Vlad Mihalcea's Blog

Teaching is my way of learning

14 High-Performance Java Persistence Tips

JUNE 28, 2016MAY 12, 2017 ⁄ VLADMIHALCEA

Follow @vlad_mihalcea 5,114 followers

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 (/2016/05/03/the-best-way-of-logging-jdbc-statements/). A testing-time assertion mechanism (/2014/02/01/taming-jpa-with-the-sql-statement-count-validator/) is even better because you can catch N+1 query problems (/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 (/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 (/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 (/2015/03/18/how-to-batch-insert-and-update-statements-with-hibernate/).

Hibernate 5.2 offers Session-level batching (/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.

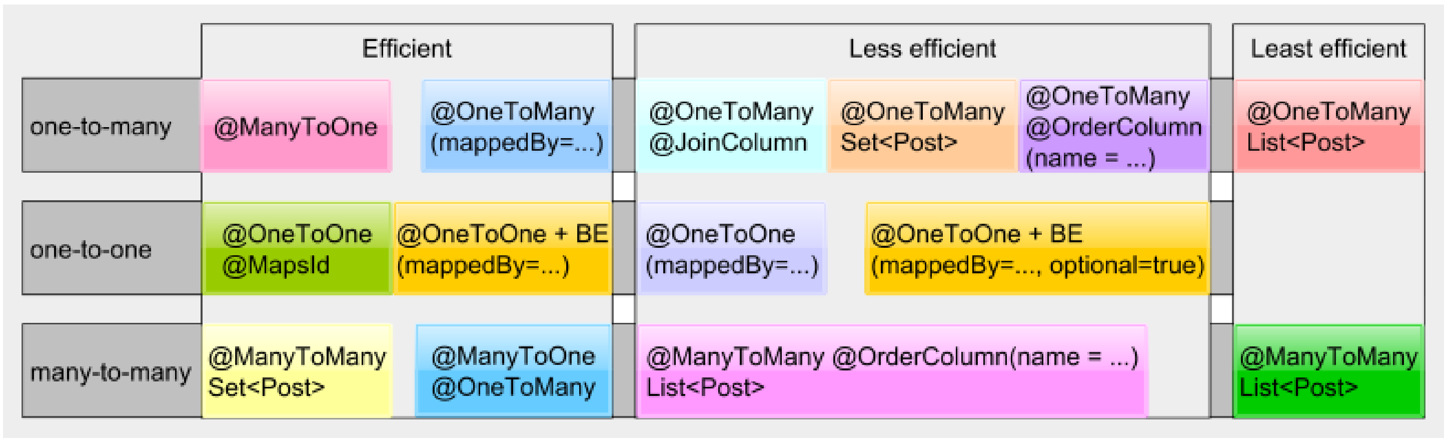
`SEQUENCE` is the right choice, and even SQL Server supports since version 2012. For `SEQUENCE` identifiers, Hibernate has long been offering optimizers like pooled or pooled-lo (/2014/07/21/hibernate-hidden-gem-the-pooled-lo-optimizer/) which can reduce the number of database roundtrips required for 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 (/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.



	Efficient	Efficient	Less efficient	Less efficient	Less efficient	Least efficient
one-to-many	@ManyToOne	@OneToMany (mappedBy=...)	@OneToMany @JoinColumn	@OneToMany Set<Post>	@OneToMany @OrderColumn (name = ...)	@OneToMany List<Post>
one-to-one	@OneToOne @MapsId	@OneToOne + BE (mappedBy=...)	@OneToOne (mappedBy=...)	@OneToOne + BE (mappedBy=..., optional=true)		
many-to-many	@ManyToMany Set<Post>	@ManyToOne @OneToMany	@ManyToMany @OrderColumn(name = ...) List<Post>			@ManyToMany List<Post>

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

Unidirectional collections (/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 (/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 `@OneToMany` associations 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 (/2014/08/21/the-anatomy-of-hibernate-dirty-checking/) is going to be more efficient as well.

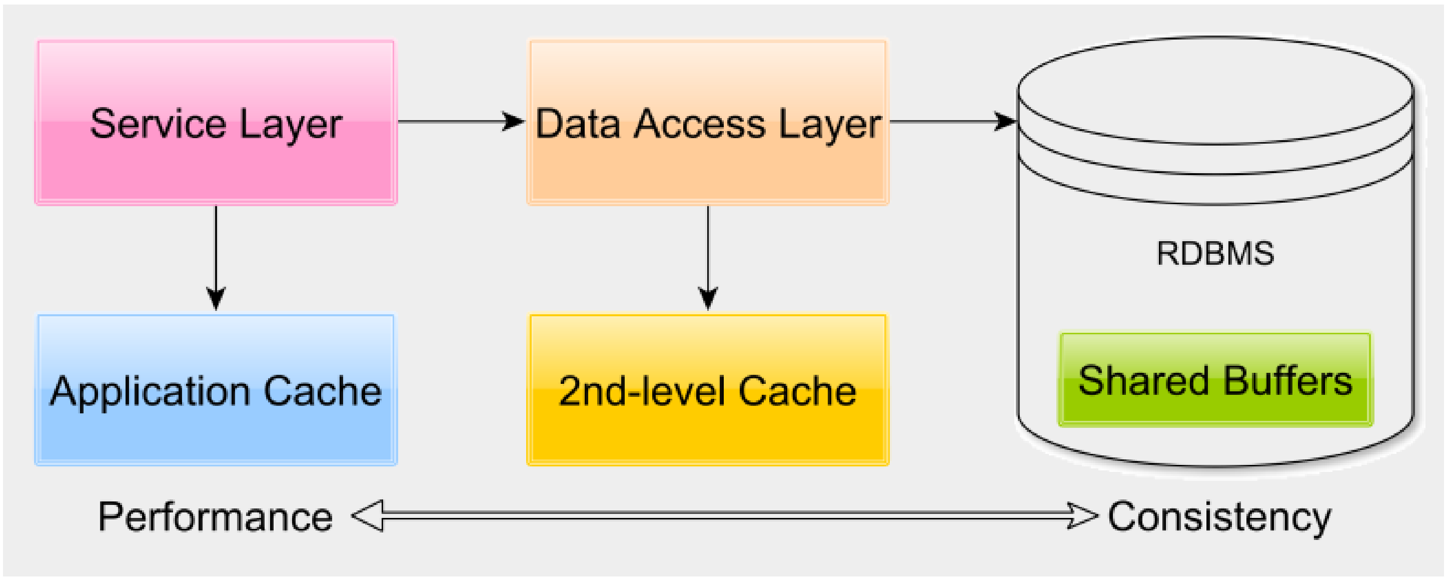
10. Fetching only what's necessary

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 are better suited for fetching custom views (/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 (/2014/12/15/eager-fetching-is-a-code-smell/), and you should avoid anti-patterns such as Open-Session in View (/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 (/2017/02/14/how-does-a-relational-database-work/). Database caching is very often overlooked (/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 (/2015/04/27/how-does-hibernate-read_only-cacheconcurrencystrategy-work/), NONSTRICT_READ_WRITE (/2015/05/18/how-does-hibernate-nonstrict_read_write-cacheconcurrencystrategy-work/), READ_WRITE (/2015/05/25/how-does-hibernate-read_write-cacheconcurrencystrategy-work/), and TRANSACTIONAL (/2015/06/01/how-does-hibernate-transactional-cacheconcurrencystrategy-work/).

12. Concurrency control

The choice of transaction isolation level (/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 (/2014/09/14/a-beginners-guide-to-database-locking-and-the-lost-update-phenomena/), you should use optimistic locking with detached entities or an `EXTENDED` Persistence Context (/2014/09/22/preventing-lost-updates-in-long-conversations/).

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

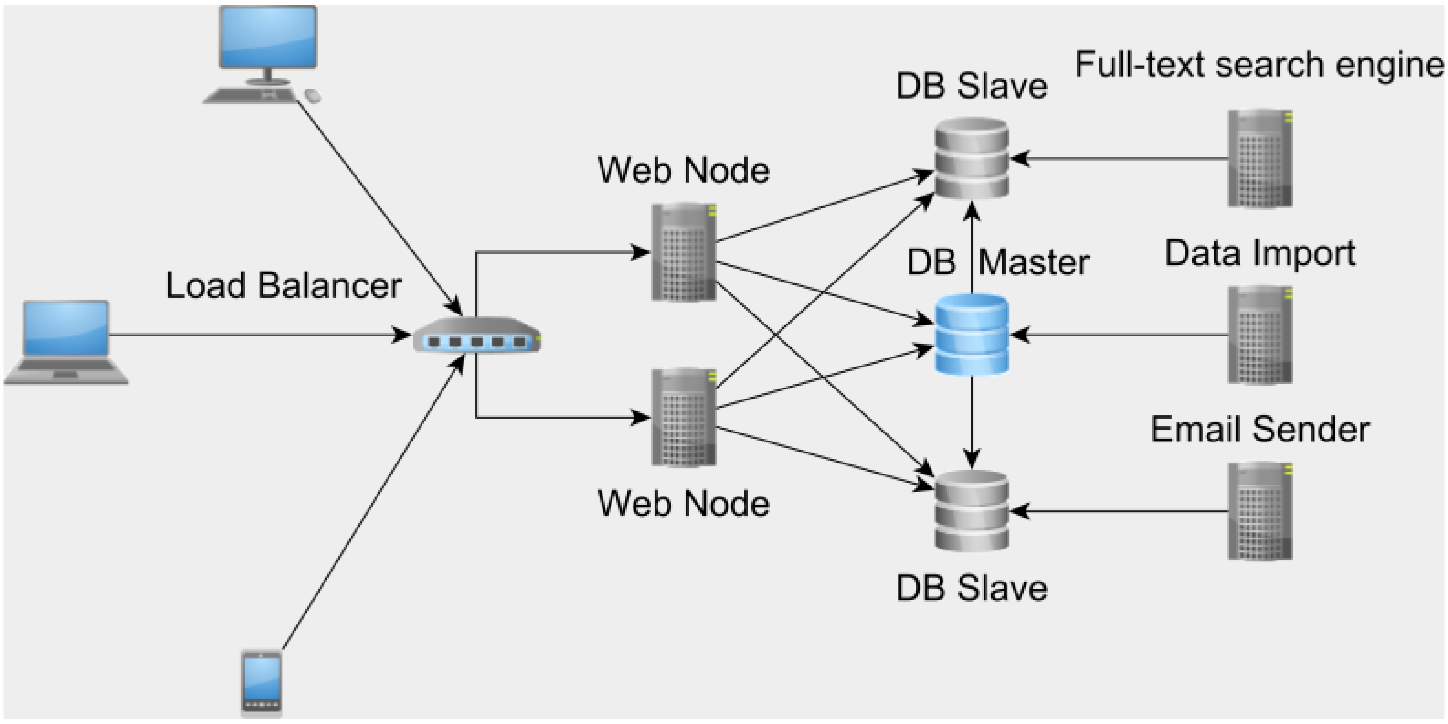
13. Unleash database query capabilities

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 (/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

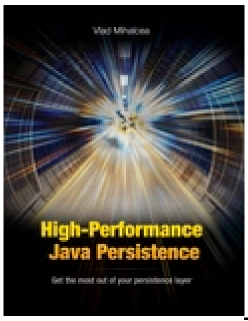
Relational databases do scale very well. If Facebook (http://highscalability.com/blog/2010/11/4/facebook-at-13-million-queries-per-second-recommends-minimiz.html), Twitter (http://highscalability.com/blog/2011/12/19/how-twitter-stores-250-million-tweets-a-day-using-mysql.html), Pinterest (http://highscalability.com/blog/2013/4/15/scaling-pinterest-from-0-to-10s-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.

If you enjoyed this article, I bet you are going to love my book (https://leanpub.com/high-performance-java-persistence?utm_source=blog&utm_medium=banner&utm_campaign=article) as well.


Vlad Mihalcea
High-Performance Java Persistence
Get the most out of your persistence layer
(https://leanpub.com/high-performance-java-persistence?utm_source=blog&utm_medium=banner&utm_campaign=article)

Conclusion

A high-performance data access layer must resonate with the underlying database system. Knowing the inner workings of a relational database and the data access frameworks in use can make the difference between a high-performance enterprise application and one that barely crawls.

There are many things you can do to improve the performance of your data access layer, and I'm only scratching the surface here.
If you want to read more on this particular topic, you should check my High-Performance Java Persistence (https://leanpub.com/high-performance-java-persistence) book as well. With over 450 pages, this book explains all these concepts in great detail.

Follow @vlad_mihalcea 5,114 followers

If you liked this article, you might want to subscribe to my newsletter (http://eepurl.com/bg3d3n) too.

Advertisements

Dell Inspiron Core i3 6th G…

15.6 inch, Black, 2.36 kg

Buy now

Dell Vostro 3000 Core i5 7t…

Categories: Hibernate, Java Tags: Database, hibernate, High-Performance Java Persistence, jpa, performance tuning, tips

8 thoughts on "14 High-Performance Java Persistence Tips"

1. **Ban Ăn Chơi** says:
JUNE 29, 2016 AT 12:53 PM
Thanks, nice tips

REPLY

2. **shubham** says:
   DECEMBER 12, 2016 AT 2:04 PM
   Thanks for the tips.Very helpful.

   REPLY
3. **Zhiqiang Liu** says:
   DECEMBER 13, 2016 AT 4:12 PM
   Very nice tips. For Inheritance strategy, can you talk about more? For table joined strategy, why you say it is the big performance impact for @OneToMany associations against base types ?

   REPLY
   1. **vladmihalcea** says:
      DECEMBER 13, 2016 AT 4:22 PM
      My book, High-Performance Java Persistence, gives a very detail explanation for this particular topic.

      REPLY
      1. **Zhiqiang Liu** says:
         DECEMBER 13, 2016 AT 4:58 PM
         Thanks, I will buy it :-). BTW, can you help me to take a look at this issue?
         http://stackoverflow.com/questions/41104720/an-issue-when-i-use-unidirectional-one-to-many-and-table-per-class-inheritancety
4. **Arno** says:
   FEBRUARY 19, 2017 AT 7:26 PM
   Thanks a lot for this article, it covers many important pitfalls and is very comprehensible.

   The most common mistake I encountered in real-life projects is failing to follow "10. Fetching only what's necessary". E.g. by trying to eager-load several to-many associations within the same query, one actually ends up with join-duplicates (AKA cartesian products), hence duplicate mapped object instantiations.

   This can often be avoided by multiple queries, fetch="subselect" or, as you stated, DTO projections, and is also explained in this little Hibernate Performance Tuning recommendation list.

   REPLY
5. **Mehmet A** says:
   MAY 27, 2017 AT 6:01 PM
   Thanks for the article. What does BE(OneToOne + BE) stands for?

   REPLY
   1. **vladmihalcea** says:
      MAY 27, 2017 AT 6:08 PM
      Thanks. BE stands for Bytecode Enhancement.

      REPLY