Concurrency Prevention and Control

November 20, 2023

1 Isolation Levels in a Database Transaction:

Isolation levels in database transactions provide a way to control how transactions interact with each other, ensuring data consistency and preventing certain concurrency issues. All the examples below are presented under the default postgresql database isolation level: Read Committed

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Figure 1: Postgresql isolation levels.

1.1 Dirty Read:

A dirty read occurs when a transaction reads data that has been modified by another uncommitted transaction. This can lead to incorrect or inconsistent information being presented to users. The use of a higher isolation level, such as Read Committed or Repeatable Read, can prevent dirty reads by ensuring that transactions only read committed data.In PostgreSQL, dirty reads don't occur even when the isolation level is set to READ UNCOMMITTED.

1.2 Non Repeatable Read:

A non repeatable read happens when a transaction reads the same data twice but gets different values each time due to changes made by another committed transaction in between. This inconsistency can lead to incorrect conclusions or decisions based on the read data. Isolation levels like Repeatable Read or Serializable can address nonrepeatable reads by ensuring that data read during a transaction remains constant until the transaction is complete.

```
// Transaction A Initial snack amount: 0
Snack.transaction do
  a = Snack.find(1)
  a.update(amount: '1000')
  a.save
end
```

```
// Transaction B
Snack.transaction do
  b = Snack.find(1)
  puts "Transaction B - Initial amount: #{b.amount}"
  // Initial amount: 0 serve

  sleep(10)

  b = Snack.find(1)
  puts "Transaction B - Updated amount: #{b.amount}"
  // Updated amount after delay: 1000 serve
end
```

1.3 Phantom Read:

Phantom reads occur when a transaction reads a set of rows that satisfy a certain condition, but another committed transaction inserts, updates, or deletes rows that meet that condition before the first transaction completes. This can result in unexpected or missing data in the subsequent read. Higher isolation levels, particularly repeatable-read/serializable, help mitigate phantom reads by preventing changes that could affect the result set until the transaction is finished.

```
// Transaction A: Initial snack amount with "1 serve" is 1
Snack.transaction do
  a = Snack.find(2)
  a.update(amount: '1 serve')
  a.save
end

// Transaction B: Using read-committed isolation level
Snack.transaction(isolation: :read_committed) do
  // Initially, Transaction B reads and finds 1 recor with "1 serve" amount
  puts Snack.where(amount: '1 serve').count // Count after reading: 1

  sleep(10)

  // After Transaction A's update, Transaction B reads and finds 2 records with '1 serve'
  // amount
  puts Snack.where(amount: '1 serve').count // Count after reading: 2
end
```

1.4 Serialization Anomaly:

A serialization anomaly, also known as a lost update problem, arises when multiple transactions concurrently update the same data, leading to one transaction's changes being overwritten by another's. This can result in data loss and incorrect outcomes. To address this, using an isolation level like Serializable ensures that transactions are executed in a way that prevents simultaneous updates to the same data, thus maintaining data consistency.

```

```

```
// Transaction A: Initial snack calories is 200
Snack.transaction(isolation: :read_committed) do
  a = Snack.find(1)
  a.calories -= 50
  sleep(10)
  a.save
  // Final calories after update: 150
end

// Transaction B: Using read-committed isolation level
Snack.transaction(isolation: :read_committed) do
  b = Snack.find(1)
  b.calories -= 30
  b.save
  p b.calories // Displaying the updated calories after Transaction B
  // Updated calories after Transaction B: 170
end
```

As a result of the Serialization Anomaly, the final calories are 150, even though Transaction B intended to reduce them to 170. The changes made by Transaction B were lost or overwritten by the changes made by Transaction A, leading to an incorrect final value.

To prevent this, we can set the isolation level to "serializable" to ensure proper serialization of transactions and avoid lost updates. When using the "serializable" isolation level, Transaction A, which attempts to update the snack calories while Transaction B is also modifying the same data, will encounter a "SerializationFailure" error. This error occurs because the database system detects a conflict between the transactions that would violate the serializability of their execution.

However, it's important to note that while setting the isolation level to "serializable" can prevent lost updates and maintain data consistency, it may also introduce efficiency issues due to increased locking and potentially longer transaction durations. Therefore, choosing the appropriate isolation level requires careful consideration of both data integrity and performance implications.

2 Database Lock

Database locks are a mechanism to control access to shared resources and prevent multiple processes from concurrently modifying the same data.

2.1 Race Condition

2.1.1 Pessimistic Locking

One way to address race conditions is through pessimistic locking. With this method, database locks are implemented to ensure that only one process is allowed to modify a shared resource at any given time. Consider an e-commerce application where a product's stock count needs safeguarding. By applying a lock, the system prevents race conditions when multiple users simultaneously attempt to purchase the same item.

- Advantages of Pessimistic Locking:

  Guaranteed Exclusivity: Pessimistic locking guarantees that only one process can access and modify a resource at a time, eliminating the risk of simultaneous modifications.

  Simple Handling: The locking mechanism itself handles concurrency control, making it straightforward to implement in applications.

- Disadvantages of Pessimistic Locking:

Reduced Concurrency: As only one process can access the resource at a time, this approach might lead to reduced concurrency and potentially impact system performance.

Potential Deadlocks: Improper use of pessimistic locking can lead to deadlocks, where multiple processes are stuck waiting for each other to release locks.

2.1.2 Optimistic Locking

An alternative strategy for addressing race conditions is optimistic locking. In optimistic locking, a process verifies whether the data it intends to modify has been altered by another process since its last read. This is often accomplished using a version number or timestamp associated with the data. If no changes are detected, the process can proceed with its modifications. In cases of conflict, the process can respond by either retrying the operation or notifying the user.

- Advantages of Optimistic Locking:

  Enhanced Concurrency: Optimistic locking promotes higher concurrency since processes are not blocked by locks, allowing multiple processes to read data simultaneously.

  Reduced Lock Contention: With no need for exclusive locks, the potential for lock contention and associated performance bottlenecks is reduced.

Rails provides comprehensive methods to implement both pessimistic and optimistic locking strategies:

- https://blog.kiprosh.com/how-to-prevent-race-condition-in-ruby-on-rails
- https://api.rubyonrails.org/classes/ActiveRecord/Locking/Pessimistic.html
- https://api.rubyonrails.org/classes/ActiveRecord/Locking/Optimistic.html
- https://ruby-china.org/topics/40991

2.2 Lost Update:

Database locks, particularly pessimistic locking, play a crucial role in preventing lost update issues. When multiple processes attempt to modify the same data concurrently, the use of locks ensures that only one process can update the record at a time. By enforcing this serialized access, the potential for lost updates is eliminated. Lost updates can occur when two or more processes modify the same data simultaneously, leading to one process's changes being overwritten by another process's changes. By implementing database locks, you ensure that modifications are applied sequentially, safeguarding data integrity and preventing inadvertent data loss.

2.3 Deadlock:

Deadlocks are situations where two or more processes are each waiting for the other to release a lock, effectively causing a standstill. This scenario can arise when processes acquire locks in a different order, leading to a circular dependency. Database locks, when not managed properly, can contribute to deadlock occurrences. To avoid deadlocks, it's essential to carefully design the order in which locks are acquired and released. Additionally, employing deadlock detection mechanisms can help identify and resolve these situations. If a deadlock is detected, the application can take corrective action, such as rolling back transactions and allowing them to retry. It's crucial to strike a balance between concurrency and deadlock prevention strategies to ensure efficient and reliable system operation.

3 Redis Distributed Lock:

In distributed systems where multiple application instances are involved, Redis distributed locks offer a crucial mechanism for orchestrating access to shared resources.

Summary: the point of using redis distributed lock is the selection of keys, for example if we are dealing product selling issues, and it will be much easier if we have individual product id, we can set the product id as the key to prevent one product being sold multiple times.

3.1 Distributed Concurrency Issues:

Redis distributed locks can effectively address various distributed concurrency issues. They ensure that only one instance at a time can access and modify a shared resource, mitigating race conditions and lost updates that might arise when multiple instances concurrently attempt to modify the same data.

```
127.0.0.1:6379> SETNX keyword1 999
(integer) 1
127.0.0.1:6379> SETNX keyword1 999
(integer) 0
127.0.0.1:6379> SETNX keyword1 888
(integer) 0
#1 if the key was set
#0 if the key was not set
```

Based on the above redis commands, we configure the Redlock gem with the URLs of Redis servers, enabling distributed lock management. We then attempt to acquire a lock for a specific "resource-key" with a timeout of 2000 milliseconds (2 seconds). If the lock is successfully acquired, we perform critical operations within the locked section, ensuring exclusive access. Once the operation is complete, we release the lock using redlock.unlock(lock-info).

```
require 'redlock'
redis_servers = [
  { url: 'redis://your-redis-server-1:6379/0' },
  { url: 'redis://your-redis-server-2:6379/0' },
  # ... add more server URLs as needed
]
redlock = Redlock::Client.new(redis_servers)

lock_info = redlock.lock("resource_key", 2000) # Lock resource for 2 seconds

if lock_info
  begin
    # Perform critical operation with the acquired lock
  ensure
    redlock.unlock(lock_info)
  end
end
```

3.2 Deadlock (in Distributed Systems):

Utilizing Redis distributed locks can serve as a powerful tool in preventing deadlocks within distributed systems. By meticulously managing the acquisition and release of locks across different nodes, the probability of encountering deadlocks is significantly minimized. Through proper lock management, potential circular dependencies causing deadlocks can be avoided.

To further enhance deadlock prevention and recovery, you can leverage Redis distributed lock expiration and Sidekiq scheduled jobs. By setting an appropriate expiration time for locks and periodically checking for and cleaning up expired locks, you can bolster your system's resilience against potential deadlocks.

4 Summary

By understanding and applying these three concurrency handling methods appropriately, developers can effectively address various concurrency issues in Ruby on Rails applications, ensuring data integrity and providing a smoother user experience.

In summary, this report navigated through key concepts in database transactions and concurrency control. It highlighted isolation levels' role in ensuring data consistency and explored scenarios such as dirty reads, non-repeatable reads, phantom reads, and serialization anomalies.

The report provided practical insights into pessimistic and optimistic locking, addressing race conditions while considering the trade-offs between concurrency and data consistency. Database locks' significance in preventing lost updates and managing deadlocks was underscored.

For distributed systems, Redis distributed locks emerged as a valuable tool in handling concurrency issues across multiple instances, preventing race conditions and deadlocks. The report emphasized the need for a balanced approach, considering both data integrity and performance implications when selecting concurrency control mechanisms.

In conclusion, this report equips readers with practical knowledge, aiding informed decision-making in designing resilient and efficient database systems.