Handling Huge Data Sets in J2EE/EJB 2.1 with a Page-By-Page Iterator Pattern Variant for CMP

Ralf Gitzel, Axel Korthaus, Nima Mazloumi
University of Mannheim, Germany
{gitzel|korthaus|mazloumi}@wifo3.uni-mannheim.de

Abstract- **The J2EE platform with its server component technology Enterprise JavaBeans (EJB) has become widely adopted today. Services provided by J2EE component containers, such as container-managed persistence (CMP), facilitate the development of distributed transactional applications and increase portability of EJB components. Thanks to its intensive web support through servlets and JSPs, the J2EE standard easily allows web application designers to bring data stored in EJB entity beans to the Internet as HTML pages.**
A major problem, however, is the fact that often there are too many instances of an entity bean to be presented on a single HTML page. In our paper, we will compare different solutions to this problem. As a starting point, we will measure the performance of a naive CMP-based approach and compare it with a bean-managed persistence (BMP) approach structured in a similar way. Since the use of CMP provides many benefits, but the naive approach shows only poor performance, we finally present a variant of Sun's page-by-page iterator pattern, which, as opposed to the original, does not rely on JDBC but rather uses the new EJBQL features introduced in the new EJB 2.1 final draft in order to provide a feasible solution within the framework of CMP. In our conclusion we also identify some possible improvements to the J2EE standard based on our findings.

I. Introduction

Distributed enterprise application systems need a way to persist objects and to present object data in the World Wide Web. The J2EE platform, which consists of several powerful APIs including the specifications of web-oriented technologies such as servlets and Java Server Pages (JSP) on the one hand, and a server component technology called Enterprise JavaBeans (EJB) for representing business objects on the other hand, fully provides the technological foundation for the achievement of these and even more complex goals in the domain of developing transactional, secure, reliable, manageable and scalable client-server business applications built in Java.
However, designing and implementing a scalable, maintainable and reasonably fast application based on J2EE technologies in general and EJB in particular is not trivial. One aspect the designer of an EJB-based application has to consider is which persistence mechanism should be used for persisting enterprise bean data in secondary storage. There are two basic options. The first is bean-managed persistence (BMP), requiring the developer to write code based on JDBC, JDO, SQLJ or similar technologies to store, load, create and remove the bean's state. The second, container-managed persistence (CMP), relies on the container to automatically takes care of maintaining the database.
There are numerous pros and cons of each of the two approaches, which we do not want to discuss here in detail. For example, using BMP, the developer has to make sure that all tables and fields are read and written properly and that the bean accurately reflects the data in the persistent store, which gives him control over inner aspects of the bean, but is also a tedious and error-prone task. Furthermore, using technologies such as JDBC, for example, for BMP decreases portability, because the enterprise beans get coupled to the concrete schema and kind of the database in use. With CMP, on the other hand, development, modification, and maintenance are made easier, and maximum portability can be guaranteed, since only an abstract schema is used and the concrete kind of database can at least theoretically be exchanged without significant problems. Another aspect of interest for the decision between BMP and CMP in a concrete project is the performance penalty or gain caused by the respective approach. While some authors even claim, that tuned CMP entity beans offer better performance than BMP entity beans [oracle2002], this assessment must not be generalized, as will become apparent in the context of our analysis. The results of our simple study covering the relative performance of a CMP-based and a BMP-based solution will be presented in section II. We focus on the specific problem of handling huge sets of data provided by enterprise beans for presentation (e.g. in a web interface).
This problem is very common and has been addressed before with the controversial page-by-page iterator pattern or its successor, the value list handler pattern [SUN] [ServerSide]. There is a caveat, however, as the page-by-page iterator pattern description leaves out many of the details - in fact, it is more of an analysis of the problem than a solution - and the example implementation is based on BMP and makes use of a fast-lane reader [SUN], which involves the use of JDBC and SQL. Since the schema of the database is always application-specific and not part of the

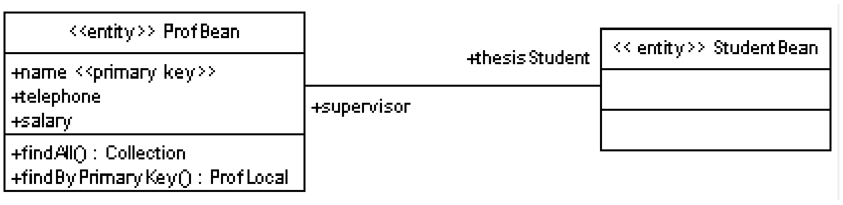

Figure 1: The Initial Design

J2EE specification [J2EE], it is desirable to look for a more portable way to realize the pattern.

In our paper, we show that a naive CMP-based approach to handling huge data sets on a page-by-page basis is too slow compared with BMP. Since CMP has several advantages, we offer a solution to the problem. We will introduce two variants of SUN's value list handler pattern that will help those web application designers who want to go for maximum standard compliance and portability by implementing a JDBC-free solution that uses some of the new features that will be introduced in the EJB 2.1 standard. Finally, we discuss some ideas of how the EJB standard could be extended to provide a more efficient solution fully integrated in the standard.

II. Performance comparison

In order to get a concrete impression of the performance of different technological solutions to the problem of extracting huge data sets on a page-by-page basis from an application built with enterprise beans, we implemented a simple problem domain model in two ways, namely using a J2EE 1.3 based CMP approach and a equivalent JDBC-based BMP solution.

Figure 1 shows our example's problem domain model. The analysis model is the basis for a university database where information on professors and students is stored. There are two finder methods planned for the professor, one to find all entries and one to find a specific professor by his or her name (which is the primary key). To allow “iteration” additional methods for retrieving subsets of the complete set of `ProfBean` instances and of the set of `StudentBean` instances linked via container-managed relationships have to be added.

For our first test, we implemented the `ProfBean` as an entity bean with CMP and had a client simply request the primary keys of all instances existing, returned in subsets of a specific size. In its Local Home Interface, the `ProfBean` provides a method `getAllSubset` which can be passed a start and an end index used to get a subset of the complete set of primary keys. We chose to return the primary keys only, because this is less resource intensive than returning complete value objects. However, the problem with EJB 2.0 CMP is, that EJBQL does not provide any possibilities for “caching, scrolling, and random access to result sets” [SUN], so that we have to read all instances from the database each time, even if we only want to return a subset. Thus, method `getAllSubset` internally calls method `ejbHomeGetAllKeys` which returns the complete set of primary keys, based on an invocation of `ejbSelectAll`, which delivers a collection of references to all `ProfBean` instances by performing EJBQL query `SELECT OBJECT(a) FROM PROF_SCHEMA AS a`. Needless to say, this overhead of access stubs is a major factor in performance loss.

In order to avoid undeterministic network traffic influencing the performance analysis, we provided a test session bean which locally accesses the entity bean and measures the amount of time the entity bean needs for retrieving subsets of different size from `ProfBean` extensions of also varying size.

In order to be as fair as possible, we designed a equivalent BMP variant based on a similar structure, i.e., reading the complete set of primary keys for each subset to be returned. However, the BMP variant access the database directly using JDBC. The design is suboptimal, because it does not take advantage of the possibilities of JDBC 3.0 for retrieving subsets of data, but we wanted to compare programs which are as identical as possible and as the results show our conclusions would be the same with even faster access.

In our test scenarios, we iterated over all instances of the `ProfBean` with increasing page (i.e., subset) size within one test run. We performed all tests in one run to avoid too much influence of the typical initial overhead during startup, which only occurs in the first run. Each page size was used 100 times, before moving on to the next page size.

Our test setup was built on a Pentium 4 PC, 1 GHz, 512 MB RAM, operated with Windows 2000. We used JBoss 3.0.4 with Tomcat 4.1.12 as our J2EE application server and web server, and the database storing our entity bean data was HyperSonicSQL, which was bundled with the JBoss distribution. While JBoss has several weaknesses [SOFTWARE], the cost factor made it the primary choice. However, our software examples can easily be tested on other servers as well.

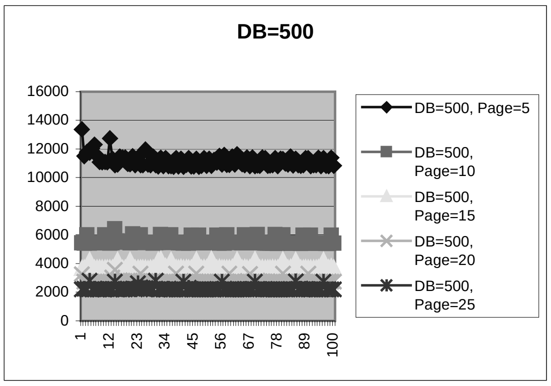

Fig. 2: Measurement of naive CMP, size 500

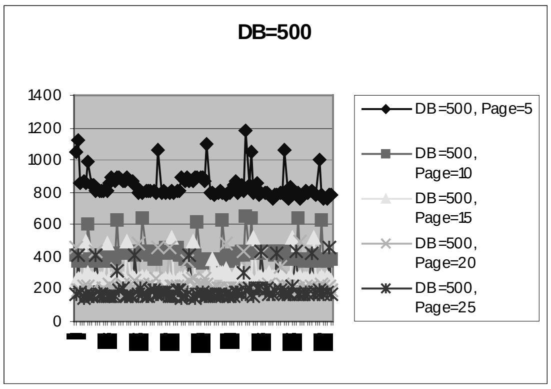

Fig. 3: Measurement of BMP, size 500

Some results from our performance analysis are shown in the figures above. Figures 2 and 3 illustrate the 100 measurements of the response time in milliseconds performed for different page sizes (see different gray scales) on a database with 500 instances of `ProfBean`, for the CMP (fig. 2) and the BMP (fig. 3) variant. Figure 4 shows a comparison of the CMP and BMP approach based on the average data for three different sizes of the database.

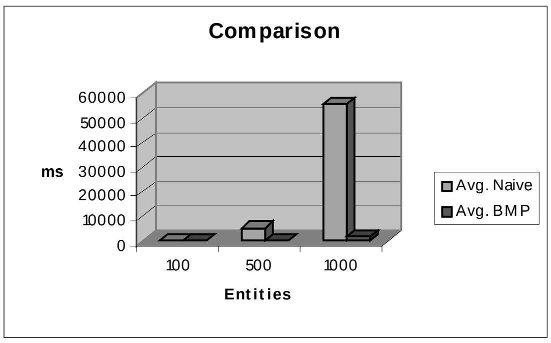

Fig. 4: Comparison

As can be observed from the figures, the CMP solution is much more sensitive to the total number of entities than the BMP solution, and for 1000 entities, which is not much in real world problems, it already becomes unacceptably slow and even lead to an out-of-memory error in our test setup, presumably because of the amount of active EJBObjects in the application server. In our scenarios, a bigger page size leads to a shorter amount of total time, which is not surprising, because increasing the page size results in fewer calls to the database in our implementation. What is a bit surprising are the little peaks produced by the BMP solution, which we suppose could be intermediate runs of the garbage collector, since they even occur if the database is empty. Recapitulating, it can be said that in our specific test scenario, the naive CMP solution performs very bad compared to the BMP solution. To improve this situation, we developed the following two patterns, which use the new EJBQL 2.1. Unfortunately, at the time of writing we did not find any application server which already implemented the new standard features required for the new patterns, so that an empirical performance comparison was impossible. However, the specific design of the patterns substantiates the claim that they will result in significant performance gains.

III. J2EE Iterator Pattern Design

When listing collections of Entity Beans in a web page, there are two possible origins for the data; they may be the result of a finder method or the contents of a CMR. While the solution of how to subdivide the results is similar in both cases, there is enough of a difference to warrant separate patterns. Both techniques are illustrated in the sample code.
Figure 1 shows our example's problem domain model. Based on this we will detail how to write an entity bean that is suitable for iteration. The analysis model is the basis for a university database where information on professors and students is stored. There are two finder methods planned for the professor, one to find all entries and one to find a specific professor by his or her name (which is the primary key).

Before we go into our solution, we will briefly talk about the options we decided against. Caching is one solution for increased speed offered by the value list handler. However, we have identified memory as a bottleneck in the BMP, this strategy seems questionable, at least for storing references to entity beans. Storing the value objects might be a better solution

but due to the other problems of caching (updates etc.) this strategy was not persued.
A Data Access Object [SUN] can be used to mask the entity beans and is used in the value list handler pattern. In order to minimize changes to be made to existing programs when introducing our solution, it seemed better to stay as close to the original entity beans as possible.

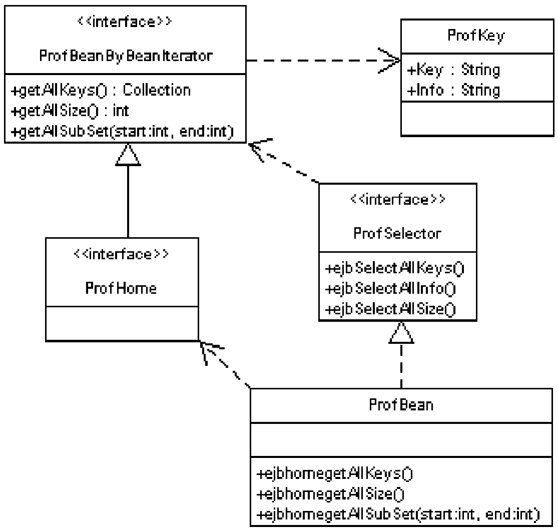

Fig. 2: The Solution for Finder Methods

The proposed page-by-page iterator pattern variant for finder methods involves the following components (see figure 2).
Since the finder methods originally defined do not suit our iteration needs, we replace or complement each of them with three home methods (defined in the *ProfBeanByBeanIterator* interface). Naturally, this only applies to those finder methods which return collections, all others need no iteration. In our example the method findAll() results in the following methods:
getAllKeys() returns the keys of all ProfBean instances, as a collection of value objects [SUN] of type ProfKey, which also contain some textual information. The method assembles these objects by using the results of the select methods ejbSelectAllKeys() and ejbSelectAllInfo() (see below).
getAllSize() returns the size of the result set as an integer, using ejbSelectAllSize().
getAllSubSet(int start, int end) allows access to subsets of the result of getAllKeys(). It uses a java.util.Iterator on the result of getAllKeys() to extract the desired subset to return. Since java.util.List is no valid return type for select methods, a direct access via indices is not possible.

The entity bean class itself must provide the implementations of the home methods as shown in the figure. For the reader's convenience these have been grouped in the the *ProfSelector* interface, to be implemented by the bean class itself. It defines several select methods which will be used by the home methods defined in the ProfBeanByBeanIterator. This indirect approach is necessary as it is not allowed to expose select methods in the home or component interface and finder methods can only return complete entity bean instances and not their fields [EJB2.1].
The *ejbSelectAllKeys()* method returns all key values (the names of the Profs) sorted in a set. The EJBQL statement depends on the intended query for the original findAll() method. Thus, if the original query was

SELECT OBJECT(a) FROM prof AS a WHERE [all-condition]

where all-condition is the condition associated with the current finder method, the new query would be

SELECT a.name FROM bean AS a WHERE [all-condition] ORDER BY a.name

Based on this query, *ejbSelectAllInfo()* must use the following query:

SELECT a.telephone FROM bean AS a WHERE [all-condition] ORDER BY a.name

Note that sorting is still based on the key (a.name), otherwise it would be impossible to know which info belongs to which key. The info can be any field the designer deems suitable to list along the key to allow a potential user to identify which database entry is represented by the key (in our case it is the phone number). This is especially useful for either short overview lists that will most likely not require any additional details (like a phone directory in this case) or entities with numerical keys. As an example for the the latter case, assume a database sorted by SINs. It is easier to identify ShortInfo = Miller (SIN = 12345) as the entry one was looking for than SIN = 12345 by itself.
Of course it is possible to drop the value object and the ejbSelectAllInfo() altogether and just return the collection of keys from ejbSelectAllKeys(), which has a better performance.
*ejbSelectAllSize()*counts the number of beans that satisfy the condition:

SELECT count(a) FROM bean AS a WHERE [all-condition]

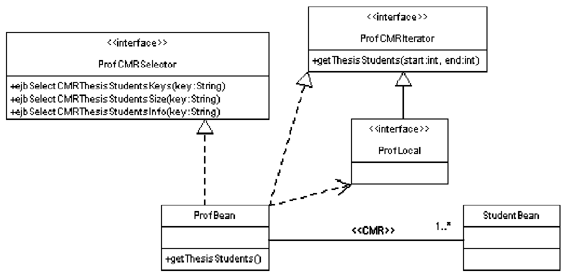
<<interface>>
ProfCMRSelector
+ejbSelectCMRThesisStudentsKeys(key:String)
+ejbSelectCMRThesisStudentsSize(key:String)
+ejbSelectCMRThesisStudentsInfo(key:String)
<<interface>>
ProfCMRIterator
+getThesisStudents(start:int, end:int)
<<interface>>
ProfLocal
ProfBean
+getThesisStudents()
<<CMR>>
1..*
StudentBean

Fig. 3: The Solution for CMR Fields

So far we have only covered access to the results of finder methods, still leaving unanswered page-by-page access to CMRs. As in the first pattern, several helper methods are required as described in figure 3. Again, we have a couple of select methods to help our iteration methods. Since there are many similarities to the example above, only some of the methods are described here, to highlight the differences.

For example, *selectCMRThesisStudentsKeys(String key)* is the equivalent of selectAllKeys(). However, this time we need to look up a specific object - therefore a key value has to be passed. The select method returns all StudentBean instances that are reachable through the ThesisStudents CMR from the ProfBean with the primary key key. The EJBQL statement is:

```
SELECT b.id FROM prof AS a, IN(a.thesisStudents) AS b WHERE prof = ?1 ORDER BY b.id
```

It returns all students that are reachable via the ThesisStudents CMR from the professor who matches the key passed to the select method.

We also need a method paralleling getAllSubSet(int start, int stop), in the our example a business method called *getThesisStudents(int start, int stop)*. Internally, it is very similar to its paragon. A nice feature is that we can simply overload the required method for the CMR, in our example getThesisStudents().

All other methods are easily derived along these lines.

Now that we know the details of these patterns, let us look at their characteristics. First of all, they share a lot with the original Page-By-Page Iterator. They reduce network traffic at the expense of server requests and are not robust to changes [SUN]. Unique properties are:

- No JDBC is needed. As a result, portability is improved significantly and performance depends on the container implementation.
- No state is kept which avoids expensive storage of user data and obsolete data representations, because cached data will not reflect changes to the database that occurred in the mean time.
- The select methods rely on functionality provided in the EJB 2.1 standard, e.g. to count or sort. Also new is the IN() operator essential for CMR iteration.
- If the EJB classes are already provided it might become necessary to write wrapper entity beans which have a one-to-one CMR to the encapsulated bean and perform the iteration functionality. A wrapper entity bean implements all the required select methods with slightly modified queries. Instead of directly accessing the bean data (e.g. a.name), the queries navigate the CMR (i.e. a.encapsuledBean.name). Otherwise, the rules stated in the patterns remain valid. However, the encapsulation will result in a more complex database structure and less performance, since each access to instance data set now involves two beans, one for the original entity and one for the wrapper.
- The pattern can be expanded to include additional ways of partitioning the data, not based on the number of data sets. This can be achieved by using parameters and LIKE statements. For example, alphabetical partitioning can be realized by a findKeyStartsWith("a%") method that returns only those keys that match the given pattern.

IV. Proposals for the Standard

The problem addressed in this paper is by no means an uncommon one. In fact, it should be relevant for all J2EE based applications that involve some sort of content management. Therefore the question arises, whether the conclusions drawn from our experiment could somehow be formed into a proposal for a change in the J2EE standard.

The least significant change would be to include java.util.List in the return types allowed for finder methods. This seemingly trivial change would at least avoid activating every single entity bean by iterating over the references stored in the collection returned by the finder.

Another suggestion is to replace by EJBQL by OQL altogether. Not only the problem described here but many others could be solved by a more powerful query language.

A more far-reaching proposal would be the inclusion of parameterized finder methods paralleling the functionality of getAllSubset() and similar methods. Implementation details would again be left to the container. While this solution mirrors our suggestion for J2EE 1.4 style programming, it has drawbacks. First of all, the *style* of EJBs would change somewhat – not a desirable aspect of a new standard version.
Comparing these suggestions, we come to the conclusion that the inclusion of Lists would be a simple but powerful change to the J2EE standard. While the advantage given above is marginal, performance can improve radically by lazy-loading For example, a collection returned by a finder could be implemented in such a way as to only retrieve elements when they are needed. The increased number of database queries would be offset by a massive reduction of entity bean references in certain types of cases. Ideally, a switch within the container-specific part of the deployment descriptor could be introduced for each finder method. Therefore a minimal change would vastly improve the usability of J2EE.

IV. Conclusion

This paper described and proved the existence of a performance problem when using value list handler-type patterns together with CMP in J2EE applications. After illustrating the problem with an empirical experiment and identifying the causes for this bottleneck, a J2EE 1.4-based solution was presented.
Finally, the insights gained through the prototypical implementation were used to identify possible changes to the J2EE standard. It became apparent that very minute changes would vastly benefit J2EE with regard to the common Page-By-Page Iterator problem.

V. References

Sample Code: Profbean, Local Home Interface, Local Interface: src.zip

[Oracle2002] EJB Best Practices. http://otn.oracle.com/sample_code/tech/java/codesnippet/j2ee/ejbbestpractices/EJB-Best.html

[SUN] Sun suggests several J2EE Patterns on their home page: http://java.sun.com/blueprints/patterns/index.html

[ServerSide] The Page-By-Page Iterator Controversy: http://www.theserverside.com/patterns/thread.jsp?thread_id=12899

[J2EE] Specification: http://java.sun.com/j2ee/

[EJB2.1] Specification: http://java.sun.com/j2ee/

[SOFTWARE] Ian Gorton, Anna Liu, Paul Brebner: "Rigorous Evaluation of COTS Middleware Technology" in: I2EE Software, Vol. 36 No. 3, pg. 50-55

Sun's Java Foundation Classes homepage: http://java.sun.com/products/jfc/