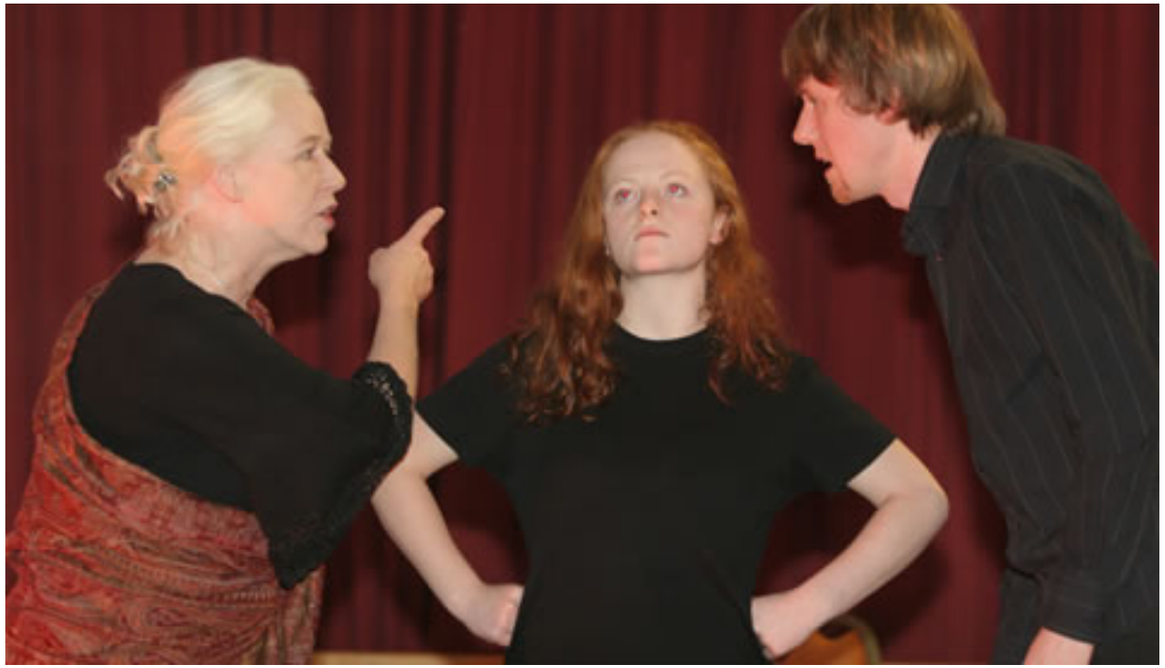
How to use JPA correctly to avoid complaints of a slow application

January 23, 2015 Roberto Cortez 21 Comments



Setting the stage with a short story

The day is clear and sunny, birds chirping happily as Lea, a project manager, makes her way to a meeting with Robert, a development team lead. “Dude, our application is slow,” she tells Robert.

Robert smirks. “Our code is impeccable, so this is probably a database problem. Let me ask our database admin, Dan,” and he picks up the phone.

“Hey Dan,” Robert says, “Lea says the application is slow. Can’t you create some indexes, or do something to speed it up?”

“Just deal with it!” exclaims Robert, hanging up.

“%$@#!!” mutters Dan.

Over the last few years, I’ve come across with several enterprise applications that use JPA to manage their data, which is cool since JPA is a very powerful and awesome specification. Unfortunately, I came to realize that this technology is commonly used improperly, which generates a lot of complaints and even full-scale wars between database administrators (DBAs) and developers.

If you have some basic knowledge of JPA (which you should to get the most out of this article), then I can bet that many of you have heard a similar exchange before.

The double-edged sword of JPA

A great thing about JPA is that abstracts your interaction with the underlying database. A bad thing about JPA is that abstracts your interaction with the underlying database.

You can write database access code very easily and get most of the general database operations out of the box without having to write all that tedious JDBC code. On the other hand, you also need to have some knowledge of what’s going on behind the scenes or you’ll be in for some unpleasant surprises.

concerned about getting the data they need, and don't worry about anything else.

And this is why I decided to write this article: I've seen the same mistakes repeated over and over again, and they actually have a huge performance impact.

I've written down four areas in which I usually find all the issues: these are the ones I check first when I have to hunt down JPA performance problems.

- Eager fetching
- Lazy fetching
- Pagination
- Column select

Shamefully, I always find them out during my analysis after the fact, even though they seem completely obvious! In any case, I believe that these mistakes are not deliberately put into the application–most of the time, it's poor knowledge about the technology itself. I hope this article could raise some awareness and guide developers to write better and faster JPA code.

There's also another great post by Thorben Janssen where he talks about JPA features that can help with the application performance: boosting your application performance with JPA.

Eager Fetching

Let's imagine that you have a Department entity with an **@OneToMany** relationship to Employee defined with Eager Fetching. If you have a page where you are only displaying the Departments, you are also selecting the Employees for each Department. This scenario suffers from an increase of the loading time of data that you don't need to fulfill your requirement.

So ask yourself: Do you really need all the data that you eagerly fetched? Most likely you don't, not for every situation. So unless you know what you are doing, don't use it.

Lazy Fetching

The Lazy Fetching strategy is a hint to the persistence provider runtime that data should be fetched lazily when it is first accessed. It seems like a good thing and usually it is, but sometimes it can slow down your application. Have you ever heard about the N+1 select problem? This happens when you select an entity and then iterate the results to access a collection in a lazy fashion.

```
      List departments = entityManager.createQuery("select d from
Department d").getResultList();
        for (Department department : departments) {
            // Issues a “select * from Employee where department
            List employees = department.getEmployees();
        }
```

Employee data, you could write the query like this:

```
List departments = entityManager.createQuery("select d from
Department d left join fetch d.employees").getResultList();
```

In this case, only a single database query is performed with all the data already populated in the return results.
Keep in mind, that Lazy Fetching is just giving a hint to the provider. The implementation is permitted to eagerly fetch data for which the Lazy strategy hint has been specified, but the most popular ones have the same behavior regarding collections and the example I just gave.

Pagination for a quick win

Paginating your results is probably one of the best ways to increase the performance of your JPA application. If you have a table with 1 million records you are not going to display them all, right? RIGHT? Performing pagination on the client side is not the answer either, because the database had to return all the records anyway. Pagination should be done directly into the database, and you only have to call...

```
setFirstResult();
setMaxResults();
```

...in the Query object to paginate results.

Smart column select

or other big sized data types? Even if you don't, you should only select the required columns for the operation you are trying to perform. This will reduce the amount of data sent by the database to your application and speedup the query that you are executing.

Instead of writing:

```
entityManager.createQuery("select d from Department d")
```

You can write, if you only need the Department id:

```
entityManager.createQuery("select d.id from Department d")
```

The numbers that prove it all

Ok, so you probably thinking that this is all hogwash, and until you see some real numbers, there is no way this could be true.

I compiled a few test cases with some of the scenarios presented above and you can check the results in the table below. These are very simple tests: I added 2000 Departments with 100 columns each, and Departments have a one-to-many relationship with Employees, where each Department record has two Employees and Employees each have 100 columns.

Find all records with relationships set to Lazy (n+1 selects)	502 ms
Find all records with relationships set to Eager	210 ms

relationships are fetched on the query	206 ms
Find all records without relationships	59 ms
Find all records without relationships, only with 10 columns	12 ms
Find all records without relationships, only with 10 columns, paginated	**8 ms**

The tests were run one at a time on three separate occasions, using Wildfly, Hibernate and H2 as the provider and with a database in a local environment. If you'd like to try it out yesterday, get the code on my GitHub page: https://github.com/radcortez/jpa-performance

Two things you can do to avoid this mess

1. Know your JPA Provider – Hibernate, EclipseLink or OpenJPA are probably the most well known JPA Providers. While providers have to comply with the standards, the specification is open in a few scenarios, which may cause different behaviors for each implementation. Think about the Lazy hint strategy that I explained a couple of lines ago as an example.

The Provider has a considerable amount of impact on your JPA application performance, so you should try to understand these little bits to get the most

mind to sacrifice portability. Stay tuned; I'm planning to write another blog post exploring these on my own blog.

2. Consider finding a DBA to join your team – Databases are complex pieces of software. There are a lot of ways you can optimize and increase the performance of your queries and this also depends on the database engine that you're using. In my opinion, having a specialist with you that can help you write optimized queries and monitor the application load is most of the times underrated.

In one of my previous jobs, I used to hear this a lot: "Database guys are not needed!" and I couldn't disagree more. The most successful projects I have ever worked I always had a DBA backing me up. Probably without their help, I would be stuck working in some random car wash.

Conclusion: Boost app performance significantly by using JPA the right way

I think the numbers speak by themselves. The n + 1 select had the worst performance as expected. Both tests for relationships set to Eager or fetched on the query have similar results, since the query performed is the same, but remember it may be getting data that you don't really need. Now, performing the query without the relationships have a very good boost, but you get better results if you just select the columns you need and squeeze the last extra bit by paginating the results.

Keep in mind that these times are only to demonstrate an order of magnitude between the different scenarios. Applying these techniques may not give you a

help you to develop faster applications.

I hope you enjoyed this article, and feel free to leave me comments below, or ping me on Twitter @radcortez. For more tech goodness, check out some **downloadable RebelLabs Reports!**

TAKE ME THERE

ROBERTO CORTEZ
RebelLabs Author
Roberto Cortez has been a professional Java Developer for the last eight years, working with technologies like JavaEE, Spring, Hibernate, GWT, JBoss AS and Maven, but saving his fanboy-ism for his favorite IDE: IntelliJ IDEA. Recently, he created the Coimbra Java User Group and started on the path of a freelancer, traveling around the world (an old dream) to customers and Java conferences.

DBA, JPA, performance, RebelLabs

what a great post, I have the big picture now, thank you !!! **@Roberto Cortez**

11 · Reply · Share ›

Tim Gatzemeier • a year ago

Great article! Thanks.
Would like to leave a hint about BatchSize for hibernate users. But as mentioned here http://stackoverflow.com/a/... its not an alternative to eager loading when necessary. Its more like, look if you need lazy loading and want to speed up your queries, use it!

1 · Reply · Share ›

Mauricio Barros • a year ago

Great article. You saved my live, Bro! TKs

Reply · Share ›

Akouri Ammar • a year ago

Great article . Thank you !

Reply · Share ›

irufus • 3 years ago

Why would anyone use FetchType.EAGER in the first place?

Reply · Share ›

max2u ➚ irufus • 3 years ago

it might make lots of sense when having a @ManyToOne relation, with the @Fetch(FetchMode.JOIN).
this will fetch everything in one shot

Reply · Share ›

John • 4 years ago

I enjoyed your article, thanks for sharing!

Hello. Even Brazil losing by 7x1, I am Brazilian. Sorry my english.

How can I improve this situation: Today I have an object with:
Two @ManyToMany you need to get the list of id;
One @OneToOne get two fields;

Today I am drawing the first object with everything Lazy because I need these links only in a query.

Reply • Share ›

Roberto Cortez ➚ Luís Antonio De Marchi • 4 years ago

Hi Luís,

I'm sorry for Brazil. I was also supporting them :(

Anyway, I'm not sure if I get your problem, but I can speak Portuguese so you might want to private me. Just have a look into my blog for my contacts: http://www.radcortez.com/

Reply • Share ›

Lukas Eder • 4 years ago

Great post! These things cannot be said often enough. There are more to add, e.g. processing data in Java memory instead of the database, when doing heavy calculations, using `DISTINCT` or `UNION` (instead of `UNION ALL`) where not absolutely needed, using in-memory sorting, instead of DB sorting, not using batch insert (you did this in your benchmark, from what I can tell!)

We've blogged about these things here and here.

I'll also challenge your 8ms in a response blog post, soon. 8ms is an eternity on a decent database. I'm sure this can be speeded up with plain SQL (using JDBC or jOOQ)

Reply • Share ›

to speed up the times, but these are very basic stuff that increase your performance by a large amount and usually the thing that you should look into first.

Thank you for the links, I will check them later, and please send me your response blog post when you have it ready :)

Reply • Share ›

Lukas Eder ➔ Roberto Cortez • 4 years ago

> and please send me your response blog post when you have it ready :)

I sure will!

Reply • Share ›

Charles • 4 years ago

That's a very helpful article. Thank you.

Reply • Share ›

Roberto Cortez ➔ Charles • 4 years ago

Hi Charles,

Thank you so much for your feedback.

Reply • Share ›

Tom Davis • 4 years ago

"If you'd like to try it out yesterday, get the code ..."

You built a time machine!? ;)

Reply • Share ›

Roberto Cortez ➔ Tom Davis • 4 years ago

Hehe, no but it would be cool if I had. Just an unfortunate mistake. Thanks for

Reply • Share ›

cst1992 ➚ Roberto Cortez • 5 months ago

Not fixed yet. Just a heads up!

Reply • Share ›

Ted Velkoff • a year ago

Roberto, this is a very helpful article but it doesn't address an issue that I'm having: incredibly long times to start a web application. I've inherited an application that can take from 30 to 60 minutes to start up. Most of that time it seems to be pausing with nothing being logged. Then in spurts, there is activity, usually database related. The application uses JPA with Hibernate provider and runs in WebSphere. If you have any suggestions about even how to gain visibility into what's going on, that would be most helpful. Thanks

Reply • Share ›

it_madness • a year ago

My advice: Prefer Hibernate over EclipseLink.
Apart from that I now prefer the JOIN FETCH for allsituations when you know what data you will need to process. It's just better to have full control in most situations.

And: Nice article, there is nothing I would disagree with.

Reply • Share ›

Anonymouse • 4 years ago

Or... use an XML database like BaseX or Marklogic and focus on solving the business problem you're trying to solve.

Reply • Share ›

David ➚ Anonymouse • 3 years ago

This article is about the performance characteristics of JPA on a relational database. I am mystified as to how your comment is relevant, and doubly-so

ALSO ON ZEROTURNAROUND

RebelLabs Developer Productivity Report 2017: Why do you use the Java

1 comment • 9 months ago

Christ_OFF — Thanks for the report. Very

Maven cheat sheet

1 comment • a year ago

Андрей Турбанов — >-DskipTests=trueNo need to write "=true"

Company

- Careers
- Contact Us
- Our story

Resources

- Developer Productivity Report
- Hidden productivity killer
- All Resources