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

ACID TRANSACTIONS: SIMPLE AND POWERFUL

PERFORMANCE VIA WEAKER ISOLATION GUARANTEES

ANSI SQL-92 ISOLATION LEVELS

<table>
<thead>
<tr>
<th>Database System</th>
<th>Default Isolation</th>
<th>Strongest Isolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL Cluster</td>
<td>Read Committed</td>
<td>Read Committed</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>Read Committed</td>
<td>Snapshot Isolation</td>
</tr>
<tr>
<td>Google Spanner</td>
<td>Serializability</td>
<td>Serializability</td>
</tr>
<tr>
<td>VoltDB</td>
<td>Serializability</td>
<td>Serializability</td>
</tr>
<tr>
<td>Oracle 12C</td>
<td>Read Committed</td>
<td>Snapshot Isolation</td>
</tr>
<tr>
<td>MemSQL</td>
<td>Read Committed</td>
<td>Read Committed</td>
</tr>
<tr>
<td>SQL Server</td>
<td>Read Committed</td>
<td>Serializability</td>
</tr>
<tr>
<td>Postgres</td>
<td>Read Committed</td>
<td>Serializability</td>
</tr>
</tbody>
</table>

- 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)
DIRTY READS

Root: Write-Read conflict

- T₁ modifies a data item.
- T₂ reads that data item before T₁ commits or aborts.
- If T₁ then aborts, T₂ 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

- T₁ reads a data item.
- T₂ then modifies or deletes that data item and commits.
- If T₁ 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

- T₁ reads a set of data items satisfying <search condition>.
- T₂ then creates data items that satisfy T₁’s <search condition> and commits.
- If T₁ then repeats its read with the same <search condition>, it gets a different set of data

WHAT’S NOT TO LIKE?

Berenson et al, SIGMOD ’95

- Ambiguous descriptions of proscribed behaviors

Dirty Reads

- Strict Interpretation (prohibits anomaly)
  - A₁: W₁[X] ... R₁[X] ... (A₁ and C₁ in any order)

- Broad Interpretation (prohibits phenomenon)
  - P₁: W₁[X] ... R₁[X] ... (A₁ or C₁) and (A₂ or C₂) 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 $A_1$...
  - $W_1[X] \ldots R_2[X] \ldots (A_1 \text{ and } C_2 \text{ in any order})$
  - $T_1$ does not abort!
- but violates Broad Interpretation $P_1$
  - $W_1[X] \ldots R_2[X] \ldots ((A_1 \text{ or } C_1) \text{ and } (A_2 \text{ or } C_2 \text{ in any order})$

PHENOMENA OR ANOMALIES?

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

WHAT’S NOT TO LIKE?

• ANSI SQL phenomena are weaker than their locking counterpart

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Read Locks</th>
<th>Write Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locking Read Uncommitted</td>
<td>None</td>
<td>Long write locks</td>
</tr>
<tr>
<td>Locking Read Committed</td>
<td>Short* read locks (both)</td>
<td>Long write locks</td>
</tr>
<tr>
<td>Locking Repeatable Read</td>
<td>Long item locks Short predicate locks</td>
<td>Long write locks</td>
</tr>
<tr>
<td>Locking Serializable</td>
<td>Long read locks (both)</td>
<td>Long write locks</td>
</tr>
</tbody>
</table>
ANSI P3 should prevent phantoms due to deletions and updates, not just creations.

**WHAT’S NOT TO LIKE?**

- ANSI SQL phenomena are weaker than their locking counterpart

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Read Locks</th>
<th>Write Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locking Read Uncommitted</td>
<td>None</td>
<td>Long write locks</td>
</tr>
<tr>
<td>Locking Read Committed</td>
<td>Short read locks (both)</td>
<td>Long write locks</td>
</tr>
<tr>
<td>Locking Repeatable Read</td>
<td>Long item locks Short predicate locks</td>
<td>Long write locks</td>
</tr>
<tr>
<td>Locking Serializable</td>
<td>Long read locks (both)</td>
<td>Long write locks</td>
</tr>
</tbody>
</table>

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 $T_1$ commits or aborts.
- Conflicting writes can interleave, violating invariants

**ANSI isolation levels should include phenomenon $P0$**

$P0$: $W_1[X]...W_2[X]...(C_1 or A_1)$ and $(C_2 or A_2)$ in any order
ANSI-92 ISOLATION LEVELS, POST CRITIQUE

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

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 multiversion 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

T1: Change green to red

T2: Change red to green

WRITE SKEW ANOMALY

T1: Change green to red

T2: Change red to green

WRITE SKEW ANOMALY

T1: Change green to red

T2: Change red to green

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 \( T_i \xrightarrow{ww} T_j \) (anti-dependency)
    - read-write \( T_i \xrightarrow{rw} T_j \)
  ‣ edge expresses temporal relation
    - start \( T_i \xrightarrow{\Delta} T_j : c_i < s_j \)

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
  ‣ \( T_2 \) cannot read value of aborted \( T_1 \)

• No intermediate reads
  ‣ \( T_2 \) cannot read value of \( T_1 \) that \( T_1 \) 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?

Why the confusion?

- 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

DON’T KNOW MUCH ABOUT HISTORIES

- Theory
  - Conflict Serializability
  - MySQL Community Edition
- Multiversion Serializability
  - Rococo
- Anomaly Serializability
  - Oracle 12C

Why the confusion?

- Semantic gap between how isolation guarantees are defined and how they are experienced by application programmers.
- In terms of histories and operations, isolation properties are formalized, but applications perceive them differently.

How isolation properties are formalized

How applications perceive them
A STATE-BASED DEFINITION

- Isolation guarantees as **constraints** on read states
  - states consistent with what the application observed
  
  $$R_2(Y_0) \quad R_2(Z_1)$$

- 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

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

$$S_e \xrightarrow{T_0} x_0 \xrightarrow{T_2} x_1 \xrightarrow{T} x_2$$
PARENT STATES AND COMPLETE STATES

• Parent state $s_p$ of $T$: state from which $T$ commits

  $S_e \xrightarrow{T_0} T \xrightarrow{T_2} T \xrightarrow{T}$

• Complete state for $T$: a read state for all read ops in $T$

  $T \xrightarrow{T_0} T \xrightarrow{T_2} T \xrightarrow{T}$

  $T \xrightarrow{R(Z_1)} R(Y_1)$
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) prescribes:

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 $T$

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

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 $T$

$$\exists s \in S_e. \land \text{COMPLETE}_{e,T}(s) \land (\Delta(s, s_p) \cap \mathcal{W}_T = \emptyset)$$
PERFORMANCE VIA WEAKER ISOLATION GUARANTEES

It-That-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

Custom code for better performance
Complexity gets quickly out of control

A CLASSIC HORROR STORY

Ease of Programming

Help!

Database       Programmer
MORE CONCURRENCY!

Time

Transfer

Part 1

c ≥ $10?
c := c-10
s := s+$10

Part 2

Transfer

Part 1

c ≥ $10?
c := c-10
s := s+$10

Part 2

Time

MORE COMPLEXITY!

Transfer

Part 1

c ≥ $10?
c := c-10
s := s+$10

Part 2

Balance

Read c

Read s

MORE COMPLEXITY!

Transfer

Part 1

c ≥ $10?
c := c-10
s := s+$10

Part 2

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

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

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

Vilfredo Pareto

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
Different isolation guarantees for different types of transactions

To BASE transactions:
a sequence of small ACID transactions
To ACID transactions:
a single, monolithic ACID transaction

Fine isolation granularity between BASE transactions

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

PROGRAMMING EFFORT VS PERFORMANCE

Throughput (transactions/sec)

Throughput (transactions/sec)

Number of BASE-ified transactions

Number of BASE-ified transactions

ACID 1 2 3 BASE

ACID 1 2 3 Raw ops

ACID 1 2 3

Fusion Ticket
Programming BASE transactions is still hard!

AND YET…

WHAT DO PROGRAMMERS WANT?

WE LOVE ACID

NOT ALL TRANSACTIONS ARE CREATED EQUAL

Use a flexible implementation
**CALLAS**

- Uniform Abstraction
- Ease of Programming
- Uniformity
- Ease of Programming
- Conservative Mechanism
- Uniform Implementation

**THE PRICE OF UNIFORMITY**

- Uniform Abstraction
- Conservative Mechanism

**MODULAR CONCURRENCE CONTROL**

- ACID Abstraction
- Cross-Group CC
- Implementation

**Insight 1: Decouple Abstraction and Implementation**

- Specialized in-group CC

**MODULAR CONCURRENCE CONTROL**

- ACID Abstraction
- Cross-Group CC
- Implementation

**Insight 2: Separation of Concerns**

- Limited Scope
- Aggressive Optimizations
CORRECTNESS ACROSS GROUPS

Goal: No dependency cycles over all transactions

1. No cycles within each group

2. No cycles spanning multiple groups

A SUBTLE PROBLEM

Time

Group 1

Txn 1
W(A)
W(C)

Txn 2
W(A)
W(B)

Txn 3
W(B)
W(C)

Group 2

T1
T2

T3

TRADITIONAL LOCKING

ties depends on to completes before

Time

C1

C2

C3

Group 1

Txn 1
W(A)
W(C)

Txn 2
W(A)
W(B)

Txn 3
W(B)
W(C)

Group 2

T1
T2

T3

ISOLATION ACROSS GROUPS

Never conflict for transactions in the same group

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

Nexus locks

Minimal interference with group-specific CC
A SUBTLE PROBLEM

Txn 1
W(A)
W(C)

Txn 2
W(A)
W(B)

Txn 3
W(B)
W(C)

T1
T3
T2

Group 1

Group 2

T1 depends on T1

T2 cannot start before T1 completes

release its nexus locks

Nexus Lock Release Order

A REFINEMENT

ENFORCE LOCK RELEASE ORDER

Txn 1
W(A)
W(C)

Txn 2
W(A)
W(B)

Txn 3
W(B)
W(C)

T1
T3
T2

Isolation within groups

Increase in-group concurrency while maintaining safety
TRANSACTION CHOPPING

Shasha et al. '95

Static Analysis

No SC cycle among all transactions
A NEW CC MECHANISM: RUNTIME PIPELINING

RUNTIME PIPELINING

Accurate Detection
RUNTIME PIPELINING

HOW TO GROUP

HOW WELL DOES IT WORK?

CALLAS PERFORMANCE

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?

Throughput (transactions/sec)

MySQL Cluster TC TC+MCC Callas Salt

MySQL Cluster TC TC+MCC Callas Salt

TPC-C Fusion Ticket
WITHIN 5% OF SALT

Throughput (transactions/sec)

<table>
<thead>
<tr>
<th>Throughput (transactions/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL Cluster</td>
</tr>
<tr>
<td>TPC-C</td>
</tr>
</tbody>
</table>

CONCLUSION

Cross-Group CC

Implementation

ACID Abstraction

txn

txn

txn