**Introduction**

A high-performance data access layer requires a lot of knowledge about database internals, JDBC, JPA, Hibernate, and this post summarizes some of the most important techniques you can use to optimize your enterprise application.

**1. SQL statement logging**

If you’re using a framework that generates statements on your behalf, [you should always validate each statement effectiveness and efficiency](https://vladmihalcea.com/2016/05/03/the-best-way-of-logging-jdbc-statements/). A [testing-time assertion mechanism](https://vladmihalcea.com/2014/02/01/taming-jpa-with-the-sql-statement-count-validator/) is even better because you can catch [N+1 query problems](https://vladmihalcea.com/2014/02/01/how-to-detect-the-n-plus-one-query-problem-during-testing/) even before you commit your code.

**2. Connection management**

Database connections are expensive, therefore you should always use a [connection pooling](https://vladmihalcea.com/2014/04/17/the-anatomy-of-connection-pooling/) mechanism.

Because the number of connections is given by the capabilities of the underlying database cluster, you need to release connections as fast as possible.

In performance tuning, you always have to measure, and setting the right pool size is no different. A tool like [FlexyPool](https://vladmihalcea.com/2014/04/30/professional-connection-pool-sizing/) can help you find the right size even after you deployed your application into production.

**3. JDBC batching**

JDBC batching allows us to send multiple SQL statements in a single database roundtrip. [The performance gain is significant](https://leanpub.com/high-performance-java-persistence/read#jdbc-batch-updates) both on the Driver and the database side. PreparedStatements are very good candidates for batching, and some database systems (e.g. Oracle) support batching only for prepared statements only.

Since JDBC defines a distinct API for batching (e.g. [PreparedStatement.addBatch](https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html#addBatch--)and [PreparedStatement.executeBatch](https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html#executeBatch--)), if you’re generating statements manually, then you should know right from the start whether you should be using batching or not. With Hibernate, you can [switch to batching with a single configuration](https://vladmihalcea.com/2015/03/18/how-to-batch-insert-and-update-statements-with-hibernate/).

Hibernate 5.2 offers [Session-level batching](https://vladmihalcea.com/2016/09/27/how-to-customize-the-jdbc-batch-size-for-each-persistence-context-with-hibernate/), so it’s even more flexibile in this regard.

**4. Statement caching**

Statement caching is one of the least-known performance optimization that you can easily take advantage of. Depending on the underlying JDBC Driver, you can cache PreparedStatements both on the client-side (the Driver) or databases-side (either the syntax tree or even the execution plan).

**5. Hibernate identifiers**

When using Hibernate, the IDENTITY generator is not a good choice since it disables JDBC batching.

[TABLE generator is even worse]/2017/01/04/why-you-should-never-use-the-table-identifier-generator-with-jpa-and-hibernate/) since it uses a separate transaction for fetching a new identifier, which can put pressure on the underlying transaction log, as well as the connection pool since a separate connection is required every time we need a new identifier.

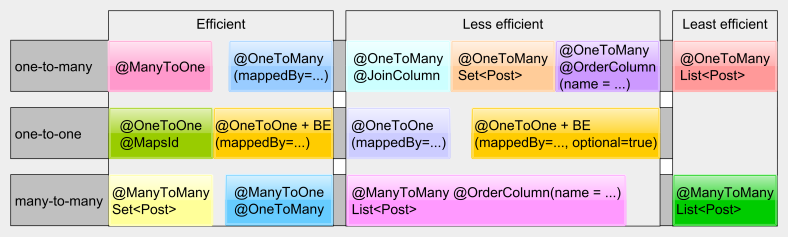
SEQUENCE is the right choice, and even SQL Server supports since version 2012. For SEQUENCE identifiers, Hibernate has long been offering optimizers like [pooled or pooled-lo](https://vladmihalcea.com/2014/07/21/hibernate-hidden-gem-the-pooled-lo-optimizer/) which can reduce the number of database roundtrips required for fetching a new entity identifier value.

**6. Choosing the right column types**

You should always use the right column types on the database side. The more compact the column type is, the more entries can be accommodated in the database working set, and indexes will better fit into memory. For this purpose, you should take advantage of database-specific types (e.g. inet for IPv4 addresses in PostgreSQL), especially since Hibernate is very flexible when it comes to [implementing a new custom Type](https://vladmihalcea.com/2016/06/20/how-to-map-json-objects-using-generic-hibernate-types/).

**7. Relationships**

Hibernate comes with many relationship mapping types, but not all of them are equal in terms of efficiency.

[](https://vladmihalcea.files.wordpress.com/2016/06/relationships.png)

[Unidirectional collections](https://vladmihalcea.com/2015/05/04/how-to-optimize-unidirectional-collections-with-jpa-and-hibernate/) and @ManyToMany List(s) should be avoided. If you really need to use entity collections, then bidirectional @OneToMany associations are preferred. For the @ManyToMany relationship, use Set(s) since [they are more efficient in this case](https://vladmihalcea.com/2017/05/10/the-best-way-to-use-the-manytomany-annotation-with-jpa-and-hibernate/) or simply map the linked many-to-many table as well and turn the @ManyToMany relationship into two bidirectional @OneToMany associations.

However, unlike queries, collections are less flexible since they cannot be easily paginated, meaning that we cannot use them when the number of child associations is rather high. For this reason, you should always question if a collection is really necessary. An entity query might be a better alternative in many situations.

**8. Inheritance**

When it comes to inheritance, the impedance mismatch between object-oriented languages and relational databases becomes even more apparent. JPA offers SINGLE\_TABLE, JOINED, and TABLE\_PER\_CLASS to deal with inheritance mapping, and each of these strategies has pluses and minuses.

SINGLE\_TABLE performs the best in terms of SQL statements, but we lose on the data integrity side since we cannot use NOT NULL constraints.

JOINED addresses the data integrity limitation while offering more complex statements. As long as you don’t use polymorphic queries or @OneToManyassociations against base types, this strategy is fine. Its true power comes from polymorphic @ManyToOne associations backed by a Strategy pattern on the data access layer side.

TABLE\_PER\_CLASS should be avoided since it does not render efficient SQL statements.

**9. Persistence Context size**

When using JPA and Hibernate, you should always mind the Persistence Context size. For this reason, you should never bloat it with tons of managed entities. By restricting the number of managed entities, we gain better memory management, and the default [dirty checking mechanism](https://vladmihalcea.com/2014/08/21/the-anatomy-of-hibernate-dirty-checking/) is going to be more efficient as well.

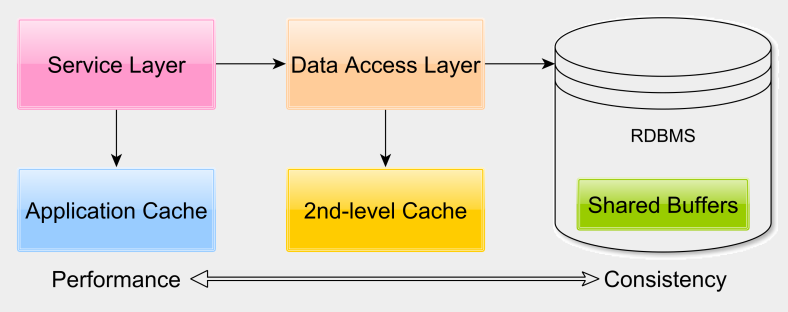
**10. Fetching only what’s necessary**

Fetching too much data is probably the number one cause for data access layer performance issues. One issue is that entity queries are used exclusively, even for read-only projections.

[DTO projections](https://vladmihalcea.com/2017/08/29/the-best-way-to-map-a-projection-query-to-a-dto-with-jpa-and-hibernate/) are [better suited for fetching custom views](https://vladmihalcea.com/2016/09/13/the-best-way-to-handle-the-lazyinitializationexception/), while entities should only be fetched when the business flow requires to modify them.

[EAGER fetching is the worst](https://vladmihalcea.com/2014/12/15/eager-fetching-is-a-code-smell/), and you should avoid anti-patterns such as [Open-Session in View](https://vladmihalcea.com/2016/05/30/the-open-session-in-view-anti-pattern/).

**11. Caching**

[](https://vladmihalcea.files.wordpress.com/2016/06/cachelayers.png)

Relational database systems use many [in-memory buffer structures to avoid disk access](https://vladmihalcea.com/2017/02/14/how-does-a-relational-database-work/). [Database caching is very often overlooked](https://vladmihalcea.com/2015/04/16/things-to-consider-before-jumping-to-enterprise-caching/). We can lower response time significantly by properly tuning the database engine so that the working set resides in memory and is not fetched from disk all the time.

Application-level caching is not optional for many enterprise application. Application-level caching can reduce response time while offering a read-only secondary store for when the database is down for maintenance or because of some serious system failure.

The second-level cache is very useful for reducing read-write transaction response time, especially in Master-Slave replication architectures. Depending on application requirements, Hibernate allows you to choose between [READ\_ONLY](https://vladmihalcea.com/2015/04/27/how-does-hibernate-read_only-cacheconcurrencystrategy-work/), [NONSTRICT\_READ\_WRITE](https://vladmihalcea.com/2015/05/18/how-does-hibernate-nonstrict_read_write-cacheconcurrencystrategy-work/), [READ\_WRITE](https://vladmihalcea.com/2015/05/25/how-does-hibernate-read_write-cacheconcurrencystrategy-work/), and [TRANSACTIONAL](https://vladmihalcea.com/2015/06/01/how-does-hibernate-transactional-cacheconcurrencystrategy-work/).

**12. Concurrency control**

The choice of [transaction isolation level](https://vladmihalcea.com/2014/12/23/a-beginners-guide-to-transaction-isolation-levels-in-enterprise-java/) is of paramount importance when it comes to performance and data integrity. For multi-request web flows, to avoid [lost updates](https://vladmihalcea.com/2014/09/14/a-beginners-guide-to-database-locking-and-the-lost-update-phenomena/), you should use [optimistic locking with detached entities or an EXTENDEDPersistence Context](https://vladmihalcea.com/2014/09/22/preventing-lost-updates-in-long-conversations/).

To avoid optimistic locking false positives, you can use [versionless optimistic concurrency control](https://vladmihalcea.com/2014/12/08/the-downside-of-version-less-optimistic-locking/) or [split entities based write-based property sets](https://vladmihalcea.com/2014/11/10/an-entity-modeling-strategy-for-scaling-optimistic-locking/).

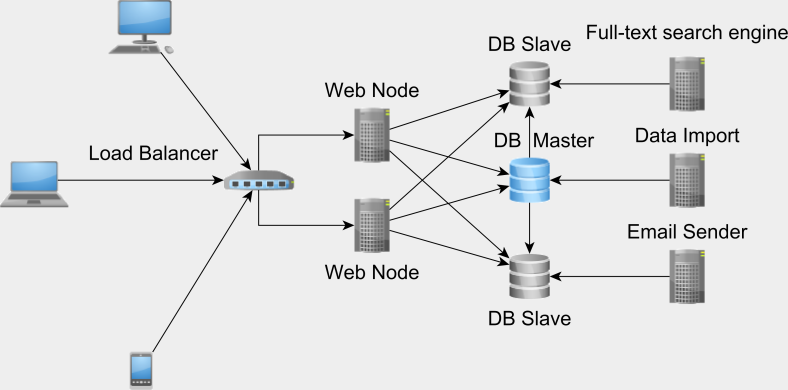
**13. Unleash database query capabilities**

Just because you use JPA or Hibernate, it does not mean that you should not use native queries. You should take advantage of [Window Functions](https://vladmihalcea.com/2014/05/12/time-to-break-free-from-the-sql-92-mindset/), CTE (Common Table Expressions), CONNECT BY, PIVOT.

These constructs allow you to avoid fetching too much data just to transform it later in the application layer. If you can let the database do the processing, you can fetch just the end result, therefore, saving lots of disk I/O and networking overhead. To avoid overloading the Master node, you can use database replication and have multiple Slave nodes available so that data-intensive tasks are executed on a Slave rather than on the Master.

**14. Scale up and scale out**

Relational databases do scale very well. If [Facebook](http://highscalability.com/blog/2010/11/4/facebook-at-13-million-queries-per-second-recommends-minimiz.html), [Twitter](http://highscalability.com/blog/2011/12/19/how-twitter-stores-250-million-tweets-a-day-using-mysql.html), [Pinterest](http://highscalability.com/blog/2013/4/15/scaling-pinterest-from-0-to-10s-of-billions-of-page-views-a.html) or [StackOverflow](https://nickcraver.com/blog/2013/11/22/what-it-takes-to-run-stack-overflow/) can scale their database system, there is good chance you can scale an enterprise application to its particular business requirements.

[](https://vladmihalcea.files.wordpress.com/2016/06/databaseintegrationpoint.png)

[Database replication and sharding](http://highscalability.com/blog/2016/5/11/performance-and-scaling-in-enterprise-systems.html) are very good ways to increase throughput, and you should totally take advantage of these battle-tested architectural patterns to scale your enterprise application.