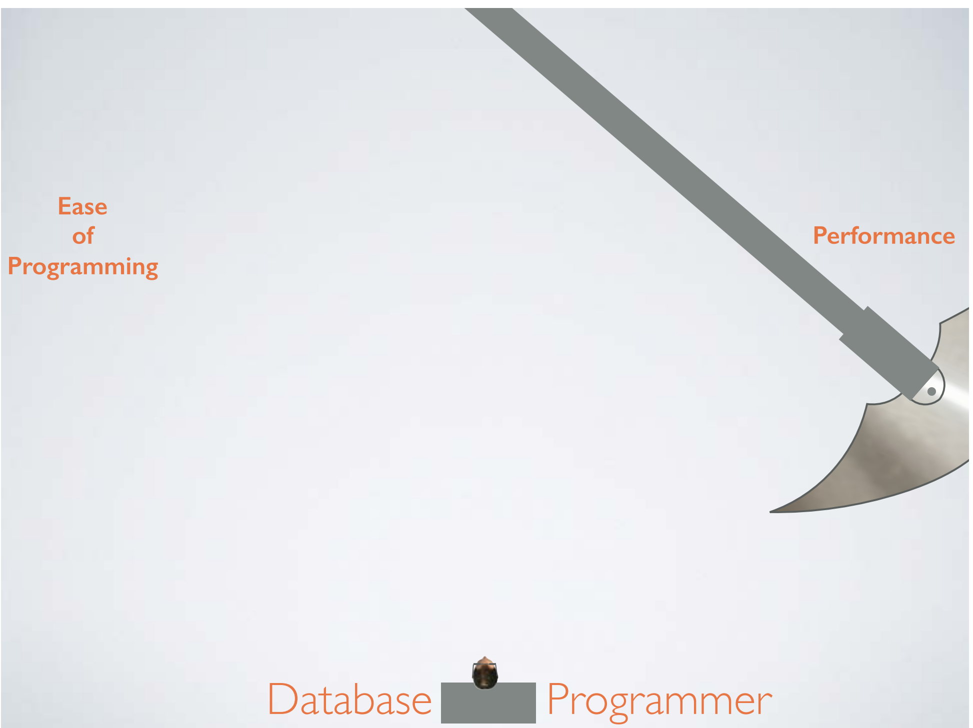
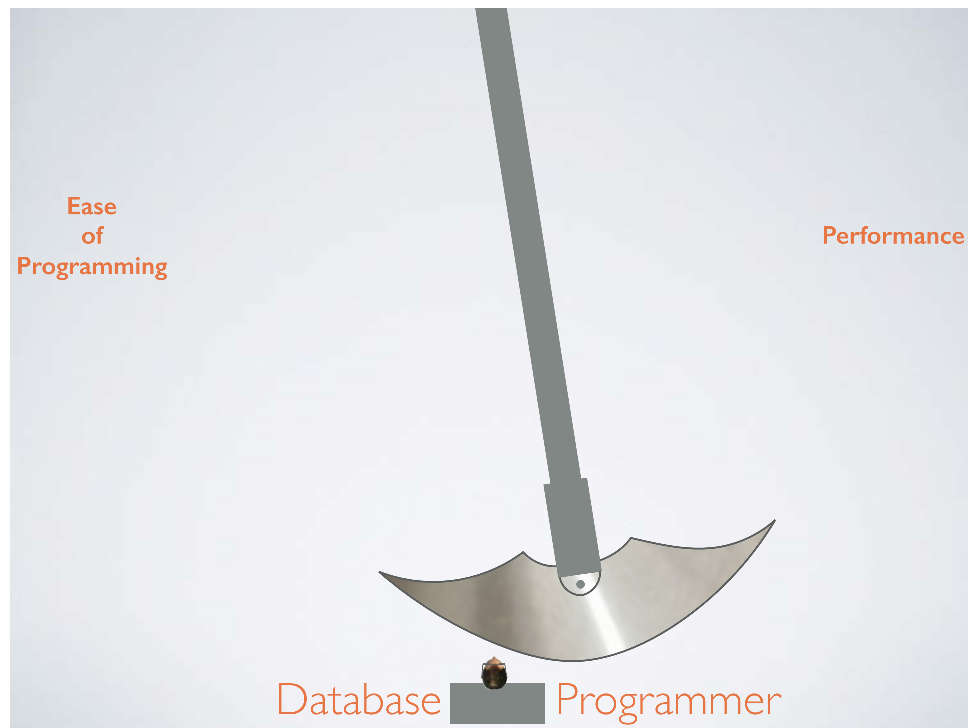

Cornell CIS
Computer Science
THE PIT AND THE PENDULUM
Lorenzo Alvisi


A CLASSIC HORROR STORY
Ease
of
Programming
Performance
Database
Programmer


Ease
of
Programming
Performance
Database
Programmer


Ease
of
Programming
Performance
Database
Programmer

CONCURRENCY

Pierre Franc Lamy
Young girl on a balcony (1911)

Carlo Carrà
Concurrency, Woman on a balcony (1912)

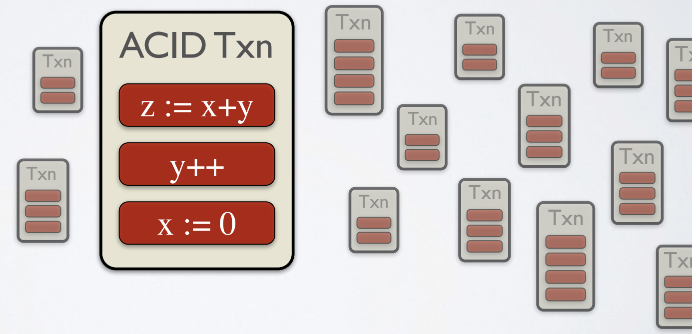
ACID TRANSACTIONS: SIMPLE AND POWERFUL

Atomicity

Consistency

Isolation

Durability



PERFORMANCE VIA WEAKER ISOLATION GUARANTEES

Database System	Default Isolation	Strongest Isolation
MySQL Cluster	Read Committed	Read Committed
SAP HANA	Read Committed	Snapshot Isolation
Google Spanner	Serializability	Serializability
VoltDB	Serializability	Serializability
Oracle 12C	Read Committed	Snapshot Isolation
MemSQL	Read Committed	Read Committed
SQL Server	Read Committed	Serializability
Postgres	Read Committed	Serializability

ANSI SQL-92 ISOLATION LEVELS

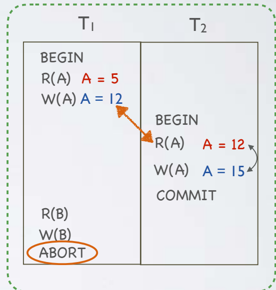
- Defined in terms of **three phenomena** that can lead to violations of serializability
- Motivated by weakening locking implementations of serializability
- Designed to be implementation independent (greater flexibility/ better performance)

Isolation Level	Proscribed Phenomena		
	Dirty Read	Fuzzy Read	Phantom
Read Uncommitted	🟢	🟢	🟢
Read Committed	🔴	🟢	🟢
Repeatable Read	🔴	🔴	🟢
(Anomaly) Serializable	🔴	🔴	🔴

DIRTY READS

Root: Write-Read conflict

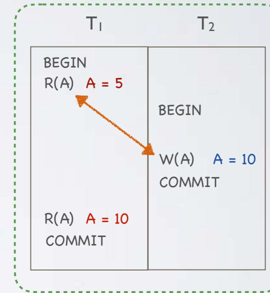
- T_1 modifies a data item.
- T_2 reads that data item before T_1 commits or aborts.
- If T_1 then aborts, T_2 has read a data item that was never committed and so never really existed.



FUZZY READS
A.K.A. NON-REPEATABLE READS

Root: Read-Write conflict

- T_1 reads a data item.
- T_2 then modifies or deletes that data item and commits.
- If T_1 then attempts to reread the item, it receives a modified value or discovers the item was deleted.

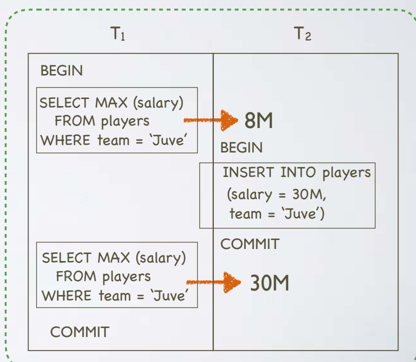


THE PHANTOM MENACE

Non-repeatable predicate-based reads

- T_1 reads a set of data items satisfying <search condition>.
- T_2 then creates data items that satisfy T_1's <search condition> and commits.
- If T_1 then repeats its read with the same <search condition>, it gets a different set of data

On July 10 2018…



WHAT'S NOT TO LIKE?

Berenson et al, SIGMOD '95

- Ambiguous descriptions of proscribed behaviors

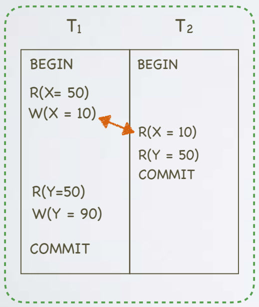
Dirty Reads

- **Strict Interpretation** (prohibits anomaly)
  - **A1:** $W_1[X] \ldots R_2[X] \ldots (A_1$ and C_2 in any order)
- **Broad Interpretation** (prohibits phenomenon)
  - **P1:** $W_1[X] \ldots R_2[X] \ldots ((A_1$ or $C_1)$ and $(A_2$ or $C_2)$ in any order)

similar distinctions for P2 (NR reads) and P3 (Phantoms)

PHENOMENA OR ANOMALIES?

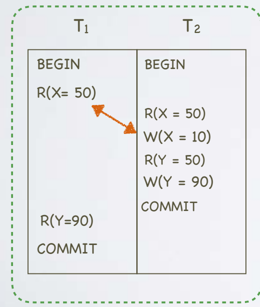
Dirty Reads



- Non serializable
  - T_2 reads the wrong balance
- Yet fine by Strict Interpretation A_1…
  - $W_1[X]$ … $R_2[X]$ … (A_1 and C_2 in any order)
  - T_1 does not abort!
- but violates Broad Interpretation P_1
  - $W_1[X]$ … $R_2[X]$ … ((A_1 or C_1) and (A_2 or C_2) in any order)

PHENOMENA OR ANOMALIES?

Non-repeatable Reads



- Non serializable
  - T_1 reads the wrong balance
- Yet fine by Strict Interpretation A_2…
  - $R_1[X]$ … $W_2[X]$ … C_2 … $R_1[X]$ … C_1
  - No transaction reads same value twice
- but violates Broad Interpretation P_2
  - $R_1[X]$ … $W_2[X]$ … ((A_1 or C_1) and (A_2 or C_2) in any order)

ANSI isolation levels should be intended to proscribe phenomena, not anomalies

WHAT'S NOT TO LIKE?

- ANSI SQL phenomena are weaker than their locking counterpart

Isolation Level	Read Locks	Write Locks
Locking Read Uncommitted	None	Long† write locks
Locking Read Committed	Short* read locks (both)	Long write locks
Locking Repeatable Read	Long item locks Short predicate locks	Long write locks
Locking Serializable	Long read locks (both)	Long write locks

Short*: Released after operation ends

Long†: Released after transaction commits

ANSI P3 should prevent phantoms due to deletions and updates, not just creations

WHAT'S NOT TO LIKE?

- ANSI SQL phenomena are still weaker than their locking counterpart

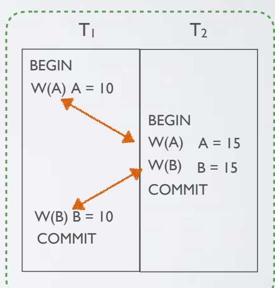
Isolation Level	Read Locks	Write Locks
Locking Read Uncommitted	None	Long† write locks
Locking Read Committed	Short* read locks (both)	Long write locks
Locking Repeatable Read	Long item locks Short predicate locks	Long write locks
Locking Serializable	Long read locks (both)	Long write locks

Short*: Released after operation ends   Long†: Released after transaction commits

DIRTY WRITES

Root: Write-Write conflicts

- T_1 modifies a data item
- T_2 further modifies that data item before T1 commits or aborts.
- Conflicting writes can interleave, violating invariants



ANSI isolation levels should include phenomenon **P0**

P0: $W_1[X] \ldots W_2[X] \ldots (C_1 \text{ or } A_1)$ and $(C_2 \text{ or } A_2)$ in any order

ANSI-92 ISOLATION LEVELS, POST CRITIQUE

Locking Isolation Level	Proscribed Phenomena	Read locks on data items and phantoms	Write locks on data items and phantoms
Degree 0	none	none	Short* write locks
Degree 1 = Locking READ UNCOMMITTED	P0	none	Long† write locks
Degree 2 = Locking READ COMMITTED	P0, P1	Short read locks	Long write locks
Locking REPEATABLE READ	P0, P1, P2	Long data-item read locks; Short phantom read locks	Long write locks
Degree 3 = Locking SERIALIZABLE	P0, P1, P2, P3	Long read locks	Long write locks

Short*: Released after operation ends

Long†: Released after transaction commits

AND YET…

- "P0, P1, P2, and P3 are a disguised version of locking"
  - no implementation independence
  - Preventing concurrent execution of conflicting operations approach rules out optimistic and multi version implementations
- P0: $W_1[X] \ldots W_2[X] \ldots (C_1 \text{ or } A_1)$
  - rules out optimistic implementations
  - similar argument holds for P1, P2. P3

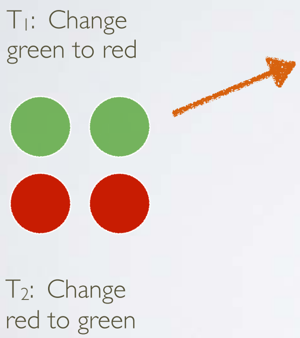
THE RUB

- Phenomena expressed through single object histories
  - but consistency often involves multiple objects
- Same guarantees for running and committed transactions
  - but optimistic approaches thrive on the difference
- Definition in terms of objects, not versions
  - no support for **multiversion systems**

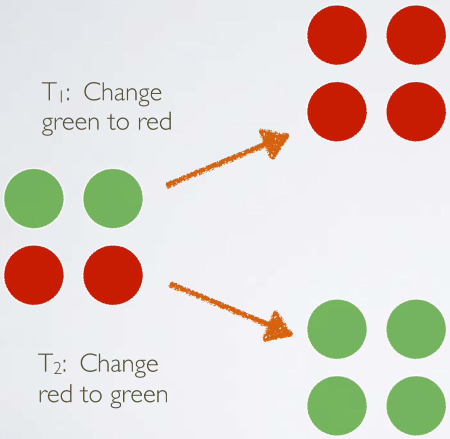
SNAPSHOT ISOLATION

- T reads from a snapshot of committed values at T's start time
- T's own writes are reflected in its snapshot
- When ready to commit, T receives a commit time
- T commits if its updates do not conflict with those of any transaction which committed in the interval between T's start time and commit time

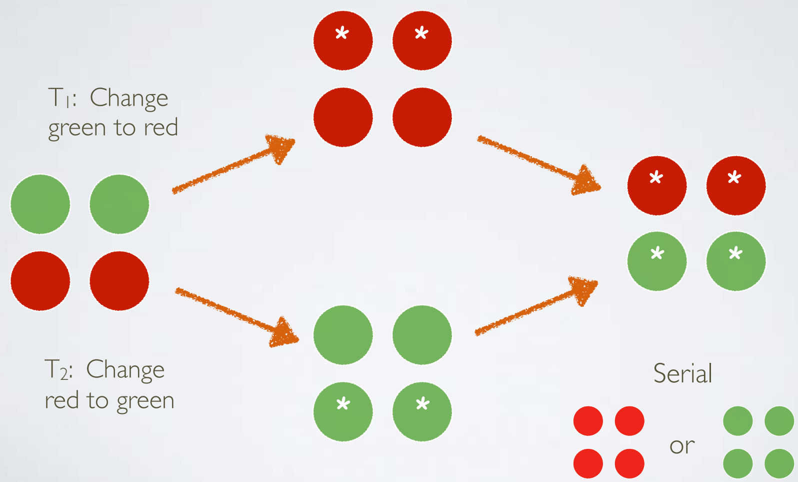
WRITE SKEW ANOMALY



WRITE SKEW ANOMALY



WRITE SKEW ANOMALY



GENERALIZED ISOLATION DEFINITIONS

Adya et al, SIGMOD '95

- Executions modeled as histories
  - a partial order of read/write operations that respects order of operations in each transaction
  - a total order << of object versions created by committed transactions

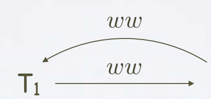
SERIALIZATION GRAPH

- Every history is associated with a Direct Serialization Graph (DSG)
  - nodes are committed transactions
  - edges express different types of direct conflicts
    - write-read $T_i \xrightarrow{wr} T_j$ (dependency)
    - write-write $T_i \xrightarrow{ww} T_j$ (dependency)
    - read-write $T_i \xrightarrow{rw} T_j$ (anti-dependency)
  - edge expresses temporal relation
    - start $T_i \xrightarrow{s} T_j \ : \ c_i < s_j$

READ UNCOMMITTED

Proscribes P0: $W_1[X] \ldots W_2[X] \ldots (C_1 \text{ or } A_1)$

Now, proscribes G0: DSG(H) contains a directed cycle consisting exclusively of WW edges



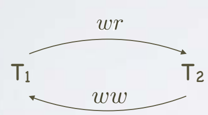
Concurrent transactions can modify the same object (as long as they don't all commit)

STRONGER ISOLATION LEVELS

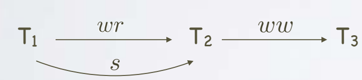
- No aborted reads
  - T2 cannot read value of aborted T1
- No intermediate reads
  - T2 cannot read value of T1 that T1 then overwrites
- No circularity in DSG graph
  - edges in cycle depend on isolation level

SNAPSHOT ISOLATION

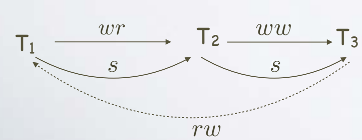
- DSG(H) proscribes:



cycles consisting of write-write or write-read dependencies



a write-read or write-write edge without a start edge

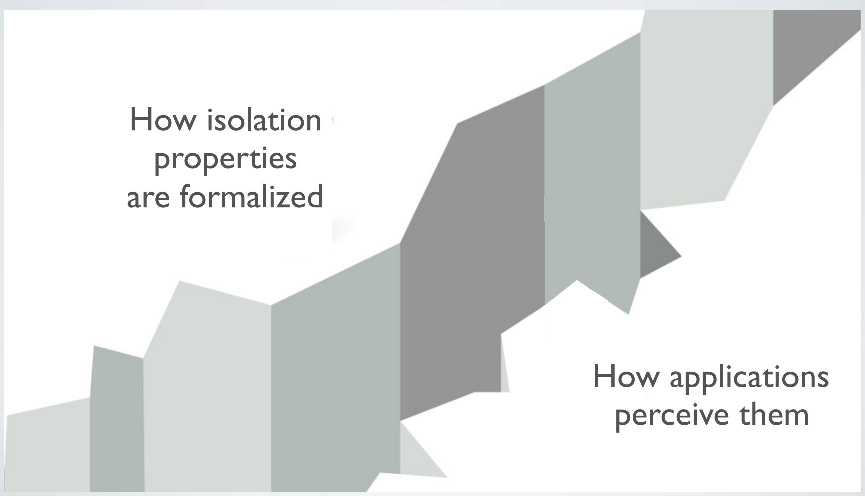


a cycle consisting of write-read/write-write/start-edges, and a single read-write edge

ALL'S WELL?

ALL'S WELL?

WHY THE CONFUSION?



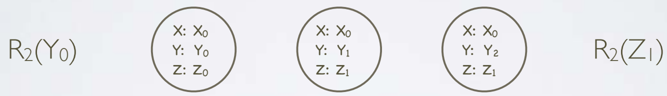
DON'T KNOW MUCH ABOUT HISTORIES

- Applications experience isolation guarantees as contracts specifying which values they can read (i.e. which states they can observe)
- Low-level read/write operations are instead
  - invisible to applications
  - encourage system-specific definitions

A STATE-BASED DEFINITION

Crooks et al, 2017

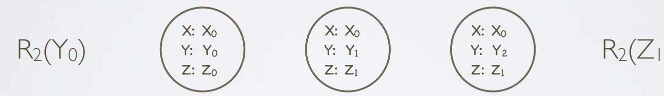
- Isolation guarantees as **constraints** on read states
  - states consistent with what the application observed



A STATE-BASED DEFINITION

Crooks et al, 2017

- Isolation guarantees as constraints on read states
  - states consistent with what the application observed



- Each transaction is associated with a set of candidate read states
- At commit, transaction must pass a commit test that narrows down which read states are acceptable

A STATE-BASED DEFINITION

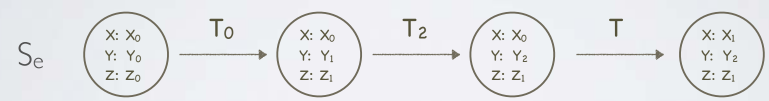
Crooks et al, 2017

A storage system guarantees a specific isolation level **I** if it can produce an execution (a sequence of atomic state transitions) that

- is consistent with every transaction's read states
- satisfies the commit test for **I**, for every transaction

If no read state prove suitable for some transaction, then **I** does not hold

PARENT STATES AND COMPLETE STATES



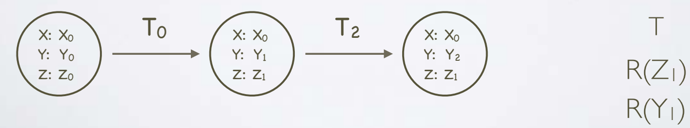
PARENT STATES AND COMPLETE STATES

- Parent state s_p of T: state from which T commits

PARENT STATES AND COMPLETE STATES

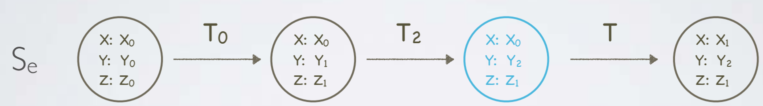
- Parent state s_p of T: state from which T commits

- Complete state for T: a read state for all read ops in T

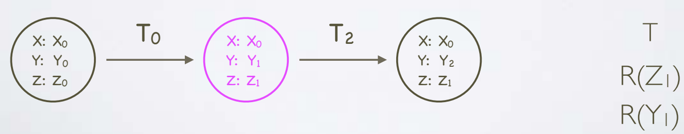


PARENT STATES AND COMPLETE STATES

- Parent state s_p of T: state from which T commits



- Complete state for T: a read state for all read ops in T



SERIALIZABILITY

SERIALIZABILITY

- Given a set of transactions **T** and their read states, serializability holds if there exists execution **e** such that for all T in **T**

$$\text{COMPLETE}_{e,T}(s_p)$$

SNAPSHOT ISOLATION

- DSG(H) proscribes:

$$\begin{tikzcd} T_1 \arrow[r, bend left, "wr"] & T_2 \arrow[l, bend left, "ww"] \end{tikzcd}$$

$$\begin{tikzcd} T_1 \arrow[r, "wr"] \arrow[r, bend right, "s"'] & T_2 \arrow[r, "ww"] & T_3 \end{tikzcd}$$

$$\begin{tikzcd} T_1 \arrow[r, "wr"] \arrow[r, bend right, "s"'] & T_2 \arrow[r, "ww"] \arrow[r, bend right, "s"'] & T_3 \arrow[ll, dashed, bend left, "rw"] \end{tikzcd}$$

SNAPSHOT ISOLATION

- Given a set of transactions **T** and their read states, snapshot isolation holds if there exists execution **e** such that for all T in **T**

$$\exists s \in S_e. \wedge \text{COMPLETE}_{e,T}(s)$$

SNAPSHOT ISOLATION

- Given a set of transactions **T** and their read states, snapshot isolation holds if there exists execution **e** such that for all T in **T**

$$\exists s \in S_e. \wedge \text{COMPLETE}_{e,T}(s)$$
$$\wedge(\Delta(s, s_p) \cap \mathcal{W}_T = \emptyset)$$

PERFORMANCE VIA WEAKER ISOLATION GUARANTEES

It-That-Shall-Not-Be-Named

dirty writes - transaction modifies item previously modified by undecided transaction

Read-Uncommitted

dirty reads: one transaction may see uncommitted state of another transaction

Read-Committed

no dirty reads or writes, but allows for non-repeatable reads

Repeatable Reads

non-repeatable range reads

Snapshot Isolation

none of the above, but write skew

BASE

Basically
Available,
Soft state,
Eventually consistent

BASE

Basically
Available,
Soft state,
Eventually consistent

Custom code for better performance

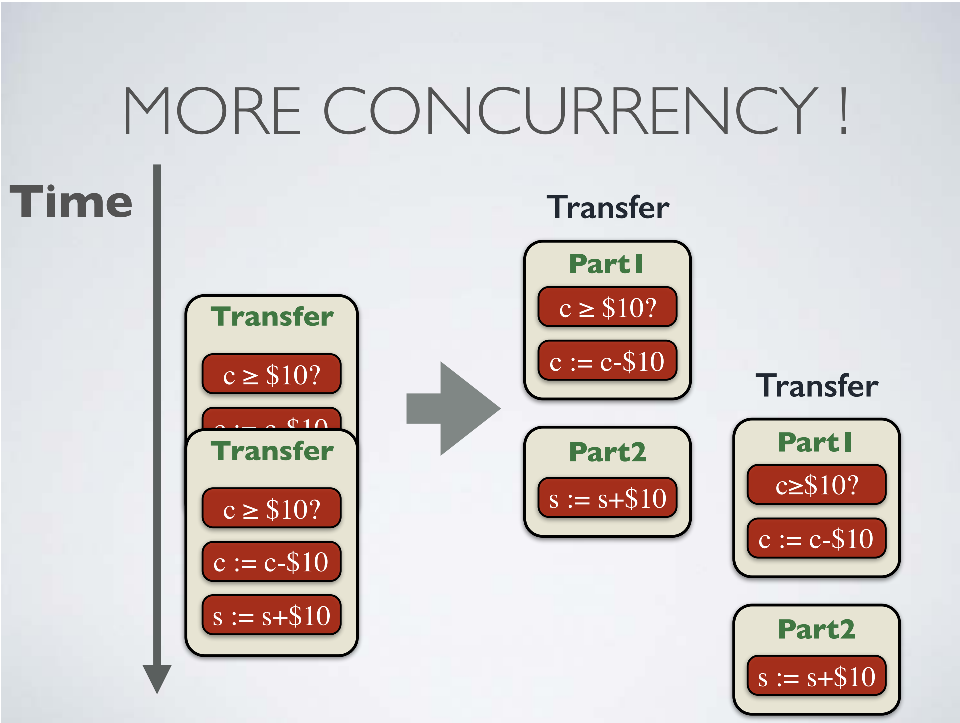
Complexity gets quickly out of control

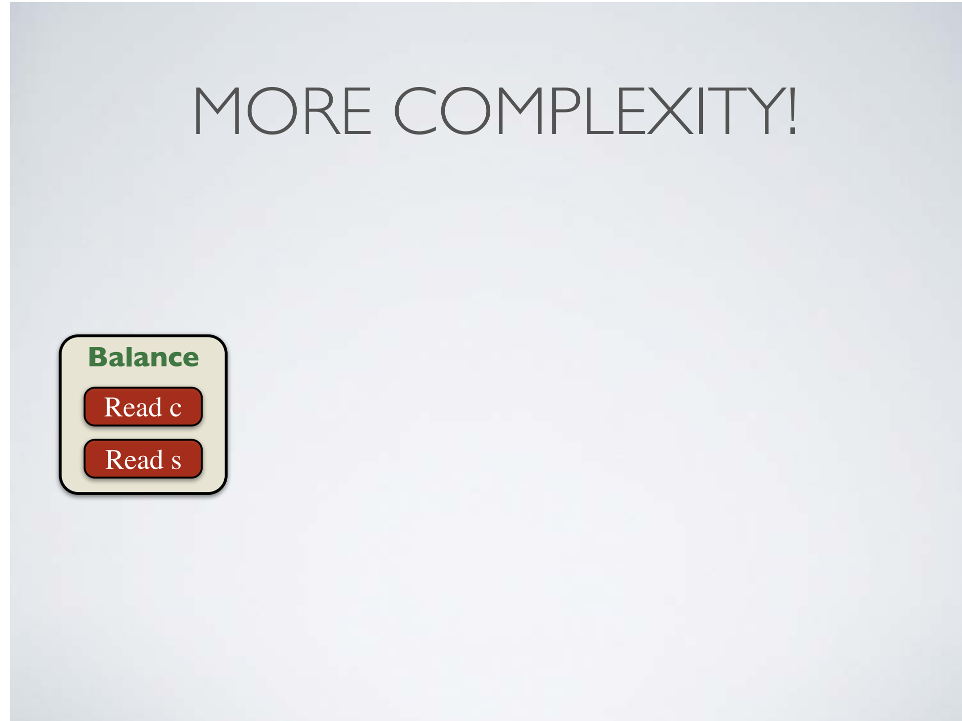
Application	Implement Consistency

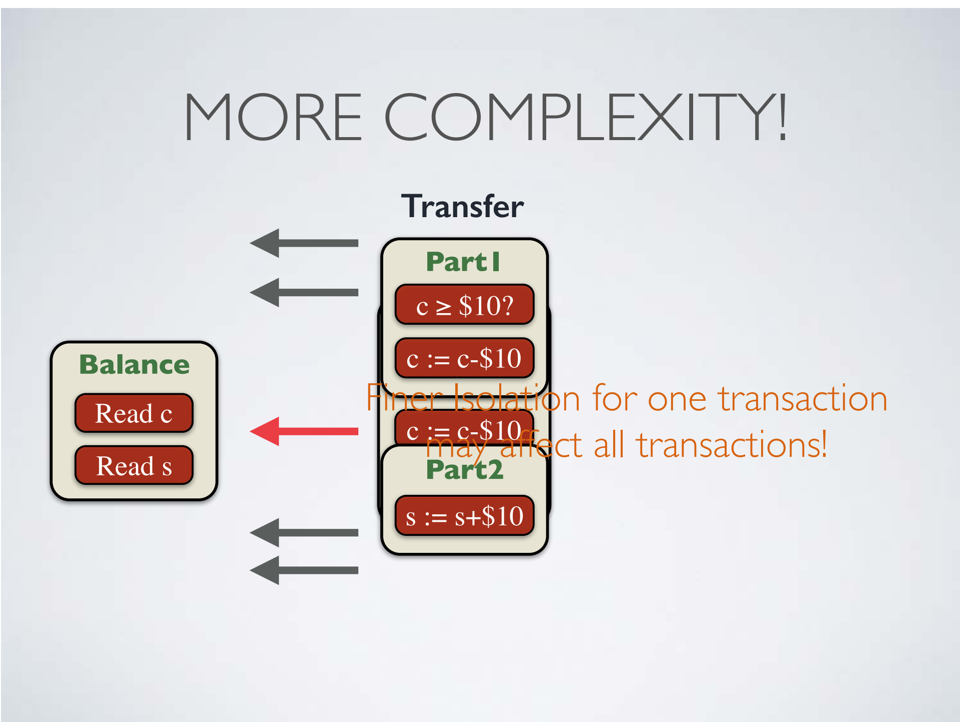
Storage Interface

ACID BASE Storage (e.g. put, get)	Transaction Guarantees

A CLASSIC HORROR STORY


MORE CONCURRENCY !
Time
Transfer
c ≥ $10?
Transfer
c ≥ $10?
c := c-$10
s := s+$10
Transfer
Part1
c ≥ $10?
c := c-$10
Part2
s := s+$10
Transfer
Part1
c≥$10?
c := c-$10
Part2
s := s+$10


MORE COMPLEXITY!
Balance
Read c
Read s


MORE COMPLEXITY!
Transfer
Part1
c ≥ $10?
c := c-$10
c := c-$10
Part2
s := s+$10
Balance
Read c
Read s
Finer Isolation for one transaction
may affect all transactions!


Performance vs Complexity
Better Performance
More Interleavings
Greater Complexity

NOT ALL TRANSACTIONS ARE CREATED EQUAL

Vilfredo Pareto

- Many transactions are not run frequently
- Many transactions are lightweight

20% of the causes account for 80% of the effects

Performance vs Complexity

Better Performance

↓

More Interleavings

↓

Greater Complexity

Performance vs Complexity

More Interleavings
selectively

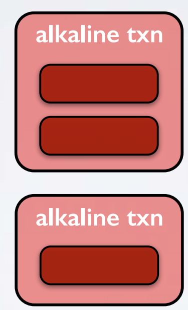
Performance vs Complexity

More Interleavings
selectively

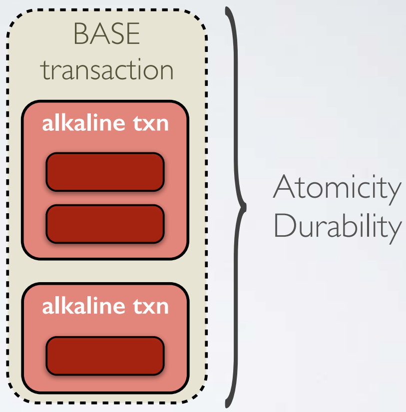
SALT

NOT ALL TRANSACTIONS ARE CREATED EQUAL

Use a flexible abstraction



BASE TRANSACTION



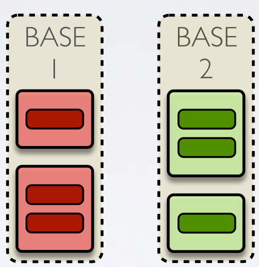
Different Isolation guarantees for
different types of transactions

SALT ISOLATION

To BASE transactions:
a sequence of small
ACID transactions

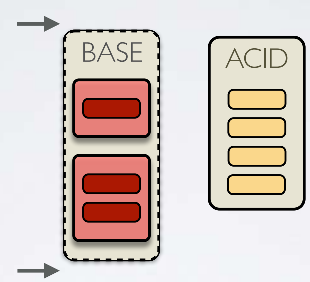
To ACID transactions:
a single, monolithic
ACID transaction

BASE WITH BASE



Fine Isolation granularity
between BASE transactions

BASE WITH ACID



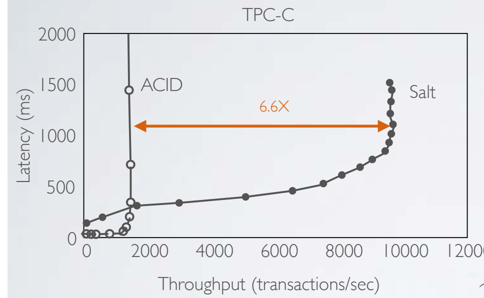
Coarse Isolation granularity
to ACID transactions

HOW WELL DOES IT WORK ?

How does the performance of Salt compare to ACID?

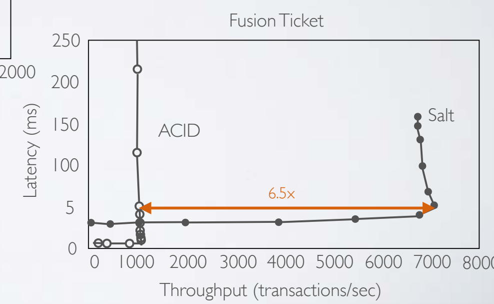
How much programming effort is required to get that performance?

PERFORMANCE GAIN



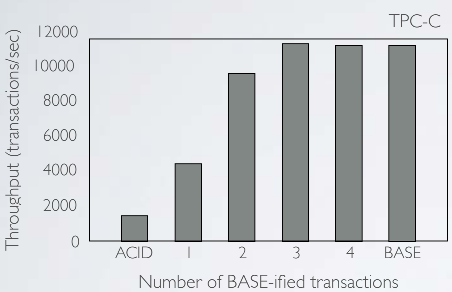
Configuration

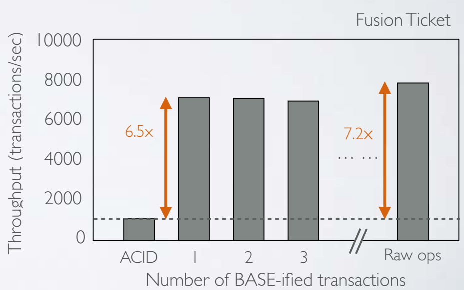
- Emulab Cluster (Dell Power Edge R710)
- 10 shards, 3-way replicated

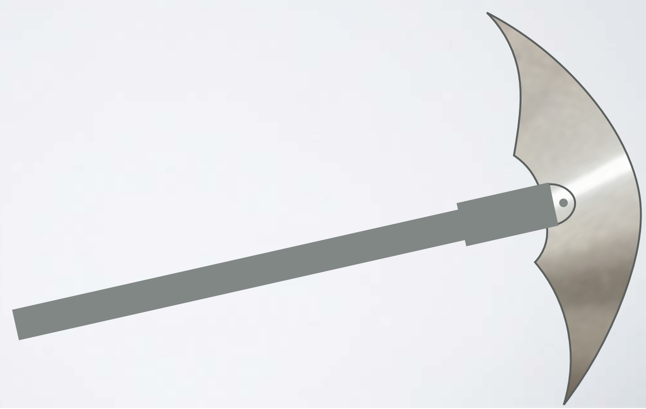


Running on MYSQL Cluster

OGRAMMING EFFORT VS PERFORMANCE







AND YET…

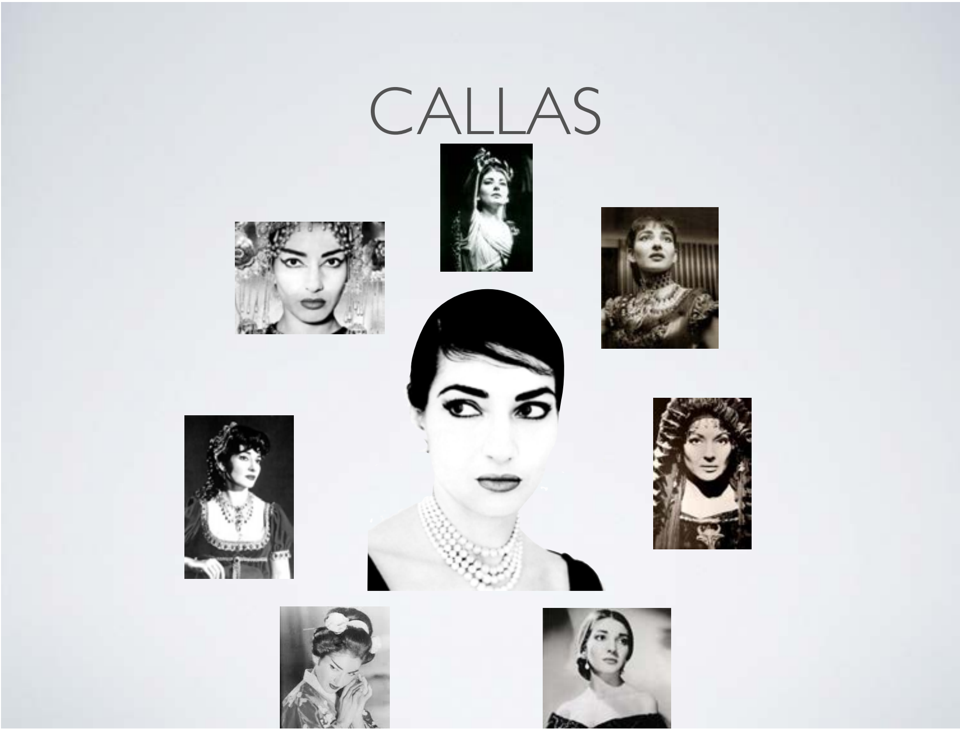
Programming BASE transactions
is still hard!

WHAT DO PROGRAMMERS WANT?

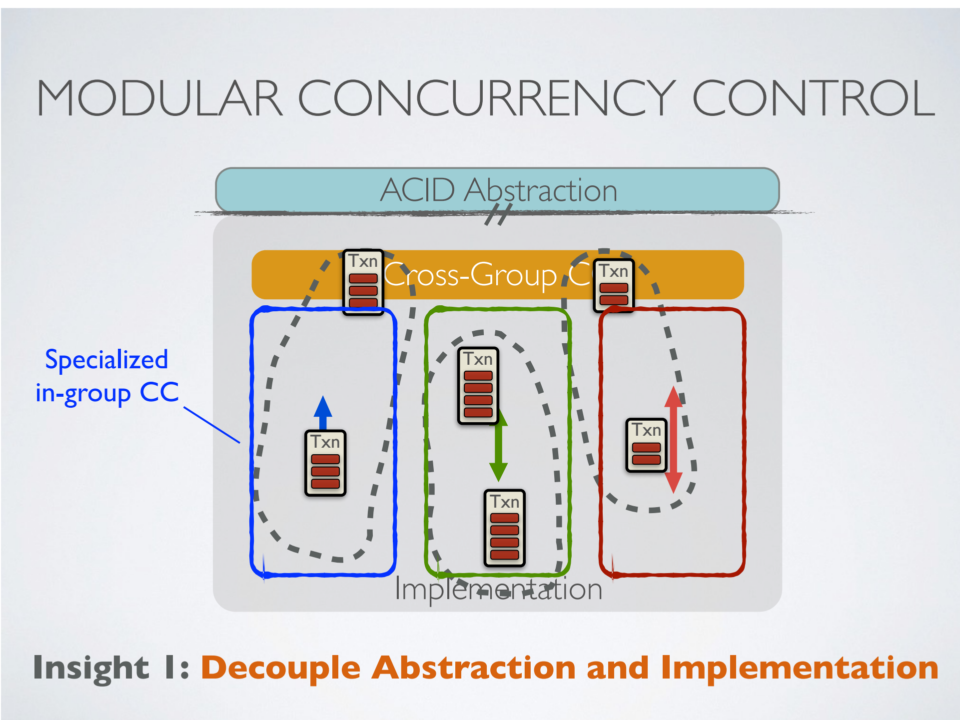
WE LOVE ACID

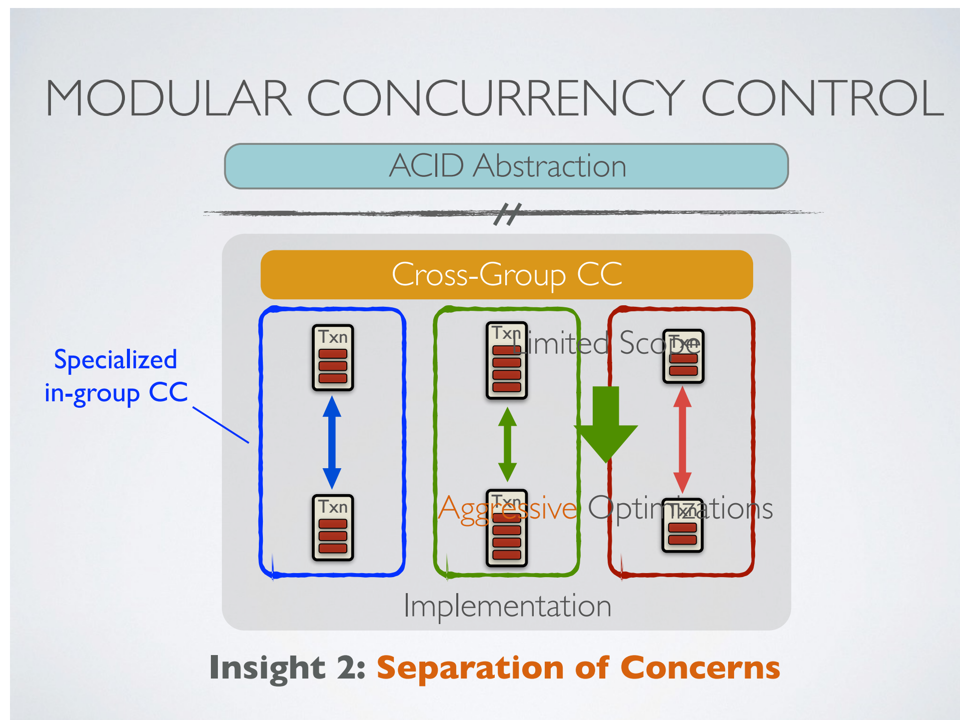
NOT ALL TRANSACTIONS ARE CREATED EQUAL

Use a flexible
implementation
abstraction


CALLAS


THE PRICE OF UNIFORMITY
Conservative Mechanism


MODULAR CONCURRENCY CONTROL
ACID Abstraction
Specialized
in-group CC
Txn
Implementation
Insight 1: Decouple Abstraction and Implementation


MODULAR CONCURRENCY CONTROL
ACID Abstraction
Cross-Group CC
Specialized
in-group CC
Txn
Implementation
Insight 2: Separation of Concerns

CORRECTNESS ACROSS GROUPS

Goal: No dependency cycles over all transactions

1. No cycles within each group

2. No cycles spanning multiple groups

ISOLATION ACROSS GROUPS

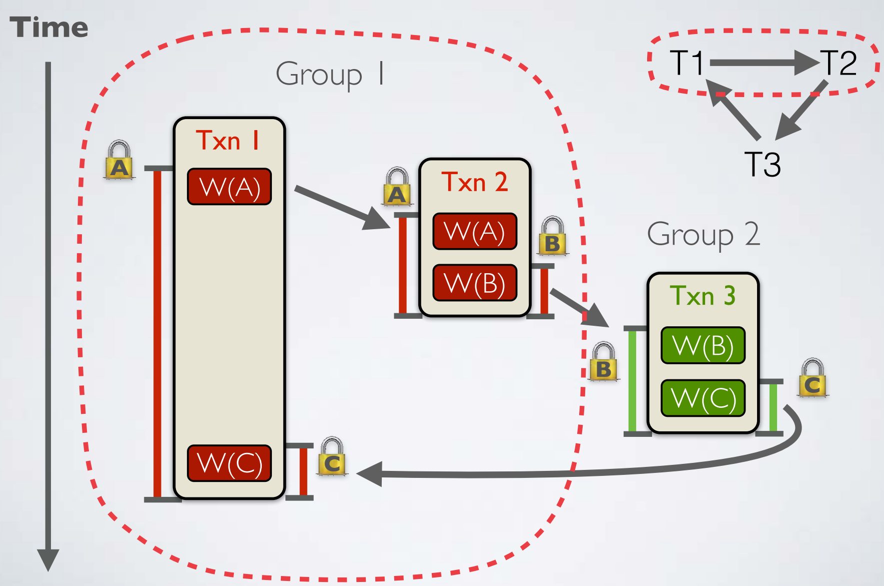
Never conflict for transactions in the same group

Always conflict for transactions in different groups
(unless both reading)

Minimal interference with group-specific CC

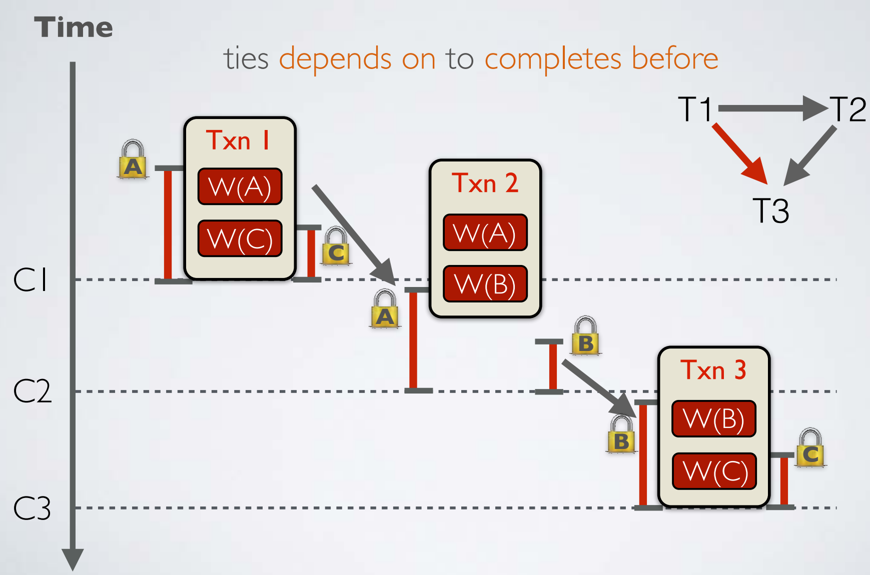
Nexus locks

A SUBTLE PROBLEM

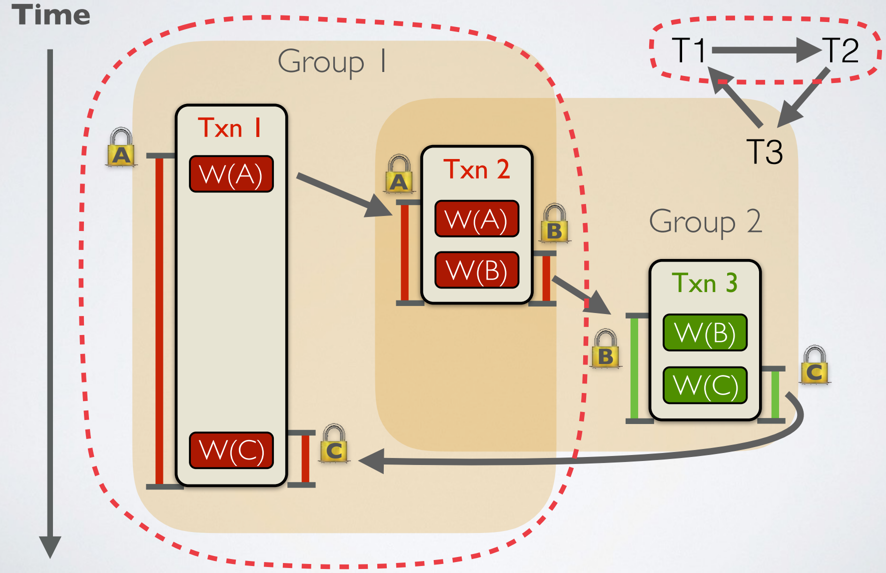


TRADITIONAL LOCKING

ties depends on to completes before



A SUBTLE PROBLEM



A REFINEMENT

T_2 depends on T_1

⬇

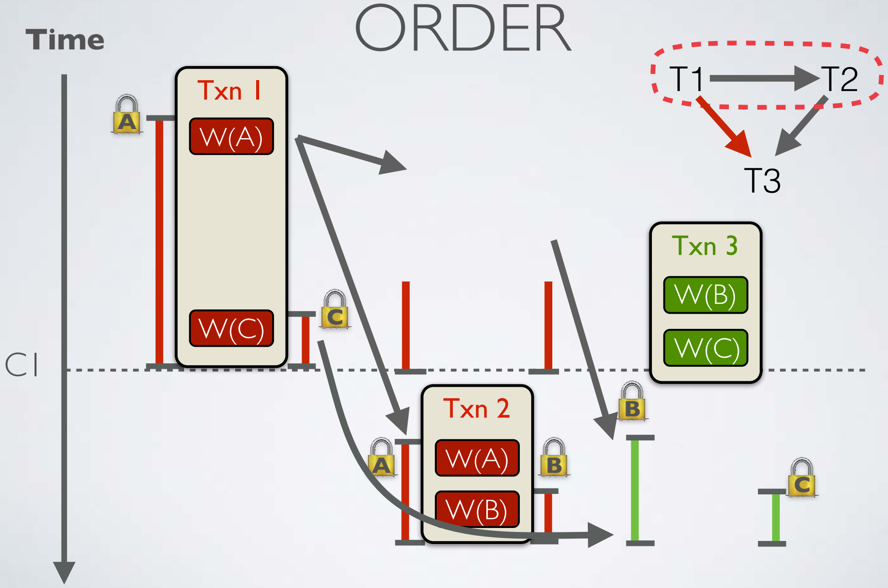
T_2 cannot ~~start~~ before T_1 ~~completes~~

release its nexus locks

releases its nexus locks

Nexus Lock Release Order

ENFORCE LOCK RELEASE ORDER



ISOLATION WITHIN GROUPS

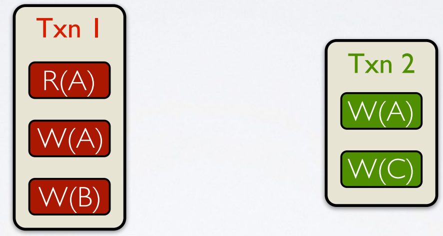
Increase in-group concurrency
while maintaining safety

TRANSACTION CHOPPING

Shasha et al. '95

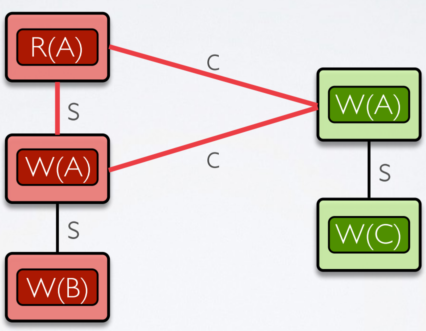
TRANSACTION CHOPPING

Shasha et al. '95



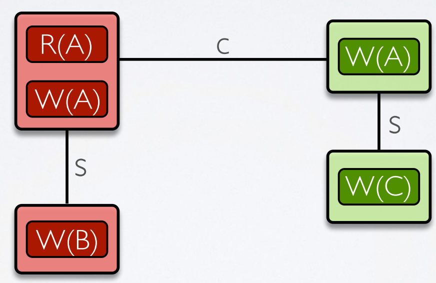
TRANSACTION CHOPPING

Shasha et al. '95



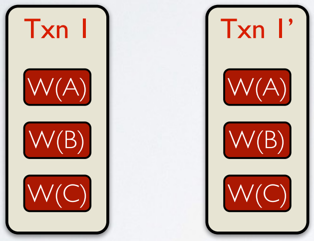
TRANSACTION CHOPPING

Shasha et al. '95

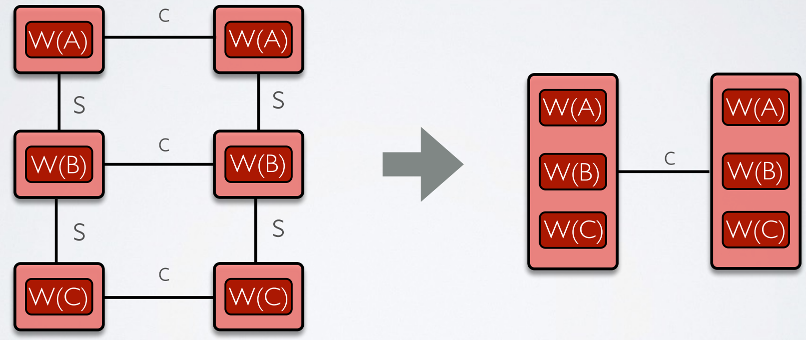


+MCC: No SC cycle ~~among all transactions~~ within each group

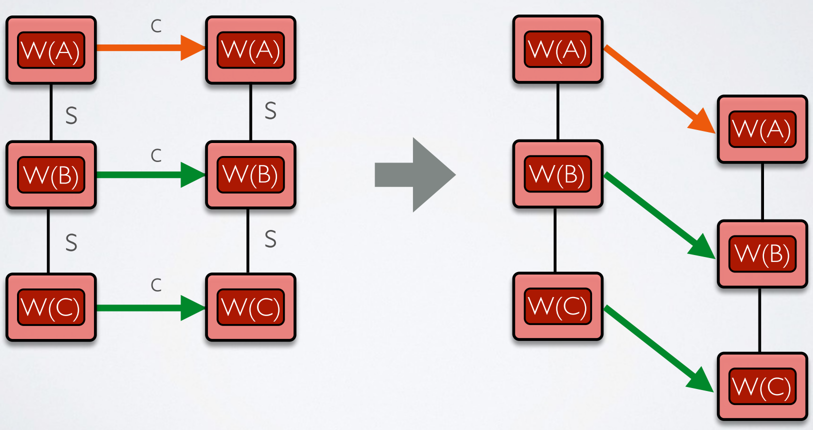
SELF CONFLICT



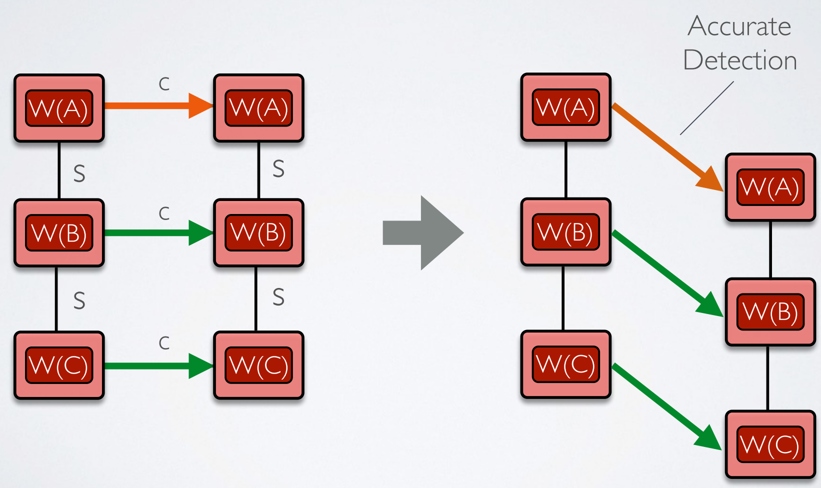
SELF CONFLICT

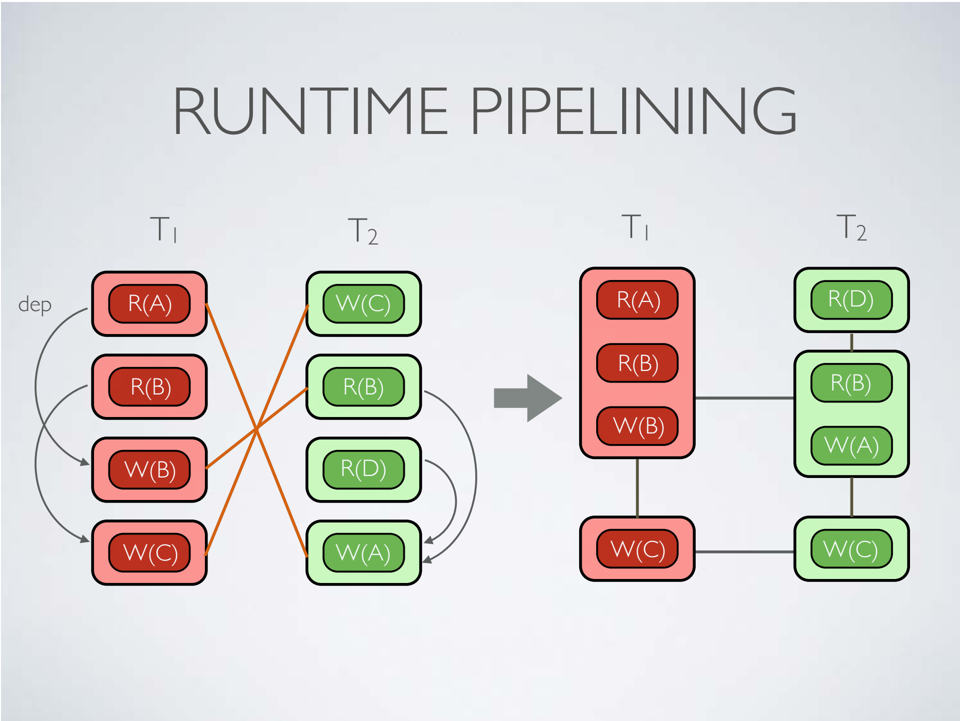


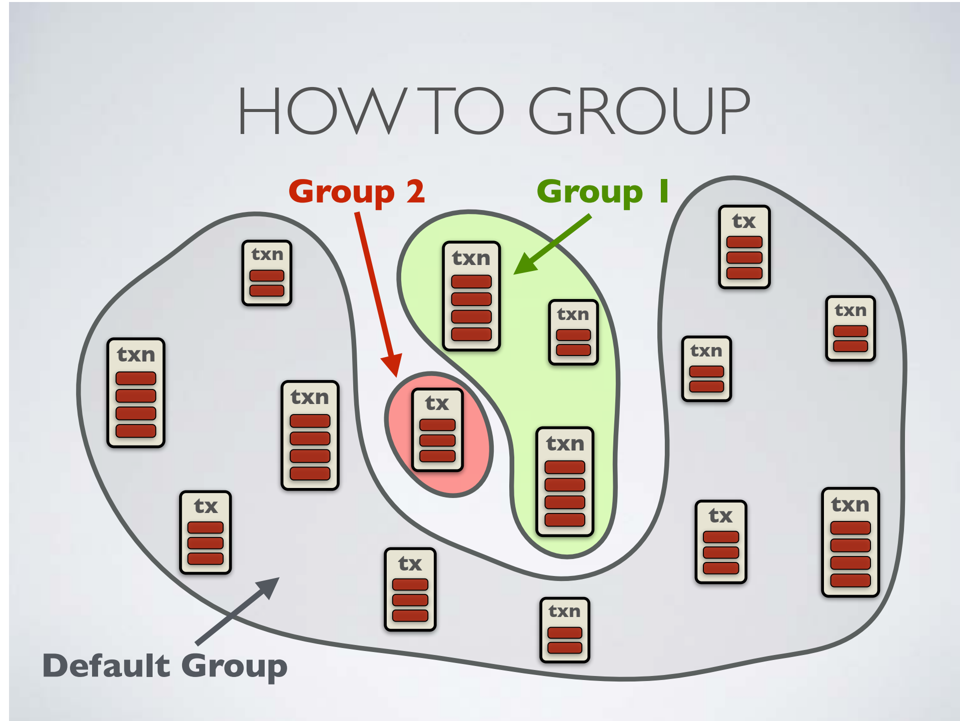
A NEW CC MECHANISM: RUNTIME PIPELINING



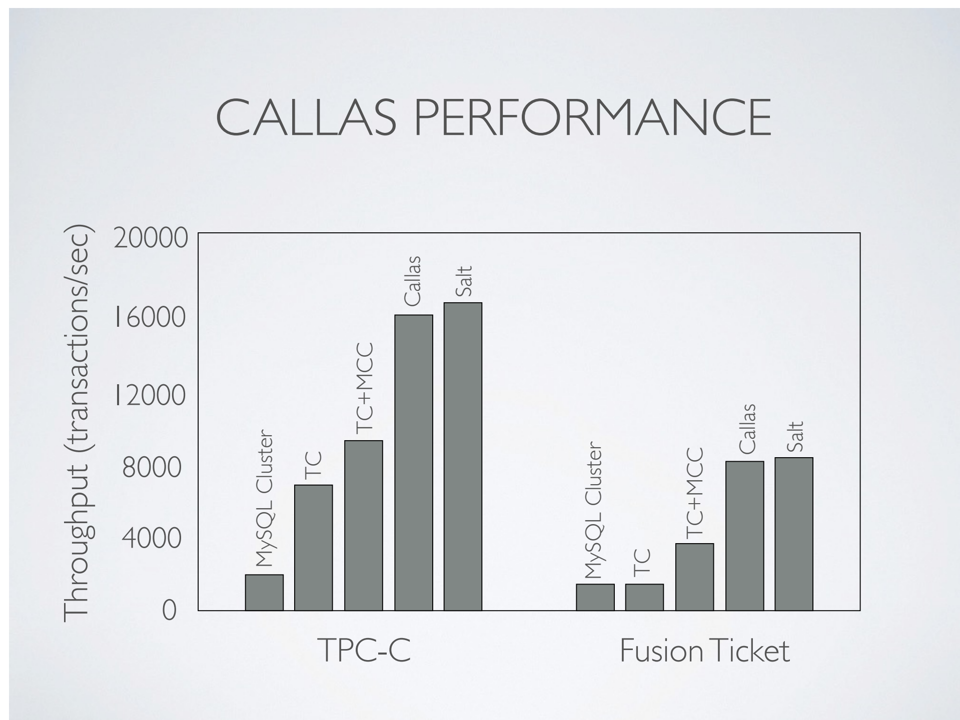
RUNTIME PIPELINING



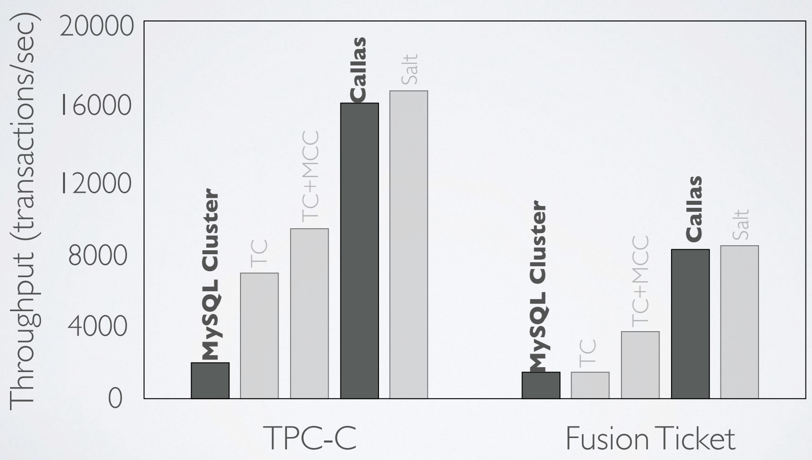

RUNTIME PIPELINING
T1
T2
dep
R(A)
R(B)
W(B)
W(C)
W(C)
R(B)
R(D)
W(A)
T1
T2
R(A)
R(B)
W(B)
W(C)
R(D)
R(B)
W(A)
W(C)


HOW TO GROUP
Group 2
Group 1
Default Group
txn
tx

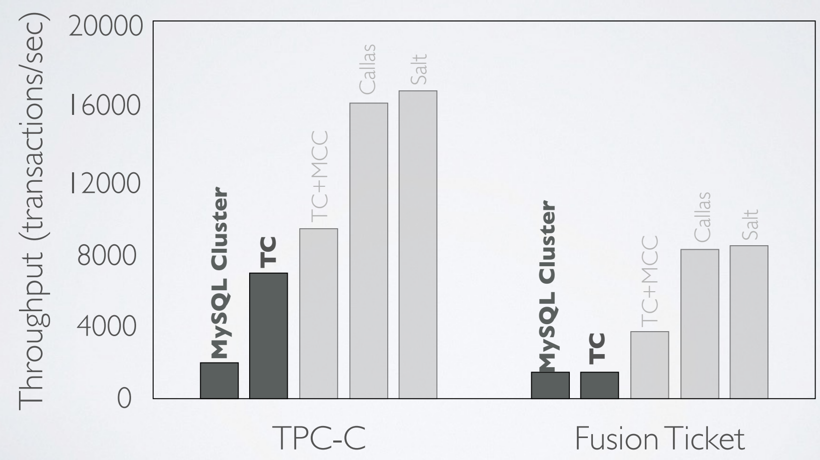

HOW WELL DOES IT WORK?
This talk…
…but of course there's more
End-to-end performance
What are the relative merits of Callas' various optimizations?
What is the overhead of nexus locks?
What is the effect of using different groupings?


CALLAS PERFORMANCE
Throughput (transactions/sec)
20000
16000
12000
8000
4000
0
MySQL Cluster
TC
TC+MCC
Callas
Salt
MySQL Cluster
TC
TC+MCC
Callas
Salt
TPC-C
Fusion Ticket

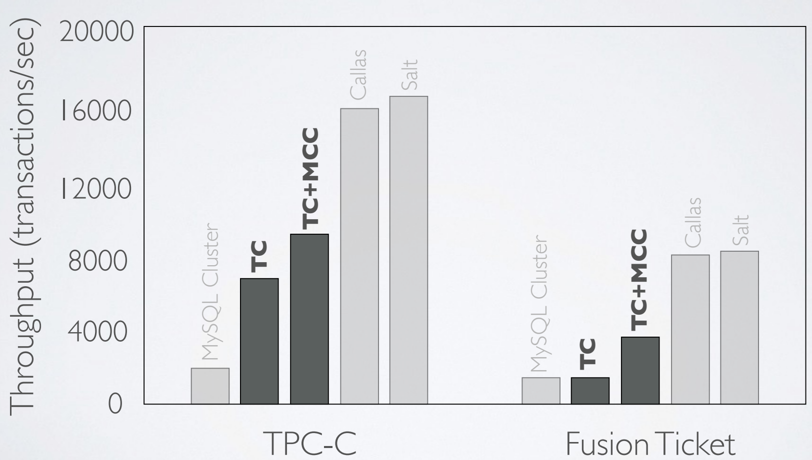
BENEFIT OVER ACID BASELINE



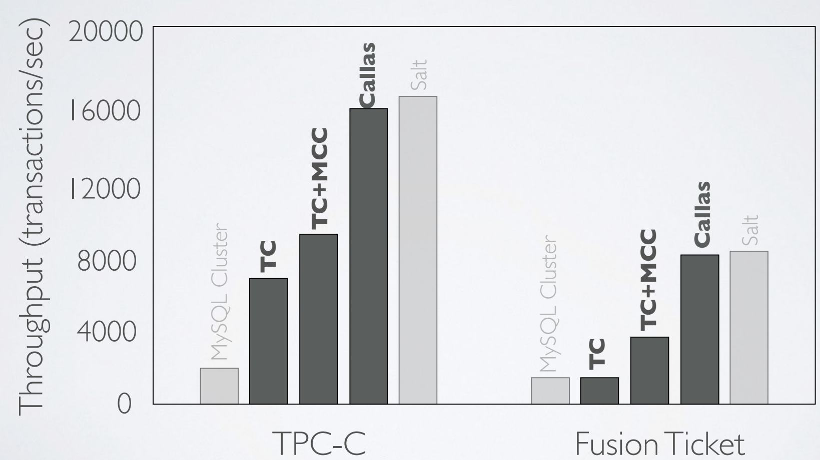
TRANSACTION CHOPPING



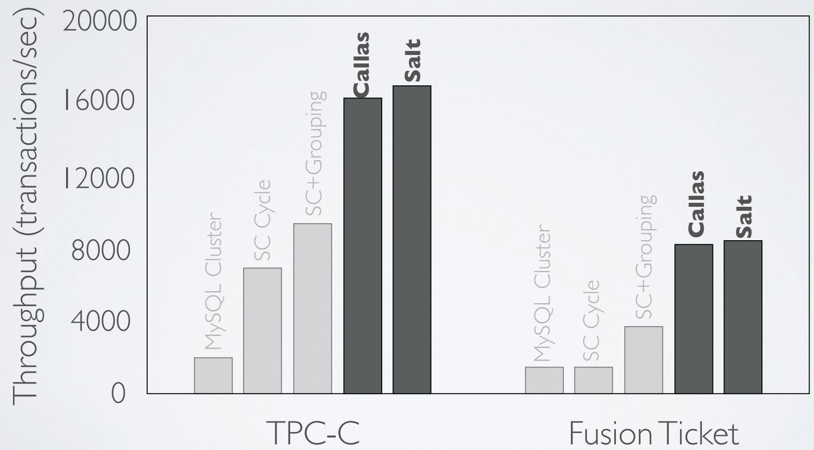
TRANSACTION CHOPPING + MCC



CALLAS: MCC + RUNTIME PIPELINING



WITHIN 5% OF SALT



CONCLUSION