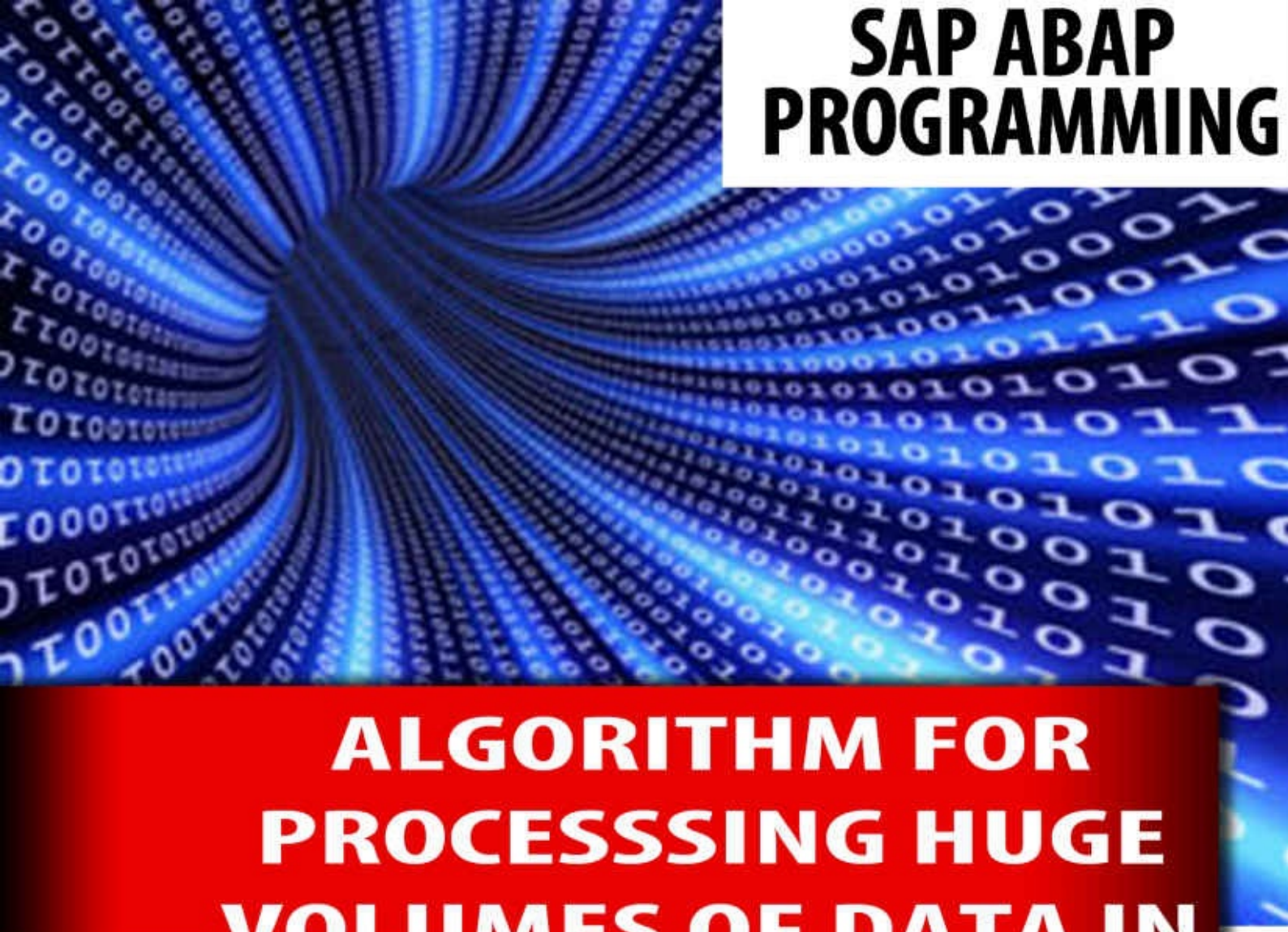
SAP ABAP PROGRAMMING



ALGORITHM FOR PROCESSSING HUGE VOLUMES OF DATA IN SAP R/3

- Develop ABAP reports that process huge volumes of data without memory short dumps.
- Use as a plug-in in any ABAP development.
- Develop fast and optimized ABAP reports

by JOHN SHANE

TABLE OF CONTENT

CHAPTER 1: INTRODUCTION

This book is intended for SAP ABAP Programmers and Functional Consultants.

This book is a culmination of experiential knowledge in performance tuning and development of programs that deal with huge volumes of data in a way that does not overwhelm the system runtime memory. Central to the book is the algorithm that enables programs/reports to successfully deal with huge volumes of data without overwhelming the SAP R/3 system memory. The algorithm has been used as a plug-in in many ABAP developments that process huge volumes of data. Key areas in developing optimized SAP programs/reports have been briefly explored. The focus for this book is the algorithm. There is already lots of literature on performance tuning and optimization of ABAP reports.

With the knowledge in this book ABAP programmers will be able to develop SAP programs or reports that process huge volumes of data without consuming excessive amount of system memory and also complete without fail. This book can be read by the SAP functional consultants too. They have to be aware that ABAP programmers can develop programs that can process huge volumes of data as long as the algorithm discussed here is plugged in. Ordinarily, after a SAP report fails for lack of system memory, the ABAP programmers will tell Functional Consultants to reduce their input parameters. They will also tell Functional Consultants to run separate reports with smaller input parameters in order to process all the data for your whole range. Well, after reading this book you can school them about the algorithm that will solve their programs' inability to handle challenging input parameters.

Chapter two will briefly highlight important areas that are crucial to a well optimized ABAP program. Chapter three will discuss the algorithm. If you are a confident ABAP programmer, you can skip right to chapter three and read about the algorithm. Chapter three gives an example on the algorithm usage.

Chapter four will look at an enhanced algorithm. And chapter five covers variations of how the same algorithm can be written. As stated before, let’s keep this book short and sweet.

CHAPTER 2: PERFORMANCE RELATED ISSUES

2.1 TYPES OF INTERNAL TABLES

The type of internal table an ABAP programmer chooses to work with has a direct impact on the performance of the developed program/report.

The most common types of internal tables a SAP ABAP developer will use are standard, sorted, and hashed. The most used one is the standard internal table. For increased performance hashed and sorted tables are the best to use. It is quicker to read from hashed and sorted table because table access is via keys. The speed with which hashed table is read does not depend on the number of records in the table. To elaborate, the time it takes to access the 1st record in an internal table is the same time it takes to access 100,000th record.

Please note that when you loop over a hashed table you lose the speed advantage. You enjoy the faster access when you use the statement READ TABLE with a hashed table.

2.2 DATBASE INDEXES AND HINTS

For increased performance, a SAP R3 program/report needs to retrieve data from the database and bring it into runtime as quick as possible. Manipulations like formatting data or calculations must be done in runtime. Do not burden the database with formatting or calculations. Unlike SAP R3, SAP HANA works by burdening the in-memory database. In order to retrieve data quickly from the DB you must formulate the WHERE conditions in such a way that your program hits right table indexes. If you are dealing with gigantic database table it might be wise to include DB Hints at the end of your WHERE statements. The DB optimizer might ignore the hints if it does not agree with the ones you specify in the hints.

As stated before, literature is available on the use of indexes and hints. The use of database indexes cannot be overstatement. It is very important your SELECT statements target the right indexes for ultra-fast data retrieval. Fast retrievals will occur if the program hits the right indexes. The DB optimizer has its own mind, though. Take time to understand how your system’s DB optimizer works and code statements that will enable program to hits the right indexes. Make use of transaction code SE05 to see what indexes have been used.

In order to achieve some of these, care must be taken to ensure that the WHERE conditions correspond to the targeted indexes. The sequence of the fields in the DB index must align with the fields in WHERE statement. From my experience the retrieval will be fast if you provide values in the conditions.

2.3 BINARY SEARCH

As a developer your first preference of an internal table type to work with should be a HASHED table. As you might already know, access speeds to the records in a hashed table is independent of the number of records. It is important that you try and see if you can use the hashed table as the first option. Many a time programmers just rush straight to declaring a standard table. However, there are a lot of instances where you will have no option but to use standard table. In such cases please use binary search function with the READ TABLE statement whenever you have to read from the standard internal table. For the binary search function to work correctly you must sort the internal table common keys fields in ascending order.

2.4 COLLECT FUNCTION

If you want to add a currency or integer or numeric field using COLLECT statement, please use a hashed internal table. Do not be lazy and just use a standard table. Using a standard internal table with the COLLECT statement will result in very poor performance.

2.5 AVOID NESTED LOOPS

Please avoid SELECT ENDSELECT statement in a loop. This statement entails the program will be making numerous trips across to the database. Expect bad performance if there are a big number of records involved. Using an array fetch is the best option.

When dealing with two internal tables that have a one-to-one relationship, then loop on one of them and use the READ TABLE statement on the other. Such an instance is a good moment to use Hashed internal table because keys available for access. If you have no choice but to loop on a second table then use parallel-cursor technique to control how your program loops in that second loop and remember to apply binary search function for faster read access.

2.6 INTO TABLE STATEMENT

A SAP R/3 ABAP program runs faster in the application server runtime. The programmer's ultimate aim is to bring records from the DB to runtime as quickly as possible. The means that not too much time must be spent retrieving data from the database. There are many dos and don'ts in this respect. As mentioned before there is plenty of literature on this matter. But one thing worth stating is that the SELECT statement must be accompanied by the INTO TABLE. Here is a warning; the sequence of the fields of the internal table must be the same as that of the database table you are retrieving from. The statement enables the program to retrieve data at fast speeds since there is no need to figure out how to arrange data into fields of the internal table. This is for the newbie: Do not use INTO CORRESPONDING FIELDS OF statement, it is slow.

2.7 DELETE STATEMENT IN A LOOP.

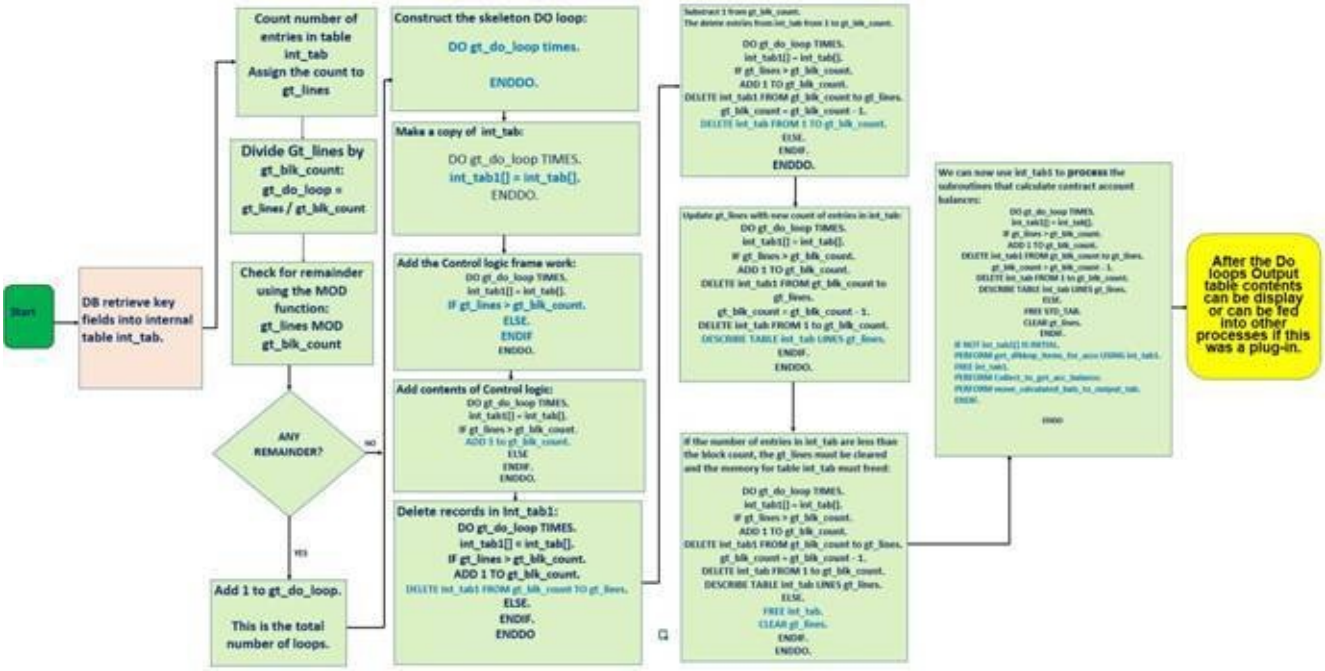
Please you must never use delete statement in a loop. You will think there is a tokoloshi in the program that makes it extremely slow. Rather transfer records into new internal table to separate what you want from what you don’t.

CHAPTER 3: THE ALGORITHM

An ABAP programmer worse nightmare is a development that crashes for lack of system runtime memory. One way to avoid memory short dumps is to develop an ABAP report that deals with one record at a time. But such a development can run for a long time before finishing. Here we will look at a way to develop ABAP programs that deal with huge volumes of data and yet not overwhelm the system runtime memory. In ABAP programmers must always develop programs that do not use too much system runtime memory and yet deal with huge volumes of data at fast speeds. Users of ABAP programs are always looking to run wide selection screen parameters that entail large volumes of data and want to be able to finish without crashing due to lack of system runtime memory.

Here we discuss an algorithm that can be used as a plug-in for any program designed to deal with vast amounts of data. The flow diagram below will show step by step things required in constructing the algorithm. Right after the flow diagram, there is a detailed description of the algorithm. It is hoped that the flow diagram and the description will be enough to make you truly understand how to use the algorithm.

3.1 THE FLOW DIAGRAM



3.2. Detailed Description:

1. In your SAP development begin by identifying a database table field that you can use as backbone/key records, e.g. contract accounts. The backbone/key records are unique and are found all along the progression of the program up to the end.

2. The backbone records/key fields should all be retrieved into a main internal table, here called Int_tab. A count the number of all entries in this main internal table must be done. Assign the count to an integer variable, here called gt_lines. Use the ABAP statement DESCRIBE TABLE to get the count (DESCRIBE TABLE int_tab LINES gt_lines).

3. The key records or backbone records are going to be processed in optimal blocks here called processing block count. This is the total number of records that will be processed in one go. Determine how big this processing block count should be and assign it to the integer variable called gt_blk_count. The size of the processing block count depends on the hardware specification, especially RAM. You have to test different counts to see which one is optimal.

4. To determine how many optimal blocks can be derived by dividing gt_lines by gt_blk_count. Remember gt_lines is the total number of records to process and gt_blk_count is the number of records that will be processed in one go. Assign the result of the division to another integer variable called gt_do_loop. The gt_do_loop is the number of times it will take to process all the gt_lines records in optimal blocks of gt_blk_count. However, because we are using integers, our division's result will not have decimals. So we have to check for any remainder by using the MOD function. If there is a remainder, then add 1 to gt_do_loop. The added 1 represents one more time that needs to be done in order to process all the remainder of records in the main internal table. The remainder records are always less than the processing block count. Hence establishing that there is a remainder from the

division is very important so that no records are left out. The variable gt_do_loop has a number that indicates how many do loops we need to execute in order to process all the data/records in the main internal table int_tab. The skeleton construct of the do-loop statement looks as follows:

```
DO gt_do_loop TIMES.

ENDDO.
```

The steps below will add more code lines inside this do-loop. *The program that has this algorithm will process loops gt_do_loop times before it proceed to the line of code outside this do loop. While looping or repeating the steps inside it, we will do certain tasks that acuminate in achieving certain of our goals. This is suitable is situations where you have to go through so much data and maybe even sum up numerous lines before getting to the desired goals. An example is summing up billing documents that normally have sub lines items and sub-sub line items.*

5. Inside the do-loop, the first line of code is for copying the key records into a new internal table int_tab1 from int_tab as highlighted code shows:

```
DO gt_do_loop TIMES.
Int_tab1[] = int_tab[].
ENDDO.
```

6. This step has the control logic that will separate key records into optimal blocks of gt_blk_count records so that they are processed blocks separately. As part of the logic, perform a check to see if gt_lines (the number of internal table entries in the main internal table) is bigger than gt_blk_count (the pre-determined block processing count). If gt_lines is greater than the gt_blk_count, then the records have to be processed in more than one processing blocks. However, if the gt_lines is less than gt_blk_count then the records would be processed in one go. The two described scenarios can happen at any point while processing the do loops. An IF statement

shall be used as highlighted below for this control logic:

```
DO gt_do_loop TIMES.
        Int_tab1[] = int_tab[].
        IF gt_lines  >  gt_blk_count.
              ELSE.
        ENDIF.
ENDDO.
```

7. Increase the block processing count by 1. The purpose of this action is to avoid overlapping. This will make much more sense later:

```
DO gt_do_loop TIMES.
Int_tab1[] = int_tab[].
IF gt_lines  >  gt_blk_count.
Gt_blk_count = gt_blk_count + 1.
ENDIF.
ENDDO.
```

8. This step continues with providing further fine details: The details are for the scenario where gt_lines is greater than the gt_blk_count. For this scenario we want to process only the first gt_blk_count records in the int_tab1, so we don’t need the rest of the records in the internal table int_tab1. We have to delete the rest of records starting from index gt_blk_count to gt_lines(which is the end):

```
DO gt_do_loop TIMES.
Int_tab1[] = int_tab[].
IF gt_lines  >  gt_blk_count.
Gt_blk_count = gt_blk_count + 1.
DELETE int_tab1 FROM gt_blk_count TO gt_lines.
ELSE.
ENDIF.
ENDDO.
```

9. Having separated the first gt_blk_count records in the copy internal table int_tab1. The main internal table int_tab has records from 1 to gt_blk_count that are also in the copy internal table int_tab1. We have to delete these from the main table, int_tab. The deletion must remove records from index 1 to gt_blk_count. But before doing that, we must subtract 1 from gt_blk_count (remember that 1 was added to gt_blk_count). The added lines of code shows how it is done:

```
DO gt_do_loop TIMES.
Int_tab1[] = int_tab[].
If gt_lines >  gt_blk_count.
Gt_blk_count = gt_blk_count + 1.
DELETE int_tab1 FROM gt_blk_count TO gt_lines.
Gt_blk_count = gt_blk_count – 1.
Delete int_tab from 1 to gt_blk_count.
ELSE.
ENDIF.
ENDDO.
```

10. The deleting of records from the main table, int_tab, means that the total number of records in it represented by gt_lines has changed. The variable gt_lines must be updated with a new count. The new count represents the still remaining records in the main internal table. The added line of code shows how to update the value in gt_lines variable.

```
DO gt_do_loop TIMES.
Int_tab1[] = int_tab[].
IF gt_lines  >  gt_blk_count.
Gt_blk_count = gt_blk_count + 1.
DELETE int_tab1 FROM gt_blk_count TO gt_lines.
Gt_blk_count = gt_blk_count – 1.
```

```
DELETE int_tab FROM 1 TO gt_blk_count.
DESCRIBE TABLE int_tab LINES gt_lines.
ENDIF.
ENDDO.
```

11. The above steps discussed the details for a scenario where gt_lines is greater than the gt_blk_count. The ELSE statement part will handle a situation in which the total number of records, gt_lines, to be processed is less than the gt_blk_count. This can happen in instances where the do-loop has processed the rest of the records or that the just initial numbers of records in the main internal table is less than the process block count, gt_blk_count. A practical example would be a situation where a user entered a smaller selection screen parameters such that the gt_lines is less than the processing block count. In any of these cases, the main internal table int_tab and gt_lines are initialized. Only the copy internal table will have records at this moment. The highlighted code shows how it is done.

```
DO gt_do_loop TIMES.
Int_tab1[] = int_tab[].
If gt_lines  >  gt_blk_count.
Gt_blk_count = gt_blk_count + 1.
DELETE int_tab1 FROM gt_blk_count TO gt_lines.
Gt_blk_count = gt_blk_count – 1.
DELETE int_tab FROM 1 TO gt_blk_count.
DESCRIBE TABLE int_tab LINES gt_lines.
ELSE.
CLEAR gt_lines.
FREE int_tab.
ENDIF
ENDDO.
```

12. Discussed above is the logic for dealing with two scenarios. Each time the loop has gone through the IF statement the copy internal table int_tab1 will have records that are optimal (gt_blk_count or less). The main internal table will have records that gt_blk_count less (or zero records if it is the last do-loop). The program can now use the records in the copy internal table as if they were the only ones it has to deal with. Doing this will ensure that the system runtime memory is not overwhelmed because only an optimal number of records is being processed. As stated before, the number of entries in a block processing count depends on how endowed your SAP server hardware. A high-end server will handle a bigger block processing count. After the control logic, an example is given of how to use the records in the copy internal table int_tab1. Used here are subroutines without the actual code just for purpose of conveying the idea of how the separated records can be used.

```
DO gt_do_loop TIMES.
Int_tab1[] = int_tab[].
If gt_lines  >  gt_blk_count.
Gt_blk_count = gt_blk_count + 1.
DELETE int_tab1 FROM gt_blk_count TO gt_lines.
Gt_blk_count = gt_blk_count – 1.
DELETE int_tab FROM 1 TO gt_blk_count.
DESCRIBE TABLE int_tab LINES gt_lines.
ELSE.
CLEAR gt_lines.
FREE int_tab.
ENDIF.
PERFORM get_dfkkop_items_for_accs USING int_tab1.
FREE int_tab1.
PERFORM Collect_to_get_acc_balance.
```

```
PERFORM move_calculated_bals_to_output_tab.
ENDDO.
```

13. The copy internal table int_tab1 is used in the first subroutine. After use, it can be freed. That will avail more memory for our processing. In the last subroutine, records of, for instance, calculated account balances are appended to an output table. The do-loop will restart the process again from the top. And actions from bullet points 5 to 12 will be applied again. This will happen gt_do_loop times. After each do-loop more records will have been appended to the output internal table, thereby growing in size. However, the records in the main internal table, int_tab will reduce in number as each do-loop is processed. The main internal table will be empty by the last do-loop.
14. After all the do-loops are processed the output internal table will have all the desired records which can be either displayed as ALV report or written to application server or send the list via RFC to an external system or even written to a database table.

In a way the do-loop will act as if we are manually running the program in separate number of times until all records are processed. You can practically run the program to calculate balances for all contract accounts on a SAP system. All that needs to be done is fill our internal table int_tab with all contract accounts on the system. The Algorithm will process these accounts in blocks of gt_blk_count count using a temp internal table. In our case the temp table is int_tab1. After processing the accounts in internal table int_tab1, this temp table is emptied so that next block of accounts can be filled up for the next do loop. This is repeated until all records/ contract accounts' balances are calculated and appended to output table.

CHAPTER 4: EXAMPLE

An example we can look at is a report that gets street and postal addresses for all the business partners on a system with 50 million Bps.

Our ABAP report would have to do the following steps:

1. Get all business partners from table BUT000 and put them internal table int_bp. This internal table can have one field/column… always be mindful of the memory you are consuming. For this example gt_lines will be equal 50, 000, 000.
2. Determine what the block processing count will be. Let's say the block processing count is 10 million. So gt_blk_count = 10,000.000. With these parameters we can calculate many do loops the program will have to do to finish processing 50 million business partners.
3. Construct the do loop:

   ```
   DO 5 TIMES.

   ENDDO.
   ```

4. Inside the do loop, we will copy int_bp into int_bp1.

   ```
   DO 5 TIMES.
   Int_bp1[] = int_bp[].
   ENDDO.
   ```

5. Introduce a test for checking whether the total number of entries, represented by gt_lines, in the main internal table int_bp is greater than the block processing count, gt_blk_count.

   ```
   DO 5 TIMES.
   Int_bp1[] = int_bp[].
   IF gt_lines > gt_blk_count.
   ELSE.
   ```

ENDIF.

ENDDO.

6. We want to process only the first 10 million BPs in the table int_bp1, so we can delete the rest of the records for the first do loop. We have to delete from int_bp1 index 10,000,001 to index 50,000,000. See the code below:

   DO 5 TIMES.

   Int_bp1[] = int_bp[].

   IF gt_lines > gt_blk_count.

   (gt_blk_count = gt_blk_count + 1)

   DELETE int_bp1 FROM 10,000,001 TO 50,000,000.

   ENDIF.

   ENDDO.

7. The main internal table, int_bp has records from 1 to 10,000,000 that are also in table int_bp1. We can delete these from the main internal table.

   DO 5 TIMES.

   Int_bp1[] = int_bp[].

   IF gt_lines > gt_blk_count.

   (gt_blk_count = gt_blk_count + 1)

   DELETE int_bp1 FROM 10,000,001 TO 50,000,000.

   (gt_blk_count = gt_blk_count - 1)

   DELETE int_bp FROM 1 TO 10,000,000.

   ENDIF.

   ENDDO.

8. By deleting the records from the main table, int_bp that means that the total number of records represented by gt_lines has changed. The variable gt_lines must be updated with the new number of records in int_bp. There are now 40,000,000 records in table int_bp.

DO 5 TIMES.

Int_bp1[] = int_bp[].

IF gt_lines > gt_blk_count.

(gt_blk_count = gt_blk_count + 1)

DELETE int_bp1 FROM 10,000,001 TO 50,000,000.

(gt_blk_count = gt_blk_count - 1)

DELETE int_bp FROM 1 TO 10,000,000.

DESCRIBE TABLE int_bp LINES gt_lines.

ENDIF.

ENDDO.

9. Our IF statement can have the ELSE part. This will handle the otherwise situation where the number of records in the main internal table is less than the block processing count. Our example will not really demonstrate how this part of the program because there are exactly 5 needed to process 50 million records in 5 loops.

DO 5 TIMES.

Int_bp1[] = int_bp[].

IF gt_lines > gt_blk_count.

(gt_blk_count = gt_blk_count + 1)

DELETE int_bp1 FROM 10,000,001 TO 50,000,000.

(gt_blk_count = gt_blk_count - 1)

DELETE int_bp FRM 1 TO 10,000,000.

DESCRIBE TABLE int_bp LINES gt_lines.

ELSE.

FREE int_bp.

CLEAR gt_lines.

ENDIF.

ENDDO.

10. At this point we are ready to process the records in internal table int_bp1. I will add

three subroutines without source code but hope that you shall understand what is going on.

```
DO 5 TIMES.
Int_bp1[] = int_bp[].
IF gt_lines > gt_blk_count.
(gt_blk_count = gt_blk_count  + 1 )
DELETE int_bp1 FROM 10,000,001 TO 50,000,000.
(gt_blk_count = gt_blk_count - 1)
DELETE int_bp FROM 1 TO 10,000,000.
DESCRIBE TABLE int_bp LINES gt_lines.
ELSE.
FREE int_bp.
CLEAR gt_lines.
ENDIF.
PERFORM get_addr_numbers_but020USING int_bp1.
FREE int_bp1.
PERFORM get_str_and_post_addr_adrc.
PERFORM append_to_output_table.
ENDDO.
```

11. Let's take further. Once the do loop has finish the first loop it will start the second one. Things will look as follows: Notice that the number has reduced to 40, 000,000.

```
DO 5 TIMES.
Int_bp1[] = int_bp[].
IF gt_lines > gt_blk_count.
(gt_blk_count = gt_blk_count  + 1 )
DELETE int_bp1 FROM 10,000,001 TO40,000,000.
(gt_blk_count = gt_blk_count - 1)
DELETE int_bp FROM 1 TO 10,000,000.
```

```
DESCRIBE TABLE int_bp LINES gt_lines.
ELSE.
FREE int_bp.
CLEAR gt_lines.
ENDIF.
PERFORM get_addr_numbers_but020 USING int_bp1.
FREE int_tab1.
PERFORM get_str_and_post_addr_adrc.
PERFORM append_to_output_table.
ENDDO.
```

12. On third loop, the total number of records in the main internal table will reduce to 30,000,000:

```
DO 5 TIMES.
Int_bp1[] = int_bp[].
IF gt_lines > gt_blk_count.
(gt_blk_count = gt_blk_count  + 1 )
DELETE int_bp1 FROM 10,000,001 TO30,000,000.
(gt_blk_count = gt_blk_count - 1)
DELETE int_bp FROM 1 TO 10,000,000.
DESCRIBE TABLE int_bp LINES gt_lines.
ELSE.
FREE int_bp.
CLEAR gt_lines.
ENDIF.
PERFORM get_addr_numbers_but020 USING int_bp1
FREE int_bp1.
PERFORM get_str_and_post_addr_adrc.
PERFORM append_to_output_table.
```

ENDDO.

13. Processing the fourth loop will leave 20,000,000 records in the main table. While doing these loops internal table Int_tab will continue becoming small since records are being deleted from it, the output table will grow is size with every loop processed as records get appended to it.

DO 5 TIMES.

Int_bp1[] = int_bp[].

IF gt_lines > gt_blk_count.

(gt_blk_count = gt_blk_count + 1)

DELETE int_bp1 FROM 10,000,001 TO 20,000,000.

(gt_blk_count = gt_blk_count - 1)

DELETE int_bp FROM 1 TO 10,000,000.

DESCRIBE TABLE int_bp LINES gt_lines.

ELSE.

FREE int_bp.

CLEAR gt_lines.

ENDIF.

PERFORM get_addr_numbers_but020 USING int_bp1.

FREE int_bp1.

PERFORM get_str_and_post_addr_adrc.

PERFORM append_to_output_table.

ENDDO.

14. After all the 5 loops are processed, internal tables int_bp and int_bp1 will be empty and the output table will have the all the business partners' street and postal addresses. You can output the internal table with the street and postal addresses to an ALV report or write a text file to application server or even write to a database table.

CHAPTER 5: EHANCED ALGORITHM

We have a good understanding of how the algorithm works. We can now enhance it by adding other features related system runtime memory management. You should have notice that in addition to optimization it is all about managing the usage of system run time memory. Processing is done in sizeable chunks so as not overwhelm the system runtime memory. In the last example we had an output internal table that was growing with every loop processed because new records are appended to it. We don’t want a situation where the output table grows to a point that it overwhelms the system runtime memory and causes program short-dumps. So we can add some feature to the algorithm that takes care of this growing internal table. We will add some program logic that enables us to write the contents of the output internal table whenever the total number of internal table lines hit a certain pre-determined threshold. Let’s say the threshold is 30 million records. (If you are going to import the data into Microsoft Excel, you should make the threshold almost as equal to the maximum number of rows an Excel worksheet supports.) The moment the output table has 30 million or more lines in it, the program logic will call a subroutine to write all the contents of the output table to the App Server. Once the all contents in the output table have been written to the app server, we can clear the output table thereby reducing the memory usage substantially. New records can be appended to the output table until the total number hits threshold. A situation where there is only one do-loop to process is another trigger for writing to the app server. Below is added source code to show this concept.

```
DATA gt_threshold TYPE i DEFAULT 30, 000,000.
DATA lv_remaining_do TYPE i.
lv_remaining_do = gt_do_loop.
lv_remaining_do = 5.
    DO 5 TIMES.
    lv_remaining_do = lv_remaining_do– 1.
    Int_bp1[] = int_bp[].
    IF gt_lines > gt_blk_count.
    (gt_blk_count = gt_blk_count + 1)
    DELETE int_bp1 FROM 10,000,001 TO 20,000,000.
    (gt_blk_count = gt_blk_count - 1)
    DELETE int_bp FROM 1 TO 10,000,000.
    DESCRIBE TABLE int_bp LINES gt_lines.
    ELSE.
    FREE int_bp.
    CLEAR gt_lines.
    ENDIF.
    PERFORM get_addr_numbers_but020 USING int_bp1.
    FREE int_bp1.
    PERFORM get_str_and_post_addr_adrc.
    PERFORM append_to_output_table.
    DESCRIBE output_tab LINES gt_threshold.
    IF gt_threshold>= 30,000,000 OR lv_remaining_do = 1.
    PERFORM write_to_App_server.
    FREE: output_tab.
    ENDIF.
    ENDDO.
```

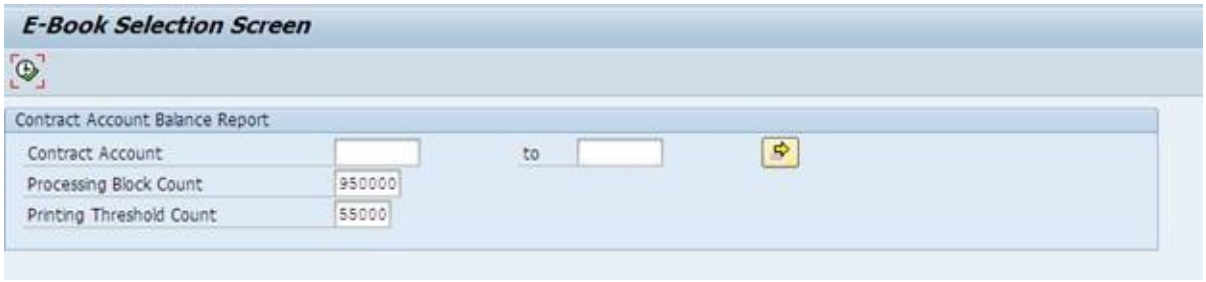
You can actually run the program for an entire SAP system without worrying about memory problems, with such an enhanced algorithm.

You all know that it is best practice not to hard-code numbers. So far in this book, the

concept of processing block count has been presented as if the ABAP developer will pick a number and hard-code in the program. The users of the developed programs can have control over the value of the processing block count. The selection screen parameters can include the processing block count so that the value can be adjusted by users of the program. The higher the number the faster the program shall finish since there would fewer do loops. However, care must be taken to ensure that the count is not so big that the system memory is depleted leading to program crashes. Different counts can be tested until an optimal number is found. An optimal count depends on the hardware resources on the server. As mentioned before, a well-heeled server will have a big optimal count.

See an example of the selection screen below:

If the report has the ability to write to the App server, then the printing threshold count can also be included on the screen, as shown below:



5. VARIATIONS OF THE ALGORITHM.

There are two more ways of specifying the algorithm discussed above. The preference is up to the reader.

5.1 USING WHILE STATEMENT.

```
Int_tab1[] = int_tab[]

Gt_blk_count = gt_blk_count + 1.

Delete int_tab1 from index gt_blk_count to gt_lines.

Gt_blk_count = gt_blk_count – 1.

Delete int_tab from index 1 to gt_blk_count.

Describe table int_tab lines gt_lines.

WHILE int_tab1[] is not initial.

PERFORM get_dfkkop_items_for_accs USING int_tab1.

FREE int_tab1.

PERFORM Collect_to_get_acc_balance.

PERFORM move_calculated_bals_to_output_tab.

Int_tab1[] = int_tab[]

Gt_blk_count = gt_blk_count + 1.

Delete int_tab1 from index gt_blk_count to gt_lines.

Gt_blk_count = gt_blk_count – 1.

Delete int_tab from index 1 to gt_blk_count.

Describe table int_tab lines gt_lines.

ENDWHILE.
```

5.2 USING DO STATEMENT

```
DO.

Int_tab1[] = int_tab[]

IF int_tab1[] is initial.
EXIT.
ENDIF.

Gt_blk_count = gt_blk_count + 1.

DELETE int_tab1 FROM gt_blk_count TO gt_lines.

Gt_blk_count = gt_blk_count – 1.

DELETE int_tab FROM 1 TO gt_blk_count.

.DESCRIBE TABLE int_tab LINES gt_lines.

PERFORM get_dfkkop_items_for_accs USING int_tab1.

FREE int_tab1.

PERFORM Collect_to_get_acc_balance.

PERFORM move_calculated_bals_to_output_tab.

ENDDO.
```