International Journal of Computer Applications (0975 – 8887)
Volume 58– No.20, November 2012

Enhancing Application Performance through GORM Optimizations

Soumya Sen Gupta
Centre for Development of Advanced Computing,
C-56/1 Sector 62, Noida,
Uttar Pradesh, India

P Govind Raj
Centre for Development of Advanced Computing,
C-56/1 Sector 62, Noida,
Uttar Pradesh, India

ABSTRACT

GORM (Grails Object Relational Mapping) is an Object Relational Mapping Framework for the Grails web framework. Object Relational Mapping (ORM) frameworks reduce the problems arising out of the object-relational impedance mismatch between the object oriented design model and the relational database design. Unlike other ORM frameworks which require application programmers to configure a lot of XML files, GORM sits transparently between the application logic and the database relieving the programmer from maintaining any sort of configuration files. The default ORM provided by Grails through GORM introduces performance issues in a web application especially when it experiences large loads. This paper identifies problems faced when applying default GORM to application which includes the N+1 select problem, issues with handling one-to-many relationships, bulk insertions as well as problems related to bulk mail transfer and keeping the query cache unused. The paper also suggests optimization techniques which could be applied to each of the problems in order to improve the overall performance of a web application using GORM as its ORM solution.

General Terms

Relational Database, Object Oriented Design, Object Relational Mapping

Keywords

ORM, Grails, GORM, CRUD, Hibermate.

1. INTRODUCTION

When a programmer uses an Object Oriented Language like Java, Grails, etc. to interact with a Relational database such as Oracle, MySQL etc., he/she often encounters difficulties in mapping the Object Oriented Design to the Relational Database Design. These difficulties are often termed as Object Relational Impedance Mismatch [1]. Object Relational Mapping (ORM) [2] is a programming technique that allows reducing this Object Relational Impedance Mismatch by managing the CRUD (Create, Read, Update, Delete) operations. Currently ORM frameworks are available for a lot of languages like QDjango for C++, Athena Framework for Flex, Hibernate, OpenJPA, iBATIS etc for Java. A list of available ORM frameworks and their comparison can be found at [3]. This paper focuses on GORM [4] which is an ORM framework for Grails [5].

GORM maps the domain classes in grails to corresponding tables in a database. GORM brings in huge set of advantages by abstracting the object relational mapping mechanisms without using large amount of xml configuration as in case of other ORM solutions like Hibernate. But it also comes with certain side effects which are capable of reducing the performance of an application to a huge extent. GORM depends on the domain classes of an application to generate the corresponding database table structures. These domain classes are modelled using Object Oriented Analysis and Design techniques [6] like Composition, Inheritance etc. The way in which the domain classes are modelled will have an influence in how data is Created, Read and Updated. Therefore, if these domain classes are not properly modelled and designed, it can result in performance issues in the application. This paper discusses the design level optimizations, which when performed at the GORM level, improves the application performance. The paper discusses certain key problems that arise when ORM frameworks like GORM are used in designing Web Applications. The paper also provides solutions in the form of design optimization for these problems. It also provides a comparison in the application performance before and after each of the optimizations has been applied. The paper has been divided into 8 sections. Section 2 deals with the N+1 select problem which causes an unprecedented number of queries to fire. The problems of using default one-to-many relationships in GORM have been identified in Section 3. Section 4 and 5 focuses on the advantages of using query cache and asynchronous mail transfer respectively. Section 6 discusses the problems occurring in case of bulk insertions and how to overcome the same. Section 7 concludes the paper.

2. N+1 SELECT PROBLEM

ORM solutions often run into the infamous N+1 select problem [7] as a result of non-optimal fetching strategies. An example would better explain the problem. Suppose there is an entity 'Author' which has a one-to-one relation with another entity named 'Location' which holds the residential address of the author and there is a requirement to find the name of all the authors and their corresponding house addresses.

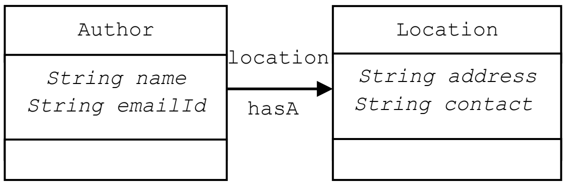

Fig 1: 'Has-A' Relationship between Author & Location

A simple solution to the problem can be defined as follows:

```
Author.list().each {
       a ->    println a.name
               println  a.location.address

}
```

This fetches all the authors from the 'Author' table and then iterates over each author to find out the corresponding address. Supposing that there are 1000 authors in the 'Author' table, GORM will issue an unexpected 1001 number of queries to print all the details. This is because it will fire one query to the database to fetch all the authors from the 'Author' table and next, it will fire another 1000 queries to the database to find the location of each of the 1000 authors. This is the N+1 select problem. One query fetches N number of parent objects followed by N number of select queries, one for each of the referenced parent objects to get the child objects associated with them. Such strategy results in too many select statements being issued and acts as catalyst for degradation in application performance.

A probable solution to avoid the unnecessary issuing of select statements is to fetch the parent and the child object together. Setting Lazy Initialization [1] to false and using Join Fetch [8] can help to avoid the N+1 select problem. It will fetch the Location object pertaining to each author along with the 'Author' object itself. This will reduce the total number of select statements issued from N+1 to only 1 as now there is no need to issue extra queries to fetch the address of each of the authors.

A comparison of the performances, before and after the optimization has been applied, is provided in Fig 2 and Fig 3.

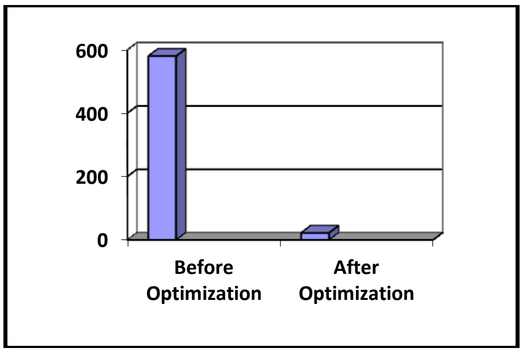

Fig 2: Time taken to fetch Data

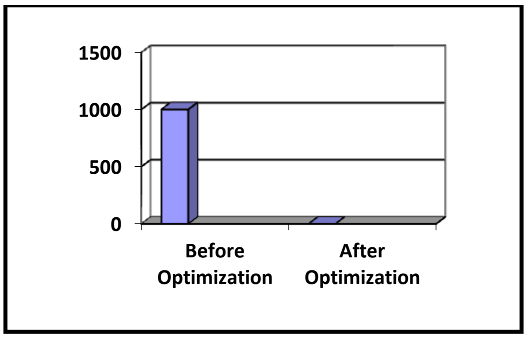

Fig 3: Number of Queries Fired

It can be seen that the time required to fetch the necessary details got drastically reduced from 590 milliseconds to 22 milliseconds after application of proper optimization strategy and the number of queries also got reduced from 1001 to 1.

3. IMPROPER MAPPING OF ONE-TO-MANY RELATIONSHIP

A normal one-to-many relationship [9] in Grails is shown in Fig 4.

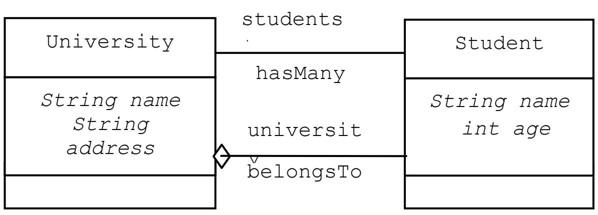

Fig 4: 'One-to-many' relationship between University & Student

The 'University' class has a collection of class 'Student'. Also each 'Student' class belongs to 'University' class. It depicts a one-to-many [9] relationship between a parent and a child class. By default when a relationship like one-to-many is defined with GORM, a java.util.Set [10] type collection is used which is an unordered collection that cannot contain duplicates. Thus the 'students' property that GORM injects is a java.util.Set. Problem occurs when we try to add another 'Student' object to an already existing collection of 'students' belonging to a 'University' object. Every time, when a new 'Student' object is being added to a 'University', GORM loads all the students already in the collection of the particular university and compares each of them with the new one to maintain uniqueness before finally adding it to the existing collection of students. This is a performance bottleneck especially when there is a university having thousands of students. In order to avoid this problem the whole of 'has-many' and 'belongs-to' can be replaced with a simple 'has-a' relationship as shown in Fig 5.

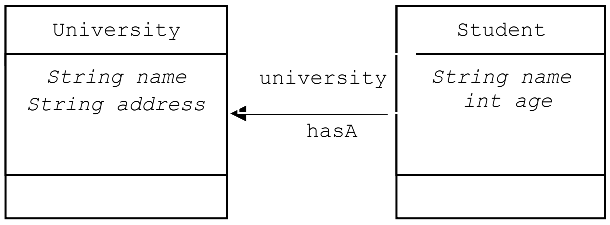

Fig 5: 'hasA' relationship between University & Student

The changed relation now conveys that a Student "has a" University (Composition). However, with this, the easy access to the collection of child items from the parent is lost. Now, to add a Student to a University, a new Student item is to be created which contains the University and saved in database. Also to delete a Student from a University, the required Student item needs to be found from the 'Student' record and deleted. But this little overhead becomes quite acceptable if one wants to avoid the loading up of all the child items each time one wants to add a new child to the parent.

Tests showed that to insert a student to a university which already has a collection of 2000 students, it took 33 milliseconds in case a 'hasMany' relationship exists between university and student, university being the parent, whereas it took just 3 milliseconds to do the same job if the collection is removed from University and each Student is made to be associated with an University.

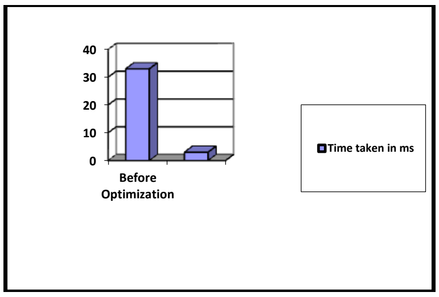

Fig 6: Comparison of time taken (in milliseconds) to fetch data with and without using collections

4. NON-USAGE OF CACHE

The query cache [11] stores the text of a SELECT statement together with the corresponding result that was sent to the client. If the same query is fired again then the results will be taken out of the cache instead of the database. This considerably reduces the fetching time of the query. The query cache can be shared across sessions. So, the cached results of a query can be sent in response to requests made by many other users. In order to test the effect of the query cache, a test was set up which would fire the same query three times and compare the results of the test between a cached and a non-cached environment. In cached environment, the first query resulted in a cache miss but subsequent queries resulted in the cache being hit whereas in the non-cached environment all the queries resulted in the database being hit. The outcome of the test has been provided in Fig 7 and Fig 8.

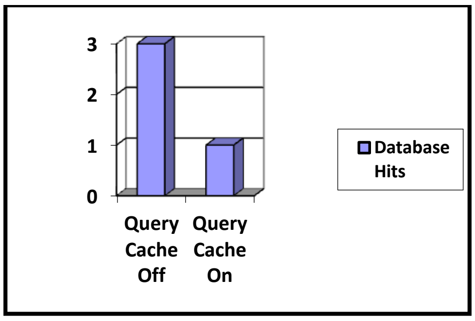

Fig 7: Database hits with and without using query cache

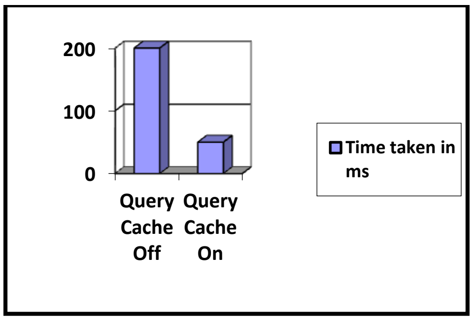

Fig 8: Time taken (in milliseconds) to fetch queries with and without using query cache

As expected, it can be seen that query cache has provided a major performance boost to the application. It can be seen that there is a significant reduction in the time taken to execute the queries in the two environments. The query cache is particularly useful for queries like fetching data from Master Tables which rarely change like name of countries, pin codes of locations etc. i.e. for data which do not change or change rarely.

5. SENDING MAILS SYNCHRONOUSLY CAN BE A BOTTLENECK

Emails are generally sent through SMTP [12] which uses synchronous methods to deliver the messages to an SMTP server. This works well with a small number of concurrent users. But when the number of users increases, the process of sending mails becomes a bottleneck for the application. Each process that involves a mail transfer has to spend a significant proportion of its time waiting for the SMTP server to respond and also there is always a chance that the SMTP server is down for. The user may be unaware of this problem with the mail server and will try again and again to resend the mails until he finally gives up. This results in significant downtime when the number of users is large.

To avoid such a problem mails can be sent asynchronously to the SMTP server. This will not block the currently running process as mail sending will now be done through a separate thread. The user process need not wait for the mail to be actually sent but will issue a separate thread for it and continue its work. The mails will be sent as and when the mail server is available. Grails has an asynchronous mail plugin [13] which stores mail messages in database and sends them in scheduled job. It allows reacting to user's actions faster. If SMTP server can't be available in time then messages can be sent later on, when the server will be available.

With this plugin, one can set the maximum attempts to send a mail, the interval between each attempt, maximum number of messages to be sent at once etc. This drastically reduces the downtime related to sending of mails.

6. PROBLEMS WITH BULK INSERTIONS

The most basic way of inserting 10,000 objects into database through grails would be

```
for ( i in 1..10000 )
{
        Book b1 = new
Book(title:"Book"+i, price:"1000");
        bookService.saveBook(b1);
}
```

For the first 500 inserts, everything goes fine but gradually the rate of insertion goes down as more and more number of insertions take place. There is a sharp increase in the time taken to insert every 500 books thereafter. With the current configuration, the first 500 insertions take place in about 2 seconds; the next 500 insertions take place in 2.5 seconds and so on. The last 500 insertions take an incredible 4.5 seconds and the total time taken to insert 5000 books is 33 seconds. This is because GORM caches all the newly created instances in the session level cache [11] which requires a huge amount of processing to maintain Moreover grails application also contains a map called the PropertyInstanceMap which holds references to newly created instances and thus prevents them from getting garbage collected. The best thing to do in order to get rid of this situation is to regularly clear the session level cache and the map, say after every 500 insertions. This number can vary according to application environment.

```
for ( i in 1..10000 )
{
index++;
Book b1 = new Book(title:"Book"+i,
price:"1000");
bookService.saveBook(b1);
if((index% 500)==0){
        def session =
        sessionFactory.currentSession
        session.flush()
        session.clear() }
}
```

This improves the situation in the way that every 500 insertions now take almost the same amount of time, between 1 and 1.5 seconds. The total time taken to make 5000 insertions now is 13 seconds compared to 33 seconds earlier. The graph below compares the time taken to insert every 500 books between the optimized and non-optimized environment.

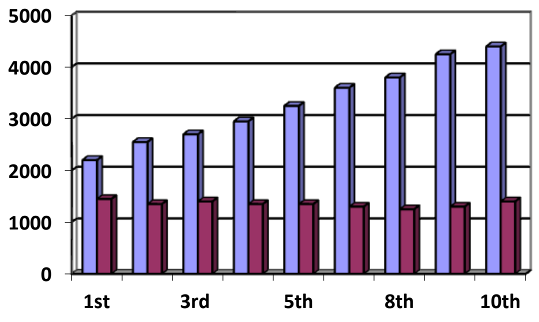

Fig 9: Time taken in milliseconds to insert 500 records each time

7. CONCLUSION

GORM provides a layer of abstraction in mapping object oriented domain classes to corresponding tables in relational databases. This makes programming quite easier as the developer is relieved of maintaining the configuration files related to the mapping of the classes. Though it seems to be quite useful at the first glance, using default configurations without proper insight into the behaviour of GORM can prove to be disastrous as it could lead to serious performance degradation. Both the domain classes and service methods of the application sometimes need to be tuned or configured in a certain manner to get the best performance out of GORM. Although most of the optimization techniques described here has been provided in the background of GORM and Grails, most of them can also be applied in any Object Relational Mapping framework to boost up the performance of the application independent of any programming language or database being used. There are available tools like p6Spy [14] and others which can be used to view the number of queries fired to the database and time taken for each transaction. Such tools, if integrated with the application at the time of performance evaluation, can provide insights into specific transactions responsible for performance bottlenecks. They are also quite useful in performance evaluation (both in respect to the time taken and number of queries fired) before and after any optimization procedure has been applied. It can be seen that each of the techniques works best in certain scenarios but they can also become an overhead for the application if the situation does not suit the particular optimization. It would be best to carefully study each and every scenario which may seem to pose a threat to an application and then apply the proper optimization which suits the best.

8. REFERENCES

[1] Roderic Geoffrey Galton Cattell. Object data management: object-oriented and extended relational database systems. Addison-Wesley Pub. Co., 1994, pp. 122.

[2] Elizabeth J. O'Neil. “Hibernate and the entity data model (edm),” in Proceedings of the ACM SIGMOD international conference on Management of data, 2008.

[3] Wikipedia. [Online] 28 8 2012. http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software.

[4] Robert Fischer. Grails Persistence with Gorm and Gsql. Apress, 2009.

[5] Christopher M. Judd, Joseph Faisal Nusairat, James Shingler. Beginning Groovy and Grails. Apress, 2008

[6] Erich Gamma, Ralph Johnson, John Vlissides, Richard Helm. Design Patterns: Elements of Reusable Object-Oriented Software. s.1. Pearson Education.

[7] Graeme Rocher, Jeff Brown. The Definitive Guide to Grails. Apress. 2007, pp. 78-79.

[8] Christian Bauer, Gavin King. Java Persistence with Hibernate. s.l. : Manning Publications, 2009, pp. 588-589.

[9] C. J. Date, A. Kannan, S. Swamynathan. An Introduction to Database Systems. s.1. : Pearson Education. 2009, pp. 345-346

[10] Paul J. Deitel, Harvey M. Deitel. Java for Programmers. s.1. Pearson Education. 2012, pp. 606-607.

[11] Eric Pugh, Joseph D. Gradecki. Professional Hibernate. s.1. : Wrox Publishing. 2004, pp. 215-216.

[12] Wikipedia. [Online] 28 8 2012. http://en.wikipedia.org/wiki/Simple_Mail_Transfer_Pro ocol.

[13] Grails. [Online] 28 8 2012. http://grails.org/plugin/asynchronous-mail

[14] [Sourceforge. [Online]. 28 8 2012. http://sourceforge.net/projects/p6spy/