How to prevent lost updates in long conversations

SEPTEMBER 22, 2014 / VLADMIHALCEA

(Last Updated On: January 29, 2018)

Introduction

All database statements are executed within the context of a physical transaction, even when we don't explicitly declare transaction boundaries (BEGIN/COMMIT/ROLLBACK). Data integrity is enforced by the ACID properties of database transactions.

Logical vs Physical transactions

A logical transaction is an application-level unit of work that may span over multiple physical (database) transactions. Holding the database connection open throughout several user requests, including user think time, is definitely an anti-pattern.

A database server can accommodate a limited number of physical connections, and often those are reused by using connection pooling. Holding limited resources for long periods of time hinders scalability. So database transactions must be short to ensure that both database locks and the pooled connections are released as soon as possible.

Web applications entail a read-modify-write conversational pattern. A web conversation consists of multiple user requests, all operations being logically connected to the same application-level transaction. A typical use case goes like this:

1. Alice requests a certain product for being displayed
2. The product is fetched from the database and returned to the browser
3. Alice requests a product modification
4. The product must be updated and saved to the database

All these operations should be encapsulated in a single unit-of-work. We, therefore, need an application-level transaction that's also ACID compliant, because other concurrent users might modify the same entities, long after shared locks had been released.

In my previous post I introduced the perils of lost updates. The database transaction ACID properties can only prevent this phenomenon within the boundaries of a single physical transaction. Pushing transaction boundaries into the application layer requires application-level ACID guarantees.

To prevent lost updates, we must have application-level repeatable reads along with a concurrency control mechanisms.

Long conversations

HTTP is a stateless protocol. Stateless applications are always easier to scale than stateful ones, but conversations can't be stateless.

Hibernate offers two strategies for implementing long conversations:

- Extended persistence context
- Detached objects

Extended persistence context

After the first database transaction ends the JDBC connection is closed (usually going back to the connection pool) and the Hibernate session becomes disconnected. A new user request will reattach the original Session. Only the last physical transaction must issue DML operations, as otherwise, the application-level transaction is not an atomic unit of work.

For disabling persistence in the course of the application-level transaction, we have the following options:

- We can disable automatic flushing, by switching the Session FlushMode to MANUAL. At the end of the last physical transaction, we need to explicitly call Session#flush() to propagate the entity state transitions.
- All but the last transaction are marked read-only. For read-only transactions Hibernate disables both dirty checking and the default automatic flushing.

  The read-only flag might propagate to the underlying JDBC Connection, so the driver might enable some database-level read-only optimizations.

  The last transaction must be writeable so that all changes are flushed and committed.

Using an extended persistence context is more convenient since entities remain attached across multiple user requests. The downside is the memory footprint. The persistence context might easily grow with every new fetched entity. Hibernate default dirty checking mechanism uses a deep-comparison strategy, comparing all properties of all managed entities. The larger the persistence context, the slower the dirty checking mechanism will get.

This can be mitigated by evicting entities that don't need to be propagated to the last physical transaction.

Java Enterprise Edition offers a very convenient programming model through the use of @Stateful Session Beans along with an EXTENDED PersistenceContext.

All extended persistence context examples set the default transaction propagation to NOT_SUPPORTED which makes it uncertain if the queries are enrolled in the context of a local transaction or each query is executed in a separate database transaction.

Detached objects

Another option is to bind the persistence context to the life-cycle of the intermediate physical transaction. Upon persistence context closing all entities become detached. For a detached entity to become managed, we have two options:

- The entity can be reattached using Hibernate specific Session.update() method. If there's an already attached entity (same entity class and with the same identifier) Hibernate throws an exception, because a Session can have at most one reference of any given entity.

  There is no such equivalent in Java Persistence API.
- Detached entities can also be merged with their persistent object equivalent. If there's no currently loaded persistence object, Hibernate will load one from the database. The detached entity will not become managed.

  By now you should know that this pattern smells like trouble:

  What if the loaded data doesn't match what we have previously loaded?
  What if the entity has changed since we first loaded it?

  Overwriting new data with an older snapshot leads to lost updates. So the concurrency control mechanism is not an option when dealing with long conversations.

Both Hibernate and JPA offer entity merging.

Detached entities storage

The detached entities must be available throughout the lifetime of a given long conversation. For this, we need a stateful context to make sure all conversation requests find the same detached entities. Therefore we can make use of:

- Stateful Session Beans

  Stateful session beans is one of the greatest feature offered by Java Enterprise Edition. It hides all the complexity of saving/loading state between different user requests. Being a built-in feature, it automatically benefits from cluster replication, so the developer can concentrate on business logic instead.

  Seam is a Java EE application framework that has built-in support for web conversations.
- HttpSession

  We can save the detached objects in the HttpSession. Most web/application servers offer session replication so this option can be used by non-JEE technologies, like Spring framework. Once the conversation is over, we should always discard all associated state, to make sure we don't bloat the Session with unnecessary storage.

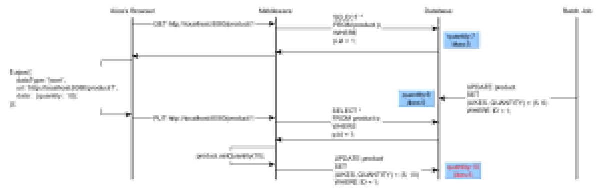
  You need to be careful to synchronize all HttpSession access (getAttribute/setAttribute), because for a very strange reason, this web storage is not thread-safe.

  Spring Web Flow is a Spring MVC companion that supports HttpSession web conversations.
- Hazelcast

  Hazelcast is an in-memory clustered cache, so it's a viable solution for the long conversation storage. We should always set an expiration policy because, in a web application, conversations might be started and abandoned. Expiration acts as the Http session invalidation.

The stateless conversation anti-pattern

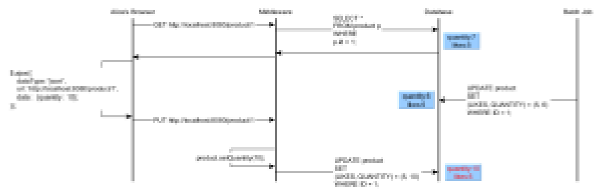
Like with database transactions, we need repeatable reads as otherwise we might load an already modified record without realizing it so:



1. Alice request a product to be displayed
2. The product is fetched from the database and returned to the browser
3. Alice request a product modification
4. Because Alice hasn’t kept a copy of the previously displayed object, she has to reload it once again
5. The product is updated and saved to the database
6. The batch job update has been lost and Alice will never realize it

The stateful version-less conversation anti-pattern

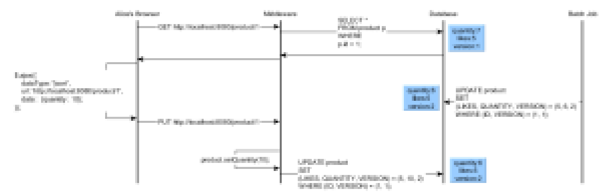
Preserving conversation state is a must if we want to ensure both isolation and consistency, but we can still run into lost updates situations:



Even if we have application-level repeatable reads others can still modify the same entities. Within the context of a single database transaction, row-level locks can block concurrent modifications but this is not feasible for logical transactions. The only option is to allow others modify any rows while preventing persisting stale data.

Optimistic locking to the rescue

Optimistic locking is a generic-purpose concurrency control technique, and it works for both physical and application-level transactions. Using JPA is only a matter of adding a @Version field to our domain models:



Video

Because this is a very interesting topic, I decided to record a video as well. Enjoy watching it!

Conclusion

Pushing database transaction boundaries into the application layer requires an application-level concurrency control. To ensure application-level repeatable reads we need to preserve state across multiple user requests, but in the absence of database locking, we need to rely on an application-level concurrency control.

Optimistic locking works for both database and application-level transactions, and it doesn't make use of any additional database locking. Optimistic locking can prevent lost updates and that's why I always recommend all entities be annotated with the @Version attribute.

Subscribe to our Newsletter

* indicates required

Email Address *

10 000 readers have found this blog worth following!

If you **subscribe** to my newsletter, you'll get:

- A **free sample** of my Video Course about running Integration tests at warp-speed using Docker and tmpfs
- **3 chapters** from my book, **High-Performance Java Persistence**,
- a **10% discount** coupon for my book.

Get the most out of your persistence layer!

Subscribe

Related

A beginner's guide to transaction isolation levels in enterprise Java
December 23, 2014
In "Hibernate"

The Open Session In View Anti-Pattern
May 30, 2016
In "Hibernate"

How does Hibernate NONSTRICT_READ_WRITE CacheConcurrencyStrategy work
May 18, 2015
In "Hibernate"

Categories: Hibernate, SQL

Tags: conversation, extended persistence context, hazelcast, hibernate, http session, lost updates, optimistic locking, Training, transactions, Tutorial

← A beginner’s guide to database locking and the lost update phenomena

The fastest way of drawing UML class diagrams →

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Sign me up for the newsletter!

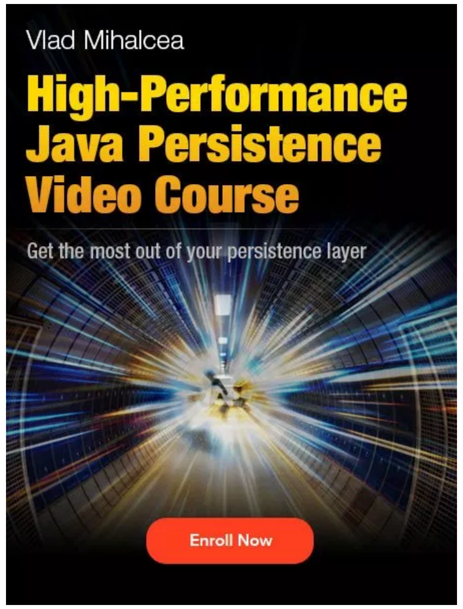
Post Comment

Notify me of follow-up comments by email.

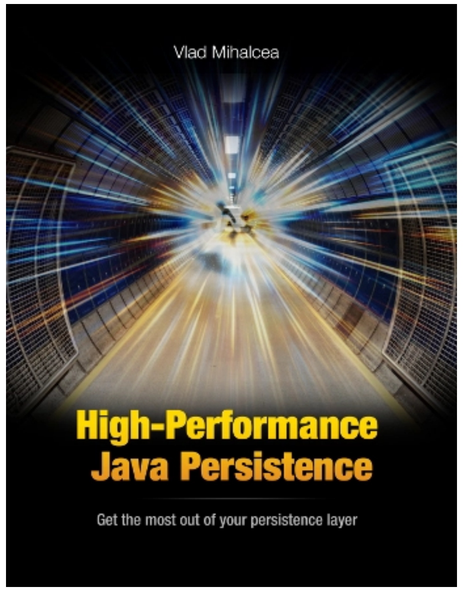
JDK.IO WORKSHOP – COPENHAGEN

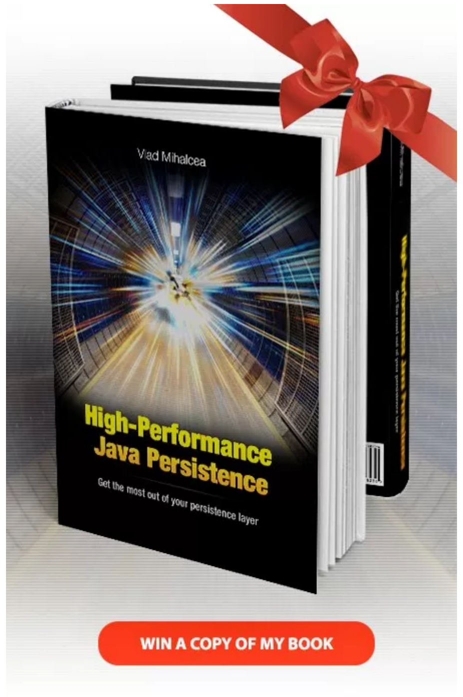
JDK.IO
0xCAFEBABE
CONFERENCE
2018
WORKSHOP
High-Performance Java Persistence

VIDEO COURSE


Vlad Mihalcea
High-Performance Java Persistence Video Course
Get the most out of your persistence layer
Enroll Now

HIGH-PERFORMANCE JAVA PERSISTENCE


Vlad Mihalcea
High-Performance Java Persistence
Get the most out of your persistence layer

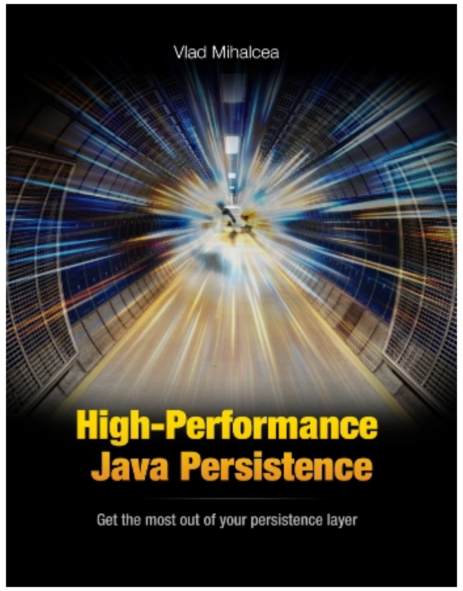

Vlad Mihalcea
High-Performance Java Persistence
Get the most out of your persistence layer
WIN A COPY OF MY BOOK

HIBERNATE PERFORMANCE TUNING TIPS

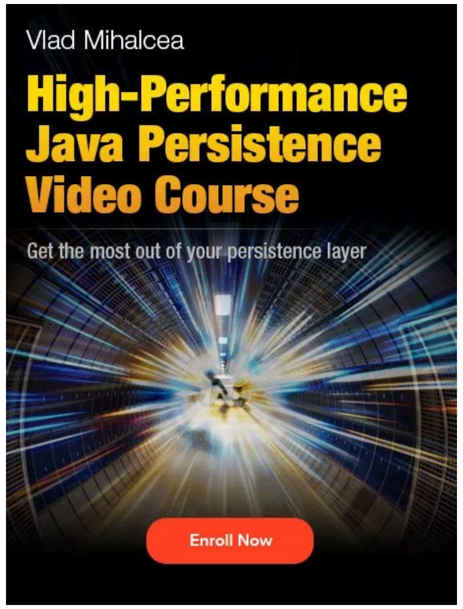
HYPERSISTENCE

WIN A COPY OF MY BOOK!

HIGH-PERFORMANCE JAVA PERSISTENCE


Vlad Mihalcea
High-Performance Java Persistence
Get the most out of your persistence layer

VIDEO COURSE


Vlad Mihalcea
High-Performance Java Persistence Video Course
Get the most out of your persistence layer
Enroll Now

JDK.IO WORKSHOP – COPENHAGEN

JDK.IO
0xCAFEBABE
CONFERENCE
2018
WORKSHOP
High-Performance Java Persistence