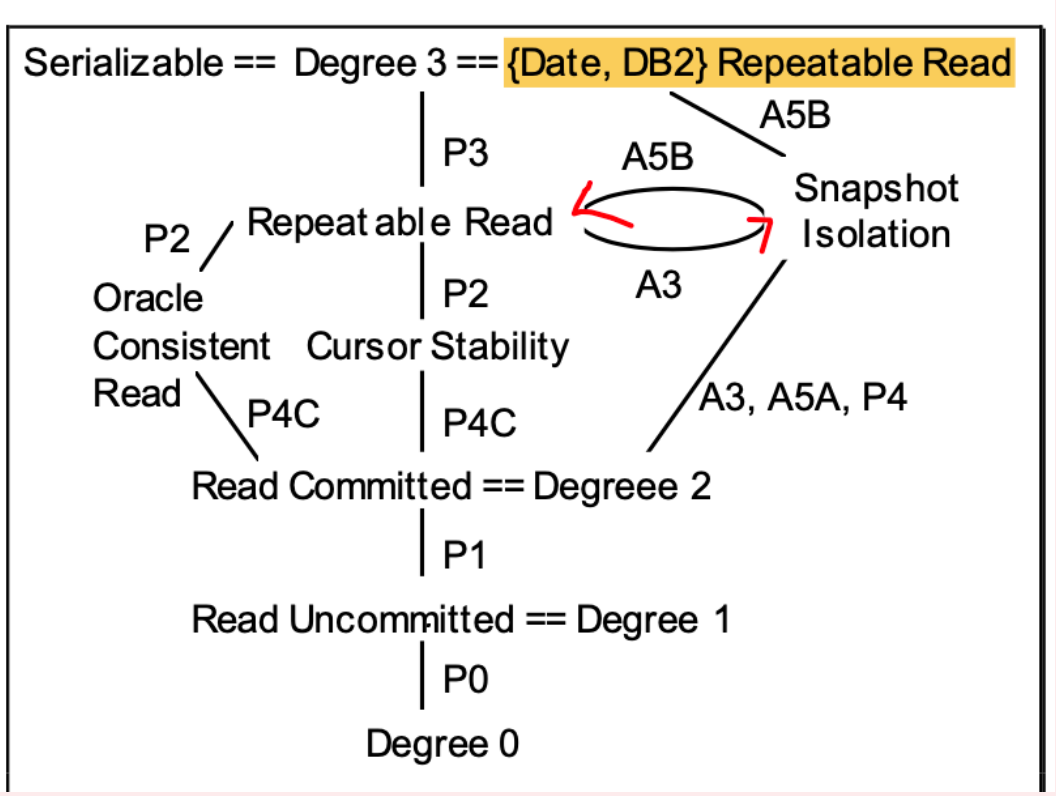
Chapter 7. Transactions

Date: @March 23, 2023 9:40 PM

Topic: Chapter 7. Transactions

Recall	Notes
In the harsh reality of data systems, many things can go wrong	• software or hardware may fail • application may crash • Network Interruption • Clients write concurrently • Read partially updated data • Race conditions
Commonly used isolation levels?	*read committed*, *snapshot isolation*, and *serializability*.

📌 **SUMMARY:**
For decades, *transactions* have been the mechanism of choice for simplifying these issues. A transaction is a way for an application to group several reads and writes together into a logical unit. Conceptually, all the reads and writes in a transaction are executed as one operation: either the entire transaction succeeds (*commit*) or it fails(*abort*, *rollback*). Transactions are created with a purpose: to *simplify the programming model* for applications accessing a database. By using transactions, the application is free to ignore specific potential error scenarios and concurrency issues.



Date: @March 23, 2023 9:40 PM

Topic: The Slippery Concept of a Transaction

Recall

Notes

New words

- status quo
- Slippery
- hype
- new crop of
- antithesis
- hyperbole
- acronym
- ambiguity
- devil
- penalty
- volatile

ACID

Atomicity: All actions in txn happen, or none happen.

"All or nothing…"

Consistency: If each txn is consistent and the DB starts consistent, then it ends up consistent.

"It looks correct to me…"

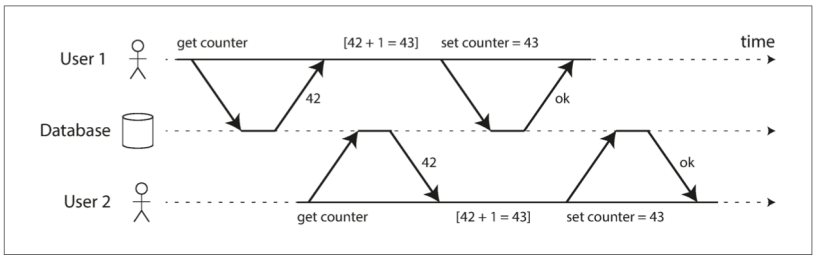


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

Isolation: Execution of one txn is isolated from that of other txns as if there is only one in executing.

"All by myself…"

Durability: If a txn commits, its effects persist.

"I will survive…"

📌 **SUMMARY:**
The safety guarantees provided by transactions are often described by the well-known acronym *ACID*, which stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability.*
Race condition problem:

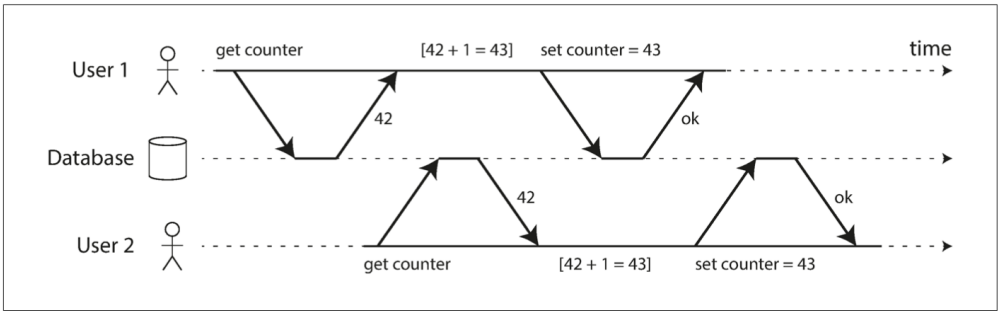


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

Date: @March 31, 2023 7:39 PM

Topic: Single-Object and Multi-Object Operations

Recall	Notes
Read uncommitted: dirty anomaly	

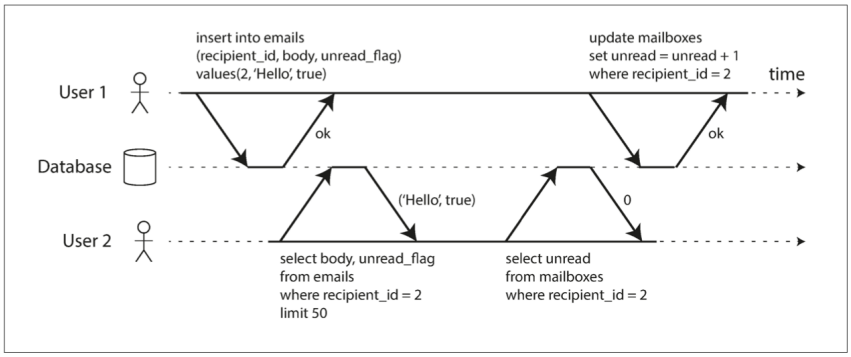
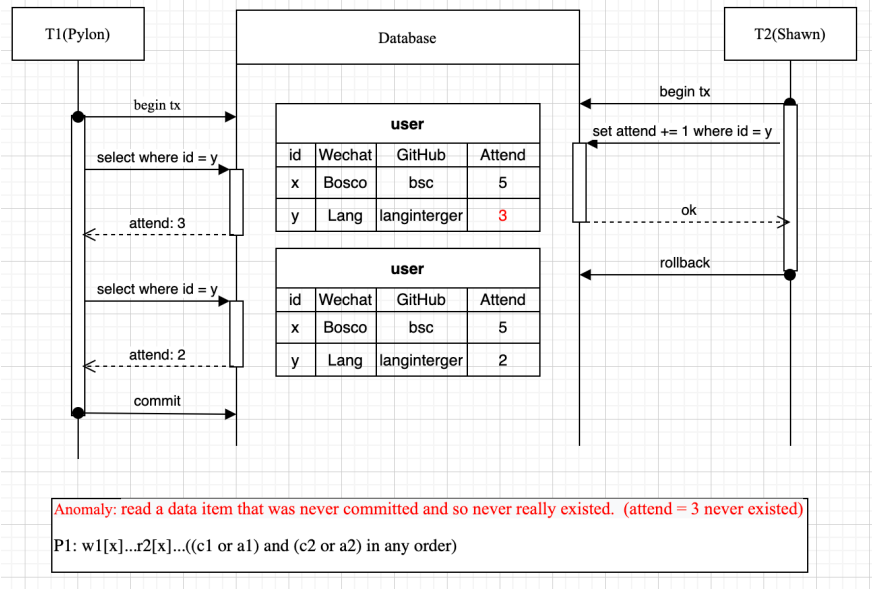


Figure 7-2. Violating isolation: one transaction reads another transaction's uncommitted writes (a "dirty read").



Single-object writes

single object also needs atomicity & isolation, e.g. write a 20KB JSON:

1. network interruption after half-finished
2. power fails
3. partial read.

📌 **SUMMARY:**

Date: @April 8, 2023 7:50 PM

Topic: Weak Isolation Levels

Recall	Notes
Weak Isolation Levels	1. Read Committed 2. Repeatable Read 3. Snapshot Isolation

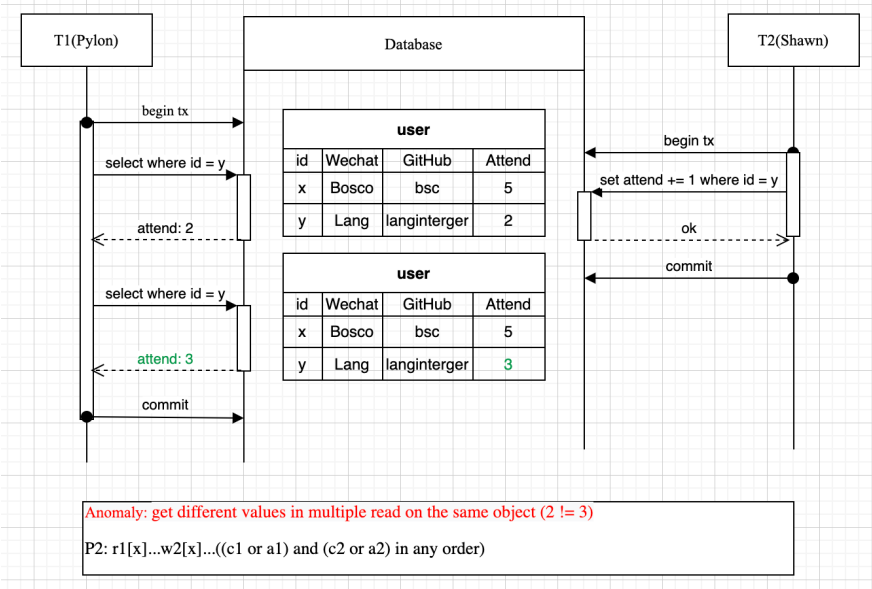
4. Serializable

📌 **SUMMARY:**
In theory, isolation should make your life easier by letting you pretend that no concurrency is happening: *serializable* isolation means that the database guarantees that transactions have the same effect as if they ran *serially.*
Serializable isolation has a performance cost, and many databases don't want to pay that price. It's therefore common for systems to use weaker levels of isolation, which protect against *some* concurrency issues, but not all.

Date: @April 8, 2023 7:54 PM

Topic: Read Committed

Recall	Notes
It makes two guarantees	• no dirty write • no dirty read
Has fuzzy read anomaly	



📌 **SUMMARY:**
prevent dirty writes by using row-level locks: two-phase locking is the most common approach.

Date: @April 8, 2023 7:58 PM

Topic: Snapshot Isolation & Repeatable Read

Recall	Notes
Guarantees	1. no fuzzy read
No guarantees: • Read Skew • Write Skew • Phantom	Read Skew

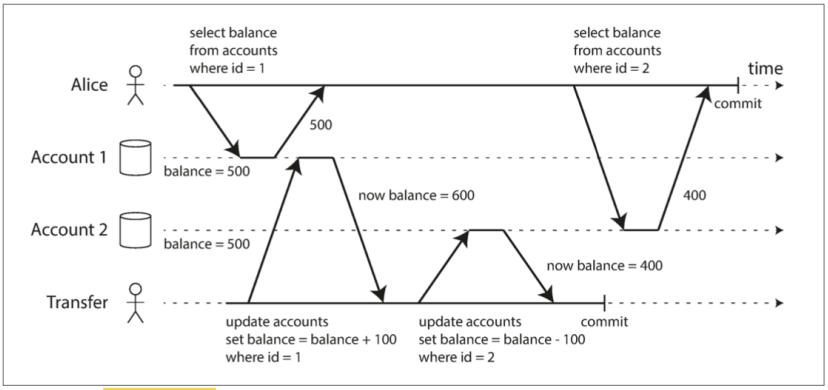
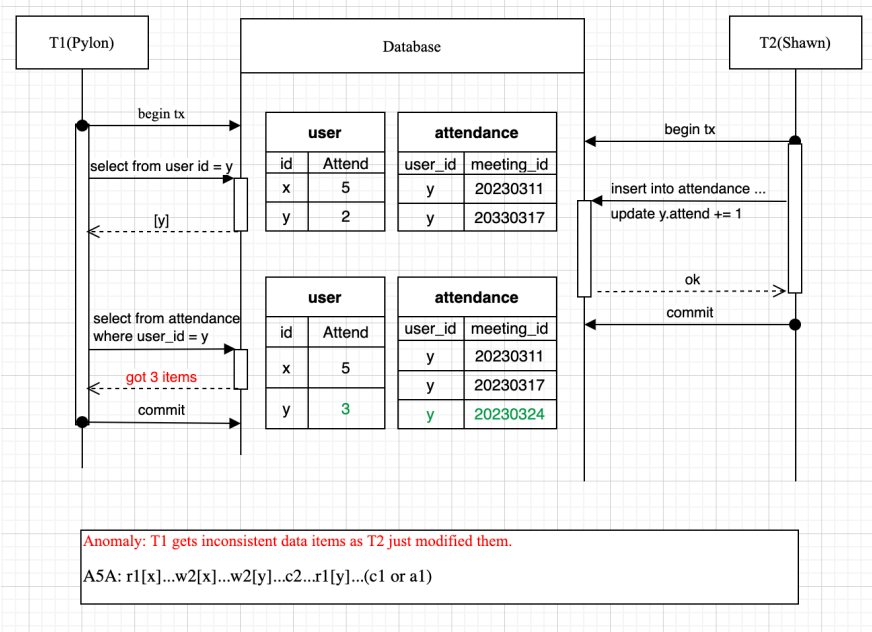
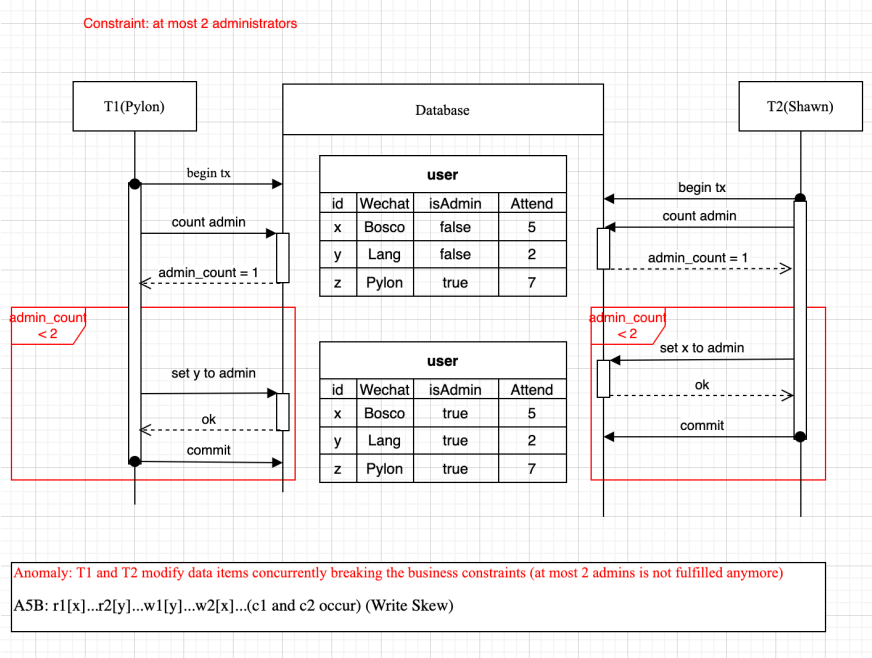


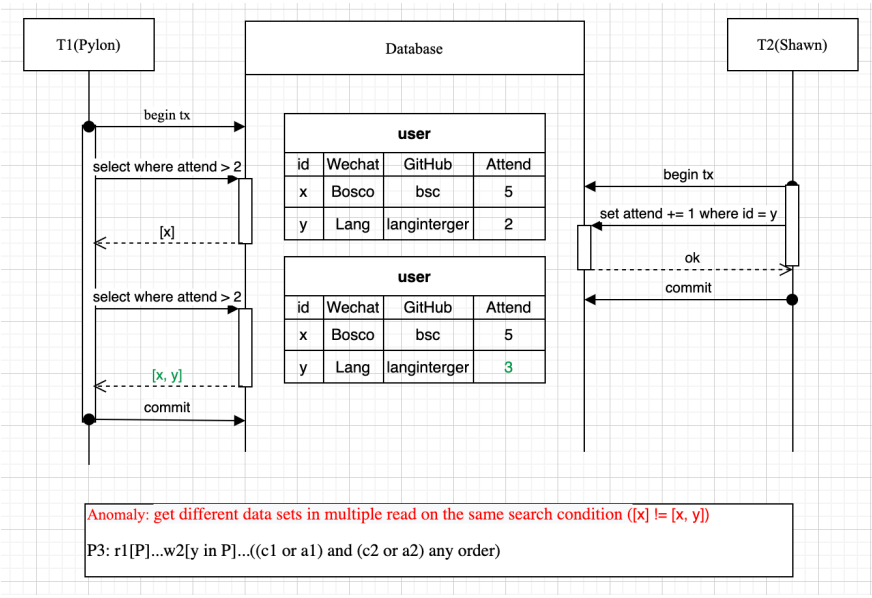
Figure 7-6. Read skew: Alice observes the database in an inconsistent state.



Write Skew



Phantom



📌 **SUMMARY:** a key principle of snapshot isolation is *readers never block writers, and writers never block readers.*

Date: @March 29, 2023 9:03 PM

Topic: Preventing Lost Updates

Recall	Notes
lost update	• two transactions do write concurrently • in a read-modify-write cycle • the second write overrides the first write
lost update scenarios	1. incrementing a counter 2. making a local change to a complex value. 3. collaborative editing of a wiki
Atomic write operations	change read-modify-write ⇒ one operation MongoDB provides automatic operations for making local modifications to a part of a JSON document Redis provides atomic operations as well by adding an exclusive lock on the read object until the update to the object has been finished. ⇒ cursor stability

cursor stability?	e.g.
	`SELECT * FROM figures`
	`WHERE name = 'robot' AND game_id = 222`
	`FOR UPDATE;`

automatically detecting lost updates	monitoring the lost updates and aborting the transaction and forcing it to retry its read-modify-write cycle. e.g. • PostgreSQL’s repeatable read • Oracle’s serializable • SQL Server’s snapshot isolation

Techniques to prevent lost update	#1. Atomic operation #2. Explicit locking #3. Automatically detecting lost updates and fixing #4. Compare-and-set
	issue: old → new → old solution: add a version number

ABA issue

> 📌 **SUMMARY:**
> The read committed and snapshot isolation levels we've discussed so far have been primarily about the guarantees of what a read-only transaction can see in the presence of concurrent writes.
> **Atomic operations** and locks are ways of preventing lost updates by forcing the read-modify-write cycles to happen sequentially.

Date: @March 29, 2023 9:53 PM

Topic: Write Skew & Phantoms

Recall Notes

Write skew

r1[x]...r2[y]...w1[y]...w2[x]...
(c1 and c2 occur)

Suppose T1 reads
x and y, which are

consistent with C(), and then a T2 reads x and y, writes x, and commits. Then T1 writes y. If there were a constraint between x and y, it might be violated.

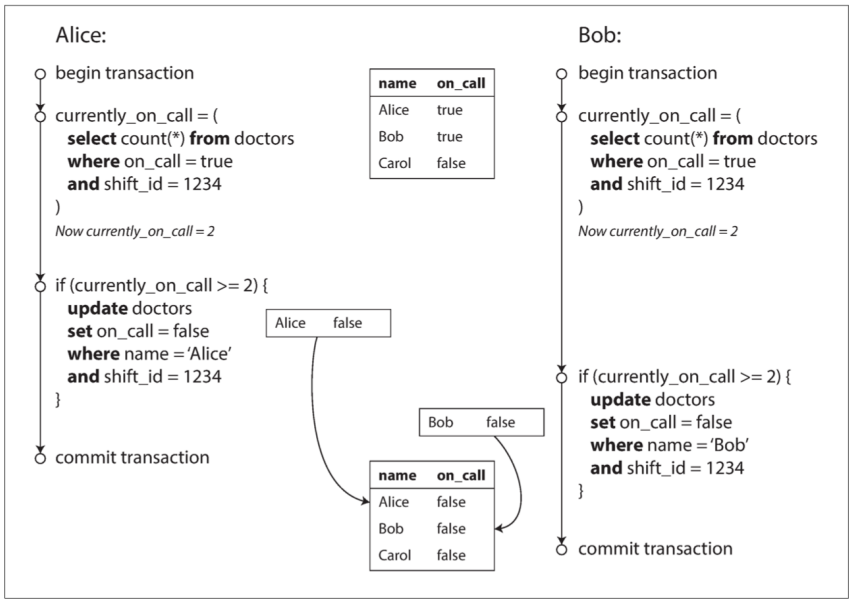
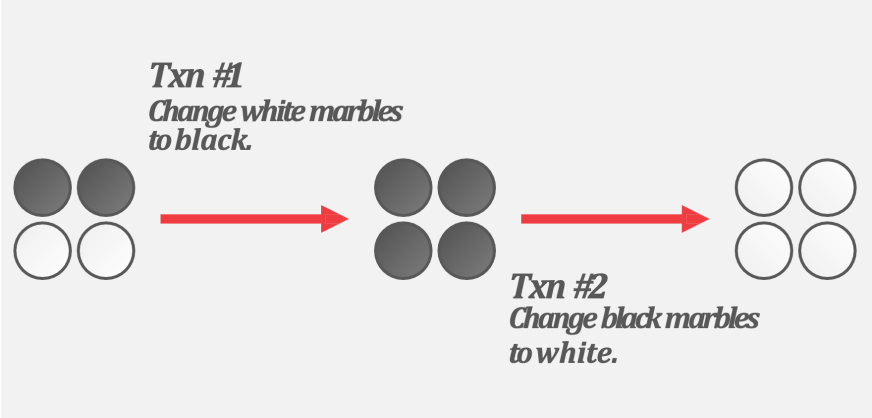
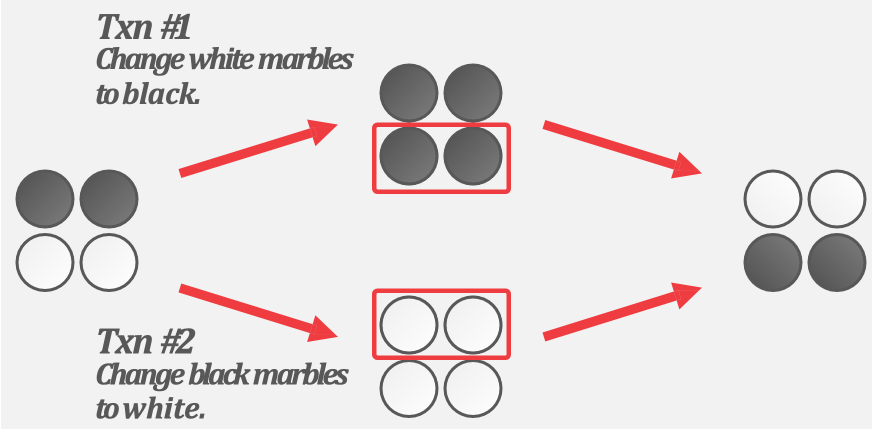


Figure 7-8. Example of write skew causing an application bug.

Expected with serial execution order:



Anomaly with write skew:



How to prevent write skew?

#1. explicitly lock

#2. built-in constraints

📌 **SUMMARY:**

Date: @April 8, 2023 8:05 PM

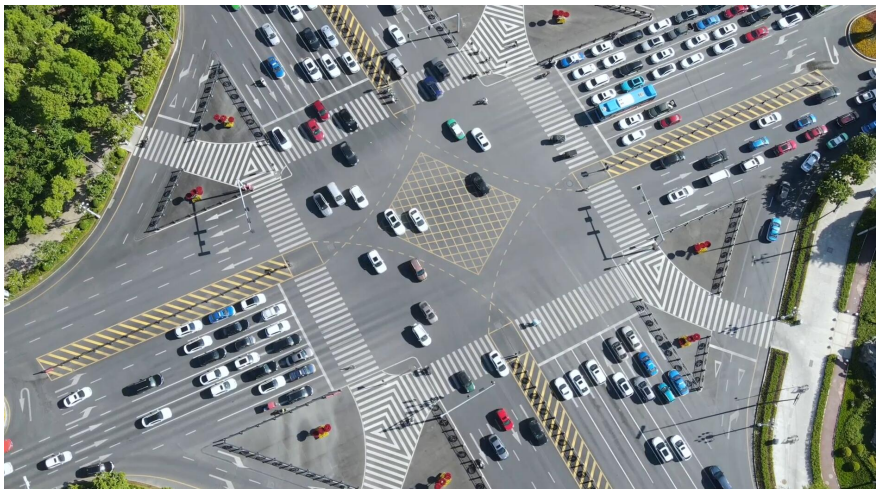
Topic: Serializability

Recall	Notes
Serializable isolation	Serializable isolation is usually regarded as the strongest isolation level. It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, *serially*, without any concurrency.
Conflicts	

Solution 1:
Actual Serial
Execution

Pessimistic

Solution 2:
Two-Phase
Locking



Solution 3: Serializable Snapshot Isolation(ISS)

📌 **SUMMARY:**
Serializable isolation is usually regarded as the strongest isolation level. It guarantees
that even though transactions may execute in parallel, the end result is the same as if
they had executed one at a time, *serially*, without any concurrency.
On top of snapshot isolation, SSI adds an algorithm for detecting serialization conflicts among writes and determining which transactions to abort.

In order to provide serializable isolation, the database must detect situations in which a transaction may have acted on an outdated premise and abort the transaction in that case.

Checklist

Question 1: What are the three types of anomalies that can occur in a database when there is no isolation?

Question 2: What is write skew, and how does it differ from dirty writes and lost updates?

Question 3: What are the options for preventing write skew?

- explicitly lock & serialization isolation

Question 4: What is a phantom in the context of database anomalies?

Question 5: What is materializing conflicts, and when should it be considered?

Question 6: How does snapshot isolation work?

Question 7: Why is serializable isolation preferable for preventing write skew?

Question 8: What is the lost update problem?

Question 9: What is a dirty write?

Question 10: In what situations can write skew occur?

Question 11: How can SELECT FOR UPDATE be used to prevent write skew?

Question 12: Why can't atomic single-object operations help prevent write skew?

Question 13: What is the difference between repeatable read and snapshot isolation?

Question 14: How can a unique constraint be used to prevent write skew in certain situations?

Question 15: Why is it difficult to use triggers or materialized views to implement constraints involving multiple objects in a database?