A beginner's guide to Java Persistence locking

JANUARY 12, 2015 / VLADMIHALCEA

(Last Updated On: January 29, 2018)

Implicit locking

In concurrency theory, locking is used for protecting mutable shared data against hazardous data integrity anomalies. Because lock management is a very complex problem, most applications rely on their data provider *implicit locking* techniques.

Delegating the whole locking responsibility to the database system can both simplify application development and prevent concurrency issues, such as deadlocking. Deadlocks can still occur, but the database can detect and take safety measures (arbitrarily releasing one of the two competing locks).

Physical locks

Most database systems use *shared (read)* and *exclusive (write)* locks, attributed to specific locking elements (rows, tables). While physical locking is demanded by the SQL standard, the pessimistic approach might hinder scalability.

Modern databases have implemented lightweight locking techniques, such as MVCC.

The implicit database locking is hidden behind the transaction isolation level configuration. Each isolation level comes with a predefined locking scheme, aimed at preventing a certain set of data integrity anomalies.

READ COMMITTED uses query-level shared locks and exclusive locks for the current transaction modified data. REPEATABLE READ and SERIALIZABLE use transaction-level shared locks when reading and exclusive locks when writing.

Logical locks

If database locking is sufficient for batch processing systems, a multi-request web flow spans over several database transactions. For long conversations, a logical (optimistic) locking mechanism is much more appropriate.

Paired with a conversation-level repeatable read storage, optimistic locking can ensure data integrity without trading scalability.

JPA supports both optimistic locking and persistence context repeatable reads, making it ideal for implementing logical transactions.

Explicit locking

While implicit locking is probably the best choice for most applications concurrency control requirements, there might be times when you want a finer-grained locking strategy.

Most database systems support query-time exclusive locking directives, such as SELECT FOR UPDATE or SELECT FOR SHARE. We can, therefore, use lower level default isolation levels (READ COMMITTED), while requesting share or exclusive locks for specific transaction scenarios.

Most optimistic locking implementations verify modified data only, but JPA allows explicit optimistic locking as well.

JPA locking

As a database abstraction layer, JPA can benefit from the implicit locking mechanisms offered by the underlying RDBMS. For logical locking, JPA offers an optional automated entity version control mechanism as well.

JPA supports explicit locking for the following operations:

- finding an entity
- locking an existing persistence context entity
- refreshing an entity
- querying through JPQL, Criteria or native queries

Explicit lock types

The LockModeType contains the following optimistic and pessimistic locking modes:

Lock Mode Type	Description
NONE	In the absence of explicit locking, the application will use implicit locking (optimistic or pessimistic)

OPTIMISTIC	Always issues a version check upon transaction commit, therefore ensuring optimistic locking repeatable reads.
READ	Same as OPTIMISTIC.
OPTIMISTIC_FORCE_INCREMENT	Always increases the entity version (even when the entity doesn't change) and issues a version check upon transaction commit, therefore ensuring optimistic locking repeatable reads.
WRITE	Same as OPTIMISTIC_FORCE_INCREMENT.
PESSIMISTIC_READ	A *shared lock* is acquired to prevent any other transaction from acquiring a PESSIMISTIC_WRITE lock.
PESSIMISTIC_WRITE	An *exclusive lock* is acquired to prevent any other transaction from acquiring a PESSIMISTIC_READ or a PESSIMISTIC_WRITE lock.
PESSIMISTIC_FORCE_INCREMENT	A database lock is acquired to prevent any other transaction from acquiring a PESSIMISTIC_READ or a PESSIMISTIC_WRITE lock and the entity version is incremented upon transaction commit.

Lock scope and timeouts

JPA 2.0 defined the *javax.persistence.lock.scope* property, taking one of the following values:

- NORMAL

Because object graphs can span to multiple tables, an explicit locking request might propagate to more than one table (e.g. joined inheritance, secondary tables).

Because the entire entity associated row(s) are locked, *many-to-one* and *one-to-one* foreign keys will be locked as well but without locking the other side parent associations. This scope doesn't propagate to children collections.

- EXTENDED

  The explicit lock is propagated to element collections and junction tables, but it doesn't lock the actual children entities. The lock is only useful for protecting against removing existing children, while permitting *phantom reads* or changes to the actual children entity states.

JPA 2.0 also introduced the *javax.persistence.lock.timeout* property, allowing us to configure the amount of time (milliseconds) a lock request will wait before throwing a PessimisticLockException.

Hibernate locking

Hibernate supports all JPA locking modes and some additional specific locking options. As with JPA, explicit locking can be configured for the following operations:

- locking an entity using various LockOptions settings.
- getting an entity
- loading an entity
- refreshing an entity
- creating an entity or a native Query
- creating a Criteria query

The LockModeConverter takes care of mapping JPA and Hibernate lock modes as follows:

Hibernate LockMode	JPA LockModeType
NONE	NONE
OPTIMISTIC READ	OPTIMISTIC
OPTIMISTIC_FORCE_INCREMENT WRITE	OPTIMISTIC_FORCE_INCREMENT
PESSIMISTIC_READ	PESSIMISTIC_READ
PESSIMISTIC_WRITE ~~UPGRADE~~ UPGRADE_NOWAIT UPGRADE_SKIPLOCKED	PESSIMISTIC_WRITE
PESSIMISTIC_FORCE_INCREMENT ~~FORCE~~	PESSIMISTIC_FORCE_INCREMENT

The UPGRADE and FORCE lock modes are deprecated in favor of PESSIMISTIC_WRITE.

UPGRADE_NOWAIT and UPGRADE_SKIPLOCKED use an Oracle-style select for update nowait or select for update skip locked syntax respectively.

Lock scope and timeouts

Hibernate also defines scope and timeout locking options:

- scope

  The lock scope allows explicit locking cascade to owned associations.
- timeout

  A timeout interval may prevent a locking request from waiting indefinitely.

Related

How does database pessimistic locking interact with INSERT, UPDATE, and DELETE SQL statements
February 7, 2017
In "Database"

How do LockModeType.PESSIMISTIC_READ and LockModeType.PESSIMISTIC_WRITE work in JPA and Hibernate
February 24, 2015
In "Hibernate"

How to prevent lost updates in long conversations
September 22, 2014
In "Hibernate"

Categories: Hibernate, Java

Tags: explicit locking, hibernate, implicit locking, isolation levels, optimistic locking, pesimistic locking, Training, transactions, Tutorial

← Why you should pay developers to learn

How to get a 10,000 points StackOverflow reputation →

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

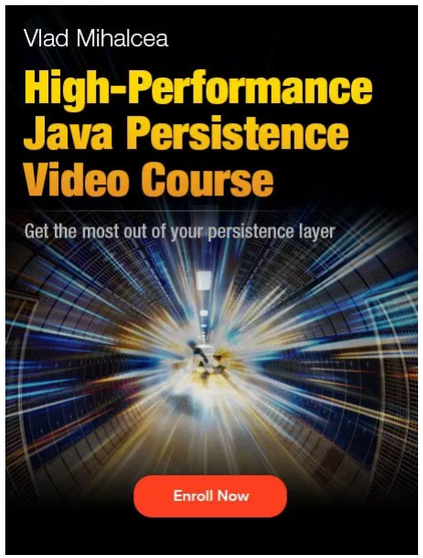
Notify me of follow-up comments by email.

JDK.IO
0xCAFEBABE
CONFERENCE
2018

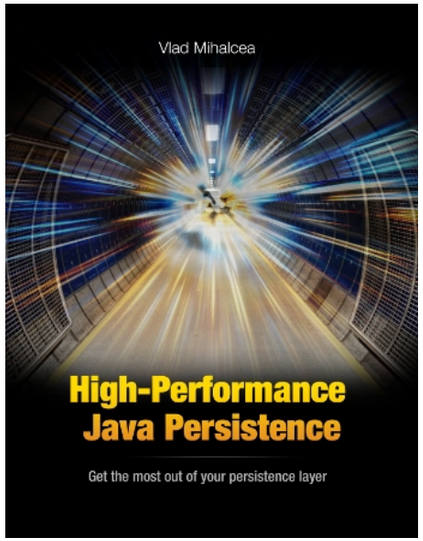
WORKSHOP

High-Performance Java Persistence

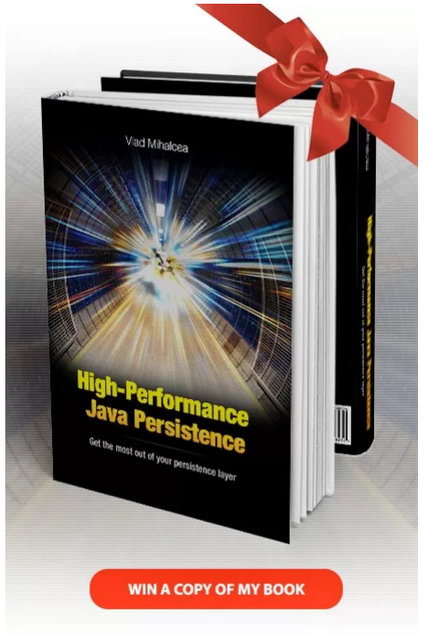
VIDEO COURSE



HIGH-PERFORMANCE JAVA PERSISTENCE



WIN A COPY OF MY BOOK!

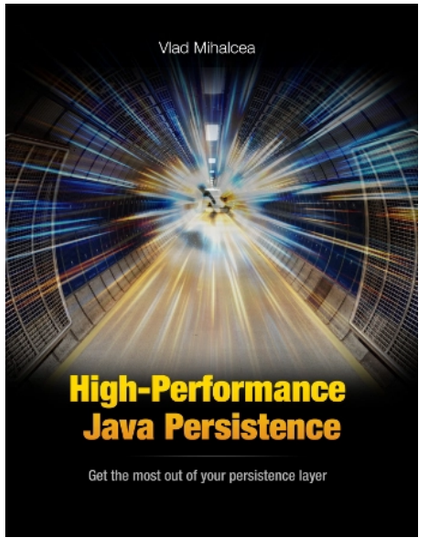


HIBERNATE PERFORMANCE TUNING TIPS

HYPERSISTENCE

WIN A COPY OF MY BOOK!

HIGH-PERFORMANCE JAVA PERSISTENCE



VIDEO COURSE

JDK.IO WORKSHOP – COPENHAGEN

JDKIO
0xCAFEBABE
CONFERENCE
2018

WORKSHOP

High-Performance Java Persistence

Search ...

RSS - Posts

RSS - Comments

ABOUT

About

Privacy Policy

Terms of Service