ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)
Vol. 3, Issue 2 (Apr. - Jun. 2015)
International Journal of Advanced Research in Computer Science & Technology (IJARCST 2015)

Performance Tuning - Accounts Receivable Process on Hibernate

Tushti Naryani
B.Tech in Information Technology, ITM, MDU, India

Abstract
Persistence layer serves as an imperative ingredient of the Enterprise Application. Hibernate being an open source Java persistence framework project performing powerful object relational mapping and querying databases using HQL and SQL. Performance of the application is one of the major concerns for production implementation. While designing the Application, myriad factors must be taken into consideration for better performance of application. Every organization has a Finance and Accounting department which uses Accounts Receivable Process, and is considered to be vital pillar of the organization since it is related to billing and finance activities. Performance tuning of this process involves optimization of multitude aspects of the system which have been analyzed here.

Keywords
Hibernate, Performance Tuning, Accounts Receivable, Optimization, Persistence

I. Introduction

Creation and maintenance of the persistence layer is chief part of the development of Enterprise Application to accumulate and retrieve objects from the database of choice. Many a times, organizations route to create home developed persistence layers which are time and again buggy. If changes are made to the underlying database schema, it can be pricey to disseminate those changes to the rest of the application. To fill this fissure, Hibernate steps in. Hibernate provides and an easy-to-use and potential object relational persistence framework for Java applications. Instead of utilizing code generation or byte code processing, it uses runtime reflection to determine persistent properties of a class. A mapping document is maintained to describe the persistent fields , associations , subclasses or proxies of the persistent object and objects to be persisted are defined. The mapping document are compiled at application startup time and supplies the framework with necessary information. From a compiled collection of mapping documents, a session factory is created. The Session interface and persistent classes are managed from a mechanism provided by Session Factory. The interface between the persistent data store and the application is provided by session class, which wraps a JDBC connection, user- managed and controlled by hibernate and can be used by single thread, created and discarded. This functionality doesn't come with much cost on performance. Its overhead is much less than 10% of the JDBC calls, and our experience in deploying applications using hibernate supports this. Hibernate can make multiple optimizations when interacting with the database, including caching objects, efficient outer join fetching and executing SQL statements only when needed. It is intricate to achieve this level of sophistication with hand-coded JDBC.

II. Accounts Receivable Process

The Accounts Receivable process is used by Finance and Accounting Department of n organization. It covers various activities:

1. Invoice Generation
2. Invoice Authorization
3. Pending Invoice follow-up
4. Fee Receipt
5. Invoice Settlement
6. Invoice Cancellation
7. Invoice Revive

Here we will tune the performance when the Accounts Receivable process is developed in Hibernate. We will focus on the areas where we face performance related hitches and we will focus on the solution. The quantitative performance improvement data will also be analyzed.

Environment

1. Hibernate version : Hibernate - 3.6.10-Final
2. Database : Oracle 11g
3. Application Server : JBoss 5.0.1 GA

III. Examining The Performance Hitch

The property tagged as ***use_sql_comments*** in the hibernate configuration file *hibernate.cfg.xml* adds a comment to apiece SQL clarifying why it was created. It helps you spot in a HQL statement if the lazy loading or a criteria query led to the statement.
<property name="use_sql_comments">true</property>
The property set as ***format_sql in*** *hibernate.cfg.xml*, adequately formats the SQL similar to the substitute of printing it on a single line.
<property name="format_sql">true</property>
There might be a lot of unexpected queriesleading to slowlness of a dialog. This can be triggered by eager loading of relations or just reuse the queries of another dialog. For instance 1:1 and n:1 relations are loaded eagerly, if you use annotations whereas XML mappings are lazy by default. The approach, considered preeminent so far, to analyze the working in the wake of the scenario is 'Debug-Configuration'. Two configuration settings are offered by Hibernate.
The first one is a property in the *hibernate.cfg.xml*
<property name="show_sql" >true</property>
If it is set to true, the SQL statements will be printed to the console. This does not use any timestamp.
The second approach is normal logging. This offers timestamps and hence is better than previous approach.

```
#logs the SQL statements
log4j.logger.org.hibernate.SQL=debug

# Some more useful loggings
# Logs SQL statements for id generation
log4j.logger.org.hibernate.id=info
# Logs the JDBC-Parameter which are passed to a query (very
```

ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)

verbose)
log4j.logger.org.hibernate.type=debug
# Logs cache related activities
log4j.logger.org.hibernate.cache=debug
Another good spring for information are the ***statistics of Hibernate.*** You can enable the statistics in the Hibernate configuration or programmatically. The statistics class offers a number methods to analyse what has happened. Here a quick example:

```
Statistics objStatistics = sessionFactory.getStatistics();
objStatistics.setStatisticsEnabled(true);
objStatistics.logSummary();
```

IV. Data Fetch Optimization

Internally hibernate makes a query. It stores the data in by-default first level cache. There is no parent child mapping. So we don't face n+1 select problem. Since we don't face n+1 select problem, enabling second level cache will only increase overhead. We have two kinds of queries in Accounts Receivable Project i.e. Static and Dynamic. For Instance, Fetching Debit note numbers for view purpose is static query and fetching only the billable and unauthorized debit notes is dynamic. Following data was run on Hibernate 3.6.10, so we observed milliseconds of difference only which is negligible. After Hibernate3.3 this performance issue with use of criteria query was handled. There would have been noticeable TAT loss if this was made on any version less than Hibernate 3.3.

Eager fetch is default fetch type in hibernate 2. Performance of Eager Fetch is undoubtedly faster because of one select query, time response is better. .Associations are fetched right at the time of fetching parent object. So in this case we don't make database call again and again. But this will be bad, if we are fetching too many objects in a session because we can get java heap error. So for memory optimization we use Lazy strategy. Since we are loading too objects in a session we go for Lazy Fetch strategy. When a user has many objects associated, it is not efficient to load all of its objects with it when they are not needed. To prevent heap error, we used lazy fetching. Since there is no parent-child or one-to-many mapping in our entities, there is no benefit of using eager fetch over lazy fetch. Hence, N+1 select problem will not be encountered.

Batch – Fetching: Case: The Accounts Receivable application retrieves all debit notes settled against a Fee Receipt received from Client by Accounting Department. It iterates through all Invoices and counts the number of occurrences. An alternative scenario could go through status of all the Fee Receipt of a Client and check if one of the Invoices is in a position to be settled. The query which is used for the operations related to Fee Receipts received from the client for invoice settlement:

```
List<EntityFeeReceipt> oList = session.createQuery("from Fee_Rcpt b where b.number like ?").setString(0, "FA_Clt1_%").list();
```

The code printing the Fee Receipt will create one SQL query per Fee Receipt to initialize the Invoice numbers. We get 1+n queries in total i.e. one for the Fee Receipt and n for the invoice, if we have n invoices settled using that fee receipt.

```
for (EntityReceipt oReceipt : Receipts) {
  int totalLength = 0;
  for (Invoice invoice: oReceipt.getInv_No()) {
    totalLength += (invoice.getContent() != null ? invoice.getContent().length() : 0);
  }
  log.info("Length of all invoice: " + totalLength);
}
```

Approach to improve**:** One way to perk up this is to define that Hibernate loads the chapters in batches. Here is the mapping:
Annotations based:

```
OneToMany(cascade = CascadeType.ALL)
<br>JoinColumn(nullable = false)
@BatchSize(size = 4)
private Set<Invoice> invoices = new HashSet<Invoice>();
```

XML based:

```
<set name=" Invoice" batch-size="4">
        <key column=" Inv_No"></key>
        <one-to-many/>
</set>
```

When iterating through 15 Fee Receipts, Hibernate will load the Invoices settled in batches of four: such as, for the first four receipts, the next four, next four and the last three Receipts together in a single query. This is achievable since the *java.util.List* returned by Hibernate is bewitched. The unsurpassed size of the batch size is the number of entries you have to print on the screen. If you print a Fee Receipt and print only the first 15 Invoices, then this could be your batch size.It is feasible to set a default for all relations in the Hibernate configuration.

```
<property name="default_batch_fetch_size">4</property>
```

PS: A size of 'Two' reduces the queries already by 50 % and size of four by 75.

V. Inserting Data Optimization

```
@Id
  @GeneratedValue(generator = "auto_increment")
   @GenericGenerator(name = "auto_increment", strategy = "increment")
  @Column(name = "EXP_CD")
  private Long EXP_CD;
```

This is a code snippet from Entity of Disbursement Fee master of Account Receivable Process. We are using @ GenericGenerator and not the sequence generator in this case. Strategy type used is 'increment'. This generates Integer, long or short type of ids. This generator reads maximum present primary key in database table and then increments it by one and inserts new row with new id.

Though the best id generators in Hibernate are *enhanced-table* and *enhanced-sequence*, coupled with an appropriate optimizer, such as *hilo*. For Instance, *enhanced-table + hilo*, inserts over 10,000 records per second, but multiple insertion at this scale is not a use case of application. So the data presented regarding insertion is using Increment type generic generator

VI. Data Consistency

If many users are working simultaneously to access and update common data from database, there would be a consistency issue. One solution to this problem is to implement row locks on the data being modified. However, should a process or a user be modifying several rows, this lock can lead to performance squalor as the DB begins to kill time for locks to be released. A better answer is to use Optimistic Locking. This is where we screen the row being modified and allow other users to access it.

```
@Entity
```

```
public class Flight implements Serializable {
   @Version
   @Column(name=”OPTLOCK”)
   public Integer getVersion() { ... }
}
```

Version entry in hibernate config xml

```
<version
      column=”version_column”
      name=”propertyName”
      type=”typename”
      access=”field|property|ClassName”
      unsaved-value=”null|negative|undefined”
      generated=”never|always”
      insert=”true|false”
            node=”element-name|@attribute-name|element/@attribute|.”
/>
```

VII. Portability Tuning

First, and perhaps most credibly, we find annotations-based mappings to be far more intuitive than their XML-based alternatives, as they are right away in the source code along with the properties with which they are associated. Most coders prefer the annotations because fewer files have to be kept synchronized with each other.

Annotations are less verbose than their XML equivalent.

```
@Entity
public class EntityDebitNoHdr implements Serializable {
   @Id
   @GeneratedValue(generator = “auto_increment”)
    @GenericGenerator(name = “auto_increment”, strategy = “increment”)
   @Column(name = “DB_NOTE_ID”)
   private Long DB_NOTE_ID;
}
```

Equivalent mapping file:

```
<hibernate-mapping default-access=”field”>
  <class name=” EntityDebitNoHdr “>
    <id type=” Long “ column=” DB_NOTE_ID “>
      <generator class=”native”/>
    </id>
   </class>
</hibernate-mapping>
```

So the tag names and the boilerplate document-type declaration have been avoided. The annotations themselves are portable across JPA implementations

Hibernate Annotations are superset of JPA Annotations. So, all the JPA annotations can be easily used. Moreover, hibernate annotations can also be used for exact features.

Also, hibernate uses and supports the JPA 2 persistence annotations. If you elect not to use Hibernate-specific features in your code and annotations, you will have the freedom to deploy your entities to environments using other ORM tools that support JPA 2. For example:

```
@Entity
public class EntityDebitNoHdr implements Serializable {
   @Id
   @GeneratedValue(generator = “auto_increment”)
    @GenericGenerator(name = “auto_increment”, strategy = “increment”)
   @Column(name = “DB_NOTE_ID”)
   private Long DB_NOTE_ID;
}
```

The @Entity annotation marks this class as an entity bean

The @Id annotation will not create a primary key generation strategy, which means that you, as the code’s author, need to determine what valid primary keys are, by setting them explicitly calling setter methods. OR you can use@GeneratedValue annotation

@GeneratedValue annotation takes a pair of attributes: *strategy* and *generator*

The Accounts Receivable Project is made Annotation based to enhance portability across JPA implementations. Portability is vital because this is common process to numerous domains. When this is implemented in other domains, making it portable will reduce re-implementation outlay.

VIII. Connection Pooling Optimization

Hibernate’s own connection pooling algorithm is, however, quite rudimentary. It is intended to help you get started and is not intended for use in a production system, or even for performance testing. One must use a third party pool for best performance and stability. Just replace the *hibernate.connection.pool_size* property with connection pool specific settings. This will turn off Hibernate’s internal pool. For example, you might like to use c3p0.

Default pooling:

```
<hibernate-configuration>
   <session-factory>
         <property name=”dialect”>
         org.hibernate.dialect.MySQLDialect
      </property>
        <property name=”connection.datasource”>java:jboss/datasources/MySqlDS</property>
   </session-factory>
</hibernate-configuration>
```

Specific pooling:

```
 <hibernate-configuration>
 <session-factory>
   <property name=”hibernate.connection.driver_class”>com.mysql.jdbc.Driver</property>
   <property name=”hibernate.connection.url”>jdbc:mysql://localhost:3306/myschema</property>
   <property name=”hibernate.connection.username”>user</property>
 <property name=”hibernate.connection.password”>password</property>
   <property name=”hibernate.dialect”>org.hibernate.dialect.MySQLDialect</property>
  <property name=”show_sql”>true</property>
   <property name=”hibernate.c3p0.min_size”>5</property>
  <property name=”hibernate.c3p0.max_size”>20</property>
  <property name=”hibernate.c3p0.timeout”>300</property>
   <property name=”hibernate.c3p0.max_statements”>50</property>
   <property name=”hibernate.c3p0.idle_test_period”>3000</property>

</session-factory>
```

ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)

IX. Performance analysis

Performance of the Accounts Receivable Application on Hibernate persistence has analyzed for tuning purpose. After successfully observing round trip times on various operations, they have been applied with discussed optimization strategy and then data is compared with non-optimized code. We have observed significant difference by modifying the insertion methods. Table 1.1 mentions all the data collected by running the selection, insertion, update and so forth operations. Start time of operation is noted along with the end time and turnaround time is calculated. This operation is done consecutively five times and average time is noted. The observations are also plotted on line graph to study the variations in turnaround time.

Table. 1 :Turnaround Time for different operations

Activity	Start Time (ms)	End Time (ms)	TAT
Select	1427110166929	1427110166953	24
	1427110233754	1427110233782	28
	1427110258774	1427110258795	21
	1427110285847	1427110285871	24
	1427110312789	1427110312813	24
Insert-Seq	1426567392220	1426567392525	305
	1426567774486	1426567774582	96
	1426567871485	1426567820358	48
	1426567871485	1426567871514	29
	1426567922859	1426567922907	48
Average			87.7
Insert using generic	1426568253710	1426568253918	208
	1426568320374	1426568320458	84
	1426568353345	1426568353376	31
	1426568386887	1426568386923	36
	1426568436451	1426568436481	30
Average			64.8
Updating	1426483940478	1426483940530	52

Fig. 1 : Selection query for 5 consecutive times

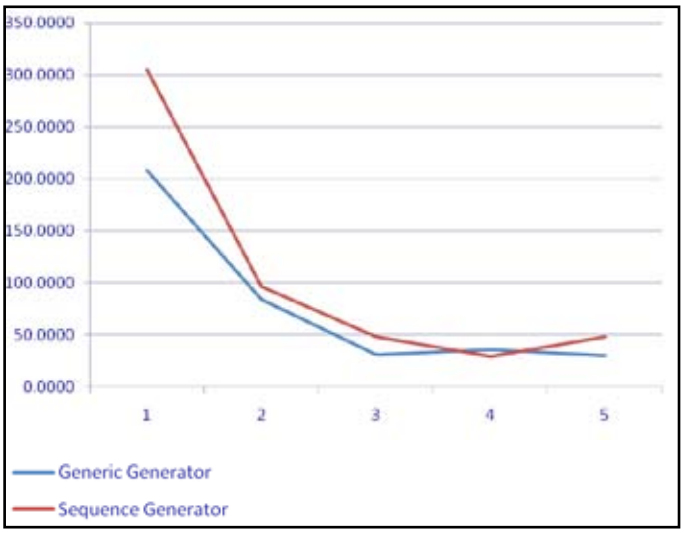

Fig. 2 : Insertion Using Different Generators

X. Conclusion

To optimize performance of Hibernate based Enterprise Application, we need to consider myriad technical, functional and operational factors which directly or indirectly tamper the performance of the application. Implementing the right combination of Hibernate mechanism will prove out to be useful for the application at Production stage. Accounts Receivable Process, involves a lot of database hits during selection, update and insertion operations. Optimizing such operations is the key to optimize Accounts Receivable Process.

References

[1]. *Hibernate Made Easy: Simplified Data Persistence with Hibernate and JPA (Java Persistence API) Annotations by Cameron Wallace McKenzie, Kerri Sheehan*
[2]. *Harnessing Hibernate By James Elliott, Timothy M. O'Brien, Ryan Fowler*
[3]. *Hibernate Recipes: A Problem-Solution Approach by Gary Mak, Srinivas Guruzu*