Database Race Condition visualizer

Adrian Szacsko

Analysis

Database race conditions represent a critical challenge in the realm of database management systems, where multiple concurrent transactions attempt to access and modify shared data simultaneously. These conditions can lead to unpredictable and erroneous outcomes, posing a significant threat to data integrity, consistency, and overall system reliability.

In order to maintain consistency, databases follow the **ACID** properties (Atomicity, Consistency, Isolation, Durability). The key component for eliminating race conditions is the **Isolation** property, which defines how a given **transaction** is visible to other users and systems. Transactions in **DBMS** implement isolation levels, which define the degree of isolation from data modification. The degree of isolation is constructed from the following phenomena:

- **Dirty Read** - transaction reads data that has not yet been committed
- **Non Repeatable Read** - transaction reads the same row twice and gets different results
- **Phantom Read** - two same queries are executed, but rows retrieved are different

According to these phenomena, SQL databases define the following isolation levels:

- **Read Uncommitted** - transaction may read not yet committed changes
- **Read Committed** - guarantees, that any data read is committed
- **Repeatable Read** - transactions holds read locks on all rows it references together with write locks for update and delete operations
- **Serializable** - concurrently executing operations appear to be serially executing

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantoms
Read Uncommitted	❌	❌	❌
Read Committed	✅	❌	❌
Repeatable Read	✅	✅	❌
Serializable	✅	✅	✅

- May Occur - ❌
- Don't Occur - ✅

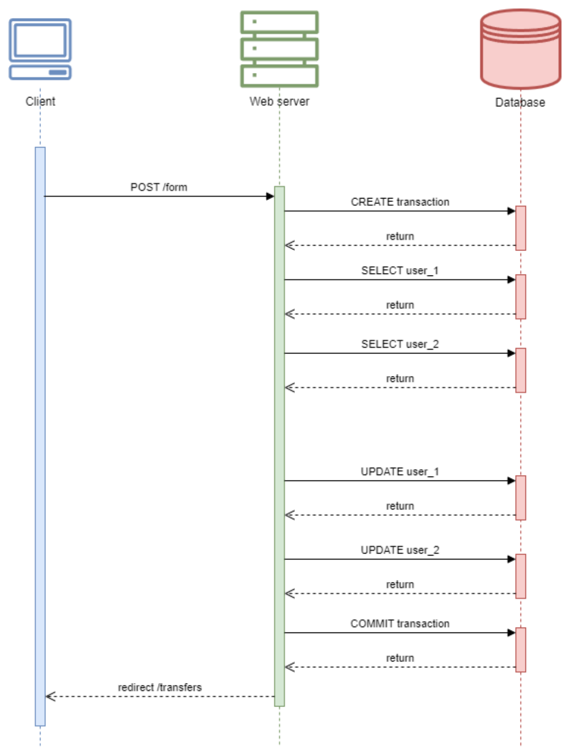
Project Description

This project offers two different ways to explore what race conditions in databases are.

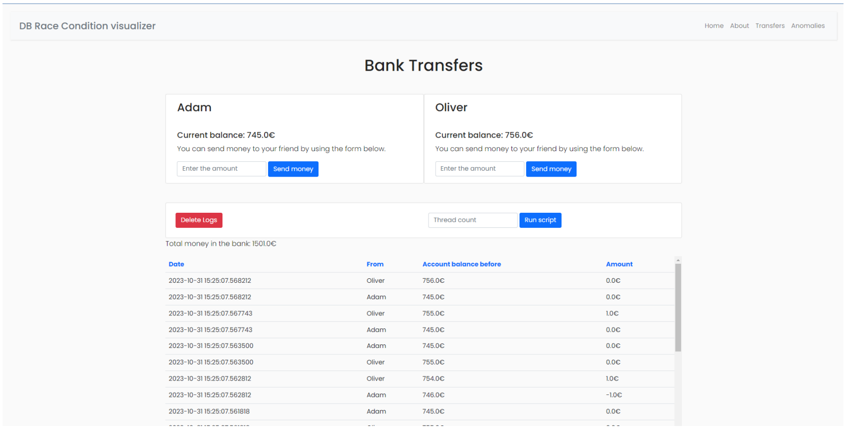
Functionality 1

It houses an exploit web page, where a user can try out an already functional exploit. This exploit sends requests through the web server in an attempt to set the database to an inconsistent state (It sends requests to transfer money from one account to another).

The exploit works by sending a `/forms` request repeatedly. The following sequence diagram describes the function of the given endpoint.



By sending many requests at the same time, Some of the requests may start processing, while the others are between the two `UPDATE`, hence the inconsistent database state.
This race condition visualizer page also supports printing out logs, which are generated at each `UPDATE` through `triggers`. These logs help to catch timestamps when an inconsistency happen.



Bank Transfers

Adam

Current balance: 745.0€

You can send money to your friend by using the form below.

Oliver

Current balance: 756.0€

You can send money to your friend by using the form below.

Total money in the bank: 1501.0€

Date	From	Account balance before	Amount
2023-10-31 15:25:07.568212	Oliver	756.0€	0.0€
2023-10-31 15:25:07.568212	Adam	745.0€	0.0€
2023-10-31 15:25:07.567743	Oliver	755.0€	1.0€
2023-10-31 15:25:07.567743	Adam	745.0€	0.0€
2023-10-31 15:25:07.563500	Adam	745.0€	0.0€
2023-10-31 15:25:07.563500	Oliver	755.0€	0.0€
2023-10-31 15:25:07.562812	Oliver	754.0€	1.0€
2023-10-31 15:25:07.562812	Adam	746.0€	-1.0€
2023-10-31 15:25:07.561818	Adam	745.0€	0.0€

Due to the speed of databases, graphs about the inconsistent states were not produced. For such graphs, a simulation of transactions needs to be implemented (2nd functionality of this project)

Functionality 2

Due to the fact, that generating graphs in real time did not produce consistently readable graphs with timestamps, the second part was constructed. This webpage consists of different tests conducted on the database, which shows how different isolation levels handle race conditions.

In PostgreSQL, there are 3 different isolation levels that can be set: `Read Committed, Repeatable Read, Serializable`. The test cases along with the results on which the database's isolation levels were tested are the following:

Test	Read Committed	Repeatable Read	Serializable
Write Cycles	✅	✅	✅
Aborted Reads	✅	✅	✅
Intermediate Reads	✅	✅	✅
Circular Information Flow	✅	✅	✅

Test	Read Committed	Repeatable Read	Serializable
Predicate Many Preceders	❌	✅	✅
Lost Update	❌	✅	✅
Read Skew	❌	✅	✅
Write Skew	❌	❌	✅
Anti-Dependency Cycles	❌	❌	✅

- May Occur - ❌
- Don't Occur - ✅

Write Cycles

Description	Consequence
In this anomaly, two or more transactions concurrently attempt to write to the same data item without proper synchronization.	The final state of the data may depend on the order in which the transactions commit, leading to inconsistent or incorrect data.

Aborted Reads

Description	Consequence
A transaction reads data that is modified by another transaction that is later aborted. As a result, the reading transaction sees uncommitted changes.	The reading transaction may use data that is not consistent with the committed state of the database, leading to incorrect results.

Intermediate Reads

Description	Consequence
A transaction reads data modified by another transaction before that transaction is committed. This results in reading uncommitted, intermediate data.	The reading transaction may base decisions on incomplete or inconsistent data, leading to incorrect results.

Circular information flow

Description	Consequence
A circular chain of transactions reads and writes data, with each transaction reading data modified by the previous one. This can result in circular dependencies.	Circular dependencies can lead to inconsistent or undefined states in the database, causing data integrity issues.

Predicate many preceders

Description	Consequence
Multiple transactions are concurrently trying to update or delete data that meets a certain predicate (e.g., all records with a specific condition), leading to contention.	This can result in performance bottlenecks and contention, as multiple transactions compete for access to the same data.

Lost Update

Description	Consequence
Two or more transactions read the same data and then update it independently without coordination. One of the updates is lost because it overwrites the changes made by the other transaction.	Data updates can be lost, leading to inconsistencies in the database, as one of the updates is discarded.

Read Skew

Description	Consequence
Multiple transactions read a set of data items and then update a subset of those items independently, creating a dependency cycle.	This can lead to read skew anomalies, where one transaction's reads are inconsistent with another's writes, potentially causing incorrect results.

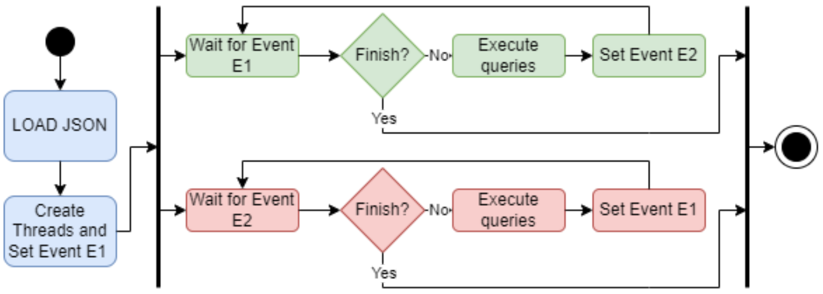
Write Skew

Description	Consequence
Two transactions read different sets of data items but update an item that is read by the other transaction. This can lead to an anti-dependency cycle.	Write skew anomalies can occur, where one transaction's writes affect another's reads, causing inconsistent and incorrect data.

Anti-Dependency Cycles

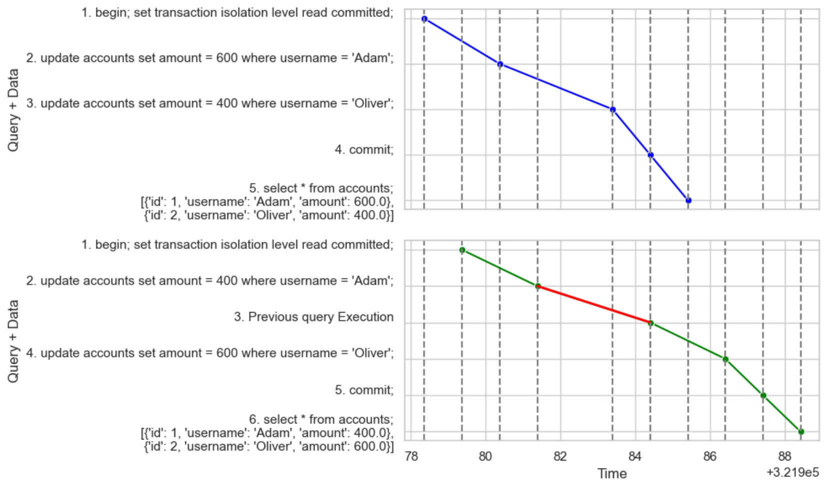
Description	Consequence
Multiple transactions read data based on a predicate (e.g., all records meeting a certain condition) and update data that overlaps or matches the predicate, leading to anti-dependency cycles.	This can result in write skew anomalies, where the overlapping updates of multiple transactions lead to inconsistent and incorrect data.

The Python script loads a JSON file, in which the SQL queries are stored. The implementation works in multithreaded mode (one thread per transaction) and waits after each query batch execution for the second thread/transaction. This way, the race condition at a given isolation level can be tested without time management. For flexibility, the queries are stored in a JSON file, so anytime new test cases can be added.



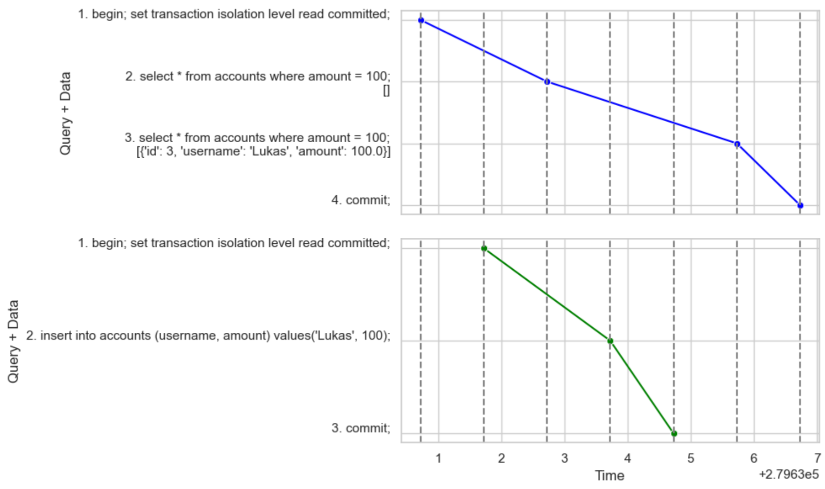
Example graph 1: Write Cycles

This graph shows the result for the anomaly **Write Cycles**. The graph shows, that on isolation level **Read Committed** the 2. query for transaction 2 `Update accounts set amount = 400 where username = 'Adam';` is being held back until the first transaction executes `commit;`. This means, that the PostgreSQL database at isolation level `Read committed` or higher handles Write Cycles anomaly correctly.



Example graph 2: Predicate-Many-Preceders

This graph shows the result for the anomaly Predicate-Many-Preceders. According to the graph, when a transaction inserts a new row into the database, other transactions can see them even before it is committed. That means, that the `Read committed` isolation level does not handle this anomaly.



The website was created on top of a FastAPI sample webpage: https://github.com/shinokada/fastapi-web-starter
The anomalies described were found on: https://github.com/ept/hermitage