Database Race Conditions Explained

A Visual Guide to Right Skew & Phantoms

1. Right Skew

Right Skew occurs when two transactions read the same data and then update different parts of it. Their individual actions seem safe, but their combined result violates a key system rule (an invariant) because they don't lock each other out.

Scenario: The On-Call Doctors

The Rule (Invariant): At least one doctor must be active at all times.
Initial State: Dr. Oz and Dr. Toboggan are both active.

The Race Condition

Transaction 1 (Dr. Oz)
1. Reads count of active doctors (sees 2).
2. Decides it's safe to leave.
3. Sets own status to `inactive`.

Transaction 2 (Dr. Toboggan)
1. Reads count of active doctors (sees 2).
2. Decides it's safe to leave.
3. Sets own status to `inactive`.

The Result: Invariant Violation

CRITICAL FAILURE: Zero doctors are now active!

The Solution: Lock the Intention

To fix this, a transaction must lock all rows involved in its decision-making, often using a `SELECT FOR UPDATE` query. This forces other transactions with conflicting locks to wait.

Corrected Flow Description

1. Dr. Oz's transaction starts and places a lock on all active doctor rows.
2. Dr. Toboggan's transaction starts and attempts to lock the same rows, but must wait.
3. Dr. Oz safely sets his status to inactive and commits his transaction. The locks are released.
4. Dr. Toboggan's transaction can now acquire the locks. It re-reads the data and sees only one doctor is active.
5. The application logic correctly prevents Dr. Toboggan from going inactive. The invariant is saved!

2. Phantoms

A Phantom appears when one transaction reads data, and a second transaction inserts a *new* row that would have matched the first transaction's read. The first transaction is now haunted by a "phantom" row it never knew existed, leading to incorrect assumptions.

Scenario: The Last Cupcake

The Rule (Invariant): Only one person can claim an item.
Initial State: The "Cupcake" item has not been claimed, so no row for it exists in the orders table.

The Race Condition

Transaction 1 (Jordan)
1. Checks if a "Cupcake" order exists (finds none).
2. Assumes it's available.
3. Inserts a new row to claim the cupcake.

Transaction 2 (Donald)
1. Checks if a "Cupcake" order exists (finds none).
2. Assumes it's available.
3. Inserts a new row to claim the cupcake.

The Result: Invariant Violation

CRITICAL FAILURE: Two people have claimed the same cupcake!

The Solution: Materialize the Conflict

The fix is to ensure a row exists to be locked for every *possible* item, even if it's not yet claimed. We "materialize" the potential conflict into a physical row that transactions can fight over.

Corrected Flow Description

1. First, create a `bakery_stock` table with a pre-populated row for "Cupcake" (and all other items).
2. Jordan's transaction starts and locks the "Cupcake" row in the `bakery_stock` table.
3. Donald's transaction starts and tries to lock the same row, but must wait because it's already locked.
4. Jordan successfully claims the cupcake (e.g., by updating the stock row or inserting into an `orders` table) and commits. The lock is released.
5. Donald's transaction proceeds, sees the cupcake is now claimed by Jordan, and correctly fails. The invariant is saved!

Summary

Today we're covering Right Skew and Phantoms, two types of database race conditions. While another common issue is "lost updates" (where two transactions read a value, and one's update is overwritten), we'll focus on these two.

Right Skew Summary

Imagine an emergency room where the rule is at least one doctor must be active. If two doctors, Dr. Oz and Dr. Toboggan, both see that two doctors are active and decide to go inactive at the same time, the rule is broken. The problem is that while they each lock their own row to update it, they don't lock the other rows that are part of the overall rule.

The solution is to expand the scope of the lock. When a doctor wants to go inactive, their transaction must first lock all other active doctors. This way, when Dr. Oz and Dr. Toboggan try to leave simultaneously, they both attempt to grab the same set of locks. Only one transaction will succeed, and the other will have to wait, preventing the invariant from being violated.

Phantoms Summary

Now, let's consider Phantoms. Imagine a bakery where two people, Jordan and Donald, both try to claim the last cupcake. They both check the database, see that no one has claimed a cupcake yet, and both proceed to write a new row claiming it. The issue here is that the row they want to lock—the cupcake claim—doesn't exist yet. You can't lock something that isn't there. This is a "Phantom" problem: new rows are written that conflict with each other.

The solution is called "materializing conflicts." This means we pre-populate the database with a row for every possible item in the bakery, including the cupcake. Now, a row exists for the cupcake, even if it's unclaimed. When Jordan and Donald try to claim it, they must first grab the lock on this pre-existing cupcake row. Only one can win the lock, and that person gets to claim the cupcake, preventing the conflict.