**ADF Performance Tuning: Tips on How to Limit ADF BC Memory Consumption**

[1](https://technology.amis.nl/2014/02/18/adf-performance-tuning-tips-on-how-to-limit-adf-bc-memory-consumption/#comments)-Djbo.debugoutput=console

BY [FRANK HOUWELING](https://technology.amis.nl/author/frank-houweling/) ON FEBRUARY 18, 2014[ORACLE ADF](https://technology.amis.nl/category/frontend-technology/oracle-adf/), [ORACLE WEBLOGIC SERVER](https://technology.amis.nl/category/platform-technology/oracle-weblogic-server/), [SOFTWARE ENGINEERING](https://technology.amis.nl/category/software-development/software-engineering/)

* 2
* 0
* -Djbo.debugoutput=console

This blog contains tips and tricks on to how limit the JVM memory consumption of ADF applications. Like other web applications, Oracle ADF applications potentially use a lot of JVM memory. Many times, the root cause of a high memory usage is that application data retrieved from the database into memory is not properly bounded; too many rows are fetched and held in ADF BC memory. This can lead to memory over-consumption, very long running JVM garbage collections, a freeze of all current requests or even OutOfMemoryErrors. To make matters worse, these rows and their attributes are frequently passivated and activated for an unnecessary long period of time. The solution to this problem can be found in reducing the size of sessions by decreasing of the amount of data loaded and held in the session. With a low memory consumption, a more responsive, stable and scalable ADF application can be delivered.

**Long JVM garbage collections**

A ‘happy JVM’ is important. If garbage collections are running very long (for example more than 30 seconds), this is often an indication of a problem – such as a freeze of all current requests because the JVM cannot clear enough memory. In the image below an example is shown of very long running garbage collections (red line). The heap space (blue) over time evolves more into a horizontal line than the saw-tooth shaped line a healthy JVM would be characterized by. In this case an OutOfMemoryError occurred and the server needed to be restarted. This should be a trigger to investigate whether the JVM heap space is set too low or that the ADF application consumes too much memory.

[](https://technology.amis.nl/wp-content/uploads/2014/02/long_running_jvm_garbage_collection.png)

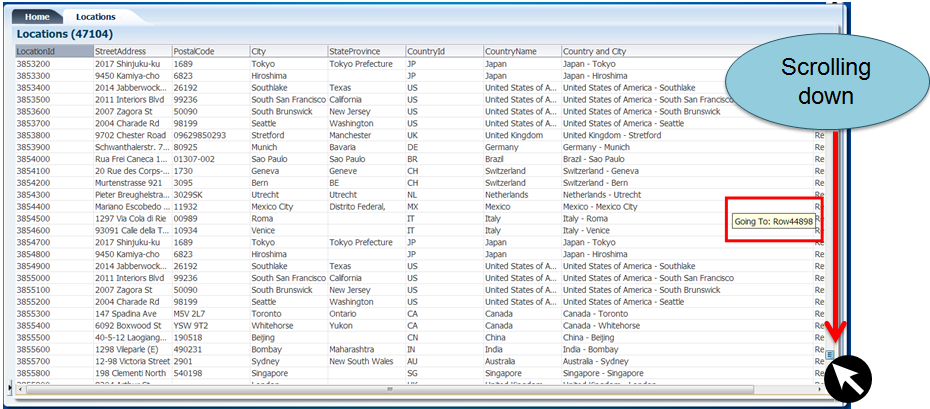
**Root cause of memory overload**

When the JVM heap space is set at a reasonable value, the root cause of memory over-consumption is often that the ADF application fetches far too many rows (and too many attributes – I will address this subject in another blog).

This can happen when the ViewObject SQL query potentially has access to thousands or even hundreds of thousands of database rows and it does not have (or not enough) restricting bind variables (or when no ViewCriteria is applied).

Further (**one or more**):

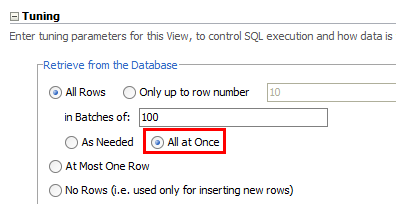
* An *af:table*, *af:treeTable*, *af:tree* component is used in a page and the underlying ViewObject’s **Access Mode** is **Scrollable**(this is the default value at the ViewObject tuning section)

[](https://technology.amis.nl/wp-content/uploads/2014/01/scrolling_down_locations.png)

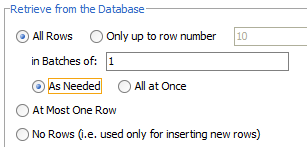
* The iterators **RangeSize** property (in the PageDefinition file) is set to **-1**

[](https://technology.amis.nl/wp-content/uploads/2014/01/rangesize_minus_one.png)

* The ViewObject fetchSize is set to **All Rows – (All at Once)**

[](https://technology.amis.nl/wp-content/uploads/2014/01/fetchsize_all_at_once1.png)

* A ViewObject is only used for inserting new records (and its fetchsize is still set at the default value **All Rows – (As Needed)** – it can still execute its SQL query (and unintentionally fetch all database rows that are accessible)

[](https://technology.amis.nl/wp-content/uploads/2014/01/fetchsize_default_as_needed.png)

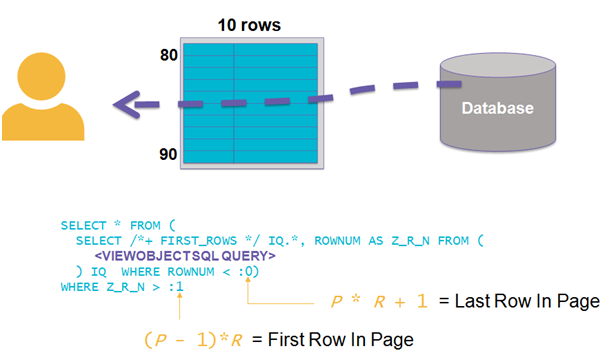
* Iterating in Java through all ViewObject rows

**Limiting rows fetched in ADF BC memory**

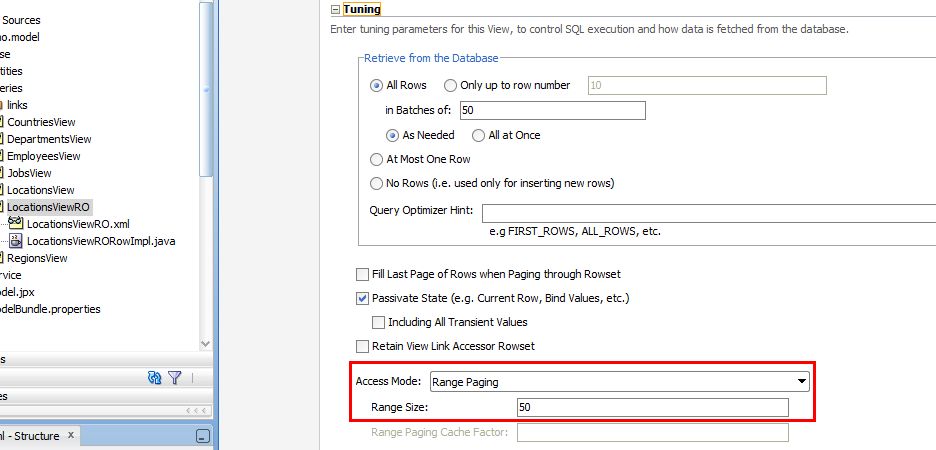
**Range paging**

ADF ViewObjects provide a mechanism to page through large data sets so that a user can navigate to a specific page in the results. Range Paging fetches and caches only the current page of rows in the ViewObject row cache (at the cost of another query execution) to retrieve each page of data. Range paging is not beneficial for small data sets where it is better to have all fetched rows in the ViewObject row cache. With Range paging you can say: “I would like to see page 9 of the search results (and 10 per page)”:

* R = Range size (rows per page)
* P = Page number of the results

[](https://technology.amis.nl/wp-content/uploads/2014/01/see_page_9_range_paging.png)

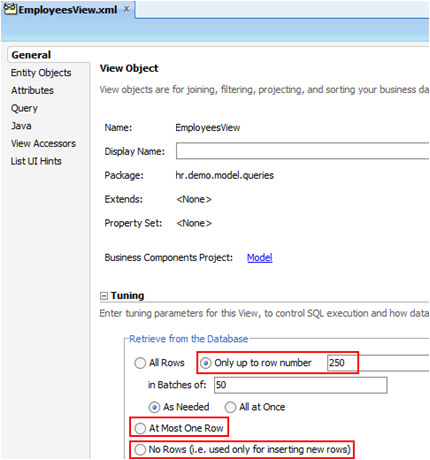
You can set the ViewObject **Access Mode** to **Range paging** in the ViewObject tuning section:

[](https://technology.amis.nl/wp-content/uploads/2014/01/vo_tuning_section_range_paging.png)

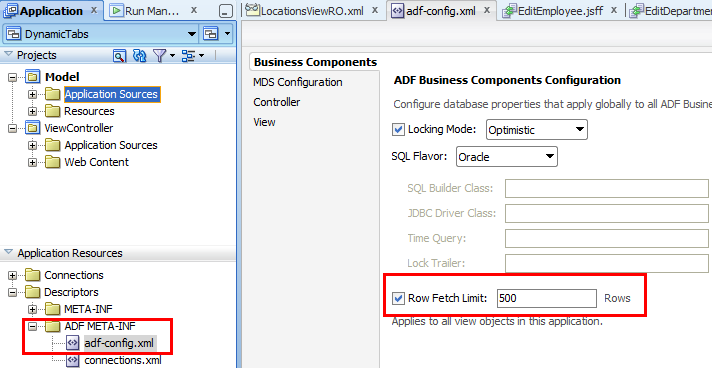
**Maximum fetchsize on ViewObject**

It is very important to limit the impact of non-selective queries that may return thousands of rows is setting a maximum fetchsize on the ViewObject instance. Recommend values are:

* Table-layout:    **Only up to row number**:  Set this to ± 250 rows and set a proper fetchSize (or use Range Paging)
* Form-layout:    **At Most One Row**(for dedicated ‘single row’ ViewObjects, when the ViewObject is being used only in the context of a single row (like the form part of a table/form layout). When a table row of a table/form layout is selected or clicked on, the form layout ViewObject is queried  with its ID value as bind parameter (for example with the *executeWithParams* operation). For table/form layout it is a good practice to create two separate fysical ViewObjects for table and form, especially when the table can fetch more than ± 250 rows database rows.
* Insert-layout:  **No Rows** (for Insert-only View Objects, when you know the Viewobject is being used only in insert mode)

[](https://technology.amis.nl/wp-content/uploads/2014/01/max_fetchsize.png)

An excellent way to prevent loading too much (database) rows is to set a global maximum fetchsize. You can set a global **Row Fetch Limit** in **ADF META-INF/adf-config.xml**:

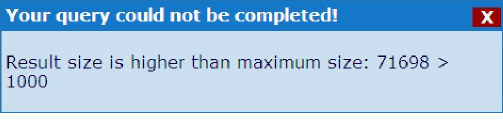
[](https://technology.amis.nl/wp-content/uploads/2014/01/global_max_fetchsize.png)

If you do want to set a higher or smaller maximum fetchsize for individual ViewObject instances, you can still set a different value at the tuning section of an instance and override the global threshold. It is recommended to think carefully for each ViewObject what the best value is.

**Extra bind variables on ViewObject**

If your ViewObject fetches/loads a large dataset, you can limit it with extra bind parameters (or extra ViewCriteria). For example, on a search screen end-users must fill in extra mandatory search fields that correspond to extra bind parameters, to make the database query resultset smaller.

Alternatively, before you execute the ViewObject query of an import (search) screen that has access to thousands of rows you can first execute a select COUNT that gets the number of rows returned by the query.

[](https://technology.amis.nl/wp-content/uploads/2014/02/found_too_many_rows_message.png)

If the number of rows exceeds a certain threshold (for example 250 rows), a message can be shown to the end user that too many results are found and that more refined, specific search terms or date restrictions are required. Only if the number of found results does not exceed the threshold, the query is executed, the database rows are fetched/loaded and the user gets the results.

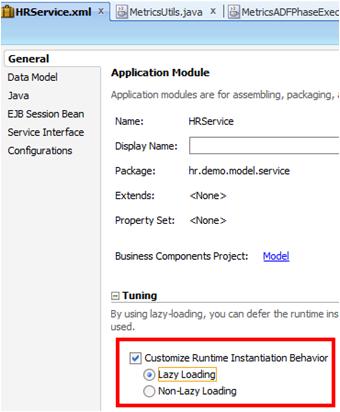
**Read-only ViewObjects**

If a ViewObject does not have to insert or update data, use a more memory friendly read-only ViewObject;

* Non updatable EO based ViewObject (deselect updatable in the ViewObject editor)
* Expert-mode read-only ViewObject

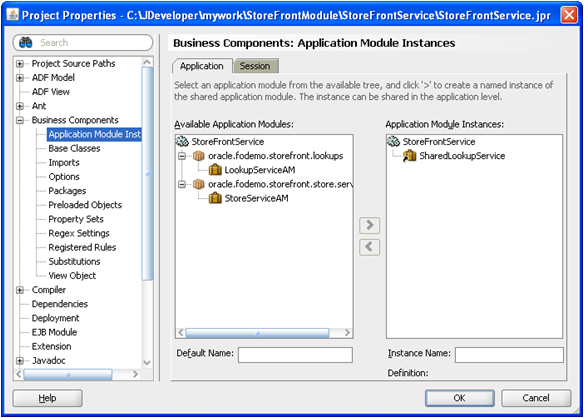
**ApplicationModule lazy loading**

It is recommended to defer the runtime instantiation of ViewObject and nested ApplicationModule instances until the time they are used. You can set this in the ApplicationModule tuning section:

[](https://technology.amis.nl/wp-content/uploads/2014/01/AM_lazy_loading.png)

**Shared ApplicationModule**

Use a shared ApplicationModule to group view instances when you want to reuse lists of static data across the application.

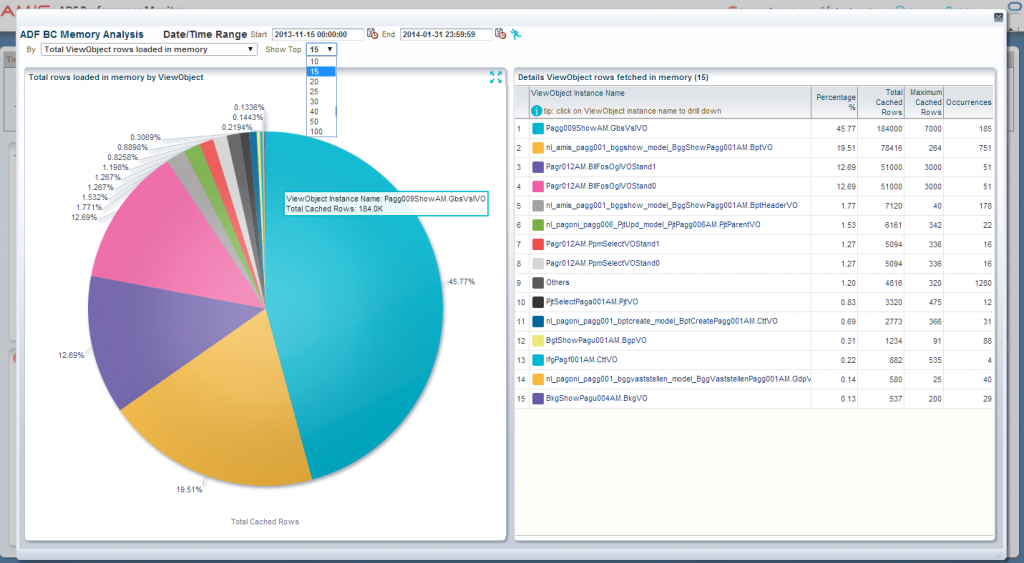
[](https://technology.amis.nl/wp-content/uploads/2014/02/shared_AM.png)

**ApplicationModule Pooling**

When a ViewObject is properly tuned and limited in its fetching, only the rows really needed are being kept in memory. This is very important for performance and to keep the memory low. This will also make ApplicationModule pooling activations and passivations much faster. ApplicationModule pooling parameter settings are also very important for performance, I will address this in another blog.

**Detecting an overload of ViewObject rows**

One of the new features of the [ADF Performance Monitor](http://www.adfpm.com/) is a kind of [ADF BC memory recorder and analyzer](https://www.adfpm.com/adf-performance-monitor-detecting-and-analyzing-a-high-adf-bc-memory-consumption/). It detects and warns when too many ViewObject (database) rows are being fetched/loaded into memory. Also, for a selected time range (month, week, day, e.g.), the tool gives aggregated overviews for each ViewObject instance how many rows have been fetched/loaded into ADF BC memory (ViewObject row cache or EntityCache). ViewObjects that frequently fetch hundreds or thousands of rows – and waste memory – can be detected. Also individual high fetches/loads can be analysed.

[](https://technology.amis.nl/wp-content/uploads/2014/02/ADFBC_memory_scan_total1.png)

With this information, ViewObject instances that frequently fetch too many database rows can be addressed by the suggestions written in this blog; adding extra bind parameters, setting a (better) maximum fetchsize or using Range Paging.