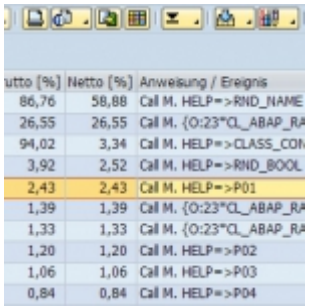


utto [%]	Netto [%]	Anweisung / Ereignis
86,76	58,88	Call M. HELP=>RND_NAME
26,55	26,55	Call M. {O:23*CL_ABAP_RA
94,02	3,34	Call M. HELP=>CLASS_CON
3,92	2,52	Call M. HELP=>RND_BOOL
2,43	2,43	Call M. HELP=>P01
1,39	1,39	Call M. {O:23*CL_ABAP_RA
1,33	1,33	Call M. {O:23*CL_ABAP_RA
1,20	1,20	Call M. HELP=>P02
1,06	1,06	Call M. HELP=>P03
0,84	0,84	Call M. HELP=>P04

ABAP 740-Features unter der Lupe

Aus einer einfachen Anfängerfrage im abapforum.com hat sich eine recht spannende Antwortserie entwickelt, die auf die neuen Sprachfeatures von ABAP740 eingeht. Ich habe diese einmal zusammen gefasst und auch Laufzeitmessungen durchgeführt.

Die Frage

Die Frage von debianfan lautete: Wie ermittele ich die Anzahl von Datensätzen bestimmter Ausprägung in einer internen Tabelle?

Die interne Tabelle NAMES besteht nur aus den Feldern

- NAME (string)
- TF (boolean)

Die folgenden Lösungen sind teilweise vereinfacht und ohne DATA-Definitionen. Die einzelnen lauffähigen Lösungen sind unten im Beispielprogramm ersichtlich.

Lösung 1 – 2xLOOP+WHERE(DATA)

Die einfachste und auf der Hand liegende Antwort von Tron war:

```
LOOP AT names INTO name WHERE tf = abap_true.
  ADD 1 TO zaehler_true.
ENDLOOP.

LOOP AT names INTO name WHERE tf = abap_false.
  ADD 1 TO zaehler_false.
ENDLOOP.
```

Die Lösung ist einfach und verständlich.

Der Einwand von Ralf war, dass bei WHERE die gesamte Tabelle durchlaufen werden muss, wenn kein Index verwendet wird. Das kann sich bei großen Tabellen negativ auf die Laufzeit auswirken.

Mein Gedanke war, dass ich zwei LOOPs nicht schön finde und außerdem ein LOOP mit einer Case-Anweisung noch einen Tacken einfacher und deutlich sein müsste. Dazu später mehr.

Lösung 2 – FILTER

Haubi hat dann den Vorschlag gemacht, die einzelnen Einträge mittels FILTER zu zählen:

```
DATA(lv_true)  = lines( FILTER #( names WHERE tf = abap_true ) ).
DATA(lv_false) = lines( FILTER #( names WHERE tf = abap_false ) ).
```

Diese Lösung finde ich sehr schlank und gut lesbar. Was mich hier stört, ist, dass durch FILTER alle verarbeiteten Tabelleneinträge kopiert werden. Es werden alle Datensätze die der WHERE-Anweisung entsprechen in eine neue Tabelle kopiert. Die Tabelle ist zwar temporär und wird nur für die Zeit der Verarbeitung des FILTER-Befehls verwendet, aber bei großen Tabellen kann sich die zusätzliche Speicherlast negativ auswirken.

Lösung 3 – REDUCE

Ich wollte dann unbedingt noch eins drauf setzen und eine Lösung haben, die auch bei vielen Ausprägungen von TF funktioniert und die Werte von TF nicht bekannt sind. Zudem wollte ich komplett die neuen Sprachfeatures verwenden.

Bei beiden vorhergehenden Lösungen fand ich es nicht gut, dass gezielt im Programm auf ABAP_TRUE und ABAP_FALSE abgefragt wurde. In diesem Beispiel ist es in Ordnung, weil das die Vorgabe war. Der häufigere Fall ist jedoch, dass eine Gruppe viele und gegebenenfalls nicht bekannte Ausprägungen hat (Verkaufsorganisation, Datum, Materialnummer, etc.).

Meine Lösung bestand dann aus einer Kombination aus VALUE und REDUCE:

```
DATA(sum) = VALUE ttf( FOR GROUPS grp OF <name> IN names
                       WHERE ( name IS NOT INITIAL )
                       GROUP BY ( tf = <name>-tf )
                         ( tf    = grp
                           count = REDUCE #( INIT i = 0
                                             FOR name IN names
                                             WHERE ( tf = grp )
                                             NEXT i = i + 1 ) ) ).
```

Diese Lösung baut eine Tabelle auf aus TF und COUNT, so dass alle Gruppenwerte mit der entsprechenden Anzahl Einträge in der Tabelle SUM landen.

Eigentlich müsste diese Lösung die langsamste sein, denn es werden zuerst die Gruppen gebildet. Dafür muss die gesamte Tabelle durchlaufen werden. Dann werden zu jedem Gruppeneintrag erneut die zugehörigen Einträge gelesen und gezählt. Deswegen wollte ich zuerst gar keine Laufzeitmessung machen. Die Herausforderung für mich war in erster Linie, die Problemstellung mit den neuen Sprachfeatures abzubilden, da ich mich mit der Syntax eher schwer tue.

Lösung 4 – 1xLOOP+WHERE(DATA)

Ich habe mit den vorhandenen drei Lösungen ein Testprogramm geschrieben um die Laufzeit mit der Transaktion SAT analysieren zu können.

Allerdings habe ich gemerkt, dass ich die Lösung von Tron falsch übernommen hatte, nämlich folgendermaßen:

```
LOOP AT names INTO name.
  CASE name-tf.
    WHEN abap_true.
      ADD 1 TO zaehler_true.
    WHEN abap_false.
      ADD 1 TO zaehler_false.
  ENDCASE.
ENDLOOP.
```

Anstatt zweier LOOPs hatte ich nur einen LOOP und eine CASE-Abfrage.

Da ich die schon dabei war zu testen, wollte ich Trons Code genau so übernehmen, da ich davon ausging, dass meine Variante mit CASE schneller sein würde. Allerdings war dem nicht so…

Update

Zusätzlich zu den LOOP-Lösungen, die mit dem Zusatz INTO workarea arbeiten, habe ich noch die Varianten mit ASSIGNING (Feldsymbol) und TRANSPORTING NO FIELDS aufgenommen.

Lösung 5 – 1xLOOP+CASE(Fieldsymbol)

Die Lösung mit einem LOOP und CASE-Anweisung jedoch mit LOOP-ASSIGNING.

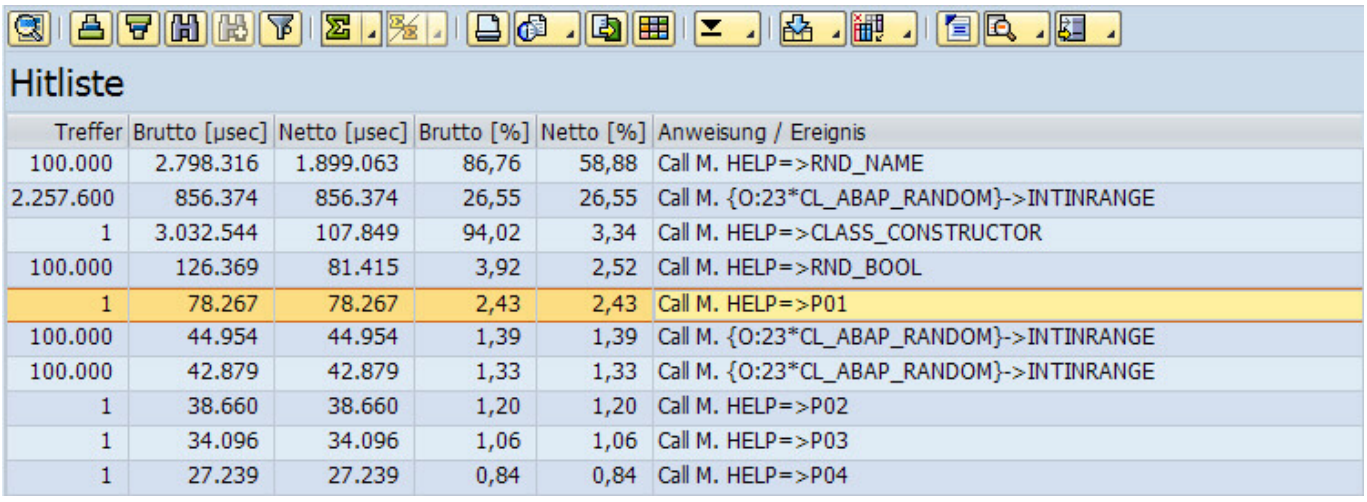
Lösung 6 – 2xLOOP+WHERE(Fieldsymbol)

Die Lösung mit zwei LOOPs und entsprechender WHERE-Bedingung jedoch mit LOOP-ASSIGNING.

Lösung 7 – 2xLOOP+WHERE(ohne Feldtransport)

Die Lösung mit zwei LOOPs und entsprechender WHERE-Bedingung jedoch mit dem Zusatz TRANSPORTING NO FIELDS.

Laufzeitanalyse



Hitliste

Treffer	Brutto [µsec]	Netto [µsec]	Brutto [%]	Netto [%]	Anweisung / Ereignis
100.000	2.798.316	1.899.063	86,76	58,88	Call M. HELP=>RND_NAME
2.257.600	856.374	856.374	26,55	26,55	Call M. {O:23*CL_ABAP_RANDOM}->INTINRANGE
1	3.032.544	107.849	94,02	3,34	Call M. HELP=>CLASS_CONSTRUCTOR
100.000	126.369	81.415	3,92	2,52	Call M. HELP=>RND_BOOL
1	78.267	78.267	2,43	2,43	Call M. HELP=>P01
100.000	44.954	44.954	1,39	1,39	Call M. {O:23*CL_ABAP_RANDOM}->INTINRANGE
100.000	42.879	42.879	1,33	1,33	Call M. {O:23*CL_ABAP_RANDOM}->INTINRANGE
1	38.660	38.660	1,20	1,20	Call M. HELP=>P02
1	34.096	34.096	1,06	1,06	Call M. HELP=>P03
1	27.239	27.239	0,84	0,84	Call M. HELP=>P04

Der Vollständigkeit halber habe ich die Messung auch noch einmal mit der Variante „SORTED

TABLE“ durchgeführt. Und wieder war ich überrascht: Die Variante mit Sorted Table ist deutlich langsamer als die Variante mit Standard Table…

Hier das Ergebnis der Laufzeitmessungen mit 100.000 Datensätzen und STANDARD TABLE:

```
Variante          Laufzeit
P01_REDUCE         76.602
P02_FILTER         36.755
P03_LOOP_CASE      33.891
P04_LOOP_WHERE     27.282
P05_LOOP_CASE_FS   25.097
P06_LOOP_WHERE_FS  18.805
P07_LOOP_WHERE_NO  17.774
```

abapgit

Eine aktuelle Version der Lösungen (inzwischen erweitert auf den Vergleich mit SORTED TABLE) findest du im github Tricktresor-Repository:

https://github.com/tricktresor/loop_performance_comparison

Code

Methode rnd_name baut aus zufälligen Buchstaben Fantasienamen auf.

Methode rnd_bool liefert per Zufall den Wert TRUE oder FALSE zurück.

Die Methoden p01 – p07 enthalten die jeweils erwähnten Lösungsvarianten.

```
REPORT.
" http://www.abapforum.com/forum/viewtopic.php?f=1&t=21900&p=82017#p82017

PARAMETERS p TYPE i DEFAULT 100000.

CLASS help DEFINITION.
 PUBLIC SECTION.
 CLASS-METHODS rnd_name RETURNING VALUE(name) TYPE string.
 CLASS-METHODS rnd_bool RETURNING VALUE(tf) TYPE boolean.
 CLASS-METHODS class_constructor.
 CLASS-METHODS p01_reduce.
 CLASS-METHODS p02_filter.
 CLASS-METHODS p03_loop_case.
 CLASS-METHODS p04_loop_where.
 CLASS-METHODS p05_loop_case_fs.
 CLASS-METHODS p06_loop_where_fs.
 CLASS-METHODS p07_loop_where_no.
 PROTECTED SECTION.
 CLASS-DATA rnd TYPE REF TO cl_abap_random.
 TYPES:
 BEGIN OF lst_names,
 name TYPE string,
```

```
 tf TYPE abap_bool,
 END OF lst_names,
 ltt_names TYPE STANDARD TABLE OF lst_names
 WITH NON-UNIQUE KEY name
 WITH NON-UNIQUE SORTED KEY key_tf COMPONENTS tf.

* ltt_names TYPE SORTED TABLE OF lst_names
* WITH NON-UNIQUE KEY name
* WITH NON-UNIQUE SORTED KEY key_tf COMPONENTS tf.
 CLASS-DATA names TYPE ltt_names.
ENDCLASS.

CLASS help IMPLEMENTATION.
 METHOD class_constructor.
 rnd = cl_abap_random=>create( ).
 names = VALUE ltt_names( FOR i = 1 THEN i + 1 WHILE i <= p
 ( name = help=>rnd_name( ) tf = help=>rnd_bool( ) ) ).

 ENDMETHOD.

 METHOD rnd_name.
 DATA(len) = rnd->intinrange( low = 5 high = 40 ).
 DO len TIMES.
 DATA(pos) = rnd->intinrange( low = 0 high = 25 ).
 name = name && sy-abcde+pos(1).
 ENDDO.
 ENDMETHOD.

 METHOD rnd_bool.
 CASE rnd->intinrange( low = 0 high = 1 ).
 WHEN 0.
 tf = abap_false.
 WHEN 1.
 tf = abap_true.
 ENDCASE.
 ENDMETHOD.

 METHOD p01_reduce.
 TYPES:
 BEGIN OF stf,
 tf TYPE abap_bool,
 count TYPE i,
 END OF stf,
 ttf TYPE SORTED TABLE OF stf WITH UNIQUE KEY tf.

 DATA(sum) = VALUE ttf( FOR GROUPS grp OF <name> IN names
 WHERE ( name IS NOT INITIAL )
 GROUP BY ( tf = <name>-tf )
 ( tf = grp
 count = REDUCE #( INIT i = 0
 FOR name IN names
```

```
 WHERE ( tf = grp )
 NEXT i = i + 1 ) ) ).
* cl_demo_output=>display_data( sum ).
 ENDMETHOD.

 METHOD p02_filter.
 DATA(lv_true) = lines( FILTER #( names USING KEY key_tf WHERE tf = abap_true
) ).
 DATA(lv_false) = lines( FILTER #( names USING KEY key_tf WHERE tf =
abap_false ) ).

* DATA(out) = cl_demo_output=>new( ).
* out->write( lv_true )->write( lv_false )->display( ).
 ENDMETHOD.

 METHOD p03_loop_case.

 DATA lv_true TYPE i.
 DATA lv_false TYPE i.

 LOOP AT names INTO DATA(name).
 CASE name-tf.
 WHEN abap_true. ADD 1 TO lv_true.
 WHEN abap_false. ADD 1 TO lv_false.
 ENDCASE.
 ENDLOOP.

* DATA(out) = cl_demo_output=>new( ).
* out->write( lv_true )->write( lv_false )->display( ).
 ENDMETHOD.

 METHOD p04_loop_where.

 DATA lv_true TYPE i.
 DATA lv_false TYPE i.

 LOOP AT names INTO DATA(name) WHERE tf = abap_true.
 ADD 1 TO lv_true.
 ENDLOOP.
 LOOP AT names INTO name WHERE tf = abap_false.
 ADD 1 TO lv_false.
 ENDLOOP.

* DATA(out) = cl_demo_output=>new( ).
* out->write( lv_true )->write( lv_false )->display( ).
 ENDMETHOD.

 METHOD p05_loop_case_fs.

 DATA lv_true TYPE i.
 DATA lv_false TYPE i.
```

```
 LOOP AT names ASSIGNING FIELD-SYMBOL(<name>).
 CASE <name>-tf.
 WHEN abap_true. ADD 1 TO lv_true.
 WHEN abap_false. ADD 1 TO lv_false.
 ENDCASE.
 ENDLOOP.

* DATA(out) = cl_demo_output=>new( ).
* out->write( lv_true )->write( lv_false )->display( ).
 ENDMETHOD.

 METHOD p06_loop_where_fs.

 DATA lv_true TYPE i.
 DATA lv_false TYPE i.

 LOOP AT names ASSIGNING FIELD-SYMBOL(<name>) WHERE tf = abap_true.
 ADD 1 TO lv_true.
 ENDLOOP.
 LOOP AT names ASSIGNING <name> WHERE tf = abap_false.
 ADD 1 TO lv_false.
 ENDLOOP.

* DATA(out) = cl_demo_output=>new( ).
* out->write( lv_true )->write( lv_false )->display( ).
 ENDMETHOD.

 METHOD p07_loop_where_no.

 DATA lv_true TYPE i.
 DATA lv_false TYPE i.

 LOOP AT names TRANSPORTING NO FIELDS WHERE tf = abap_true.
 ADD 1 TO lv_true.
 ENDLOOP.
 LOOP AT names TRANSPORTING NO FIELDS WHERE tf = abap_false.
 ADD 1 TO lv_false.
 ENDLOOP.

* DATA(out) = cl_demo_output=>new( ).
* out->write( lv_true )->write( lv_false )->display( ).
 ENDMETHOD.


ENDCLASS.

START-OF-SELECTION.


 help=>p01_reduce( ).
 help=>p02_filter( ).
```

```
help=>p03_loop_case( ).
help=>p04_loop_where( ).
help=>p05_loop_case_fs( ).
help=>p06_loop_where_fs( ).
help=>p07_loop_where_no( ).
```