**Top Performance issues every developer/architect must know — part 1-Database**



Hello everyone. This is one of the interesting articles I am writing as it talks about the performance issue in an application. This will help everyone to understand where are those issues happening and how to fix them if they come across.

Let me classify the issues into different segments as follows

* Database problems
* Memory problems
* Concurrency problems

Let me talk about database-related performance problems in this article

**1. Database**

Let us look at the database problems as they are the most critical ones. These are the common issues that occur at the DB level which hampers the latency of the application. They are as follows

**a. The database N+1 Problem**

A very common problem in DB is the [“N+1” problem](https://www.java67.com/2016/02/top-20-hibernate-interview-questions.html). The ***N+1 query problem*** is said to occur when an ORM, like [hibernate](https://medium.com/javarevisited/8-best-spring-and-hibernate-training-courses-for-java-developers-acf09aa0e244), executes 1 query to retrieve the parent entity and N queries to retrieve the child entities. As the number of entities in the database increases, the queries being executed separately can easily affect the performance of the application

Let’s say I wanted to retrieve the last 100 entries from my Order table. To do this, I’d have to first execute a query to find the [primary keys](https://www.sqlrevisited.com/2022/02/what-is-primary-key-in-sql-how-to.html) of each item, such as:

**SELECT id FROM Order WHERE** … And then I’d execute one query for each record: **SELECT \* FROM Order WHERE id =?**

In other words, it would take me **101 queries to retrieve 100 records (N+1)**. Persistence technologies have improved since then, but the “N+1” problem hasn’t completely disappeared.

**b. Solution - EntityGraph**

**EntityGraphs** provide a mechanism by which entities can be fetched eagerly from the [database](https://medium.com/javarevisited/top-10-free-courses-to-learn-microsoft-sql-server-and-oracle-database-in-2020-6708afcf4ad7)in a single select statement, helping in improving the performance of the application. You can also use Subgraphs to define the entities for the child class, that needs to be fetched eagerly along with the parent class

***EntityGraphs*** provides a way to formulate better-performing queries by defining which entities need to be retrieved from the database using SQL JOINS. There are two types of EntityGraphs, ***Fetch*** and ***Load***, which define if the entities ***not*** specified by attribute nodes of EntityGraphs should be fetched ***lazily*** or ***eagerly***. Attributes specified by attribute nodes of EntityGraph are *always fetched eagerly*. Thus, N+1 queries were reduced to only a single query using [JOINS](https://javarevisited.blogspot.com/2020/05/self-join-example-sql-query-to-find-employee-more-than-managers-leetcode-solution.html)to fetch data from both tables.

**2. Caching**

If we do not use a cache, there is a latency in hitting the database every time. The frequently used data can be cached and serving the content from memory is super fast. But there comes the problem of how to configure the cache.

There are different types of Caching strategies namely

**a. Level 2 Cache or In-Memory Cache**

This cache sits between the persistence provider like Hibernate and the database. It is called an In-Memory cache because it uses the RAM from the machine where the server is hosted. This is very useful in the case of a Monolith application where there is only a single instance of the application running.

The famous ones are **Hazelcast, Caffeine Cache, Eh-Cache**, etc. Once we configure the dependencies, a Spring boot application will automatically use it as an in-memory Cache.

**b. Distributed Cache**

If the application runs as a [Microservices architecture](https://medium.com/javarevisited/8-best-online-courses-to-learn-service-oriented-soa-and-microservices-architecture-94c01d6b94e6), every service will be scaling up and down most likely. Hence there will be more than one instance of an application. In that case, we cannot use the In-Memory cache because the other instance will not have the access to data in Cache. There are scenarios where a request can be routed to different instances by the load balancer.

Here comes the usage of *Distributed Caching* which is a stand-alone cache that runs across multiple instances in the cluster, that can be used by any application in the cluster. The common use case for the distributed cache is storing a user cart or a login JWT token.

The famous ones are [Redis](https://javarevisited.blogspot.com/2022/02/top-5-courses-to-learn-redis.html), Memcached, etc

**c. Cache Consistency**

Cache consistency sometimes referred to as cache coherence, refers to the validity of data across your entire cache instances. A cache is consistent if every instance of the same object in the cache has the same value. In recent times, large-scale applications have adopted eventual consistency, which means that there will be periods of time when different instances of the same object will have different values, but eventually, they will have the same value which is similar to the replication in the database instances

**d. Problems with Caching**

The main symptom of an application that is not using a cache properly is increased database load and slow response times as a result of that load. If you’re using caching correctly, database load should not increase in proportion to application load, because the majority of requests should be hitting the cache. The negative impact of this problem is that as your load increases, the database machine is subjected to more load, which increases its CPU overhead and potentially disk I/O rate, which degrades the performance

**3. Database Connection Pools**

**a. What is a Connection Pool**

It is a pool of connections that are used by the application to access the database. There are various Connection Pool providers and a famous one for [Spring Boot](https://medium.com/javarevisited/top-10-courses-to-learn-spring-boot-in-2020-best-of-lot-6ffce88a1b6e) is HikariCP. When the application is initialized, the provider creates a default of 10 connections per server instance and keeps them in its pool. This DB connection pool resides in the Server memory. When the application needs the connections, these connections from the pool are recycled as creating new connections for every request is a costly operation.

Database connections are pooled for several reasons:

* Database connections are relatively expensive to create, so rather than create them on the fly we opt to create them beforehand and use them whenever we need to access the database.
* The database is a shared resource so it makes sense to create a pool of connections and share them across all business transactions.
* The[database connection pool](https://javarevisited.blogspot.com/2012/06/jdbc-database-connection-pool-in-spring.html) limits the amount of load that you can send to your database.

**b. Problems**

We pool connections to reduce the load on the database because otherwise we might saturate the database with too much load and bring it to a halt. The point is that not only do you want to pool your connections, but you also need to configure the size of the pool correctly.

If you do not have enough connections, then business transactions will be forced to wait for a connection to become available before they can continue processing.

If you have too many connections, however, then you might be sending too much load to the database and then all business transactions across all application servers will suffer from [slow database performance](http://javarevisited.blogspot.sg/2012/01/improve-performance-java-database.html). The trick is finding the middle ground

The main symptoms of a database connection pool that is sized too small are increased response time across multiple business transactions, with the majority of those business transactions waiting and the symptoms of a database connection pool that is sized too large are increased response time across multiple business transactions, with the majority of those business transactions waiting on the response from queries, and high resource utilization in the database machine.

**c. Solution**

The fix is simple but will require time and effort: use load testing and a performance analysis tool to find the optimal value for the size of your database connection pool, and then make the configuration change.