MASTER'S THESIS 2022

Database Loading Strategies for an In-Memory Cache in Java

Ivar Henckel, David Söderberg

ISSN 1650-2884
LU-CS-EX: 2022-21

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-21

Database Loading Strategies for an In-Memory Cache in Java

Strategier för databasladdning till en in-memory cache i Java

Ivar Henckel, David Söderberg

Database Loading Strategies for an In-Memory Cache in Java

Ivar Henckel
iv4307he@student.lu.se

David Söderberg
da4060so-s@student.lu.se

May 20, 2022

Master's thesis work carried out at Nasdaq Technology AB.

Supervisors: Alma Orucevic-Alagic, alma.orucevic-alagic@cs.lth.se
Inger Klintmalm, inger.klintmalm@nasdaq.com

Examiner: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Abstract

The efficiency of software systems can be negatively affected by database latency which can take a significant fraction of execution time. To mitigate run-time latency, different levels of caching can be introduced with different strategies to load the cache.

In this thesis such strategies are investigated, mainly focusing on lazy loading and parallel preloading of the cache. We implement some of the identified strategies and conduct an experimental analysis of the performance.

All of the strategies are implemented using Java together with the Hibernate ORM framework. The caching strategies could be translated to any other ORM framework.

After running experiments and comparing the collected measurements we conclude that one of the lazy loading solutions relying on Hibernate proxies is inefficient. Another lazy loading solution, which is based on lookup tables, effectively moves the latency from startup to run-time while also removing the cost of fetching data preemptively. The solution using inter-query parallelism with parallel preloading achieves efficient startup and run-time latency when all data is not requested from the cache directly at startup.

In conclusion, the caches using parallel preloading and lookup tables perform the best and are recommended to be used.

Keywords: Databases, Hibernate, ORM, In-memory Cache, Java, Lazy Loading, Inter-query Parallelism

Acknowledgements

We would like to express our gratitude towards our supervisor at the university Alma Orucevic-Alagic. Without her, the completion of this thesis would not have been possible. We also want to thank our examiner Jonas Skeppstedt, both for his work as an examiner and for the knowledge regarding multi-threading we have gained from taking his courses previously.

The whole team at Nasdaq also deserves a big thank you. Thanks to them we have had the opportunity to research our subject in a real life environment while also receiving a lot of feedback from the experienced members of the team we worked in. Special gratitude goes out to Inger Klintmalm, our supervisor at Nasdaq who provided us with her expertise and weekly discussions. The team manager, Zhe Wang, also deserves special acknowledgment for providing the opportunity to conduct our master thesis at Nasdaq.

Our friends and family who have provided feedback and discussion around the master thesis also deserve acknowledgment. In particular, we want to thank our opponents, Jonathan Frisk and Linnea Johnsson who have given us appreciated feedback on the thesis report.

Last but not least, we want to thank the authors of the papers on which we have based our information. The Hibernate community members which are active on internet forums have also been instrumental to get an initial understanding of Hibernate. A special thank you goes out to Thorben Janssen because of his instructive videos on Hibernate and ORM.

Contents

1 Introduction **9**

2 Background and Related Work **11**
2.1 Problem Statement . . . 11
2.1.1 Research Questions . . . 12
2.2 Object-Relational Mapping . . . 13
2.2.1 Data Access Object . . . 13
2.2.2 Java Persistence API . . . 14
2.2.3 Hibernate . . . 14
2.2.4 Hibernate Built-In Caches . . . 15
2.2.5 The N + 1 Problem . . . 15
2.2.6 Hibernate Statistics API . . . 16
2.3 Incremental Loading . . . 16
2.3.1 Change Data Capture . . . 17
2.3.2 Lazy Loading . . . 17
2.3.3 Pagination . . . 19
2.4 Parallelizing Database Queries . . . 20
2.4.1 Parallel Query Processing . . . 20
2.4.2 Parallelism in Hibernate . . . 20
2.5 Related Work . . . 21
2.6 Contribution . . . 22
2.6.1 Scientific Contribution . . . 22
2.6.2 Distribution of Work . . . 22

3 Approach **25**
3.1 Method . . . 25
3.2 System Overview . . . 26
3.2.1 Test Environment . . . 26
3.2.2 Application . . . 26
3.2.3 Cache Service . . . 26

3.2.4 Generating Test Data 27
3.3 Performance Measurement in Java 27
3.4 Experiment 28
3.5 Maintainability 30
3.6 Testing 31

4 Result **33**
4.1 RQ1 - Identifying Inefficiencies 33
4.1.1 Scope 33
4.1.2 Overview of Startup Execution Time 34
4.1.3 Software Layers 34
4.2 RQ2 - Proposed Solutions 37
4.2.1 Lazy Loading - Hibernate Proxy 37
4.2.2 Lazy Loading - Lookup Table 40
4.2.3 Asynchronous Preloading 42
4.2.4 Hibernate Built-In Caches 44
4.3 RQ3 - Gathering Experiment Data 45

5 Discussion **51**
5.1 RQ1 - Interpreting Inefficiencies 51
5.2 RQ2 - Dissecting Implementations 53
5.2.1 Lazy Loading - Hibernate Proxy 53
5.2.2 Lazy Loading – Lookup Table 54
5.2.3 Asynchronous Preloading 55
5.2.4 Alternative Solutions 56
5.3 RQ3 - Comparing Solutions 57
5.3.1 Startup Time 57
5.3.2 Run-time Request Latency 58
5.3.3 Maintainability 59
5.4 Future Work 60
5.5 Validity Threats 61
5.5.1 Construct Validity 61
5.5.2 Conclusion Validity 62
5.5.3 Internal Validity 63
5.5.4 External Validity 64

6 Conclusion **65**

References **67**

Appendix A Code **75**
A.1 SQL Script to Generate Data 76
A.2 Hibernate Statistics Command 81
A.3 Mutual Code 82
A.3.1 AccountIndex 82
A.3.2 EntityService 83
A.3.3 EntityDaoHibernate 84

A.3.4 EntityProxyService . . . 86
A.3.5 EntityProxyDaoHibernate . . . 87
A.3.6 EntityProxyInfoDTO . . . 89
A.4 Hibernate Proxy Cache . . . 90
A.4.1 Cache . . . 90
A.4.2 Provider . . . 93
A.5 Lookup Table Cache . . . 95
A.5.1 Cache . . . 95
A.5.2 Provider . . . 98
A.6 Asynchronous Preloading Cache . . . 99
A.6.1 Cache . . . 99
A.6.2 Provider . . . 103
A.7 Experiment Code . . . 105

Chapter 1
Introduction

Each day a massive amount of data is generated and gathered about the global economy and the markets around the world. As a result, large financial corporations have played an important part in the advancement of the field of database management. Trillions of pieces of data are used each day in the financial sector for analysis, decision making, risk management, and more [21]. As the amount of data collected has been growing steadily during the last decades [12][31], many companies have invested in the research of new technologies to improve the efficiencies of databases and the handling of a large amount of data. When new technologies and methods continuously emerge, companies have to evaluate them and adapt for their services to be efficient and stay relevant.

At Nasdaq, a world-leading financial service corporation, the importance of efficiently handling large amounts of data is growing each day. The company has been one of the main drivers of the digitalization of the world's markets and the financial system as a whole, as it was the first company in the world to establish an all-electronic exchange in 1971 [38]. Nasdaq has transitioned with the market to be more of a technology company today. The company's technology stack is not only used to run its own exchanges but also customers' exchanges and other backbone components of the world's financial markets. It is important that Nasdaq's products are fast and efficient as the world of finance moves in small fractions of a second, even as low as nanoseconds [50]. The amount of data processed each day is also huge as, for example, the daily number of transactions are in the hundreds of millions [21]. An implication of this is that the way the data is handled and the use of database management system is of importance to not be a bottleneck of the whole system.

The product that has been examined during this thesis is used for risk mitigation and is part of the core of the financial infrastructure. As the market moves in fractions of a second it is important that this product is fast and reliable. Risk mitigation is key for a stable and well-functioning market. What if the service crashes and needs a failover? How much time will it take to get it up and running again? It could take a long time if there is a massive amount of data that has to be loaded initially and the handling of the data is poor.

The way the data is loaded and handled by the application plays a significant role. In this

thesis, we will call the load between the database and the application at startup an initial load. If there is an initial load between a database and an application that takes too much time, as the size of the data grows, it could create issues and bottlenecks. The current solution has been examined and new strategies have been proposed that handles the load to the cache more efficiently.

The thesis starts with a description of the investigated problem and the relevant theory used in chapter 2. In chapter 3 the method used is laid out, together with a more detailed description of the system and motivation behind the experiments that were conducted. In chapter 4 results are gathered by investigating the initial solution, proposing new solutions, and finally comparing all implementations with experiments. The discussion based on these results is done in chapter 5, together with subsections about future work and possible validity threats. The conclusion can be found in chapter 6.

Chapter 2

Background and Related Work

2.1 Problem Statement

Nasdaq has many big internal projects with millions of lines of code. However, only a minor part of the code base is related to our master thesis work. In figure 2.1 we present a simplified view of the system architecture.

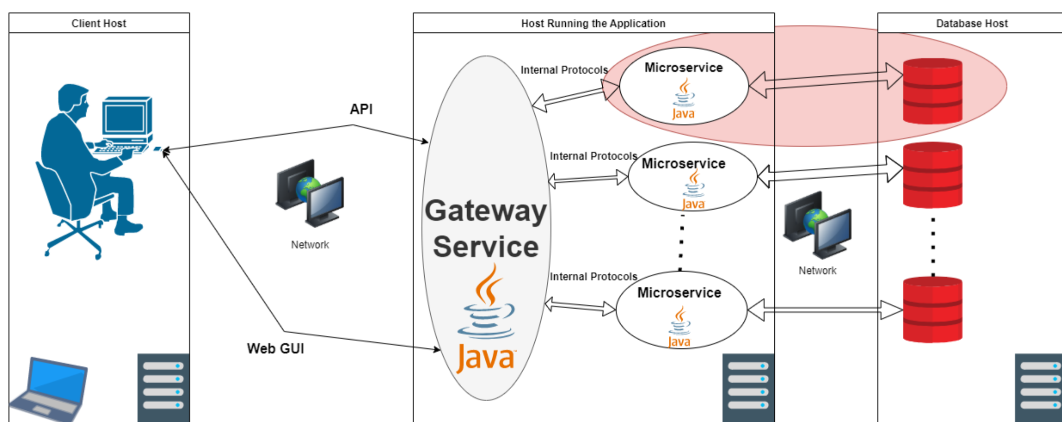


Figure 2.1: A simplified view of the system architecture. The area that this thesis focuses on is marked by a red ellipse.

The application we are working with is built up by several micro-services, it is one of these micros-services in particular that is investigated in this thesis. The product is a software solution used by many customers worldwide but the application is in most cases hosted at the customers' hardware- and network infrastructure. Generally the service is not stored on the cloud, although there is a possibility that the product will be moved to the cloud in the future.

There are also different ways to interact with the product, for example some customers use an API and others use a web-browser GUI. Since this report, focuses on the startup process of the back-end we are not concerned with data loading between the user and the micro-service and therefore we do not provide details on how the customer interacts with the application. The part of the codebase and architecture that we will be investigating is marked by the red ellipse in figure 2.1.

The code in the repository is written in Java and much of the code, especially the code that we are working on, is focused on database operations. For the database operations, the company uses a framework called Hibernate which is described in detail in section 2.2.3. The database management system used for this product is Oracle Database. Many of the services have their own centralized database where most of the data they access is stored. However, the services do sometimes access data tables located in other databases. All of these databases are running on one host, while the services are running on another, as can be seen in figure 2.1. From the perspective of our master thesis, we focus on loading of data from one database into one service.

An issue with today's solution is that loading the data from the database into the program can take a lot of time when there is a large amount of data. This is specifically at startup and restart when everything needs to be loaded. Many of the services have implemented caches in order for any loaded objects not to be loaded again during program execution. These caches are loaded at startup which takes a considerable amount of time. Some of the database tables can contain millions of rows and if all services need to restart and load all of the data from several of these big tables into its cache, the startup process can take a long time. To limit the scope we will be focusing on the initial load process in the micro-service previously mentioned, but it is still important to understand the bigger picture. Loading a table which takes 30 seconds may not be an issue on its own, but loading multiple tables for multiple services for 30 seconds could result in unacceptable startup time.

2.1.1 Research Questions

Based on the presented scenario we arrive at the research questions given below. In RQ1 we analyze the given problem using debug tools and a statistics API attempting to find causes of issues that can be mitigated. RQ1 provides needed insight into the issues present in the current solution. This information is used as input to RQ2 where we research tools and methods that can be used to manage and alleviate the identified issues. In RQ3 the different potential solutions that we identified in RQ2 are compared by running experiments.

RQ1. What are some of the major causes of inefficiencies in data loading for the case company?

RQ2. Which strategies can be used to alleviate the issues in data loading identified in RQ1?

RQ3. How do different loading strategies compare in terms of efficiency and maintainability?

2.2 Object-Relational Mapping

Relational database management systems (RDBMS) have been around since the 70s [2]. Even though newer technologies, such as non-relational databases (NoSQL), have grown in popularity, RDBMS are still actively used today. The reason for this is that RDBMS has been proven and used in applications built with enormous investments behind it, especially historically. RDBMS also has an understandable format and consistency properties that most newer database systems do not have [2].

The reason databases are used in the first place is that data needs to be persistent, and the persisted data needs to be managed efficiently. Data would have little use if it was erased each time a system shutdown. In object-oriented applications, persistence is key to making objects outlive the processes that created them [4, p.3].

An issue with relational databases in combination with object-oriented programming languages is the mismatch between the paradigms way of storing data. This is called the Impedance Mismatch Problem and refers to that objects are built up by data, behavior, and inheritance, while relational databases are built up by tables and relations between them [55]. To mitigate this issue a technique called object-relational mapping (ORM) can be used [49].

An ORM creates a persistence layer, between the database and the applications, that solves this mismatch problem for developers. The layer can populate the application with objects based on the data from the database, and also gives the developer the tools to persist the objects in an object-oriented way. This way the developer does not need to interact directly with the database through SQL, instead the database operations are performed indirectly by interacting with an ORM framework in the programming language of the application. This also comes with the advantage that it is easier to switch database systems if needed as the logic of communicating with the database is in the persistence layer [55].

There could be downsides to the use of ORMs. As with almost all higher-level tools they are usually not optimal for pure performance compared to other lower-level frameworks, such as JDBC in this case [20]. Adding a layer naturally leads to adding overhead. Additionally, much of the details are hidden by the higher level of abstractions and this can make it harder for the developer to write performance-optimized code. Other issues and anti-patterns could arise which in return could impact the performance of both the database and the application [49].

2.2.1 Data Access Object

In object-oriented software, it is a good practice to separate responsibilities between objects by adding layers, for example, the persistence layer that has been mentioned above. There are many well-known software patterns that can be used to achieve this separation. One such pattern that is widely used is the Data Access Object (DAO) pattern. The DAO pattern separates the business logic with the logic needed to access the data store for persistence. With these layers, the application becomes more modular and simpler to maintain. The pattern can be described as follows. For each Java object that needs persistence, a DAO interface with needed Create, Read, Update, Delete (CRUD) and find operations is created. Then implementations of this interface can be written depending on what database management system is used and how the communication should be done. This means that the type of

database and/or communication to the database can be changed without the business layer or the underlying Java object having to be changed [32].

2.2.2 Java Persistence API

Object-relational mapping is just a concept in itself. ORM frameworks need to be used to apply this concept in applications. For Java there are some frameworks available, for example, Hibernate, which is open source, and Oracle's TopLink [25][44]. Each of these frameworks have its own features and ways to obtain ORM functionality. However, the community has created a standardization called Java Persistence API (JPA) [43]. JPA is an interface that makes an application independent of the ORM tool used, so if a developer uses the JPA interface, the work needed to switch between ORM frameworks like Hibernate and TopLink is minimal [8].

The JPA specifies methods and collections that are to be used, and the frameworks have their own implementations of these. To map Plain Old Java Objects to the relational database annotations or XML files are used. The annotations give the ORM tool the necessary metadata for the transformation between the domain models [5][16].

2.2.3 Hibernate

As mentioned there are frameworks that are used for the concept of Object-relational mapping. Hibernate is one of the most popular frameworks for ORM in Java [25]. Hibernate is an open-source project that was founded by Gavin King in 2002 and has today grown to be more than just an ORM as it has tools for other types of domain models [25][39].

Hibernate uses metadata for the transformation between the different domain models. The metadata is provided by the developer by using annotations in code. As mentioned Hibernate has support for the JPA standard, but it also has its own native API if the developer prefers it. If the native API is used, the applications are more locked to Hibernate, as it will be more complex to switch to another ORM tool. Hibernate's native API also allows the developer to use raw SQL queries if needed [5].

Hibernate uses Java Database Connectivity API (JDBC) and other existing Java APIs to create the persistence layer. JDBC is the API used to connect to the relational databases, which means that every database that has a JDBC driver is supported with Hibernate. As mentioned before the persistence layer itself creates the ability to switch databases in a less complicated way [55].

In Hibernate, queries can be expressed in a programmatic way using criteria. The developer can build up the criteria by choosing which table or tables to target, and optionally add other functionality like order by and filters. There is also a way to only load a specific set of columns from a table, which is called a projection. As a standard, this will return an array of objects, where one object is a column [24]. However, a functionality called result transformer can be used to map the columns to an object. The standard is to map it to a so-called Data Transfer Object (DTO). The DTO stores the returning values by mapping the column specified in the projection to its own fields. This will limit the use of typecasting and let the code move around data in a more standardized way [48].

Hibernate works with something called sessions which represents an abstract unit of

work. A session is similar to a database transaction but a session can span multiple transactions [22].

2.2.4 Hibernate Built-In Caches

Hibernate has built-in functionality for caching data. These caches can be split into the primary cache and the secondary (or second-level) cache. The primary cache is connected to a Hibernate session and is mandatory, it cannot be deactivated. Hibernate sessions are often opened and closed with transactions. As transactions are short-lived the primary cache does not add that much performance boost, it is more used to keep the session and the database synchronized [29].

The secondary cache is not mandatory and has to be activated by the developer. The cache requires a third-party implementation to work, for example, EhCache. This cache is not connected to a single session, instead, it caches for the whole application process. Therefore the cached data is not deleted when a session closes, it lives on together with the program. If the second-level cache is used, a session will first check its primary cache. If the data is not in that cache it will check the secondary cache before having to request data from the database. The second-level cache can therefore improve the performance of the program with a decrease of the amount of calls to the database [29].

To access the entities stored in the second level cache the developer needs to access that entity directly by using its id [23]. Therefore, it will not work directly on queries. There is however an option that can be activated as well alongside the secondary cache that caches queries. This works well if the same queries are executed repeatedly, or if the data is not changed. However, the overhead of using this in other scenarios can impact the performance of the application negatively, and therefore it is deactivated by default, and should not be used for most applications [23] [29].

2.2.5 The N + 1 Problem

Hibernate and other ORM frameworks help the programmer immensely since it handles many aspects of SQL querying and mapping of data to Java objects. But as already mentioned there are some drawbacks of higher-level ORM frameworks. It should be noted however that Hibernate allows different ways of making requests to the database. If we want to specify a native SQL query, to avoid redundant extra queries, that is completely possible. Nonetheless, when working with Hibernate we aim to simplify the interface to the persistence layer, and therefore one usually does not write native SQL unless problems with ordinary Hibernate queries have been identified. As an example, a common problem that can arise in this case is the so-called $N + 1$ problem [28][4, p. 286–289]. We will illustrate the $N + 1$ problem with an example.

Consider a scenario with a one-to-many relationship. For example, we can have one table for universities and another for students. For the sake of the example, we assume that a student can only attend one university at a time but every university can of course have many students. The number of universities is denoted by N. When storing the tables in a database, this would normally result in each student having a column, acting as a foreign key, with the university name.

Now let us say that we at some point are interested in printing out all students for every university. In Hibernate code this could look something like shown below.

```
List<University> universities = session.createQuery("From University",
    University.class).getResultList();
for (University university : universities) {
   List<Students> student = university.getStudents();
   for (Student student : students) {
      System.out.println(student);
   }
}
```

In this scenario, Hibernate will fire one query to the database to get the list of universities. But then for each of the N universities, there will be one query collecting the list of students at that university. Hence this is called the $N+1$ problem. Note that if we would have written native SQL queries it would have made no sense to make this many queries. Instead, one query joining the two tables would have been sufficient and much faster.

It is possible to solve the $N + 1$ problem in Hibernate in many ways. For example, the first line in the previous code example can just be switched to the code in the listing below. One can easily check that this solves the issue by using Hibernate's built-in statistics tools for analysis, see 2.2.6. Since we can solve the problem it is not a deal-breaker, but it is something that is easily missed.

```
List<University> universities = session.createQuery("From University u
    JOIN fetch u.students", University.class).getResultList();
```

The previously mentioned DAO pattern is another layer of abstraction on top of Hibernate. Again this means that our objects are more conveniently used but the cost is that problems such as the $N + 1$ problem could be harder to identify.

2.2.6 Hibernate Statistics API

In every subcategory of computer science, metrics and statistics are essential for developing any product that runs efficiently. To quote one of the most well-known statements in computer science made famous by Donald Knuth: "Premature optimization is the root of all evil (or at least most of it) in programming" [33]. Premature optimization leads to spending time on improvements that in the end have very little to no impact. In a scientific report, tools and metrics can be even more important since it is needed to back up any conclusions.

Hibernate provides a built-in statistics API to collect metrics on the database operations performed. This can be information such as the number of JDBC statements executed and the time spent on a particular query. These statistics can be used to analyze performance and avoid issues such as the $N + 1$ problem.

2.3 Incremental Loading

Incremental loading, also called delta loading or incremental querying, is a term that is used to describe the concept of loading data in smaller batches in intervals. This is compared to a

full load where we would load everything at once in a big query [7].

The purpose of using the concept of incremental loading is that a full load of all data makes the application unresponsive for a long time while the data is fetched. When loading data incrementally the user can access the data fetched so far while new data can be fetched with only a small latency for each query. We should point out that incremental loading will lead to a larger number of queries. This likely leads to a longer execution time for the complete loading process than if we would load everything in one big query [4, p. 14]. Still, splitting up the loading process into several queries, each with short latency, can be less disturbing to the user.

2.3.1 Change Data Capture

Incremental loading is often used in Extract Transform Load (ETL) systems. ETL can be summarized as a way of loading data from multiple heterogeneous data sources such as databases into one big data warehouse. Incremental loading is in ETL systems used as a method of propagating changes from one database to another. This way it is not needed to load all data from the source when updating the target, we only need to load any new changes since the last time we synchronized the databases [7][30].

Change Data Capture (CDC) is the terminology used for the method of capturing the differences in the source and target data [7], it is these differences that will be propagated from one database to the other. There are many different implementations of CDC. One method is to use a log of changes since the last load. This can be either the DBMS internal logs or logs explicitly managed by the developers. Another way is to use a separate column with a watermark such as a timestamp or an incremented number to identify when the row was last updated. Lastly, you can use triggers to log or mark the rows that were changed since the last load [7].

In this thesis, we are not working with ETL. We are moving data into the program memory as Java objects, instead of moving data from databases or other sources into another database which is typical in an ETL process. Another point of difference is that when talking about CDC the two sources of data are usually mostly synced, and the usually relatively small differences will be captured and then completely fetched to the other data source [30]. In our case, we are investigating initial loading which means that we will start from zero data in our program memory. The similarities are that we are still working with two sources of data and we could split up the initial loading process into fetching data in smaller batches. In that case, we could use methods of CDC to detect differences between source and target.

2.3.2 Lazy Loading

ORM frameworks like Hibernate allow the programmer to specify different strategies regarding when to load data from the database. As the frameworks make up a connection between an object and its representation in the database the question is when the framework should fetch the data to populate the object. The two main strategies can be split up into eager and lazy. The eager strategy fetches all the data directly. This could cause issues if the amount of data is huge. The load will take time, and the application might also load a lot of data it will never use. However, the number of queries will be reduced. The opposite of eager is the lazy strategy. With this strategy, the application only fetches the data when it is used.

This means that the use of an object might lead to an execution of a query unless the object has been loaded from the database before and cached. Because of this lazy loading can lead to the previously described $N + 1$ problem [54]. As eager loading loads all objects directly, and does so with larger but fewer queries, the network round trip time is reduced compared to fetching one object at a time. On the other hand, lazy loading decreases network traffic by not fetching data that is never used [10].

It should be pointed out that there are many strategies that are a combination of eager and lazy fetching. For example, the default strategy in Hibernate is to fetch the object requested eagerly, directly upon request, but it uses lazy fetching for any collections or associations to other tables that are members of the object [18].

Since the application does not load everything at once, but a little at a time only when asked for, lazy loading is one way to achieve incremental loading. The startup time will be significantly reduced, but the latency of loading from the database comes during run-time instead when the application asks for data that has not been cached. Eager loading could avoid this latency as the data has already been fetched when asked for.

Hibernate Proxies

Dynamic proxy classes are a feature in Java and this type of classes implements a list of interfaces specified at run-time. When there is an invocation of a method on the proxy object it will be sent to the real object that the proxy class implemented the interface from. If a proxy is created using the real class User, as an example, the following statement will return true: *proxy instanceof User*. It will look like the proxy is the real object, and this can be used by APIs and frameworks [41].

The way that Hibernate implements lazy loading is by the use of proxies. As mentioned Hibernate by default uses lazy loading for associations to other tables by an entity. An entity can also be loaded as a proxy, not just the associations. To keep track of whether the associate entity has been loaded or not a proxy object is used. If a function call is made on the proxy, it will have to communicate with the real object. To be able to do this Hibernate needs to retrieve that object's data from the database. However, to do this conversion the application needs to be connected to a Hibernate session, or it will throw an exception. Therefore, the developer needs to understand when and where it can "unproxy" a proxy. To reattach a proxy to a Hibernate session, as Hibernate sessions should be closed when leaving the persistence-layer, locks with lock mode *NONE* can be used [26].

In figure 2.2 we can see what a proxy representation would look like for the class User. The proxy works like an interface between the method calls and the real object, while also knowing what to retrieve from the database when it is time to load the object data. If the proxy has not converted yet and the proxy gets a method call, like in figure 2.2, the proxy will communicate with the database to load the object data. After this load, the proxy can be converted into the real entity.

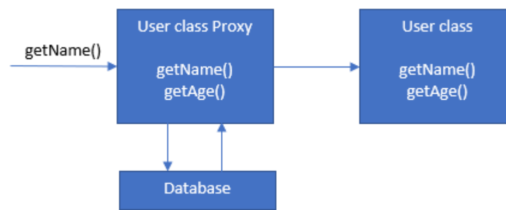


Figure 2.2: A figure representing a Hibernate proxy

Caching Strategy

Lazy loading is a strategy that caches can use to load if and only if the data is going to be used [34]. This could be done without ORM-frameworks. Instead, the cache itself could keep track of whether it has loaded or not. The way to achieve this is to set up the cache so that it can check if it has cached the data the application asks for. If the requested data is in the cache we have a cache hit and can simply return it. However, when the requested data is not in the cache we get a cache miss. Then the cache needs to load the requested data from the database, insert it into its internal structure and return it to the application. This way, only data that has been asked for by the application are loaded and stored by the cache, in other words the cache is lazy. The figure 2.3 shows how the cache works in a simple way by illustrating cache hits and misses.

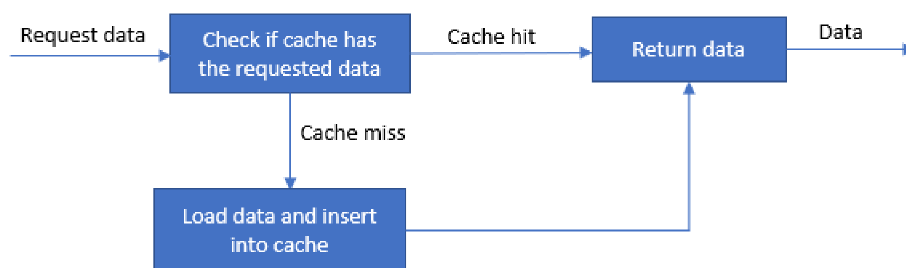


Figure 2.3: A figure showing cache hits and cache misses

2.3.3 Pagination

A third way to split up the full load process into smaller loads is to have explicit pagination of the database tables. Pagination means that we would split up the table into pages. We can split the table up based on any column that we can order by such as id, timestamp, etc. The user will explicitly have to request data from a certain page. This will decrease the size of the queries and should definitely decrease startup time [9]. But pagination only works in some circumstances where the user is fine with only using parts of the table at any moment. If the user always needs to access the full table pagination is not possible.

2.4 Parallelizing Database Queries

In many areas of computer science, parallelism is a commonly used approach to enhance performance. When parallelizing a sequential algorithm the level of concurrency needs to be decided and the potential of using a parallelized implementation depends on the task at hand. One limit of concurrency is the number of hardware threads available at the machine running the task. Another limit is the nature of the tasks at hand, i.e. how can we split the task up into several sub-tasks running in parallel? This partitioning is commonly known as decomposition [15]. If these sub-tasks transfer data over the network, for example database queries, the network will be another limiting factor. Additionally, dependencies between tasks limit how big fractions of a program can run in parallel and how much overhead will be spent on synchronization between threads. Amdahl's law is a well-known theory that states that the gain in performance is limited to how large a fraction of the program we can parallelize [1].

2.4.1 Parallel Query Processing

The DBMS used is responsible for transforming high-level queries into execution plans that can be efficiently executed. Many DBMS provides methods to exploit parallelism. We can divide the query processing parallelism into two forms: inter-query parallelism and intra-query parallelism. Inter-query parallelism means that we can run several queries in parallel while intra-query parallelism refers to dividing one query into several subtasks which can run in parallel in the query execution plan [46].

As mentioned in 2.1 the case company is using Oracle DB as their DBMS. Oracle DB supports multiple users accessing the database at the same time, i.e. inter-query parallelism [40]. Intra-thread concurrency is also supported [42].

2.4.2 Parallelism in Hibernate

As always when parallelizing a program it is important to consider thread-safety both in the application source code and in all third-party libraries used. Hibernate supports multi-threaded applications but certain rules must be followed [22].

A Hibernate session factory is thread-safe while a session is not. Therefore it is important to create a new session for every thread. When working with lazy fetching strategies extra care must be used. Each proxy saves a reference to the session used when the proxy was created. Therefore, if a proxy is created by one thread and later on initialized by another thread you will have to reattach the proxy to a session owned by the thread initializing the proxy [22].

By running Hibernate queries in parallel instead of running them sequentially hardware and network resources can be utilized to a larger extent. Inter-query parallelism can also be exploited at the database. Finally, any operations in the Hibernate framework or the application source code that is interacting with Hibernate may also benefit from parallelism.

2.5 Related Work

This section summarizes a couple of sources from previous research that are relevant to the scenario investigated in this master thesis. There are many research papers investigating database performance and possible improvements. In fact, so much that through the course of our master thesis work we have several times felt that it can be quite hard to navigate through all the literature to find the scenarios and methods that are most relevant to us.

Several research papers such as [37], [36] and [35] provide guidelines for efficient SQL querying, both from the perspective of a database administrator and from the perspective of a developer. In this report, an ORM framework is used which abstracts away parts of the lower-level functionality compared to pure SQL. However, since the higher-level frameworks like Hibernate are dependent on lower-level frameworks using SQL it can still be necessary to understand the efficiency guidelines of the lower level-frameworks to achieve high performance. In [13] Colley and Steiner describe how previous well-known SQL tuning guidelines relate to performance tuning when using ORM frameworks.

There is also a lot of previous work studying performance in Hibernate in particular. We have used the book in [4] made by Bauer and King as a foundation for general knowledge about the Hibernate framework. Hibernate also provides a lot of online documentation with guidelines on how to efficiently use the framework [18]. In [20], the performance of Hibernate is compared to its lower-level counterpart JDBC.

As mentioned in section 2.3.1 there has been previous research, such as [31][30][7], on how to limit the size of data transferred between data sources in ETL systems with Incremental loading. We have not found any papers which use CDC together with Hibernate although the concept of loading smaller chunks of data instead of a full load is present in the research papers and online documentation about lazy loading and pagination.

Lazy loading is common practice in Hibernate and many other frameworks. As such, there is much research on how to use it most efficiently and how it compares to eagerly loading data. [10] and [54] are both examples of such papers.

There is a lot of previous research regarding parallelism in database systems but most of them are focused on parallelism from the perspective of the DBMS engine, for example [11] and [51]. In other words, researching how DBMS engines can handle multiple requests at once and how they can split up one query into several parallel parts. We have not found detailed research regarding performance benefits due to parallelizing a number of database queries rather than running them sequentially from the perspective of the application. However, the Hibernate documentation and community forums indicates that this is an established strategy [22].

In [56] Zhou et al propose a technique of preloading a database cache with multithreading. In their scenario, they are focusing on simultaneous multithreading and parallelism at the DBMS engine but a similar technique can be applicable in our scenario as well. Kohler and Specht investigated database cache loading in [34] and how it can be enhanced by using lazy fetching strategies and parallelization techniques.

The previous research conducted in our area that we have just mentioned will be used as a foundation to provide a solution to today's problem when answering RQ2. Exactly which of the methods will be used and how it relates to previous research depends on the answer to RQ1 and is discussed further on in the report.

2.6 Contribution

2.6.1 Scientific Contribution

By conducting this research and attempting to solve the issue present at the case company we provide guidelines for others to do the same. In particular, the methods we propose in RQ2 should be possible to utilize for any other organization loading large amounts of data from a database into their application data structures at startup.

Compared to the previous research discussed in 2.5 we aim to provide a relatively direct approach from the perspective of a software developer. Many of the papers we investigated are only focused on the theory concerning small sub-parts rather than the whole process of loading data to an application. Additionally, many of these papers describe how something works, from a highly theoretical perspective, rather than what the developer can actually do about it in practice. In this master thesis, we investigate implementations at a real company and we have to practically integrate our solutions with the other parts of the codebase.

2.6.2 Distribution of Work

As per requirement at our university, we need to describe in a clear manner who did what in our master thesis work. We have been working very closely together with communication on each section. Still, we have divided the work between us so that one of us is mainly responsible for each section. There are a few exceptions where both of us has contributed equally much. In table 2.1 we outline who is mainly responsible for each section.

Section	Ivar Henckel	David Söderberg
Introduction		X
Problem Statement	X	
Object-Relational Mapping	X	X
Incremental Loading	X	X
Parallelizing Database Queries	X	
Related Work	X	
Contribution	X	
Method	X	
System Overview		X
Performance Measurement in Java		X
Experiment		X
Maintainability		X
RQ1 - Identifying Inefficiencies	X	
RQ2 - Proposed Solutions	X	X
RQ3 - Gathering Experiment Data		X
Validity Threats	X	
RQ1 - Interpreting Inefficiencies	X	
RQ2 - Dissecting Implementations	X	X
RQ3 - Comparing Solutions	X	X
Future Work	X	
Conclusion		X
SQL Script to Generate Data		X
Hibernate Statistics Command	X	
Mutual Code		X
Hibernate Proxy Cache		X
Lookup Table Cache		X
Asynchronous Preloading Cache	X	
Experiment Code		X

Table 2.1: Distribution of work. The person mainly responsible for a section is marked with an X. If both of us are approximately equally responsible both persons are marked with an X.

Chapter 3
Approach

3.1 Method

The methodology used for this thesis is *action research* [17]. The action research process can be split into two stages, examination and practical experiments. For the first stage, the researchers analyze the problem situation to formulate theories that could be used to solve the issue. These theories are then used in the second stage where the practical attempts are done. Any effects caused by changes in the implementation, based on the theories, are then evaluated. The result of the evaluation is then analyzed, and if the problem is still there, or other issues have emerged, the process will restart at the first stage. This means that action research is an iterative process.

Action research is a common methodology when working with research of a problem-solving nature. As software engineering is rooted in problem-solving this method of working is often suitable in computer science research [3]. A significant benefit of using action research compared to other scientific methodologies is the flexibility gained due to the iterative approach. When attempting to solve a problem, the circumstances related to the issue could change as one progresses on the path to solving the problem. The understanding of the issue will most likely increase as one implements and evaluates various solutions. Additionally, when new solutions are implemented other problems can arise [17].

The steps in action research can be related to our research questions. The first part of action research examines the problem at hand and identifies which methods can be used to solve the problem. We have divided this part into RQ1 and RQ2 where answering RQ1 provides detailed information about the issue we are trying to solve and RQ2 identifies different approaches on how the issues can be managed and solved. The second part of action research is to use experiments to evaluate the implementations from the previous steps, this is done when answering RQ3. As expected we answer these three questions with an iterative workflow. When running the experiments to answer RQ3 we may get more information on what the underlying issue is, leading to improvements in RQ1. Although we will be working with

the questions in an iterative manner, we will not present our iterations explicitly unless it is needed to clarify the problem-solving process.

When working with this methodology, and problem-solving in general, it is often difficult to keep discussions and conclusions unbiased since the researchers are evaluating solutions that they themselves implemented. Therefore as we make progress through this report we regularly let our supervisors at the university and at the case company review any implementations, experiments or discussions we make.

3.2 System Overview

3.2.1 Test Environment

To run the experiments a test environment provided by Nasdaq is used. These test nodes (as they are called by the case company) are set up with a customer-specific configuration to get an environment close to what it would look like in production. Each node is connected with its own database, which has some initial data. The provided data was not sufficient to replicate the quantities of data a production environment would have. Therefore, we had to generate data to get a more realistic view of how large the quantities could be in production, see 3.2.4.

The environments are running with the help of VMWare Virtual Platform with RedHat 7.9 x86_64 as operative system. It uses a 4x Intel(R) Xeon(R) Gold 6230R @ 2.10GHz CPU with Java version 11 and Oracle Database version 19.0.0.

3.2.2 Application

The test environments have been configured to be able to easily install, update and run the application. The application in itself uses a micro-service architecture. These services run by themselves and can be shut down and restarted without the need of restarting the whole system. When a service starts it will first go into the state of "running". However, it is fully ready when it reaches the state of "server", the server will not answer any requests until this state has been reached. Between running and server is where the initial load of all the data takes place in the initial solution. The execution time of this initial load is one of the time measurements that will be looked at.

The services also have commands that can be executed via the command line and this is one way to communicate with a service. By using these commands we can tell a service to do certain steps, for example, reload the caches. Such commands are used to start the experiments described later on in section 4.1 and section 3.4.

3.2.3 Cache Service

To get a better understanding of the proposed solutions, given for RQ2 in section 4.2, the cache service needs to be explained in more detail. The cache service is currently built up by four different in-memory caches, one for each table that we want to cache. In the original reference solution, each of these caches are initialized with all the data from the corresponding table at startup of the service. This is called a full initial load as mentioned before. This

original cache implementation is referred to as the *Full Load Cache* throughout this thesis. Records in the caches can be added, removed, and updated during run-time. This means that the application will by itself make sure the caches are up to date with the latest changes. The cache service needs to be thread-safe as multiple threads could request, add, remove or update data.

There are also three ways to retrieve objects from the cache by the application. Either the application asks for all the records, records from a specific account, or just one record based on an unique id number. We will name these ways of retrieving data *getAll* , *getAccount* , and *get* respectively and these names will be used throughout the report.

To keep track of the data two data structures are used for each of the caches. The code snippet below shows the two structures. The first called *cache* is the main structure and holds all the entities. These can be found by their unique id number. The second structure *accountIndexMap* is used to map the entities to a specific account. The class *IndexKey* is used to map an account to a unique object. The class *AccountIndex* stores all the data related to a specific account.

```
private final Map<Long, Entity> cache;
private final Map<IndexKey, AccountIndex> accountIndexMap;
```

3.2.4 Generating Test Data

Test data was generated to fill the tables used during the experiment phase. A SQL script was developed to generate rows in the table that resembled production data, see A.1. Note that the script has been masked to protect the internal system of the case company. Some of the columns values would have more variety in a production environment. The script has options regarding which table to insert values in, amount of rows to insert, and deleting all generated rows if needed. This is to make it easy to test different sizes of data sets during the experiments.

Through discussions with our supervisor at the case company, we have determined that up to around 750 000 rows in each table is a reasonable size to test with. Using sizes up to 750 000 rows makes the issue apparent and at the same time it is not just an extreme corner case. Some customers, but very few, have much larger databases. In our results we will however experiment with tables of different sizes. Some solutions may be better at handling larger tables while others are better at handling smaller tables.

3.3 Performance Measurement in Java

To run performance measurements in Java there are some things to consider to get more accurate results. There are various factors that can affect each experiment, the Java Virtual Machine (JVM), the garbage collator, the size of the heap, etc. Java is also not deterministic at run-time, which means a Java program differs from run to run. The Just-In-Time (JIT) compiler is an example of optimizations that occur at run-time, but the JVM also makes optimizations that will affect the measurements [19].

To get more accurate experiments these issues must be kept in mind to get the best measurements possible. First of all the program needs to "warm-up" by calling the method you

want to benchmark a couple of times before the real experiments begin. There are plenty of reasons for this, for example the optimizations by the JIT compiler and that Java classes are loaded on demand, not when declared. However, the initial performance at the first run could also be an interesting factor to look at, as this is something the Java program will have to do in production as well [27].

The JIT compiler does not only create an issue with its gradual optimization of the program. The optimizations it does could defeat the purpose of the experiment. Dead code elimination is an example of an optimization that would do this. Dead code elimination could occur when the returned object from a method call is not used, and therefore an optimization is to eliminate the method call from the code. It could also lead to optimizations where the method call is moved outside of the loop it is located in, constant propagation for example. Therefore, we need to make sure the objects returned are used in such a way that the JVM or JIT does not consider the method calls dead code or the objects as constants [14][27].

As mentioned the garbage collector could also affect the execution time while running experiments. Triggering collections between runs could lower the cost of garbage collection while running the experiment. However, letting the garbage collector trigger automatically as usual could result in a more realistic result. Here the setup of the experiment needs to be considered to get the most realistic experiment possible [19][27].

To measure time accurately *System.nanoTime* can be used and the behavior of this method is dependent on the hardware. Linux uses the CLOCK_MONOTONIC system, which has high precision when you use System.nanoTime [27]. The experimental setup that will be used is RedHat's operative system based on Linux, so *System.nanoTime* will be a good time measurement function for the experiments.

3.4 Experiment

The experiments that are conducted will help to get a better understanding of the solutions to be able to answer RQ3. Based on the proposed solutions from RQ2, we decided to run two different parts of the experiment. The first part is to understand how much an implementation affects startup time. This is important as startup time was one main issue of the initial solution. The second part of the experiment will be done during run-time. The different implementations will act differently during run-time which means the measurement of the startup time is not enough to compare the solutions. For example lazy loading, as described in 2.3.2, will move latency from startup to run-time. Therefore, experiments based on different scenarios during run-time need to be tested to get a fair comparison of how the implementations affect the program as a whole.

To measure how much the implementation impacts the startup time of the service the log file for the service can be used. The startup time is calculated using the timestamps in the log for the state changes of the service. This part is important as the proposed solutions and the original solution differs in what has to be done at startup, and a slow startup time is the main issue for the Full Load Cache when the data in the database is large. The startup time will differ depending on the size of the underlying database. Therefore we will conduct three startup experiments with 250k, 500k and 750k rows in each table.

The run-time scenarios cover all three methods used to retrieve data from the cache, *getAll*, *getAccount* and *get*. The experiment will be conducted using both 250k and 750k rows

in the tables to see the difference between the solutions depending on the size of the tables in the database.

The experiments for the method *getAll* are listed in table 3.1. All tests will be conducted on one database table as this is enough to measure the performance of the method. Experiments 1-3 test how the cache acts when none or some of the data has been previously accessed before requesting all the data for the first time. Test 4 measures the time when *getAll* is called for the second time. The reason for these experiments is that the proposed solutions will act differently regarding how much will be loaded from the database based on what has been previously accessed. We have decided to use 50% and 90% for test 2 and 3. 50% is used to show how the cache acts when we have an equal amount loaded and not loaded. 90% is used to show how the cache acts when almost all of the total data has been loaded into the cache. To get 50% and 90% of the data previously accessed for tests 2 and 3 the chosen fraction of the data is loaded with the help of *get* and the unique id numbers, at setup of the experiment. The numbers that will be used are randomized to mitigate any effect the JVM or JIT optimizations might have on the result.

1. Get all from one table, first time called
2. Get all from one table, first time called 50% previously accessed
3. Get all from one table, first time called 90% previously accessed
4. Get all from one table, second time called

Table 3.1: Experiment to test the functionality of *getAll*

The experiments for the method *getAccount* are listed in table 3.2. For each test it will be one method call per run using a randomized account. The reason for the randomization is to mitigate the impact the JVM and JIT have on the measured time. We want to run experiments 1 and 2 to see the difference between how the solutions compare if data has been previously accessed or not.

1. Get account from one table, account not accessed before
2. Get account from one table, account accessed before

Table 3.2: Experiment to test the functionality of *getAccount*

The experiments for the method *get* are listed in table 3.3. Both these tests will work in the same way as the ones listed for *getAccount*. Instead of randomizing an account, a valid unique id number will be randomized to be used as an argument in the method call. The reason behind the randomization is to mitigate the impact the JVM and JIT have on the measured time.

1. Get entity from one table, not accessed before
2. Get entity from one table, accessed before

Table 3.3: Experiment to test the functionality of *get*

In the pseudo-code below we can see the setup of how all the experiment scenarios will run. For the whole experimental code, see A.7.

```
// 2 runs warm up + 10 runs experiment
for(12 runs):
    Reset and setup cache (+ trigger garbage collection)
    Start time
    Run experiment
    Stop time
    Save the execution time
    Use the returned data from the experiment
```

The first two runs will be to warm up the JVM, then each experiment will run 10 times more. Between each run, there will be a reset and setup of the cache to prepare for the experiment. We will trigger garbage collection after the reset of the cache so that the data that has been cleared from the previous run can be collected. This will create a more realistic scenario, as the cleared data would not be in the heap in production. To not get dead code elimination or other optimizations that would impact our experiment in a negative way we will make sure to use the returned data after we have stopped the time measurement.

As the JVM and the JIT compiler are non-deterministic, the whole experiment suite will run three times, with a new invocation of the JVM each time. This will give a more accurate total result. All in all each experiment will run 36 times including the warm-up runs. With all measurements, we can calculate an average time per experiment. For the result of the experiments, see 4.3.

3.5 Maintainability

Even though a solution is the most efficient it might not be the best to actively use depending on the maintainability. Lower execution time can lead to satisfied customers in the short term. But a higher level of complexity and a lower level of maintainability will lead to costs in development and slower improvements in the future. Therefore, from a business perspective, the gain in performance always has to be related to the cost of development.

Additionally, even though the original implementation and the solutions that we present in 4.2 do not have a very high number of lines of code, the cache only serves as one component in a bigger software project. In other words, if all smaller components have a high percentual increase in lines of code the complete project will get a lot harder to maintain.

To answer this part of RQ3 we will calculate the number of lines of code. Lines of code can be used as a predictor of the effort to maintain the code, because of the simple principle that more code means more work and more bugs [47]. What should be pointed out though is that lines of code itself is not an accurate metric of maintainability and should preferably be used together with other metrics [47]. However, we use this metric as a quick overview and not for a detailed comparison. To add depth to the discussion on maintainability we will also consider the knowledge needed to understand each solution.

The number of lines of code can be calculated in different ways. We have chosen to include all the classes and methods that are specific to the solution. Rows that are blank and comments will not be counted. We will also not count rows that have to do with the logging,

as they are not part of the functionality of the solutions. The number of import- and package statements are not counted either since we do not want to disclose the internal frameworks used. The same code style has been enforced to all solutions, which makes them comparable.

3.6 Testing

To test the proposed solutions we implemented commands that could be executed from the command line. These commands were executed manually and can interact with the cache while the application is running. No automatic tests have been developed for this thesis, and we leave that for future work if any of the caches get integrated into production.

All of the three types of get requests, described in 3.2.3 have been tested individually as well as requests for updating, removing and adding one entity to the cache. These requests were also tested in different scenarios. All of the requests were tested both when requesting data that did and did not exist in the database. Additionally, the get requests were tested both when data had and when data had not already been cached.

For the proposed solution described in 4.2.3 some additional tests were run. This solution is based on separate preloading threads loading data into the cache and the cache will act differently depending on if these threads are finished or not. Because of this, we tested the behavior described in the last paragraph both when these preloading threads were finished and when they were not. To simulate the threads not being finished a *Thread.sleep(long)* was added manually to the code.

Chapter 4
Result

4.1 RQ1 - Identifying Inefficiencies

In this section, we attempt to answer RQ1. We collect different metrics to understand the issue that is causing slow startup time at the case company. This investigation is started by defining a suitable scope.

4.1.1 Scope

As always when analyzing performance issues it is important to define a suitable scope. If only a very small part of the codebase and hardware is analyzed there will be a high risk that we fail to see the bigger picture. Because of this, we may spend time researching and improving something that is not actually a major issue compared to other performance bottlenecks. At the same time many technology solutions, such as the one we are working with, are made up of large codebases and many different types of hardware. In this case, it is also important to use a suitable approach to narrow down the search for performance issues. Otherwise, every small detail will have to be analyzed which would demand a much larger team of experts and the budget to pay for them. Because of this, we begin by collecting more general and higher-level statistics to narrow down our search for the issue and then go more and more into detail.

There are usually many aspects of a large software project that can benefit from optimization. But for this thesis we limit the scope to cache response time latency during startup and run-time, since that is something that the case company considers a priority.

An application communicating with a database over a network could have a slow startup process due to inefficient network- or hardware capacity. In that case, the issue could perhaps be solved by just buying new hardware. Of course, one could not expect that the newest and most expensive equipment is always being used. Still, we can expect that hardware- and network infrastructure is set up to match business requirements to a reasonable extent.

As mentioned in section 2.1 Nasdaq is only hosting the service with network and hardware to a small extent, most of the customers have their own setup. Therefore it is not a feasible solution, in this case, to just buy a more expensive network infrastructure. If we instead solve the issue with software the solution would automatically be propagated to all customers.

Since increasing network bandwidth is not a preferred solution, and in most cases handled by the customer, we do not investigate network issues further. Instead, we concentrate our efforts towards optimizing software or modifying the software in such a way that less data needs to be loaded.

4.1.2 Overview of Startup Execution Time

The application that we are working with uses logs where one can debug and analyze the order of when different parts of the code base execute. Since this log is timestamped we can use it to get a first overview of how the execution time is distributed on different parts of the code. In table 4.1 we present time measurements from this log when running with and without the cache enabled. We are looking at the full time for the startup process to finish and the time for the cache to be initialized by loading all the data it needs. In this case, we used four tables of 750 000 rows each.

Running with Cache	**Startup Execution Time**	**Load Cache Execution Time**
Disabled	80 s	0 s
Enabled	179 s	103 s

Table 4.1: Approximate results on startup execution time obtained by reading log files. All values are averaged over three executions.

Looking at table 4.1 we can clearly see that loading the cache takes up a large portion of the startup time. When the tables have this size, 58% of startup execution time is spent on loading the cache. This is enough to motivate further research on how to improve the performance regarding this. Knowing that the initial loading of the caches from the databases takes a considerable portion of the startup execution time we now want to go into detail about what is taking time.

4.1.3 Software Layers

Looking at the issue from a software perspective there are still several abstraction layers where the issue can be found. When the cache is loaded Hibernate is used to get the results from the database and to convert them into Java objects. These objects are then inserted into data structures used in the cache. There is also some logging in the cache.

When looking at the logs it is clear that there is extensive logging when the tables are this big. Logging is of course good and it is not reasonable to get rid of the logging completely just to decrease execution time. But in this particular case, there is one log message per row meaning that with $750\,000 \cdot 4 = 3\,000\,000$ rows this could take a considerable amount of time and make the log output hard to read. Additionally, the info logged in this case does not seem to be that useful when debugging.

By using Hibernates built-in Statistics API we can determine how much time is spent in the Hibernate framework. This can then be compared to the full execution time for the cache loading which also includes writing log messages and performing operations on data structures. We can also determine how many queries there are. As explained in section 3.2.2, it is possible to implement command line commands to communicate with the application. We implemented two such commands: one to reload the cache and another to present statistics from Hibernates Statistics API. An extract from the source code for the latter command can be seen in A.2.

In table 4.2 we present a summary of statistics when reloading the cache with all four tables. The real table names have been replaced in order to not disclose unnecessary confidential information. By using the Statistics API we can also see that there are no cache hits in Hibernates internal caches, a cache hit could otherwise have given false comparisons. Lastly, the size of the table is calculated with Oracle Database's *vsize* command which calculates the size for each entry of the tables, we have then calculated the sum of this value over all entries in each table.

Table Name	Queries Executed	Execution Time	Table Rows	Table Size
Table 1	1	18 512 ms	750 000	104 MB
Table 2	1	19 691 ms	750 000	106 MB
Table 3	1	18 415 ms	750 000	92 MB
Table 4	1	14 810 ms	750 000	98 MB
Sum over all tables	4	71 428 ms	3 000 000	400 MB

Table 4.2: Summary of statistics generated with Hibernate Statistics API. The query execution time is averaged over 3 executions.

We can make several observations regarding the table in 4.2. To begin with, there is only one query per table, which is a good thing since many queries cause a lot of overhead. The tables were built without any associations to other tables on purpose by the developers to prevent issues with one cache request resulting in many queries. This means that problems like the $N + 1$ problem, described in section 2.2.5, are not an issue in our case.

Another observation is that the sum of execution time for all tables is 71,428 seconds. Comparing this with 4.1 where we saw that 103 s were spent on loading the cache. We conclude that the fraction of time spent in the Hibernate framework firing the queries and delivering the response is $71s/103s = 70\%$. Note that this measurement is approximate but it is enough to consider the time spent on things outside of the Hibernate framework, like logging and data-structure operations, not as significant.

Our Hibernate Statistics command also outputs the query executed. In the code snippet in listing 4.1 we can see the query executed to fetch Table 4. The output is similar in structure for all the four tables with only a changed number of columns. The column names have been changed in order to not disclose confidential information. Note that all entities fulfill the condition in the *where* clause with our current setup and the data we generated. I.e. all of the rows are fetched to the application.

Listing 4.1: Query used to fetch Table 4.

```
select
    this_.column_1 as column_1_name,
    this_.column_2 as column_2_name,
    this_.column_3 as column_3_name,
    ...
    ...
    this_.column_24 as column_24_name
from table_4 this_
where 0='0' and this_.column_3=?
order by this_.column_7 asc, this_.column_6 desc
```

Looking at the size of the tables in table 4.2 we can see that we are working with quite large data tables and this is likely to be a major factor causing slow loading. To show the effect of the large query results we generated the graph in figure 4.1 that illustrates how the performance is affected when the query result size is changed. In this experiment, a limit clause was added to the query in listing 4.1 to limit the size of the result. Note that the table size is still the same, just that we are extracting results of different sizes. The execution time seems to increase linearly in relation to the number of rows fetched. This is a clear indicator that limiting the size of the data fetched to the cache is a good idea.

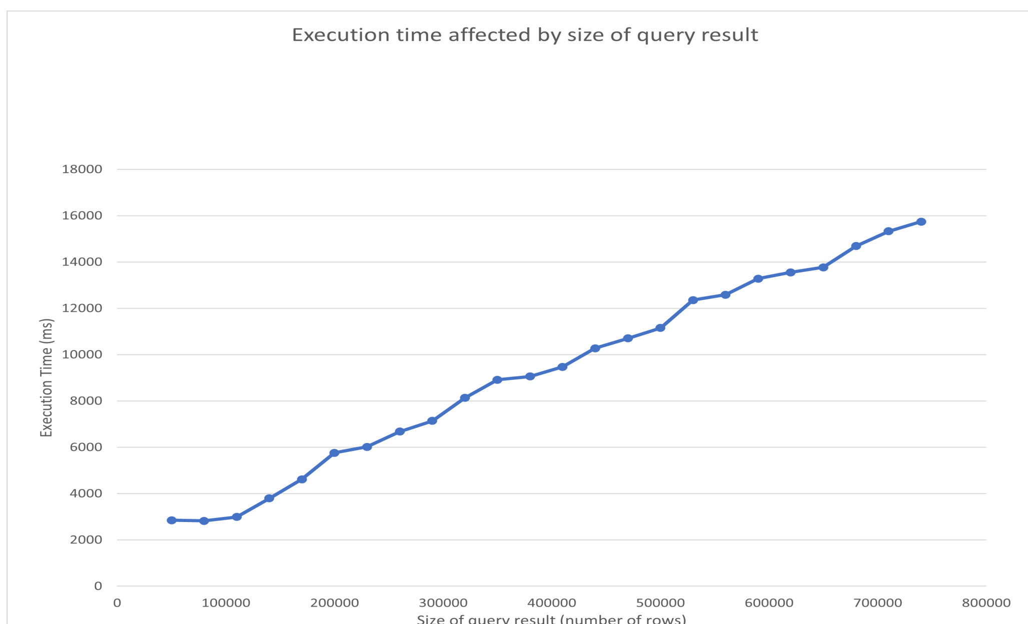


Figure 4.1: A graph illustrating how the execution time of loading one table into the cache is affected when the amount of rows is limited to the number on the x-axis.

It is clear in figure 4.1 that decreasing the size of the data fetched also decreases the query latency. Decreasing the query latency means that the execution time for the startup process will also decrease. Based on what we have found it makes sense to implement new solutions

that do not fetch all of the data in the startup process. The application also has the potential to only fetch parts of the data at startup and postpone the rest of the data loading to a later moment, if and when the data is needed. We provide a more in-depth interpretation of the results presented in this section later on in section 5.1.

4.2 RQ2 - Proposed Solutions

As explained in 4.1.3 we consider the main issue causing long startup latency to be the size of the data fetched. One way to mitigate this is to use lazy loading as described in section 2.3.2. Two proposed solutions of lazy loading have been implemented based on the strategies presented in 2.3.2. There are some major differences between the two strategies, for example, what needs to be done at startup of the cache service. This is explained in more detail in sections 4.2.1 and 4.2.2 below. The performance of the implementations is presented in 4.3. The Java class for the Hibernate Proxy Cache and the corresponding provider class can be found in A.4. Note that this code also has references to code in A.3.1, A.3.4, A.3.5 and A.3.6. For the Lookup Table Cache the code is found in A.5 with some references to code in A.3.1, A.3.2 and A.3.3.

Lazy loading means that the cost in latency of a query is only paid once we actually need the query result. If a particular result is never needed we will never pay the cost. However, if the user asks for everything in our database tables we will have to load that data at some point, lazy loading will then not get rid of this latency, it will only move it to the time of request instead of the time of startup. To eliminate the latency between the database and the application we implemented a third solution described in section 4.2.3. This implementation uses a combination of lazy loading and inter-query parallelism to let other threads handle the latency of a full load while the main thread can continue with the startup process. The code for this implementation can be found in the appendix in A.6, with some references to mutual code in A.3.1, A.3.2 and A.3.3.

4.2.1 Lazy Loading - Hibernate Proxy

This implementation uses Hibernate proxies to keep track of whether an entity has been loaded or not. There is an issue though that the proxies need to be set up in the internal data structures in the cache at startup. Therefore there are some steps that need to be done at startup, which also include some loading from the database. However, it will only load parts of the tables. If we look at graph 4.1 we can see that fetching less data will decrease the time spent on the query, which makes this solution still viable even though it has an initial load. The steps are shown in figure 4.2. As mentioned the cache service includes four different caches that are linked to one table each. The figure illustrates one of these caches being initialized.

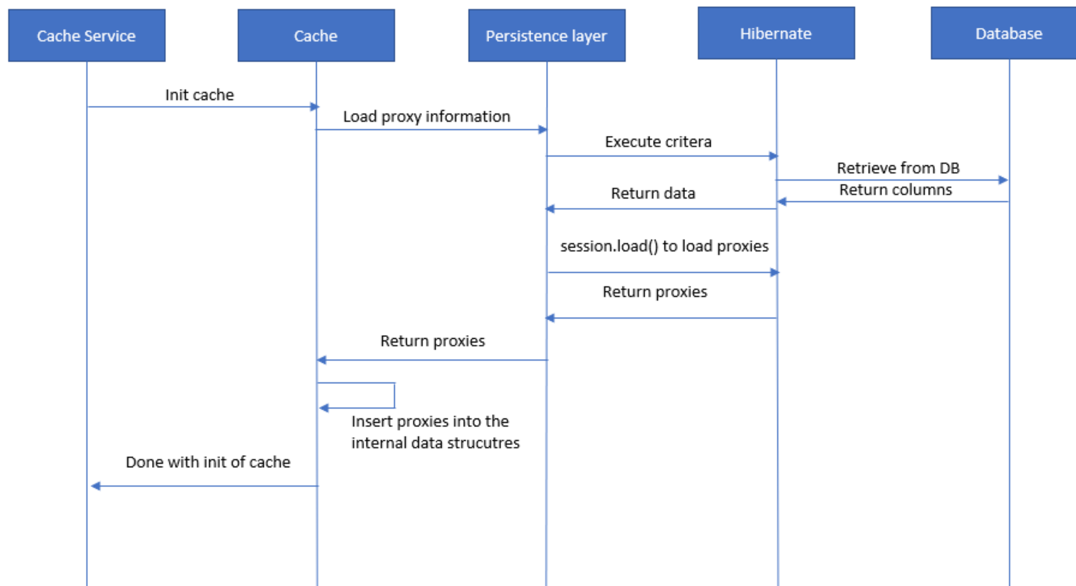


Figure 4.2: A graph illustrating the steps the cache need to take at startup with the use of the Hibernate Proxy Cache

To be able to load the proxies and to insert the proxies into the cache five columns need to be loaded from the database for each entity. Two of the columns are the primary key and are used to create the proxy. The other three columns together correspond to an account and are used to keep track of which account the proxy is connected to. We will call these five columns together *proxy information*. To retrieve the proxy information the persistence layer will create a Hibernate criteria with projections on the columns we need to retrieve. The criteria will transform the retrieved columns from the database into a DTO that saves the data and has a proxy that can be added to it. The code snippet below shows the creation of the criteria with projections and the mapping of the columns to the DTO.

```
final ProjectionList projectionList = Projections.projectionList()
    .add(Projections.property("key"), "key")
    .add(Projections.property("column A"), "column A")
    .add(Projections.property("column B"), "column B")
    .add(Projections.property("column C"), "column C");
return DetachedCriteria.forClass(entitylType)
    .add(Restrictions.eq("columnX", ColumnX.MASKED_VALUE))
    .setProjection(projectionList)
    .setResultTransformer(Transformers
        .aliasToBean(EntityProxyInfoDTO.class))
    .getExecutableCriteria(getSessionFactory().getCurrentSession())
    .list();
```

The proxy information is then used to load the proxies. By using the Hibernate function *load*, we make sure that proxies are returned and no load from the database is done. The proxies that are returned from Hibernate are saved into the DTOs and the list of DTOs are

returned to the cache. The cache loops through all the DTOs and inserts the proxies into the internal data structures and uses the information about the account to fill the account index. After this step, the initial setup of the cache is done, and with the help of the proxies, it will look like the cache is filled with all the data, even though a whole entity has not been loaded yet. See in the code snippet below how the proxy is added to the DTO.

```
final Session session = getSessionFactory().getCurrentSession();
for (EntityProxyInfoDTO<T> proxyInfo : proxyInfoList) {
    final T proxy = session.load(entityType, proxyInfo.getKey());
    proxyInfo.setEntity(proxy);
}

return proxyInfoList;;
```

In figure 4.3 we can see an example of the steps the cache will take when the mehod *getAccount* is called. As mentioned the cache's internal structures are filled with entities or the proxies of them. Therefore the cache needs to loop through all the objects and check if it is a "real" entity or a proxy. If it is a proxy, it needs to be "unproxied". The proxies are added to a list, that is sent to the persistence layer. The persistence layer sets up a Hibernate session, and with the use of locks with lock mode *NONE*, it attaches the proxies to the session. With this attachment, the Hibernate session will convert the proxies into a real object by loading them from the database. When the proxy is converted it will be so everywhere it has been used, so we do not have to return and replace anything in the cache, it is done automatically by Hibernate. When all entities have been unproxied, we can return what the application asked for. The code snippet below shows the way the proxy is attached to the Hibernate session and converted into a real object.

```
final Session.LockRequest lockRequest = getSessionFactory()
    .getCurrentSession()
    .buildLockRequest(LockOptions.NONE);
for (final T proxy : proxyList) {
    lockRequest.lock(proxy);
}
```

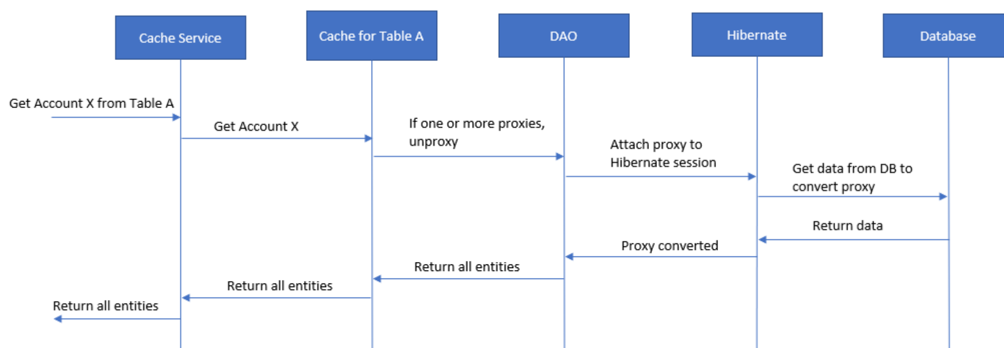


Figure 4.3: A graph illustrating the steps the cache need to take with a call to *getAccount* using the Hibernate Proxy Cache

The code snippet below gives an example of how the cache works when the application wants to retrieve a single object from the cache when the function *get* is called. As the cache already has all the objects in its internal structures a get on the map is done to retrieve the object. A check is done inside the unproxy method to check if it is a proxy or not. If it is a proxy it will be sent to the persistence layer to be converted.

```
public synchronized T get(final long entityNumber) {
    final T entity = cache.get(entityNumber);
    unproxy(entity);
    return entity;
}

private synchronized void unproxy(final T entity) {
    if (!Hibernate.isInitialized(entity)) {
        entityProxyService.unproxy(entity);
    }
}
```

4.2.2 Lazy Loading - Lookup Table

With the use of this strategy, no load from the database is done at startup. This means the internal data structures in the cache are empty when the application is fully started. Therefore there will not be any unnecessary load, only loads of data that has been requested. The loading is only done during run-time and therefore it will not impact the startup time at all. The cache needs to keep track of what is has loaded and inserted into the internal structures.

Checking if an entity has been loaded into the cache can be done by checking if it is located in the map that stores all entities, with their unique number as the key. If the map does not contain the number, it has not been loaded and a request to the database needs to be done. The code snippet below shows this function. Note that the object will be *null* if the unique number does not correspond to anything in the database. To minimize the amount of calls to the database a *null* value will be inserted into the cache. Next time the program asks for the same number, we can return *null* without having to communicate with the database.

```
public synchronized T get(final long entityNumber) {
    if (!cache.containsKey(entityNumber)) {
        final T entity = entityService.findActive(entityNumber,
            entityType);
        if (entity == null) {
            cache.put(entityNumber, null);
            return null;
        }

        add(entity);
    }

    return cache.get(entityNumber);
}
```

When the application calls the method *getAccount* we need to be sure that everything has been loaded for the account that has been passed as an argument. To check if everything has been loaded for an account the class AccountIndex mentioned in 3.2.3 has been expanded with a boolean called *loadedFromDatabase*. This boolean is initially set to *false* and will be *false* until the cache knows for sure that the account has been fully loaded. After a request to the database of all entities connected to the given account, the boolean can be set as *true*. After this load no loading has to be done for this account in the future. The code snippet below shows the implemention of *getAccount* . Notice that we first need to retrieve the account index instance that is linked to this specific account. The boolean is then checked, to see if the cache needs to load or not. After a potential load, the cache can return all the entities connected to the account.

```
public synchronized List<T> getAccount(final String accountInfo) {
    final IndexKey indexKey = AccountIndex.createIndexKey(accountInfo);
    if (!accountIndexMap.containsKey(indexKey)) {
        accountIndexMap.put(indexKey, new AccountIndex());
    }

    final AccountIndex accountIndex = accountIndexMap.get(indexKey);
    if (!accountIndex.getLoadedFromDatabase()) {
        final List<T> entityList =
            entityService.findActive(accountInfo, entityType);
        for (final T entity : entityList) {
            add(entity);
        }

        accountIndex.setLoadedFromDatabase();
    }

    return accountIndex
            .getCollateralKeys()
            .stream()
            .map(cache::get)
            .filter(Objects::nonNull)
            .collect(Collectors.toList());
}
```

The last way the application can retrieve data is to call the function *getAll* . The cache will use a boolean here to check if everything has been loaded or not. The boolean will be false until the method has been run for the first time. This means that the first time the method is called will always result in a cache miss. When we get a cache miss all the data is loaded from the database. So even though the cache might have loaded let us say 50% of all the data, or even 100% without calling *getAll* , the whole table will be loaded and the missing data will be inserted. Here we can also set all booleans in the *AccountIndex* objects to true, as we know all accounts have been loaded as well. Figure 4.4 shows the steps the cache will take in order to load everything. This means that the first call of *getAll* will lead to a load from the database as the boolean located in the cache will be *false*. Once the load is done, the boolean can be set to *true* and all that was loaded is inserted into the internal structures and returned.

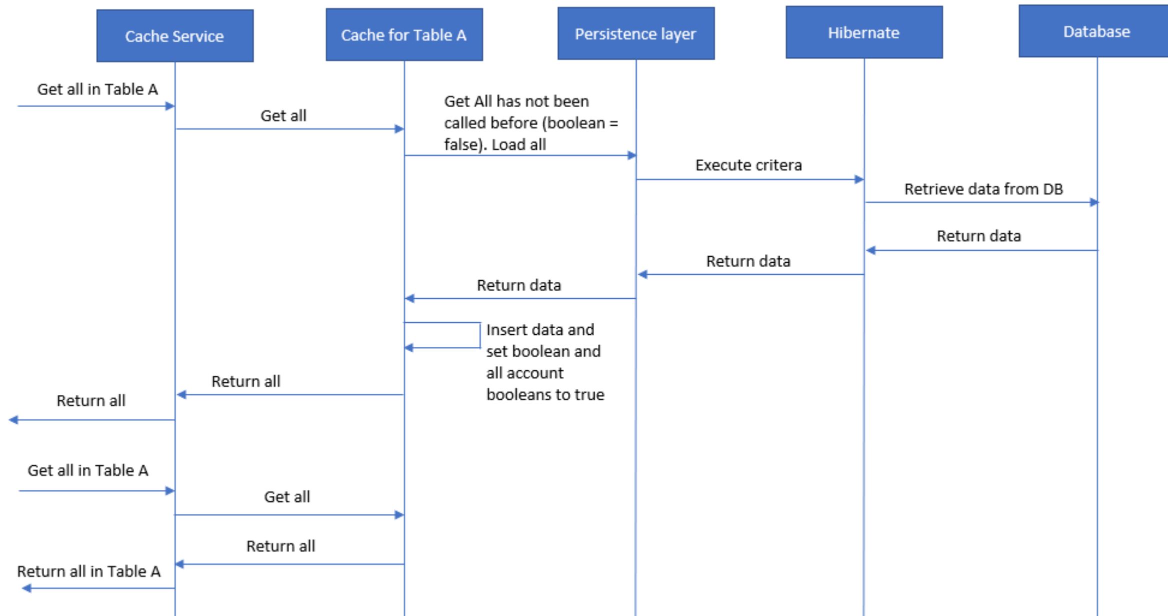


Figure 4.4: A graph illustrating the steps taken after a call to *getAll* using the Lookup Table Cache

The code snippet below shows the *getAll* method. As mentioned the boolean is first checked if the cache has loaded everything or not. If not, a load is done and the boolean is set to *true*. When we know everything has been loaded we can just take all values in the cache data structure and return the result. We also need to make sure that all account indices have their internal boolean set to true, which is done inside *loadAll*.

```
public synchronized List<T> getAll() {
    if (!hasLoadedEverything) {
        loadAll();
        hasLoadedEverything = true;
    }

    return cache
            .values()
            .stream()
            .filter(Objects::nonNull)
            .collect(Collectors.toList());
}
```

4.2.3 Asynchronous Preloading

This implementation can be considered an extension to the Lookup Table Cache implementation described in section 4.2.2. In fact, since many parts are identical we decided to make

this cache class a subclass of the cache class in 4.2.2. However, there is a relevant difference regarding how the cache is loaded.

In this implementation, a number of threads are spawned with the purpose of preloading the cache with all relevant entries for a specific table. This way another thread takes care of retrieving the data from the database and if these preloading threads are sufficiently quick the data has already been fetched when the user requests it.

As described in section 2.4, when parallelizing a sequential algorithm the number of threads to run in parallel needs to be decided. Additionally, the original task needs to be split up into sub-tasks that can run concurrently. We initially started with only 1 parallel thread taking care of preloading all the 4 tables into the cache. The machines we are running our experiments on have 4 hardware threads available so there was potential for a higher level of concurrency. In our case, splitting up the preloading task into more subtasks is also very simple, we can simply run one thread to preload each table. In table 4.3 we present the time it takes for the preloading to finish for each table when using a single preloading thread compared to 4, i.e one for each table. As can be seen, spawning 4 threads clearly speed up the process of preloading so this is what we use in the final implementation. Note that we are running 5 threads concurrently while preloading is in progress, 1 main thread and 4 preloading threads. Of course, other programs running on the same machine or other threads created by other parts of the application may also run their own threads so the hardware resources have to be shared between software threads.

Table Name	1 Preload Thread	4 Preload Threads
Table 1	42 s	32 s
Table 2	73 s	37 s
Table 3	103 s	36 s
Table 4	132 s	34 s

Table 4.3: Time since the first thread started until completion of the preloading process of the specific table. With only one thread it has to preload each table sequentially, i.e. one after the other. With 4 threads all of the tables can be preloaded concurrently. The results were obtained by reading log files. All values are averaged over 3 executions. In this test, we used tables of 400 000 rows.

When the preload threads have completed this cache implementation acts as the Full Load Cache. Everything have been loaded so there is no run-time latency in the cache regarding database queries after this point in time. However, there is a possibility that the user requests data from the cache directly at startup before the preload threads have completed which needs to be handled. In the first implementation of an asynchronous preloading cache, we simply waited until the preloading threads were finished before returning data from the cache to the user. However, this is not optimal since if the user is only asking for a particular row of a table, waiting for the preload thread to finish means that we have to wait for a load of the whole table which is a lot slower than just fetching one row. Instead, we observed that this is exactly the problem we are solving with our lazy loading caches, i.e. only fetching the data we need. Therefore, we decided to let this cache work as the Lookup Table Cache until the moment when the preloading is finished. This means that we can have 5 software threads accessing the database concurrently while preloading is still in process. The main thread is 1

of these and the other 4 are preload threads. In figure 4.5 the described behavior is visualized. There is an exception to this behaviour when a user asks for everything in a table before the preload is finished. When that happens there is no reason to start a new request since fetching everything from a table is exactly what the preload thread is already doing in parallel so we might as well wait for it to finish.

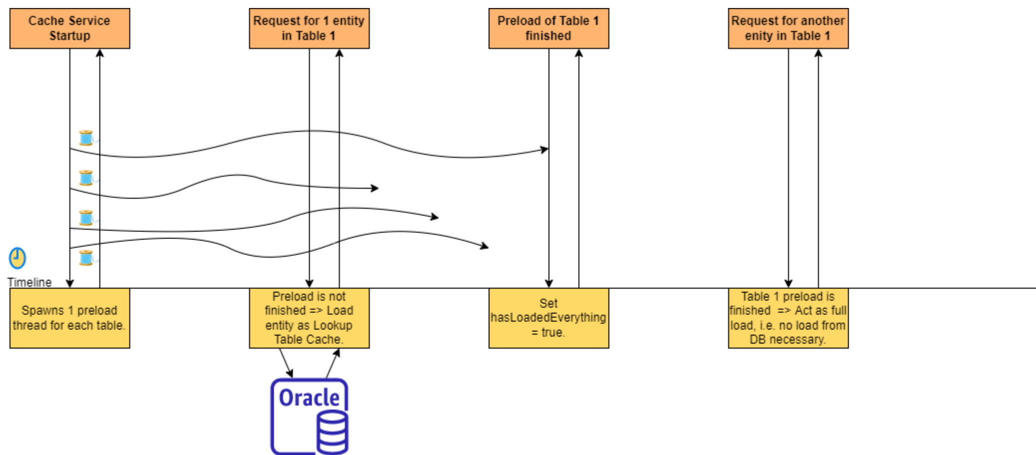


Figure 4.5: A simplified view of the timeline at startup for the Asynchronous Preloading Cache . This cache acts as the Lookup Table Cache until everything has been loaded by the preload threads.

To spawn the preload threads we use a thread pool class provided by the spring framework called *ThreadPoolTaskExecutor*. At the start of implementing this solution, instances of Java's standard *Thread* class were used instead, but we realized that in a large project such as this it might be better to use a common thread pool. This way the common thread pool can orchestrate the scheduling of all threads in the project instead of each developer having to worry about this. Java's built-in *synchronized* keyword is used to synchronize any shared state.

4.2.4 Hibernate Built-In Caches

As explained in section 2.2.4 Hibernate has its own built-in caches. First of all, we looked at if the second-level cache and/or the query cache could replace the current cache service in its entirety. If we look at the second-level cache there is an issue that the entities have to be accessed by their primary key. The way we want to access the data, as described in 3.2.3, is based on the methods *getAll*, *getAccount* and *get*. None of these uses the primary key to access data. The unique number used for *get* is part of but not the whole primary key that would have to be used to retrieve an entity from the secondary cache. This creates a mismatch of how the program needs to be able to handle data, versus how the second-level cache handles it. Therefore, it would not be sufficient to only activate this functionality and make sure the entities are cached, we need some sort of caching service.

We could create a cache service that would work as a middleman, a connection between the second-level cache and the program. This would require an initial load of some columns to get the data that the cache service would be required to have to structure the calls based on

the primary key. The requests would then use this data to retrieve entities from the second-level cache, or if not there, from the database. The point of the secondary cache however should be that it is pluggable and easy to use out of the box. Therefore, we decided to not implement a solution based on this either. If we use some sort of cache service, the cache service in itself could store the data in its caches as with the other proposed solutions, no need to add two layers for caching.

There is also the query cache that could be used to cache queries. But as mentioned in 2.2.4 this is not recommended to be used in most applications because of the added overhead. The program will most likely not ask for the same data over and over again, which means there will be an execution of different queries. Because of this, we decided that the query cache would not be an option as well.

4.3 RQ3 - Gathering Experiment Data

As described in the approach section 3.4, the experiment has been divided into two parts. The first part is a measurement of the effect a solution has on the total startup time of the program. The result of this part can be found in figure 4.6. The measurement of the startup time with no cache service is not dependent on the size of the database and therefore only one set of runs are needed.

It is also interesting to look at the time it takes for the preload threads to finish, as this is when the Asynchronous Preloading Cache will achieve fast run-time access. The result can be found in figure 4.7.

The second part is the measurement of the time it will take to retrieve data from the cache with different scenarios. Some scenarios have been grouped in the same table as they test the same thing with different modifications. Each experiment has been executed with 250k and 750k rows in each table. The result of each experiment can be found below. Bar charts is used to illustrate the times, but as some of solutions are much slower in some tests we also provide a table inside the figure to show more detailed times. The time values are presented in milliseconds.

To get an overview of the maintainability of the code we have calculated the number of lines of code as described in 3.5. The result of this can be found in table 4.4.

After running the experiments we could see that the Hibernate Proxy Cache implementation is much slower at certain tests. By looking at figures 4.8 and 4.9, we can see that the Hibernate Proxy Cache solution is a lot slower when the method to retrieve everything is called for the first time. We know that the Lookup Table Cache loads everything, which makes it a good comparison, and the Hibernate Proxy Cache is much slower. We investigated why this is, and by using the Hibernate Statistics described in A.2 we found that Hibernate creates one JDBC statement per entity to be unproxied. So as an example, if we have to convert 500k entities, there will be 500k JDBC statements created by Hibernate. This is even though the whole unproxy process is executed inside a single transaction.

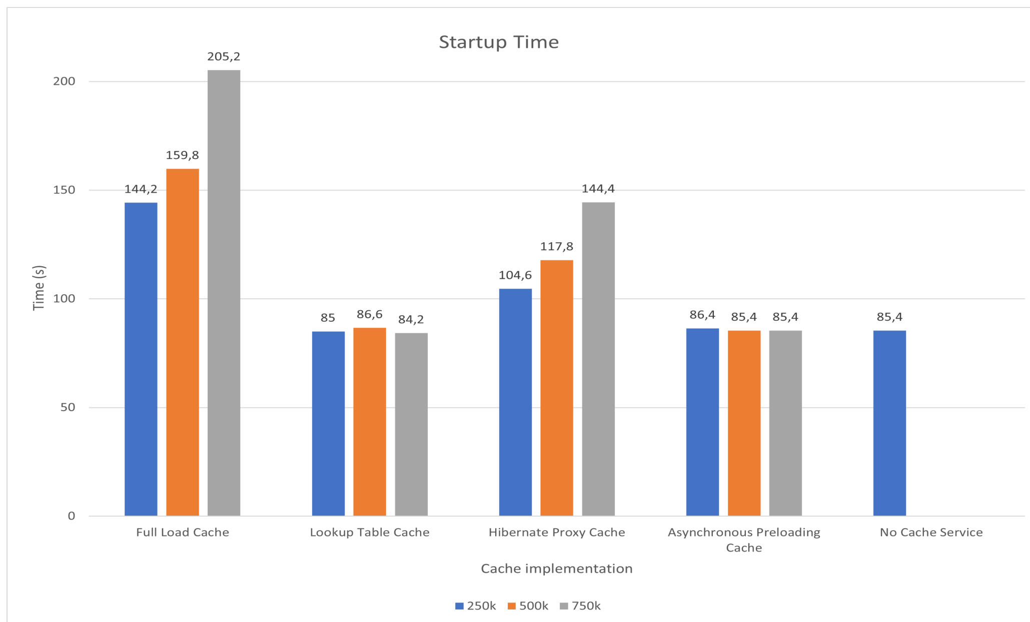


Figure 4.6: A bar chart illustrating the startup time of the service for each cache implementation with 250k, 500k, or 750k rows per table. The shown result is an average of 5 runs.

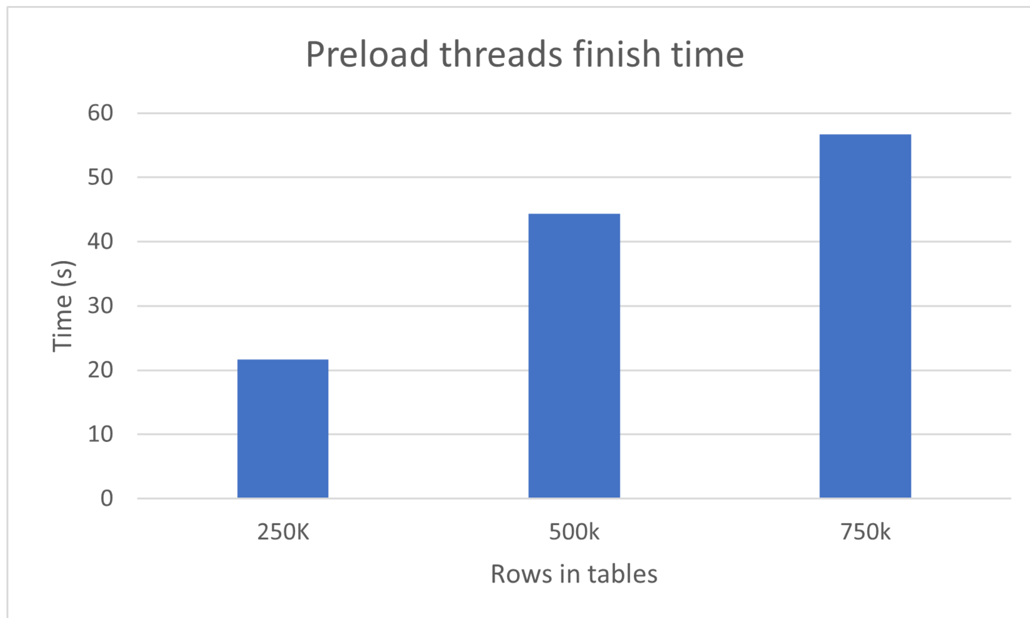
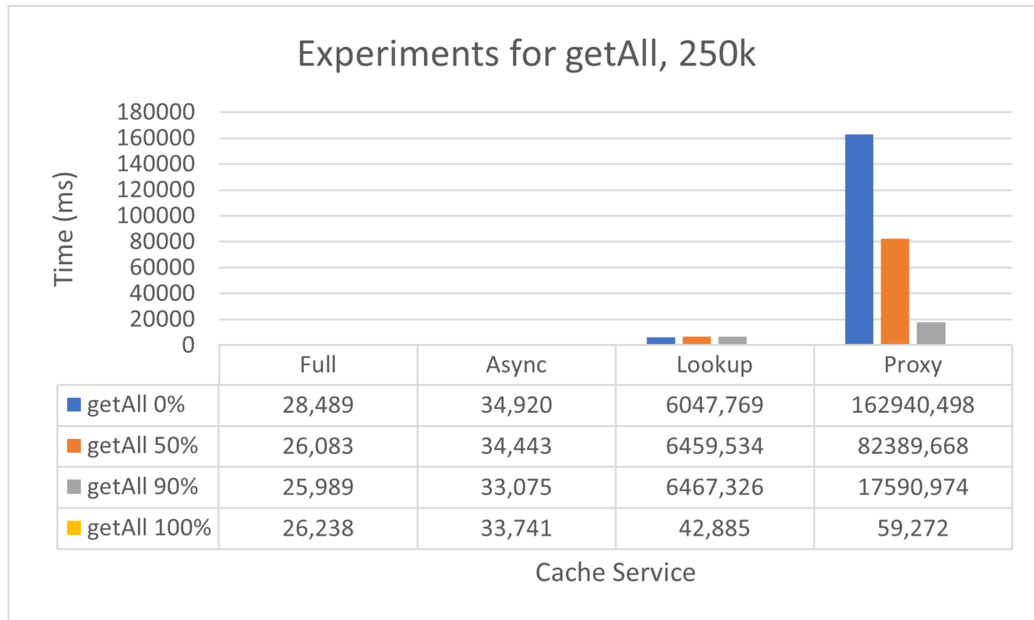
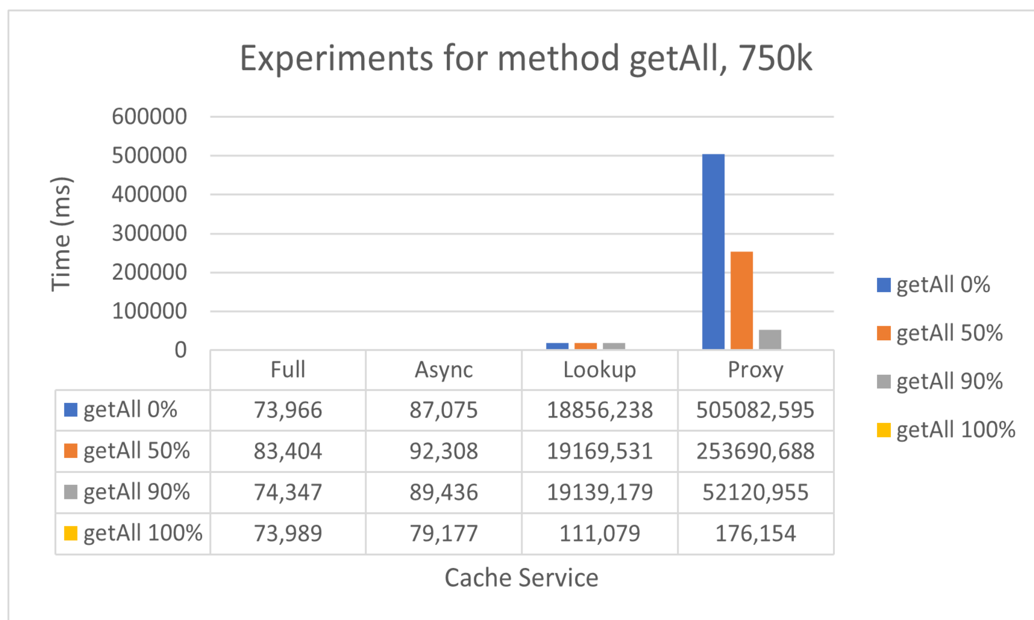


Figure 4.7: A bar chart illustrating the time it takes for the preload threads in the Asynchronous Preloading Cache to finish with 250k, 500k, or 750k rows per table. The shown result is an average of 5 runs.



	Full	Async	Lookup	Proxy
getAll 0%	28,489	34,920	6047,769	162940,498
getAll 50%	26,083	34,443	6459,534	82389,668
getAll 90%	25,989	33,075	6467,326	17590,974
getAll 100%	26,238	33,741	42,885	59,272

Figure 4.8: A bar chart illustrating the average measured time for the method *getAll* using 250k rows.



	Full	Async	Lookup	Proxy
getAll 0%	73,966	87,075	18856,238	505082,595
getAll 50%	83,404	92,308	19169,531	253690,688
getAll 90%	74,347	89,436	19139,179	52120,955
getAll 100%	73,989	79,177	111,079	176,154

Figure 4.9: A bar chart illustrating the average measured time for the method *getAll* using 750k rows.

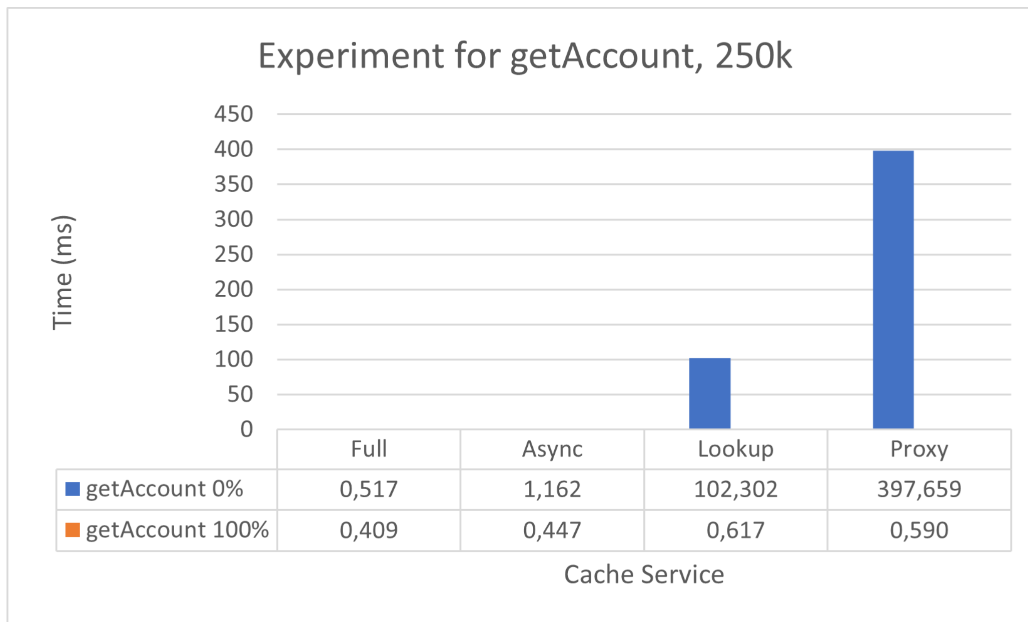


Figure 4.10: A bar chart illustrating the average measured time for the method *getAccount* using 250k rows.

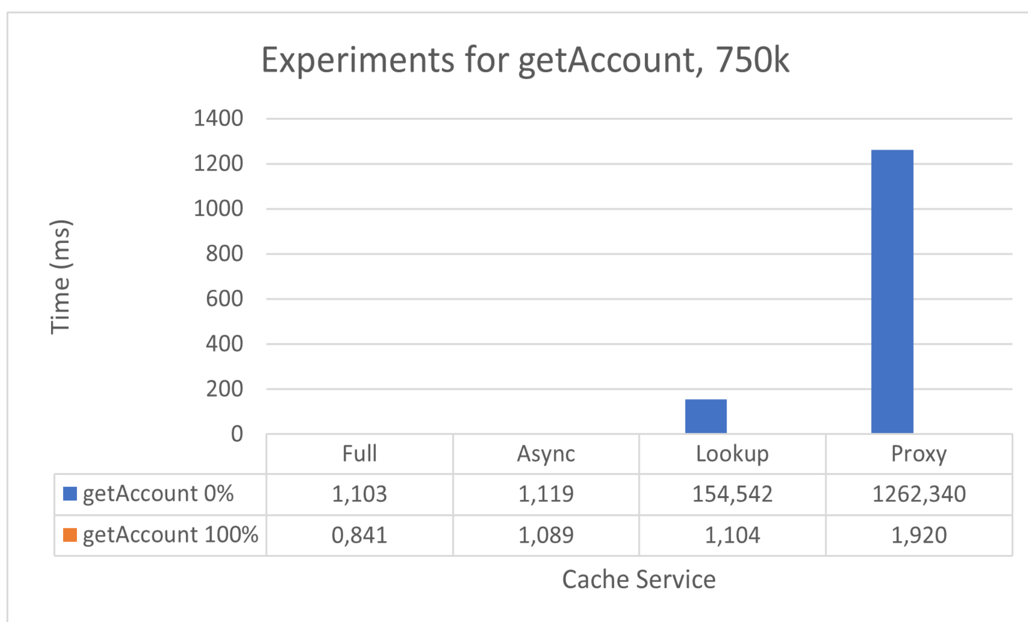
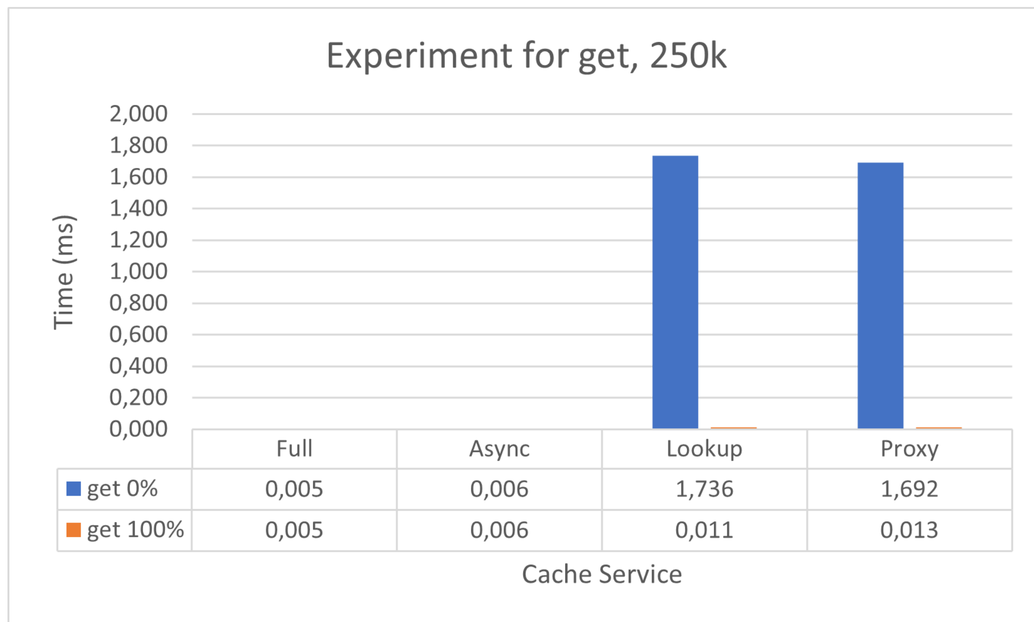
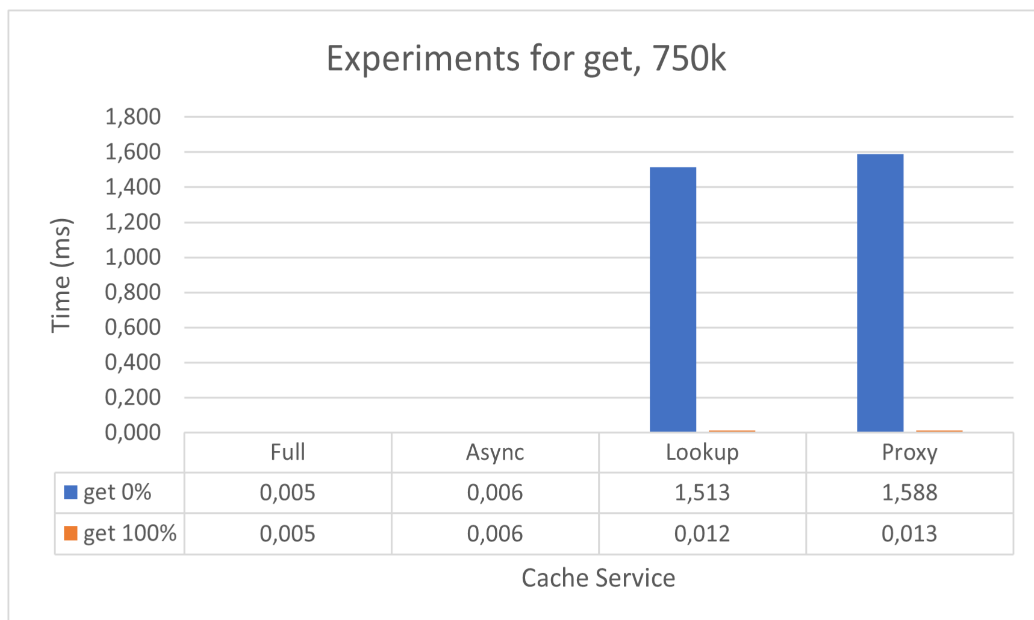


Figure 4.11: A bar chart illustrating the average measured time for the method *getAccount* using 750k rows.



	Full	Async	Lookup	Proxy
get 0%	0,005	0,006	1,736	1,692
get 100%	0,005	0,006	0,011	0,013

Figure 4.12: A bar chart illustrating the average measured time for the method *get* using 250k rows.



	Full	Async	Lookup	Proxy
get 0%	0,005	0,006	1,513	1,588
get 100%	0,005	0,006	0,012	0,013

Figure 4.13: A bar chart illustrating the average measured time for the method *get* using 750k rows.

Implementation	Lines of Code		
	Cache and Provider	Database Access	Total
Full Load Cache	92	28	120
Lookup Table Cache	139	53	192
Asynchronous Preloading Cache	186	53	239
Hibernate Proxy Cache	146	89	235

Table 4.4: Lines of Code for each solution. Calculated as described in 3.5. The code is shown in different sections of the appendix. Note that the lines of code may appear different to due a different text-format in this report compared to when viewing the code in a code editor.

Chapter 5

Discussion

5.1 RQ1 - Interpreting Inefficiencies

We can observe in table 4.2 that Table 2 takes a bit more time than the other three tables to load. A probable reason for this is that Table 2 is the biggest table but since there is some ambiguity regarding the size of the query result, as described in section 5.5, we cannot be sure. Although this observation implies that Table 2 should be prioritized over the other tables we believe that the solutions we provide in section 4.2 applies to all tables, therefore we will not investigate further why Table 2 takes a bit more time to load. Additionally, the query used in listing 4.1 is very similar for all 4 tables.

When it comes to long-running database queries there are several components where execution time can be spent. We summarize an overview of these components in image 5.1. First, all database requests are started in the application source code. When the query result arrives the application source code performs some operation on the result. Secondly, the database operations are in this case handled by the Hibernate framework. In the third step the request, and later on the result, has to be transferred over the network. Finally, the query processing at the database, calculating the result, will take some time.

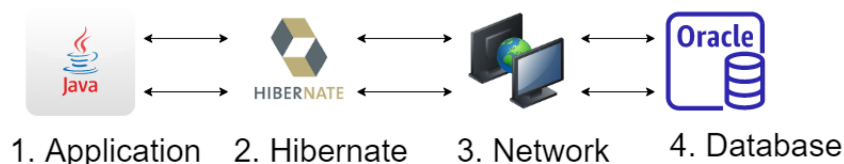


Figure 5.1: Components of a database query.

As explained in section 4.1.3 the time spent in the Hibernate queries were measured to 70% of startup time for the initial solution. Note that time spent in a Hibernate query includes operations in the Hibernate framework as well as time spent in the network and on

query processing at the database. This means that the rest of the execution time spent on other parts of the source code in the startup process only consists of 30%.

No measurements have been collected on what fraction of time is spent on the operations in the Hibernate framework alone, excluding network and query processing. There are two aspects of performance to consider. Firstly, ORM frameworks can produce unnecessarily complex and redundant queries as explained in 2.2.5. This is not an issue in our case as shown by the simplicity of the query in 4.1. Secondly the framework will add overhead when performing operations to compose the queries and map the results to Java objects. An alternative framework could have been tested to improve performance. However, changing the ORM framework leads to changes in many parts of the service. Additionally, it is easier for the developers to use the same ORM framework in all services. It should also be pointed out that the performance of Hibernate has been tested in many previous papers, such as [20], [53] and [6], with the results showing good performance. Hence, the potential gain in performance of switching to another framework is limited.

Network capacity will definitely have an impact on performance. Even if excluding the other parts of a database query a large amount of data still have to be transferred over the network if everything is fetched. Consider a customer with a bandwidth of 50 Mbps and that we need to transfer our 400 MB = 3200 Mb, as seen in table 4.2, it would take 3200 Mb / 50 Mbps = 64 s just to transfer the data. This is in the best case assuming no overhead in the data packets transferred, which is of course not the reality, and that we achieve the full bandwidth speed. However, as already described in section 4.1.1, improving the network capacity is not a feasible solution since the product is a software solution often hosted at the customers network setup. Even though changing the network is not suitable, changing the way data is requested can lead to having to transfer less data. This is achieved with the lazy loading strategies described in 4.2.

It would be theoretically possible to make more detailed measurements on how much time is spent with query processing at the database. There are tools in Oracle SQL Developer which gives statistics on time spent on query processing at the database and time spent in the network. However, we do not have the database administrator access rights necessary to use these tools. Similarly, we don't have the permission rights necessary to install and run network profiling tools at the database nodes. If the time spent on query processing would be large one could look at ways to speed up the efficiency of the database. This could perhaps be done by removing query complexity like the *order by* clause, adding an index to the column in the *where* clause, or trying another DBMS for example. Queries fetching large amounts of data are always likely to be slower than queries fetching less data but it is still possible to improve scalability with the size of the result. The loading strategies that we have looked at could be combined with optimizations in the database query processing. However, since the query in 4.1 is already very simple, we did not see much potential in removing complexity and adding indexes. Additionally, we felt that this report should focus on one topic, rather than mixing loading strategies and query processing optimization.

Looking at the measurements on time collected in section 4.1 it is clear that decreasing the size of the data fetched also decreases the execution time of the query. Decreasing the size of the data fetched in a query optimizes performance across all the layers that a database query is composed of. Fewer objects need to be handled in the application code, more precisely in the cache data structure in our case. The Hibernate framework needs to allocate less memory and perform operations on less entities. A smaller amount of data is transferred

over the network. The database query processing also doesn't need to handle as many entities, improving performance.

We observed that the original solution queried for everything at startup although all of the rows and all of the columns may not be needed. Even if all of the data would be needed eventually, the data loading could be postponed until after the startup process. Fetching data in the startup process, as done in the original solution, makes the application unresponsive for a longer time. This is a problem especially if the server crashes and a quick restart is needed. In this scenario, one could look at how the Hibernate framework is used in the application and only make the appropriate calls when and if they are necessary. In this manner, different loading strategies were investigated to load data in smaller portions at a time or to load data from a parallel thread.

5.2 RQ2 - Dissecting Implementations

In this section we discuss the implementations in more detail by giving more insight into the functionality and logic to make them work. We also discuss the advantages and disadvantages based on the theory and functionality of the solutions.

5.2.1 Lazy Loading - Hibernate Proxy

As mentioned in the description of this implementation 4.2.1, there is a load of five columns at startup. As we can see in the figure 4.6 this has an effect on the startup time. However, not as much as loading all columns as the Full Load Cache does. There is a risk that the cost at startup and converting the proxies at run-time is too costly if we add them up together. If we look at the Lookup Table Cache there is no cost at startup, all the cost of loading is moved to run-time. If the Hibernate Proxy Cache does not gain any advantage during run-time compared to the Lookup Table Cache , the cost at startup is just an overhead. Therefore you could argue that this implementation must be faster than the Lookup Table Cache during run-time to be a viable solution.

One advantage of this implementation compared to the Lookup Table Cache is that no entity will be loaded twice. After startup it will look like the cache is filled as with the Full Load Cache. The difference is that before returning an entity, there needs to be a check if it is a proxy or not. For example, if only 10% of all entities are still proxies, and the program calls *getAll* , only these 10% of the data needs to be handled by Hibernate and be unproxied with a load from the database.

Using high-level frameworks like Hibernate could reduce the complexity of the code. Much of the logic is done by Hibernate instead of the programmer doing it with a more low-level approach. One issue with high levels is that the performance could degrade. The developer loses control over what exactly is done, for example how the database queries look like. The potentially negative effect on performance could be worth it, with the reduced complexity. But if the negative effect is too large, it is not worth it. This is a balance that needs to be analyzed when using high-level solutions.

Thread-safety is also an important topic that needs to be analyzed. To unproxy the proxies, the implementation uses a Hibernate Session. These sessions are not thread-safe in themselves. Therefore it is important that there is only one thread connected to each session. If

there are two threads connected to one session there will be data races. By default Hibernate's *getCurrentSession()* method, which is used in this implementation, returns different sessions for different threads. We also checked that this setting had not been changed and still had the default value. In order to attach a proxy to a new thread when unproxying, if another thread created the proxy, we use Hibernates built-in function in the listing below.

```
Session.buildLockRequest(LockOptions.NONE).lock(Object proxy)
```

5.2.2 Lazy Loading – Lookup Table

To reduce the startup time this solution is optimal as there is no load into the caches at all during startup. However, the latency is moved into run-time instead. This could create long response times when a large amount of data is requested for the first time. But after the data has been loaded it will not have to reload it. This means the added latency cost during run-time is a one-time concern.

An advantage of this solution is that only what is used is loaded into the caches. This means that the program memory does not get filled with entities that will never be used. This could in the long run decreases the chance of running out of memory on the heap which could crash the whole service. However, if the program asks for the cache to retrieve all or almost all of the data the number of objects stored on the heap will be the same. But at least all the data that has been added to the heap were explicitly asked for and not added to the heap preemptively.

There is however an issue with this solution that loads could be done of entities that are already in the cache. This is because the cache cannot know until it has made sure that it has loaded everything that the program asks for. In figure 5.2 a scenario is shown that exemplifies this issue. Let us say that the entity with the number N is the only object connected with account X. As seen in the first call in figure 5.2 the entity is located in the cache as no database loading had to be done. However, when the application calls *getAccount* with account X as an argument there is a load anyways. The cache cannot know that account X only has one entity before it has been checked once. So what the load here returns is one object that is already in the cache, so nothing new is added and an unnecessary load has been done. It could be much worse, for example when 90% of all data has been loaded, and then the program asks for everything. The cache will then still have to load everything from the specific table, although 90% was already in the cache.

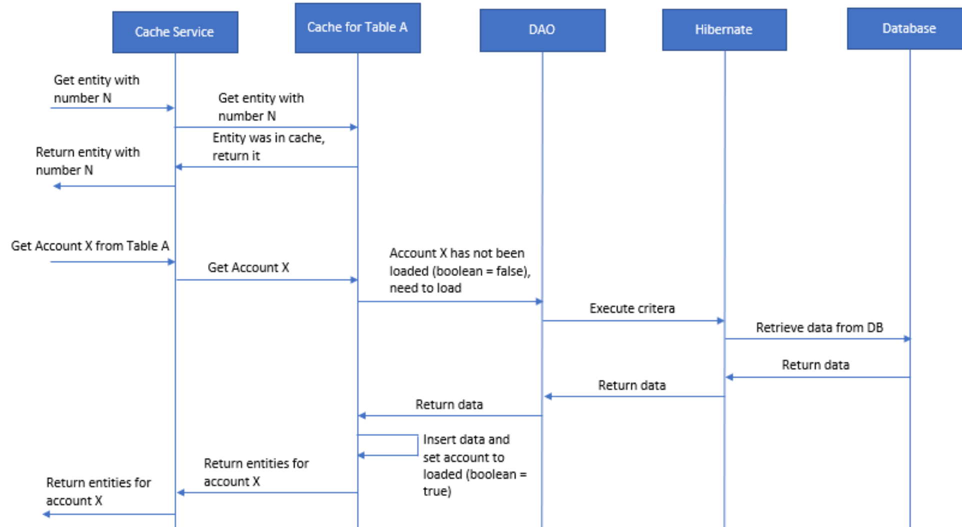


Figure 5.2: A graph illustrating unnecessary loads when using the Lookup Table Cache

5.2.3 Asynchronous Preloading

This solution has great potential in terms of both startup and execution time since it moves the startup latency to other threads while still achieving the same short run-time latency as the Full Load Cache. But the run-time latency is slightly ambiguous because it depends on how quickly the data is requested from the cache compared to how quickly the preload threads can finish.

A downside with this solution and multithreading in general is that working with threads tends to introduce a higher level of complexity. Our task decomposition and thread-orchestration are relatively simple since each thread can work on its own dedicated table and only synchronize with the main thread. Still, there are a few places where thread-safety needs to be discussed.

As can be seen in the code in A.6, the three different get-methods are very similar to the ones in the Lookup Table Cache which can be seen in A.5. The difference is that we do not synchronize over the database operations since we want the preloading threads to be able to run them in parallel with the main thread accessing the cache data-structures. A consequence of this is that we need to check twice if a certain entity has been loaded. First, we check if the entity needs to be fetched from the database, but when this is done another thread may have already fetched the entity in parallel in which case we do not want to overwrite the entity, it might have been updated or removed.

When the main thread fetches entities before a preload thread has finished special care is needed so that the preload thread does not overwrite the entities already fetched. When an entity has not been removed after fetching it this is straightforward, simply check if the entity exists in the cache's map data structures. However, if the main thread both fetches an entity and removes it before the corresponding preload thread is finished extra care is needed. If an entity is removed from the data structures completely, there will be no way for the preload thread to check if the entity has not been loaded before, since there is no trace

left of that entity. We solved this by instead storing null as the value for entities that have been removed. Note that this means that if entities are added to and removed from the cache over and over the cache will simply grow larger and larger. If such a behavior is expected a strategy to clean up these removed entities when the preload thread is finished should be added.

Regarding this solution there are some small modifications that could have been done to either increase efficiency at the cost of more complexity or the other way around.

It is possible to increase the level of concurrency in this solution. This could be done by letting a number of threads fetch a certain fraction of the table rows rather than the whole table. Each part would then have to be joined together with the other parts of the table, fetched by other threads, in the cache. Methods of CDC, as described in 2.3.1, could be used in that scenario to determine what has already been loaded to the cache by another thread. The optimal number of threads to start in such a solution would depend on the overhead of joining the tables into the cache and the resources available at the host and in the network. On the machines that have been used for our experiments, there are only four hardware threads as previously described. Therefore, we did not see much potential in increasing the level of concurrency at the cost of complexity in the application code, since in this case hardware is the limiting factor.

In this solution we used four preloading threads to speed up execution as described in table 4.3 and in the source code in A.6. It should be pointed out that this parallelization is possible in the Full Load Cache as well, with the difference that it would wait for all the preload threads to complete before progressing. This waiting should be simple to implement, since the thread pool class that we are using includes such functionality. Such a solution should improve the startup time of the Full Load Cache . Although the startup time of the Full Load Cache would improve, it would not be as fast as the Asynchronous Preloading Cache solution since it would still have to wait for the threads to finish. However, the complexity would be a lot lower in such a solution since there would be no need for the synchronization and the lazy loading behaviour which is in place when the parallel preloading and the main thread are allowed to run in parallel.

5.2.4 Alternative Solutions

The solutions described in 4.2 are the methods we judged to have the most potential while still only requiring a reasonable amount of work for a master thesis. But of course, there are other alternative solutions that we have not evaluated in this thesis that still deserves to be mentioned.

One idea is to prioritize certain entities, either each entity has its own priority or the priority is decided by the corresponding account. These prioritized entities could then be loaded already at startup in a lazy loading solution. The priorities could be determined by reading some statistics from the database, such as the last access date. An alternative is to use a machine learning algorithm for the cache to learn which entities are accessed most frequently.

Another idea is to decrease the size of the data elements in the tables. Looking at 4.1 we can see that the size of the data transferred is correlating very closely to the startup execution time. However, such a solution would probably mean large changes in the whole system since the data types might need to change everywhere. Determining the data type size necessary for

each column is also a difficult problem relaying heavily on knowledge of the business needs.

In the solutions presented in 4.2 we use lazy loading for the rows. Lazy loading strategies could also be used to only fetch some of the columns rather than all columns of the entities. Another way to only load some columns at a time would be to explicitly split up the current tables horizontally and let the current entity be represented by two or more entities. This way you may only have to fetch one of the entities, corresponding to the group of columns required at the moment. Both of these changes should be a lot more complex to implement compared to the lazy loading of rows since it would require a mechanism to determine if each column for each entity is already fetched or not.

Pagination can be introduced to only load parts of the tables at a time as explained in section 2.3.3. However, with pagination at most a single page of data can be accessed at all times. This would introduce big changes in the whole system, not just the cache, leading to a scope too large for a master thesis. Additionally, pagination is only possible when the business requirements allow the application to handle one page of data at a time. Therefore pagination would require a more detailed understanding of the business requirements.

5.3 RQ3 - Comparing Solutions

This section will compare the implementations based on the result gathered and presented in section 4.3.

5.3.1 Startup Time

In figure 4.6 we can see the effect the implementations have on startup time. As expected the Full Load Cache takes the most time, especially in the scenario with 750k rows per table. This is one of the main issues of the original solution that our proposed implementations are attempting to alleviate. The measurements show that all our implementations reduce the effect the cache service has on the total startup time.

The Lookup Table Cache and Asynchronous Preloading Cache solutions have the shortest startup time. This is expected as no loading has to be done before the service is fully up and running. In figure 4.6 there is some difference between the averages. These differences are so small that we cannot draw any conclusions considering the accuracy of our measurements. Both of these strategies are as fast as having no cache service at all. There is of course some overhead with starting up the cache service, but it is an insignificant part of the total startup time.

The Hibernate Proxy Cache is faster than the Full Load Cache but slower than the other two of our implementations. Compared to the other two, the proxy solution loads from the database at startup to create the proxies and structure the caches. However, by comparing the startup time between the Hibernate Proxy Cache and Full Load Cache we can see that only loading parts of the whole table is reducing the impact that the loading has on the total startup time. To determine if this extra load is worth the effort we need to analyze the run-time experiments.

Besides the startup time of the Asynchronous Preloading Cache, it is interesting to look at how long it takes for the preloading threads to finish. This is an important measurement as the solution will have loaded everything from the database at that point and the latency

of requesting data from the cache will be insignificant. Before preloading is finished, this cache will have a run-time latency similar to the Lookup Table Cache. What we can see from the measured time in figure 4.7 is that preloading finishes faster than the Full Load Cache when it loads in startup. This is a good sign, the parallelism is faster than loading each cache sequentially. The efficiency of the preloading threads seems to not be ideal though. If we take a look at how long it takes for the Lookup Table Cache to load all the data in figure 4.9 we can see that it is faster for one individual query. This comparison is not fully accurate since we are comparing the average execution time in 4.9 while we are looking at the slowest thread out of four in 4.7. It gives us a hint however what an ideal scenario would look like if the threads could run in parallel without any bottleneck or overhead. It may be that the preload threads are slower for an individual query since the threads have to share resources and we have a limited amount of hardware threads. Additionally, the preloads threads work in both startup and run-time, running in startup might mean that they have to share resources with other parts of the startup process.

5.3.2 Run-time Request Latency

By analyzing the experiment results presented in section 4.3, we can understand that the difference in the solutions is mostly based on if the cache needs to load from the database or not. Both the Full Load Cache and Asynchronous Preloading Cache are fast in all of the experiment, as both of them already have loaded everything when the tests are executed. The Lookup Table Cache and Hibernate Proxy Cache solutions need to load from the database in some scenarios, which makes them take more time overall in these run-time experiments.

The result in figures 4.8 and 4.9 shows that the Hibernate Proxy Cache solution is considerably slower than the other solutions when *getAll* is called for the first time. As mentioned in the result section we found that this is because Hibernate creates one JDBC statement per unproxy. Adding up all these JDBC statements together creates an enormous overhead that leads to a slower execution time. The Lookup Table Cache is much faster in comparison. As we know that the Lookup Table Cache loads everything from the database for this specific experiment we could say that all solutions that have worse execution times are not a viable option. At least not viable to load a large amount of data from the database. This seems to be the major issue of the Hibernate Proxy Cache implementation.

If we look closer into the experiment that requests data from the cache that has been previously accessed we can see if there is any notable overhead in any of the solutions. In tables 4.8 and 4.9 we can compare the measured time when we call the *getAll* method for the second time. In figures 4.10, 4.11, 4.12 and 4.13 we can look at the methods *getAccount* and *get* and how both of them act when the data requested have been previously accessed. What we can see is that the difference in time is too small to draw any conclusion for all implementations except for the Hibernate Proxy Cache solution. The solution seems to create a bit more overhead for the function *getAll*. This is likely because there is a check for each entity if it is a proxy or not. For the other solutions, the difference is insignificant, so the overhead for the logic implemented for the Lookup Table Cache and Asynchronous Preloading Cache is not notable compared to the initial solution.

For both the Full Load Cache and Asynchronous Preloading Cache there is never a load to the database in any of the run-time experiments as mentioned. We can also see that how much has been accessed beforehand makes no difference in the latency. To clarify, the reason

the Asynchronous Preloading Cache does not have any load is that we wait for the preload threads to finish before running the experiments. That is why the measured times are small for all the experiments that tested the method *getAll* shown in figures 4.8 and 4.9. If data is requested before the preload threads are finished the Asynchronous Preloading Cache will act like the Lookup Table Cache. In other words, when discussing the run-time latency of the Asynchronous Preloading Cache the probability of data being requested before the preload threads are finished also has to be considered. Table 4.7 shows how long it takes for the preload threads to finish, while how quickly the data is requested depends entirely on the use case.

There is however a difference in how both the Hibernate Proxy Cache and Lookup Table Cache solutions act depending on previously accessed data in the experiments. For the Lookup Table Cache we can see that the time is stable for the three experiments that have not called the function *getAll* yet. This is because as mentioned it will load everything again as it cannot know what the database has stored. However, the second time the method is called it is as fast as it would be using the Full Load Cache or Asynchronous Preloading Cache. The Hibernate Proxy Cache solution on the other hand can detect what it has to load, which means the execution time decreased depending on how much has been previously accessed. But if we look at the execution times this implementation is slow for most tests, for the same reason as before with one JDBC statement per entity to be unproxied.

It is interesting to see how the implementations differ when there is a request for a smaller portion of data for the first time. A request to the method *getAccount* is interesting because it is a fairly large subset of the whole database. These experiments can be found in figures 4.10 and 4.11. A call to the method *get* is also interesting because it shows how the cache implementation acts with the smallest subset. These experiments can be found in figures 4.12 and 4.13. What we can see is that the Full Load Cache and Asynchronous Preloading Cache produce the fastest times, as expected. What is interesting is that the Lookup Table Cache is faster than the Hibernate Proxy Cache for the method *getAccount*, but similar in execution time with a call to *get*. Why the Hibernate Proxy Cache implementation is slower with accounts can be explained by the same flaw described earlier. This flaw makes the Hibernate Proxy Cache a worse implementation overall. It is worth pointing out as well that the latency for *get* requests are very small for all implementations. Latency of a few milliseconds is not likely to cause a disturbance to the customer, at least not in the case company's use case.

5.3.3 Maintainability

Table 4.4, shows a comparison of the lines of code for each implementation. As mentioned in section 3.5, this measurement gives an overview of how maintainable a solution is. The table shows that the Full Load Cache solution has the least amount of code. This is mostly because the Full Load Cache only needs one way to fetch data from the database (*getAll*). The Full Load Cache additionally does not need any logic to check if data has already been fetched during run-time since everything is cached already at startup.

In the Lookup Table Cache three different types of requests are needed to the database (*get*, *getAccount* and *getAll*) which requires more code for database access. Additionally, there is logic for checking whether an entity has already been fetched during run-time. The provider is simple since no loading needs to be done at startup, but still the lines of code, i.e. the complexity of the solution, is higher than for the Full Load Cache .

As can be seen in table 4.4 the Hibernate Proxy Cache and the Asynchronous Preloading Cache solutions have a higher number of lines of code, but the two solutions have complexity in different parts of the code. The Asynchronous Preloading Cache has additional lines due to the orchestration of the preload threads, in both the cache class and the provider class. However, the database access is exactly the same as for the Lookup Table Cache. In the Hibernate Proxy Cache implementation, much of the cache logic for checking if something is already loaded is moved into the Hibernate framework. This removes some complexity, but the logic to set up data structures with proxies leads to about the same amount of lines as the Lookup Table Cache for the cache and provider while the complexity for database access is increased.

The lines of code corresponds fairly well to how complex we felt that each solution was to implement, but we can make some additional remarks on how difficult it is to understand each solution. The Full Load Cache is straightforward as there is not much logic that handles different scenarios during run-time. The Hibernate Proxy Cache solution uses Hibernate functionality to create proxies and unproxy them when needed. Therefore, the complexity completely depends on a developer's previous knowledge of the Hibernate framework and Hibernate proxies in particular. For example, there needs to be an understanding of how to make the solution thread-safe. The other implementations only use Hibernate for simple database queries but not for the cache logic. For the Lookup Table Cache and Asynchronous Preloading Cache, the logic mostly relays on standard data structures in Java, but when and how the data structure operations are executed needs to be understood. A developer with previous knowledge about Hibernate Proxies may find the Hibernate Proxy Cache solution easier to understand and maintain. The Asynchronous Preloading Cache is an extension of the Lookup Table Cache but the addition of running threads in parallel adds complexity to it. The possible issues with running multiple parallel threads need to be understood and handled to make the solution thread-safe. This makes the Asynchronous Preloading Cache more difficult to understand and maintain than the Lookup Table Cache .

5.4 Future Work

From the perspective of the case company more detailed testing of the functionality would be needed. When implementing our solutions, we have been running manual testing with command-line commands that we have implemented. But if any of our solutions get integrated into the real production code, automatic testing would be needed for efficiency. Additionally, the case company would need to add exception handling when the cache fails due to losing connection to the database for example.

This master thesis report focuses on efficient loading from a database at startup. We have not looked into cache eviction strategies, nor does the initial reference implementation have any eviction strategy implemented. However, if the case company expects huge tables to be loaded at startup it might be necessary to implement behavior to replace data when the cache is full, otherwise the program may run out of memory.

It should be noted that the Lazy Loading - Lookup Table and the Lazy Loading - Hibernate Proxy implementations seems to be easier to convert to a solution with an eviction strategy. In the lazy loading caches, we simply need to implement behavior to only cache the latest accessed data and evict the older data. In the Full Load Cache on the other hand there

is currently no functionality at all to cache data after the startup process, this makes eviction strategies impossible. In the Asynchronous Preloading solution the preloading threads could not load everything if we only store parts of the data in the cache. Because of this, the gain of parallelizing the preloading might not be as big as it currently is, depending on how much data is allowed in the cache.

The memory footprint of caching strategies can be investigated in more detail. To determine if an account has been loaded or not in the lazy loading solutions, some additional logic was introduced through an additional class. This means that more memory is used than in the Full Load Cache when all of the data has been loaded. On the other hand, lazy loading uses less memory as long as all of the data has not been requested since data is not fetched preemptively. The Asynchronous Preloading Cache runs queries in parallel, if Hibernate uses a significant amount of temporary memory this may lead to a higher peak in memory usage. In other words, the preloading threads could allocate additional temporary memory at the same time rather than one after the other. Allocating too much memory could lead to the program crashing.

Further, the scalability of the proposed solutions should be investigated. What happens if all of the caches in a big system with several micro-services use a cache based on lazy loading or multithreading at the same time? A hypothesis is that a multithreaded cache performs worse because hardware threads would have to be shared by a lot of software threads. For the Asynchronous Preloading Cache solution that uses multithreading the number of preload threads could be adjusted to one in order to use less resources. It may be that a cache based on lazy loading performs relatively better since the load on the network is flattened out. Lazy loading would spread out the database requests more compared to all micro-services requesting all of the data from all tables at startup.

In 4.1 we showed that the majority of time in the startup process was spent on the database requests. Due to the size of the data transferred we deemed the best approach to be a decrease of the size of the data transferred at startup as discussed in 5.1. However, many other things could improve performance although likely not to the same extent. For example, one could simplify the query, add an index or investigate the choice of DBMS, thread synchronization, data structures and ORM-framework. In particular, it would be interesting to investigate in further detail how different solutions perform in terms of run-time latency after the initial database load to the cache is finished. In this case, the overhead of synchronization methods and data structures could be more relevant.

5.5 Validity Threats

The validity of the research is discussed in this section with respect to validity threats described in [52].

5.5.1 Construct Validity

Construct validity refers to what degree of certainty the measurements collected correspond to the theory that is supposed to be analyzed [52]. In our case, the problem that we analyze is clearly and simply defined as the execution time of loading the caches which are easy to measure. Additionally, the independent variable is our four implementations, and this is easy

to change without any ambiguity. Therefore, we do not see any threats to construct validity of significant importance.

5.5.2 Conclusion Validity

Conclusion validity threats are concerned with inaccuracies that may impact the ability to draw correct conclusions [52]. There are a few threats to conclusion validity worth mentioning. However, we do not believe any of them has a significant impact considering conclusions are only drawn when the differences in the outcome are large.

Third-Party Frameworks

Several of our measurements depends on a few third-party libraries. The measurements on startup time conducted in 4.1, 4.3 and 4.6 are collected from timestamps in the logs and are therefore dependent on the accuracy in the log framework used. In table 4.2 we have used Oracle's *vsize* command to calculate the table size and the Hibernate Statistics API to calculate query execution time. We are not looking for high precision measurements in these experiments but rather an overview. Therefore, the exact precision of these third-party frameworks being used is not researched further. It is assumed that these third-party libraries provide enough accuracy for us to make general conclusions when there is a big difference in the measurements results, but we need to be careful not to make any firm conclusions when the differences in time or data size are too small.

When it comes to the time measurements from Hibernate Statistics API in particular, running with a configuration that enables Hibernate statistics can negatively affect performance. This means that in reality, our queries might be a bit quicker. Note that this is only relevant for the measurements that specifically use Hibernate Statistics API in 4.2, when running the other experiments we have disabled these statistics.

Regarding the table size calculations in table 4.2, there is some ambiguity. The size of the table is different depending on if we are referring to the data stored on disk, the data transferred over the network or the size of the objects stored in Hibernate. In this case, the size is referring to the pure sum of the size, as calculated by Oracle DB's *vsize* function, of all data elements in each table.

Varying System Load

In all of the conducted experiments, the load in the system at the moment of execution can have an effect. The machine running the experiments is only used by one developer at a time, therefore it is not the case that other heavy processes can be started randomly during the execution of our experiments. However, when running the experiments the whole application with several micro-services is running. This means that while our experiments only collect measurements on one of the micro-services, the other micro-services could possibly allocate a varying amount of resources and thereby affect our results.

Most of our experiments include database requests. When collecting statistics on the database requests there are two types of loads to consider. The database node is shared by a few developers but not all, this means that we can have a varying number of other requests hitting the database node while we are running experiments. As a consequence, the resources

at the database node, such as the CPU, might have to be shared. Any database requests, even if sent to another database node, can also contribute to a higher load on the network.

However, since the measurements are averaged over a number of experiment runs we do not consider the varying load in the system to be a concern. Additionally, the deviation between runs has also been quite small which indicates that random spikes in resource allocation by the other micro-services or hosts in the network are not a big problem.

Even though developers are working on setups that can share the same host as the database node, they are working with their own database running at that host. This means that there is no risk that other database requests are holding transaction locks on the data queried in the experiments.

Java Virtual Machine

As already mentioned in 3.3 there are many aspects of the JVM that can make performance measurements ambiguous or inaccurate. Importantly, the JIT compiler, the garbage collector and any other feature of the JVM will be included in the production environment. This means that disabling any of these features altogether may lead to skewing the results as they would be less realistic.

To combat the difference in results due to the JIT compiler, warm-up runs were excluded. But of course, we cannot be sure that the JIT compiler has no effect on the execution after the warm-up runs. However, as already described in section 3.3 warm-up runs are an established way to reduce inaccuracies in measurements introduced by the JIT compiler in Java.

In section 3.3 it is described how the Java garbage collection can affect performance measurements and in section 3.4 we describe that we trigger garbage collection between every experiment to combat this. However, this does not mean that the garbage collection cannot be triggered again during the experiment execution. Also note that the *System.gc()* call is not guaranteed to collect all unused objects. The method call will only trigger the garbage collector to make an effort toward recycling unused objects [45].

By doing so we make sure that all of the experiments can start without a lot of leftover garbage from the last experimental runs.

Lines of Code

We base our discussion of maintainability on the number of code lines. These measurements are very ambiguous as there are usually many ways to write a section of code with the same logical output. Therefore, this measurement should not be considered very accurate.

To achieve better accuracy regarding this measurement we have tried to keep our coding style consistent between different implementations. The case company uses plugins to check, when compiling, that the code follows the company's coding convention. These plugins have also led to our different implementations to use the same coding conventions.

5.5.3 Internal Validity

When the independent variable can be changed or influenced without the researcher's knowledge there is a threat to internal validity [52]. In our case, this is not relevant since our in-

dependent variables are our implementations or change in the code. Changes in the code cannot happen without our knowledge, it is instead controlled very explicitly.

5.5.4 External Validity

Threats regarding external validity mean that our findings can only be applied to a generalized setting to a limited extent [52]. There are some threats to external validity discussed below.

Scope Limitation

To begin with, the experiment setup and the interpretation of our results is of course affected by the scope that we have defined in 4.1.1. As previously stated, the product is a software solution and the hardware and network vary from customer to customer. Because of this, we cannot solve the issue with network or hardware solutions. But it is still possible that the measurements that we have collected would be different if we collected them at a customer's setup.

It would be theoretically possible to run our experiments on different hardware and network to get insight into which inefficiencies are apparent on different setups. On the other hand, it would require too much time and resources from the company to set up another type of machine just for this purpose. The test nodes that Nasdaq works with are meant to be representative of the customers' setup. Therefore we trust that the measurements collected on these test nodes are not significantly different from what they would be on the general customer setup.

Generated Data

As mentioned in 3.2.4 we generated the data in our database tables. The column data types are the same as in production and the values are inspired by production data, but we are not working with an exact copy of production data. The production data will of course also vary from customer to customer, but the generated data is perhaps not completely representative of the usual production data.

One relevant difference is that we do not know if the number of accounts relates to the number of entities in a way that is similar to the usual production data.

Another point of difference is that the generated data probably has less randomness. In some of the columns, the same value is generated for all rows, while these values would probably vary in production.

The relation between the size of the data and the number of rows may also be slightly different compared to what is normal in production. As an example, it may be that a NULL value is common in production for a particular column while the generated data store something else, or the other way around.

Chapter 6
Conclusion

The identified inefficiency with the initial solution, which we call the Full Load Cache, at the case company was that all data was loaded at startup. This would lead to a long startup time if the dataset would be relatively large. A faster startup time is especially important if the server crashes and everything needs to be restarted quickly. To alleviate this issue three solutions were proposed. What all of them have in common is to limit what has to be loaded during startup. However, these solutions affect the program during run-time in different ways instead.

The proposed solutions are called the Hibernate Proxy Cache, Lookup Table Cache, and Asynchronous Preloading Cache. The Hibernate Proxy Cache loads five columns at startup for each entity to fetch proxies of the real entity and insert them into the cache. When the cache service gets a request, it needs to check if the requested entities are proxies or not. If it is a proxy it will need to be converted to a real object.

Lookup Table Cache does not load anything from the database at startup. This means that it will have a minimal impact on the startup time. The cost of loading from the database is instead moved to run-time, but the cost of fetching data preemptively is removed. When a cache with this implementation gets a request it needs to check if it has loaded the data or not. If not, load it and insert it into the cache.

The Asynchronous Preloading Cache is an extension of the Lookup Table Cache. The difference is that this implementation starts threads at startup that load from the database in parallel. Until these preload threads are done, the cache works similarly to the Lookup Table Cache.

All the proposed solutions reduce the startup time compared to the initial solution. The Hibernate Proxy Cache takes a bit more time than the other two as it loads from the database. The Lookup Table Cache and Asynchronous Preloading Cache have an insignificant effect on the startup time compared with having no cache service at all.

The run-time experiments showed that both the Full Load Cache and Asynchronous Preloading Cache were fast in all scenarios. The Lookup Table Cache and Hibernate Proxy Cache took longer times in scenarios where data was accessed for the first time. In the Hiber-

nate Proxy solution, there is a flaw in that Hibernate creates one JDBC statement per proxy to be converted. This created an enormous overhead when the cache had to convert a large number of proxies. The Lookup Table Cache had more stable and predictable loading times.

In conclusion, the Hibernate Proxy Cache is not viable since the run-time request latency can be much larger than for the Lookup Table Cache . The original solution using an initial full load can lead to long-running startup processes with large data sets, as such it is not a viable solution if startup time is prioritized. The Lookup Table Cache is fast at startup but moves the latency into run-time, this cost is still a one-time concern. Experiments showed that the run-time latency for this cache is still small for smaller requests. The Asynchronous Preloading Cache has a minimal impact on the startup time and is fast during run-time after the preloading threads are done. This makes the Asynchronous Preloading Cache a solution with good potential, although it comes with a higher cost in complexity.

References

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[2] Paolo Atzeni, Christian S Jensen, Giorgio Orsi, Sudha Ram, Letizia Tanca, and Riccardo Torlone. The relational model is dead, sql is dead, and i don't feel so good myself. *ACM Sigmod Record*, 42(2):64–68, 2013.

[3] David E Avison, Francis Lau, Michael D Myers, and Peter Axel Nielsen. Action research. *Communications of the ACM*, 42(1):94–97, 1999.

[4] Christian Bauer and Gavin King. *Hibernate in action*, volume 1. Manning Greenwich CT, 2005.

[5] Emmanuel Bernard. Hibernate annotations. `https://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/`. Accessed: 2022-01-27.

[6] Saleem N Bhatti, Zahid H Abro, and Farzana Rufabro. Performance evaluation of java based object relational mapping tool. *Mehran University Research Journal of Engineering and Technology*, 32(2):159–166, 2013.

[7] Neepa Biswas, Anamitra Sarkar, and Kartick Chandra Mondal. Efficient incremental loading in etl processing for real-time data integration. *Innovations in Systems and Software Engineering*, 16(1):53–61, 2020.

[8] Heiko Böck. Java persistence api. In *The Definitive Guide to NetBeans™ Platform 7*, pages 315–320. Springer, 2012.

[9] Junkuo Cao, Weihua Wang, and Yuanzhong Shu. Comparison of pagination algorithms based-on large data sets. In *International Symposium on Information and Automation*, pages 384–389. Springer, 2010.

[10] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). *ACM Transactions on Database Systems (ToDS)*, 41(2):1–42, 2016.

[11] John Cieslewicz, Jonathan Berry, Bruce Hendrickson, and Kenneth A Ross. Realizing parallelism in database operations: insights from a massively multithreaded architecture. In *Proceedings of the 2nd international workshop on Data management on new hardware*, pages 4–es, 2006.

[12] Cisco. The zettabyte era: Trends and analysis. Technical report, Cisco, July 2016.

[13] Derek Colley and Clare Stanier. Identifying new directions in database performance tuning. *Procedia computer science*, 121:260–265, 2017.

[14] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. What's wrong with my benchmark results? studying bad practices in jmh benchmarks. *IEEE Transactions on Software Engineering*, 47(7):1452–1467, 2019.

[15] Yajnaseni Dash. An insight into parallel computing paradigm. In *2019 2nd International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*, volume 1, pages 804–808. IEEE, 2019.

[16] Neha Dhingra, Emad Abdelmoghith, and Hussien T Mouftah. Review on jpa based orm data persistence framework. *International Journal of Computer Theory and Engineering*, 9(5), 2017.

[17] Paulo Sergio Medeiros Dos Santos and Guilherme Horta Travassos. Action research can swing the balance in experimental software engineering. In *Advances in computers*, volume 83, pages 205–276. Elsevier, 2011.

[18] Vlad Mihalcea et al. Hibernate orm 5.6.5.final user guide. `https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html`. Accessed: 2022-02-07.

[19] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[20] Shahram Ghandeharizadeh and Ankit Mutha. An evaluation of the hibernate object-relational mapping for processing interactive social networking actions. In *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services*, pages 64–70, 2014.

[21] Md Morshadul Hasan, József Popp, and Judit Oláh. Current landscape and influence of big data on finance. *Journal of Big Data*, 7(1):1–17, 2020.

[22] Red Hat. Hibernate community documentation chapter 11. transactions and concurrency. `https://docs.jboss.org/hibernate/core/3.3/reference/en/html/transactions.html`. Accessed: 2022-03-14.

[23] Hibernate. Hibernate caching. `https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html#caching-entity`. Accessed: 2022-03-21.

[24] Hibernate. Hibernate criteria. `https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html#criteria`. Accessed: 2022-03-03.

[25] Hibernate. Hibernate. everything data. `https://hibernate.org/`. Accessed: 2022-01-27.

[26] Hibernate. Hibernate session lock. `https://docs.jboss.org/hibernate/orm/4.1/javadocs/org/hibernate/Session.html#lock(java.lang.Object,%20org.hibernate.LockMode)`. Accessed: 2022-03-03.

[27] Vojtech Horky, Peter Libic, Antonin Steinhauser, and Petr Tuma. Dos and don'ts of conducting performance measurements in java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 337–340, 2015.

[28] Sani M Isa et al. Hibernate orm query simplication using hibernate criteria extension (hce). In *2016 3rd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS)*, pages 23–28. IEEE, 2016.

[29] Wang Jing and Rui Fan. The research of hibernate cache technique and application of ehcache component. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 160–162. IEEE, 2011.

[30] Thomas Jörg and Stefan Deßloch. Towards generating etl processes for incremental loading. In *Proceedings of the 2008 international symposium on Database engineering & applications*, pages 101–110, 2008.

[31] Kamal Kakish and Theresa A Kraft. Etl evolution for real-time data warehousing. In *Proceedings of the Conference on Information Systems Applied Research ISSN*, volume 2167, page 1508, 2012.

[32] Hrvoje Kegalj, Dino Butorac, et al. Data access architecture in object-oriented applications using design patterns. *Journal of information and organizational sciences*, 27(2):81–91, 2003.

[33] Donald E Knuth. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

[34] Jens Kohler and Thomas Specht. A performance comparison between parallel and lazy fetching in vertically distributed cloud databases. In *2015 International Conference on Cloud Technologies and Applications (CloudTech)*, pages 1–6. IEEE, 2015.

[35] Dandan Li, Lu Han, and Yi Ding. Sql query optimization methods of relational database system. In *2010 Second International Conference on Computer Engineering and Applications*, volume 1, pages 557–560. IEEE, 2010.

[36] Vamsi Krishna Myalapalli, ASN Chakravarthy, and Keesari Prathap Reddy. Accelerating sql queries by unravelling performance bottlenecks in dbms engine. In *2015 International Conference on Energy Systems and Applications*, pages 7–12. IEEE, 2015.

[37] Vamsi Krishna Myalapalli and Pradeep Raj Savarapu. High performance sql. In *2014 Annual IEEE India Conference (INDICON)*, pages 1–6. IEEE, 2014.

[38] Nasdaq. About nasdaq. `https://www.nasdaq.com/about`. Accessed: 2022-01-26.

[39] Elizabeth J O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356, 2008.

[40] Oracle. Data concurrency and consistency. `https://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm`. Accessed: 2022-03-14.

[41] Oracle. Dynamic proxy classes. `https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html`. Accessed: 2022-03-07.

[42] Oracle. How parallel execution works. `https://docs.oracle.com/cd/E11882_01/server.112/e25523/parallel002.htm`. Accessed: 2022-03-14.

[43] Oracle. javax.persistence (java(tm) ee 8 specification apis). `https://javaee.github.io/javaee-spec/javadocs/javax/persistence/package-summary.html`. Accessed: 2022-01-28.

[44] Oracle. Oracle toplink. `https://www.oracle.com/middleware/technologies/top-link.html`. Accessed: 2022-01-28.

[45] Oracle. System (java platform se 7). `https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#gc()`. Accessed: 2022-05-17.

[46] Esther Pacitti. Parallel query processing. In *Encyclopedia of Database Systems*, pages 2038–2040. Springer US, 2009.

[47] Jarrett Rosenberg. Some misconceptions about lines of code. In *Proceedings fourth international software metrics symposium*, pages 137–142. IEEE, 1997.

[48] Spring. Spring data jpa - reference documentation - projections. `https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#projections.dtos`. Accessed: 2022-03-07.

[49] Alexandre Torres, Renata Galante, Marcelo S Pimenta, and Alexandre Jonatan B Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and software technology*, 82:1–18, 2017.

[50] Richard Trotter, Alex Chythlook, et al. High-frequency trading and internet crime: One cannot" trust the screen". *Journal of Financial Service Professionals*, 70(5), 2016.

[51] Gunter von Bultzingsloewen. Optimizing sql queries for parallel execution. *ACM SIGMOD Record*, 18(4):17–22, 1989.

[52] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[53] Qinglin Wu, Yanzhong Hu, and Yan Wang. Research on data persistence layer based on hibernate framework. In *2010 2nd International Workshop on Intelligent Systems and Applications*, pages 1–4. IEEE, 2010.

[54] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 800–810. IEEE, 2018.

[55] Joseph W Yoder, Ralph E Johnson, Quince D Wilson, et al. Connecting business objects to relational databases. In *Conference on the Pattern Languages of Programs*, volume 5, 1998.

[56] Jingren Zhou, John Cieslewicz, Kenneth A Ross, and Mihir Shah. Improving database performance on simultaneous multithreading processors, 2005.

Appendices

Appendix A
Code

Here we provide the code used throughout this thesis. Both the resulting implementations and some tools used to investigate performance. In order to not disclose any confidential information we have masked certain parts of the code. See the two listings below for how we have handled this.

```
/*The following means that one or more packages are imported in the real
    implementation but these are internal packages that are not relevant
    for the logic of our implementations.*/
import ...;
```

```
/*In the real implementation a certain type of entity is used with a more
    descriptive name. Since we don't want to disclose the type of data
    stored in the database we just refer to them as entities.
    Additionally, to differentiate between different kinds of data types
    we use entityA, entityB, etc.*/
this.entityACache = new HibernateLazyCache<>(entityProxyService,
EntityA.class);
this.entityBCache = new HibernateLazyCache<>(entityProxyService,
EntityB.class);
```

```
/*A similar approach was used to mask any column names. The real column
    names have been replaced with something like "columnX".*/
...
DetachedCriteria.forClass(entityType)
.add(Restrictions.eq("columnX", ColumnX.MASKED_VALUE))
.addOrder(Order.asc("key.entityNumber"))
.addOrder(Order.desc("key.columnY"))
...
```

A.1 SQL Script to Generate Data

Note that the the names of tables and columns have been masked to not expose Nasdaq's internal system.

```
-- Amount of rows to insert into specified table
define AMOUNT_OF_ROWS = 10000;

-- Table that should be used. Specify number
-- 0 = TABLE A
-- 1 = TABLE B
-- 2 = TABLE C
-- 3 = TABLE D
define TABLE_CHOICE = 0;

-- Table with all columns, shared and unique:
-- Some columns have been changed from NOT NULL to constants to
    make the code cleaner
CREATE TABLE tmp (
    COLUMN_A NUMBER(10, 0) NOT NULL,
    COLUMN_B DATE NOT NULL,
    COLUMN_C DATE NOT NULL,
    COLUMN_D NUMBER(10, 0) DEFAULT 4,
    COLUMN_E VARCHAR2(12 CHAR) DEFAULT 'value',
    COLUMN_F VARCHAR2(12 CHAR) DEFAULT 'value',
    COLUMN_G NUMBER(10, 0) NOT NULL,
    COLUMN_H NUMBER(20, 0) NOT NULL,
    COLUMN_I VARCHAR2(32 CHAR) DEFAULT 'value',
    COLUMN_J VARCHAR2(2 CHAR) DEFAULT 'value',
    COLUMN_K VARCHAR2(5 CHAR) DEFAULT 'value',
    COLUMN_L VARCHAR2(16 CHAR) DEFAULT 'value',
    COLUMN_M NUMBER(20, 0) NOT NULL,
    COLUMN_N VARCHAR2(64 CHAR) DEFAULT 'GENERATED',
    COLUMN_O NUMBER(10, 0) DEFAULT 5,
    COLUMN_P DATE NOT NULL,
    COLUMN_Q DATE DEFAULT DATE'2070-01-01',
    COLUMN_R VARCHAR2(12 BYTE) DEFAULT 'value',
    COLUMN_S DATE,
    COLUMN_T NUMBER(10, 0) DEFAULT 0,
    COLUMN_S_A NUMBER(20, 0) NOT NULL,
    COLUMN_S_B NUMBER(10, 0) DEFAULT 2,,
    COLUMN_S_C NUMBER(20, 0) NOT NULL,
    COLUMN_S_D NUMBER(10, 0) DEFAULT 0,
    COLUMN_S_E NUMBER(20, 0) DEFAULT 0,
    COLUMN_S_F NUMBER(10, 0) DEFAULT 0,
    COLUMN_S_G NUMBER(10, 0) DEFAULT 0,
    COLUMN_D_A NUMBER(10, 0) DEFAULT 0,
    COLUMN_C_A DATE NOT NULL,
    COLUMN_B_A DATE DEFAULT DATE'2070-01-01',
```

```
    COLUMN_B_B NUMBER(20, 0) DEFAULT 0,
    COLUMN_B_C VARCHAR2(34 BYTE),
    COLUMN_B_D NUMBER(10, 0) DEFAULT 0,
    COLUMN_B_E VARCHAR2(3 BYTE),
    COLUMN_B_F NUMBER(20, 0) DEFAULT 0,
    COLUMN_C_A VARCHAR2(12 BYTE) DEFAULT ' ',
    COLUMN_C_B NUMBER(10, 0) DEFAULT 9,
    COLUMN_C_C NUMBER(10, 0)
);

INSERT INTO tmp (COLUMN_A, COLUMN_B, COLUMN_C, COLUMN_G, COLUMN_H,
    COLUMN_M, COLUMN_P, COLUMN_S_A, COLUMN_S_C, COLUMN_C_A)
SELECT
    gen.COLUMN_A,
    gen.COLUMN_B,
    gen.COLUMN_B,
    DBMS_RANDOM.value(1, 10),
    DBMS_RANDOM.value(power(10, 18), power(10, 19)-power(10, 18)),
    gen.COLUMN_M,
    gen.COLUMN_B,
    DBMS_RANDOM.value(power(10, 3), power(10, 4)-1),
    gen.COLUMN_M,
    gen.COLUMN_B
FROM (
        SELECT
            LEVEL as COLUMN_A,
            TO_DATE(TRUNC(
               DBMS_RANDOM.value(TO_CHAR(DATE '2021-01-01','J')
                              ,TO_CHAR(DATE
                                  '2022-01-01','J'))),'J') as
                                  COLUMN_B,
            DBMS_RANDOM.value(0, 100000) as COLUMN_M
        FROM dual
        CONNECT BY LEVEL <= &AMOUNT_OF_ROWS
    ) gen;

DECLARE nbrOfInstances NUMBER;
BEGIN
    SELECT count(1) INTO nbrOfInstances FROM "DB 2"."TABLE E";
    FOR counter IN 1..nbrOfInstances
    LOOP
        UPDATE tmp
        SET
        (COLUMN_J, COLUMN_K, COLUMN_L) = (
            SELECT COLUMN_E_A, COLUMN_E_B, COLUMN_E_C
            FROM (
                SELECT COLUMN_E_A, COLUMN_E_B, COLUMN_E_C, rownum
                    as rn
                FROM (
```

```
                SELECT COLUMN_E_A, COLUMN_E_B, COLUMN_E_C
                FROM "DB 2"."TABLE E"
            )
        )
        WHERE rn=counter)
    WHERE counter = MOD(id, nbrOfInstances);
END LOOP;
IF &TABLE_CHOICE = 0 THEN
    INSERT INTO TABLE A
    SELECT
        COLUMN_B,
        COLUMN_C,
        COLUMN_D,
        COLUMN_E,
        COLUMN_F,
        COLUMN_G,
        COLUMN_H,
        COLUMN_I,
        COLUMN_J,
        COLUMN_K,
        COLUMN_L,
        COLUMN_M,
        COLUMN_N,
        COLUMN_O,
        COLUMN_P,
        COLUMN_C_A,
        COLUMN_Q,
        COLUMN_S,
        COLUMN_S_C,
        COLUMN_R,
        COLUMN_S_A,
        COLUMN_S_B,
        COLUMN_S_E,
        COLUMN_S_D,
        COLUMN_S_F,
        COLUMN_S_G,
        COLUMN_T
        FROM tmp;
ELSIF &TABLE_CHOICE = 1 THEN
    INSERT INTO TABLE B
    SELECT
        COLUMN_B,
        COLUMN_C,
        COLUMN_D,
        COLUMN_E,
        COLUMN_F,
        COLUMN_G,
        COLUMN_H,
        COLUMN_I,
```

```
            COLUMN_J,
            COLUMN_K,
            COLUMN_L,
            COLUMN_M,
            COLUMN_N,
            COLUMN_O,
            COLUMN_P,
            COLUMN_Q,
            COLUMN_B_A,
            COLUMN_B_B,
            COLUMN_S,
            COLUMN_S_C,
            COLUMN_S_E,
            COLUMN_R,
            COLUMN_S_A,
            COLUMN_S_B,
            COLUMN_S_D,
            COLUMN_S_F,
            COLUMN_S_G,
            COLUMN_B_C,
            COLUMN_B_D,
            COLUMN_T,
            COLUMN_B_E,
            COLUMN_B_F
        FROM tmp;
    ELSIF &TABLE_CHOICE = 2 THEN
        INSERT INTO TABLE C
        SELECT
            COLUMN_B,
            COLUMN_C,
            COLUMN_D,
            COLUMN_E,
            COLUMN_F,
            COLUMN_G,
            COLUMN_H,
            COLUMN_I,
            COLUMN_J,
            COLUMN_K,
            COLUMN_L,
            COLUMN_M,
            COLUMN_N,
            COLUMN_O,
            COLUMN_C_B,
            COLUMN_P,
            COLUMN_Q,
            COLUMN_S,
            COLUMN_C_C,
            COLUMN_R,
            COLUMN_C_A,
```

```
            COLUMN_T
        FROM tmp;
    ELSIF &TABLE_CHOICE = 3 THEN
        INSERT INTO TABLE D
        SELECT
            COLUMN_B,
            COLUMN_C,
            COLUMN_D,
            COLUMN_E,
            COLUMN_F,
            COLUMN_G,
            COLUMN_H,
            COLUMN_I,
            COLUMN_J,
            COLUMN_K,
            COLUMN_L,
            COLUMN_M,
            COLUMN_N,
            COLUMN_O,
            COLUMN_D_A,
            COLUMN_P,
            COLUMN_Q,
            COLUMN_S,
            COLUMN_S_C,
            COLUMN_R,
            COLUMN_S_A,
            COLUMN_S_B,
            COLUMN_S_D,
            COLUMN_T
        FROM tmp;
    END IF;
END;
/
DROP TABLE tmp;

-- DELETE GENERATED ROWS
-- DELETE FROM TABLE A WHERE COLUMN_N = 'GENERATED';
-- DELETE FROM TABLE B WHERE COLUMN_N = 'GENERATED';
-- DELETE FROM TABLE D WHERE COLUMN_N = 'GENERATED';
-- DELETE FROM TABLE C WHERE COLUMN_N = 'GENERATED';
```

A.2 Hibernate Statistics Command

```
final SessionFactory sf = getSessionFactory();
final Statistics stats = sf.getStatistics();

if (findParamValueBoolean(parameters, "clear")) {
    stats.clear();
    return null;
}

operationOutput.println(">>>>>>>>>>>SUMMARY<<<<<<<<<<<<<");
operationOutput.println("Is statistics enabled?: " +
    stats.isStatisticsEnabled() + "\n");
final long preparedStatements = stats.getPrepareStatementCount();
operationOutput.println("Registered " + preparedStatements + " JDBC
    statements.");
operationOutput.println("Registered " + stats.getQueryExecutionCount() +
    " queries, with " + stats.getQueryCacheHitCount() + " query cache
    hits.");
operationOutput.println("Registered " +
    stats.getNaturalIdQueryExecutionCount() + " NaturalId queries");
operationOutput.println("The slowest query executed in " +
    stats.getQueryExecutionMaxTime() + " ms");
operationOutput.println("Query string for the slowest query: " +
    stats.getQueryExecutionMaxTimeQueryString());
operationOutput.println("The number of collections loaded (fetching and
    cache) " + stats.getCollectionLoadCount());
operationOutput.println("The number of entities loaded " +
    stats.getEntityLoadCount());

if (findParamValueBoolean(parameters, "showQueries")) {
    operationOutput.println("\n>>>>>>>>>>>Query details<<<<<<<<<<<<<");
    final String[] queries = stats.getQueries();
    for (String q: queries) {
        final QueryStatistics qs = stats.getQueryStatistics(q);
        operationOutput.println("\nQuery: " + q);
        operationOutput.println("Executed " + qs.getExecutionCount() + "
            times with an average execution time of " +
            qs.getExecutionAvgTime() + " ms");
    }
}

return null;
```

A.3 Mutual Code

A.3.1 AccountIndex

```
package ...;

import java.util.HashSet;
import java.util.Set;

import ...;

public class AccountIndex {
    private final Set<Long> entityKeys = new HashSet<>();
    private boolean loadedFromDatabase = false;

    public Set<Long> getEntityKeys() {
        return entityKeys;
    }

    public void addEntityKey(final long entityKey) {
        entityKeys.add(entityKey);
    }

    public void removeEntityKey(final long entityKey) {
        entityKeys.remove(entityKey);
    }

    public boolean getLoadedFromDatabase() {
        return loadedFromDatabase;
    }

    public void setLoadedFromDatabase() {
        loadedFromDatabase = true;
    }

    public static IndexKey createIndexKey(final Collateral<?> entity) {
        return createIndexKey(entity.getAccountInfo());
    }

    public static IndexKey createIndexKey(final String accountInfo) {
        return new IndexKey(accountInfo);
    }
}
```

A.3.2 EntityService

```java
package ...;

import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import ...;

/**
 * This class contain the database transaction support for Entity.
 */
@SuppressWarnings("unused")
@Service
public class EntityService {
    private EntityDao entityDao;

    @Autowired
    public void setEntityDao(final EntityDao dao) {
        this.entityDao = dao;
    }

    @Transactional(readOnly = true)
    public <T extends Entity<?>> List<T> findAllActive(final Class<T>
        entityType) {
        return entityDao.findAllActive(entityType);
    }

    @Transactional(readOnly = true)
    public <T extends Entity<?>> List<T> findActive(final String
        accountInfo, final Class<T> entityType) {
        return entityDao.findActive(accountInfo, entityType);
    }

    @Transactional(readOnly = true)
    public <T extends Entity<?>> T findActive(final long entityNumber,
        final Class<T> entityType) {
        return entityDao.findActive(entityNumber, entityType);
    }
}
```

A.3.3 EntityDaoHibernate

```java
package ...;

import java.util.List;

import org.apache.log4j.Logger;
import org.hibernate.criterion.DetachedCriteria;
import org.hibernate.criterion.Order;
import org.hibernate.criterion.Restrictions;
import org.springframework.orm.hibernate5.support.HibernateDaoSupport;
import org.springframework.stereotype.Component;

import ...;

@SuppressWarnings("unchecked")
@Component("entityDao")
public class EntityDaoHibernate extends HibernateDaoSupport implements
    EntityDao {
    private final Logger logger =
        Logger.getLogger(EntityDaoHibernate.class);

    @Override
    public <T extends Entity<?>> List<T> findAllActive(final Class<T>
        entityType) {
        return DetachedCriteria.forClass(entityType)
            .add(Restrictions.eq("columnX", ColumnX.MASKED_VALUE))
            .addOrder(Order.asc("key.entityNumber"))
            .addOrder(Order.desc("key.columnY"))
            .getExecutableCriteria(getSessionFactory().getCurrentSession())
            .list();
    }

    @Override
    public <T extends Entity<?>> List<T> findActive(final String
        accountInfo, final Class<T> entityType) {
        return DetachedCriteria.forClass(entityType)
            .add(Restrictions.eq("accountInfo", accountInfo)))
            .add(Restrictions.eq("columnX", ColumnX.MASKED_VALUE))
            .getExecutableCriteria(getSessionFactory().getCurrentSession())
            .list();
    }

    @Override
    public <T extends Entity<?>> T findActive(final long entityNumber,
        final Class<T> entityType) {
        final List<T> result = DetachedCriteria.forClass(entityType)
            .add(Restrictions.eq("key.entityNumber", entityNumber))
            .add(Restrictions.eq("columX", ColumnX.MASKED_VALUE))
```

```
            .getExecutableCriteria(getSessionFactory().getCurrentSession())
            .list();

        if (result.size() > 1) {
            logger.warn("Query return more that one active entity !!");
            for (T entity : result) {
                logger.warn(entity);
            }
        }

        return result.isEmpty() ? null : result.get(0);
    }
}
```

A.3.4 EntityProxyService

```
package ...;

import java.util.List;

import ...;

@Service
public class EntityProxyService {
    private EntityProxyDao entityProxyDao;

    @Autowired
    public void setEntityProxyDao(final EntityProxyDao dao) {
        this.entityProxyDao = dao;
    }

    @Transactional(readOnly = true)
    public <T extends Entity<?>> List<EntityProxyInfoDTO<T>>
        loadAllProxy(final Class<T> entityType) {
        return entityProxyDao.loadAllProxy(entityType);
    }

    @Transactional(readOnly = true)
    public <T extends Entity<?>> void unproxy(final List<T> proxyList) {
        entityProxyDao.unproxy(proxyList);
    }

    @Transactional(readOnly = true)
    public <T extends Entity<?>> void unproxy(final T proxy) {
        entityProxyDao.unproxy(proxy);
    }
}
```

A.3.5 EntityProxyDaoHibernate

```
package ...;

import java.util.List;

import org.hibernate.LockOptions;
import org.hibernate.Session;
import org.hibernate.criterion.DetachedCriteria;
import org.hibernate.criterion.ProjectionList;
import org.hibernate.criterion.Projections;
import org.hibernate.criterion.Restrictions;
import org.hibernate.transform.Transformers;
import org.springframework.orm.hibernate5.support.HibernateDaoSupport;
import org.springframework.stereotype.Component;

import ...;

@SuppressWarnings("unchecked")
@Component("entityProxyDao")
public class EntityProxyDaoHibernate extends HibernateDaoSupport
    implements EntityProxyDao {
    @Override
    public <T extends AbstractEntity<?>> List<EntityProxyInfoDTO<T>>
        loadAllProxy(final Class<T> entityType) {
        final List<EntityProxyInfoDTO<T>> proxyInfoList =
            loadAllActiveProxyInfo(entityType);
        final Session session = getSessionFactory().getCurrentSession();
        for (EntityProxyInfoDTO<T> proxyInfo : proxyInfoList) {
            final T proxy = session.load(entityType, proxyInfo.getKey());
            proxyInfo.setEntity(proxy);
        }

        return proxyInfoList;
    }

    @Override
    public <T> void unproxy(final List<T> proxyList) {
        final Session.LockRequest lockRequest =
            getSessionFactory().getCurrentSession().buildLockRequest(LockOptions.NONE);
        for (final T proxy : proxyList) {
            lockRequest.lock(proxy);
        }
    }

    @Override
    public <T> void unproxy(final T proxy) {
        getSessionFactory().getCurrentSession()
        .buildLockRequest(LockOptions.NONE).lock(proxy);
```

```
    }

    private <T extends AbstractEntity<?>> List<EntityProxyInfoDTO<T>>
        loadAllActiveProxyInfo(final Class<T> entityType) {
        final ProjectionList projectionList = Projections.projectionList()
            .add(Projections.property("key"), "key")
            .add(Projections.property("accountInfo"), "accountInfo");
        return DetachedCriteria.forClass(entityType)
            .add(Restrictions.eq("columnX", ColumnX.MASKED_VALUE))
            .setProjection(projectionList)
            .setResultTransformer(Transformers.aliasToBean(EntityProxyInfoDTO.class))
            .getExecutableCriteria(getSessionFactory().getCurrentSession())
            .list();
    }
}
```

A.3.6 EntityProxyInfoDTO

```
package ...;

import ...;

public class EntityProxyInfoDTO<T extends AbstractEntity<?>> {
    private EntityKey key;
    private String accountInfo;
    private T entity;

    public EntityKey getKey() {
        return key;
    }

    public void setKey(final EntityKey key) {
        this.key = key;
    }

    public String getAccountInfo() {
        return accountInfo;
    }

    public void setAccountInfo(final String accountInfo) {
        this.accountInfo = accountInfo;
    }

    public T getEntity() {
        return entity;
    }

    public void setEntity(final T entity) {
        this.entity = entity;
    }
}
```

A.4 Hibernate Proxy Cache

A.4.1 Cache

```
package ...;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.function.Predicate;
import java.util.stream.Collectors;

import org.apache.log4j.Logger;
import org.hibernate.Hibernate;

import ...;

public class HibernateLazyCache<T extends Entity<R>, R> implements
    Cache<T, R> {
    private static final Logger LOGGER =
        Logger.getLogger(HibernateLazyCache.class);
    private final EntityProxyService entityProxyService;
    private final Map<Long, T> cache = new HashMap<>();
    private final Map<IndexKey, AccountIndex> accountIndexMap = new
        HashMap<>();
    private final Class<T> entityType;

    public HibernateLazyCache(final EntityProxyService
        entityProxyService, final Class<T> entityType) {
        this.entityProxyService = entityProxyService;
        this.entityType = entityType;
    }

    @Override
    public synchronized void clear() {
        cache.clear();
        accountIndexMap.clear();
    }

    @Override
    public synchronized void initCache() {
        final List<EntityProxyInfoDTO<T>> entityProxyList = loadAllProxy();
        for (final EntityProxyInfoDTO<T> entityProxyDTO: entityProxyList) {
            insertProxy(entityProxyDTO);
        }
    }
```

```
@Override
public synchronized List<T> getAllActive() {
    final List<T> entityList = new ArrayList<>(cache.values());
    unproxy(entityList);
    return entityList;
}

@Override
public synchronized T getActive(final long entityNumber) {
    final T entity = cache.get(entityNumber);
    unproxy(entity);
    return entity;
}

@Override
public synchronized List<T> getActive(final String accountInfo) {
    final IndexKey indexKey = AccountIndex.createIndexKey(accountInfo);
    if (!accountIndexMap.containsKey(indexKey)) {
        return new ArrayList<>();
    }

    final List<T> entityList = accountIndexMap
        .get(indexKey)
        .getEntityKeys()
        .stream()
        .map(cache::get)
        .collect(Collectors.toList());
    unproxy(entityList);
    return entityList;
}

@Override
public synchronized void add(final T entity) {
    final long entityNumber = entity.getEntityNumber();
    cache.put(entityNumber, entity);
    final IndexKey indexKey = AccountIndex.createIndexKey(entity);
    if (!accountIndexMap.containsKey(indexKey)) {
        accountIndexMap.put(indexKey, new AccountIndex());
    }

    final AccountIndex accountIndex = accountIndexMap.get(indexKey);
    accountIndex.addEntityKey(entityNumber);
}

@Override
public synchronized void remove(final T entity) {
    final long entityNumber = entity.getEntityNumber();
    cache.remove(entityNumber);
    final IndexKey indexKey = AccountIndex.createIndexKey(entity);
```

```
        final AccountIndex accountIndex = accountIndexMap.get(indexKey);
        accountIndex.removeEntityKey(entityNumber);
    }

    @Override
    public synchronized void update(final T entity) {
        final long entityNumber = entity.getEntityNumber();
        cache.put(entityNumber, entity);
    }

    private synchronized void unproxy(final List<T> entityList) {
        final List<T> proxyList = entityList
            .stream()
            .filter(Predicate.not(Hibernate::isInitialized))
            .collect(Collectors.toList());
        if (!proxyList.isEmpty()) {
            entityProxyService.unproxy(proxyList);
        }
    }

    private synchronized void unproxy(final T entity) {
        if (!Hibernate.isInitialized(entity)) {
            entityProxyService.unproxy(entity);
        }
    }

    private List<EntityProxyInfoDTO<T>> loadAllProxy() {
        return entityProxyService.loadAllProxy(entityType);
    }

    private synchronized void insertProxy(final EntityProxyInfoDTO<T>
        entityProxyInfoDTO) {
        final long entityNumber =
            entityProxyInfoDTO.getKey().getEntityNumber();
        final T entity = entityProxyInfoDTO.getCollateral();
        final String accountInfo = entityProxyInfoDTO.getAccountInfo();
        cache.put(entityNumber, entity);
        final IndexKey indexKey = AccountIndex.createIndexKey(accountInfo);
        if (!accountIndexMap.containsKey(indexKey)) {
            accountIndexMap.put(indexKey, new AccountIndex());
        }

        final AccountIndex accountIndex = accountIndexMap.get(indexKey);
        accountIndex.addEntityKey(entityNumber);
    }
}
```

A.4.2 Provider

```java
package ...;

import org.apache.log4j.Logger;

import ...;

public class HibernateLazyCacheServiceProvider extends ... {
    private static final Logger LOGGER =
        Logger.getLogger(HibernateLazyCacheServiceProvider.class);

    public HibernateLazyCacheServiceProvider(final Service service, final
        String instanceName, final String serviceName) {
        super(service, instanceName, serviceName);
        final EntityProxyService entityProxyService =
            FrameworkService.getDefault(icore).getBean("entityProxyService");
        this.entityACache = new HibernateLazyCache<>(entityProxyService,
            EntityA.class);
        this.entityBCache = new HibernateLazyCache<>(entityProxyService,
            EntityB.class);
        this.entityCCache = new HibernateLazyCache<>(entityProxyService,
            EntityC.class);
        this.entityDCache = new HibernateLazyCache<>(entityProxyService,
            EntityD.class);
    }

    @Override
    public void resetCache() {
        entityACache.clear();
        entityBCache.clear();
        entityCCache.clear();
        entityDCache.clear();
        entityACache.initCache();
        entityBCache.initCache();
        entityCCache.initCache();
        entityDCache.initCache();
    }

    @Override
    public void start() {
        super.start();
        this.entityBCache.initCache();
        this.entityDCache.initCache();
        this.entityACache.initCache();
        this.entityCCache.initCache();
        LOGGER.info("caches initialized");
    }
}
```

A.5 Lookup Table Cache

A.5.1 Cache

```
package ...;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.stream.Collectors;

import org.apache.log4j.Logger;

import ...;

public class LazyCache<T extends Entity<R>, R> implements Cache<T, R> {
    private static final Logger LOGGER =
        Logger.getLogger(LazyCache.class);
    protected EntityService entityService;
    protected final Map<Long, T> cache = new HashMap<>();
    protected final Map<IndexKey, AccountIndex> accountIndexMap = new
        HashMap<>();
    protected boolean hasLoadedEverything = false;
    protected final Class<T> entityType;

    public LazyCache(final EntityService entityService, final Class<T>
        entityType) {
        this.entityService = entityService;
        this.entityType = entityType;
    }

    @Override
    public synchronized void clear() {
        cache.clear();
        accountIndexMap.clear();
        hasLoadedEverything = false;
    }

    @Override
    public void initCache() {
        LOGGER.info("No initialization");
    }

    @Override
    public synchronized List<T> getAllActive() {
        if (!hasLoadedEverything) {
            loadAll();
```

```
        hasLoadedEverything = true;
    }

    return cache
            .values()
            .stream()
            .filter(Objects::nonNull)
            .collect(Collectors.toList());
}

@Override
public synchronized List<T> getActive(final String accountInfo) {
    final IndexKey indexKey = AccountIndex.createIndexKey(accountInfo);
    if (!accountIndexMap.containsKey(indexKey)) {
        accountIndexMap.put(indexKey, new AccountIndex());
    }

    final AccountIndex accountIndex = accountIndexMap.get(indexKey);
    if (!accountIndex.getLoadedFromDatabase()) {
        final List<T> entityList =
            entityService.findActive(accountInfo, entityType);
        for (final T entity : entityList) {
            add(entity);
        }

        accountIndex.setLoadedFromDatabase();
    }

    return accountIndex
            .getEntityKeys()
            .stream()
            .map(cache::get)
            .filter(Objects::nonNull)
            .collect(Collectors.toList());
}

@Override
public synchronized T getActive(final long entityNumber) {
    if (!cache.containsKey(entityNumber)) {
        final T entity = entityService.findActive(entityNumber,
            entityType);
        if (entity == null) {
            cache.put(entityNumber, null);
            return null;
        }

        add(entity);
    }
```

```
        return cache.get(entityNumber);
    }

    @Override
    public synchronized void add(final T entity) {
        final long entityNumber = entity.getEntityNumber();
        if (!cache.containsKey(entityNumber)) {
            cache.put(entityNumber, entity);

            final IndexKey indexKey = AccountIndex.createIndexKey(entity);
            if (!accountIndexMap.containsKey(indexKey)) {
                accountIndexMap.put(indexKey, new AccountIndex());
            }

            accountIndexMap.get(indexKey).addEntityKey(entityNumber);
        }
    }

    @Override
    public synchronized void remove(final T entity) {
        final long entityNumber = entity.getEntityNumber();
        cache.put(entityNumber, null);
        final IndexKey indexKey = AccountIndex.createIndexKey(entity);
        if (accountIndexMap.containsKey(indexKey)) {
            accountIndexMap.get(indexKey).removeEntityKey(entityNumber);
        }
    }

    @Override
    public synchronized void update(final T entity) {
        final long entityNumber = entity.getEntityNumber();
        cache.put(entityNumber, entity);
    }

    private synchronized void loadAll() {
        final List<T> entityList = entityService.findAllActive(entityType);
        for (final T entity : entityList) {
            add(entity);
        }

        for (final AccountIndex accountIndex : accountIndexMap.values()) {
            accountIndex.setLoadedFromDatabase();
        }
    }
}
```

A.5.2 Provider

```
package ...;

import org.apache.log4j.Logger;

import ...;

public class LazyCacheServiceProvider extends ... {
    private static final Logger LOGGER =
        Logger.getLogger(LazyCacheServiceProvider.class);

    public LazyCacheServiceProvider(final Service service, final String
        instanceName, final String serviceName) {
        super(service, instanceName, serviceName);
        final EntityService entityService =
            FrameworkService.getDefault(service).getBean("entityService");
        this.entityACache = new LazyCache<>(entityService, EntityA.class);
        this.entityBCache = new LazyCache<>(entityService, EntityB.class);
        this.entityCCache = new LazyCache<>(entityService, EntityC.class);
        this.entityDCache = new LazyCache<>(entityService, EntityD.class);
    }

    @Override
    public void resetCache() {
        entityACache.clear();
        entityBCache.clear();
        entityCCache.clear();
        entityDCache.clear();
    }

    @Override
    public void start() {
        super.start();
        LOGGER.info("No initial load.");
    }
}
```

A.6 Asynchronous Preloading Cache

A.6.1 Cache

```
package ...;

import java.util.HashSet;
import java.util.List;
import java.util.Objects;
import java.util.Set;
import java.util.stream.Collectors;

import org.apache.log4j.Logger;

import ...;

public class AsyncPreloadCache<T extends Entity<R>, R> extends
    LazyCache<T, R> {
    private static final Logger LOGGER =
        Logger.getLogger(AsyncPreloadCache.class);
    private Set<InitTask> activeInitTasks = new HashSet<>();

    public AsyncPreloadCache(final EntityService entityService, final
        Class<T> entityType) {
        super(entityService, entityType);
    }

    @Override
    public synchronized void clear() {
        //Invalidate all active init tasks
        activeInitTasks = new HashSet<>();
        super.clear();
    }

    public Runnable getInitTask() {
        final InitTask task = new InitTask();
        synchronized (this) {
            if (hasLoadedEverything) {
                LOGGER.warn("Cache already initialized. New init task is
                    redundant.");
            }

            activeInitTasks.add(task);
            if (activeInitTasks.size() > 1) {
                LOGGER.warn("More than one init task active.");
            }
        }
```

```
        return task;
    }

    private final class InitTask implements Runnable {
        @Override
        public void run() {
            final List<T> entityList =
                entityService.findAllActive(entityType);
            synchronized (AsyncPreloadCache.this) {
                if (activeInitTasks.contains(this)) {
                    for (final T entity : entityList) {
                        add(entity);
                    }

                    for (AccountIndex accountIndex :
                        accountIndexMap.values()) {
                        accountIndex.setLoadedFromDatabase();
                    }

                    hasLoadedEverything = true;
                    AsyncPreloadCache.this.notifyAll();
                    LOGGER.info("Init task (" + entityType.getSimpleName()
                        + ") finished. Exiting thread.");
                } else {
                    LOGGER.info("Init task (" + entityType.getSimpleName()
                        + ") cancelled. Exiting thread.");
                }
            }

            activeInitTasks.remove(this);
        }
    }

    @Override
    public synchronized List<T> getAllActive() {
        waitInitCompletion();
        return super.getAllActive();
    }

    @Override
    public List<T> getActive(final String accountInfo) {
        final IndexKey indexKey = AccountIndex.createIndexKey(accountInfo);
        final boolean alreadyLoaded;
        final AccountIndex accountIndex;
        synchronized (this) {
            if (!accountIndexMap.containsKey(indexKey)) {
                accountIndexMap.put(indexKey, new AccountIndex());
            }
```

```
            accountIndex = accountIndexMap.get(indexKey);
            alreadyLoaded = accountIndex.getLoadedFromDatabase();
        }

        List<T> entityList = null;
        if (!alreadyLoaded) {
            entityList = entityService.findActive(accountInfo, entityType);
        }

        synchronized (this) {
            if (!accountIndex.getLoadedFromDatabase() && entityList !=
                null) {
                // Check again since another thread could have finished
                    before this thread.
                accountIndex.setLoadedFromDatabase();
                for (final T entity : entityList) {
                    add(entity);
                }
            }

            return accountIndex
                    .getEntityKeys()
                    .stream()
                    .map(cache::get)
                    .filter(Objects::nonNull)
                    .collect(Collectors.toList());
        }
    }

    @Override
    public T getActive(final long entityNumber) {
        final boolean alreadyLoaded;
        synchronized (this) {
            alreadyLoaded = cache.containsKey(entityNumber);
        }

        T entity = null;
        if (!alreadyLoaded) {
            entity = entityService.findActive(entityNumber, entityType);
        }

        synchronized (this) {
            if (!cache.containsKey(entityNumber)) {
                // Check again since another thread could have finished
                    before this thread.
                if (entity == null) {
                    LOGGER.info(entityType.getSimpleName() + " with entity
                        number " + entityNumber + " does not exist.");
                    cache.put(entityNumber, null);
```

```
                return null;
            }

            add(entity);
        }

        return cache.get(entityNumber);
    }
}

private synchronized void waitInitCompletion() {
    if (!hasLoadedEverything && activeInitTasks.size() < 1) {
        LOGGER.error("No active init tasks. Will wait until one is
            created.");
    }

    try {
        while (!hasLoadedEverything) {
            wait();
        }
    } catch (InterruptedException e) {
        LOGGER.error("Thread interrupted while waiting for init
            task.");
    }
}
}
```

A.6.2 Provider

```
package ...;

import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;

import ...;

public class AsyncPreloadCacheServiceProvider extends ... {
    private static final Logger LOGGER =
        Logger.getLogger(AsyncPreloadCacheServiceProvider.class);
    @Autowired
    private ThreadPoolTaskExecutor taskExecutor;
    private final Runnable[] taskList;

    public AsyncPreloadCacheServiceProvider(final Service service, final
        String instanceName, final String serviceName) {
        super(service, instanceName, serviceName);
        final FrameworkService frameworkService =
            FrameworkService.getDefault(service);
        final EntityService entityService =
            frameworkService.getBean("entityService");

        final AsyncPreloadCache<EntityA, EntityARec>
            entityAAsyncPreloadCache = new
            AsyncPreloadCache<>(entityService, EntityA.class);
        this.entityACache = entityAAsyncPreloadCache;

        final AsyncPreloadCache<EntityB, EntityBRec>
            entityBAsyncPreloadCache = new
            AsyncPreloadCache<>(entityService, EntityB.class);
        this.entityBCache = entityBAsyncPreloadCache;

        final AsyncPreloadCache<EntityC, EntityCRec>
            entityCAsyncPreloadCache = new
            AsyncPreloadCache<>(entityService, EntityC.class);
        this.entityCCache = entityCAsyncPreloadCache;

        final AsyncPreloadCache<EntityD, EntityDRec>
            entityDAsyncPreloadCache = new
            AsyncPreloadCache<>(entityService, EntityD.class);
        this.entityDCache = entityDAsyncPreloadCache;

        taskList = new Runnable[]{
            entityAAsyncPreloadCache.getInitTask(),
            entityBAsyncPreloadCache.getInitTask(),
            entityCAsyncPreloadCache.getInitTask(),
```

```
            entityDAsyncPreloadCache.getInitTask()
        };
    }

    @Override
    public void resetCache() {
    }

    @Override
    public void start() {
        super.start();
        LOGGER.info("No initial load.");
        for (Runnable task : taskList) {
            taskExecutor.execute(task);
        }
    }
}
```

A.7 Experiment Code

```
package ...;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;

import org.apache.log4j.Logger;

import ...

public class CacheExperiment extends ... {
    private static final Logger LOGGER =
        Logger.getLogger(CacheExperiment.class);
    private static final String COMMAND = "Cache Experiment";
    private static final String DESCRIPTION = "Experiments on the cache
        implementations.";
    protected final List<Long> numbers = new ArrayList<>();
    protected final Map<IndexKey, AccountInfo> accountMap = new
        HashMap<>();
    private final Service service;

    public CacheExperiment(final Service service) {
        super(COMMAND, DESCRIPTION,
            new Parameter("cacheService", "Cache service to use. Initial,
                Lookup, Proxy or Async.", false, null));
        this.service = service;
    }

    @Override
    public Object execute(final OperationOutput operationOutput, final
        Collection<ValueParameter> parameters) throws Exception {
        final String cacheServiceName = findParamValueString(parameters,
            "cacheService");
        final String experiment = findParamValueString(parameters,
            "experiment");
        final CacheService cacheService =
            getCacheService(cacheServiceName);
        if (cacheService == null) {
            operationOutput.println("Invalid cache service. Use Initial,
                Lookup, Proxy or Async");
            return null;
        }
```

```
    LOGGER.info("Setting up experiment environment...");
    setupBeforeExperiment(cacheService);
    LOGGER.info("Setup of experiment environment done.");

    // GET ALL
  executeExperiment("getAll from one table, 0% previously accessed",
      new GetAllFromOneTable(0), cacheServiceName, cacheService);
  executeExperiment("getAll from one table, 50% previously accessed",
      new GetAllFromOneTable(0.5), cacheServiceName, cacheService);
  executeExperiment("getAll from one table, 90% previously accessed",
      new GetAllFromOneTable(0.9), cacheServiceName, cacheService);
  executeExperiment("getAll from one table, 100% previously
      accessed", new GetAllFromOneTable(1), cacheServiceName,
      cacheService);

    // GET ACCOUNT
  executeExperiment("getAccount from one table, 0% previously
      accessed", new GetAccountFromOneTable(true), cacheServiceName,
      cacheService);
  executeExperiment("getAccount from one table, 100% previously
      accessed", new GetAccountFromOneTable(false), cacheServiceName,
      cacheService);

    // GET
  executeExperiment("get from one table, 0% previously accessed", new
      GetFromOneTable(true), cacheServiceName, cacheService);
  executeExperiment("get from one table, 100% previously accessed",
      new GetFromOneTable(false), cacheServiceName, cacheService);

    return null;
}

private void executeExperiment(final String experimentName, final
    Experiment experiment, final String cacheServiceName, final
    CacheService cacheService) {
    final int runs = 12;
    LOGGER.info("----------------------------------");
    LOGGER.info(cacheServiceName + " - Executing experiment " +
        experimentName + ":");
    for (int i = 0; i < runs; i++) {
        LOGGER.info(cacheServiceName + " - Setting up run " + i +
            "...");
        cacheService.resetCache();
        System.gc();
        experiment.setup(cacheService);
        experiment.startExperiment(cacheService, cacheServiceName, i);
        experiment.useReturned();
    }
```

```
    LOGGER.info("---------------------------------");
}

private CacheService getCacheService(final String cacheService) {
    switch (cacheService.toLowerCase()) {
    case "initial":
        return CacheService.getDefault(service, "initial");
    case "lookup":
        return CacheService.getDefault(service, "lookup");
    case "proxy":
        return CacheService.getDefault(service, "proxy");
    case "async":
        return CacheService.getDefault(service, "async");
    default:
        return null;
    }
}

private void setupBeforeExperiment(final CacheService cacheService) {
    final List<Entity> entityList = cacheService.getAllActiveEntity();
    for (Entity entity : entityList) {
        numbers.add(entity.getNumber());
        final IndexKey indexKey = new
            IndexKey(entity.getAccountInformation());
        if (!accountMap.containsKey(indexKey)) {
            accountMap.put(indexKey, new AccountInfo(entity));
        }
    }
}

private <T extends AbstractEntity<?>> void setOnAllEntity(final
    List<T> entityList) {
    final int randomInt = new Random().nextInt();
    for (AbstractEntity<?> entity : entityList) {
        entity.setInfo(randomInt + " Experiment done.");
    }
}

public abstract static class Experiment {
    abstract void execute(CacheService cacheService);

    abstract void setup(CacheService cacheService);

    abstract void useReturned();

    public void startExperiment(final CacheService cacheService, final
        String cacheServiceName, final int i) {
        LOGGER.info(cacheServiceName + " - Starting run " + i + "...");
```

```
        final long start = System.nanoTime();
        this.execute(cacheService);
        final long stop = System.nanoTime();
        LOGGER.info(cacheServiceName + " - Stopping run " + i + "...");
        final long elapsedTime = stop - start;
        LOGGER.info(cacheServiceName + " - Run " + i + ": " +
            elapsedTime + " nanoseconds.");
    }
}

public class GetAllFromOneTable extends Experiment {
    private final double fractionToLoad;
    private List<Entity> entityList;

    public GetAllFromOneTable(final double fractionToLoad) {
        this.fractionToLoad = fractionToLoad;
    }

    @Override
    public void execute(final CacheService cacheService) {
        entityList = cacheService.getAllActiveEntity();
    }

    @Override
    public void setup(final CacheService cacheService) {
        if (fractionToLoad != 0 && fractionToLoad < 1) {
            final int size = numbers.size();
            Collections.shuffle(numbers);
            for (int i = 0; i < size * fractionToLoad; i++) {
                cacheService.getActiveEntity(numbers.get(i));
            }
        }

        if (fractionToLoad == 1) {
            cacheService.getAllActiveEntity();
        }
    }

    @Override
    public void useReturned() {
        setOnAllEntity(entityList);
        entityList.clear();
    }
}

public class GetFromOneTable extends Experiment {
    private final List<Entity> entityList = new ArrayList<>();
    private final boolean nothingLoaded;
    private long number;
```

```
    public GetFromOneTable(final boolean nothingLoaded) {
        this.nothingLoaded = nothingLoaded;
    }

    @Override
    public void execute(final CacheService cacheService) {
        entityList.add(cacheService.getActiveEntity(number));
    }

    @Override
    public void setup(final CacheService cacheService) {
        if (!nothingLoaded) {
            cacheService.getAllActiveEntity();
        }

        Collections.shuffle(numbers);
        number = numbers.get(0);
    }

    @Override
    public void useReturned() {
        setOnAllEntity(entityList);
        entityList.clear();
    }
}

public class GetAccountFromOneTable extends Experiment {
    private final List<Entity> entityList = new ArrayList<>();
    private final List<AccountInfo> accountInfos;
    private final boolean nothingLoaded;
    private AccountInfo accountInfo;

    public GetAccountFromOneTable(final boolean nothingLoaded) {
        this.nothingLoaded = nothingLoaded;
        accountInfos = new ArrayList<>(accountMap.values());
    }

    @Override
    public void execute(final CacheService cacheService) {
        entityList.addAll(cacheService.getActiveEntity(accountInfo.info));
    }

    @Override
    public void setup(final CacheService cacheService) {
        if (!nothingLoaded) {
            cacheService.getAllActiveEntity();
        }
```

```
            Collections.shuffle(accountInfos);
            accountInfo = accountInfos.get(0);
        }

        @Override
        public void useReturned() {
            setOnAllEntity(entityList);
            entityList.clear();
        }
    }

    private static class AccountInfo {
        public String info;

        AccountInfo(final AbstractEntity<?> entity) {
            this.info = entity.getAccountInformation();
        }
    }
}
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-06-09

EXAMENSARBETE Database loading strategies for an in-memory cache in Java
STUDENTER Ivar Henckel, David Söderberg
HANDLEDARE Alma Orucevic-Alagic (LTH), Inger Klintmalm (Nasdaq)
EXAMINATOR Jonas Skeppstedt (LTH)

Loading data into an in-memory cache, the choice of being full, lazy, or asynchronous

POPULÄRVETENSKAPLIG SAMMANFATTNING **Ivar Henckel, David Söderberg**

Slow database loading can be a nuisance, causing unresponsive programs and irritated customers. In our thesis, we investigate different strategies to load data into a cache efficiently. We propose new implementations, some of which manage to decrease the inconvenience of data loading.

In the field of computer science databases are commonly used as a means to effectively store and fetch data. Although databases are optimized for fetching data with the high performance there is still a cost for each load, especially with a huge amount of data. To mitigate this cost caches can be used to temporarily store previously accessed data in a location that is faster to access.

In this thesis, we investigated different strategies to load data into the cache. The original cache implementation at the case company loaded all of the data into the cache at startup, called a *full load*. This meant that startup time would increase which could be a problem, especially if a server crashes during business hours and everything needs to be quickly restarted and restored.

One alternative solution is to load data into the cache *lazily*. Lazy loading means that no load is done until the data is used in the program. This will move the cost of loading from startup to run-time while also removing the cost of fetching data that is never used. A third option is to utilize parallelism and load data from separate threads while the main thread can continue without having to wait.

Three solutions were implemented, two of which used lazy loading, and the third used parallelism. These implementations were compared to the original full load solution in experiments. It was found that all three implementations decreased startup time but impacted the run-time request latency in different ways. Additionally, all of our solutions added a higher level of complexity to a varying degree, compared to the full load solution. One of the lazy loading implementations was shown to be inefficient. This solution relied heavily on a higher level framework meaning that we as developers had less control over the details in the database operations. The second lazy loading solution worked efficiently, serving its purpose to reduce startup time but slightly increase run-time latency as expected. Our implementation using parallelism was shown to be efficient both at startup and run-time, but it also adds the most complexity.