Estándares de Performance ABAP

2012

1. Tabla de Contenido

[1. Tabla de Contenido 3](#_Toc346902838)

[2. Objetivo 4](#_Toc346902839)

[3. Estándares Generales de Performance 4](#_Toc346902840)

[Estructuras de los programas 6](#_Toc346902841)

[Legibilidad 7](#_Toc346902842)

[Técnicas de programación 7](#_Toc346902843)

[Tablas Internas 7](#_Toc346902844)

[Selecciones de bases de datos – Reglas generales 14](#_Toc346902845)

[Cláusulas Select 17](#_Toc346902846)

[Definición de variables 20](#_Toc346902847)

[Prueba de un campo para los valores múltiples 20](#_Toc346902848)

[Optimizando la estructura IF y CASE 21](#_Toc346902849)

[Ejecución de cálculos 21](#_Toc346902850)

[Aritmética de números embalados 22](#_Toc346902851)

[Moviendo estructuras idénticas 22](#_Toc346902852)

[Moviendo contenido de una tabla: 22](#_Toc346902853)

[Comandos 22](#_Toc346902854)

[Catch/End Catch 23](#_Toc346902855)

[Check 24](#_Toc346902856)

[Commit Work y Rollback Work 24](#_Toc346902857)

[Describe 25](#_Toc346902858)

[Free 25](#_Toc346902859)

[Include 25](#_Toc346902860)

[Textos 26](#_Toc346902861)

# Objetivo

El objetivo del presente documento es establecer una “guía rápida” para una buena práctica de programación, que permitirá una mayor eficiencia en el manejo de ABAP/4.

# Estándares Generales de Performance

* Los patrones de los programas estándares deberían ser siempre usados por todos los programas e includes.
* Los programas ABAP que actualizan datos maestros y transaccionales deben siempre usar transacciones de SAP (dónde las transacciones están disponibles) utilizando módulos de función estándar de SAP, herramientas BDC o ‘**CALL TRANSACTION**’. Esto asegura que unidades lógicas de trabajo, rollback, operaciones de fijación y correcciones sean realizadas. Las tablas de SAP nunca deben ser actualizadas directamente.
* Los objetos provistos por SAP, incluyendo tablas, programas ABAP, DYNPROS, transacciones de SAP, etc., no pueden ser modificados de acuerdo a la Política de modificación de programa del cliente.
* Si es posible, escribir código que sea reutilizable y pueda ser almacenado en una librería central. Los módulos de función son un ejemplo de esto.
* Cuidar la longitud de los programas a un mínimo. Cada programa debería manejar un problema concreto.
* Todos los programas deben incluir apropiado manejo de errores para evitar indeseables finalizaciones del programa. Esto significa que el estado del sistema (**SY-SUBRC**) debe ser chequeado después de cada evento que cambia en el programa.
* Usar la sentencia ‘**CATCH/ENDCATCH**‘ para capturar errores en tiempo de ejecución.
* Si hay más de dos valores posibles para el **SY-SUBRC** después de un evento, se deben chequear y manejar todos los valores explícitamente en el programa.
* Si hay un gran número de declaraciones de datos necesarios como parte de un programa, considerar separar sus declaraciones dentro de un **INCLUDE**. El nombre del include debería ser el mismo que el nombre del programa con el sufijo \_TOP.
* Usar subrutinas (**FORMS** y módulos de función) siempre que sea posible para mejorar la lectura del código y el flujo lógico. Considerar crear una subrutina siempre que un bloque de código sea llamado desde varios lugares en un programa.
* R/3 provee muchas funciones que realizan varias tareas como por ejemplo convertir monedas, calcular fechas, explorar BOM’s etc. En la mayoría de los casos, estos módulos son eficientes y pueden reducir tiempo de código, por lo tanto deberían ser utilizados dónde sea posible. Vale la pena tomarse el tiempo para buscar un módulo de función de la librería para una función que cubre sus necesidades.
* Evite la función **COMPUTE** si las funciones de sumas, restas, multiplicaciones o divisiones pueden ser usadas.
* Cuando se declaran variables internas, utilizar la opción **LIKE** únicamente para objetos internos del programa. Referenciar a los Objetos del Diccionario ABAP usando **TYPE** ya que éstos son tipos de datos. Usar la referencia de objetos en el diccionario ABAP siempre que sea posible para asegurarse que las variables tienen los mismos atributos de campo como el objeto existente, el cual ya se ha declarado. Si el tipo de campo, al cual se refiere cambia, el sistema en tiempo de ejecución de ABAP actualizará todas las referencias automáticamente. Esto además evita al sistema de realizar innecesarias (y puede ser indeseable) conversiones de tipo de datos.

Por ejemplo:

**DATA**: ls\_lifnr **TYPE** lfa1-lifnr.

* Utilizar la sentencia **INCLUDE** para las declaraciones de datos, módulos **PBO**, módulos PAI, subrutinas, y procesamiento de evento individual.
* Considerar usar un guión bajo para conectar varias palabras en el nombre de una variable y/o nombre de módulos.
* Donde sea posible, califique todas las especificaciones de los campos. Por ejemplo, usar KA-KTRNA, no KTRNA solamente.
* Cuando se declara una variable local, ésta debería ser inicializada en la sección de declaraciones siempre que sea posible.
* Utilizar elementos de texto para definir una constante descriptiva de cadena de caracteres.
* Nunca desloguearse o irse sin compilar el código en forma satisfactoria.
* El usuario no está autorizado a actualizar las tablas. Nosotros tenemos que crear una transacción asociada.
* Para diagnosticar los problemas de performance, es recomendable usar la transacción **SE30 Y ST05**.

ST05 es la transacción del rastreo del SQL y se puede utilizar para medir el funcionamiento de las declaraciones SELECTS del programa.

SE30 es la transacción que analiza en Tiempo de Ejecución nuestro programa, nos provee de información del consumo de tiempo en Acceso al Sistema, Acceso a Base de Datos y Ejecución de Lógica Algorítmica del Programa.

Una de las mejores herramientas para analizar estático del funcionamiento es el Code Inspector (SCI). Hay muchas opciones para encontrar errores comunes y embotellamientos posibles del funcionamiento.

* Evite dejar código “muerto” en el programa. Comente (o borre) variables que no son referenciadas y códigos que no son ejecutados.

## Estructuras de los programas

ABAP es un lenguaje de programación ejecutado por eventos. Sin embargo, la estructura apropiada del programa y la codificación modular sigue siendo importante para mantener la comprensión y eficacia de un programa ABAP. Un programa se debe dividir en distintas secciones las cuáles son correctamente documentadas en el código. Siempre que sea posible, utilice los módulos de función ya disponibles en vez de crear nuevos. Refiera a la plantilla del programa en la sección de apéndices para un orden y una estructura apropiados de su programa.

Los Forms no deben ser demasiado largos para asegurar la modularidad y la facilidad de seguir la lógica del programa.

Por ejemplo:

**PERFORM** intialization.

**PERFORM** main\_processing.

**FORM** main\_processing.

**………………………………**

**ENDFORM**.

**FORM** main\_processing.

**PERFORM** modify\_data.

**PERFORM** calculation.

**ENDFORM**.

Nota: Si la funcionalidad de un **FORM** es muy general tal que se podría utilizar en otros programas, puede ser útil crearlo como módulo de función o colocarlo en un Include.

## Legibilidad

* Comience cada nuevo comando o cláusula en una nueva línea. No ponga los comandos múltiples en la misma línea.
* Si una línea de código supera el límite de ancho continuar en la línea posterior.
* Las líneas en blanco y la sangría se deben utilizar para dar claridad entre las secciones del código así como entre definiciones y declaraciones.
* Utilice el **PRETTY PINTER** en sus programas para asegurar el formato uniforme de los títulos en forma estándar en el código del programa.
* Agregue la descripción apropiada del **FORM** a estos títulos. Nota: “**PRETTY PRINTER**” crea solamente títulos del **FORM** si ninguna línea de comentario existe después de la declaración anterior de ABAP.

## Técnicas de programación

### Tablas Internas

Todos los programas ABAP deben utilizar las tablas internas que son declaradas en la fase inicial de los datos. Todo el trabajo en el programa se debe hacer en estas tablas internas. Este método es más rápido que usar la tabla real de la base de datos y reduce el riesgo de los bloqueos de la tabla.

#### Definición de Tablas Internas.

Al definir una tabla usted debe referir a un lineobject o a un linetype, que podrían ser una estructura de datos o una definición de tipo estructurada.

Por ejemplo:

**DATA**: **BEGIN OF** gt\_rec\_person,   
 name(20) **VALUE** 'May',   
 age **TYPE** **I,**   
 **END OF** gt\_rec\_person.

o:

**TYPES**: **BEGIN OF** gty\_person,   
 name(20),   
 age **TYPE I**,   
 **END OF** gty\_person.

En la definición de tabla refiera a ese objeto o tipo.

Por ejemplo:

**DATA** : gt\_persons **LIKE** **STANDARD TABLE OF** gty\_person.

o:

**TYPES**: gty\_htab **TYPE HASHED TABLE OF** gty\_person

**WITH UNIQUE KEY** name.

**DATA**: gt\_persons **TYPE** gty\_htab.

Si usted desea utilizar definiciones de tabla con la sentencia **OCCURS**, tenga presente que puede no estar permitido en un contexto de objetos ABAP. Ver ‘declaraciones con **OCCURS** no permitidas’ en la documentación online.

Fuera de un contexto de objetos en ABAP la sentencia **OCCURS** todavía se permite por razones de compatibilidad con versiones más viejas. Sin embargo, se recomienda que solamente las nuevas versiones de las definiciones de tablas internas sean utilizadas. Esto facilita reutilizar el código en un contexto de objetos en ABAP.

Al definir una tabla la sentencia OCCURS solamente asigna un área en la memoria del tamaño especificado. Si se necesitan más ocurrencias durante la ejecución del programa el tamaño se amplía dinámicamente. La ejecución es más lenta cuando se pasan los límites del tamaño, por lo tanto hágalo suficientemente grande como para englobar la mayoría de los casos típicos pero no tan grande como para que quede espacio innecesario sin utilizar.

Por ejemplo:

**DATA**: **BEGIN OF** gt\_tab **OCCURS** 100,  
 telf1 **LIKE** kna1-telf1, “Phone no  
 banks **LIKE** knbk-banks, “Bank  
 name1 **LIKE** kna1-name1, “Last Name  
 **END OF** gt\_tab.

Nota: Si usted no está seguro del tamaño necesario de la tabla o si puede variar dramáticamente es decir, 10 o 10.000 registros dependiendo de los parámetros de entrada, es conveniente especificar el tamaño como **OCURRS** 0.

#### Ordenar Tablas Internas.

Al ordenar las tablas internas, especifique los campos por los cuáles se desea ordenar.

**SORT** gt\_itab **BY** fld1 fld2.

es más eficiente que

**SORT** gt\_itab.

El tipo de acceso a una tabla interna define la manera en la cual ABAP tiene acceso a las entradas individuales de la tabla. Hay tres tipos de accesos: Estándar, Sorted, y Hashed. Refiera a la plantilla del programa para el formato apropiado de las declaraciones internas de la tabla.

#### Tablas Internas Estándar

Las tablas internas tienen un índice lineal. Se puede acceder a ellas usando el índice o la clave. Si se utiliza la clave, el tiempo de respuesta está en relación lineal al número de entradas de la tabla. La clave de una tabla estándar es siempre no única, y no se puede incluir ninguna especificación para la unicidad en la definición de la tabla.

Este tipo de tablas es particularmente apropiado si usted desea realizar entradas individuales en la tabla utilizando el índice. Ésta es la manera más rápida de tener acceso a registros de la tabla. Para llenar una tabla estándar, añada las líneas usando **APPEND**. Usted debe leer, modificar y suprimir líneas refiriendo al índice (opción **INDEX** con el comando de ABAP). El tiempo de respuesta para tener acceso a una tabla estándar está en relación lineal al número de entradas de la tabla. Si usted necesita utilizar la clave de acceso, las tablas estándar son apropiadas si usted puede llenar y procesar la tabla en pasos separados. Por ejemplo, usted puede llenar una tabla estándar agregando registros y después ordenarla. Si usted entonces utiliza la clave de acceso con la opción de búsqueda binaria (**BINARY**), el tiempo de respuesta está en relación logarítmica al número de entradas de la tabla.

#### Tablas internas Sorted

Las tablas sorted siempre se guardan ordenadas correctamente por clave. También tienen una clave lineal, y, como las tablas estándar, se puede acceder a ellas usando el índice de la tabla o la clave. Cuando se utiliza la clave, el tiempo de respuesta está relacionada logarítmicamente con el número de entradas de la tabla, debido a que el sistema utiliza una búsqueda binaria. La clave de una tabla sorted puede ser única, o no, y se debe especificar **UNIQUE** o **NON-UNIQUE** en la definición de la tabla. Tanto las tablas estándar como las tablas sorted pertenecen al grupo genérico de tablas indexadas.

Este tipo de tablas es particularmente conveniente si se quiere que la tabla se ordene mientras se están agregando entradas a ella. Usted llena la tabla usando la declaración (**INSERT**), según la secuencia definida en la clave de la tabla. Las entradas de la tabla que no coinciden se reconocen antes de ser insertadas. El tiempo de respuesta para el acceso usando la clave está relacionado logarítmicamente al número de entradas de la tabla, debido a que el sistema utiliza automáticamente una búsqueda binaria. Las tablas sorted son apropiadas para el procesamiento secuencial parcial en un **LOOP**, mientras que la condición **WHERE** contiene el principio de la clave de la tabla.

No se recomienda hacer INSERT a tablas sorted, sino llevarlo de un solo “porrazo”, sea mediante SELECT … INTO TABLE…. O MOVE T1[] TO T2[].

*Código 1*

*LOOP AT STAB INTO WA WHERE K = SUBKEY.*

*" ...*

*ENDLOOP.*

*Código 2*

*LOOP AT HTAB INTO WA WHERE K = SUBKEY.*

*" ...*

*ENDLOOP.*

El Código 1 es más eficiente que el Código 2.

#### Tablas internas Hashed

Las tablas **HASHED** no tienen ningún índice lineal interno. Se puede tener acceso solamente especificando la clave. El tiempo de respuesta es constante, sin importar el número de las entradas de la tabla, debido a que la búsqueda utiliza un algoritmo hash. La clave de una tabla hashed debe ser única, y usted debe especificar **UNIQUE** en la definición de tabla.

Este tipo de tablas es particularmente conveniente si usted desea principalmente utilizar la clave de acceso para las entradas de la tabla. Usted no puede tener acceso a las tablas hashed usando el índice. Cuando usted utiliza la clave de acceso, el tiempo de respuesta es constante, sin importar el número de entradas de la tabla. Como con las tablas de la base de datos, la clave de una tabla hashed es siempre única. Las tablas hashed son por lo tanto una manera útil de construir y de usar las tablas internas que son similares a las tablas de la base de datos.

Si la llave se repite en una tabla hashed el programa se “caerá” en ejecución.

*Código 1*

*DO 250 TIMES.*

*N = 4 \* SY-INDEX.*

*READ TABLE HTAB INTO WA WITH TABLE KEY K = N.*

*IF SY-SUBRC = 0.*

*" ...*

*ENDIF.*

*ENDDO.*

*Código 2*

*DO 250 TIMES.*

*N = 4 \* SY-INDEX.*

*READ TABLE STAB INTO WA WITH KEY K = N.*

*IF SY-SUBRC = 0.*

*" ...*

*ENDIF.*

*ENDDO.*

El Código 1 es más eficiente que el Código 2.

#### Lectura de Tablas Internas Standard

Cuando lee un registro individual en una tabla interna standard, **READ TABLE WITH KEY** no es una lectura directa. Esto significa que si los datos son ordenados de acuerdo a la clave, el sistema leerá secuencialmente la tabla. Por lo tanto, se debe:

* Hacer **SORT** a la tabla
* Usar **READ TABLE WITH KEY BINARY SEARCH** para mayor eficiencia ya que se realize una búsqueda binaria en vez de una búsqueda lineal.

#### Copiando o agregando Tablas Internas

Use esto:

<tab2>**[]** **=** <tab1>**[]**.  (si <tab2> esta vacía)

En vez de esto:

**LOOP AT** <tab1>.

**APPEND** <tab1> **TO** <tab2>.

**ENDLOOP**.

Sin embargo, si <tab2> no esta vacía y no debe ser sobre-escrito, entonces use:

**APPEND LINES OF** <tab1> [**FROM** index1] [**TO** index2] **TO** <tab2>.

#### Modificación de campos dentro de las tablas internas

#### Modificando lineas simples:

Use esto:

*WA-DATE = SY-DATUM.*

*MODIFY ITAB FROM WA INDEX 1 TRANSPORTING DATE.*

Es más eficiente que:

*WA-DATE = SY-DATUM.*

*MODIFY ITAB FROM WA INDEX 1.*

**Modificando grupos de lineas directamente:**

Use esto:

*Field-Symbols:<WA> like ITAB.*

*LOOP AT ITAB ASSIGNING <WA>.*

*I = SY-TABIX MOD 2.*

*IF I = 0.*

*<WA>-FLAG = 'X'.*

*ENDIF.*

*ENDLOOP.*

Es más eficiente que:

*LOOP AT ITAB INTO WA.*

*I = SY-TABIX MOD 2.*

*IF I = 0.*

*WA-FLAG = 'X'.*

*MODIFY ITAB FROM WA.*

*ENDIF.*

*ENDLOOP.*

#### Eliminación de registros dentro de las tablas internas

Use esto:

*SORT ITAB by k.*

*DELETE ADJACENT DUPLICATES FROM ITAB*

*COMPARING K.*

Es más eficiente que:

*READ TABLE ITAB INDEX 1 INTO PREV\_LINE.*

*LOOP AT ITAB FROM 2 INTO WA.*

*IF WA = PREV\_LINE.*

*DELETE ITAB.*

*ELSE.*

*PREV\_LINE = WA.*

*ENDIF.*

ENDLOOP.

#### Loops anidados

Use esto:

*SORT ITAB1.*

*SORT ITAB2.*

*I = 1.*

*LOOP AT ITAB1 INTO WA1.*

*LOOP AT ITAB2 INTO WA2 FROM I.*

*IF WA2-K <> WA1-K.*

*I = SY-TABIX.*

*EXIT.*

*ENDIF.*

*" ...*

*ENDLOOP.*

Es más eficiente que:

*LOOP AT ITAB1 INTO WA1.*

*LOOP AT ITAB2 INTO WA2*

*WHERE K = WA1-K.*

*" ...*

*ENDLOOP.*

*ENDLOOP.*

### Selecciones de bases de datos – Reglas generales

* Reduzca al mínimo los accesos de base de datos. Mantenga el número de los procesos de comunicación entre la base de datos y los servidores de aplicación, así como también el conjunto de datos transferido tan pequeño como sea posible.
* Haga las llamadas más restrictivas primero. Por ejemplo, para procesar todos los materiales con una cantidad en mano mayor a 50, las tablas que contienen las cantidades de stock se deben leer antes de las tablas de maestro de materiales.
* Chequear siempre el DELETION FLAG en la clave de las tablas de datos maestros. (Es decir, KNA1, KNVV, MARA...). Los DELETION FLAGS generalmente se nombran LOEVM, LOEKZ, LVORM, etc.

Para información adicional sobre el trabajo con operaciones en la base de datos, refiérase al checklist de la optimización de la llamada de base de datos en índice de este documento.

#### No utilice la orden por cláusula en non-key o campos non-indexed.

Traiga los datos en una tabla interna con una sola llamada y después ordene la tabla interna. Esto releva a la base de datos del requisito adicional de la ordenamiento y de utilizar el procesamiento y la memoria del servidor.

#### Disminuya el número de las llamadas de base de datos

En todos los casos considere disminuir el número de llamadas a la base de datos. Siempre que sea posible trate de almacenar todos los registros en una tabla interna. Luego, realice el proceso usando la misma.

\*Copy document flow table into an internal table.

\*

**SELECT** vbeln “Subsequent SD Doc. No.

posnn “Subsequent SD Doc. Item.

**FROM** vbfa “Sales Document Flow

**INTO TABLE** gt\_vbfa

**WHERE** vbelv **IN** s\_doc\_number “Preceding SD Doc. No

**AND** posnv **IN** s\_item\_no. “Preceding Item of SD Doc.

#### Sentencias SELECT anidadas

Se deben evitar los **SELECT** anidados mientras dan lugar a un volumen grande de accesos de base de datos (dependientes en el tamaño de tablas). Donde accesos a la base de datos más futuros dependen de los datos almacenados en una tabla interna, la sintaxis siguiente puede ser utilizada:

**SELECT** field1 field2

**FROM** table

**INTO TABLE** gt\_tab2

**FOR ALL ENTRIES IN** gt\_tab1

**WHERE** field3 = itab1-field3

**AND** field4 = s\_field.

Si varios **SELECT** anidados parecen ser requeridos, crear una vista de la base de datos siempre que sea posible que una las tablas o utilice un **JOIN** (**OUTER JOIN** o **LEFT INNER JOIN**). Esto ayudará liberar a la aplicación actual del uso para que otras aplicaciones compartan el uso. Debido a que es la manera a en que SAP ha implementado los JOINS es importante seguir las reglas generales:

* No incluir más de 3 tablas en un **SELECT**.
* Asegurarse que el Join usa todas las claves posibles y utilizar el campo MANDT para el JOIN entre cada tabla que la posea, esto mejorará la selección.
* Asegurarse que hay una relación uno a uno en las tablas incluidas.

**SUBQUERIES:** Es mas eficiente usar Subqueries en vez de JOINs siempre y cuando no se requiera usar campos de la tabla al cual se hace el subquery. Por ejemplo en el siguiente ejemplo es eficiente este subquery, pero si quisieramos el campo AUFNR u otro campo de la tabla J\_3ABDSI, no se podria realizar mediante un subquery sino por un JOIN.

- Subquery:

SELECT VBAK~VBELN   
    FROM VBAK  
    INTO TABLE T\_ORDERS  
    WHERE EXISTS   
      ( SELECT AUFNR FROM J\_3ABDSI  
               WHERE J\_3ABDSI~AUFNR = VBAK~VBELN ).

- Join:

SELECT VBAK~VBELN J\_3ABDSI~ AUFNR  
  INTO TABLE T\_ORDERS

FROM VBAK INNER JOIN J\_3ABDSI ON J\_3ABDSI~AUFNR = VBAK~VBELN.

#### Cláusula Having

En un **SELECT**, la cláusula **HAVING** permite que usted especifique una condición lógica para los grupos en una cláusula **GROUP-BY**. El uso eficaz del **HAVING** puede reducir el set de datos transferidos de la base de datos a la aplicación del servidor. Cuando se utiliza la cláusula **HAVING**, los agregados y grupos son construidos en la base de datos en vez del servidor, de tal modo se reduce el set de datos.

#### Programas que actualizan la base de datos.

Al crear un programa que crea nuevos o modifica registros existentes se debe asegurar que su programa ejecuta **COMMIT** en una frecuencia razonable. Como regla general, usted debe hacer un **COMMIT** cada 1000 o 10000 registros. Si su programa está funcionando por un período de tiempo largo sin hacer un **COMMIT**, la base de datos y el sistema entero se colgará. Esta es la única opción para terminar la ejecución sino todos los recursos son consumidos por su programa.

Si múltiples **INSERT** se deben hacer a una tabla, el funcionamiento será optimizado insertando un array de una tabla, más que realizando los múltiples **INSERT** de registros únicos.

Por ejemplo:

**INSERT**  cust   
 **FROM TABLE** lt\_tab.

Será más eficiente que:

**LOOP AT** lt\_tab.  
 **MOVE-CORRESPONDING lt\_**tab **TO** cust.  
 **INSERT** cust.  
**ENDLOOP**.

#### Utilice Open SQL

Open SQL es la versión de SAP de SQL y está diseñado para trabajar con la mayoría de los sistemas RDBMS (Relational Database Management Systems). Evite usar cualquier característica nativa del SQL que no esté documentado ni se utilice en el Open SQL.

#### Base de datos lógica

Las bases de datos lógicas pueden ser ineficaces porque son pensadas generalmente como Selects Jerarquizados donde los volúmenes de datos grandes según lo recuperado pueden no cumplir exactamente los requisitos del programa. Hay, sin embargo, casos cuando el uso de una base de datos lógica puede ser justificado. Por ejemplo, en Asset Management se realizan una gran cantidad de cálculos que tomarían una cantidad de tiempo grande para realizar en código normal.

Cuando se utiliza base de datos lógica debe realizarse el authority check correspondiente para evitar un mal uso de la información.

### Cláusulas Select

* Para evitar problemas de funcionamiento la mayor parte de la clave como sea posible debe ser especificada en una cláusula **SELECT**
* Los campos se deben enumerar en la cláusula **WHERE** en el mismo orden que aparecen en la tabla.
* Si se está interesado solamente en unos pocos campos de una tabla,

Hacer :

**SELECT** <field1> <field2>...

**FROM** <tab> **INTO** <wa-field1> <wa-field2>

**WHERE**....

Esta técnica mejorará el funcionamiento y uso de la memoria interna.

En vez de:

**SELECT \* FROM** <tab> **WHERE**....

* Usar **SELECT SINGLE** < campo1 > < campo2 > **FROM**... tanto como sea posible. Solamente usar **SELECT \* FROM**... cuando sea absolutamente necesario.
* **SELECT SINGLE** devolverá un único registro. Sin embargo, si usted no especifica la clave entera a la tabla, el registro devuelto será cualquier línea de los registros que resuelva el criterio(cuando la búsqueda no sea de un registro único). Solamente usar **SELECT SINGLE** cuando la clave completa de la tabla es conocida.
* Especificar varios valores de los campos clave de la tabla en un **WHERE** hará el **SELECT** más eficiente que comprobando valores después del **SELECT**.

Por ejemplo:

**PARAMETERS**: p\_langu **LIKE** **SY-LANGU**.

...

**SELECT \***

**FROM** t005t

**WHERE** spras **EQ** p\_langu.

...

**ENDSELECT.**

es más eficiente que:

**PARAMETERS**: p\_langu **LIKE SY-LANGU**.

...

**SELECT** \*

**FROM** t005t.

**CHECK** t005t-spras = p\_langu.

...

**ENDSELECT**.

* Cuanto más a la izquierda se ubiquen los campos clave en la tabla de la base datos más eficiente será el **SELECT**.

Por ejemplo, los campo clave (en secuencia) de la tabla **KNC3** (Customer special G/L transaction )

MANDT - Client

KUNNR - Customer number

BUKRS - Company code

GJAHR - Fiscal year

SHBKZ - Special G/L indicator

Al seleccionar datos de esta tabla, será mas eficiente especificar un valor en el WHERE para el campo KNC3-BUKRS que KNC3-GJAHR.

Esto es:

**SELECT** \*

**FROM** knc3

**WHERE** kunnr **EQ** ‘0000000001’

**AND** bukrs **EQ** ‘us01’.

....

**ENDSELECT**.

será más eficiente que:

**SELECT** \*

**FROM** knc3

**WHERE** kunnr **EQ** ‘0000000001’

**AND** gjahr **EQ** ‘1996’.

....

**ENDSELECT**.

Se puede especificar tantas condiciones en el **WHERE** como se crea conveniente en todos los tipos de base de datos. Como por Ej., tablas transparentes, tablas pool y tablas cluster. Sin embargo, tu debes saber que las condiciones del **WHERE** no son pasadas a la base de datos. Ellas deben ser procesadas por la interfaz de base de datos de SAP.

* Al tener acceso a las tablas pool y cluster, éstas se deben acceder usando la clave primaria completa.
* Evite la colocación ' **SELECT**' o un ' **SELECT SINGLE**' en un **LOOP** para reducir al mínimo el número de las peticiones de la base de datos.
* Evite usar ' **SELECT**.... **INTO CORRESPONDING FIELD** ' para registros menores a 1000, ya que la asociación con el campo correspondiente podría ser significativo, para este caso usar ' **SELECT**.... **INTO TABLE**'
* Evite usar ' **SELECT**.... **INTO TABLE**' para registros mayores a 1000. para este caso usar **SELECT**.... **INTO CORRESPONDING FIELD** '
* Al seleccionar registros de una base de datos cuando solamente la parte de un campo (en qué selección se basa) se sabe, utilice la opción **LIKE** como parte del **WHERE**.

Por ejemplo:

**SELECT** **\***

**FROM** t001g

**WHERE** bukrs **EQ**  ‘us01’

**AND** txtko **LIKE**  ‘\_\_pers%’.

....

**ENDSELECT**.

es más eficiente que:

**SELECT \***

**FROM** t001g

**WHERE** bukrs **EQ** ‘us01’.

**CHECK** t001g-txtko**+**2(4) = ‘pers’.

### Definición de variables

* Al definir variables utilice la referencia al diccionario ABAP tanto como sea posible para asegurar consistencia automática con los ítems en las tablas R/3.
* Todas las variables deben tener una descripción a la derecha de la declaración.
* Cuide siempre los nombres de SAP. No traduzca. Si usted necesita el mismo tipo de datos de diversas tablas como los números de documento para diversos documentos, prefije el nombre con el nombre de la tabla como:

WS\_VBAK\_VBELN “Sales Order No.  
 WS\_LIKP\_VBELN “Delivery No.

### Prueba de un campo para los valores múltiples

Al probar un campo individual para los valores múltiples, usted puede utilizar:

**IF** field = value1.

....

**ELSEIF** field = value2.

....

**ELSEIF** field = value3.

....

o

**CASE**  field.

**WHEN**  value1.

....

**WHEN**  value2.

....

**WHEN** value3.

....

**ENDCASE**.

El primer método es más eficiente al comprobar un campo hasta cerca de cinco valores. Superando este límite, usted debe utilizar la declaración del **CASE**. A este punto llega a ser más eficiente que el **IF** jerarquizado, y también mejorará la legibilidad.

### Optimizando la estructura IF y CASE

Para optimizar las estructuras **IF** y **CASE**, siempre chequea los valores en orden según el grado de probabilidad de cada valor.

Por ejemplo, ls\_fieldx puede tener los valores ‘A’, ‘B’, o ‘C’. Un valor de ‘B’ es el mas probable, seguido por ‘C’, luego ‘A’. Para optimizar un **CASE**, el código debería ser el siguiente:

**CASE** ls\_fieldx.

**WHEN**  ‘B’. “Valor más probable

....

**WHEN** ‘C’. “Próximo valor más probable

....

**WHEN**  ‘A’. “Valor menos probable

....

**ENDCASE**

Aquí, si ws\_fieldx tiene un valor de ' B ', sólo una prueba se realiza, si tiene un valor de ' C ', dos pruebas se deben realizar, etcétera. La codificación de este modo reduce el número medio de las pruebas realizadas por el programa.

### Ejecución de cálculos

Al realizar cálculos en ABAP, la cantidad de tiempo de la CPU usada depende del tipo de datos. En términos muy simples, los números enteros (tipo **I**) son los más rápidos, flotantes (el tipo **F**) requiere más tiempo, y packed (el tipo **P**) es el más costoso. Normalmente, los tipos packed se utilizan para evaluar expresiones aritméticas. Si, sin embargo, la expresión contiene una función punto flotante, o hay por lo menos un operando del tipo F, o el campo de resultado es el tipo **F**, la aritmética de punto flotante se utiliza en lugar de otro para la expresión entera. Por otra parte, si solamente ocurren los campos del tipo I o los campos de fecha y de la hora, el cálculo implica operaciones de número entero.

Puesto que la aritmética de punto flotante es relativamente rápida en plataformas de hardware SAP, debe ser utilizada cuando una mayor gama del valor es necesaria y el redondeo de errores puede ser tolerado.

El redondeo de errores puede ocurrir al convertir el formato (decimal) externo al formato interno correspondiente (base 2 o 16) o viceversa.

### Aritmética de números embalados

Si la aritmética de punto fijo en los atributos del programa no está seteado, los campos del tipo **P** se interpretan como números enteros sin lugares decimales. El parámetro de los DECIMALES de la declaración de los DATOS afecta solamente el formato de la salida del **WRITE**.

Por esta razón, SAP recomienda que siempre se establezca en los atributos del programa Aritmética de punto fijo cuando trabaja con campos tipo **P**.

### Moviendo estructuras idénticas

Use:

**MOVE** structure1 **TO** structure2.

en vez de:

**MOVE-CORRESPONDING** structure1 **TO** structure2.

### Moviendo contenido de una tabla:

Use:

**MOVE** structure**[ ]** **TO**  structure2**[ ]**

o use:

structure**[ ]** **=** structure2**[ ]**

### Comandos

Lo que sigue es una lista seleccionada de comandos preferidos junto con una descripción corta de la funcionalidad del comando. Para más detalles, utilice la ayuda del editor ABAP. Usted encontrará una lista de comando de ABAP a través del path del menú,

Utilities->ABAP keyword doc. en la pantalla inicial del editor ABAP. Si se utiliza el modo comando del editor ABAP, tipee “help <command>” en la línea de comandos.

### Catch/End Catch

Se puede capturar errores en tiempo de ejecución en un bloque encerrado en las cláusulas **CATCH**... **ENDCATCH**.

Sintaxis:

**CATCH SYSTEM-EXCEPTIONS** except1 = rc1... exceptn = rcn.

Observe lo siguiente:

* rc1... rcn deben ser los literales numéricos.
* **CATCH**... **ENDCATCH** puede estar ubicado en cualquier parte donde **IF**... **ENDIF** (por ejemplo) puede ocurrir. Esto es local, no cross-event.
* Puede ser jerarquizado a cualquier profundidad.
* Captura solamente errores en tiempo de ejecución en el nivel actual de la llamada. Esto significa, por ejemplo, que los errores en tiempo de ejecución que resultan de **PERFORM** or **CALL FUNCTION** no son capturados por **CATCH** ... **ENDCATCH.**

Por ejemplo:

**PARAMETERS li­\_**fact **TYPE** i.

**DATA**: **li­\_**fact\_save **TYPE** i,

**lp\_**res(16) **TYPE** p.

\*\*\* ARITHMETIC\_ERRORS contains COMPUTE\_BCD\_OVERFLOW \*\*\*

**CATCH SYSTEM-EXCEPTIONS** ARITHMETIC\_ERRORS = 5.

**lp\_**res = **li­\_**fact\_save = **li­\_**fact.

**SUBTRACT** 1 **FROM li­\_**fact.

**DO li­\_**fact **TIMES**.

**MULTIPLY** **lp­\_**res **BY** **li­\_**fact. "<- COMPUTE\_BCD\_OVERFLOW

**SUBTRACT** 1 **FROM** **li­\_**fact.

**ENDDO.**

**ENDCATCH**.

**IF** SY-SUBRC = 5.

**WRITE**: / 'Overflow! Factorial of', **li­\_**fact\_save,

'can not be calculated.'.

**ELSE**.

**WRITE**: / 'Factorial of', **li­\_**fact\_save, 'gives', **lp\_**res.

**ENDIF**.

### Check

**CHECK** evalúa la expresión lógica subsiguiente. Si es verdad, el programa continúa con la declaración siguiente. En un **LOOP** (**DO**, **SELECT**, **LOOP AT**, etc.), se termina el ciclo actual y el control se pasa al principio del **LOOP** para realizar el ciclo siguiente. En una rutina o una función del **FORM**, el **CHECK** termina el subprograma.

Si usted utiliza el comando **CHECK** correctamente, usted encontrará mucho más fácil leer y entender el código fuente del programa. Utilice el **CHECK** en vez de **IF**...**ENDIF** donde sea posible.

### Commit Work y Rollback Work

**COMMIT WORK** se utiliza al hacer operaciones de la base de datos. Concluye un proceso lógico. **SY-SUBRC** es cero si el **COMMIT WORK** fue realizado con éxito. Una vez que el **COMMIT WORK** se ha ejecutado no es posible el Rollback (**ROLLBACK WORK**).

Nota: Usted no puede realizar un **COMMIT WORK** dentro de un **SELECT** loop puesto que todo los punteros abiertos de la base de datos serán cerrados. Para realizar un **COMMIT WORK** al procesar una cantidad grande de datos, un loop lógico debe ser creado donde se procesa el read-loop.

Por ejemplo:

**DATA**: ls\_commit\_recs(4) **TYPE** C **VALUE** ‘5000’.

**CLEAR**: ls\_matnr, ls\_flg\_eof.

**WHILE** ls\_flg\_eof **IS INITIAL**.  
 **CLEAR** li\_cnt\_mara.  
 **SELECT** \*   
 **FROM** mara   
 **WHERE** matnr **GT** ls\_matnr.  
 **IF** li\_cnt\_mara = ls\_commit\_recs.  
 **EXIT**.  
 **ENDIF**.  
 **PERFORM** f3120\_upd\_material.

li\_cnt\_mara = li\_cnt\_mara + 1.  
 ls\_matnr = mara-matnr.  
 **ENDSELECT**.  
 **IF** sy-subrc **NE** 0.

ls\_flg\_eof = ‘X’.  
 **ENDIF.**  
 **COMMIT WORK**.  
 **ENDWHILE**.

### Describe

La cláusula **DESCRIBE** devuelve los atributos de un campo, una tabla interna o una lista. Use este comando para chequear que una tabla interna contiene datos.

Sintaxis: **DESCRIBE** <itab> **LINES** li\_lines.

### Free

En programas donde las tablas internas contienen muchos datos, el comando **FREE** debe ser usado en vez de utilizar **REFRESH** para limpiar la tabla.

El comando **FREE** vacía la tabla de todos los registros y restaura la tabla tamaño original en memoria (el número de ocurrencias mide la longitud de la tabla/estructura). El comando **REFRESH** limpia la tabla solamente.

### Include

Utilice la declaración **INCLUDE** para utilizar una estructura de datos existente. Todo include se debe describir con un comentario en tu código.

### Textos

* Es más rápido utilizar instrucciones del sistema que Funciones del sistema, e aquí un ejemplo:

STRING\_CONCATENATE...   ---> CONCATENATE  
STRING\_SPLIT...  ---> SPLIT  
STRING\_LENGTH...  ---> STRLEN()  
STRING\_CENTER...  ---> WRITE..TO. ..CENTERED  
STRING\_MOVE\_RIGHT  ---> WRITE...TO...RIGHT-JUSTIFIED

* Es más rápido borrar espacios en blanco con la derecha con **SHIFT...LEFT DELETING** **LEADING...** que utilizar la instrucción **CONDENSE**,
* Utilizar el **SPLIT** es más rápido que programar un tratamiento de un string realizado por nosotros. Usar la instrucción **SPLIT** es mejor que usar **SEARCH** y **MOVE** con offset.
* Copiar y Pegar: Click en el área a copiar y presione **CTRL+Y**, esto te permite copiar muchas líneas en un solo paso y luego pegarlas.
  + Es bueno documentar los programas o insertar comentarios en los programas. Importante también notificar los cambios que se realicen en el programa.
  + Perder tiempo en analizar lo que se necesita, que procesos utilizaremos y que tipo de datos y estructura vamos a utilizar.
  + Tener un plan adecuado para planear las pruebas que se van hacer al programa.
  + Use los operadores especiales **CO**, **CA**, **CS** en vez de que programe las operaciones, si las sentencias son ejecutadas para caracteres en cadenas largas, la eficiencia del CPU puede crecer sustancialmente.