How to Work with ABAP Shared Memory Objects

Applies to:

SAP Netweaver 7 onwards. For more information, visit the ABAP homepage.

Summary

This article gives a detailed introduction to creating and using Shared Memory Objects in your ABAP programs. The concept of shared memory objects is useful if your programs rely on heavy access to large quantities of relatively static data, and can greatly improve the performance of your applications.

Author: Trond Strømme

Company: Statoil

Created on: 07 February 2011

Author Bio

Trond Strømme has been working as a freelance ABAP consultant for more than 15 years, for a range of international companies. He has extensive experience within most areas of ABAP development. Currently, he is working as a senior developer for Statoil in his native Norway.

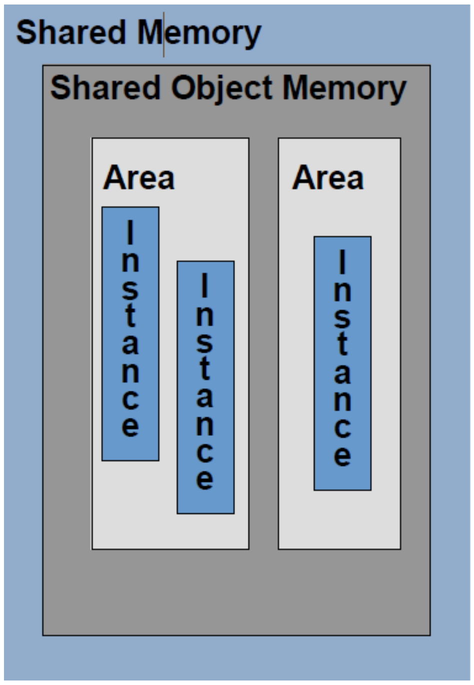
Table of Contents

What are Shared Memory Objects?

The Shared Memory Objects concept is available from release 6.40 onwards. It consists of storing data in SAP memory (much like the EXPORT/IMPORT statements before ECC6), for rapid data retrieval without the need for physically reading persistent DB tables. Unlike the “old” EXPORT/IMPORT technique, however, the Shared Memory Object concept also allows for business logic to be stored (for instance, sorting or calculations based on the stored data).

In addition, it is, as the name implies, shared between user sessions, so that the data can be freely used and updated by any number of users or sessions. This makes shared memory objects more flexible than the session memory technique.

Basics of Shared Memory Objects



- Shared memory-enabled classes (root classes) are defined in SE24, using the “shared memory-enabled” addition
- Each shared memory area relates to one such specific global root class
- Shared memory areas can have several area instances
- Each area instance can have several versions (if versioning is allowed)
- An area class is generated by defining an area in SHMA – this is used as the area handle (attach methods)
- The root class is used for manipulating the data of the shared object
- Shared memory objects can be accessed by any number of users/sessions
- Write/update locks are exclusive
- All locks are for complete area instance and version (no partial locks possible on specific attributes of the object)

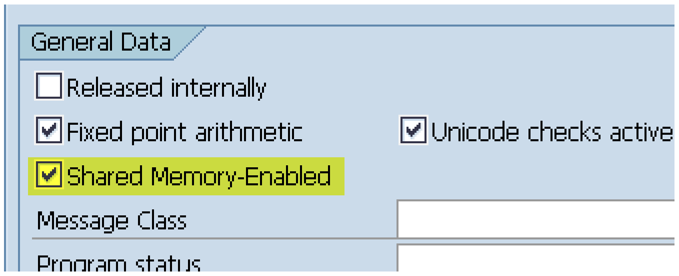
The Simplified Example

We will use a simplified example for this test case: a program that repeatedly reads material master records (view MARAV) for reporting purposes. In order to reduce DB access, we decide to make use of the Shared Memory Objects concept to store this data in memory, instead of executing repeated select statements on the above view.

Note: shared memory objects are primarily used for “static” data – data that doesn’t change often. Material master records are usually “stable”, in the sense that this information is not too frequently updated. Other data types, such as sales documents, are not good candidates for shared memory objects as these are frequently created/modified and not at all “static”.

The Two Steps to Define a Shared Memory Object

There are two main steps to creating a shared memory object. Usually, we start by defining the object itself – this means creating an ordinary class using transaction SE24 (or SE80). We define attributes – the data going to be held by the class – and the methods required to enable our ABAP programs to access and work with these attributes. This class is also known as the “root class”, since it is used in step 2 to define some vital properties of the memory area. Most importantly, we define the class as being “shared memory enabled”:


General Data

[ ] Released internally

[✓] Fixed point arithmetic [✓] Unicode checks active

[✓] Shared Memory-Enabled

Message Class

Program status

The second step is to allocate a new memory area for our shared object. This is done via transaction SHMA, and here, we will use our newly created root class as the “global area root class” of our new memory area. This second step gives us the “area handle” – a specific class which is needed to interact with the memory object we created in the first step.

The next sections will provide a detailed description on how to do this, based on a real world example.

Step 1: The Root Class

The Purpose of the Root Class

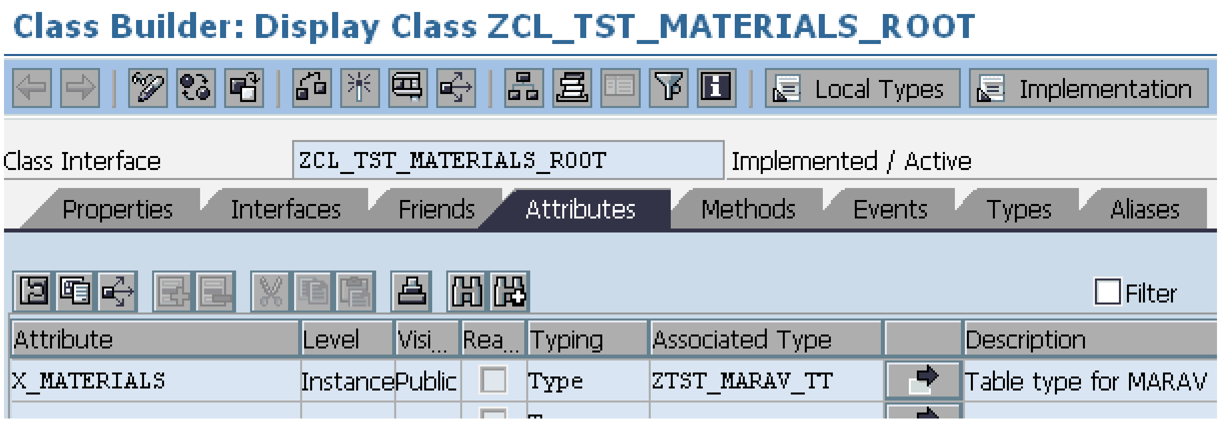
As already mentioned, the concept of shared memory objects is based on memory areas, each related to a global area root class. The root class contains the attributes and methods you will use in your ABAP applications. It is created as any other class, the only difference being it having the “shared memory-enabled” attribute set – thus preparing it for being used as a shared memory object.

Basically, the root class will need attributes to hold the data you want to store (in our example, an internal table of type MARAV), as well as methods for manipulating the data. In our example, this will include a method to initialize the table with specific data from the persistent layer, as well as retrieving data from it.

The root class can also have specific attributes not related to the persistent layer, just like ordinary classes. It can even contain attributes referring to other shared memory objects, although this is outside the scope of this tutorial.

Defining Attributes of the Root Class

First of all, we will need to define the data to be stored in our memory object. We want material data, based on view MARAV (material master and related texts), so we simply define a table attribute to hold this data:


Class Builder: Display Class ZCL_TST_MATERIALS_ROOT

Class Interface ZCL_TST_MATERIALS_ROOT Implemented / Active

Attribute	Level	Visi...	Rea...	Typing	Associated Type		Description
X_MATERIALS	Instance	Public	☐	Type	ZTST_MARAV_TT		Table type for MARAV

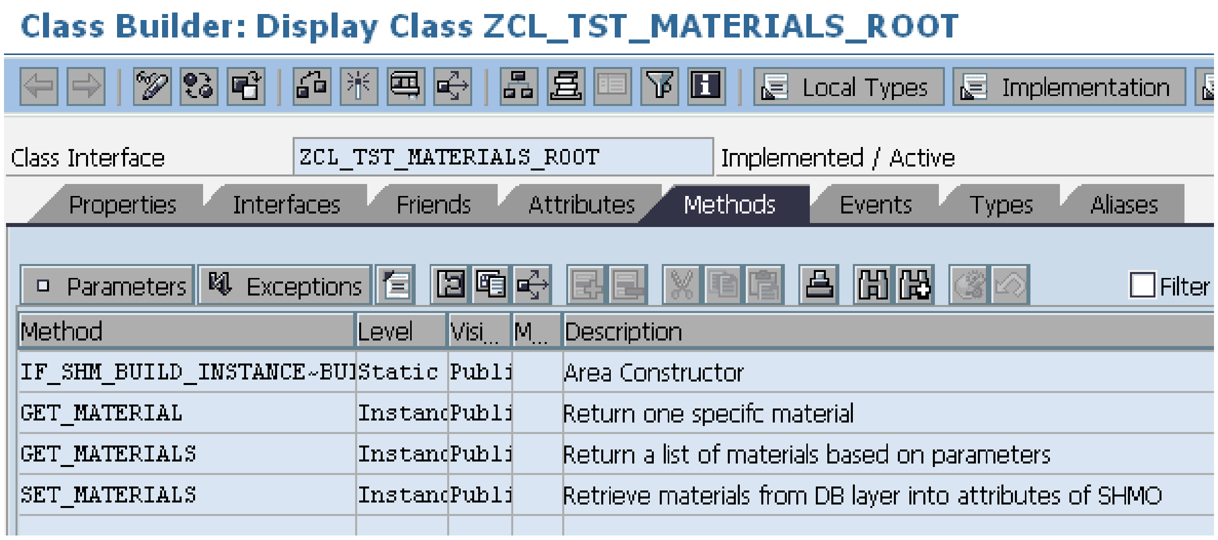
(Note that I’ve cheated by already defining a suitable table type for view MARAV…!)

Defining Methods for the Root Class

Now, we need methods to allow for interacting with the data. We need at least two methods in our root class, one for initializing the X_MATERIALS attribute with data from the underlying DB tables (the persistent layer), and another method that can be used by our ABAP applications to retrieve data from the memory object.

In addition, I have created a third method, allowing us to retrieve data for one specific material (based on material number as input parameter).

The fourth method visible below, IF_SHM_BUILD_INSTANCE~BUILD, relates to what is known as “automatic pre-load” of the object, and will be discussed later.



Class Interface ZCL_TST_MATERIALS_ROOT Implemented / Active

Method	Level	Visi...	M...	Description
IF_SHM_BUILD_INSTANCE~BUI	Static	Publi		Area Constructor
GET_MATERIAL	Instanc	Publi		Return one specifc material
GET_MATERIALS	Instanc	Publi		Return a list of materials based on parameters
SET_MATERIALS	Instanc	Publi		Retrieve materials from DB layer into attributes of SHMO

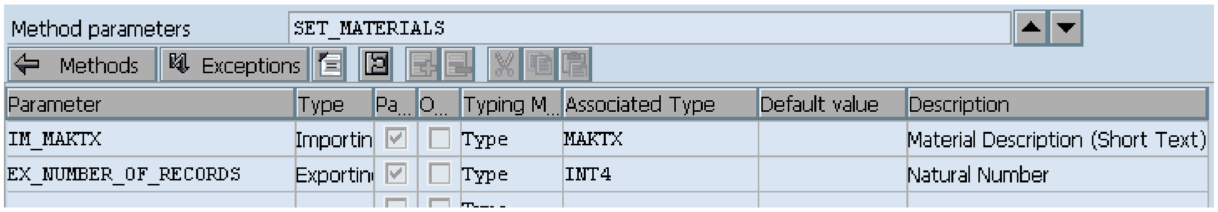
Method SET_MATERIALS

The SET_MATERIALS method is used for retrieving data from the DB layer. When our shared memory object is created for the first time, it will be empty, meaning we need to invoke the SET_MATERIALS method to populate the X_MATERIALS attribute.

In order to avoid reading the complete material master into our object (something that could violate our allocated memory space and (even worse) make the basis team somewhat annoyed), we will limit ourselves to materials in a specific name range – based on importing parameter IM_MAKTX:

```
method set_materials.

  translate im_maktx using '*%'.

  select * from marav
    appending corresponding fields of table x_materials
    where spras = sy-langu
    and maktx like im_maktx.

  describe table x_materials lines ex_number_of_records.
endmethod.
```

The method has the following parameters:



Method parameters SET_MATERIALS

Parameter	Type	Pa...	O...	Typing M...	Associated Type	Default value	Description
IM_MAKTX	Importin	✓		Type	MAKTX		Material Description (Short Text)
EX_NUMBER_OF_RECORDS	Exportin	✓		Type	INT4		Natural Number

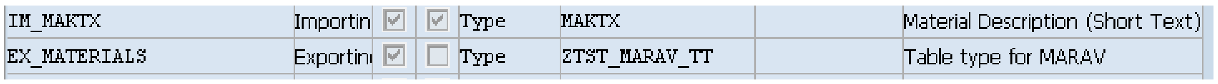
- IM_MAKTX limits the materials selected
- EX_NUMBER_OF_RECORDS returns the number of materials retrieved.

Method GET_MATERIALS

The GET_MATERIALS method is used to retrieve data from the material attribute table of the object. This is the method we will normally call from our ABAP applications – instead of selecting records from the DB layer.

```
method get_materials.

  field-symbols: <fs_materials> type marav,
                 <fs_re_materials> type marav.

  loop at x_materials
    assigning <fs_materials>
    where maktx cp im_maktx.
    append initial line to ex_materials assigning <fs_re_materials>.
    move-corresponding <fs_materials> to <fs_re_materials>.
  endloop.

endmethod.
```

The parameters are defined as follows:



IM_MAKTX	Importin	✓	✓	Type	MAKTX		Material Description (Short Text)
EX_MATERIALS	Exportin	✓	☐	Type	ZTST_MARAV_TT		Table type for MARAV

- IM_MAKTX allows for searching materials based on their name
- EX_MATERIALS returns the list of retrieved materials in table form.

Method GET_MATERIAL

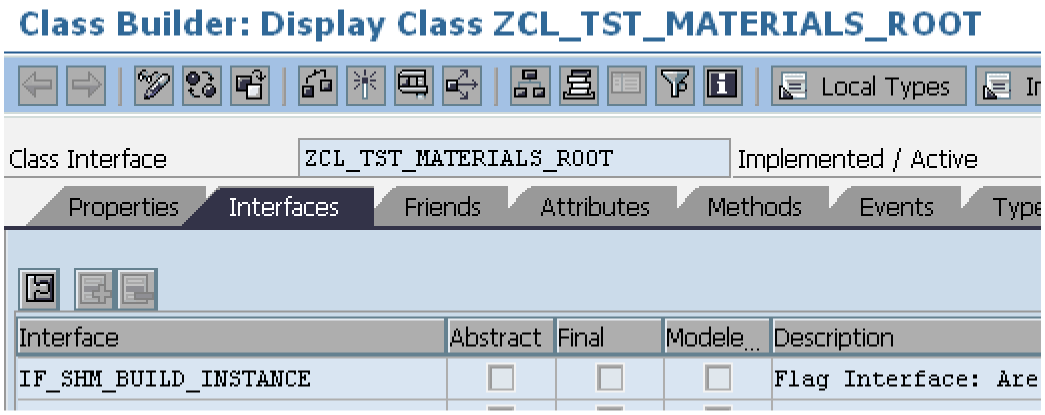
We also have a specific method (GET_MATERIAL), used to retrieve one single material based on the material number. The code should be pretty straight-forward – I'll leave it to you to create as an exercise ☺

Method IF_SHM_BUILD_INSTANCE~BUILD

Finally, we need to look at the method IF_SHM_BUILD_INSTANCE~BUILD. This method is related to what is known as automatic pre-load (or auto-start) of the memory area related to our memory object. Basically, when a memory area is defined, we can explicitly set its properties so that it will be automatically initialized whenever an application tries to access it. Alternatively, if we do not define the memory area in this way, any access to it without the area having been initialized would trigger an exception of type CX_SHM_NO_ACTIVE_VERSION. Whatever we choose, we have to keep in mind that without automatic pre-load, our ABAP applications will have to check for existence of the memory area and initialize it manually if required.

The automatic pre-load option removes this hassle by ensuring that any memory areas that are not yet initialized will be automatically started whenever an application tries to access the data. This means that if your ABAP program tries to read materials from the shared memory object, and the memory area hasn't been initialized, it will be done "on the fly" – removing the need for handling the above exception in your applications.

Automatic pre-loading requires us to set a few parameters when defining the memory area (more on this in the next session – creating the memory area). It also means we have to add the interface IF_SHM_BUILD_INSTANCE to our root class:


Class Builder: Display Class ZCL_TST_MATERIALS_ROOT

Class Interface ZCL_TST_MATERIALS_ROOT Implemented / Active

Interface	Abstract	Final	Modele...	Description
IF_SHM_BUILD_INSTANCE	☐	☐	☐	Flag Interface: Are

As a result, the method IF_SHM_BUILD_INSTANCE~BUILD is automatically added to the list of available methods – as already seen.

The method will be empty, however, and in order for pre-loading to work, we need to fill it with some ABAP code!

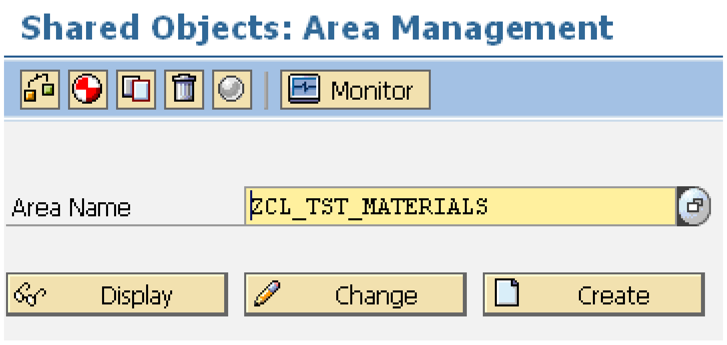
```
method if_shm_build_instance~build.

* Autoload of SHMO (first time use, after system restart and so on)

  data: lcl_materials type ref to zcl_tst_materials,
        lcl_materials_root type ref to zcl_tst_materials_root,
        lx_exception type ref to cx_root.

  try.
      lcl_materials = zcl_tst_materials=>attach_for_write( ).
    catch cx_shm_error into lx_exception.
      raise exception type cx_shm_build_failed
      exporting previous = lx_exception.
  endtry.

  try.
      create object lcl_materials_root area handle lcl_materials.
      lcl_materials->set_root( lcl_materials_root ).
      lcl_materials_root->set_materials( im_maktx = 'A*' ).
      lcl_materials->detach_commit( ).
    catch cx_shm_error into lx_exception.
      raise exception type cx_shm_build_failed
      exporting previous = lx_exception.
  endtry.

  if invocation_mode = cl_shm_area=>invocation_mode_auto_build.
    call function 'DB_COMMIT'.
  endif.

endmethod.
```

The code above initializes the memory object with some material data. This is done by invoking the attach_for_write method of the memory area, followed by the instantiation of the root class. When this is done, the set_materials method of the memory object is called in order to fill the X_MATERIALS table with some initial data (in our example, we limit ourselves to materials whose names start with A*).

If the above code doesn't immediately make sense, don't worry! We will examine the specific methods above in greater detail in section 3 – how to use memory objects in your ABAP programs.

Step 2: Creating the Memory Area

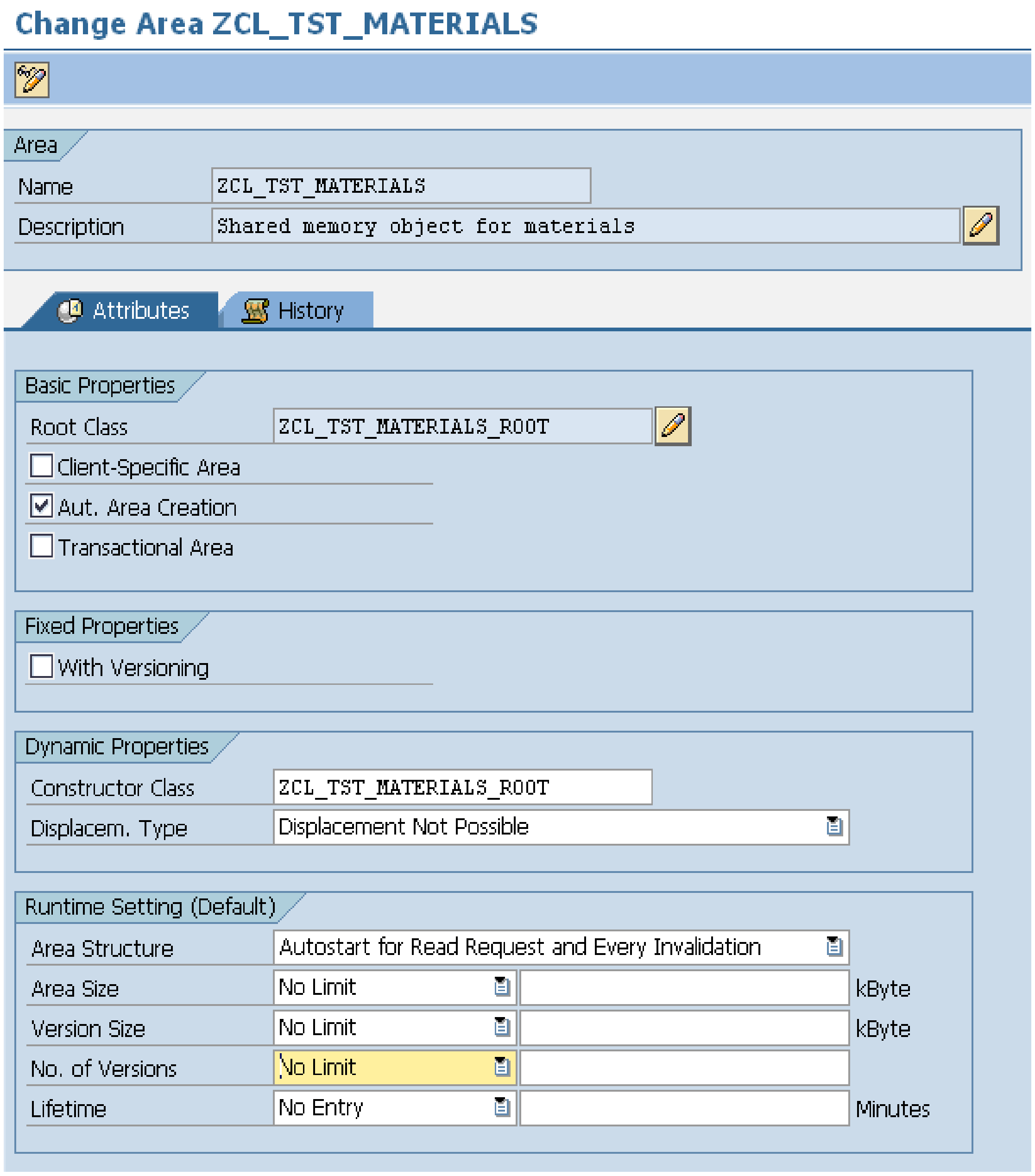
In order to create a shared memory area for our newly created memory object, we execute transaction SHMA and provide a name of the shared memory area.



Shared Objects: Area Management

Area Name	ZCL_TST_MATERIALS

After clicking Create, some properties of the object have to be defined:



Change Area ZCL_TST_MATERIALS

Area

Name	ZCL_TST_MATERIALS
Description	Shared memory object for materials

Basic Properties

Root Class	ZCL_TST_MATERIALS_ROOT

- [ ] Client-Specific Area
- [✓] Aut. Area Creation
- [ ] Transactional Area

Fixed Properties

- [ ] With Versioning

Dynamic Properties

Constructor Class	ZCL_TST_MATERIALS_ROOT
Displacem. Type	Displacement Not Possible

Runtime Setting (Default)

Area Structure	Autostart for Read Request and Every Invalidation		
Area Size	No Limit		kByte
Version Size	No Limit		kByte
No. of Versions	No Limit		
Lifetime	No Entry		Minutes

Here, we relate our new memory area to the memory object we just created – the root class.

Note that we also define the root class also as constructor class for the memory area. This is strictly speaking not mandatory – we could have used another class as our constructor or omitted it altogether. However, since we have decided to use the automatic pre-load option, we have to specify a constructor class, which must implement the IF_SHM_BUILD_INSTANCE interface. For the sake of simplicity, we are using our root class also as the constructor class for our area.

Further, and also related to the automatic pre-load option, we need to check “Automatic area creation” – this ensures the memory object is automatically created whenever it is invoked by a running program – if it does not already exist. As already discussed, the alternative is to manually ensure there is an active memory object whenever you need it – risking an exception of type CX_SHM_NO_ACTIVE_VERSION if this is not the case.

Note: such manual memory area initialization can be done using the BUILD method of the area class – please refer to SAP documentation on how to do this.

Finally, we have selected “Autostart for read request and every invalidation” – again related to the the automatic pre-load feature of our area.

The remaining properties allow for sizing the memory object, as well as setting max number of versions and lifetime (expiration). For now, we will leave these (but feel free to experiment). Notably, the lifetime property can be handy if your object relates to specific activities (such as year-end tasks). Please refer to SAP documentation for a thorough discussion on versioning and other advanced concepts.

This is it – we now have a working shared memory object, along with it’s related memory area. We are finally ready to start using the advantages of SMO’s in our ABAP applications!

Step 3: Using Shared Memory Objects in your ABAP Programs

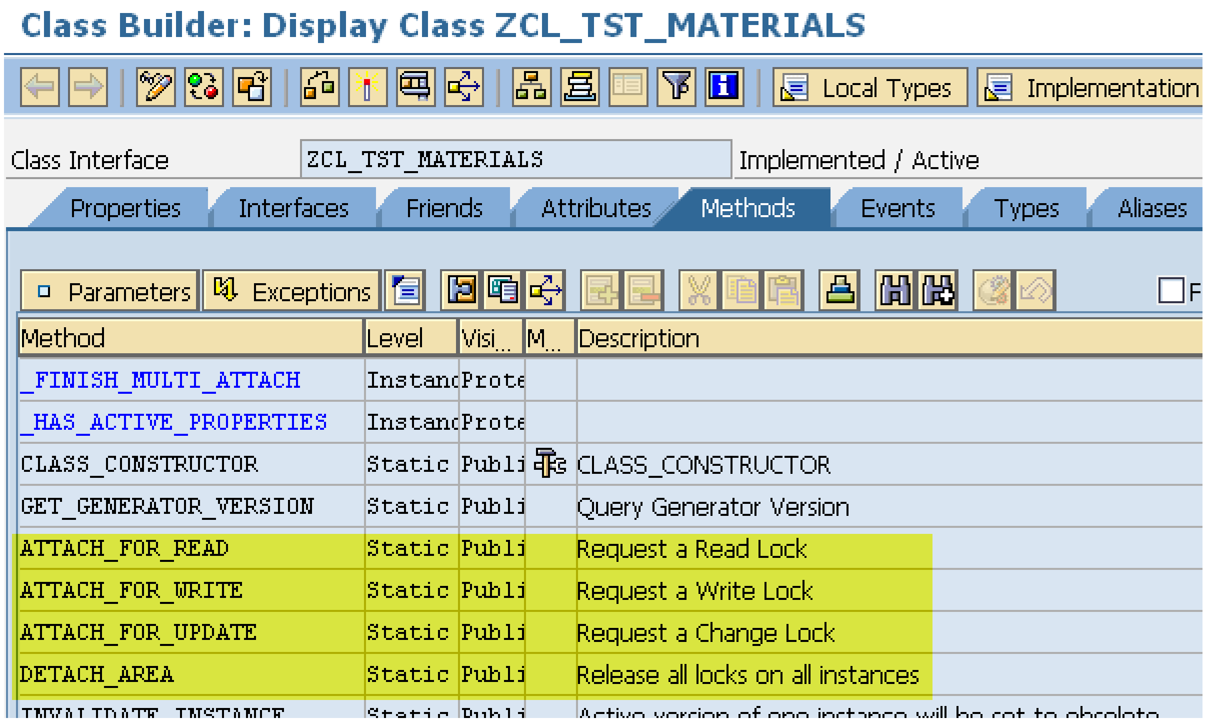
Using the memory object in an ABAP program normally follows the following four steps:

- Create a reference to the shared memory area class
- Call one of the ATTACH methods of the area class to get an area handle for the shared memory area
- Call any of the data manipulation methods of the area's root class
- Call a DETACH method to "free" the area handle

When writing an ABAP program to read data from the memory object, we need to define references to both the memory area and the memory object (root class). This is because we will use the ATTACH methods of the memory area to establish a handle for the area, followed by the specific methods of the memory object in order to make use of the attributes.

Defining and using an Area Handle

When we defined our memory area using transaction SHMA, a related class was automatically generated. This class provides the methods for establishing a handle for the memory area. We can examine the class using transaction SE24 (it has the same name as the memory area):


Class Builder: Display Class ZCL_TST_MATERIALS

Class Interface ZCL_TST_MATERIALS Implemented / Active

Method	Level	Visi...	M...	Description
_FINISH_MULTI_ATTACH	Instanc	Prote		
_HAS_ACTIVE_PROPERTIES	Instanc	Prote		
CLASS_CONSTRUCTOR	Static	Publi		CLASS_CONSTRUCTOR
GET_GENERATOR_VERSION	Static	Publi		Query Generator Version
ATTACH_FOR_READ	Static	Publi		Request a Read Lock
ATTACH_FOR_WRITE	Static	Publi		Request a Write Lock
ATTACH_FOR_UPDATE	Static	Publi		Request a Change Lock
DETACH_AREA	Static	Publi		Release all locks on all instances
INVALIDATE_INSTANCE	Static	Publi		Active version of one instance will be set to obsolete

The ATTACH methods are, as indicated, used for requesting locks on the memory area. This is a prerequisite for accessing the related memory object.

Most of the time, we will use attach_for_read and possibly attach_for_update methods. The attach_for_write method is usually only used during the creation of a memory object – as seen in the code for method IF_SHM_BUILD_INSTANCE~BUILD in the previous section.

Schematically, the use of a shared memory object looks like this:

```
* Define the area and memory object references
data: lcl_materials_area type ref to zcl_tst_materials,
      lcl_materials_root type ref to zcl_tst_materials_root,

* Call the area method attach_for_read to get an area handle
  try.
       lcl_materials_shmo = zcl_tst_materials=>attach_for_read( ).
        catch cx_shm_no_active_version
              cx_shm_read_lock_active
              cx_shm_change_lock_active
              cx_shm_exclusive_lock_active
              cx_shm_inconsistent
        into lv_exception.
  endtry.

* If successful, we invoke the GET method of the root class:
  if lcl_materials_shmo is initial.
    call method lcl_materials_shmo->root->get_materials(
         exporting
           im_maktx = p_maktx
         importing
      ex_materials = lt_materials ).
* Finally, detach the area after use
    lcl_materials_shmo->detach( ).
  endif.
```

As we can see, the ATTACH/DETACH methods of the area class are invoked to establish the "handle" for the memory area. Provided this succeeds, we then invoke the specific methods from the memory object itself (as defined in the root class) in order to manipulate the data.

The sequence for updating the attributes of the memory object is not too different; it basically requires the use of the ATTACH_FOR_UPDATE method to establish a change lock – as opposed to a write lock above. Please refer to the SAP documentation for more info on the various types of memory area locking.

Sample Program for using a Memory Object

Below is the complete code of the sample program for using the memory object for materials master. The program reads and updates the memory object, depending on the parameter checked on the selection screen.

When reading, the attach_for_read method of the memory object is called, followed by the get_materials method of the root class.

When updating, the attach_for_update method of the memory object is called, followed by the set_materials method of the root class.

Finally, the program contains code to delete the memory object (method free_instance of the memory object).

Note: you will note that the below program uses the ATTACH_FOR_READ method twice within the same try-endtry construct. This is due to the automatic pre-loading feature: the first call to attach the object will fail if the area has not yet been initialized. This is why we wait 10 seconds, then give it a second go. This wait will provide the area with a chance to initialize itself and hopefully ensure the second call to establish the handle is successful!

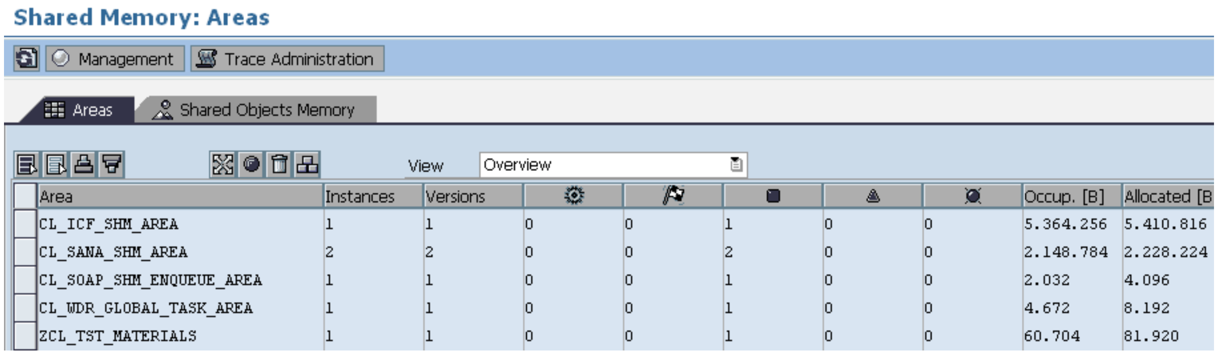
```
*&---------------------------------------------------------------------*
*& Report  ZTST_USE_MATERIALS_SHMO
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*
report  ztst_use_materials_shmo.
data: lcl_materials_shmo type ref to zcl_tst_materials,
      lcl_materials_root type ref to zcl_tst_materials_root,      lt_materials type z
tst_marav_tt,
      lv_area_name type shm_inst_name value '$DEFAULT_INSTANCE$',
      lv_rc type sy-subrc,
      lv_exception type ref to cx_root,
      lv_lines type i,
      lv_text type string.

field-symbols: <fs_material> type marav.

parameters: p_del radiobutton group r1,
            p_upd radiobutton group r1,
            p_get radiobutton group r1 default 'X',
            p_matnr like marav-matnr,
            p_maktx like marav-maktx.
start-of-selection.

  case 'X'.
    when p_del.
      try.
          call method zcl_tst_materials=>free_instance
            exporting
              inst_name         = lv_area_name
*          terminate_changer = ABAP_TRUE
            receiving
              rc                = lv_rc.
        catch cx_shm_parameter_error
          into lv_exception.
      endtry.

      if not lv_exception is initial.
        call method lv_exception->if_message~get_text
          receiving
            result = lv_text.
        write lv_text.
      else.
        write : / 'The memory object has been deleted'.
      endif.

    when p_get.

      try.
          lcl_materials_shmo = zcl_tst_materials=>attach_for_read( ).
        catch cx_shm_no_active_version
              cx_shm_read_lock_active
              cx_shm_change_lock_active
              cx_shm_exclusive_lock_active
```

```
                cx_shm_inconsistent
            into lv_exception.
            wait up to 10 seconds. "And give it a second go...
            try.
                lcl_materials_shmo = zcl_tst_materials=>attach_for_read( ).
              catch cx_shm_no_active_version
                    cx_shm_read_lock_active
                    cx_shm_change_lock_active
                    cx_shm_exclusive_lock_active
                    cx_shm_inconsistent
                          into lv_exception.

            endtry.
        endtry.

        if not lv_exception is initial.
          call method lv_exception->if_message~get_text
            receiving
              result = lv_text.
          write lv_text.
        endif.

        if lcl_materials_shmo is initial.

          write : / 'The materials SHMO does not exist - create it first'.

  * If materials exist, get data
        else.

          call method lcl_materials_shmo->root->get_materials(
            exporting
            im_matnr = p_matnr
            im_maktx = p_maktx
            importing ex_materials = lt_materials ).

          lcl_materials_shmo->detach( ).

  * If no materials data found, means materials
  * for this date do not exist yet.
          if lt_materials is initial.
            write : / 'No materials data for these parameters - please update first'.
          else.
            describe table lt_materials lines lv_lines.
            write : / 'Number of lines retrieved: ', lv_lines.
            uline.
            loop at lt_materials assigning <fs_material>.
              write : / <fs_material>-matnr, <fs_material>-maktx.
            endloop.
          endif.

        endif.

      when p_upd.

        if p_matnr is initial and p_maktx is initial.
          message 'Fill in some of the selection criteria' type 'E'.
```

```
    else.

      try.
          lcl_materials_shmo = zcl_tst_materials=>attach_for_update( ).

          call method lcl_materials_shmo->root->set_materials(
              exporting
              im_maktx = p_maktx
              importing ex_number_of_records = lv_lines ).
          lcl_materials_shmo->detach_commit( ).

          commit work.
          if sy-subrc eq 0.
            write : / 'Number of lines added to SHMO: ', lv_lines.
          else.
            write : / 'No lines added'.
          endif.

        catch cx_shm_no_active_version
              cx_shm_read_lock_active
              cx_shm_change_lock_active
              cx_shm_exclusive_lock_active
              cx_shm_pending_lock_removed
              cx_shm_version_limit_exceeded
              cx_shm_inconsistent
          into lv_exception.

      endtry.

      if not lv_exception is initial.
        call method lv_exception->if_message~get_text
          receiving
            result = lv_text.
        write lv_text.
      endif.

    endif.

endcase.
```

Step 4: Examining the Shared Memory Object

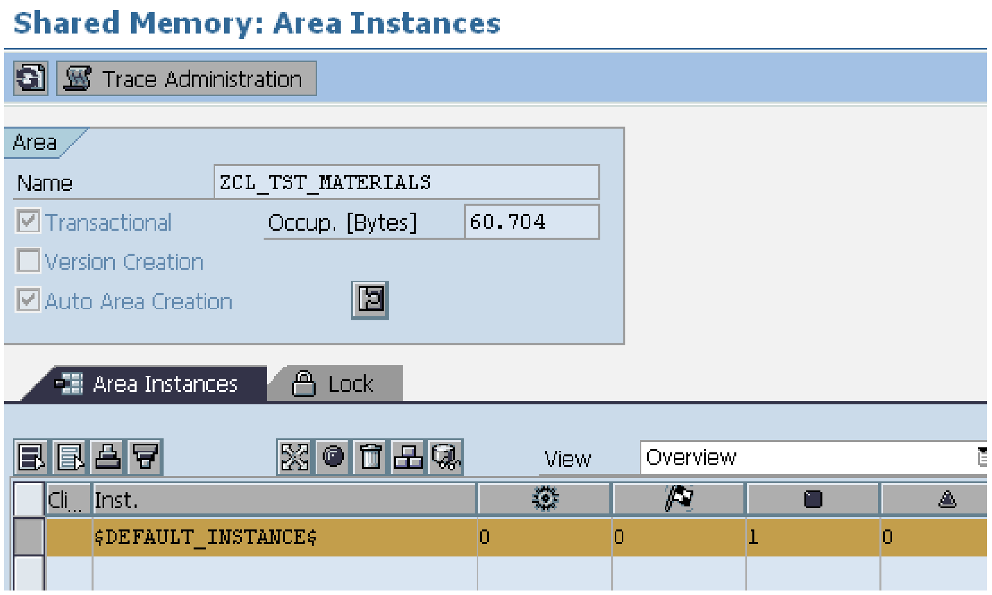
After the shared memory object has been created, we can use transaction SHMM to examine it:



Area	Instances	Versions	⚙	⚑	●	▲	✱	Occup. [B]	Allocated [B
CL_ICF_SHM_AREA	1	1	0	0	1	0	0	5.364.256	5.410.816
CL_SANA_SHM_AREA	2	2	0	0	2	0	0	2.148.784	2.228.224
CL_SOAP_SHM_ENQUEUE_AREA	1	1	0	0	1	0	0	2.032	4.096
CL_WDR_GLOBAL_TASK_AREA	1	1	0	0	1	0	0	4.672	8.192
ZCL_TST_MATERIALS	1	1	0	0	1	0	0	60.704	81.920

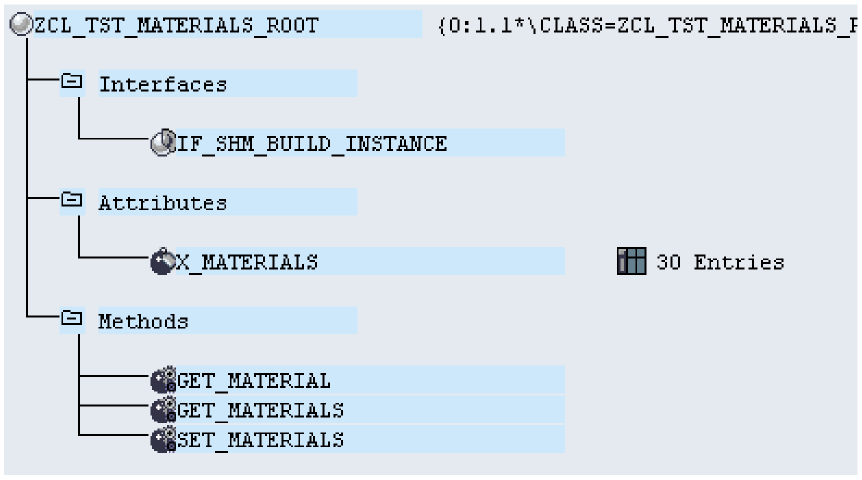
The shared memory object now "lives" in SAP memory, and will remain there until the SAP instance goes down. It can however easily be deleted by selecting it and using the delete button. Note that if the root class is changed in any way (new logic, new methods, adding new parameters or even changing the type or visibility of existing ones etc.), any existing memory objects must be deleted and a new object created – either using the creation program or the mentioned autostart functionality. Failure to do so will result in inconsistencies and lead to dumps when using the shared memory object!

Checking attributes (data) of the object by drilling down (SHMM):

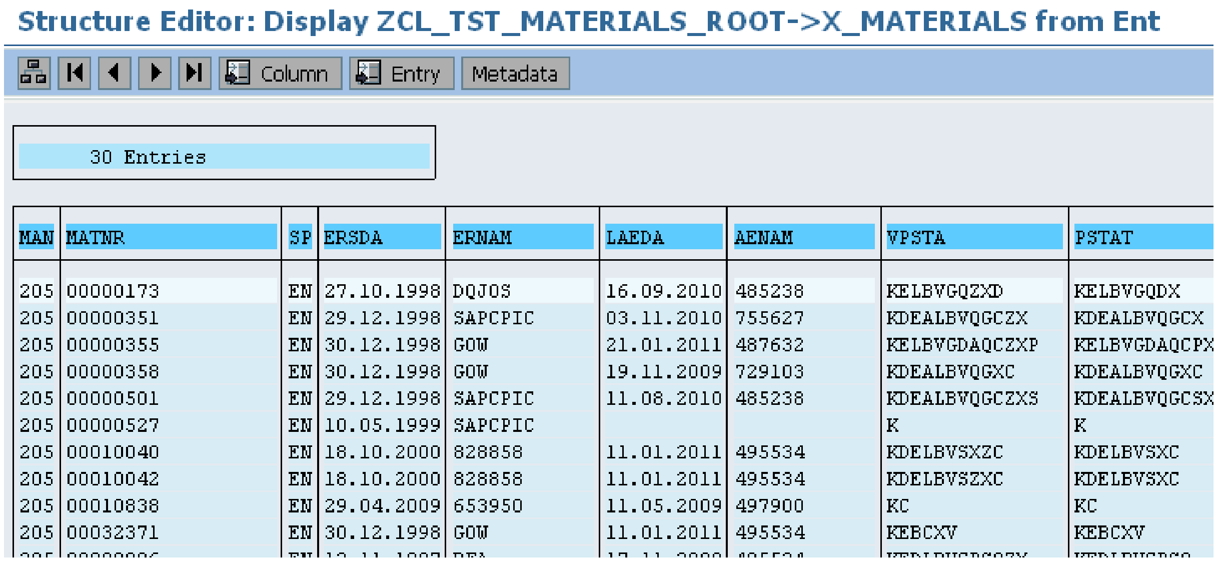


Name	ZCL_TST_MATERIALS		
✓ Transactional		Occup. [Bytes]	60.704
☐ Version Creation			
✓ Auto Area Creation			

Cli...	Inst.	⚙	⚑	●	▲
	$DEFAULT_INSTANCE$	0	0	1	0



```
ZCL_TST_MATERIALS_ROOT          {O:1.1*\CLASS=ZCL_TST_MATERIALS_F
├─ Interfaces
│   └─ IF_SHM_BUILD_INSTANCE
├─ Attributes
│   └─ X_MATERIALS                       30 Entries
└─ Methods
    ├─ GET_MATERIAL
    ├─ GET_MATERIALS
    └─ SET_MATERIALS
```


Structure Editor: Display ZCL_TST_MATERIALS_ROOT->X_MATERIALS from Ent

Column | Entry | Metadata

30 Entries

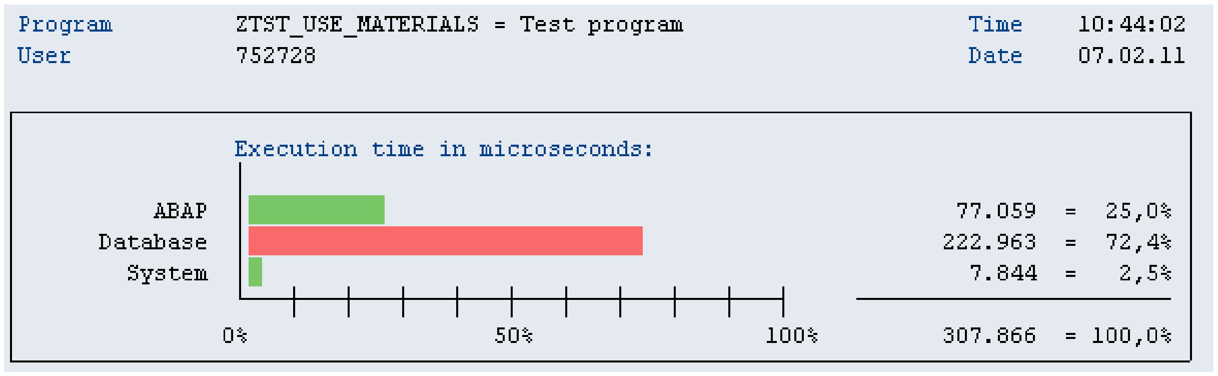
MAN	MATNR	SP	ERSDA	ERNAM	LAEDA	AENAM	VPSTA	PSTAT
205	00000173	EN	27.10.1998	DQJOS	16.09.2010	485238	KELBVGQZXD	KELBVGQDX
205	00000351	EN	29.12.1998	SAPCPIC	03.11.2010	755627	KDEALBVQGCZX	KDEALBVQGCX
205	00000355	EN	30.12.1998	GOW	21.01.2011	487632	KELBVGDAQCZXP	KELBVGDAQCPX
205	00000358	EN	30.12.1998	GOW	19.11.2009	729103	KDEALBVQGXC	KDEALBVQGXC
205	00000501	EN	29.12.1998	SAPCPIC	11.08.2010	485238	KDEALBVQGCZXS	KDEALBVQGCSX
205	00000527	EN	10.05.1999	SAPCPIC			K	K
205	00010040	EN	18.10.2000	828858	11.01.2011	495534	KDELBVSXZC	KDELBVSXC
205	00010042	EN	18.10.2000	828858	11.01.2011	495534	KDELBVSZXC	KDELBVSXC
205	00010838	EN	29.04.2009	653950	11.05.2009	497900	KC	KC
205	00032371	EN	30.12.1998	GOW	11.01.2011	495534	KEBCXV	KEBCXV

Conclusion: Comparing the use of a Shared Memory Object to Classical ABAP with DB read

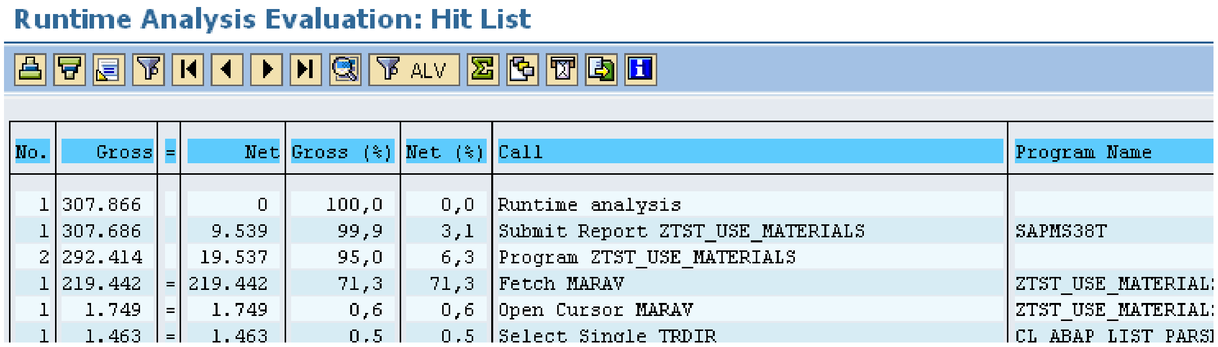
As expected, the DB load is completely removed when using shared memory objects. We can examine this by checking the run time analysis.

Result of Runtime Analysis:

For the classical ABAP ZTST_USE_MATERIALS, (whose logic resembles the select statement in method SET_ORDERS_LIST of the shared memory area root class), the analysis looks like this:

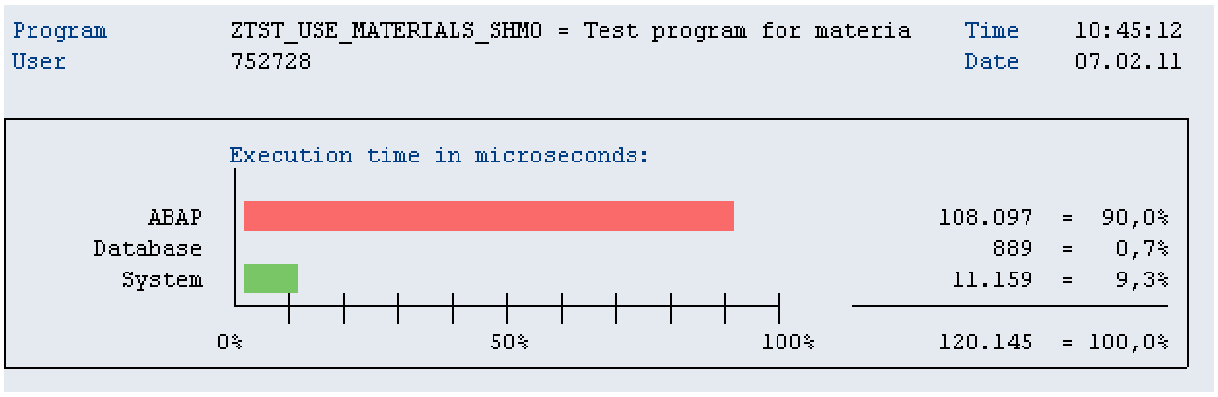

```
Program        ZTST_USE_MATERIALS = Test program            Time    10:44:02
User           752728                                       Date    07.02.11

               Execution time in microseconds:

     ABAP                                              77.059  =  25,0%
 Database                                             222.963  =  72,4%
   System                                               7.844  =   2,5%

           0%              50%              100%      307.866  = 100,0%
```

Some details:


Runtime Analysis Evaluation: Hit List

No.	Gross	=	Net	Gross (%)	Net (%)	Call	Program Name
1	307.866		0	100,0	0,0	Runtime analysis	
1	307.686		9.539	99,9	3,1	Submit Report ZTST_USE_MATERIALS	SAPMS38T
2	292.414		19.537	95,0	6,3	Program ZTST_USE_MATERIALS	
1	219.442	=	219.442	71,3	71,3	Fetch MARAV	ZTST_USE_MATERIAL
1	1.749	=	1.749	0,6	0,6	Open Cursor MARAV	ZTST_USE_MATERIAL
1	1.463	=	1.463	0,5	0,5	Select Single TRDIR	CL_ABAP_LIST_PARSI

For the shared memory objects program ZTST_ORDERS_SHMO_READ, the runtime analysis looks like this:



```
Program          ZTST_USE_MATERIALS_SHMO = Test program for materia    Time     10:45:12
User             752728                                                Date     07.02.11

                  Execution time in microseconds:

      ABAP                                                   108.097  =   90,0%
  Database                                                       889  =    0,7%
    System                                                    11.159  =    9,3%

          0%                  50%                 100%       120.145  = 100,0%
```

The ABAP portion of the program “eats” a bit more resources than the classical ABAP. However, the database access is zero (as expected), whereas the system load is somewhat higher than the classic ABAP program. Overall resource use, however, is considerably less using the memory object.

Of course, the initial resources needed to load the shared memory object have to be counted, but this is supposedly done only once (or very rarely), not every time the program runs.

Related Content

Shared Objects - SAP help

Shared Objects - Implementation

ABAP shared objects made easy

For more information, visit the ABAP homepage

Disclaimer and Liability Notice