Developer’s Responsibility or Database’s Responsibility?
Rethinking Concurrency Control in Databases

Chaoyi Cheng*, Mingzhe Han*, Nuo Xu, Spyros Blanas, Michael D. Bond, Yang Wang
Ideal World vs. Reality

• Ideal world: Serializable transactions
  • Developers don’t need to worry about concurrency issues

• Reality: Most applications don’t use serializable transactions
  • Many use weaker isolation levels (READ COMMITTED, etc.)
  • Some use ad-hoc transactions (i.e., individual SQL statements + external concurrency control mechanisms)
Ideal World vs. Reality

“Buy One Item” transaction

1. If (quantity > 0):
2. Update (quantity = quantity - 1)

“The unit test always passes but in my production database quantity is sometimes -1, please investigate.”
Ideal World vs. Reality

“Buy One Item” transaction

1. If (quantity > 0):
2. Update (quantity = quantity - 1)

“The unit test always passes but in my production database quantity is sometimes -1, please investigate.”

Under Read Committed:

<table>
<thead>
<tr>
<th></th>
<th>TX1</th>
<th>TX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reads 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Reads 1</td>
</tr>
<tr>
<td>3</td>
<td>Updates to 0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Commits</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Updates to -1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Commits</td>
</tr>
</tbody>
</table>
Overview

• A study of real-world concurrency bugs
  • Looked at 93 bugs from 46 open-source applications
  • Understand their consequences, root causes, and their fixes

• What can we do?
A Study of Real-world Concurrency Bugs

Questions

• Do weakly isolated or ad-hoc transactions actually cause anomalies in real-world applications?

• How much effort does it require to handle these anomalies?

• Why aren’t people more eager to use Serializable transactions?
  • Maybe the contention level is low and correctness issues don’t arise?
  • Maybe the occasional data inconsistency is OK and can be manually fixed?
  • Maybe fixing correctness issues in an application is easy for its developers?
A Study of Real-world Concurrency Bugs

Methodology

• Investigated active, open-source database applications
  • Domains: e-commerce, gaming, chatting, ORM tools
• Inspected bug reports, discussions and code commit history
  • No specific keyword or phrase
  • Started from broad search: “SQL”, “transaction”, “red committed”, “race condition”, “concurrent”, “for update”, “duplicate”, ...
  • Manually inspected each bug
  • Ignored deadlock-related issues, unless related to isolation level
A Study of Real-world Concurrency Bugs

Outcome
• Found 93 isolation bugs in 46 different applications
• Identified consequence, root cause, and fix for each bug

• Full bug list: http://go.osu.edu/isolation-bug-study
A Study of Real-world Concurrency Bugs

Limitation: selection bias

• Looked at open-source applications only
  • Corollary: most bug reports on open-source database systems

• Keyword search limitation
  • May have over-represented behaviors that can be described succinctly
  • May have missed bugs that are described only in domain-specific terms

• We cannot answer what percentage of applications experience concurrency issues
  • But we know that it affects many applications
Consequences of concurrency bugs

- Most common inconsistencies:
  - Wrong count (30)
  - Duplicate IDs (16) – big problem!

- New insight: Mostly idle applications are also impacted!
  - “About 2-5% chance this happens. I have about 100 orders per day.”
  - “It is rare, but it has happened 5 times over the past few months in a set of 10,000 orders”.

![Pie chart showing distribution of issues: Inconsistency (78), Unavailability (13), Others (2).]
Root causes of concurrency bugs

- Read followed by relevant write:
  - Single row check-then-update (42)
    - If (quantity > 0), then
      Update (quantity = quantity – 1)
  - Multi-row check-then-update (10)
    - INSERT (SELECT MAX(id) + 1)
- Inappropriate error handling:
  - None/insufficient exception handling (9)
  - Exception over-handling (1)
- Unnecessary concurrency:
  - TXs don’t commit in order issued (3)
    - No isolation level provides this guarantee!
How developers fix concurrency bugs

No universally acceptable solution!

- FOR UPDATE: 22
- Add constraint: 8
- Additional locking: 11
- Additional versioning: 11
- Check before commit: 8
- Serial execution: 7
- Stronger isolation: 6
- Weaker isolation: 4
- Exception handling: 3
- Business process fix: 3
- Others: 3

No universally acceptable solution!
Answering the questions

• Do weakly isolated or ad-hoc transactions actually cause anomalies in real-world applications? Yes.
Answering the questions

• How much effort does it require to handle these anomalies? **A lot.**

1. Reproduction effort: **Moderate.**
   • Most bugs involve very few transactions.
   • Nondeterministic bugs, so “luck” and patience needed to hit.
     • Common tricks to increase conflict rate: increase concurrency, slow transactions down

2. Diagnosis effort: **Low.**
   • Very little discussion or uncertainty on the cause.

3. Fix and verify effort: **Very high.**
   • Lengthy discussions debating possible fixes and pros/cons of each.
   • First suggestions are often flawed or introduce more issues.
Answering the questions

• Why not Serializable transactions? **Long locking.**

• Common concern is performance
  • Reasons of poor performance not a focus of our study

• Error cases are very important too
  • Failures do not propagate through layers → wasted resources

• Deadlock detection is very sensitive to timeout parameter
  • One application implemented a queue-based solution (serial execution) to eliminate guessing of deadlock timeouts in deployment
What can we do?

• Short-term solutions

• How much can automatic analysis help?

• How can we incorporate developer effort to fixing concurrency bugs?

• Closing thought: the role of developer education
Short-term solutions

• Point to database systems that implement:
  • Snapshot Isolation
  • Unique/auto-incremented IDs
  • Optimistic concurrency control

• Likely a non-starter for the open-source applications we studied

• But stronger isolation will not address all problematic behaviors:
  • Duplicate inserts/deletes
  • Transaction ordering assumption
Can automatic analysis help?

• Consider two transactions on rows A, B, C:
  T1: Write(A), Read(B)
  T2: Write(A), Read(C)

• Lock-based CC will block on write lock for A
• Optimistic CC will abort one transaction, due to write conflict on A

• Automatic analysis can easily determine that execution is serializable, no matter how operations interleave
  • Theoretically possible to run with no concurrency control*

  *latching still needed
Can automatic analysis help?

• Can source code analysis identify such “no concurrency control necessary” phases in actual executions?
  • Essentially: identify when isolation is guaranteed by query semantics

• Challenges:
  • Sound static analysis struggles to be precise
  • Dynamically-typed languages are harder to analyze precisely, tools are not as advanced
  • Extracting query predicates from SQL is non-trivial
  • Tension between abstract CC model and vendor-specific CC implementation
    • Problematic behaviors on an abstract model may not occur in practice, and vice versa

• Opportunities:
  • Application logic is often very simple, does not stress scalability of static analysis
  • No data sharing in application logic, avoids imprecise analysis due to shared objects
How can we incorporate developer effort?

• Developers can convey application-specific constraints and invariants that automatic analysis will never reliably extract

• Examples:
  • Transaction is for reporting purposes and does not need serializable results
  • No concurrent transactions from same client
  • Which tables or records are likely to be hot

• Challenges:
  • How are we presenting concurrency bugs in an understandable way?
    • How can we reliably track executions through software layers?
  • How do we incorporate user feedback and allow user control of concurrency?
  • Can we automatically suggest modifications to SQL in application code?
Closing thought: developer education

• Application developers are much more comfortable at the programming language level rather than the database level.
  • Noted widespread use of the term “race condition” for weak isolation errors.
• First developer reaction is: “let’s add a lock somewhere”
  • Locking is undergraduate concept. Easy to understand, hard to get right.
• Challenges in introducing isolation earlier in the curriculum:
  1. Early definitions of isolation are operational: they dictate how to implement using locks
  2. Dependency graph-based definitions of isolation are hard to grasp, even harder to use to analyze real transactions
Conclusion

• A study of real-world concurrency bugs
  • Looked at 93 bugs from 46 open-source applications
  • Understand their consequences, root causes, and their fixes:
    • Consequence: Most bugs manifest as data inconsistencies
    • Root cause: Mainly the assumption of atomic read-then-write
    • Fix: No universally acceptable solution

• What can research do?
  • Automatic analysis is a very promising path
  • New abstractions to convey concurrency bugs and incorporate user input