IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 19, Issue 5, Ver. III (Sep.- Oct. 2017), PP 38-46
www.iosrjournals.org

Hiperspace: The Forgotten Giant of Mainframe Storage

*Antariksha Gupta
(B.Tech, West Bengal University of Technology, India)
Corresponding Author: Antariksha Gupta

Abstract: *Hiperspace, a framework on IBM Mainframes, enables application programs to store data up to very large capacities. Normally, a COBOL program can have a working storage section storing up to a few Megabytes of data. Hiperspace allows the temporary storage to expand up to Terabytes. While doing so, it does not burden the application program with the I/O overheads of the large amount of data. It provides for an efficient I/O operation handling. In spite of its usefulness, Hiperspace is a gray area. Very few are aware about it and even fewer actually know how to work with it. This paper tries to demystify the Hiperspace from an application programmer's perspective.*

Keywords: *CSRIDAC, CSRSCOT, CSRVIEW, Hiperspace, Window Services*

Date of Submission: 23-09-2017 Date of acceptance: 06-10-2017

I. Introduction

Hiperspace stands for High Performance Space. It is a specific area in expanded memory which can be used to temporarily store large volumes of data. This data can be fetched later on, during the execution of the program, as and when needed. It is a part of the Window Services available on IBM Mainframes.

This paper covers the life-cycle and functional details of the Hiperspace. We will take a quick look at the Window Services as Hiperspace comes under its purview. Then we will move on to the various phases and sub-phases in the life-cycle of the Hiperspace – both from a functional standpoint and a programming perspective. Working with Hiperspace involves the use of a few specific subroutines called Window Service programs. These Window Service programs have to be called using their respective set of parameters/arguments. The working details of the Window Service programs have also been included to help the application programmers gain a thorough understanding of Hiperspace.

COBOL has been used to demonstrate the programming details because of two reasons – it is still a widely used language in mainframe programming and its English like nature makes it easy to understand for non-programmers too. However, Hiperspace can be used with other high level languages such as ADA, C/C++, FORTRAN, Pascal and PL/1 as well.

II. Need for Hiperspace

Since ages COBOL programmers have been told that there is a limit to working storage section and it should be used wisely. Indeed there is a limit - Enterprise COBOL supported 16,777,215[1] bytes. However, Enterprise COBOL version 3.4 onwards, the limit now is 134,217,727[2] bytes. At first glance this might seem enough but is it really enough? Let us consider a use case:

In Payments Industry, there can be huge corporate purchase transactions with each transaction spanning over 10 Million - 1 Billion records (lines of data). The records can be anywhere between 20 - 100 bytes in length. And, the transaction cannot be processed until data from all the records have been accumulated. So this eventually becomes a problem of temporarily storing 200 Million bytes. But, COBOL cannot store more than 134 Million bytes. Most programmers would suggest the use of VSAM files here - store the transaction records as key-value pairs. The key can be a number which identifies a transaction uniquely plus a sequence number to maintain unique key constraint. The value portion will contain the actual records from the file.

But, there can be serious questions regarding the efficiency of this approach. This approach needs an extra intermediate step to convert the input file to key-value pairs in VSAM. The mainframe OS has to do additional housekeeping in order to store the data in VSAM as the keys (and the associated data) must always be in sorted order. In addition, the entire key portion is composed of duplicate/redundant data; and if we have a file with millions of lines, the wastage of space can be alarming.

This example is very specific to Payments Industry but there can be similar issues in other industries too. And, often they are dealt with in an inefficient manner which increases the processing time as well as required disk storage space while putting more loads on the OS at the same time. The working storage space isn't enough for such situations. The most suitable solution for such situations is the Hiperspace.

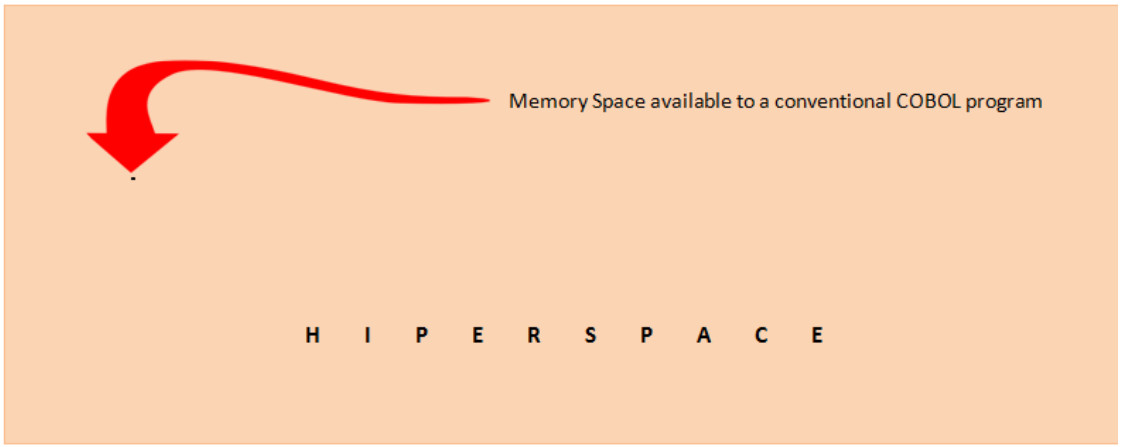

Figure 1: Comparison of COBOL Working Storage area against Hiperspace

Fig. 1 gives a visual comparison between the memory spaces available to a conventional COBOL program and the Hiperspace. The miniscule black dot at the tip of the arrow is the maximum temporary storage available i.e. 134,217,727 bytes. And, the big buff colored rectangle is the Hiperspace – almost 16 TB ! Using Hiperspace in situations stated above can keep the architecture simple.

III. A Brief Study of Window Services

Hiperspace comes under the umbrella of Window Services on IBM Mainframes. And, Window Services is a framework which can interact directly with data objects residing on DASD or the expanded storage area[3]. Window Services can be invoked from high level language application programs to store or retrieve large chunks of data from the DASD and expanded memory without utilizing the application program's I/O mechanism. All Window Services use a variable which, by convention, is called the "Window". It is this "Window" that renders Window Services its characteristic name. All interactions between the application program and the DASD/expanded memory happen through this Window.

The functionalities of Window services are best observed from the perspective of data objects they work upon. There are two types of data objects –

i. Temporary Data Objects - A temporary data object is an area of expanded storage that the application program can use to hold temporary data, such as a temporary buffer for a file. It exists until it is explicitly terminated or the application program finishes.
ii. Permanent Data Object - A permanent data object is a virtual storage access method (VSAM) linear data set that resides on DASD. You can create a new VSAM, read data from an existing VSAM and also update its contents using window services. Because this type of data object can be saved on DASD, it is called a permanent object.

The expanded storage that Window Services use for a temporary object or for the scroll area is called a Hiperspace. It is a range of contiguous virtual storage addresses that a program can indirectly access through a variable in the program's virtual storage. This paper covers interactions with only the temporary objects.

IV. Life Cycle of Hiperspace

There are 4 phases when working with Hiperspace – Set-Up Phase, Storage / Push Phase, Retrieval / Fetch Phase and Termination Phase. It is the application programmer's responsibility to ensure that these phases occur in the correct sequence. Fig. 2 shows the phases and sub-phases in the life cycle of the Hiperspace.

i. Set-Up Phase – Just like any other variable the Hiperspace needs to be defined before it can be used. This is done by allocating a memory space on the expanded memory. Its size, properties, usage and identifying criteria have to be specified as per one's requirement.
   This phase also involves creating a "Window" to act as a bridge between the application program and the Hiperspace. The Window should be compatible with the Hiperspace in terms of size and the data type that needs to be handled.

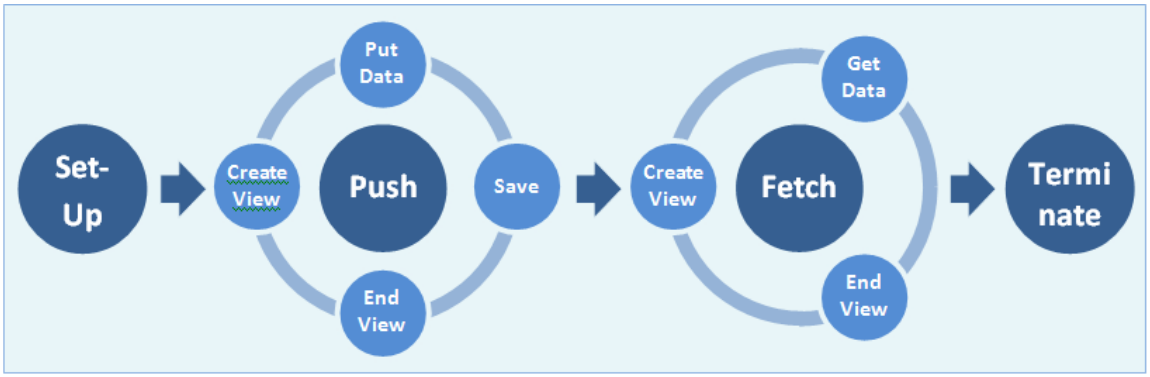

Figure 2: Life-Cycle of Hiperspace

i. Storage Phase Or Push Phase – Once the Hiperspace has been set-up, the data can be pushed into the Hiperspace. Actually, the data is put into the Window and then a Window Service program is invoked with appropriate options to identify the Hiperspace and the exact location within the Hiperspace and other relevant information.
The Window Service program interprets the options and accordingly creates a view at the appropriate location in the Hiperspace. Then data movement is done and the changes saved. Finally, the view is ended.
This phase can occur multiple times either successively or interleaved with the Fetch phase.

ii. Retrieval Phase Or Fetch Phase – The data present in the Hiperspace is retrieved in this phase. The application program invokes a Window Service program with appropriate options which in turn loads the Window with the requested data from the Hiperspace.
This phase is similar to the Push phase in terms of the Window Service programs required with the exception that there is no need to save the data in the Hiperspace.

iii. Termination Phase – After successful use of Hiperspace, it needs to be terminated so that the expanded memory space reserved for Hiperspace gets released.

V. Working Details of Window Service Programs

Each phase of the Hiperspace framework has an associated Window Service program. These programs behave similar to functions and have specific signatures (i.e. argument list and returned values). The application programmer has to just 'CALL' the appropriate program with proper parameters to get the job done. This keeps the complexities of Hiperspace hidden to the application programmer. Fig. 3 shows Window Service Programs associated with different phases in life-cycle of the Hiperspace.

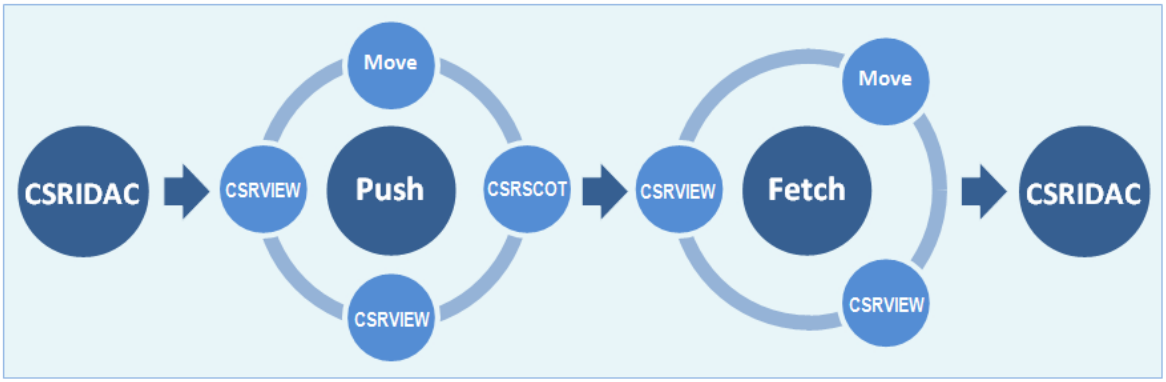

Figure 3: Life-Cycle of Hiperspace – In terms of Window Service Programs

i. Parameter Details of CSRIDAC – The Window Service program associated with the Set-Up and Termination phases is CSRIDAC. The parameter list for CSRIDAC is given in Table 1. The 'operation_type' parameter is used to distinguish between set-up and termination. Other parameters define the properties of the Hiperspace.

Parameter	Size	Values
operation_type	PIC X(05)	• BEGIN - To request access to an object • END - To terminate access to an object
object_type	PIC X(09)	• TEMPSPACE - The object is a temporary data object • DDNAME - DDNAME of an existing VSAM linear data set • DSNAME - The file name of a new/existing linear VSAM dataset
object_name	PIC X(44)	• Not required for TEMPSPACE. For permanent objects : • If object_type is DDNAME, it must contain the DD name • If object_type is DSNAME, it must contain the dataset name
scroll_area	PIC X(03)	• YES - Create a scroll area • NO - Do not create a scroll area
object_state	PIC X(03)	• OLD - The object exists • NEW - The object doesn't exist and Window Services must create it
access_mode	PIC X(03)	• READ - Read access. • UPDATE - Update access.
object_size	PIC 9(04)	Specifies the maximum size of the new object in units of 4 KB
object_id	PIC X(08)	• If operation_type is BEGIN, the object identifier is returned here • If operation_type is END, supply the object identifier in here
high_offset	PIC 9(04)	When CSRIDAC completes, high_offset contains the size of the existing object expressed in blocks of 4096 bytes
return_code	PIC 9(04)	When CSRIDAC completes, return_code contains the return code
reason_code	PIC 9(04)	When CSRIDAC completes, reason_code contains the reason code.

Table 1: Parameter Details of CSRIDAC

ii. Parameter Details of CSRVIEW – There are two Window Service programs in the Push and Fetch phases. CSRVIEW is used to create and end the view. Data movement happens through the normal application program constructs. So, for COBOL programs the MOVE verb is used to put data into the Window or to get data from the Window. Once, the data movement is done, CSRSCOT is called to save the data in the Hiperspace. The parameter list for CSRVIEW is given in Table 2. Again, the operation_type is used to distinguish between view creation and view termination. Other parameters specify the Window and the data movement options.

Parameter	Size	Values
operation_type	PIC X(05)	• BEGIN - To request access to an object • END - To terminate access to an object
object_id	PIC X(08)	Supply the object identifier that CSRIDAC returned when you obtained access to the object.
offset	PIC 9(04)	Specify the offset of the view into the object in blocks of 4096 bytes
span	PIC 9(04)	Specify the Window size in blocks of 4096 bytes
window_name	PIC X(08)	Specify the symbolic name assigned to the Window in your address space
usage	PIC X(06)	• **SEQ** - The reference pattern will be sequential. Window Services will brings up to 16 blocks of data into the Window at a time, depending on the window-size and availability of resources. • **RANDOM** - The reference pattern is expected to be random. If you specify RANDOM, Window Services brings data into the Window one block at a time.
disposition	PIC X(07)	• **When CSRVIEW operation_type is BEGIN** **REPLACE** - The first time you reference a block to which the Window is mapped, CSRVIEW replaces the data in the Window with the data from the referenced block. **RETAIN** - When you reference a block to which the Window is mapped, the data in the Window remains unchanged. When you call CSRSAVE to save the mapped blocks, CSRSAVE saves all of the mapped blocks because CSRSAVE considers them changed. • **When CSRVIEW operation_type is END** **REPLACE** - CSRVIEW discards the data that is in the Window, making the Window contents unpredictable. CSRVIEW does not update mapped blocks of the object or scroll area. **RETAIN** - CSRVIEW retains the data that's in the Window and updates the mapped blocks of the object or scroll area.
return_code	PIC 9(04)	When CSRIDAC completes, return_code contains the return code
reason_code	PIC 9(04)	When CSRIDAC completes, reason_code contains the reason code.

Table 2: Parameter Details of CSRVIEW

iii. Parameter Details of CSRSCOT – The parameter list for CSRSCOT is given in Table 3. It just takes the location of the view that has to be saved to Hiperspace.

Parameter	Size	Values
object_id	PIC X(08)	Supply the object identifier that CSRIDAC returned when you obtained access to the object.
offset	PIC 9(04)	Specify the offset of the view into the object in blocks of 4096 bytes
span	PIC 9(04)	Specify the window-size in blocks of 4096 bytes
return_code	PIC 9(04)	When CSRSCOT completes, return_code contains the return code
reason_code	PIC 9(04)	When CSRSCOT completes, reason_code contains the reason code.

Table 3: Parameter Details of CSRSCOT

VI. Life Cycle of Hiperspace - Revisited

This section will revisit the Hiperspace framework from an application programmer's perspective. The variable names used in the code snippets below are standard names used as per convention, but any other name can be used as per application programmer's choice. But, the respective program signature must be strictly adhered to.

i. Set-Up Phase – This phase involves setting up the Window and setting up the Hiperspace.

i.1. Set-Up Window

The Window is a variable in the application program with a few constraints – its size must be a multiple of 4096 bytes and it should be aligned on 4096 byte boundary. The actual size of the Window should be dictated by the business requirements.

Defining a variable of n*4096 bytes is an easy task but aligning it to 4096 byte boundary can appear to be a bit daunting but it can be done using pointers as shown below in Fig. 4.

```
WORKING-STORAGE SECTION.
  01   P                        POINTER.
  01   PR REDEFINES P           PIC 9(9) COMP.
  01   DUMMY                    PIC 9(9) COMP.
  01   R                        PIC 9(9) COMP.
  01   INP-DATA.
       02 DATA-REC              PIC X(40) OCCURS 4096 TIMES.
  . . .      . . .                 . . .

  LINKAGE SECTION.
  01   INWORK                   PIC X(167935).
  01   WINDOW.
       02 FILLER                PIC X(4096) OCCURS 40 TIMES.

  PROCEDURE DIVISION USING INWORK.
      SET P TO ADDRESS OF INWORK
      DIVIDE PR BY 4096   GIVING DUMMY REMAINDER R
      IF R NOT = 0
         COMPUTE PR = PR + 4096 - R
      END-IF
      SET ADDRESS OF WINDOW TO P
```

Figure 4: Setting up the Window

Here, WINDOW is 40 * 4096 bytes long. In Hiperspace terminology, 4096 bytes s considered a 'PAGE', a data-record is called an element and the Hiperspace is termed as the object. So, this WINDOW is 40 PAGES long. And, to align it to 4096 byte boundary we take another variable INWORK. INWORK is 40 * 4096 + 4095 bytes long. The extra 4095 bytes ensure that when we push the WINDOW to align it to 4096 byte boundary, it does not go outside the memory space.

i.2. Define Hiperspace

In order to define the Hiperspace, we call CSRIDAC with operation_type as "BEGIN". Since Hiperspace is a temporary data object, the object_type shall always be "TEMPSPACE". The object_name can have any name as per programmer's choice. At the time of creation, the object_state will be "NEW" while the access_mode will be "UPDATE". The Hiperspace should be large enough to hold the entire data to be stored temporarily. We start by calculating the number of elements Window can hold.

$$\text{Number of Records the Window can hold} = \frac{\text{No.of Pages in Window} \times \text{Size of a Page}}{\text{Size of a single element}}$$
$$= \frac{40 \text{ X } 4096}{40}$$
$$= 4096$$

So, we can accommodate 4096 elements (or data-records) in a single Window. If we know beforehand the maximum number of data-records that we can get, then we can easily calculate the size of the Hiperspace. For example if there can be a maximum of 50,000 records each of 40 byte length, then we can calculate the size of Hiperspace as

$$\text{Size of Hiperspace} = \frac{\text{Maximum Number of Data Records}}{\text{Number of Data Records the Window can hold}}$$
$$= \frac{50{,}000}{4096} = 12.21 \text{ Windows} = 488.4 \text{ Pages} \approx 489 \text{ Pages}$$

Though we can set the size of the Hiperspace to 489 Pages, it is advised to keep the Hiperspace size an integral multiple of the Window size to avoid any accidental data spillage. So in this case we should set the Hiperspace size to 13 Windows' worth i.e. 520 Pages.

If we are directly using hard-coded values as the input parameters, then we need to call 'BY CONTENT'. Instead of hard-coding these options, we can also create variables for these option. However, in that case, we would need to call CSRIDAC using those variables 'BY REFERENCE'. Fig. 5 shows a part of the Working Storage section along with the call to CSRIDAC.

```
 WORKING-STORAGE SECTION.
* PAGE-SIZE (IN BYTES)
 01   PGSIZE                    PIC 9(9) COMP VALUE 4096.
* ELEMENT-SIZE (IN BYTES)
 01   ELMSZE                    PIC 9(9) COMP VALUE 40.
* NO. OF PAGES IN THE WINDOW
 01   NWINPG                    PIC 9(9) COMP VALUE 40.
* NO. OF ELEMENTS IN THE WINDOW
 01   NWINEL                    PIC 9(9) COMP.
* NO. OF PAGES IN DATA OBJECT
 01   NOBJPG                    PIC 9(9) COMP.
* WINDOWS WILL BEGIN ORIGIN-ING AT OFFSET 0 IN DATA OBJECT
 01   WINOFF                    PIC 9(9) COMP VALUE 0.
 01   RETRN1                    PIC 9(9) COMP.
 01   REASON                    PIC 9(9) COMP.
 01   OBSIZ                     PIC 9(9) COMP.
 01   TOKEN                     PIC X(8).
 01   K                         PIC 9(9) COMP.

 PROCEDURE DIVISION.
* WINDOW COMPOSED OF 40-BYTE ELEMENTS
     COMPUTE NWINEL = PGSIZE * NWINPG / ELMSZE.

* HERE, HIPERSPACE SIZE = 13 WINDOWS WORTH
     COMPUTE NOBJPG = 13 * NWINPG

* SET UP ACCESS TO A HIPERSPACE OBJECT
     CALL "CSRIDAC" USING
       BY CONTENT
         "BEGIN",
         "TEMPSPACE",
         "MY FIRST HIPERSPACE",
         "YES",
         "NEW",
         "UPDATE",
       BY REFERENCE
          NOBJPG,
          TOKEN,
          OBSIZ,
          RETRN1,
          REASON
```

Figure 5: Setting up access to the Hiperspace

ii. Storage Phase Or Push Phase

Storage or the Push phase is relatively simple. The data is transferred to Hiperspace in a 4-step process. A view is created in the Hiperspace, then the data is moved into the Window using the standard constructs of the application program followed by saving the data in the view. Finally the view is ended.

The view has to be created in the Hiperspace. The view which identifies the exact location in the Hiperspace where the data is to be stored. It also determines the direction of data movement i.e., whether the data is to be moved from the Window to the Hiperspace or vice-versa. CSRVIEW is called with operation_type as "BEGIN". The object_id returned at the time of defining Hiperspace is passed along with the offset (from the beginning) in the Hiperspace, size of the Window and the Window itself. The usage is kept as "RANDOM" as it allows the program to access any location of the Hiperspace. The disposition decides how the data movement is to be done. It is kept as "RETAIN" to ensure that the Window contents do not change and at the time of saving and the Hiperspace gets updated with the Window contents.

After the data movement is done, the view has to be saved by calling CSRSCOT. It takes the object_id of the Hiperspace along with the location and size of the view. The location is specified in terms of offset and the size is same as the Window-size.

And, lastly the view is ended by again calling the CSRVIEW but this time with operation_type as "END" and disposition as "RETAIN".

```
* CREATE A VIEW IN THE HIPERSPACE
     CALL "CSRVIEW" USING
       BY CONTENT
          "BEGIN",
       BY REFERENCE
          TOKEN,
          WINOFF,
          NWINPG,
          WINDOW,
       BY CONTENT
          "RANDOM",
          "RETAIN",
       BY REFERENCE
          RETRN1,
          REASON
    MOVE INP-DATA   TO WINDOW
* CAPTURE THE VIEW IN THE WINDOW
     CALL "CSRSCOT" USING
          TOKEN,
          WINOFF,
          NWINPG,
          RETRN1,
          REASON
* END THE VIEW
     CALL "CSRVIEW" USING
       BY CONTENT
          "END ",
       BY REFERENCE
          TOKEN,
          WINOFF,
          NWINPG,
          WINDOW,
       BY CONTENT
          "RANDOM",
          "RETAIN ",
       BY REFERENCE
          RETRN1,
          REASON
```

Figure 6: Storage Phase or Push Phase

iii. Retrieval Phase Or Fetch Phase

The Fetch phase is similar to the Push phase with the exception that there is no save sub-phase. During the view creation step, the disposition is kept as "REPLACE". This ensures that the Window picks up the data from the Hiperspace instead of writing to it. At the time of ending the view, the disposition is kept as "RETAIN".

```
* CREATE A VIEW IN THE HIPERSPACE
     CALL "CSRVIEW" USING
       BY CONTENT
         "BEGIN",
       BY REFERENCE
          TOKEN,
          WINOFF,
          NWINPG,
          WINDOW,
       BY CONTENT
         "RANDOM",
         "REPLACE",
       BY REFERENCE
          RETRN1,
          REASON
     MOVE WINDOW TO INP-DATA
* END THE VIEW
     CALL "CSRVIEW" USING
       BY CONTENT
         "END ",
       BY REFERENCE
          TOKEN,
          WINOFF,
          NWINPG,
          WINDOW,
       BY CONTENT
         "RANDOM",
         "RETAIN ",
       BY REFERENCE
          RETRN1,
          REASON
```

Figure 7: Retrieval Phase or Fetch Phase

iv. Termination Phase

Termination is a one step process involving a call to CSRIDAC but with operation type as "END". All other options remain the same as those during the set-up phase.

```
* TERMINATE ACCESS TO THE HIPERSPACE OBJECT
     CALL "CSRIDAC" USING
       BY CONTENT
         "END   ",
         "TEMPSPACE",
         "MY FIRST HIPERSPACE ENDS HERE ",
         "YES",
         "NEW",
         "UPDATE",
       BY REFERENCE
          NOBJPG,
          TOKEN,
          OBSIZ,
          RETRN1,
          REASON
```

Figure 8: Termination Phase

VII. Hiperspace in a Nutshell

The Window and the Hiperspace reside on two different memory locations. But when CSRVIEW is called then it maps the Window to a specific location in the Hiperspace. This location is specified by the offset parameter. Once the view has been created and thus mapping is done then the Window can be considered to overlap the specified location of the Hiperspace. If data movement occurs now between Window and any other variable then, for all practical purposes it can be considered that the data movement is happening between the variable and the Hiperspace. Once the data movement is done in the view, the view needs to be saved. This is done using CSRSCOT. Finally, the view is terminated using CSRVIEW. This severs the connection between the Window and the Hiperspace and they again become two separate entities.

Usually, the storage is done multiple times in succession. This involves making a push (as described above), then advancing the offset by the window size and then again establishing a view at the new location. Then the data movement occurs and the view is terminated and the cycle keeps on repeating. Retrieval is also done in a similar fashion with just one exception – during Fetch, the data movement happens from the Hiperspace to the application program. So, it is through the offset that the application developer controls where the data resides on the memory. This gives a lot of power and flexibility in the developer's hands. But this also means that the application developer needs to handle the offset very carefully.

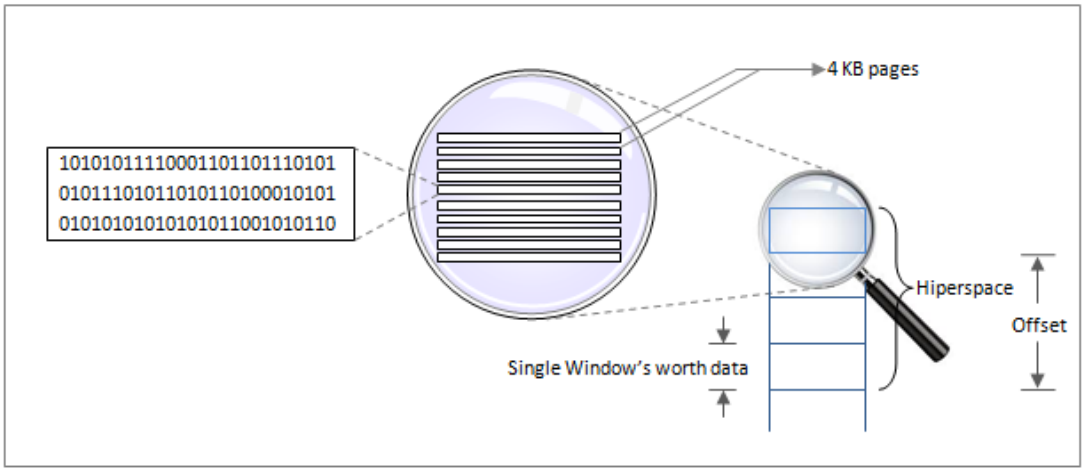

Figure 9: Data Storage in Hiperspace

VIII. Conclusion

The Hiperspace is an extension of the temporary space available at an application programmer's disposal. It uses specifically designed Window Service programs which handle the I/O operations very efficiently. Out of the entire lot of Window Service Programs, only three are used – CSRIDAC, CSRVIEW and CSRSCOT. Calling these Window Service programs in the right sequence and with the right set of parameters is the key to working with Hiperspace.

The working storage area is often enough for simple applications. However, for complex applications which need a huge temporary storage, Hiperspace should be considered as an option. In all cases where the shortage of working storage area forces the system architects to explore other design option, Hiperspace can be used to ensure a simple architecture. And in most cases where custom sorting, searching or matching operations are being done, Hiperspace can reduce the I/O overheads to a minimum. In fact, very few people know that even the sort products including DFSORT internally use Hiperspace. The business problems justifying the use of Hiperspace will be few in number. But, having this giant of storage in one's toolkit is always a plus.

References

[1]. IBM Knowledge Centre - https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/rzase/mgrtwrkstgsect.htm
[2]. IBM Support: Enterprise Cobol - http://www-01.ibm.com/support/docview.wss?uid=swg21220835
[3]. IBM Knowledge Centre - https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ieac100/ws.htm

IOSR Journal of Computer Engineering (IOSR-JCE) is UGC approved Journal with Sl. No. 5019, Journal no. 49102.

Antariksha Gupta. "Hiperspace: The Forgotten Giant of Mainframe Storage." IOSR Journal of Computer Engineering (IOSR-JCE), vol. 19, no. 5, 2017, pp. 38–46.