Handling Large Database Result Sets

THE ONE-WRITE APPROACH IN LARGE-SCALE APPLICATIONS

BY **THOMAS KRUSE & ALOIS LECHICKI**

The Value List Handler is a well-known design pattern for dealing with large database results. There are, however, many trade-offs to consider when implementing this pattern. Here are some practical tips to make the pattern work, especially in large-scale J2EE applications.

Many J2EE applications include the requirement to display lists of database records as a result of the user search. Such search operations can involve processing large data sets on the server side. For example, think of the typical search in an online product catalog containing several thousand products.

Applications processing large result sets can face scalability and performance issues when the number of concurrent clients increases. When a query is executed according to user-supplied search criteria there may be no way to anticipate how much data will be returned. The result set might include a huge amount of data, so you cannot blindly send the query result back to the client without causing undue performance problems.

For managing large database result sets the design pattern Value List Handler is frequently used. The J2EE Blueprint recommends using this pattern to control the search, cache the results, and provide the results to the client. In this solution, when the client wants to search data the ValueListHandler is called which intercepts the client search request and returns data iteratively in chunks of pages.

The ValueListHandler provides query execution functionality and results caching. It implements an Iterator interface (see Figure 1).

The ValueList is a collection class that holds a set of Transfer Objects representing database records. In this design pattern the handler does not return the entire result list but only a subset of data. In a production application the ValueListHandler is usually hidden behind a façade according to the Business Service Façade pattern.

The J2EE Blueprint describes some general implementation strategies for the Value List Handler pattern. For example, it recommends using a Data Access Object (DAO) instead of entity beans if performance is an issue. However, when implementing a particular application a developer needs to consider further design trade-offs in order to meet specific functional and performance requirements.

In this article we present an implementation of the Value List Handler pattern proven in a large-scale WebLogic project. You can apply this approach when you develop an application that:

- Is to be used by several thousand concurrent users
- Has to display some part of the user search results in chunks of pages
- Assumes that the paging is consistent across requests, i.e. the underlying data is not expected to change when scrolling the result set

A Case Study

A travel retailer decided to develop a Web-based system offering travel agents direct access to information and reservation systems of a wide range of travel providers. The system was to be used by thousands of travel agents worldwide. Agents could select from over 100 million package holidays and over 5 million last-minute travel

AUTHOR BIO

Thomas Kruse has been working in the IT industry for 13 years. He has experience with J2EE technology and has completed two-large scale J2EE applications during this time. His professional background ranges from developer to system designer to system architect.

Alois Lechicki is a principal consultant at Softlab, Munich. He works with customers to help design and implement J2EE architectures. Alois has more than 15 years of experience in software engineering.

CONTACT...

thomas.kruse@kruse-it.de
alois.lechicki@softlab.de

bargains, updated daily. The system enabled users to search for a desired holiday package, check room and flight availability, then book and track customer reservations.

The site had to support the following use cases:

- A travel agent enters a customer's requirements, such as holiday destination, hotel facilities, room furnishings, and flight time.
- The site generates travel options based on requirements provided by the agent.
- The customer selects a travel option and the agent makes reservations.
- The agent completes the transaction by supplying a credit card number.

The second use case required running complex queries in the database containing package holidays. Searching in this database could produce result sets of extremely different lengths. Depending on the holiday destination, hotel category, room furnishings, and flight time, one could find a small or a huge number of options. However, the travel agent was not expected to browse through long result lists. She or he was rather expected to repeat the search operation after narrowing the search criteria.

Consequently, one requirement was to sort the result list according to a customer's preferences and then to cut it at a given position. The retailer estimated that it was enough to show only the first few hundred holiday packages.

Another requirement was to avoid a heavy load on online reservation systems when checking for availability. Thus the result list should not be checked at once but rather in chunks of, say, 20 maximum package holidays. Consequently, one design choice was to implement a page-by-page iterator based on the Value List Handler pattern. In this case, we could assume that the paging was consistent across requests. In other words, the underlying data in the database was not expected to change when scrolling the returned result set.

Since the application was required to have a sophisticated GUI with a complex workflow, another design choice was to develop a rich client using the applet technology. As a result, the client-side approach was used to manage session information (a session identifier) across multiple requests.

The server-side application design followed the principle of a service-oriented architecture (SOA). Groups of methods are clustered together in the services layer and exposed to the client via a service object – an EJB session bean. For example, the session bean HolidayPackageServices provides methods for searching and booking holiday packages.

For transferring data between components the design pattern Data Transfer Object (DTO) was applied. However, in order to minimize the impact of frequent changes in the domain layer, we decided to use dynamic data transfer objects (DDTO; see sidebar).

The system was implemented using BEA WebLogic Server 8.1 SP1 and an Oracle database for storing data related to package holidays. The application accessed online reservation systems of tour operators via SOAP over HTTP(S). The client (applet) accessed the server-side services via the Web services infrastructure provided by WebLogic Server. This was easy to develop using sever tools that generate Web services interfaces and require little or no coding. Figure 2 shows an overview of the application architecture.

Problems

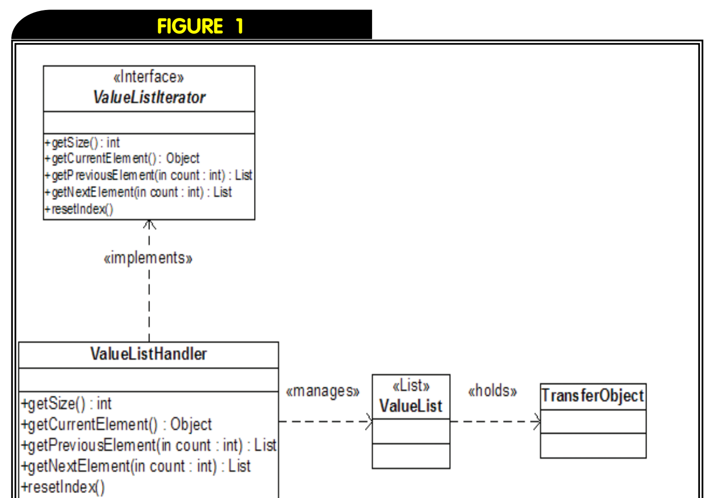
Since the application had to manage several thousand concurrent clients, we decided to use the stateless approach. However, search operations in the database turned out to be too resource intensive to be repeated for each paging request. Thus one design decision was to store search results in the database whenever a query result was too large to be returned to the client in one chunk.

A holiday package was represented by a graph of objects, called package items. In the first implementation packages were stored in the database as full-blown object graphs using an object-to-relational mapping (O/R-mapping). Unfortunately, load and performance tests soon revealed a poor response time primarily due to extensive interactions between the application server and the database server. Each "store query result" request led to several hundred "writes" of the package item records.

Improved Solution

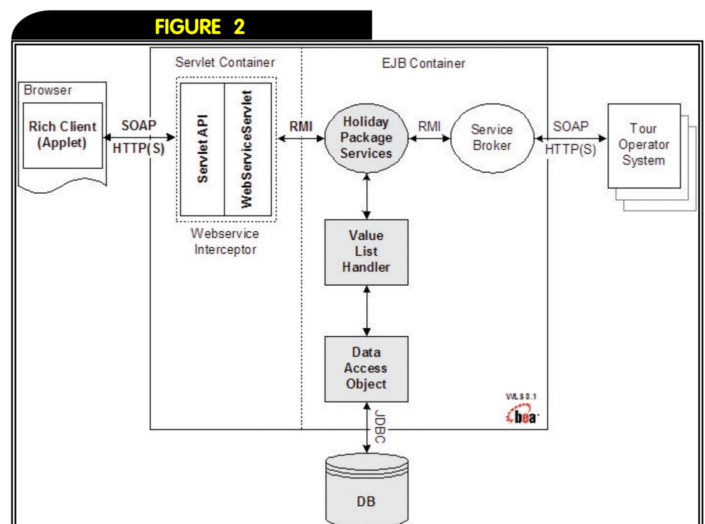
Due to the performance issues described above, the original implementation of the Value List Handler pattern had to be changed. The key idea was to minimize the number of "write" operations. We started by storing all items of a holiday package in one Oracle BLOB field. The next improvement was to write the result set in chunks of pages, each chunk as one BLOB. Finally we decided to store the entire result set (i.e., the actually fetched part

FIGURE 1



Class diagram for the Value List Handler pattern

FIGURE 2



Application architecture overview

of it) in a BLOB field in a single "write" operation.

After executing a query according to user-supplied search criteria, the first N records were fetched (if available) and stored in a list of dynamic data transfer objects (DDTOList). The number N was configurable and could be changed at run time. When the actual number of returned records was greater than the page size, the DDTOList was serialized and stored as a single record in the HOLIDAY-_PACKAGE_BAG table (see Table 1). Of course, a clean-up mechanism was implemented as well.

In this solution, the record containing the result set was read each time the client required the next page. It means that more data was actually read than needed. However, we found that in this trade-off "read" operations were less expensive than "write" operations.

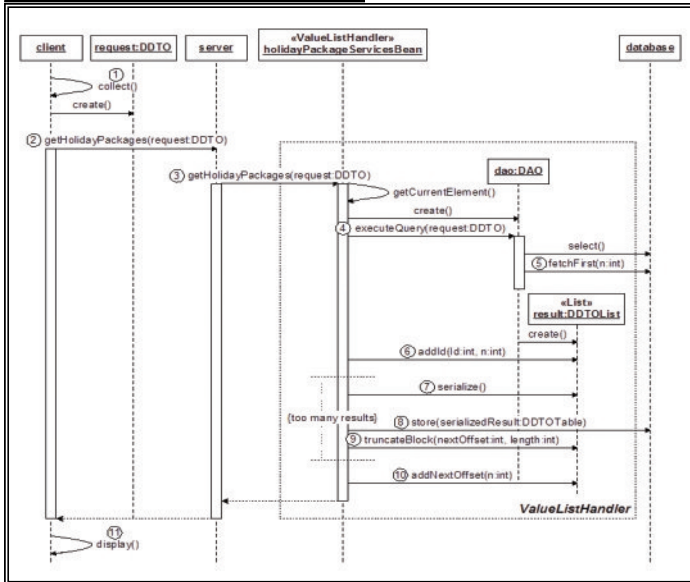
Finally, further load and performance tests confirmed that the "one-write" solution improved the performance of the system by an order of magnitude compared to the initial implementation.

TABLE 1

COLUMN NAME	DATA TYPE	REMARK
ID	NUMBER(16)	Primary key
RESULT_SET	BLOB	Serialized DDTOList object containing query result
CREATION_TIMESTAMP	DATE	Timestamp of record creation

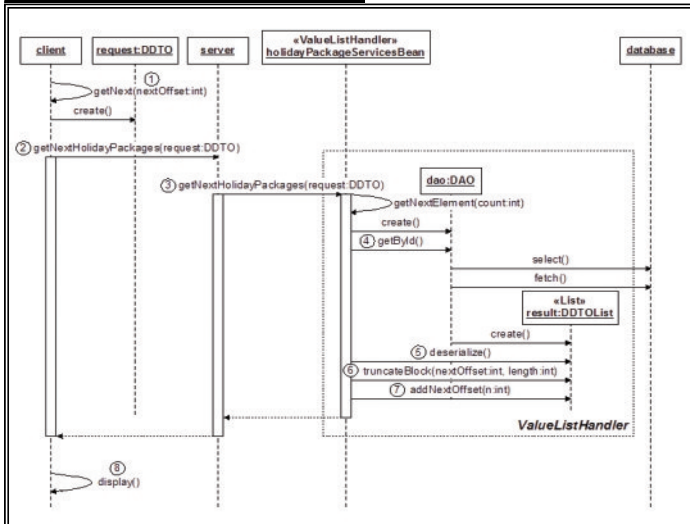
Database table for storing result sets

FIGURE 3



Sequence diagram: get the first chunk of holiday packages

FIGURE 4



Sequence diagram: get next block of holiday packages

Under the Hood

Let's look a little more closely at the implementation – and consider a typical use case.

When selling holidays the travel agent asks the customer about his wishes for a vacation destination, hotel facilities, required room furnishings, and so on. After supplying all mandatory information the travel agent sends a request to the system and waits for the first page of holiday packages. He then presents the result list to the customer, asking him to make a choice. However, it may happen that none of the offers on the list appeals to the customer. In this case the travel agent either asks the system for the next page or he changes the search criteria and repeats the search request.

From a technical viewpoint, this use case can be split into two sequence diagrams, which are discussed in the next sections.

Scenario 1: Get First Block of Holiday Packages

This scenario (see Figure 3) shows the flow of control when the customer's request is defined and the search operation is started.

- ***Message (1):*** At first, all information required for the getHolidayPackages request is collected by the applet. The client-side workflow supports the user in entering only a valid set of search criteria. This data is stored in a dynamic data transfer object (DDTO).
- ***Message (2):*** After filling the DDTO, the applet calls the Web services method getHoliday-Packages using a local stub object. The stub serializes the DDTO to a SOAP message and sends it via HTTPS to the server.
- ***Message (3):*** When the server receives the SOAP message it deserializes its payload and puts the data into a DDTO. Then it calls the method getHolidayPackages of the stateless session bean HolidayPackageServicesBean (for better readability we have skipped aspects of session tracking and security and authorization issues).
- ***Message (4):*** Now a DAO object builds a select statement using the search criteria contained in the DDTO and executes the query. In this process, the result set is sorted according to user preferences and business rules.
- ***Message (5):*** The DAO object gets the first N records from the database (the number N is configurable), if available, and stores them in a DDTOList object. When the DAO completes reading records, it closes the database connection.
- ***Message (6):*** A unique holidayPackageBagId is created and stored in the DDTOList object. This id has to be supplied by the client among other items in subsequent requests to identify the result set.
- ***Messages (7) – (9):*** These messages are sent only if the actual number of retrieved records exceeds the (configurable) page size. The DDTOList is serialized using the standard Java mechanism and stored as one record (as BLOB) in the HOLIDAY_PACKAGE_BAG table. The holidayPackageBagId is

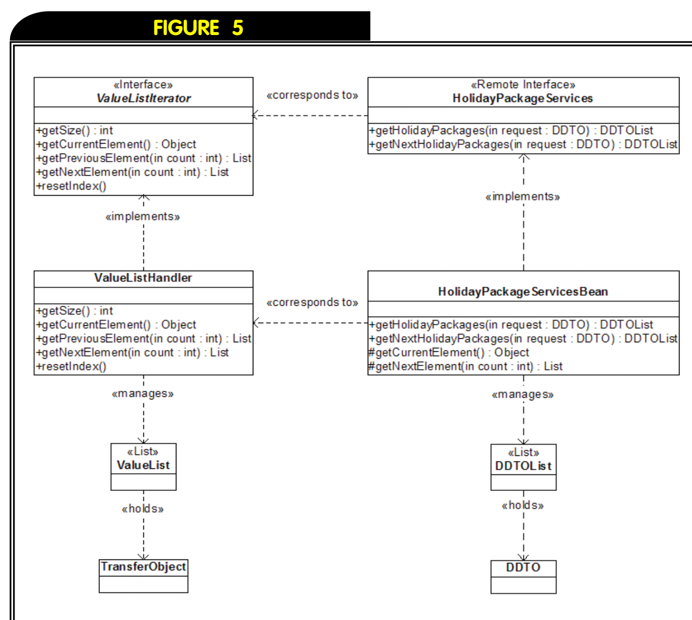
used here as key. After that, the DDTOList is so truncated that it contains only the first chunk of the result list and the length of the original list.

- ***Message (10):*** The position of the next page nextOffset is added to the DDTOList and the list is sent back to the client using the WLS WebServices infrastructure.
- ***Message (11):*** The client receives the response and displays the first page of the result list. The total number of records available on the server is displayed as well.

Scenario 2: Get the Next Block of Holiday Packages

The second scenario (see Figure 4) shows the flow of control when the client asks for the next chunk of the result list. It means that the initial search operation returned a record set having more records than one page could contain.

FIGURE 5



Class diagram: the implemented Value List Handler pattern

DYNAMIC DATA TRANSFER OBJECTS (DDTO)

The concept of dynamic data transfer object (DDTO) is based on the Generic Attribute Access pattern. This approach is used to minimize the impact of frequent changes in the domain layer on the rest of the system. A DDTO is a generic data container that can hold an arbitrary set of data. It provides a generic attribute access interface using HashMaps and key-value notation.

DDTOs implemented in the case study could contain both simple data types (String, Integer, etc.) and complex ones (structures). Moreover, nested DDTOs and lists of DDTOs (DDTOList) were supported as well.

The implemented framework provided a type-safe access to the content of a DDTO. It also contained superclasses for DAOs and entity beans implementing completely generic interface methods. Thus DAOs and entity beans could expose a unified interface to their attributes, with almost no extra coding required.

- ***Message (1):*** The applet creates a dynamic data transfer object (DDTO) and puts both holidayPackageBagId and nextOffset into it. Note that it is up to the client to manage session information.
- ***Message (2):*** The applet calls the Web services method getNextHolidayPackages using a local stub object. The stub serializes the DDTO to a SOAP message and sends it via HTTPS to the server.
- ***Message (3):*** The server deserializes the message payload and puts the data into a DDTO. It then calls the method getNextHolidayPackages of the stateless session bean HolidayPackageServicesBean.
- ***Message (4):*** A DAO object is used to retrieve the record corresponding to holidayPackageBagId from the table HOLIDAY_PACKAGE_BAG.
- ***Messages (5)–(7):*** The BLOB field RESULT_SET is deserialized into a DDTOList object using the standard Java mechanism. Let us recall (see Scenario 1) that the DDTOList contains all holiday packages retrieved in the initial search operation. Now using the parameter nextOffset the next chunk of the result list is created and sent as DDTOList back to the client. Before that, the value of nextOffset is updated so that it points at the subsequent page.
- ***Message (8):*** The client receives the response and displays the next page of the result list.

Back to the Pattern

In the solution presented above the HolidayPackageServices Bean implemented only a part of the interface ValueListIterator. For example, the method getPreviousElement was not needed because the client (applet) cached read records. From the client point of view it was enough to provide only two methods of the ValueListIterator: getHolidayPackages and getNextHoliday Packages. Figure 5 shows the Value List Handler pattern as implemented in the case study in relation to Sun's archetype.

Conclusion

In the case study presented here we used the Value List Handler pattern to provide a Web service for searching holiday packages. In this pattern long result sets are sent to the client iteratively in chunks of pages.

While this approach reduces network overhead, caching of result sets can still be an issue. Stateful session beans are not always an option if you have several thousand concurrent users and the ValueList object has to hold long result lists.

The key design decision in the case study was to use stateless session beans and to store result sets, longer than one page, in the database. The chosen implementation improved the performance of database operations by reducing the number of write operations. This was done by storing result sets in a compact form as Oracle BLOBs.

An additional advantage of this approach was easy deployment of the load balancing mechanism.

References

- *Sun Microsystems:* http://java.sun.com/blueprints/corej2eepatterns/Patterns/ValueListHandler.html
- *SOA:* webservices.xml.com/pub/a/ws/2003/09/30/soa.html
- Marinescu, Floyd. (2002) *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. Wiley Computer Publishing.