28. Performance Tuning and Best PracticesEvery enterprise system is unique. However, having a very efficient data access layer is a common requirement for many enterprise applications. Hibernate comes with a great variety of features that can help you tune the data access layer. 28.1. Schema management Although Hibernate provides the update option for the hibernate.hbm2ddl.auto configuration property, this feature is not suitable for a production environment. An automated schema migration tool (e.g. Flyway, Liquibase) allows you to use any database-specific DDL feature (e.g. Rules, Triggers, Partitioned Tables). Every migration should have an associated script, which is stored on the Version Control System, along with the application source code. When the application is deployed on a production-like QA environment, and the deployment worked as expected, then pushing the deployment to a production environment should be straightforward since the latest schema migration was already tested. You should always use an automatic schema migration tool and have all the migration scripts stored in the Version Control System. 28.2. Logging Whenever you’re using a framework that generates SQL statements on your behalf, you have to ensure that the generated statements are the ones that you intended in the first place. There are several alternatives to logging statements. You can log statements by configuring the underlying logging framework. For Log4j, you can use the following appenders: ### log just the SQL
log4j.logger.org.hibernate.SQL=debug
### log JDBC bind parameters and extracted values ###
log4j.logger.org.hibernate.type=trace
log4j.logger.org.hibernate.orm.jdbc.bind=trace
log4j.logger.org.hibernate.orm.jdbc.extract=trace However, there are some other alternatives like using datasource-proxy or p6spy. The advantage of using a JDBC Driver or DataSource proxy is that you can go beyond simple SQL logging: statement execution time JDBC batching logging database connection monitoring Another advantage of using a DataSource proxy is that you can assert the number of executed statements at test time. This way, you can have the integration tests fail when a N+1 query issue is automatically detected. While simple statement logging is fine, using datasource-proxy or p6spy is even better. 28.3. JDBC batching JDBC allows us to batch multiple SQL statements and to send them to the database server into a single request. This saves database roundtrips, and so it reduces response time significantly. Not only INSERT and UPDATE statements, but even DELETE statements can be batched as well. For INSERT and UPDATE statements, make sure that you have all the right configuration properties in place, like ordering inserts and updates and activating batching for versioned data. Check out this article for more details on this topic. For DELETE statements, there is no option to order parent and child statements, so cascading can interfere with the JDBC batching process. Unlike any other framework which doesn’t automate SQL statement generation, Hibernate makes it very easy to activate JDBC-level batching as indicated in the Batching chapter. 28.4. Mapping Choosing the right mappings is very important for a high-performance data access layer. From the identifier generators to associations, there are many options to choose from, yet not all choices are equal from a performance perspective. 28.4.1. Identifiers When it comes to identifiers, you can either choose a natural id or a synthetic key. For natural identifiers, the assigned identifier generator is the right choice. For synthetic keys, the application developer can either choose a randomly generated fixed-size sequence (e.g. UUID) or a natural identifier. Natural identifiers are very practical, being more compact than their UUID counterparts, so there are multiple generators to choose from: IDENTITY SEQUENCE TABLE Although the TABLE generator addresses the portability concern, in reality, it performs poorly because it requires emulating a database sequence using a separate transaction and row-level locks. For this reason, the choice is usually between IDENTITY and SEQUENCE. If the underlying database supports sequences, you should always use them for your Hibernate entity identifiers. Only if the relational database does not support sequences (e.g. MySQL 5.7), you should use the IDENTITY generators. However, you should keep in mind that the IDENTITY generators disables JDBC batching for INSERT statements. If you’re using the SEQUENCE generator, then you should be using the enhanced identifier generators that were enabled by default in Hibernate 5. The pooled and the pooled-lo optimizers are very useful to reduce the number of database roundtrips when writing multiple entities per database transaction. 28.4.2. Associations Jakarta Persistence offers four entity association types: @ManyToOne @OneToOne @OneToMany @ManyToMany And an @ElementCollection for collections of embeddables. Because object associations can be bidirectional, there are many possible combinations of associations. However, not every possible association type is efficient from a database perspective. The closer the association mapping is to the underlying database relationship, the better it will perform. On the other hand, the more exotic the association mapping, the better the chance of being inefficient. Therefore, the @ManyToOne and the @OneToOne child-side association are best to represent a FOREIGN KEY relationship. The parent-side @OneToOne association requires bytecode enhancement so that the association can be loaded lazily. Otherwise, the parent-side association is always fetched even if the association is marked with FetchType.LAZY. For this reason, it’s best to map @OneToOne association using @MapsId so that the PRIMARY KEY is shared between the child and the parent entities. When using @MapsId, the parent-side association becomes redundant since the child-entity can be easily fetched using the parent entity identifier. For collections, the association can be either: unidirectional bidirectional For unidirectional collections, Sets are the best choice because they generate the most efficient SQL statements. Unidirectional Lists are less efficient than a @ManyToOne association. Bidirectional associations are usually a better choice because the @ManyToOne side controls the association. Embeddable collections (@ElementCollection) are unidirectional associations, hence Sets are the most efficient, followed by ordered Lists, whereas bags (unordered Lists) are the least efficient. The @ManyToMany annotation is rarely a good choice because it treats both sides as unidirectional associations. For this reason, it’s much better to map the link table as depicted in the Bidirectional many-to-many with link entity lifecycle section. Each FOREIGN KEY column will be mapped as a @ManyToOne association. On each parent-side, a bidirectional @OneToMany association is going to map to the aforementioned @ManyToOne relationship in the link entity. Just because you have support for collections, it does not mean that you have to turn any one-to-many database relationship into a collection. Sometimes, a @ManyToOne association is sufficient, and the collection can be simply replaced by an entity query which is easier to paginate or filter. 28.5. Inheritance Jakarta Persistence offers SINGLE\_TABLE, JOINED, and TABLE\_PER\_CLASS to deal with inheritance mapping, and each of these strategies has advantages and disadvantages. SINGLE\_TABLE performs the best in terms of executed SQL statements. However, you cannot use NOT NULL constraints on the column-level. You can still use triggers and rules to enforce such constraints, but it’s not as straightforward. JOINED addresses the data integrity concerns because every subclass is associated with a different table. Polymorphic queries or @OneToMany base class associations don’t perform very well with this strategy. However, polymorphic @ManyToOne associations are fine, and they can provide a lot of value. TABLE\_PER\_CLASS should be avoided since it does not render efficient SQL statements. 28.6. Fetching Fetching too much data is the number one performance issue for the vast majority of Jakarta Persistence applications. Hibernate supports both entity queries (JPQL/HQL and Criteria API) and native SQL statements. Entity queries are useful only if you need to modify the fetched entities, therefore benefiting from the automatic dirty checking mechanism. For read-only transactions, you should fetch DTO projections because they allow you to select just as many columns as you need to fulfill a certain business use case. This has many benefits like reducing the load on the currently running Persistence Context because DTO projections don’t need to be managed. 28.6.1. Fetching associations Related to associations, there are two major fetch strategies: EAGER LAZY EAGER fetching is almost always a bad choice. Prior to Jakarta Persistence, Hibernate used to have all associations as LAZY by default. However, when Java Persistence 1.0 specification emerged, it was thought that not all providers would use Proxies. Hence, the @ManyToOne and the @OneToOne associations are now EAGER by default. The EAGER fetching strategy cannot be overwritten on a per query basis, so the association is always going to be retrieved even if you don’t need it. Moreover, if you forget to JOIN FETCH an EAGER association in a JPQL query, Hibernate will initialize it with a secondary statement, which in turn can lead to N+1 query issues. So, EAGER fetching is to be avoided. For this reason, it’s better if all associations are marked as LAZY by default. However, LAZY associations must be initialized prior to being accessed. Otherwise, a LazyInitializationException is thrown. There are good and bad ways to treat the LazyInitializationException. The best way to deal with LazyInitializationException is to fetch all the required associations prior to closing the Persistence Context. The JOIN FETCH directive is good for @ManyToOne and OneToOne associations, and for at most one collection (e.g. @OneToMany or @ManyToMany). If you need to fetch multiple collections, to avoid a Cartesian Product, you should use secondary queries which are triggered either by navigating the LAZY association or by calling Hibernate#initialize(Object proxy) method. 28.7. Caching Hibernate has two caching layers: the first-level cache (Persistence Context) which provides application-level repeatable reads. the second-level cache which, unlike application-level caches, doesn’t store entity aggregates but normalized dehydrated entity entries. The first-level cache is not a caching solution "per se", being more useful for ensuring READ COMMITTED isolation level. While the first-level cache is short-lived, being cleared when the underlying EntityManager is closed, the second-level cache is tied to an EntityManagerFactory. Some second-level caching providers offer support for clusters. Therefore, a node needs only to store a subset of the whole cached data. Although the second-level cache can reduce transaction response time since entities are retrieved from the cache rather than from the database, there are other options to achieve the same goal, and you should consider these alternatives prior to jumping to a second-level cache layer: tuning the underlying database cache so that the working set fits into memory, therefore reducing Disk I/O traffic. optimizing database statements through JDBC batching, statement caching, indexing can reduce the average response time, therefore increasing throughput as well. database replication is also a very valuable option to increase read-only transaction throughput. After properly tuning the database, to further reduce the average response time and increase the system throughput, application-level caching becomes inevitable. Typically, a key-value application-level cache like Memcached or Redis is a common choice to store data aggregates. If you can duplicate all data in the key-value store, you have the option of taking down the database system for maintenance without completely losing availability since read-only traffic can still be served from the cache. One of the main challenges of using an application-level cache is ensuring data consistency across entity aggregates. That’s where the second-level cache comes to the rescue. Being tightly integrated with Hibernate, the second-level cache can provide better data consistency since entries are cached in a normalized fashion, just like in a relational database. Changing a parent entity only requires a single entry cache update, as opposed to cache entry invalidation cascading in key-value stores. The second-level cache provides four cache concurrency strategies: READ\_ONLY NONSTRICT\_READ\_WRITE READ\_WRITE TRANSACTIONAL READ\_WRITE is a very good default concurrency strategy since it provides strong consistency guarantees without compromising throughput. The TRANSACTIONAL concurrency strategy uses JTA. Hence, it’s more suitable when entities are frequently modified. Both READ\_WRITE and TRANSACTIONAL use write-through caching, while NONSTRICT\_READ\_WRITE is a read-through caching strategy. For this reason, NONSTRICT\_READ\_WRITE is not very suitable if entities are changed frequently. When using clustering, the second-level cache entries are spread across multiple nodes. When using Infinispan distributed cache, only READ\_WRITE and NONSTRICT\_READ\_WRITE are available for read-write caches. Bear in mind that NONSTRICT\_READ\_WRITE offers a weaker consistency guarantee since stale updates are possible. For more about Hibernate Performance Tuning, check out the High-Performance Hibernate presentation from Devoxx France.