# Hypersistence Optimizer – can the optimization process of the data access layer be simple?

Behind the most of applications and websites stands a data source (e.g. database or data files) that stores all kinds of information needed for a proper functioning of the system. Therefore, a communication between an application and a data source is the key element of a system. Responsible for this communication is the data access layer. In the Java world, a popular way to implement this layer is object-relational mapping, which is based on reflecting an object-oriented system architecture in a relational database. The most popular framework used in object-relational mapping is Hibernate, which is an implementation of the JPA standard. Hibernate is a powerful tool, and therefore it is not easy to use at all. During configuration and implementation, many mistakes can be made that could significantly affect the performance of the built system. Searching for errors and subsequent optimization can be very time consuming. Hypersistence Optimizer comes to the rescue!

*„Imagine having a tool that can **automatically** detect if you are using JPA and Hibernate properly.”* – Hypersistence Optimizer

## Hypersistence Optimizer - what is that?

Hypersistence Optimizer is a tool for an automatic analysis of the data access layer, which supports a wide range of Hibernate versions (v.3.3 - 5.4). The analysis process covers Hibernate configuration, entity mapping, database queries and Persistence Context. Based on the analysis, we receive a detailed report with problems found in the application. Such a report contains information

about the significance of the found problem, its location, a brief description of the problem and a link to a more detailed explanation along with tips and examples. The tool was created by the Hypersistence company and is constantly being developed. Currently, the tool is able to detect over 50 different types of errors. Hypersistence Oprimizer comes in two versions: full and trial. The trial is free, but very limited compared to the full version. The trial version catches only the type and number of bugs, without indicating where these bugs occur and how to deal with them. Before buying the full version, it is a good idea to check with the trial version if there are any errors in the application being created.

## First steps

The installation process consists of a few simple steps. Before installation, make sure that Maven is installed on the computer, which is needed to install the tool. After downloading and unpacking the archive, just run the "maven-install" batch file, which is located in the extracted folder. Hypersistence Optimizer will be added to the Maven repository. The next step is to add dependencies to the pom.xml file in the project:

```
1.  <dependency>
2.          <groupId>io.hypersistence</groupId>
3.          <artifactId>hypersistence-optimizer</artifactId>
4.          <version>2.1.1</version>
5.  </dependency>
```

After this step, the tool is ready to use. The analysis is turned on from the unit testing level. Create a test and initialize Optimizer in it. It may look like this:

```
1.    class HypersistenceTest {
2.
3.              @PersistenceUnit
4.              private EntityManagerFactory entityManagerFactory;
5.
6.              @PostConstruct
7.              public void init() {
8.                      new HypersistenceOptimizer(new JpaConfig(entityManagerFactory));
9.              }
10.
11.             @Test
12.             void someTest() {
13.             }
14. }
```

As we can see, using the Hypersistence Optimizer is very simple. During initialization, we have to specify only the EntityManager, from which the tool retrieves a metadata needed for analysis. This is

obviously the most basic configuration. The configuration options are more extensive. You can, among other things, define what area of the data access layer should be analyzed (configuration, mapping or queries), what events should be caught or what should happen with the events that has been detected. Each encountered problem during an analysis causes an event related to a currently checked rule. Therefore, all detected problems are returned in a form of events that can be assigned to a specific rule and description. By default, the tool analyzes all areas and catches all encountered events. The authors provide a detailed description of the configuration with which you can adapt the tool to your needs.

The last step is to wrap the default Hibernate SessionFactory configuration with the decorator provided by the creators. This step is necessary if you want your queries to be analyzed in real time. When using Hibernate 5, all you have to do is to copy the file located in ..\configs\META-INF\services to ..\src\test\resources of the current project. The tool configured in this way will work in a test environment without causing any problems in the production environment. For older versions of Hibernate, you have to decorate the SessionFactory manually. A detailed description can be found in the supplied configuration guide.

## Optimizer in action

When you run a unit test with the configured Hypersistence Optimizer, the report appears in the console. The tool was tested on a small application for booking company cars. The resulting report looks like this:

```
CRITICAL - IdentityGeneratorEvent - The [id] identifier attribute in the
[pl.fis.carrs.domain.model.Vehicle] entity uses the [IdentityGenerator] strategy, which prevents
Hibernate from enabling JDBC batch inserts. Consider using the SEQUENCE identifier strategy
instead. For more info about this event, check out this User Guide link -
https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#IdentityGeneratorEvent

MAJOR - EnumTypeStringEvent - The [fuelType] enum attribute in the
[pl.fis.carrs.domain.model.Vehicle] entity uses the EnumType.STRING strategy, which has a bigger
memory footprint than EnumType.ORDINAL. For more info about this event, check out this User
Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-
guide/#EnumTypeStringEvent

CRITICAL - BidirectionalSynchronizationEvent - The [reservations] bidirectional association in the
[pl.fis.carrs.domain.model.Vehicle] entity requires both ends to be synchronized. Consider adding
the [addReservation(pl.fis.carrs.domain.model.Reservation reservation)] and
[removeReservation(pl.fis.carrs.domain.model.Reservation reservation)] synchronization methods.
For more info about this event, check out this User Guide link -
https://vladmihalcea.com/hypersistence-optimizer/docs/user-
guide/#BidirectionalSynchronizationEvent

CRITICAL - EagerFetchingEvent - The [vehicle] attribute in the
[pl.fis.carrs.domain.model.Reservation] entity uses eager fetching. Consider using a lazy fetching,
which not only is more efficient, but it is way more flexible when it comes to fetching data. For
```

more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#EagerFetchingEvent

MAJOR - SkipAutoCommitCheckEvent - You should set the [hibernate.connection.provider_disables_autocommit] configuration property to [true] while also making sure that the underlying DataSource is configured to disable the auto-commit flag, whenever a new Connection is being acquired. For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#SkipAutoCommitCheckEvent

CRITICAL - JdbcBatchSizeEvent - If you set the [hibernate.jdbc.batch_size] configuration property to a value greater than 1 (usually between 5 and 30), Hibernate can then execute SQL statements in batches, therefore reducing the number of database network roundtrips. For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#JdbcBatchSizeEvent

CRITICAL - QueryPaginationCollectionFetchingEvent - You should set the [hibernate.query.fail_on_pagination_over_collection_fetch] configuration property to the value of [true], as Hibernate can then prevent in-memory pagination when it joins fetching a child entity collection. For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#QueryPaginationCollectionFetchingEvent

MAJOR - QueryInClauseParameterPaddingEvent - You should set the [hibernate.query.in_clause_parameter_padding] configuration property to the value of [true], as Hibernate entity queries can then make better use of statement caching and fewer entity queries will have to be compiled while varying the number of parameters passed to the in query clause. For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#QueryInClauseParameterPaddingEvent

8 issues were found: 0 BLOCKER, 5 CRITICAL, 3 MAJOR, 0 MINOR

Based on the information contained in the report, we can begin to optimize the data access layer. For example, consider an EagerFetchingEvent error that has been classified as a critical error. The relationship between a reservation and a vehicle has been defined as many-to-one but no strategy has been given for loading the vehicle collection.

```
1.  @ManyToOne
2.  private Vehicle vehicle;
```

When using many-to-one and one-to-one relationships, the default strategy for loading a collection is FetchType.EAGER. Loading the EAGER collection means that all data is downloaded when the parent is downloaded, even if we don't need it at all. In this case, it is recommended to use the FetchType.LAZY

strategy, which downloads the collection when it is needed. The code after optimization could look like this:

```
1. @ManyToOne (fetch = FetchType.LAZY)
2. @JoinColumn (name = "vehicle_id")
3. private Vehicle vehicle;
```

Hypersistence Optimizer catches potential threats in the data access layer that can (but don't have to!) cause performance problems. However, these are only tips that can be taken into account to eliminate existing or potential problems. Concrete solutions depend on the complexity of the system being built, which is why each problem found should be considered individually. An example would be the EnumTypeStringEvent. In this case, it is recommended that the enumeration types should be saved in the database in a form of EnumType.ORDINAL. In the application being tested, enums are saved as EnumType.STRING. The transition from EnumType.STRING to EnumType.ORDINAL will save some memory, but the readability of the code and the readability of information in the database will decrease. Here the choice is up to the programmer whether to keep the code clean or save some memory.

The second part of the report contains caught problems related to queries (appears only after configuring Hibernate SessionFactory):

```
CRITICAL - PaginationWithoutOrderByEvent - The [select v from VEHICLES v ] query uses pagination without an ORDER BY clause. Therefore, the result is not deterministic since SQL does not guarantee any particular ordering unless an ORDER BY clause is being used. For more info about this event, check out this User Guide link – https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#PaginationWithoutOrderByEvent

CRITICAL - PassDistinctThroughEvent - The [SELECT DISTINCT v FROM VEHICLES v left join v.reservations res where not exists… ] query uses DISTINCT to deduplicate returned entities. However, without setting the [hibernate.query.passDistinctThrough] JPA query hint to [false], the underlying SQL statement will also contain DISTINCT, which will incur extra sorting and duplication removal execution stages. For more info about this event, check out this User Guide link – https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#PassDistinctThroughEvent
```

In the current version of the Optimizer, only 2 types of events are currently caught when it comes to query analysis: PaginationWithoutOrderByEvent and PassDistinctThroughEvent. In the example we can see that even in such a simple query "select v from VEHICLES v" we can make a mistake. If you restrict the results with setMaxResults, you have to add the ORDER BY clause to your query.

## Summary

Huge knowledge and commitment of the author - Vlad Mihalcea as a Developer Advocate for the Hibernate project, resulted in the creation of a great tool for automatic analysis of the data access layer, which is Hypersistence Optimizer. It saves developers a lot of time and gray hair when looking for the causes of poor application performance. In addition, detailed problem descriptions with examples can serve as a knowledge base for any developer.

## Author: Denis Wiesner

Engineer, Java programmer with several years of professional experience in backend and frontend. Interests: numerical methods, parallel programming, new technologies.

#java #hibernate #jpa #orm #hypersistenceOptimizer #optimization