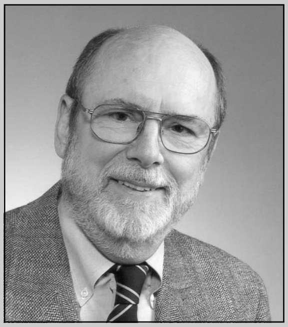
Learn How to Avoid Lackluster System Performance When Using the "FOR ALL ENTRIES IN" Clause: An ABAP Developer's Guide

David F. Jenkins



David Jenkins joined RCG Information Technology in 1995 after 29 years as a Consulting Systems Representative at IBM. Currently he is a Director Consultant at RCG, where he develops and teaches classes in beginning and advanced ABAP programming, ABAP performance and tuning, and Java, and consults with clients on all aspects of ABAP development.

(complete bio appears on page 114)

All ABAP programmers are familiar with the Open SQL SELECT … FOR ALL ENTRIES IN construct[1] frequently used to access data from a database conditioned on other data that has been previously loaded into an internal ABAP table. For instance, the ABAP statement:

```
select * from mseg
         for all entries in int_driver
         where mblnr = int_driver-mblnr.
```

instructs the system to access all rows from the database table MSEG where the MBLNR field contents in the database match any one of the MBLNR fields in the internal table INT_DRIVER. One recent study that came across my desk concluded that 10 percent of all SELECT statements issued in custom programs employ FAEI.

The use of the FAEI construct is not without significant implications, especially as it pertains to program performance — I have seen an actual case where a SELECT ran for 16+ hours using a poorly planned FAEI. (This time dropped to 4 minutes after rewriting the SELECT statement to remove the FOR ALL ENTRIES IN clause.)

In this article, I will present information that goes far beyond standard SAP documentation — I will explain a little about how FAEI *really* works, and some of the more significant ramifications of its use. You will learn techniques to minimize the performance impact of FAEI, and techniques commonly used to evaluate the impact of various approaches for using FAEI.

[1] For simplicity's sake, I will contract that to FAEI for the remainder of this article.

For ABAP programmers, this article may supply additional detailed information regarding a common programming construct; for DBAs, this article may provide insight into the need for adjustments to database table settings; and for you development managers struggling with lackluster system performance, this article is intended to provide information at a high enough level that you can effectively guide your development staff in its use of this common Open SQL construct. I will also discuss alternative `SELECT` strategies that you may be able to employ to bolster program performance. And finally, as an added bonus, I will reveal an Oracle rule-based optimizer peculiarity that may be costing your installation hundreds of lost hours per month.

Before beginning this discussion, let me state that the thoughts to follow throughout this article result from experiences with SAP Releases 3.1H and 4.6B using Oracle. All SQL traces shown were run on a 3.1H system, which was *not* configured with separate application and database servers, and hence the impact of network delays is minimized — on systems with separate database servers you may observe markedly different results. If you are using a database other than Oracle, or earlier or later releases of SAP, the tips to follow may not be directly applicable but should nevertheless be of interest. With those caveats out of the way, let's begin.

What Does "FOR ALL ENTRIES IN" Do?

So what *does* the `FOR ALL ENTRIES IN` clause do in Open SQL `SELECT` constructs? I'll start by quoting directly from the 4.6B Help Library regarding `FOR ALL ENTRIES IN` (bracketed text is my own):

> *The WHERE clause of [an Open SQL] SELECT statement has a special variant that allows you to derive conditions from the lines and columns of an internal table:*
>
> *SELECT ...* ***FOR ALL ENTRIES IN <itab> WHERE <cond>*** *...*
>
> *<cond> [is a normal selection condition and] may be formulated as described above [in previous Help text]. If you specify a field of the internal table <itab> as an operand in a condition, you address all lines of the internal table. The comparison is then performed for each line of the internal table. For each line, the system selects the lines from the database table that satisfy the condition. The result set of the SELECT statement is the union of the individual selections for each line of the internal table. Duplicate lines are automatically eliminated from the result set. [More on this later.] If <itab> is empty, the addition FOR ALL ENTRIES [IN] is disregarded, and all entries are read.*
>
> .
>
> .
>
> .
>
> *You can use the option FOR ALL ENTRIES [IN] to replace nested select loops by operations on internal tables. This can significantly improve the performance for large sets of selected data. [Maybe — the operative word here is "can." More on this later, also.]*

So, the `FOR ALL ENTRIES [IN]` clause is used primarily to reduce the number of ABAP statements in a program. In particular, it is useful for reducing the number of nested `LOOP` or nested `SELECT` statements in a program, and accomplishes an inner join between data in an internal table and a database table.

Making Performance a Priority

From a programming viewpoint, `FAEI` is one of several techniques that can be used to reduce the amount of ABAP code to be written when joining multiple tables. As always, however, when accessing database data, performance must be a primary consideration. Among the concerns that must be addressed when using FAEI are the efficient utilization of transport buffers between the database and the application, and the elimination of retrieval and transport of redundant information.

> ✓ ***Tip***
>
> *Database data is passed back and forth to the application in buffered blocks, and the size, number, and buffering scheme of these transport buffers is highly dependent on hardware, software, and database configuration. You will find a wealth of notes in OSS that deal with configuration issues pertaining to database buffering — our concern here will be to point out areas where we can take best advantage of whatever has already been installed.*

Consider the following ABAP statement from the sample program[2] used in generating most of the results described in this article:

```
select     mblnr
           mjahr
           zeile
from mseg[3] up to 2 rows
into table int_mseg
for all entries in int_driver
where mblnr = int_driver-mblnr
and   mjahr = int_driver-mjahr.
```

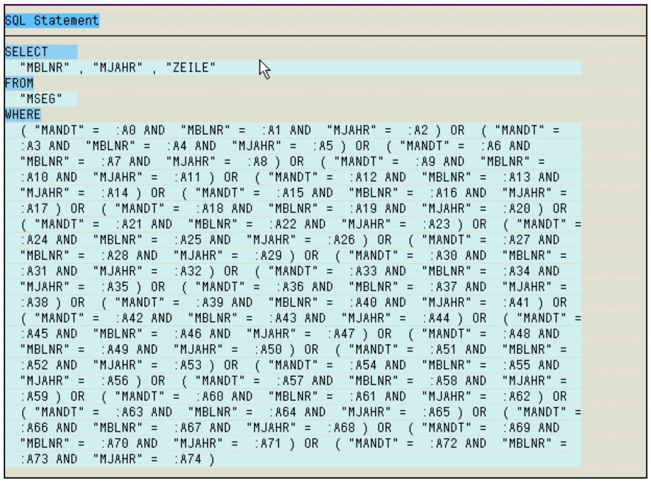
The INT_DRIVER table referenced in the FOR ALL ENTRIES IN clause is known as a *driver* or *reference* table. When the FOR ALL ENTRIES IN clause is used, the SAP database interface creates a WHERE clause that translates the entries in the driver table into separate conditions, which are then combined through the use of ORs with WHERE conditions specified in the SELECT itself.

Understanding How Open SQL SELECT Statements Are Translated

A first step in understanding Open SQL SELECT statements is to understand how they are translated into SELECT statements that are usable by the database. The easiest way to see the actual database SELECT derived from an ABAP SELECT is to SQL trace your program, and then request an *Execution Plan* for the SELECT of interest.

In a test program using the previous code example, the ABAP database interface created the SQL statement shown in **Figure 1**.

Figure 1 The SQL Statement Created By the ABAP Database Interface



SQL Statement

```
SELECT
  "MBLNR" , "MJAHR" , "ZEILE"
FROM
  "MSEG"
WHERE
  ( "MANDT" =  :A0 AND  "MBLNR" =  :A1 AND  "MJAHR" =  :A2 ) OR  ( "MANDT" =
  :A3 AND  "MBLNR" =  :A4 AND  "MJAHR" =  :A5 ) OR  ( "MANDT" =  :A6 AND
  "MBLNR" =  :A7 AND  "MJAHR" =  :A8 ) OR  ( "MANDT" =  :A9 AND  "MBLNR" =
  :A10 AND  "MJAHR" =  :A11 ) OR  ( "MANDT" =  :A12 AND  "MBLNR" =  :A13 AND
  "MJAHR" =  :A14 ) OR  ( "MANDT" =  :A15 AND  "MBLNR" =  :A16 AND  "MJAHR" =
  :A17 ) OR  ( "MANDT" =  :A18 AND  "MBLNR" =  :A19 AND  "MJAHR" =  :A20 ) OR
  ( "MANDT" =  :A21 AND  "MBLNR" =  :A22 AND  "MJAHR" =  :A23 ) OR  ( "MANDT" =
  :A24 AND  "MBLNR" =  :A25 AND  "MJAHR" =  :A26 ) OR  ( "MANDT" =  :A27 AND
  "MBLNR" =  :A28 AND  "MJAHR" =  :A29 ) OR  ( "MANDT" =  :A30 AND  "MBLNR" =
  :A31 AND  "MJAHR" =  :A32 ) OR  ( "MANDT" =  :A33 AND  "MBLNR" =  :A34 AND
  "MJAHR" =  :A35 ) OR  ( "MANDT" =  :A36 AND  "MBLNR" =  :A37 AND  "MJAHR" =
  :A38 ) OR  ( "MANDT" =  :A39 AND  "MBLNR" =  :A40 AND  "MJAHR" =  :A41 ) OR
  ( "MANDT" =  :A42 AND  "MBLNR" =  :A43 AND  "MJAHR" =  :A44 ) OR  ( "MANDT" =
  :A45 AND  "MBLNR" =  :A46 AND  "MJAHR" =  :A47 ) OR  ( "MANDT" =  :A48 AND
  "MBLNR" =  :A49 AND  "MJAHR" =  :A50 ) OR  ( "MANDT" =  :A51 AND  "MBLNR" =
  :A52 AND  "MJAHR" =  :A53 ) OR  ( "MANDT" =  :A54 AND  "MBLNR" =  :A55 AND
  "MJAHR" =  :A56 ) OR  ( "MANDT" =  :A57 AND  "MBLNR" =  :A58 AND  "MJAHR" =
  :A59 ) OR  ( "MANDT" =  :A60 AND  "MBLNR" =  :A61 AND  "MJAHR" =  :A62 ) OR
  ( "MANDT" =  :A63 AND  "MBLNR" =  :A64 AND  "MJAHR" =  :A65 ) OR  ( "MANDT" =
  :A66 AND  "MBLNR" =  :A67 AND  "MJAHR" =  :A68 ) OR  ( "MANDT" =  :A69 AND
  "MBLNR" =  :A70 AND  "MJAHR" =  :A71 ) OR  ( "MANDT" =  :A72 AND  "MBLNR" =
  :A73 AND  "MJAHR" =  :A74 )
```

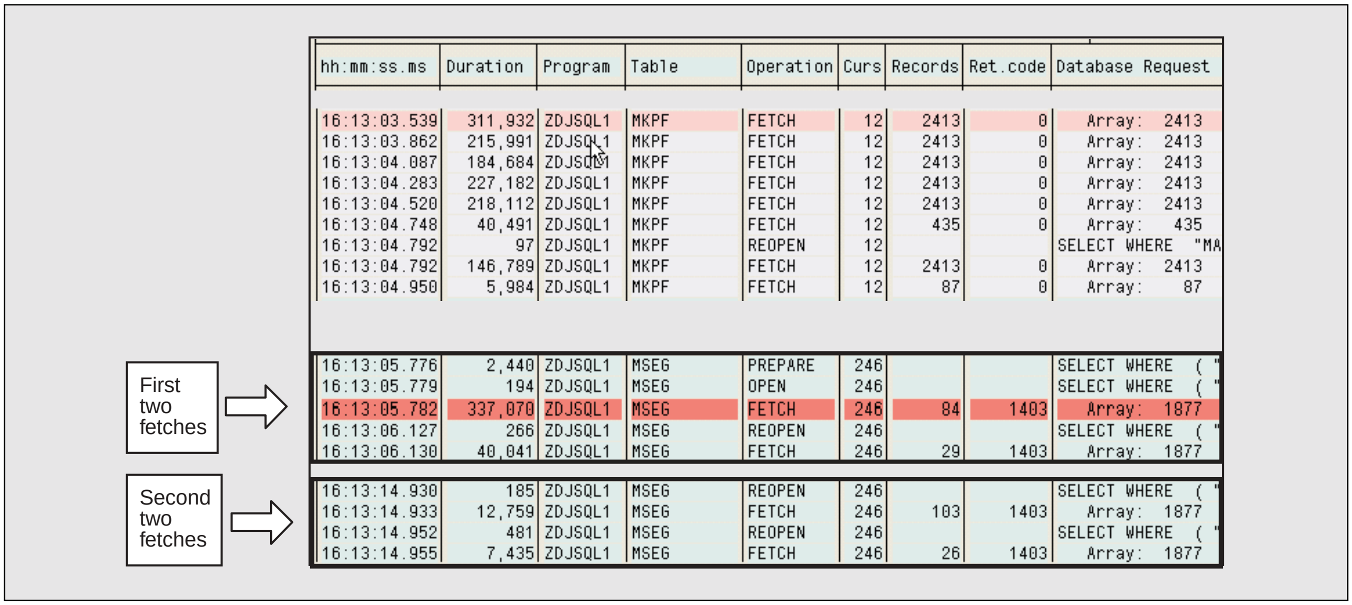
> ✓ ***Tip***
>
> *Evaluation of SELECT performance starts with a thorough understanding of SAP SQL Trace (ST05) and SQL Statement Explain (SDBE) — these two capabilities constitute the core of almost all ABAP SELECT performance and tuning activity.*

In this particular selection, the driver table contained 15,000 entries, each of which furnished a MBLNR/MJAHR pair. Note, however, that although there are 15,000 distinct rows in the driver table, this

[2] The complete listing of this program is in Appendix A. If you would like the program in soft form, contact the author at djenkins@pointecom.net, or you can download it from **www.SAPpro.com**.

[3] The MSEG table used to illustrate this article was unbuffered, and contained about 17 million rows.

Figure 2 *Partial SQL Trace for the SELECT Statement*



hh:mm:ss.ms	Duration	Program	Table	Operation	Curs	Records	Ret.code	Database Request
16:13:03.539	311,932	ZDJSQL1	MKPF	FETCH	12	2413	0	Array: 2413
16:13:03.862	215,991	ZDJSQL1	MKPF	FETCH	12	2413	0	Array: 2413
16:13:04.087	184,684	ZDJSQL1	MKPF	FETCH	12	2413	0	Array: 2413
16:13:04.283	227,182	ZDJSQL1	MKPF	FETCH	12	2413	0	Array: 2413
16:13:04.520	218,112	ZDJSQL1	MKPF	FETCH	12	2413	0	Array: 2413
16:13:04.748	40,491	ZDJSQL1	MKPF	FETCH	12	435	0	Array: 435
16:13:04.792	97	ZDJSQL1	MKPF	REOPEN	12			SELECT WHERE "MA
16:13:04.792	146,789	ZDJSQL1	MKPF	FETCH	12	2413	0	Array: 2413
16:13:04.950	5,984	ZDJSQL1	MKPF	FETCH	12	87	0	Array: 87
16:13:05.776	2,440	ZDJSQL1	MSEG	PREPARE	246			SELECT WHERE ("
16:13:05.779	194	ZDJSQL1	MSEG	OPEN	246			SELECT WHERE ("
16:13:05.782	337,070	ZDJSQL1	MSEG	FETCH	246	84	1403	Array: 1877
16:13:06.127	266	ZDJSQL1	MSEG	REOPEN	246			SELECT WHERE ("
16:13:06.130	40,041	ZDJSQL1	MSEG	FETCH	246	29	1403	Array: 1877
16:13:14.930	185	ZDJSQL1	MSEG	REOPEN	246			SELECT WHERE ("
16:13:14.933	12,759	ZDJSQL1	MSEG	FETCH	246	103	1403	Array: 1877
16:13:14.952	481	ZDJSQL1	MSEG	REOPEN	246			SELECT WHERE ("
16:13:14.955	7,435	ZDJSQL1	MSEG	FETCH	246	26	1403	Array: 1877

particular `SELECT` encodes the selection conditions represented in only 25 of those rows, and these conditions are "`OR`'d" together. The database rows satisfying this particular request will be fetched, and then another request will be sent to the database representing the next 25 values in the driver table. In this manner, repetitive requests will be issued to the database until all 15,000 driver table conditions have been requested.

You can see then that depending on the number of rows in the driver table, a large number of `OR` conditions can be generated. To calculate the number of `OR` statements generated for each SQL `SELECT`, the R/3 work process takes the *smaller* of the following:

- The number of entries in the driver table `INT_DRIVER`
- The *rsdb/max_blocking_factor* R/3 profile parameter

If the number of entries in the driver table is larger than *rsdb/max_blocking_factor*, the work process executes several similar SQL statements on the database to limit the length of the generated SQL `WHERE`. Each SQL statement is executed until all driver table entries have been used, and then the R/3 work process joins the partial results into a final result set that excludes duplications.

Figure 2 displays part of the SQL trace for the previous `SELECT`. There's a whole bunch of revealing information here: the first few fetches (from `MKPF`, starting at time `16:13:03.539`) were used to load the driver table, and you can see that 15,000 (6 * 2413 + 435 + 87) rows from `MKPF` were returned by the database. (As we'll see shortly, for illustrative purposes, the sample program was written such that 2,500 of the 15,000 rows were duplicates.)

✓ *Tip*

The SAP-supplied program RSPARAM can be used to display current R/3 profile parameter settings. The optimum setting for rsdb/max_blocking_factor is highly dependent on platform, database (and database version), and database optimizer. This number may be quite low — recommendations found in OSS notes range from 5 to 50. In our example, this parameter has been set to 25. Since there are 15,000 entries in the driver table, an rsdb/max_blocking_factor setting of 25 will result in processing 600 separate database requests, each with its attendant processing overhead, to satisfy the original SELECT.

Figure 3 *Execution Time Table for the “Baseline” SELECT Scenario*

SELECT Strategy	Total Fetches	Total Time (seconds)	Time per Fetch (ms)	Record Count	Time per Record (µs)	Records per Database Fetch			
						mean	s.d.	min	max
Baseline	600	9.173	15.3	34,228	268	57.0	7.1	29	110
Sorted									
Dupes removed									
Key Value range (1)									
Key Value range (2)									

The entire trace of MSEG accesses is too lengthy to reproduce here (recall that there were 600 separate cursor REOPENs and FETCHes) — only the first and last two trace entries are shown in Figure 2. The complete trace showed, however, that 34,228 records were transmitted to the application, even though we only asked for a maximum of but two. Returning to the generated SQL (Figure 1), we see that the UP TO 2 ROWS specified in the ABAP SELECT has been ignored, inasmuch as it pertains to the actual generated SQL.

Why? A partial answer can be found in the ABAP Help text for FAEI, which states, in part:

> *“SELECT ... FOR ALL ENTRIES IN itab WHERE cond returns the union of the solution sets of all SELECT statements that would result if you wrote a separate statement for each line of the internal table replacing the symbol itab-f with the corresponding value of component f in the WHERE condition. Duplicates are discarded from the result set...”*

This tells us that *all* records satisfying the SELECT are returned, *then* duplicates are removed, and presumably then, and only then, is the two-record limitation imposed.

We’ll leave further vagaries of the database interface to another discussion, but suffice to say here that although we asked for but two records, 34,228 were transmitted to the application server from the database server. Only two, however, were actually stored in the internal result set table (INT_MSEG).

If we were to look at the entire SQL trace for this operation, we would see that each of the 600 fetches returned an average of 57.0 records — far fewer than the 1,877 per fetch the system was capable of. This represents a very low mean transport buffer utilization — about 3 percent.

> ✓ ***Tip***
>
> *In SAP Release 3.1x, the Database Request column of a FETCH SQL trace line contains the word “Array,” followed by a number — that number represents the maximum number of records that can be transported per fetch request. In Release 4.6, this maximum number appears in a column of its own, labeled “Array.”*

Comparing the Cost of Various FAEI Strategies

The total elapsed time for the baseline SELECT scenario we just discussed was 9.173 seconds. In order to compare the results of various FAEI strategies, I’ll build a table of execution times as we go (see **Figure 3**).

Figure 4 *Execution Time Table for the "Sorted" SELECT Scenario*

SELECT Strategy	Total Fetches	Total Time (seconds)	Time per Fetch (ms)	Record Count	Time per Record (µs)	Records per Database Fetch			
						mean	s.d.	min	max
Baseline	600	9.173	15.3	34,228	268	57.0	7.1	29	110
Sorted	600	7.982*	13.3	28,572	279	47.6	17.3	13	92
Dupes removed									
Key Value range (1)									
Key Value range (2)									

* This time does *not* include the time required to sort the data prior to selection. However, our experience has shown that ABAP is a very efficient sorter, and that this time can be safely ignored for comparison purposes here.

The columns in Figure 3 have the following meanings:

- **Total Fetches:** Total number of data transfers from database to application
- **Total Time:** Elapsed time from cursor open to completion of last fetch (seconds)
- **Time per Fetch:** Total Time / Total Fetch Count (milliseconds)
- **Record Count:** Number of database rows passed to application
- **Time per Record:** Total Time / Record Count (microseconds)
- **Records per Database Fetch:** Statistics pertaining to the number of rows of data passed in each fetch

The number of records (and hence bytes) returned per fetch (mean = 57.0, in this case) can have a profound impact on performance. The Oracle system used for this exercise will return data to the application server in packets of about 34K bytes. Since we were requesting only 18 bytes per record (fields `MBLNR`, `MJAHR`, and `ZEILE`), in the case of the first fetch (Figure 2, at time 16:13:05.782) the 34K buffer contained only 1,512 (84 * 18) bytes of usable information. The other fetches were similarly inefficient, with the least efficient using only 522 bytes of the 34K transmitted for useful information, and the average only 1,026. In those cases where data is being transmitted across a network, this wasted bandwidth can result in significant performance degradation.

A "Sorted" SELECT

One approach to minimizing total fetch time might be to find some mechanism for inducing the database to find data more quickly. For a given fetch request, would it be faster for the database to retrieve records if the values requested were "closer"? If you read the code in the sample program in Appendix A, you'll see that the `MBLNR`s in the driver table were loaded in pseudo-random order. In the second `SELECT`, the driver table `MBLNR`s have been sorted — the results of that run are reflected in **Figure 4**.

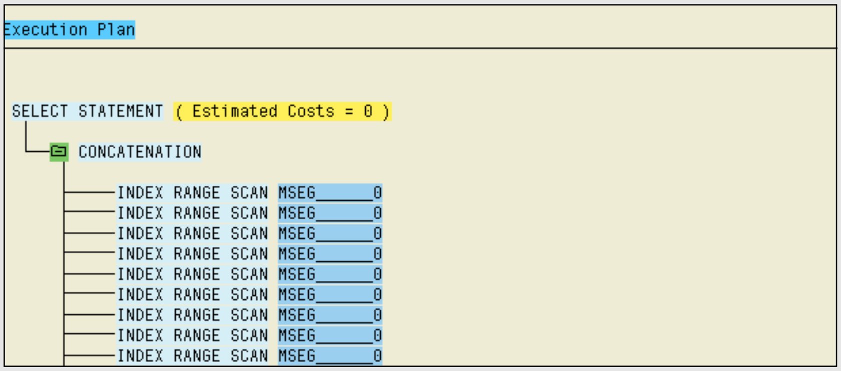
In this case, even though the mean time per record increased slightly (from 268 to 279 µs), the total number of records transported dropped dramatically — from 34,228 to 28,572 — about 17 percent. Because the values used to do the selection had been sorted, the probability of duplicate values occurring in a single fetch request *increased*, and hence the need for Oracle to pass back redundant data was largely removed. The resulting total `SELECT` time was somewhat better than our baseline case, and numerous other tests show that there are marginally better results when using this technique: sort the driver table on the values that will be used to select the data.

Figure 5 *Execution Time Table for the “Dupes Removed” SELECT Scenario*

SELECT Strategy	Total Fetches	Total Time (seconds)	Time per Fetch (ms)	Record Count	Time per Record (µs)	Records per Database Fetch			
						mean	s.d.	min	max
Baseline	600	9.173	15.3	34,228	268	57.0	7.1	29	110
Sorted	600	7.982*	13.3	28,572	279	47.6	17.3	13	92
Dupes removed	500	6.955	13.9	28,342	245	56.7	13.9	25	92
Key Value range (1)									
Key Value range (2)									

* This time does *not* include the time required to sort the data prior to selection. However, our experience has shown that ABAP is a very efficient sorter, and that this time can be safely ignored for comparison purposes here.

Figure 6 *Partial Execution Plan Generated for the Sample SELECT*



```
Execution Plan

SELECT STATEMENT ( Estimated Costs = 0 )
  └─ CONCATENATION
       ├──INDEX RANGE SCAN MSEG______0
       ├──INDEX RANGE SCAN MSEG______0
       ├──INDEX RANGE SCAN MSEG______0
       ├──INDEX RANGE SCAN MSEG______0
       ├──INDEX RANGE SCAN MSEG______0
       ├──INDEX RANGE SCAN MSEG______0
       ├──INDEX RANGE SCAN MSEG______0
       ├──INDEX RANGE SCAN MSEG______0
       ├──INDEX RANGE SCAN MSEG______0
```

Removing Duplicates

Perhaps it’s not the time per fetch that should be attacked, but the number of fetches required to retrieve all the desired data — how can this be reduced? Since each fetch request only retrieves data satisfying that particular request, we run the risk of retrieving duplicate records (which must then be discarded by the SAP system).

One simple way to eliminate the transmission of duplicate data is to reduce the duplicates in the driver table. The third `SELECT` in the sample program is executed after having removed duplicate `MBLNR/MJAHR` entries. Even though the number of records returned by Oracle has remained roughly the same, making this simple change has allowed us to reduce the time/record by about 12 percent (**Figure 5**).

Using Index Range Scans

Once we have optimized the contents of the driver table, it’s time to look to the database itself to find further possibilities for enhancing `FAEI` processing. The number of entries in the driver table, the fields used in the `WHERE` clause, and the database optimizer, among other factors, all drive the execution plan for a given `SELECT`.

Part of the execution plan generated for the sample `SELECT` looks like **Figure 6**.

Note the multiple index range scans — each scan is caused by a line in the driver table (and hence an associated OR clause in the generated SQL statement). So, depending on the contents of the driver table, we may end up with many SQL SELECTs being generated, and each of those may result in multiple index scans.

The index scans are generated in this sample because the SELECT included a WHERE on MANDT (supplied by the database interface, unless we use CLIENT SPECIFIED in the SELECT) and on MBLNR and MJAHR, which are also high-order primary key fields for MSEG. Furthermore, note that the Execution Plan shows no access of table information per se[4] — all information requested by the SELECT is contained solely within Index 0 itself.

In this sample, I have also included in the WHERE clause the additional high-order index key field MJAHR to illustrate that even though we may be furnishing most of the high-order index information, index scans can still take significant time. On the system on which this sample was run, each set of index scans and data fetches took an average of about .014 seconds, and in one case almost as much as .089 seconds.[5]

If the generated SQL WHERE had not included *any* high-order key fields from some index for the table, we might have ended up executing multiple full-table scans (depending again on the number of rows in the driver table and the complexity of the WHERE clause), with possibly disastrous results from the point of view of performance. It is therefore *extremely* important to know which index(es) your database optimizer has chosen to use when fulfilling your SELECT request.

✓ *Tip*

Supporting material for all program reviews should include Execution Plans for each client-generated SELECT in custom programs. These should be reviewed by senior ABAP programmers and DBAs to ensure that data is being accessed using the most efficient techniques possible.

Although the time to scan the index may be small relative to the time required to access actual table entries, as I have already shown it is nevertheless measurable and can be considerable, even in the case of relatively small tables. As a guiding principle, we should take steps to minimize the number of scans (index and/or table) wherever possible. One way to minimize this time is to make sure that we are not requesting duplicate records, as discussed previously.

Let's stop right here and talk about how result set duplicates are excluded. This process takes place in two places: in the database itself, and in SAP once the data has been transferred to the application server from the database.

Remember that each set of OR'd selection criteria is presented to the database as a separate fetch request. While the database can prevent duplicate information from being returned for a single OR request (as we saw when we sorted the driver table), each request is handled separately and independently. Duplicates can therefore be retrieved for one or both of two reasons:

1. If the driver table includes duplicate entries (as far as the fields used in the SELECT are concerned) then there is the possibility that the database may spend valuable time accessing and transmitting redundant data. The recommendation, therefore, is to *remove all duplicate rows from the driver table before initiating the* SELECT.

[4] To see an Execution Plan for a SELECT where index data is used to access a specific row from the table data itself, refer to Figure 10.

[5] In a separate test on another system, no index fields (other than MANDT) were provided in the WHERE clause, and each fetch took around 2.5 to 3 seconds!

2. Furthermore, if the fields being retrieved are such that duplicate rows in the result set might be generated, and multiple fetch requests are generated by the database interface, then the duplicate information will need to be removed by the application server. There is no automated way to prevent this from occurring — careful scrutiny of an SQL trace, however, should reveal that extra data is being accessed (compare the retrieved record counts in the trace to the actual number of records returned to the program). If this is occurring, then an altered `SELECT` strategy should be considered.

Suppose we've removed the possibility of retrieving redundant information, but an SQL trace shows that we are making many database fetches that are returning little or no usable data in each fetch, as we saw in Figure 2. What further techniques can we employ to make maximum use of the transport buffers between the database and application server?

In the normal case, the number of selection criteria represented by each fetch is equal to the number of `OR`'d requests for data (25, in the case of the first sample program being discussed here). If we can devise a way to increase the number of selection criteria reflected by each fetch request, we might be able to make much more efficient use of transmission buffers. Here's a mechanism to enhance fetch effectiveness you might find useful, plus an astounding related discovery that you may be able to apply to your production jobs to cut their times by factors of 100, 1,000, or even 10,000! Read on…

The sample program is constructed to contrast five different ways of accessing joined data. Using the traditional `FAEI` model, there were 600 fetches, averaging about 57 records per fetch, resulting in about 3 percent utilization of the transmission buffers. The total elapsed time to retrieve all the data was about 9.2 seconds.

Assuming that the number of records returned per fetch is normally distributed, roughly 95 percent of the fetches contained from 43 to 71 records, and all contained less than 111, as opposed to the theoretical maximum of 1,877 each fetch could have accommodated.

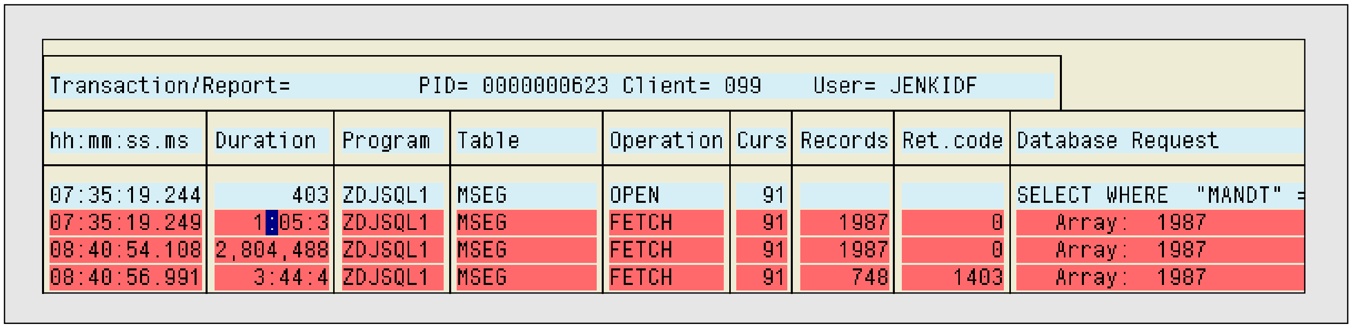
How can we make maximum use of the transport buffers between the database and application server? The generated SQL (shown in Figure 1) for the `SELECT` traced earlier (see Figure 2) revealed that a maximum of 25 `OR`'d values can be specified for each fetch (although more records than that may be returned).

One way that we can expand the number of records requested by each fetch is by using sets of `BETWEEN`s. If a single `BETWEEN`, for instance, represents a request for 1,877 records, such as those being processed by the sample program, then a single fetch request would return almost a full 34K buffer's worth of data. ABAP provides a handy mechanism for processing data in sets of ranges — the `RANGE` or `SELECT-OPTION`. The problem then becomes one of creating a useable `RANGE` and modifying the `SELECT` so that it references that `RANGE` effectively.

In Appendix B, I have included ABAP code for a simple `FORM` routine that can be called to create a range table for a selected field in an internal table. In the case of the sample program, this form can be used to create a range table whose rows reflect all "runs" (first and last number in a range of consecutive numbers) of `MBLNR`s occurring in internal table `INT_DRIVER`. Note that this form will only work properly with character data containing numbers only. Once we have created such a range (`R_MBLNR` in the sample program), we can then issue a `SELECT` specifying `IN R_MBLNR`.

Using this technique, only valid data will be returned with each fetch, and the probability of empty blocks being returned will be greatly reduced.

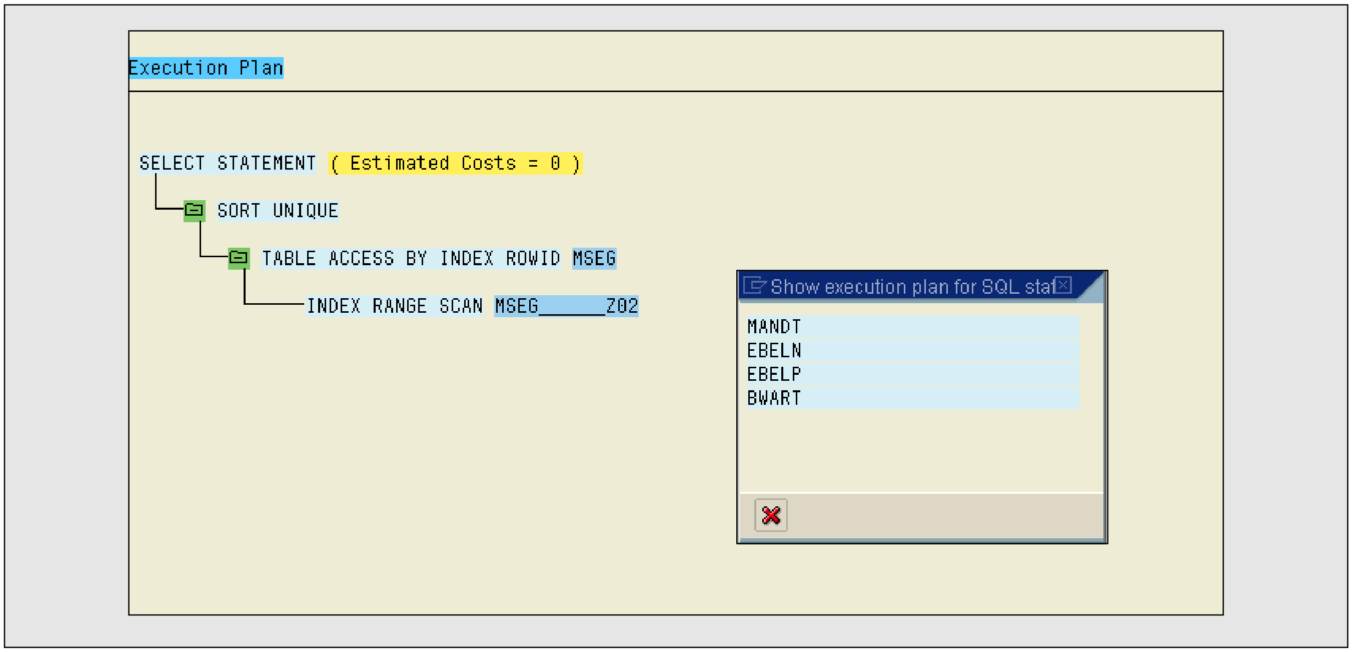
Figure 7 *SQL Trace for SELECT Using Range Table*



Transaction/Report= PID= 0000000623 Client= 099 User= JENKIDF

hh:mm:ss.ms	Duration	Program	Table	Operation	Curs	Records	Ret.code	Database Request
07:35:19.244	403	ZDJSQL1	MSEG	OPEN	91			SELECT WHERE "MANDT" =
07:35:19.249	1:05:3	ZDJSQL1	MSEG	FETCH	91	1987	0	Array: 1987
08:40:54.108	2,804,488	ZDJSQL1	MSEG	FETCH	91	1987	0	Array: 1987
08:40:56.991	3:44:4	ZDJSQL1	MSEG	FETCH	91	748	1403	Array: 1987

Figure 8 *Execution Plan for SELECT Using Range Table*



```
Execution Plan

SELECT STATEMENT ( Estimated Costs = 0 )
 └─ SORT UNIQUE
     └─ TABLE ACCESS BY INDEX ROWID MSEG
         └─INDEX RANGE SCAN MSEG______Z02
```

```
MANDT
EBELN
EBELP
BWART
```

Figure 7 shows the SQL trace for a slightly different program that used this technique to select `MSEG` data based solely on document numbers (`MBLNR`). Bear in mind that this case was run using Oracle rule-based optimization and SAP Release 3.1H.

The total time required for getting the data using the range table was almost five hours![6]

Even though we gained superb utilization of the transmission buffers (1,987 records fetched versus 1,987 possible in two of three fetches), we failed miserably to improve our `SELECT` performance.

How could that be? A closer look at the Execution Plan for this case (**Figure 8**) reveals the answer, and may show you a way to decrease some of your production runtimes by huge factors.

Note that the primary index (`MSEG_______0`) was not used for the index scans, even though the

[6] Since this example accessed `MSEG` data purely on `MBLNR`, to provide a strictly apples-to-apples comparison to previous cases, we would have to add to this time the amount of time it took to delete records for incorrect fiscal year (`GJAHR`) and to read all 15,000 of the records available.

Figure 9 Execution Time Table for the “Key Value Range (1)” SELECT Scenario

SELECT Strategy	Total Fetches	Total Time (seconds)	Time per Fetch (ms)	Record Count	Time per Record (µs)	Records per Database Fetch			
						mean	s.d.	min	max
Baseline	600	9.173	15.3	34,228	268	57.0	7.1	29	110
Sorted	600	7.982*	13.3	28,572	279	47.6	17.3	13	92
Dupes removed	500	6.955	13.9	28,342	245	56.7	13.9	25	92
Key Value range (1)	3	4h 49m	1h 36m	4,722	3.67 sec	1,574	—	748	1,987
Key Value range (2)									

* This time does *not* include the time required to sort the data prior to selection. However, our experience has shown that ABAP is a very efficient sorter, and that this time can be safely ignored for comparison purposes here.

selection criteria called out a range on field MBLNR, the high-order key field of MSEG. Instead, the Oracle optimizer has decided on index MSEG___Z02, which contains no fields that help us (other than MANDT). This has happened because the WHERE MANDT = <*client*> (supplied by the interface) apparently was not distributed across all of the OR’d WHEREs by the Oracle rule-based optimizer, and the optimizer therefore chose an inefficient (for us, at least) index.

When this occurs, the effect on job performance may be dramatic — a SELECT for two subranges may take thousands of times longer than twice the time for two SELECTs, each using one subrange![7]

In my experience, this appears to happen any time a RANGE or SELECT-OPTION contains two or more rows with sign/option = “IBT”.[8]

Figure 9 is an updated table showing our progress so far, reflecting the degraded performance of the 3.1H test case.

If index usage is the problem, is there a way around it?

There are at least two ways to deal with this problem — *alternative use of a range* and *hints*.

Let’s look at the **alternative use of a range** first. In this approach, which is demonstrated in the sample program, the generated range table is used in a somewhat non-traditional manner, and is based on FAEI usage.

Here is the rewritten SELECT:

[7] If you’re on a 3.1x or 4.6x system and using Oracle rule-based optimization, here’s an experiment you can try at home. Pick a table with a relatively large number of rows. In our case, we used MSEG, which has about 17 million records. Use SE16 to display some of those records, and then pick two short sequences of the high-order key field (exclusive of MANDT, if the table is client-dependent). That is, compose two ranges of high-order keys that will return at most one or two records each. Then use SE16 to display the records for *only the first* of those two sequences, using SE16. In our MSEG test case, it took about 1 second to access the two records requested. Now add a second row to the select-option criteria to include the second sequence and rerun the SE16 query. In our case, the additional time to fetch the second set of two records took an amazing 3 hours and 57 minutes — a decrease in performance on the order of 14,000!

[8] This phenomena also exists in 4.6x systems, when Oracle is forced to use rule-based optimization. However, under cost-based optimization a more appropriate index is chosen and the system performs as expected.

Figure 10 Execution Time Table for the "Key Value Range (2)" SELECT Scenario

SELECT Strategy	Total Fetches	Total Time (seconds)	Time per Fetch (ms)	Record Count	Time per Record (µs)	Records per Database Fetch			
						mean	s.d.	min	max
Baseline	600	9.173	15.3	34,228	268	57.0	7.1	29	110
Sorted	600	7.982*	13.3	28,572	279	47.6	17.3	13	92
Dupes removed	500	6.955	13.9	28,342	245	56.7	13.9	25	92
Key Value range (1)	3	4h 49m	1h 36m	4,722	3.67 sec	1,574	—	748	1,987
Key Value range (2)	38	4.249	112	28,342	150	745.8	681.4	66	1,987

* This time does *not* include the time required to sort the data prior to selection. However, our experience has shown that ABAP is a very efficient sorter, and that this time can be safely ignored for comparison purposes here.

```
SELECT DISTINCT MBLNR
                MJAHR
                ZEILE
FROM MSEG
INTO CORRESPONDING FIELDS OF
TABLE INT_MSEG
FOR ALL ENTRIES IN R_MBLNR
WHERE MBLNR >= R_MBLNR-LOW
AND   MBLNR <= R_MBLNR-HIGH
.
.
.
```

From an elapsed time standpoint, the results are encouraging: a total of only 4.2 seconds, which represents an almost 2.5 increase in performance over the original case.

We can now update our table to show the results of this final strategy (**Figure 10**).

There are some caveats to this last method, however:

- The type of data in the driver table fields being used should be amenable to conversion to a range — document numbers, for instance, fall into this category.
- Hopefully, the set of values in the driver table will collapse into very few "runs" — in a perfect world the driver table data would result in a single range. In the real world, however, things don't always work out that way — the point here is that you don't want to end up with so many individual subranges ("rangelets"?) that you end up with as many rows in the range table as you had in the driver table.
- Note that using `FOR ALL ENTRIES IN` against the range table exposed us once again to the problem of inefficient buffer utilization — certainly not as bad as when we specified the `INT_BUFFER` table as the driver table, but certainly not as good as when we used the `R_MBLNR` table as a `RANGE` or `SELECT-OPTION`.

Now let's turn our attention to **hints**. Is there any way to get the best of both worlds — the efficient buffer utilization of the `RANGE` used in a `WHERE … IN`, coupled with an acceptable index usage? It may be that the judicious use of *database interface hints* could have been used to entice the Oracle rule-based optimizer into using an appropriate index, but that subject is outside the scope of this article, and I leave the exploration of that possibility as an exercise for the reader.

> ***✓ Tip***
>
> *Hints can be used to influence the decisions made by the Oracle query optimizer. An Index hint can be used to force the use of a particular index for the specified table. Heretofore, hints could only be utilized by coding in SQL using EXEC SQL, but recent versions of SAP now allow the use of a %_HINTS clause in OPEN SQL SELECTs. OSS note 129385 contains information that can help you get started using hints.*

Conclusion

With all the foregoing as a foundation, we are in a position to make a set of general recommendations on the use of `SELECT … FOR ALL ENTRIES IN`:

✓ The driver table should not be empty — if it is, *all* records in the database table will be accessed (constrained, of course, by the other elements in the `WHERE` clause).

✓ The benefits of using `UP TO x ROWS` in conjunction with `FAEI` may be overshadowed by the extra time required to transport unneeded records.

✓ The driver table should contain no duplicates (as it pertains to fields being used in the `SELECT`'s `WHERE` clause) — if it exists in the database, a record will be fetched from the table for each row in the driver table, regardless of redundancy.

✓ The driver table should be sorted by the fields that will be used in index range scans when the `SELECT` for detail data is executed.

✓ If the `SELECT` statement `WHERE` clause does not contain high-order fields from some index, each generated SQL statement may cause a full scan of the database table. In this case, it may be much more efficient to do a single `SELECT/ENDSELECT` on the table and use a `CHECK` to filter out the unwanted rows.

✓ Contrary to what we'd like to believe, it can take considerable time to scan an index. Depending on the index fields you have available, you may find it cheaper (in terms of time) to forgo multiple index scans in favor of a single full-table scan.

✓ Use debug and SQL trace to make sure that you have a good understanding of the data you'll be accessing using `FAEI`. This includes the entries you expect to see in the driver table (counts, distributions, data types, etc.) and the database table characteristics (row counts and available indexes, etc.)

✓ If the number of entries in the driver table is high, but the expectation of finding entries in the database table is low, then you run the risk of making many empty fetches from the database — that is, fetches containing no records. Each of these fetches costs, in terms of time, as much as one where the block is fully populated with data records. Therefore, you should attempt to find some way to enhance the blocking of valid data into blocks returned from the database.

In summary, the use of `FOR ALL ENTRIES IN` can ease your programming load, but can cause immense performance problems if not wisely used. Before you place a program into production, make sure that you've fully evaluated the effect of this coding technique on your program's performance, and have evaluated alternative methods such as `VIEW`s and `JOIN`s to accomplish the same results. Each of these other methods has its own advantages and disadvantages, and you should understand these well before implementing your particular application solution.

David Jenkins joined RCG Information Technology in 1995 after 29 years as a Consulting Systems Representative at IBM, and 4 years in the Ph.D. program at the University of Houston. In the fall of 2000, David was named a Director Consultant at RCG, where he develops and teaches classes in beginning and advanced ABAP programming, ABAP performance and tuning, and Java, and consults with clients on all aspects of ABAP development.

David Jenkins has been involved with Data Processing since 1957, when he entered the business as a punchcard machine operator. Dave worked for a Houston bank for 10 years before receiving his B.S. in Math from the University of Houston. He then joined IBM, where he worked in various marketing support positions, supporting contractors at the NASA Johnson Space Center. Since leaving IBM, Dave has received a Master's in Management, Computing and Systems from Houston Baptist University, and has finished coursework for a Ph.D. in Management Information Systems at the University of Houston. For the last six years, Dave has been a Consultant for RCG Information Technology, specializing in ABAP development — his latest assignment has been at Texaco, supporting their installation of SAP IS-Oil Upstream.

Dave is married with four children, and one beautiful granddaughter. He and his wife Joy live in the country 80 miles from Houston, where they enjoy the wide-open spaces and fresh air, and their horse, dog, cats, rabbits, and fish. Dave can be reached at djenkins@pointecom.net.

About RCG Information Technology

*RCG Information Technology (**www.rcgit.com**) provides ERP practices utilizing strategic alliances with leading providers such as SAP to bring in-depth knowledge and experience to client customers, employees, partners, and suppliers. RCG IT combines seven essential practices: CRM, Data Warehousing, Application Management, e-Solutions, ERP, Project Management, and Discovery. Its 1,600 consultants average over eight years of experience and are trained in e-business strategy and design, application development and integration, and management. RCG IT serves over 500 customers in a broad range of markets, including financial services, energy, health care, pharmaceutical, telecommunications, government, manufacturing, distribution, and retail. RCG's project management methodology is certified company-wide at Level 2 of the Capability Maturity Model® of the Software Engineering Institute. For further information, call 800-350-9770.*

Appendix A — Sample Program

This appendix accompanies the article “Learn How to Avoid Lackluster System Performance When Using the ‘FOR ALL ENTRIES IN’ Clause: An ABAP Developer’s Guide” and contains the sample program used to generate most of the results described in the article. This sample program contains references to the `FORM` routines `F_START_SQL_TRACE` and `F_END_SQL_TRACE`. These two routines consist primarily of calls to transaction ST05 via `CALL TRANSACTION`. While they are not included in this appendix due to space constraints, these routines, in addition to the sample program, are available for download at **www.SAPpro.com**, or from the author at djenkins@pointecom.net.

```
REPORT ZDJSQL1 MESSAGE-ID ZV.

*----------------------------------------------------------------------*
* PROGRAM:    zdjsql1                         DATE WRITTEN: 11/1/2000  *
*                                                                      *
* AUTHOR(s):  David F. Jenkins                LAST UPDATE:             *
*             RCG Information Technology                               *
*                                                                      *
* PROGRAM TITLE: Using FOR ALL ENTRIES IN in SELECTs:                  *
*                Comparisons of the effects of various techniques      *
*                on overall performance                                *
*                                                                      *
* DESCRIPTION: This program demonstrates the usage of the SELECT...    *
*              FOR ALL ENTRIES IN construct, and presents various test *
*              cases to help in the evaluation of alternative usages   *
*              of that construct.                                      *
*                                                                      *
*----------------------------------------------------------------------*

TABLES:
   mkpf,
   mseg.

************************************************************************
*                                                                      *
*  The following parameters were included so that separate SELECT      *
*  strategies could be evaluated independently - especially important  *
*  since SQL trace table storage allocated was insufficient to hold    *
*  trace data for all five SELECTs at one time.                        *
*                                                                      *
************************************************************************
```

```
SELECTION-SCREEN BEGIN OF BLOCK DRIVER
                   WITH FRAME
                     TITLE TEXT-DRV.
Parameters:

* maximum number of driver records to use
   P_MAXCT     LIKE DATATYPE-INTEGER4 DEFAULT 15000,

* percent of driver records that are duplicates
   P_DUPES     LIKE DATATYPE-INTEGER4 DEFAULT 20.

SELECTION-SCREEN END OF BLOCK DRIVER.


SELECTION-SCREEN BEGIN OF BLOCK PROC
                   WITH FRAME
                     TITLE TEXT-PRC.
SELECTION-SCREEN SKIP.

Parameters:
* maximum number of keys to be retrieved with one fetch
CB_PRE      AS CHECKBOX DEFAULT ' ',
CB_DBFLD    AS CHECKBOX DEFAULT ' '.

SELECTION-SCREEN SKIP.

SELECTION-SCREEN BEGIN OF BLOCK BASE
                   WITH FRAME
                     TITLE TEXT-BAS.
Parameters:
* execute initial baseline select
   CB_INIT  AS CHECKBOX DEFAULT 'X',
   CB_TRBAS AS CHECKBOX DEFAULT 'X'.
SELECTION-SCREEN END OF BLOCK BASE.

SELECTION-SCREEN BEGIN OF BLOCK SORT
                   WITH FRAME
                     TITLE TEXT-SRT.
Parameters:
* execute after sort of driver table
   CB_SORT  AS CHECKBOX DEFAULT 'X',
   CB_TRSRT AS CHECKBOX DEFAULT 'X'.
SELECTION-SCREEN END OF BLOCK SORT.

SELECTION-SCREEN BEGIN OF BLOCK DUPE
                   WITH FRAME
                     TITLE TEXT-DUP.
Parameters:
* execute after dupes removed
```

```
   CB_DUPE  AS CHECKBOX DEFAULT 'X',
   CB_TRDUP AS CHECKBOX DEFAULT 'X'.
SELECTION-SCREEN END OF BLOCK DUPE.

SELECTION-SCREEN BEGIN OF BLOCK RNG1
                   WITH FRAME
                     TITLE TEXT-RG1.
Parameters:
* execute with normal range table use
   CB_RNG1  AS CHECKBOX DEFAULT 'X',
   CB_TRRG1 AS CHECKBOX DEFAULT 'X'.
SELECTION-SCREEN END OF BLOCK RNG1.

SELECTION-SCREEN BEGIN OF BLOCK RNG2
                   WITH FRAME
                     TITLE TEXT-RG2.
Parameters:
* execute with modified range table use
   CB_RNG2  AS CHECKBOX DEFAULT 'X',
   CB_TRRG2 AS CHECKBOX DEFAULT 'X'.
SELECTION-SCREEN END OF BLOCK RNG2.

SELECTION-SCREEN END OF BLOCK PROC.

* generated range table for requested MBLNRs
ranges:
   r_mblnr for mseg-mblnr.

* global data
DATA:

* used to compute elapsed times in second resolution
  Z_ELAPSED_TIME    LIKE SY-UZEIT,
  Z_END_TIME        LIKE SY-UZEIT,
  Z_START_TIME      LIKE SY-UZEIT,

* table containing names of mseg fields to be selected
  INT_FIELDS(72)  OCCURS 0 WITH HEADER LINE,

* target table for SELECTed data
  INT_mseg LIKE mseg OCCURS 0 WITH HEADER LINE,

* driver table
  BEGIN OF INT_DRIVER OCCURS 0,
     mjahr LIKE mkpf-mjahr,
     mblnr LIKE mkpf-mblnr,
     RAND  LIKE DATATYPE-INTEGER4,     " random integer
  END OF INT_DRIVER.
```

```
************************************************************************
*  end-of-selection                                                    *
************************************************************************
END-OF-SELECTION.

* initial load of driver table with requested number of entries
  perform f_fill_initial_data using p_maxct
                                    p_dupes.

* initialize table with names of mseg fields to be selected
  PERFORM F_SETUP_FIELDS.

  PERFORM F_BASELINE_SELECT.

  PERFORM F_SORTED_SELECT.

  PERFORM F_DUPE_REMOVED_SELECT.

  PERFORM F_RANGE_TABLE_VERSION_1_SELECT.

  PERFORM F_RANGE_TABLE_VERSION_2_SELECT.

* form for building a range from a table of values
  include zbldrang.

* forms for SQL tracing: start, end, and display
  INCLUDE ZTRACING.
*&---------------------------------------------------------------------*
*&      Form  F_FILL_INITIAL_DATA
*&---------------------------------------------------------------------*
*      —>P_P_MAXCT  text
*      —>P_P_DUPES  text
*----------------------------------------------------------------------*
FORM F_FILL_INITIAL_DATA USING    P_P_MAXCT  LIKE DATATYPE-INTEGER4
                                  P_P_DUPES  LIKE DATATYPE-INTEGER4.

  data:
    z_norm_count type i,
    z_dupe_count type i.

  if p_p_dupes <> 0.

    z_norm_count = p_p_maxct / ( '1.0' + p_p_dupes / '100.0' ).
    z_dupe_count = p_p_maxct - z_norm_count.

  else.

    z_norm_count = p_p_maxct.
    z_dupe_count = 0.

  endif.

  SELECT mjahr
         MBLNR
     FROM MKPF UP TO Z_NORM_COUNT ROWS
     INTO CORRESPONDING FIELDS OF TABLE INT_DRIVER
     WHERE MJAHR >= '1999'.

  if z_dupe_count > 0.

    SELECT
             mjahr
             mblnr
           FROM MKPF UP TO Z_DUPE_COUNT ROWS
           APPENDING CORRESPONDING FIELDS OF TABLE INT_DRIVER
           WHERE MJAHR >= '1999'.

  endif.

* randomize contents of driver table
  loop at int_driver.
    call function 'RANDOM_I4'
       exporting
           rnd_max = p_p_maxct
       importing
           rnd_value = int_driver-rand.
    modify int_driver.
  endloop.

  sort int_driver by rand.
  COMMIT WORK.

ENDFORM.                               " F_FILL_INITIAL_DATA

*&---------------------------------------------------------------------*
*&      Form  F_BASELINE_SELECT
*&---------------------------------------------------------------------*
FORM F_BASELINE_SELECT.

  DATA:
    INT_LOC_DRIVER LIKE INT_DRIVER OCCURS 0.

  IF CB_INIT = 'X'.

    INT_LOC_DRIVER[] = INT_DRIVER[].
    FREE INT_MSEG.
```

```
* issue SELECT of interest: pseudo-random, with dupes

* demonstrate effect of UP TO 2 ROWS on SELECT time and establish
* base timing case

* run a "pre-load" run to load buffers?
    IF CB_PRE = 'X'.
      SELECT (INT_FIELDS)
         FROM mseg UP TO 2 ROWS
         INTO CORRESPONDING FIELDS OF TABLE INT_mseg
         FOR ALL ENTRIES IN INT_LOC_DRIVER
         WHERE MBLNR = INT_LOC_DRIVER-MBLNR
         AND   MJAHR = INT_LOC_DRIVER-MJAHR.
    ENDIF.

    IF CB_TRBAS = 'X'.
      PERFORM F_START_SQL_TRACE.
    ENDIF.

    GET TIME.
    Z_START_TIME = SY-UZEIT.
    SELECT (INT_FIELDS)
     FROM mseg UP TO 2 ROWS
     INTO CORRESPONDING FIELDS OF TABLE INT_mseg
     FOR ALL ENTRIES IN INT_LOC_DRIVER
     WHERE MBLNR = INT_LOC_DRIVER-MBLNR
     AND   MJAHR = INT_LOC_DRIVER-MJAHR.
    GET TIME.
    Z_END_TIME = SY-UZEIT.
    Z_ELAPSED_TIME = Z_END_TIME - Z_START_TIME.
    WRITE: / 'Base elapsed time:        ', Z_ELAPSED_TIME.

    IF CB_TRBAS = 'X'.
      PERFORM F_END_SQL_TRACE.
    ENDIF.

  ENDIF.
  COMMIT WORK.

ENDFORM.                               " F_BASELINE_SELECT

*&----------------------------------------------------------------------*
*&      Form  F_SORTED_SELECT
*&----------------------------------------------------------------------*
*&----------------------------------------------------------------------*
*  —>  p1        text
*  <—  p2        text
*-----------------------------------------------------------------------*
FORM F_SORTED_SELECT.
```

```
  IF CB_SORT = 'X'.

    DATA:
      INT_LOC_DRIVER LIKE INT_DRIVER OCCURS 0.

    FREE INT_MSEG.
    INT_LOC_DRIVER[] = INT_DRIVER[].

* issue SELECT of interest: sorted, but still with dupes
    SORT INT_LOC_DRIVER BY MBLNR
                           MJAHR.

* run a "pre-load" run to load buffers?
    IF CB_PRE = 'X'.
      SELECT (INT_FIELDS)
         FROM MSEG
         INTO CORRESPONDING FIELDS OF TABLE INT_mseg
         FOR ALL ENTRIES IN INT_LOC_DRIVER
         WHERE MBLNR = INT_LOC_DRIVER-MBLNR
         AND   MJAHR = INT_LOC_DRIVER-MJAHR.
    ENDIF.

    IF CB_TRSRT = 'X'.
      PERFORM F_START_SQL_TRACE.
    ENDIF.

    GET TIME.
    Z_START_TIME = SY-UZEIT.
    SELECT (INT_FIELDS)
     FROM MSEG
     INTO CORRESPONDING FIELDS OF TABLE INT_mseg
     FOR ALL ENTRIES IN INT_LOC_DRIVER
     WHERE MBLNR = INT_LOC_DRIVER-MBLNR
     AND   MJAHR = INT_LOC_DRIVER-MJAHR.
    GET TIME.
    Z_END_TIME = SY-UZEIT.
    Z_ELAPSED_TIME = Z_END_TIME - Z_START_TIME.
    WRITE: / 'Sorted elapsed time:      ', Z_ELAPSED_TIME.

    IF CB_TRSRT = 'X'.
      PERFORM F_END_SQL_TRACE.
    ENDIF.

  ENDIF.
  COMMIT WORK.

ENDFORM.                               " F_SORTED_SELECT
```

```
*&---------------------------------------------------------------------*
*&      Form  F_DUPE_REMOVED_SELECT
*&---------------------------------------------------------------------*
FORM F_DUPE_REMOVED_SELECT.

  DATA:
    INT_LOC_DRIVER LIKE INT_DRIVER OCCURS 0.

  IF CB_DUPE = 'X'.

    FREE INT_MSEG.
    INT_LOC_DRIVER[] = INT_DRIVER[].

* issue SELECT of interest: sorted and with dupes eliminated
    SORT INT_LOC_DRIVER BY MBLNR
                           MJAHR.
    DELETE ADJACENT DUPLICATES FROM INT_LOC_DRIVER COMPARING MBLNR
                                                             MJAHR.

* run a "pre-load" run to load buffers?
    IF CB_PRE = 'X'.
      SELECT (INT_FIELDS)
         FROM MSEG
         INTO CORRESPONDING FIELDS OF TABLE INT_mseg
         FOR ALL ENTRIES IN INT_LOC_DRIVER
         WHERE MBLNR = INT_LOC_DRIVER-MBLNR
         AND   MJAHR = INT_LOC_DRIVER-MJAHR.
    ENDIF.

    IF CB_TRDUP = 'X'.
      PERFORM F_START_SQL_TRACE.
    ENDIF.

    GET TIME.
    Z_START_TIME = SY-UZEIT.
    SELECT (INT_FIELDS)
     FROM MSEG
     INTO CORRESPONDING FIELDS OF TABLE INT_mseg
     FOR ALL ENTRIES IN INT_LOC_DRIVER
     WHERE MBLNR = INT_LOC_DRIVER-MBLNR
     AND   MJAHR = INT_LOC_DRIVER-MJAHR.
    GET TIME.
    Z_END_TIME = SY-UZEIT.
    Z_ELAPSED_TIME = Z_END_TIME - Z_START_TIME.
    WRITE: / 'Dupe elapsed time:        ', Z_ELAPSED_TIME.

    IF CB_TRDUP = 'X'.
      PERFORM F_END_SQL_TRACE.
    ENDIF.
  ENDIF.
  COMMIT WORK.

ENDFORM.                               " F_DUPE_REMOVED_SELECT
*&---------------------------------------------------------------------*
*&      Form  F_RANGE_TABLE_VERSION_1_SELECT
*&---------------------------------------------------------------------*
FORM F_RANGE_TABLE_VERSION_1_SELECT.

  DATA:
    INT_LOC_DRIVER LIKE INT_DRIVER OCCURS 0,
    Z_COUNT        TYPE I VALUE 0,
    Z_DELETE       TYPE I.

  IF CB_RNG1 = 'X'.

* build a range table reflecting all the MBLNRs in the driver table
    INT_LOC_DRIVER[] = INT_DRIVER[].
    SORT INT_LOC_DRIVER BY MBLNR
                           MJAHR.
    DELETE ADJACENT DUPLICATES FROM INT_LOC_DRIVER COMPARING MBLNR
                                                             MJAHR.
    PERFORM F_BUILD_RANGE TABLES R_MBLNR
                                 INT_LOC_DRIVER
                           USING 'MBLNR'.

    FREE INT_MSEG.

* issue SELECT of interest using IN RANGE

* run a "pre-load" run to load buffers?
    IF CB_PRE = 'X'.
      SELECT (INT_FIELDS)
         FROM MSEG
         INTO CORRESPONDING FIELDS OF TABLE INT_MSEG
            WHERE MBLNR IN R_MBLNR.
    ENDIF.

    IF CB_TRRG1 = 'X'.
      PERFORM F_START_SQL_TRACE.
    ENDIF.

    GET TIME.
    Z_START_TIME = SY-UZEIT.
    SELECT (INT_FIELDS)
     FROM MSEG
     INTO CORRESPONDING FIELDS OF TABLE INT_MSEG
        WHERE MBLNR IN R_MBLNR.
    GET TIME.
    Z_END_TIME = SY-UZEIT.
    Z_ELAPSED_TIME = Z_END_TIME - Z_START_TIME.
    WRITE: / 'Range (version 1):        ', Z_ELAPSED_TIME.

    IF CB_TRRG1 = 'X'.
      PERFORM F_END_SQL_TRACE.
    ENDIF.

  ENDIF.
  COMMIT WORK.

ENDFORM.                               " F_RANGE_TABLE_VERSION_1_SELECT

*&---------------------------------------------------------------------*
*&      Form  F_RANGE_TABLE_VERSION_2_SELECT
*&---------------------------------------------------------------------*
FORM F_RANGE_TABLE_VERSION_2_SELECT.

  DATA:
    INT_LOC_DRIVER LIKE INT_DRIVER OCCURS 0.

  IF CB_RNG2 = 'X'.

* build a range table reflecting all the MBLNRs in the driver table
    INT_LOC_DRIVER[] = INT_DRIVER[].
    SORT INT_LOC_DRIVER BY MBLNR
                           MJAHR.
    DELETE ADJACENT DUPLICATES FROM INT_LOC_DRIVER COMPARING MBLNR
                                                             MJAHR.
    PERFORM F_BUILD_RANGE TABLES R_MBLNR
                                 INT_LOC_DRIVER
                           USING 'MBLNR'.
    FREE INT_MSEG.

* issue SELECT of interest using FOR ALL ENTRIES IN the RANGE

* run a "pre-load" run to load buffers?
    IF CB_PRE = 'X'.
      SELECT (INT_FIELDS)
         FROM MSEG
         INTO CORRESPONDING FIELDS OF TABLE INT_MSEG
         FOR ALL ENTRIES IN R_MBLNR
            WHERE MBLNR >= R_MBLNR-LOW
            AND   MBLNR <= R_MBLNR-HIGH.
    ENDIF.

    IF CB_TRRG2 = 'X'.
      PERFORM F_START_SQL_TRACE.
    ENDIF.

    GET TIME.
    Z_START_TIME = SY-UZEIT.
    SELECT (INT_FIELDS)
     FROM MSEG
     INTO CORRESPONDING FIELDS OF TABLE INT_MSEG
     FOR ALL ENTRIES IN R_MBLNR
        WHERE MBLNR >= R_MBLNR-LOW
        AND   MBLNR <= R_MBLNR-HIGH.
    GET TIME.
    Z_END_TIME = SY-UZEIT.
    Z_ELAPSED_TIME = Z_END_TIME - Z_START_TIME.
    WRITE: / 'Range (version 2):        ', Z_ELAPSED_TIME.

    IF CB_TRRG2 = 'X'.
      PERFORM F_END_SQL_TRACE.
    ENDIF.

  ENDIF.
  COMMIT WORK.

ENDFORM.                               " F_RANGE_TABLE_VERSION_2_SELECT

*&----------------------------------------------------------------------*
*&      Form  F_SETUP_FIELDS
*&----------------------------------------------------------------------*
*       text
*-----------------------------------------------------------------------*
*  —>  p1        text
*  <—  p2        text
*-----------------------------------------------------------------------*
FORM F_SETUP_FIELDS.

  APPEND 'MBLNR' TO INT_FIELDS.
  APPEND 'MJAHR' TO INT_FIELDS.
  APPEND 'ZEILE' TO INT_FIELDS.
  IF CB_DBFLD = 'X'.
    APPEND 'BWART' TO INT_FIELDS.
  ENDIF.

ENDFORM.                               " F_SETUP_FIELDS
```

Appendix B — ABAP Code for the “F_BUILD_RANGE” Form

This appendix accompanies the article “Learn How to Avoid Lackluster System Performance When Using the ‘FOR ALL ENTRIES IN’ Clause: An ABAP Developer’s Guide” and contains the ABAP code for a simple FORM routine that can be called to create a range table for a selected field in an internal table. In the case of the sample program (Appendix A), this form can be used to create a range table whose rows reflect all “runs” (first and last number in a range of consecutive numbers) of MBLNRs occurring in internal table INT_DRIVER. Note that this form will only work properly with character data containing numbers only. This code is also available for download at **www.SAPpro.com**, or from the author at djenkins@pointecom.net.

```
FORM F_BUILD_RANGE TABLES   P_RANGE
                            P_DATA
                   USING    P_FIELD_NAME TYPE C.  "name of field

* NOTE: This routine assumes that p_data has a header line.  It also
*       sorts the data table by the key for which the table is to be
*       built, and removes duplicates (using the specified field to
*       identify duplicates).
*
*       Note also that the header is not preserved across this
*       subroutine.

* NOTE: This routine will create some entries with sign and option =
*       'IEQ' and with both high and low set to the same value.
*       Although setting the high value may appear redundant, this
*       ploy will allow the resulting range table to be used in a
*       SELECT...FOR ALL ENTRIES IN... referencing the range table
*       as a driver table.

* Note that the maximum length of the key field is 128.
  DATA:
    Z_CHECK(128),
    Z_PREV(128),
    Z_TEST(128).

  DATA:
      Z_LENGTH  TYPE I.
  FIELD-SYMBOLS:
    <FS_CHECK>,
    <FS_PREV>,
    <FS_TEST>,
    <FS_KEY>,
    <FS_R_LOW>,
    <FS_R_HIGH>,
    <FS_R_SIGN>,
    <FS_R_OPT>.

  SORT P_DATA BY (P_FIELD_NAME).
  DELETE ADJACENT DUPLICATES FROM P_DATA COMPARING (P_FIELD_NAME).

* This form assumes that the field to be used in building the range
* table is a character (or numeric field containing numeric-only data).
  ASSIGN COMPONENT P_FIELD_NAME OF STRUCTURE P_DATA TO <FS_KEY>.
  IF SY-SUBRC <> 0.
    EXIT.
  ELSE.
    DESCRIBE FIELD <FS_KEY> LENGTH Z_LENGTH.
    ASSIGN Z_CHECK(Z_LENGTH) TO <FS_CHECK>.
    ASSIGN Z_TEST(Z_LENGTH) TO <FS_TEST>.
    ASSIGN Z_PREV(Z_LENGTH) TO <FS_PREV>.

    ASSIGN COMPONENT 'LOW' OF STRUCTURE P_RANGE TO <FS_R_LOW>.
    IF SY-SUBRC <> 0.
       EXIT.
    ENDIF.

    ASSIGN COMPONENT 'HIGH' OF STRUCTURE P_RANGE TO <FS_R_HIGH>.
    IF SY-SUBRC <> 0.
       EXIT.
    ENDIF.

    ASSIGN COMPONENT 'SIGN' OF STRUCTURE P_RANGE TO <FS_R_SIGN>.
    IF SY-SUBRC <> 0.
       EXIT.
    ENDIF.

    ASSIGN COMPONENT 'OPTION' OF STRUCTURE P_RANGE TO <FS_R_OPT>.
    IF SY-SUBRC <> 0.
       EXIT.
    ENDIF.

    CLEAR P_RANGE.
    FREE  P_RANGE.
    <FS_R_SIGN> = 'I'.
    <FS_R_OPT> = 'BT'.
    LOOP AT P_DATA.
      IF SY-TABIX = 1.
        <FS_R_LOW> = <FS_R_HIGH> = <FS_KEY>.
      ELSE.
        IF NOT <FS_PREV> CO '0123456789 '.
          APPEND P_RANGE.
          IF <FS_R_LOW> = <FS_R_HIGH>.
            <FS_R_OPT> = 'EQ'.
          ELSE.
            <FS_R_OPT> = 'BT'.
          ENDIF.
          <FS_R_LOW> = <FS_R_HIGH> = <FS_KEY>.
        ELSE.
          <FS_CHECK> = <FS_PREV> + 1.
          SHIFT <FS_CHECK> LEFT DELETING LEADING '0'.
          SHIFT <FS_CHECK> LEFT DELETING LEADING ' '.
          <FS_TEST> = <FS_KEY>.
          SHIFT <FS_TEST> LEFT DELETING LEADING '0'.
          SHIFT <FS_TEST> LEFT DELETING LEADING ' '.
          IF <FS_TEST> = <FS_CHECK>.
            <FS_R_HIGH> = <FS_KEY>.
          ELSE.
            IF <FS_R_LOW> = <FS_R_HIGH>.
              <FS_R_OPT> = 'EQ'.
            ELSE.
              <FS_R_OPT> = 'BT'.
            ENDIF.
            APPEND P_RANGE.
            <FS_R_LOW> = <FS_R_HIGH> = <FS_KEY>.
          ENDIF.
        ENDIF.
      ENDIF.
      <FS_PREV> = <FS_KEY>.
    ENDLOOP.

    IF <FS_R_LOW> = <FS_R_HIGH>.
      <FS_R_OPT> = 'EQ'.
    ELSE.
      <FS_R_OPT> = 'BT'.
    ENDIF.
    APPEND P_RANGE.

  ENDIF.                               " field exist in input table?

ENDFORM.                               " F_BUILD_RANGE
```