

Distributed Result Set Iterator: a design pattern for efficient retrieval of large result sets from remote data sources

Brad Long*

Australian Development Centre,
Oracle Corporation,
300 Ann St, Brisbane,
QLD 4000, Australia.
Email: brad.long@oracle.com

Abstract

Retrieving large amounts of information over wide area networks, including the Internet, is problematic due to issues arising from latency of response, lack of direct memory access to data serving resources, and fault tolerance. As both the public and industry require an ever increasing amount of data to be transferred in a timely fashion, we face the challenge of providing the technology to meet these consumer-driven demands. We must be able to deliver large amounts of information promptly with immediate feedback to user requests. In addition, this must be achieved in a robust manner whilst conserving computing resources. This paper describes a design pattern for solving the issues of handling results from queries that return large amounts of data. Typically these queries would be made by a client process across a wide area network (or Internet) to a relational database residing on a remote server. The solution is discussed in detail in this paper and has recently been incorporated into the framework of a commercial software product developed at Oracle Corporation.

1 Introduction

As computer systems are becoming more distributed in nature and an ever increasing amount of information is being demanded by the public, the requirement for efficient retrieval of large amounts of information over wide area networks is of paramount importance.

Distributed computing inherently adds complexity to computer software. Latency, distributed memory management, heterogeneous networks, fault tolerance and security are some of the issues contributing to this added complexity [5, 9, 12]. Specific key challenges concern server reliability, availability, and robustness; bandwidth allocation and sharing; and scalability of web and data servers [10].

Object oriented languages also add extra challenges. Results from a query are usually returned in some form of result set object. When the query is requested against a local data source, the iterator on the result set returned can be a reference to a current database row. That is, the results can be *streamed*. However, a truly robust distributed query is stateless, so does not retain a reference to a row in the remote database. A robust query can only deal with the results transmitted over the network. Now, what if the result contains millions of rows? The client machine may run out of memory whilst trying to deserialize the returned results into a collection of millions of objects. In this case, clearly the performance of the system will also be impacted.

Design patterns capture solutions that have developed and evolved over time. They assist in communicating problems and providing solutions to those problems. Patterns allow us to document a known recurring problem and its solution in a particular context, and to communicate this knowledge to others. One of the goals of a design pattern is to foster conceptual

*also School of ITEE, University of Queensland, Brisbane, QLD 4072, Australia. Email: brad@itee.uq.edu.au

reuse over time. Often they reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form.

We propose five requirements for robust, efficient, distributed retrieval of large result sets. Specifically, results returned from a query:

1. must conserve resources
2. must respond within a reasonable time
3. must be robust
4. must not be limited to a partial result set
5. must be intuitive to use

This paper presents a design pattern for retrieving large amounts of information from remote data sources in a robust and efficient manner. It adds to previous work [8] by providing explicit details of the design pattern in the familiar format and style found in [4]. The design pattern is called a Distributed Result Set Iterator and satisfies the five requirements mentioned above.

Related work is reviewed in Section 2. In Section 3 the five requirements are described in more detail. The design pattern is presented in detail in Section 4. Additional enhancements to the pattern are discussed in Section 5.

2 Related Work

Sun's page-by-page iterator pattern [7] goes some way to solving the issues of retrieving large amounts of data from remote data sources. However, it requires the developer to consider *pages* of information being fetched from the database. The approach presented in this paper abstracts such issues. Sun's page-by-page iterator was specifically designed for cases when:

- the user will be interested in only a portion of the list at any time.
- the entire list will not fit on the client display.
- the entire list will not fit in memory.
- transmitting the entire list at once would take too much time.

Although the last two bullet points are important to consider and are two of the issues our design pattern addresses, we do not require developers to consider the first two bullet points. Although these points may be satisfied in many applications, they are not required to be considered when applying the Distributed Result Set Iterator. In addition, the application developer is not required to think in terms of *pages* as they are with the page-by-page iterator.

It is also important to note that Sun's page-by-page iterator is not a robust iterator. By design, the iterator does not keep its own copy of the list being traversed. Consequently, insertions or removals will interfere with the traversal; the iterator may access a remote item twice or miss it completely. This issue is considered in the Implementation section. We provide the application developer with a robust iterator.

Other similar work has concentrated on using cached objects with enough information to connect back to the server to request more information [2]. However, such solutions require resources to be held open on the server, waiting for client responses.

3 Five Requirements of Robust, Efficient, Distributed Queries

Requirement 1: Must conserve resources

Since there is no restriction on how many rows (or objects) a query may return, resources must clearly be conserved to cater for the potentially unlimited set of results. Conservation of resources is achieved on the server by never holding onto any database resources. For example, a local implementation may use an iterator or placeholder to store the current position

in the result set. Remote queries should not do this. Remote queries should not rely on server resources being held open whilst clients are iterating over the results. Imagine a user browsing pages of results, one at a time. The user decides to get a cup of coffee after browsing one of ten pages. The resource is being held open on the server. Now, imagine hundreds of clients doing this. Consider how inefficient and unacceptable this strategy is. The application will not scale. In addition, on the client side, potentially millions of objects need to be created before returning the results to the client. Creating millions of objects may consume vast amounts of memory. Also, it may take a significant amount of time, thus violating the Second Requirement.

Requirement 2: Must respond in a reasonable amount of time

The query should take about the same time to execute as a local query (i.e. within the same order of magnitude). That is, it must respond with the first row(s) of information with the same latency as a local query.

Requirement 3: Must be robust

A robust query is one in which concurrent insertions and deletions, made by other client processes do not cause loss or duplication of data in the results of a query during result retrieval.

Requirement 4: Must not be limited to a partial result set

This requirement simply states that any method for retrieving information must not limit the number of rows returned. All rows of query data should be returned from the server.

Requirement 5: Must be intuitive to use

The client developer should find retrieving data similar to current local query approaches. In addition, the client using the query method should not be required to make allowances for the fact that it is a remote query. They should be able to use the same client code for local and remote queries.

4 The Design Pattern: Distributed Result Set (DRS) Iterator

4.1 Intent

Provide a way to access the elements of a very large query result obtained from a remote data source in a robust, efficient, and easy-to-use manner.

4.2 Motivation

As with the Iterator pattern [4], an aggregate object such as a list should give you a way to access its elements without exposing its internal structure. In addition, the DRS Iterator hides the mechanics involved in retrieving chunks or *pages* of data from the server.

When retrieving very large amounts of information:

- The entire set of results may not fit into memory. In object-oriented systems the results returned from the server are often constructed as a collection objects on the client machine. Rows of data equate to objects in memory. Therefore, if the results are very large, the client may run out of memory when attempting to construct the objects.
- Transmitting the entire list at once would take too much time. This is related to both network bandwidth and creating client objects. Sending massive amounts of information over a remote network could degrade performance to unacceptable levels. In addition, creating client objects could take a long time for very large results. For example, a query returning one million rows would require the client to create one million objects. This could severely impact response time.

By breaking up the query result into pages, it eliminates the problem of network bandwidth contention, places very little demand on client memory requirements, and improves query response time.

The DRS Iterator solves these problems and allows you to access very large results from remote data sources as you would access the elements from any aggregate object using a standard Iterator. To the user, the DRS Iterator appears exactly as a normal Iterator. However, the DRS Iterator is actually a combination of collaborating classes and patterns. The standard Iterator pattern is used as a Facade [4] into a more complex subsystem that retrieves pages of information from the remote data source. For example, a ResultsIterator implements the Iterator interface. The ResultsIterator calls for a DataRetriever with the relationship as shown in Figure 1.

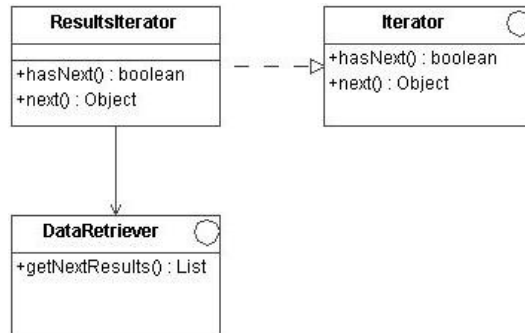


Figure 1. Relationship between ResultsIterator and DataRetriever

Before you can instantiate ResultsIterator, you must supply a DataRetriever which manages the retrieval of the data pages from the server as a result of a query request. Once you have the ResultsIterator instance, you can access the elements of the query results sequentially. Java already provides an adequate Iterator interface (java.util.Iterator). The *hasNext* operation returns true if there are more elements, and *next* operation returns the next element in the iteration. The *remove* operation is optional and is not implemented in the DRS Iterator pattern.

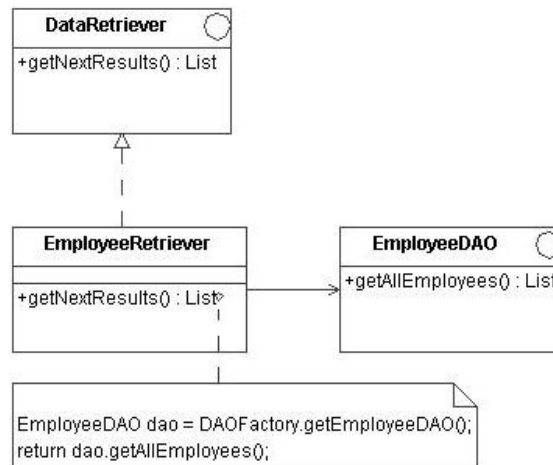


Figure 2. Relationship between DataRetriever and Data Access Object

Separating the DataRetriever from the ResultsIterator allows you to specify different queries (or data page retrieval strategies) to provide the result list for the iterator to operate on. This includes holding specific state information (for example, key values) for in-progress queries. Here, a java.util.List collection is returned by the DataRetriever. This could also be an array of typed objects, or even an array of Value Objects [1]. The DataRetriever is simply an Adaptor [4] which adapts the interface of a Data Access Object (DAO) [1] to the interface required by the ResultsIterator. A DAO is used to abstract and

encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data. The DataRetriever and DAO interact as shown in Figure 2.

A DAOFactory is used to return a particular DAO implementation. For example, the DAO may be implemented using JDBC or some other data access technology. Based on application configuration, the DAOFactory would return the appropriate data access technology implementation. More information on the Abstract Factory and Factory patterns is contained in [4]. The getNextResults method of the DataRetriever delegates to the required method of the DAO which performs the query. For applications running locally, the DAO would connect and execute the query directly. For clients requesting queries from a remote server, the DataRetriever could delegate directly to a remote proxy (see Figure 3).

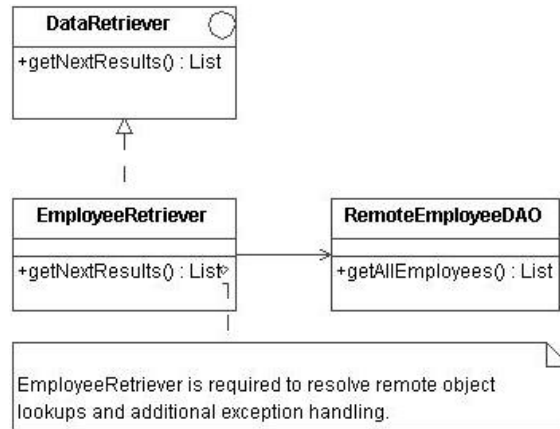


Figure 3. DataRetriever delegating directly to a remote object

However, this requires the DataRetriever to handle the added complexity of remote object lookups and requires it to change implementation depending on deployment mode. The implementation change occurs since remote interfaces often require extra exceptions on their method signatures [6, 11]. To deal with this issue, the client DAOFactory would return an implementation that would adapt the remote interface to the expected DAO interface. This means that the DataRetriever would not need to change implementation since the remote adaptor would implement the same interface as the local implementation. The actual DAO implementation on the server would receive the request via the remote call. Figure 4 illustrates this approach.

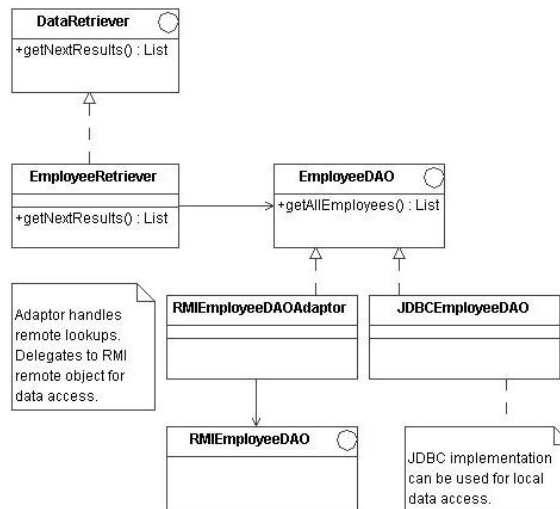


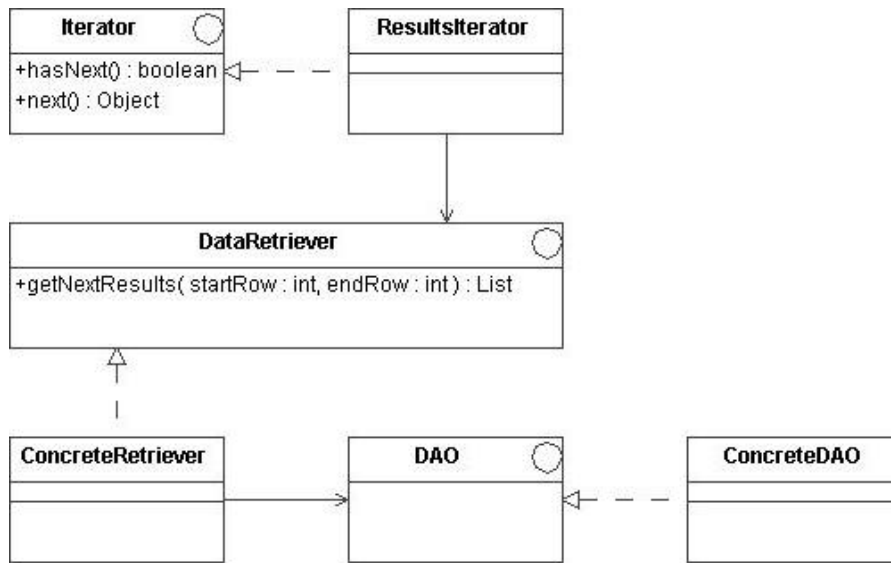
Figure 4. Using an Adaptor to adapt the remote interface

4.3 Applicability

Use the DRS Iterator pattern

- to efficiently access the very large results of a remote query.
- to provide a well-known and intuitive interface which abstracts the underlying mechanics of remote result set retrieval.
- to access the collection of query results without exposing its internal representation.
- to provide a uniform interface for traversing different collections of query results (that is, to support polymorphic iteration).

4.4 Structure



4.5 Participants

- **Iterator**
 - defines an interface for accessing and traversing elements.
- **ResultsIterator**
 - ResultsIterator is a ConcreteIterator (as described in the standard Iterator pattern [4]) that abstracts the iteration of results over very large result sets.
 - implements the Iterator interface.
 - keeps track of the current position in the traversal of the collection of results.
 - delegates to the DataRetriever for query result retrieval.
- **DataRetriever**
 - defines an interface for retrieving pages of results.
- **ConcreteRetriever**

- implements the DataRetriever interface.
- requests pages of information from the remote server.
- delegates to the Data Access Object for executing the query.

- **DataAccessObject (DAO)**

- defines an interface for executing a query against a data source.

- **ConcreteDAO**

- implements the DAO interface.
- executes a query against the remote data source.

4.6 Collaborations

- The ResultsIterator keeps track of the current object in the collection of results (or data page) returned from the remote server and can compute the succeeding object in the traversal. It also keeps track of the current data page itself. When there is no more data in the current data page, the ResultsIterator requests more information from the DataRetriever.
- The DataRetriever is an adaptor that converts a DAO's interface to the interface required by the ResultsIterator. The DataRetriever delegates requests for information to a DAO.

4.7 Consequences

In addition to the consequences of the standard Iterator pattern [4], the DRS Iterator pattern has three important consequences:

1. *It provides an efficient and robust method of retrieving very large remote result sets.* Result sets of many rows (e.g. millions) can be efficiently traversed in a simple manner. The DRS Iterator does not hold onto any server resources between retrieving data pages. On the client side, the DRS Iterator constructs only enough objects to hold the current data page.
2. *It simplifies traversal of very large remote result sets.* Retrieving result sets can be complex. The DRS Iterator presents a uniform approach to the client for dealing with the complexities in the retrieval of very large data sets from remote data sources.
3. *More than one traversal can be pending on a result set.* A DRS Iterator keeps track of its own internal state. Therefore you can have more than one traversal in progress at once.

4.8 Implementation

1. *How is the DRS Iterator used for both small and large result sets?*

If you want to use DRS Iterators for all queries, make sure that you take the data page size into account. For example, for a table with millions of rows, you may want to set the page size to around 2000 rows per page. However, for a table that will only ever contain, say, 3000 rows, set the data page size to 3000 or above. This allows the DataRetriever to get all results from the table in one fetch. Otherwise, the DataRetriever would make two remote calls, one for the first page of 2000 rows and one for the second page of 1000 rows.

2. *How big should the data page fetch size be?*

This depends on the tables you are querying. For very large tables with millions of rows, anywhere from 5000 to 10000 should be adequate. If you are looking for maximum efficiency, experimentation may be required since it depends on the capabilities and resources of the systems the application will be running on. Data page sizes of as low as 1000 rows have proved to be adequate.

3. How do you use the DRS Iterator?

The following is an example of how a client developer would use the DRS Iterator:

```
Iterator iter = new ResultsIterator(new EmployeeRetriever());
```

4. How are data pages queried from the data source and how robust is the ResultsIterator?

To provide an efficient implementation, pages of data must be returned from the server. Each results in each data page are disjoint and the union of all data pages is the full set of results from the query. To retrieve data in pages, *query slicing* is used.

As an example, consider the single query:

```
SELECT PATIENT_ID, PATIENT_NAME FROM PATIENT
```

This may potentially return many thousands of rows of information. This query can be sliced by surrounding the single query with the slicing code. Three options are presented: 1) a vendor specific (Oracle) version, 2) an ANSI SQL version, and 3) a hybrid approach. It should be noted that the ANSI SQL version is extremely slow. Many database vendors provide enhanced SQL language features that allow the developer to select a range of rows based on some artificial row number. It is suggested that the more efficient row selection capability of a particular database is used. In the following examples, slices of 1000 rows are demonstrated.

An efficient, but not robust, query

```
SELECT *
FROM (SELECT ROWNUM RNUM, INLINE_VIEW.*
      FROM (SELECT PATIENT_ID, PATIENT_NAME
            FROM PATIENT
            ) INLINE_VIEW
      )
WHERE RNUM BETWEEN x AND y;
```

Figure 5. Oracle specific slicing technique

Figure 5 details the SQL for the Oracle specific slicing technique. Assuming our slice size is set to 1000, the first time the query is called $x = 1$ and $y = 1000$. After each call, the values of x and y are updated for the next call to the query as follows: $x = y + 1$ and $y = y + 1000$. The next call to retrieve data will use $x = 1001$ and $y = 2000$.

```
SELECT *
FROM (SELECT ROWNUM RNUM, PATIENT_ID, PATIENT_NAME
      FROM PATIENT
      )
WHERE RNUM BETWEEN x AND y;
```

Figure 6. Simplified Oracle specific slicing technique

Although the query in Figure 5 can be simplified to the one in Figure 6, it does not require modification to the original SQL. The simplified query requires modification to the SQL, specifically the addition of the `rownum` field to the original query, whereas the structure used in Figure 5 allows the original unmodified query to be wrapped with the slicing code.

This solution is robust only for read-only tables. But, it is not robust if data is inserted during retrieval. The problems are described in the following example:

Consider a table of more than 1000 rows (e.g. 10,000).

- (a) A data page of 1000 rows is retrieved, specifically, rows 1 to 1000.
- (b) A new row is inserted somewhere into the lower 1000 records in the database table.
- (c) Then, rows 1001 to 2000 are retrieved. But, since a row has been inserted, row 1001 has already been retrieved. It was row 1000 last time! Hence, the same row has been retrieved twice.

Also, if a row was deleted in (b), we would actually miss a row of data, since row 1001 would become row 1000, the number of which had already been retrieved.

A robust, but inefficient, query

The solution to the problem of stateless retrieval is not to rely on `ROWNUM`, but to rely on the primary key values of the table being accessed. The ANSI SQL slicing technique (Figure 7) is quite different. The value *id* will begin as some initial low value, for a character column this might be a space character. It is the primary key, in this case patient identifier, of the last row returned from the previous execution of the query. Unfortunately, the query's performance is completely unacceptable.

```
SELECT P1.PATIENT_ID, P1.PATIENT_NAME
FROM PATIENT P1
WHERE P1.PATIENT_ID > id AND
1000 > (SELECT COUNT(*)
        FROM PATIENT P2
        WHERE P1.PATIENT_ID > P2.PATIENT_ID AND P2.PATIENT_ID > id)
```

Figure 7. ANSI SQL slicing technique

Again, consider a table of more than 1000 rows (e.g. 10,000).

- (a) A data page of 1000 rows is retrieved.
- (b) A new row is inserted somewhere into the lower 1000 records in the database table.
- (c) Then, the next 1000 rows are retrieved. But, the new row has an *id* less than the one we are looking for, so it is correctly ignored, and does not throw out our query, that is, the next correct set of rows is retrieved.

Also, if a row was deleted in (b), we would not miss a row of data, because we are not relying on ordinal position, but primary key value. This approach is robust but is inefficiently executed by database SQL engines. One reason is that the join on the patient tables *P1* and *P2* is asymmetric (a '`>`' is required for the join on `PATIENT_ID`).

An efficient and robust query

By combining the two approaches, we can come up with a robust and efficient approach (Figure 8).

```
SELECT *
FROM (SELECT ROWNUM RNUM, INLINE_VIEW.*
      FROM (SELECT PATIENT_ID, PATIENT_NAME
            FROM PATIENT
            WHERE PATIENT_ID > id
            ) INLINE_VIEW
      )
WHERE RNUM < 1001
```

Figure 8. Hybrid slicing technique

As with the ANSI SQL approach, *id* is the primary key value of the last row retrieved from the previous call. This query combines both the relatively efficient *rownum* approach with the robust primary key approach, providing us with

a solution for using iterators in stateless situations. Performance is good and as the latter data pages are retrieved, the query performs even more efficiently. This is due to the WHERE clause of the inner (wrapped) query causing the return of a diminishing result set as the *id* increases.

For tables with composite keys, instead of *where key > id* use:

- for 2 columns: *where key1 >= id1 and key2 > id2*.
- for 3 columns: *where key1 >= id1 and key2 >= id2 and key3 > id3*.

That is, all comparisons are \geq except the final comparison which is $>$.

Using this method the query can be cut into manageable data pages. These examples use data pages with a page size of 1000 rows. Other page sizes may be chosen as appropriate. This solution allows all server database resources to be released between queries, providing a scalable and robust solution.

5. How responsive is the DRS Iterator?

The responsiveness depends on the query executed. Run the query locally, without slicing. The time to execute the query (that is, the time to return the first row) will be about the same time that the DRS Iterator will return the first row. The DRS Iterator does not enhance the original query execution time. It handles the complexities of bundling and transmitting the resulting data from a query on a remote system.

4.9 Sample Code

We'll look at an implementation of a DRS Iterator that retrieves a list of all employees from a remote data source. The iterator interface will be Sun's `java.util.Iterator` (see Figure 9).

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
}
```

Figure 9. Iterator interface definition

1. Implementing ResultsIterator

`ResultsIterator` is a concrete implementation of the `Iterator` interface (see Figure 10). The variable `pageSize` is the actual number of rows to be returned. The results for each data page are held in `page`. The variable `isEndOfServerResults` returns *true* if there are no more results to come from the server. The `index` variable stores a cursor into the current row of the data page. The `DataRetriever` for the `ResultsIterator` is stored in `retriever`.

```
public class ResultsIterator implements Iterator
{
    private int pageSize;
    private List page;
    private boolean isEndOfServerResults;
    private int index;
    private DataRetriever retriever;
    public ResultsIterator(DataRetriever retriever, int pageSize) { ... }
    public boolean hasNext() { ... }
    public Object next() { ... }
    private getNextPage() { ... }
}
```

Figure 10. Skeleton implementation of ResultsIterator

A `DataRetriever` is required to be passed into the `ResultsIterator` constructor (see Figure 11). The `getNextPage` method in the constructor delegates to the `DataRetriever` to request a page of data from the server.

```

public ResultsIterator(DataRetriever retriever, int pageSize) {
    this.retriever = retriever;
    isEndOfServerResults = true;
    index = 0;
    this.pageSize = pageSize;
    // get the first page of data
    getNextPage();
}

```

Figure 11. ResultsIterator constructor

The `hasNext` and `next` methods are required to be implemented to meet the contract of the `Iterator` interface (see Figure 12).

```

public boolean hasNext() {
    // There is more data if we haven't received
    // the last of the server data or we have more
    // data in our current retrieved page.
    return (!isEndOfServerResults || index < page.size());
}
public Object next() {
    // If we haven't run out of rows in our current page
    // then return the next value
    if (index < page.size()) {
        ++index;
        return page.get(m_index-1);
    }
    // If we have exhausted ALL data from the server,
    // throw an exception, as per the standard operation
    // of Sun's iterator implementations.
    if (isEndOfServerResults) throw new NoSuchElementException();
    // Otherwise, get another page of data from the server
    getNextPage();
    // recurse now to get some data from our cache
    return next();
}

```

Figure 12. Implementations of Iterator methods

The `getNextPage` method retrieves a new data page from the server when the data in the current data page is exhausted. Notice in Figure 13 that the request is delegated to the `DataRetriever` with the page size specified.

```

private void getNextPage() {
    index = 0;
    page = retriever.getNextResults(pageSize);
    startRow = startRow + page.size();
    isEndOfServerResults = (page.size() < pageSize);
}

```

Figure 13. Delegation to the DataRetriever to request data

2. *Implementing a DataRetriever*

The `DataRetriever` interface has only one method (see Figure 14). `ResultsIterator` relies on this method for delegation to the data retriever and ultimately to the appropriate data access object.

```
public interface DataRetriever { public List getNextResults(int pageSize); }
```

Figure 14. The DataRetriever interface

The `EmployeeRetriever` class implements the `DataRetriever` interface (see Figure 15). The class looks up the data access object from the DAO factory and makes the appropriate call to the data access object. In this case, the `getAllEmployees` method is called. Note that the key and page size is passed into the DAO method. Also, note that other parameters, as desired, may be passed into the DAO method. This last point demonstrates how the `DataRetriever` adapts the DAO interface to the interface required by the `ResultsIterator`.

```
public class EmployeeRetriever implements DataRetriever {

    // Holds the last key value retrieved
    String key;

    // Retrieves the next page of data from the server.
    public List getNextResults(int pageSize) {

        List page = null;

        // look up factory to get employeeDAO
        EmployeeDAO employeeDAO = daoFactory.getEmployeeDAO();
        page = employeeDAO.getAllEmployees(key, pageSize);

        // Store the key field of the last row retrieved
        key = ((Employee)(page.get(page.size()-1))).getEmployeeId();

        return page;
    }
}
```

Figure 15. An implementation of a DataRetriever

In cases where the data access object is remote, either a new `RemoteEmployeeRetriever` class can implement the remote lookup logic and adapt the remote interfaces to the data retriever interface, or the DAO factory can do this by returning a class that implements the appropriate DAO interface, in this case `EmployeeDAO`. The latter approach is more flexible. More information on implementing the Abstract Factory pattern is contained in [4]. The Data Access Object pattern is described in [1].

4.10 Known Uses

The DRS Iterator pattern has been used in situations requiring the retrieval of very large result sets from remote data sources. The pattern has been subsequently enhanced to include double buffering of data pages.

4.11 Related Patterns

An Iterator [4] is used to provide the consistent traversal over result sets. The Adaptor pattern [4] is used in two places. Firstly, to adapt the data access (DAO) method to the signature required by the `ResultsIterator`. Secondly, to adapt remote data access interfaces to standard data access interfaces. An Abstract Factory [4] for our data access objects helps contain implementation changes caused by deployment mode. The DRS Iterator acts as a Facade [4] into a more complex subsystem for retrieving very large result sets from remote data sources. Data Access Objects [1] are used to provide the handling of database connections and actual query execution.

5 Further Discussion

This section describes some of the techniques used in the implementation of the DRS Iterator and how they satisfy the Five Requirements proposed in this paper. Also, some additional enhancements that have been applied to the design are discussed.

5.1 Data Paging

A *result set* is defined as the set of all results of a query returned from the server. A *data page* is a subset of a result set. Each result set has n disjoint data pages, and corollary, the result set is the union of all n data pages.

Data is required to be passed from the server to the client in data pages to enable us to deal with huge amounts of data. A large result set could easily consist of hundreds and thousands of rows (or more). By setting the maximum number of rows for each data page to a manageable size, for example 1000 rows, it enables us to (a) limit the amount of data being sent across the network at a time, and (b) limit the number of objects that need to be constructed in memory. Eventually all data pages that make up the result set will be received (Fourth Requirement).

This ensures a speedy response to queries by keeping network bandwidth low and only requiring the server process to retrieve, package, and transmit 1000 objects at a time. It also gives the client an immediate response to large queries. This technique solves the speed of response (Second Requirement) and memory resource problems (First Requirement).

The iterator retrieves rows from the returned data page of size m rows. When the iterator reaches the last entry, a query is automatically sent to the server to fetch another m rows. On the server, m rows at a time can be retrieved by employing a technique we call *query slicing*.

5.2 Query Slicing

By using the query slicing technique (as described in the Implementation section of the design pattern) the query can be cut into manageable data pages. This solution allows all server database resources to be released between queries, providing a scalable (Second Requirement) and robust (Third Requirement) solution.

5.3 The Iterator Pattern: A Powerful Facade

The Iterator pattern [4] is used to provide a familiar level of abstraction to the application developer thus satisfying the Fifth Requirement. It is clear that returning an array of results is not a satisfactory solution. Since an array is of fixed length, it has all the shortcomings associated with deserializing the entire result set into objects before returning them to the client. Although we make use of fixed-length data pages in the transport layer when communicating with the server, the iterator provides an abstraction which allows the client to deal consistently with the entire result set. The concrete implementation (ResultsIterator) of the Iterator interface provides a powerful facade over a complex subsystem of remote data retrieval.

5.4 Double-Buffering Data Pages

Double-buffering is a technique used in graphics programming. A double-buffered system is one in which the image in one buffer is displayed while the image in the other buffer is computed [3].

This enhancement has been applied to data pages retrieved from a server. Initially, a data page is retrieved and loaded into one result set. Whilst the first result set is being used by the client the second result set is loaded. When the client has finished with the first result set, the empty result set is returned to the loader, and the second result set is given to the client.

5.5 Multithreaded Data Fetching

To retrieve the double-buffered pages into memory, multithreaded data fetching is required. When the iterator object is created, the first data page is retrieved. This is called the *primary* data page. A call to `next()` will return the first row in the query. In addition, for the first call to `next()` on the primary data page, a thread is spawned to retrieve the secondary data page. The secondary data page is added to a buffer for later use.

When the iterator has traversed all rows in the primary data page, the next call to `next()` automatically swaps the secondary data page with the primary data page. Now that this is the first call on a new primary page, a thread is spawned to

retrieve the secondary data page and the pattern continues. When all data has been retrieved from the server, `hasNext ()` is false.

6 Conclusion

The world's computers are already connected to vast repositories of information and demand for accessing and transporting this information continually increases. To meet this demand we must be able to provide the necessary delivery mechanisms and technologies. In this paper, a design pattern has been presented for retrieving large amounts of information from remote data sources.

The Distributed Result Set Iterator allows information providers to conserve resources, provide reasonable response times to user requests, and provide robust and full data access in an easy-to-use manner. The design pattern presented in this paper has improved on design patterns that require developers to think in terms of *pages* and other segmented data sets, by providing the developer with a standard iterator interface which abstracts such issues.

The approach satisfies the five requirements proposed for robust, efficient, distributed retrieval of large result sets:

1. must conserve resources
2. must respond within a reasonable time
3. must be robust
4. must not be limited to a partial result set
5. must be intuitive to use

Further work will consist of extending the design pattern query slicing technique when retrieving data in a variety of sort orders.

References

- [1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.
- [2] E. Chan and K. Ueda. Efficient query result retrieval over the web. In *Proceedings International Conference on Parallel and Distributed Systems*. IEEE Computer Society, 2000.
- [3] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 1990.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1999.
- [5] S. Ghosh and A.P. Mathur. Issues in testing distributed component-based systems. In *First ICSE Workshop on Testing Distributed Component-Based Systems*, May 1999.
- [6] Sun Microsystems Inc. Java remote method invocation (RMI) specification. Available online at <http://java.sun.com/products/jdk/rmi/>.
- [7] Sun Microsystems Inc. Page-by-page iterator. Available online at http://java.sun.com/blueprints/patterns/j2ee_patterns/page_by_page_iterator/.
- [8] B. Long. A design pattern for efficient retrieval of large data sets from remote data sources. In *Proceedings of the 4th International Symposium on Distributed Objects and Applications*, pages 650–660. Springer Verlag, 2002.
- [9] B. Long and P. Strooper. A case study in testing distributed systems. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, pages 20–29. IEEE Computer Society, 2001.
- [10] M.R. Lyu. Guest editor's introduction to the special issue on web technologies. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):505–508, 1991.

- [11] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 3rd edition, 2001.
- [12] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report 94-29, Sun Microsystems Laboratories, Inc., California, November 1994.