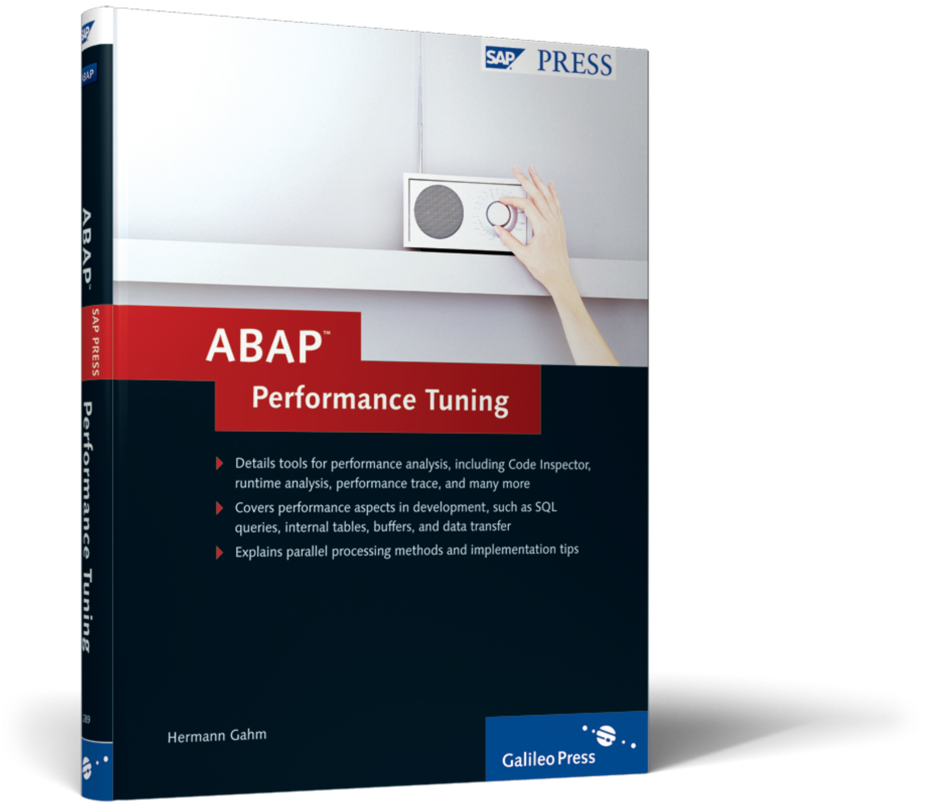
Hermann Gahm

ABAP™ Performance Tuning



Galileo Press

Bonn • Boston

Contents at a Glance

Contents

Inefficient accesses to internal tables are a frequent cause of long-running ABAP programs. This particularly applies to the processing of large data volumes. This chapter describes the most critical aspects for ABAP developers for the processing of internal tables.

7 Processing of Internal Tables

Internal tables are among the most complex data objects available in the ABAP environment. The use of internal tables lets you store dynamic datasets in the main memory. Internal tables are comparable to arrays and they spare the programmer the effort of program-controlled memory management thanks to their dynamic nature. The data in internal tables is managed per row, whereas each row has the same structure.

In most cases, internal tables are used for the buffering or formatting of contents from database tables. The type of access to internal tables plays an important role for performance, as is the case with database tables. Experience shows that the tuning of internal tables enables similarly major effects as the tuning of database accesses. The negative effects of inefficient accesses to internal tables for the overall system can be compensated more easily than inefficient database accesses by adding further CPUs or application servers. Inefficient database accesses affect the database as a central resource, whereas inefficient accesses to internal tables impact the better scalable application layer (see Chapter 2).

The following sections first provide a general overview of the internal tables. This is followed by a description of how the internal tables are organized in the main memory. The subsequent section discusses the different types of internal tables. The major part of this chapter then details the performance aspects for the processing of internal tables. Typical problematic examples and solution options are presented here.

7.1 Overview of Internal Tables

Internal tables are completely specified by four properties:

1. **Table type**
   The access type to the table type determines how ABAP accesses the individual table rows. Section 7.3, Table Types, discusses this topic in great detail.
2. **Row type**
   The row type of an internal table can be any ABAP data type.
3. **Uniqueness of the key**
   The key can be specified as unique or non-unique. In case of unique keys, there are no multiple entries (regarding the key) in the internal tables. The uniqueness is based on the table type. Standard tables only allow for non-unique keys and hashed tables only for unique keys.
4. **Key components (taking the sequence into account)**
   The key components and their sequence specify the criteria based on which the table rows are identified.

Figure 7.1 illustrates this syntactically.

Field1	Field2	Field3
A	1	10
A	2	5
B	1	7
B	2	25

```
TYPES: <itabtype> TYPE <tablekinddef> of <linetype>
       [WITH [UNIQUE | NON-UNIQUE] <keydef> ]
       [INITIAL SIZE <n>].

DATA:  <itab> TYPE <tablekind> of <linetype>
       WITH [UNIQUE | NON-UNIQUE] <keydef>
       [INITIAL SIZE < n>] .

<tablekinddef>:
[STANDARD] TABLE | SORTED TABLE | HASHED TABLE
for types also:    INDEX TABLE  | ANY TABLE

<keydef>
KEY f1 ...  fn |
KEY TABLE LINE |
DEFAULT KEY
```

Figure 7.1 Internal Tables — Declaration

The combination of access type and table type is mainly relevant for the performance. Section 7.3, Table Types, discusses the various access types and table types.

Before describing the table types in detail, let's first discuss the organization of internal tables in the main memory.

7.2 Organization in the Main Memory

In the main memory, the internal tables, just like the database tables, are organized in blocks or pages. In the context of internal tables, the following sections use the term *pages*.

When an internal table is declared in an ABAP program, the system only creates a reference (table reference) in the main memory initially. Only when entries are written to the table does the system create a table header and a table body. Figure 7.2 shows a schematic diagram of the organization in the main memory.

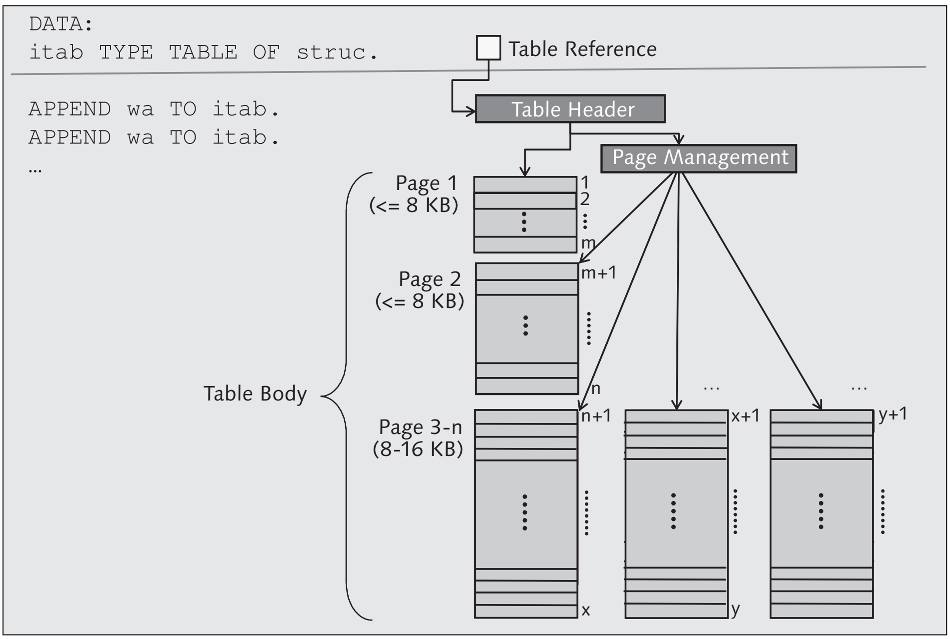


Figure 7.2 Schematic Diagram of the Organization of the Internal Tables in the Main Memory

The table header has a reference to the first page of the table body and another reference to page management. Page management manages the addresses of the pages in the main memory.

The table reference currently occupies 8 bytes of memory space. The table header occupies about 100 bytes of memory space depending on the platform. The space required for page management depends on the number of pages.

The table body consists of pages that can include the table rows. The first two pages are — depending on the row length and other factors — usually smaller than the pages 3 to *n* (if the row lengths are not so long that the maximum page size is reached already at the beginning).

As of the third page, the pages are created with the maximum page size, which is usually between 8 KB and 16 KB. This depends on the length of the row. Unlike database tables, the access is not per page but per row. So if you access a row of an internal table, the system reads only one row. The effort for searching table entries (or data records) is comparable to the database tables. For this purpose, the index or hash administration provides support for the internal tables. You learn more about internal tables in Section 7.3, Table Types, for the table types because they are directly related to this topic.

The table header includes the most important information about an internal table. For example, you can quickly query the number of rows using `DESCRIBE TABLE <itab> LINES <lines>` or the integrated function, `LINES( itab )`, from the table header.

As very small internal tables with only a few rows can result in wastage due to the memory use of the automatically calculated first page, `INITIAL SIZE` is added for the declaration of internal tables. It can provide information on the size of the first page, so a smaller memory allocation than in the standard case occurs.

However, if considerably more rows are required than originally specified for `INITIAL SIZE`, the third page is created faster with the maximum page size. For example, if 4 was specified for `INITIAL SIZE`, the third page may already be required as of the 13th row if the second page is twice as large as the first page. Relatively few rows (13, for example) require relatively much memory (three pages, third page with a size of 8 to 16 KB), whereas one page would have been sufficient if a higher value (for example, 14) had been specified for `INITIAL SIZE`. Consequently, for small tables it is important that `INITIAL SIZE` is not selected too small. Select a value that provides sufficient space in the first (or first and second) page for most cases.

`INITIAL SIZE` should always be specified if you require only a few rows and the internal table exists frequently. For nested tables, if an internal table is part of a row of another internal table, this is likely for the inner internal table. It can also occur for attributes of a class if there are many instances of this class.

Caution: INITIAL SIZE and APPEND SORTED BY

In conjunction with the `APPEND wa SORTED BY comp` command, the `INITIAL SIZE` addition not only has a syntactic but also a semantic meaning (see documentation). However, don't use the `APPEND wa SORTED BY comp` command; instead, work with the `SORT` command.

Depending on the table type or type of processing, you also require a management for the access to the row, that is, an index for the index tables and a hash administration for the hashed tables. At this point, memory may be required for the management of entries in addition to the pages. This management also occupies memory. Both in the Debugger and in the Memory Inspector, this memory is added to the table body and not displayed separately. Compared to the user data, this management can generally be neglected.

But how can you release allocated space in the internal tables again? The deletion of individual or multiple rows from the internal table using the `DELETE itab` command doesn't result in any memory release. The rows concerned are only "selected" as deleted and not deleted from the pages.

Only when you use the `REFRESH` or `CLEAR` statements the system does release the pages of the internal tables again. Only the header and a small memory area remain.

Note

In this context, *released* means that the occupied memory can be reused. As the memory allocation from the Extended Memory (EM) for a user is usually done in blocks (see Section 6.1 in Chapter 6), which are considerably larger than the pages of an internal table, this is referred to as a two-level release. Release initially means that the pages within an EM block are released and this space can then be reused by the *same* user. Only if the EM block is completely empty and doesn't contain any data (variables, and so on) of the user any longer is this block returned to the SAP memory management and available for the other users again.

The `FREE itab` ABAP statement, however, results in the complete de-allocation of the table body, that is, *all pages* and the index (if available) of the internal tables are released. Additionally, the table header is added to a system-internal "free list" for reuse.

If an internal table should be reused, it is advisable to use `REFRESH` or `CLEAR` instead of `FREE` because this way the creation of the first page can be omitted. If a large part of the rows of an internal table was deleted using `DELETE` and the occupied memory should be released, it is recommended to copy the table rows. A simple copy to another internal table is not sufficient because of table sharing, which is discussed in Section 7.4, Performance Aspects. Alternatively, you can revert to ABAP statements (`INSERT` or `APPEND`) or to the `EXPORT/IMPORT` variants (see Section 6.2.2 in Chapter 6) for copying. In the context of performance, the "release" of memory only plays a secondary role (as long as no memory bottleneck exists in the system). In contrast to fragmented database tables, fragmented internal tables have no negative effects on the performance because the entries can always be addressed efficiently because internal tables are always managed per row.

Background: Difference between Internal Tables and Database Tables

Internal tables can be compared to database tables in many respects, but there is one major difference:

Internal tables are always processed on a row basis, whereas database tables are always processed on a set basis. A set-based processing, possible with Open SQL on database tables, is not possible on internal tables because the single row is the main processing criterion for internal tables, whereas a set of data records is the main processing criterion for database tables. Set-based accesses to internal tables, for instance, `LOOP ... WHERE` or `DELETE ... WHERE`, are emulated by the ABAP VM and can be mapped in an optimized way for some table types (see Section 7.4, Performance Aspects). More complex, set-based operators, such as joins and aggregates,... are not possible on internal tables. They must be programmed using the existing ABAP language techniques.

After you've learned about the organization of internal tables in the main memory, the next section focuses on the organization of internal tables and discusses the different types of internal tables.

7.3 Table Types

Internal tables can be subdivided into index tables and hashed tables. The index tables, in turn, can be divided into standard tables and sorted tables. Figure 7.3 shows an overview of the table types.

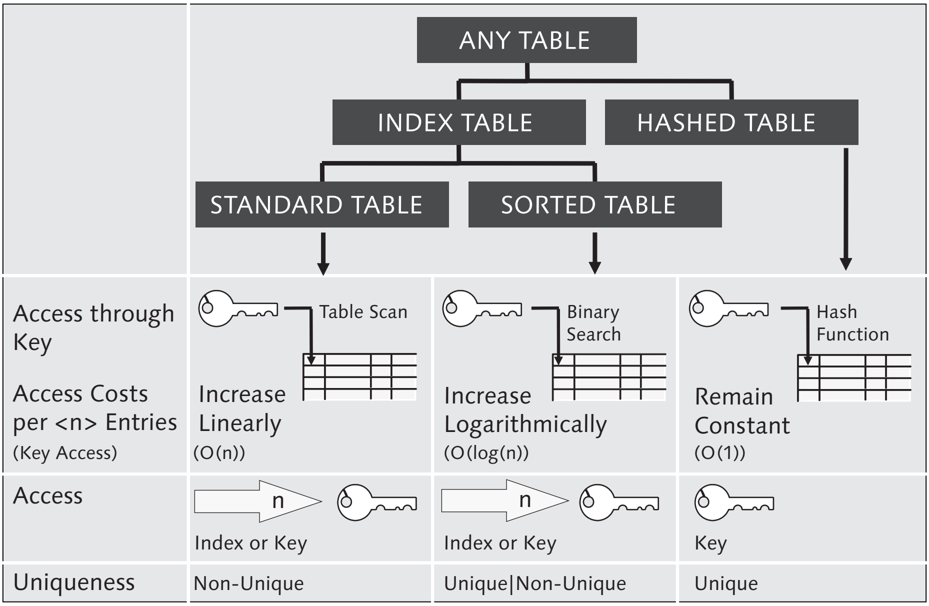


Figure 7.3 Overview of the Table Types

The table type specifies how you can access individual table rows via ABAP.

For *standard tables,* the access can be implemented via the table index or a "key." For a key access, the response time depends linearly on the number of table entries because the read access corresponds to a linear scan of the entries, which is canceled after the first hit. The key of a standard table is always non-unique. If no key is specified, the standard table receives the *default key,* which is a combination of all character-like fields.

Sorted tables are always sorted by the key. The access can be carried out via the table index or the key. For a key access, the response time depends logarithmically on the number of table entries because the read access is carried out via a binary search. The key of sorted tables can be unique or non-unique. On sorted tables, you can process partial keys (initial parts of the complete key) in an optimized manner. An over-specification of the table key is also possible; only the components of the key are used for the search and the remaining components are then utilized for the filtering.

Standard tables and sorted tables are also referred to as *index tables* because both tables can be accessed using the table index.

The read access to *hashed tables* is only possible by specifying a key. Here, the response time is constant and doesn't depend on the number of table entries because the access is carried out via a hash algorithm. The key of hashed tables must be unique. Neither explicit nor implicit index operations are permitted on hashed tables. If a hashed table is accessed with a "key" that is different to the unique table key, the table is handled like a standard table and searched linearly according to the entries. This is also the case for a partial key. Different to the sorted table, this partial key cannot be optimized for the hashed table. Over-specified keys are processed in an optimized manner.

By means of the `DESCRIBE TABLE <itab> KIND <k>` statement, you can determine the current table type at runtime. Of course, this is also possible using Run Time Type Identification (RTTI).

An index or a hash administration is available for the efficient management or access optimization of internal tables. The following section describes which types are available and when they are created.

Index Tables

Indexes for index tables are only created when the physical sequence no longer corresponds to the logical sequence, that is, when one of the `INSERT`, `DELETE`, or `SORT` statements is executed on the table and the following conditions apply:

1. **`INSERT`**
   The entry to be inserted should be inserted before an already existing entry. (An `INSERT` statement that inserts *behind* the last record largely corresponds to an `APPEND` statement.)
2. **`DELETE`**
   The entry to be deleted is not the last entry of the table.
3. **`SORT`**
   The table has a certain size and is sorted.

An index is used for the efficient index access in the "logical sort sequence" or the efficient finding of "valid rows" if the table pages have gaps due to deletions. By means of the index, the logical sequence of the table is mapped on the physical memory addresses of the entries.

An index is available in two types:

1. As a linear index
2. As a tree-like index

The index structure is always maintained without any gaps, whereas the table pages may have gaps due to the deletion of records. In comparison to the management of the index without gaps, the management of the table pages without gaps would be too time consuming for larger tables.

Due to the management of the index structure without gaps, the insertion and deletion of records incur movement costs because the existing entries must be moved. Strictly speaking, these costs are overheads for copying. For large indexes (as of about 5,000 entries), they get dominant; this is why a tree-like index is created for large tables.

In addition to the index, a free list exists that manages the addresses of the entries that were deleted using `DELETE` for reuse.

Figure 7.4 shows a schematic diagram of a linear index.

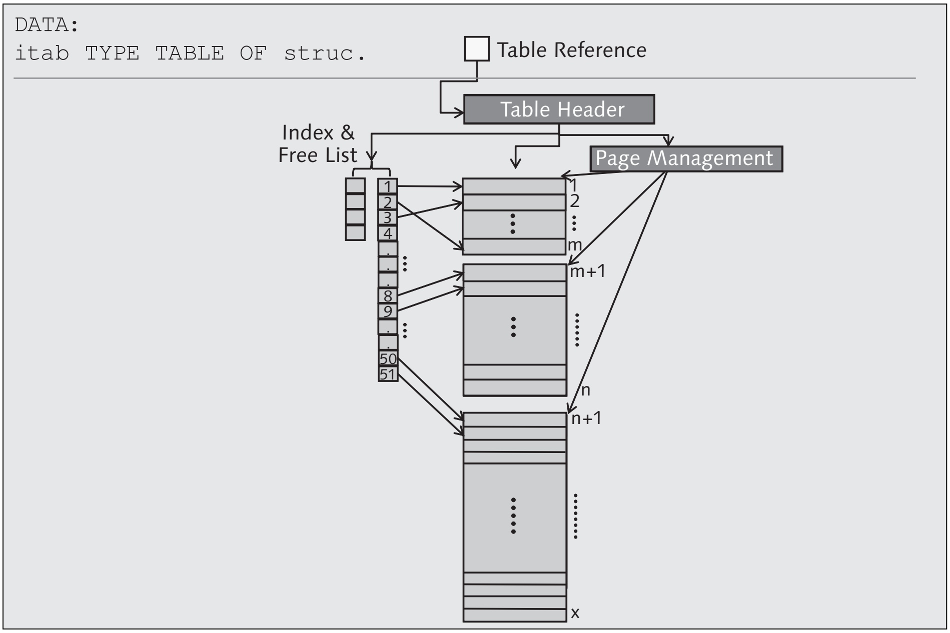


Figure 7.4 Schematic Diagram of a Linear Index

Whether a tree-like index is created depends on system-internal rules, for example, the number of entries (to be expected), and other factors. Figure 7.5 shows a schematic diagram of a tree-like index.

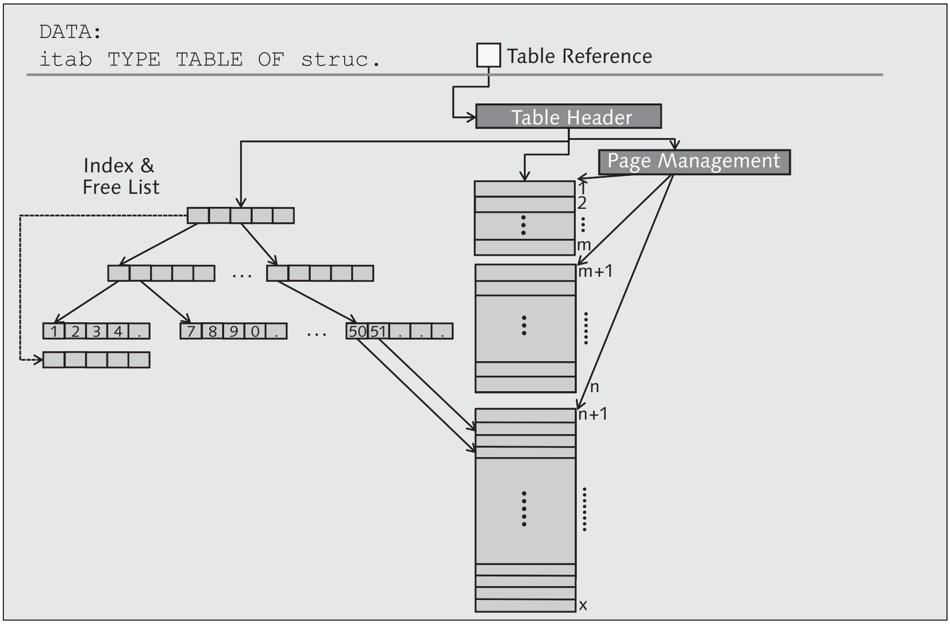


Figure 7.5 Schematic Diagram of a Tree-Like Index

For the tree-like index, the index entries are organized in *leaves*. The previously mentioned movement or copy costs only incur at the leaf level. The index doesn't have to be allocated at once; you only require continuous memory at leaf level. In return, you must first navigate through the tree structure when you access the index to reach the respective index entry.

Apart from that, the tree-like indexes on index tables are comparable to the database indexes presented in Chapter 5. A tree-like index requires about 50% more space than a linear index.

If the logical sequence of entries corresponds to the physical sequence in the main memory when the index tables are processed, you don't need to create an index. In this case, the insertion sequence corresponds to the physical sequence, and the table was filled with a sorting and not deleted or sorted. If no index is necessary, the internal table requires less memory.

Hash Administration

The hash administration is based on the unique key of the table. The hash administration is created for hashed tables only. It is established using the unique key of the internal table. Index accesses (for example, second entry of the internal table) are not possible, hashed tables can only be accessed with the key.

For the hashed table, each key value is assigned to a unique number using a hash function. For this number, the memory address of the respective data record is stored in a corresponding hash array.

If a `DELETE` or `SORT` is executed on a hashed table, you must create a double-linked list (previous and next pointer), so sequential accesses (`LOOP`) via the data are still possible according to the insertion sequence (or in a sort sequence generated using `SORT`). The double-linked list requires about 50% more space for the hash administration.

Figure 7.6 shows a schematic diagram of a hash administration.

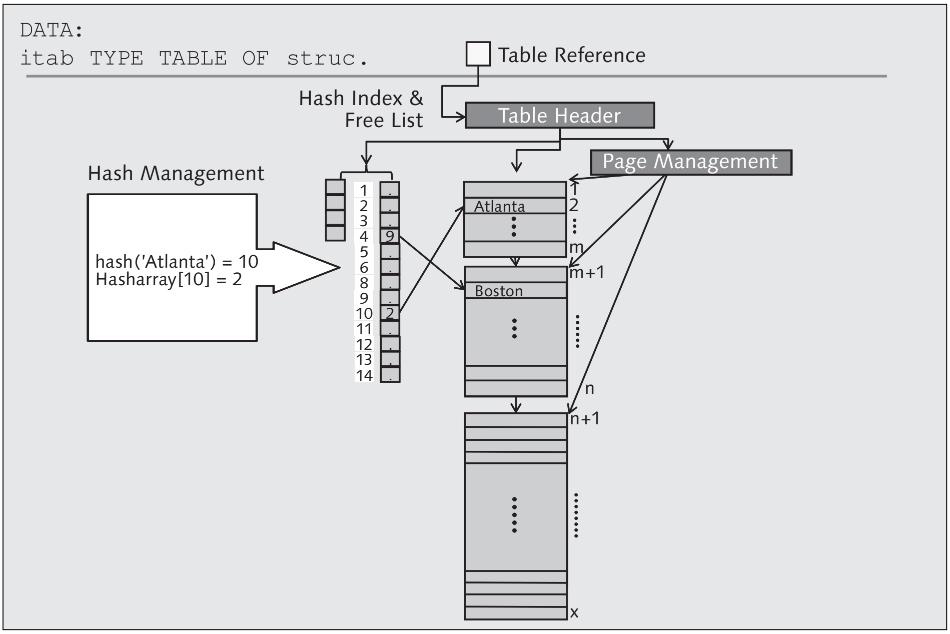


Figure 7.6 Schematic Diagram of a Hash administration

Limitations

Besides the memory that is available to the user, there are further limitations for internal tables:

A limit for the number of rows in internal tables results because they are addressed internally and in ABAP statements via 4 byte integers, which limits them to 2,147,483,647 entries.

The size of hashed tables is further limited by the biggest memory block available at once. The maximum size is 2 GB, but it is usually further limited by the `ztta/max_memreq_MB` *profile parameter*. The maximum number of rows of hashed tables depends on the required size of the hash administration that must be stored there.

The actual maximum size of internal tables is usually smaller than specified by the previous limits because the overall available memory is usually not only used by a string or an internal table (see ABAP documentation: MAXIMUM SIZE OF DYNAMIC DATA OBJECTS).

Summary of the Table Types

Table 7.1 lists the most important characteristics of the table types. This is followed by a recommendation for when you should use which table type.

	Standard Table	Sorted Table	Hashed Table
Possible Accesses	Index access or key access	Index access or key access	Key access
Uniqueness	Non-unique	Non-unique or unique	Unique
Optimal Access	Index or binary search (if the table is sorted by the search components)	Index or key	Key

Table 7.1 Characteristics of Table Types

Standard tables should only be used if all entries should be processed sequentially after filling or if the internal tables should be accessed flexibly and efficiently using multiple different keys. For this purpose, the table must be sorted by the search field and scanned using the binary search. The resorting is carried out only as often as necessary. If a resorting is only required for one or a few read accesses, the sort times far outweigh the time savings for reading. Use key accesses without binary search only for small tables or better avoid them completely. If you search only via a specific field, use a sorted or hashed table.

Sorted tables are particularly suited for partially sequential processing, for example, when a small part of a table should be processed via key accesses for which only the initial part of the key is given. Key accesses to the table key can also be carried out efficiently by the sorted tables.

Hash tables are optimal if you access only via the table key. If the key has a high left significance, you can also use a unique sorted table because in this case performance benefits arise for the binary search when you access individual rows. In this context, *left significance* means that the selective part of a key should be positioned at the beginning of the key (as far to the left as possible).

7.4 Performance Aspects

This section discusses all performance-relevant aspects when working with internal tables. For this purpose, the most important commands for internal tables are discussed. The examples are indicated with a work area (`wa`). Processing with header lines is still supported but should not be used any longer because the header lines of internal tables are obsolete and prohibited in the OO context.

7.4.1 Fill

Like for the database accesses, array operations and single record operations are also available for the internal tables.

Array Operations

ABAP documentation generally describes this type of processing as block operation, whereas the `SELECT` statement uses the term array operation with regard to the database.

When internal tables are filled from database tables, the `INTO TABLE itab` keyword causes the `SELECT` statement to insert the data records en bloc to the internal tables (see Section 5.7 in Chapter 5).

An array interface is also available for filling internal tables from other internal tables. The corresponding ABAP statements are:

```
APPEND LINES OF itab1 TO itab2.
INSERT LINES OF itab1 INTO  TABLE itab2.
```

For hashed tables, you can only use the `INSERT` statement, and for index tables you can use both `APPEND` and `INSERT`. If you append rows using `APPEND`, for sorted tables you must ensure that the sort sequence of the internal tables is maintained.

Assignments using `MOVE` and = also belong to the array operations to internal tables.

Here, minor runtime differences arise between the table types, which depend on the insertion position and the quantity of inserted entries. The management of indexes incurs relatively low costs.

Prefer array operations on internal tables to single record operations (next section) wherever possible because the kernel can process administrative work (for example, memory allocation) more efficiently.

Note that in contrast to the database tables the sequence of the rows in internal tables is always well defined:

- For duplicates and non-unique keys, the sequence in the target table and within the duplicates in the source table will always be the same for array operations. This is not the case for single record operations; here, the sequence of the duplicates can change.
- For duplicates and unique keys, the block operations result in non-catchable runtime errors, whereas the single record operations only set the `sy-subrc` return code.

Real-Life Example — Transaction SE30, Tips & Tricks

In the Tips & Tricks under Internal Tables • Array `Operations`, Transaction SE30 provides various examples whose runtime you can measure.

Single Record Operations

The ABAP statements, `APPEND` and `INSERT`, are also available for the single record operations:

```
APPEND wa TO itab.
INSERT wa INTO itab INDEX indx.
INSERT wa INTO TABLE itab.
```

Whereas you can use an `APPEND` and an `INSERT` statement with the `INDEX` addition only in index tables, the third variant is available for all tables.

For standard tables, an `INSERT` statement without `INDEX` mostly corresponds to the `APPEND` statement. (For `APPEND`, the row to be appended must be convertible, while for `INSERT`, the row to be inserted must be compatible; see ABAP documentation.) The costs for the `APPEND` statement are constant. An `APPEND` is the fastest variant for inserting single records because in this process only one entry is appended to the end of the table.

The insertion at a specific position (`INSERT ... INDEX`) incurs movement costs depending on the insertion position. These costs increase the "closer" the entry is inserted to the beginning (more movement costs) and decrease the "farther" the entry is inserted to the end (less movement costs). Up to a certain limit (currently 4,096), the costs for inserting depend on the insertion position and linearly on the number of entries. As soon as the index table has more entries, the system switches to a tree-like index internally in which the movement costs and the insertion position are only relevant at leaf level. When a tree-like index is present, the costs don't scale linearly any longer but logarithmically with the number of entries.

An insertion with an index for the standard tables is useful to structure them in a sorted manner. For this purpose, you must first determine the correct insertion position if it is not known. The best way to achieve this is by using a binary search (see next section).

For sorted tables, you can only use an `APPEND` and an `INSERT` statement with the `INDEX` addition if the sort sequence remains unchanged. In this case, you must check whether the key of the new entry is suitable for the desired position in the table.

A binary search is carried out for a generic `INSERT` (without the `INDEX` addition), which determines the correct insertion position internally. The costs for finding the position correspond to a read access to this table using a key and scale logarithmically with the number of entries. Like for the standard table, movement costs also occur. These costs depend on the insertion position and the index (linear or tree-like).

For hashed tables, the insertion is based on the table key. The costs are constant here and don't depend on the number of entries. Using the hash administration is somewhat more complex than appending entries to the standard table.

In summary, use array operations for insertion wherever possible. However, note the previously mentioned behavior of these operations.

Table 7.2 provides an overview of the costs for the single record statements. The costs for the reorganization of the index or hash administration when extending the internal memory or managing the tree-like index are not considered here.

	Standard	Sorted	Hashed
`APPEND`	O(1) Constant	O(1) Constant (higher than standard, check required)	–

Table 7.2 Costs of Single Record Operations for Filling Internal Tables

	Standard	Sorted	Hashed
`INSERT ... INTO ... INDEX`	Linear index: O(1) – O(n) Constant—linear Tree-like index: O(1) – O(log n) Constant—logarithmic (depending on the position)	Linear index: O(1) – O(n) Constant—linear Tree-like index: O(1) – O(log n) Constant—logarithmic (depending on the position) Constant (a bit higher, check required)	–
`INSERT ... INTO TABLE`	O(1) Constant	O(log n) Logarithmic	O(1) Constant (higher than standard, hash administration)

Table 7.2 Costs of Single Record Operations for Filling Internal Tables (Cont.)

7.4.2 Read

For read accesses, you differentiate reading of multiple and individual rows.

Multiple Rows (LOOP)

Here, you differentiate between reading all rows and reading a specific section of rows.

All rows are read using the `LOOP AT itab` ABAP statement. In this process, all rows of an internal table are read. The costs for reading all data records scales linearly with the number of data records. These costs are independent of the table type because each entry in the internal table must be processed. Without any further specification, each entry is *copied* into the work area specified with `INTO`.

A part of the rows in an internal table is read with `LOOP ... FROM ix1 TO ix2` (for index tables) or generally with `LOOP ... WHERE`. The costs for reading a subarea of the internal table depend on the size of this part and whether the part to be read can be found efficiently. Costs for providing the resulting set in the output area

(`LOOP ... INTO`) accrue. However, the costs for finding the relevant entries are far more important.

For standard tables, the costs are linear to the number of entries.

For hashed tables, you can implement a search via the hash administration if the complete key of the table is specified in the `WHERE` condition. Then the `LOOP ... WHERE` corresponds to a read of a unique record. The costs are constant then. In all other cases, the read accesses to the hashed table are linear, depending on the number of entries because the table is searched completely.

When you access sorted tables, the kernel can optimize an incomplete key because the table is available in a sorted manner by definition. For this purpose, the following conditions must be met:

1. The `WHERE` condition has the form, `WHERE k1 = b1 AND ... AND kn = bn`.
2. The `WHERE` condition covers an initial part of the table key.

In contrast to hashed tables, partially sequential accesses are optimized for sorted tables, too. This way, you can find the starting point for the searched area in an efficient manner.

If standard tables are sorted by the key, you can also achieve an optimization by first searching for the first suitable entry using the binary search and then starting a loop from this position. This loop is exited as soon as the system determines with an `IF` statement that the search condition no longer applies. The costs for this procedure correspond approximately to the costs of the sorted table and scale logarithmically to the table entries. The following listing provides a pseudo code example for this procedure:

```
READ TABLE itab INTO wa WITH KEY ... BINARY SEARCH.
   INDEX = SY-TABIX.
   LOOP AT itab INTO wa FROM INDEX.
   IF ( key <> search_key ).
      EXIT.
   ENDIF.
   ENDLOOP.
```

Mass access incurs the following costs, which are also listed in Table 7.3. The costs include both search costs for finding the relevant entries (as shown in the table) and costs for providing the hit list (for example, in the work area or a data reference). The costs for providing the hit list are of secondary importance in case of small hit lists. Only for `LOOP ... FROM ... TO`, for which the search costs are constant, can the provision of the hit list dominate the costs.

For longer hit lists or the extreme case that all rows of the internal table are included in the hit list due to duplicates relating to the key, the costs on index tables are dominated by the provision costs, which scale linearly with the number of hits.

	Standard	Sorted	Hashed

`LOOP ...`
`ENDLOOP`
(all rows)
	O(n)
	Linear (full table scan)
		O(n)
		Linear (full table scan)
			O(n)
			Linear (full table scan)

`LOOP ... WHERE`
`ENDLOOP`
(complete key)
	O(n)
	Linear (full table scan)
		O(log n)
		Logarithmic
			O(1)
			Constant

`LOOP ... WHERE`
`ENDLOOP`
(incomplete key, initial part)
	O(n)
	Linear (full table scan)
	Can be optimized manually using a sorted standard table and a binary search O(log n).
		O(log n)
		Logarithmic
			O(n)
			Linear (full table scan)

`LOOP ... WHERE`
`ENDLOOP`
(incomplete key, no initial part)
	O(n)
	Linear (full table scan)
		O(n)
		Linear (full table scan)
			O(n)
			Linear (full table scan)

`LOOP ... FROM` `... TO`	O(1) Constant	O(1) Constant	–

Table 7.3 Costs for Reading Multiple Rows from Internal Tables

Single Rows

The following statements are available to read single rows from internal tables:

```
READ TABLE itab INTO wa INDEX ...
READ TABLE itab INTO wa WITH [TABLE] KEY ...
READ TABLE itab INTO wa FROM wa1
```

Index accesses can only be executed on index tables and have constant costs.

Usually, you want to access an internal table using the key and not using the index. In this case, the costs depend on the effort required to find the correct entry.

For standard tables, the costs depend linearly on the number of entries because the table is scanned entry by entry until the proper entry is found. If the entry is positioned at the beginning of the table, the search finishes earlier than if the entry is positioned at the end of the table.

The use of the binary search is an option to accelerate the search in a standard table. For this purpose, the standard table must be available sorted by the search term and an initial part of the sort key must be provided. With the `READ itab WITH KEY ... BINARY SEARCH` statement, a binary search is used for the standard table. In this case, the costs scale logarithmically with the number of entries.

Background: Binary Search

The binary search on standard or sorted tables uses the bisection method. For this purpose, the table must be available sorted by the respective key. Here, the search doesn't start at the beginning of the table but in the middle, and then the half that contains the entry is bisected again, and so on, until a hit is available or no record can be found. If duplicates exist, the first entry is returned in the duplicate list.

Ensure that the standard table is not sorted unnecessarily because the sorting of a standard table is also an expensive statement (see Section 7.4.6, Sort); for this reason, the number of sorting processes must be kept as small as possible.

Real-Life Example — Transaction SE30, Tips & Tricks

In the Tips & Tricks under Internal Tables • Linear Search vs. Binary Search, Transaction SE30 provides an example whose runtime you can measure.

The binary search can also be used for the optimization of partially sequential accesses as shown at the beginning of this section for the `LOOP ... WHERE` to standard tables. You can also use a binary search to establish a standard table in a sorted manner. For this purpose, have another look at the example from Section 6.2.1 in Chapter 6:

```
READ TABLE it_kunde INTO var_kunde
WITH KEY it_order_tab-kunnr BINARY SEARCH.
save_tabix = sy-tabix.
IF SY-SUBRC <> 0.
       SELECT *
       INTO var_kunde
       FROM db_kunden_tab
       WHERE kundennr = it_order_tab-kunnr.
          IF SY-SUBRC = 0.
            INSERT var_kunde INTO it_kunde INDEX save_tabix.
           ...
```

The `it_kunde` table is scanned for a suitable entry using the binary search. If no suitable entry can be found, the `sy-tabix` table index is positioned on the row number on which the entry is. You can use this index to insert the entry at the correct position. This way, the standard table is organized in a sorted manner without requiring a `SORT` statement.

For read accesses to sorted tables, a binary search is used internally if an initial part of the table key is available. The costs scale logarithmically with the number of entries.

For hashed tables, the hash administration is used in case a fully specified key access exists. The costs are constant then. If the system accesses the hashed table with a key that is not fully specified, the costs depend linearly on the number of entries.

For all accesses, it is irrelevant for the performance whether the access is carried out using the table key (`...WITH TABLE KEY...`) or a free key `(...WITH KEY...)`. The only decisive factor for the performance is that the key fields referred to comply with the beginning or the entire table key. So, over-specified keys (with more fields than the key fields) can also be used to optimize to internal tables.

Single record access incurs the costs listed in Table 7.4. As already mentioned for `LOOP ... WHERE`, a linear share is added for duplicates in the binary search for the index tables, which can exhibit a linear runtime behavior in extreme cases (all entries relate to the duplicates key).

	Standard	Sorted	Hashed
`READ ... INDEX`	O(1) Constant	O(1) Constant	–
`READ ... WITH KEY ...` (Complete key)	O(n) Linear Binary search: O(log n) Logarithmic	O(log n) Logarithmic	O(1) Constant
`READ ... WITH KEY ...` (Incomplete key, initial part)	O(n) Linear Binary search: O(log n) Logarithmic	O(log n) Logarithmic	O(n) Linear
`READ ... WITH KEY ...` (Incomplete key, no initial part)	O(n) Linear	O(n) Linear	O(n) Linear

Table 7.4 Costs for Reading Single Rows from Internal Tables

7.4.3 Modify

Internal tables are changed using the `MODIFY` command. `MODIFY` to internal tables *only* involves a change and not a change or insertion as is the case for the `MODIFY` command to a database table.

Multiple rows of an internal table are modified with the following statement:

```
MODIFY itab FROM wa TRANSPORTING ... WHERE ...
```

The costs are the same as for the `LOOP ... WHERE` statement and depend on the number of entries to be modified and the effort for finding the entries.

Single entries in internal tables can be modified as follows:

```
MODIFY itab [INDEX n] [FROM wa]
MODIFY TABLE itab [FROM wa]
```

The costs are constant if you access index tables via the index (variant 1). Within loops, you can also use this variant for the sequential modification of multiple rows without `INDEX`. In this case, the current row where the loop is used is modified. This is an implicit index operation that is only permitted for index tables.

For the key accesses (variant 2) with a complete key, the costs scale linearly for standard tables and logarithmically with the number of entries for sorted tables. The costs are constant for hashed tables. Because this variant includes a separate search of the proper entries, it shouldn't be used in the loop via the same table. This could result in a nonlinear runtime behavior.

The costs for `MODIFY` correspond to those of the `LOOP`; the same restrictions apply for the duplicates (see Table 7.5).

	Standard	Sorted	Hashed
`MODIFY ... TRANSPORTING ... WHERE` (complete key)	O(n) Linear (full table scan)	O(log n) Logarithmic	O(1) Constant
`MODIFY... TRANSPORTING... WHERE` (incomplete key, initial part)	O(n) Linear (full table scan)	O(log n) Logarithmic	O(n) Linear (full table scan)
`MODIFY... TRANSPORTING... WHERE` (incomplete key, no initial part)	O(n) Linear (full table scan)	O(n) Linear (full table scan)	O(n) Linear (full table scan)

Table 7.5 Costs for Modifying Internal Tables

	Standard	Sorted	Hashed
`MODIFY ... [INDEX n] FROM` wa (index access)	O(1)	O(1)	–
`MODIFY TABLE... FROM wa` (search effort as for `WHERE`)	O(n) Linear (full table scan)	O(log n)	O(1) Constant

Table 7.5 Costs for Modifying Internal Tables (Cont.)

7.4.4 Delete

The following statements are available to delete multiple entries from internal tables:

```
DELETE itab FROM ix1 TO ix2
DELETE itab WHERE...
```

The costs depend on the effort for finding and the quantity of rows to be deleted. For the index access, the costs for finding are constant; for the key access, they correspond to the costs of `MODIFY`.

Accesses to individual entries are implemented using the following statements:

```
DELETE itab [INDEX n].
DELETE TABLE itab WITH TABLE KEY .../DELETE TABLE itab FROM wa
```

For the accesses to individual rows, the costs correspond to those of `LOOP` or `MODIFY` (see Table 7.6).

	Standard	Sorted	Hashed
`DELETE ... FROM ... TO`	O(1)	O(1)	–
`DELETE ... WHERE` (complete key)	O(n) Linear (full table scan)	O(log n) Logarithmic	O(1) Constant
`DELETE ... WHERE` (incomplete key, initial part)	O(n) Linear (full table scan)	O(log n) Logarithmic	O(n) Linear (full table scan)
`DELETE ... WHERE` (incomplete key, no initial part)	O(n) Linear (full table scan)	O(n) Linear (full table scan)	O(n) Linear (full table scan)

Table 7.6 Costs for Deleting Entries from Internal Tables

	Standard	Sorted	Hashed
`DELETE ... INDEX`	O(1)	O(1)	–
`DELETE FROM WA /` `DELETE TABLE WITH TABLE KEY`	O(n) Linear (full table scan)	O(log n)	O(1) Constant

Table 7.6 Costs for Deleting Entries from Internal Tables (Cont.)

7.4.5 Condense

Using the `COLLECT` command, you can create condensed datasets in internal tables. For this purpose, the numeric data of all fields that aren't key fields are added to already existing values with the same key in the internal table. For standard tables without explicit key specification, all non-numeric fields are handled as key fields. The costs of the command are significantly determined by the effort of finding the relevant row.

A temporary hash administration is created for standard tables if a standard table is filled with `COLLECT` only. This is rather unstable compared to other modifying statements (`APPEND`, `INSERT`, `DELETE`, `SORT`, `MODIFY`, changes using the field symbols/references). However, this optimization has become obsolete because of the implementation of key tables (sorted tables, hashed tables) and therefore the `COLLECT` command to standard tables, too.

If the temporary hash administration is intact, the finding of entries is a constant process just like for hashed tables. If the hash administration is destroyed, the effort for searching entries depends linearly on the number of entries in the internal table. You can use the `ABL_TABLE_HASH_STATE` function module to check whether a standard table has an intact hash administration.

For sorted tables, the entry is specified internally using a binary search, whereas the effort for searching entries depends logarithmically on the number of entries in the internal table.

In hashed tables, the entry is determined using the hash administration of the table. The costs are constant and don't depend on the number of entries.

`COLLECT` should be used mostly for hashed tables because they have a unique table key and a stable hash administration.

Real-Life Example — Transaction SE30, Tips & Tricks
In the Tips & Tricks under Internal Tables • Building Condensed Tables, Transaction SE30 provides an example whose runtime you can measure.

7.4.6 Sort

Standard and hashed tables can be sorted by any field of the table using the `SORT` command. Sorted tables cannot be sorted using the `SORT` command because they are already sorted by the key fields by definition and cannot be resorted by other fields.

During the sorting process, the data is sorted in the main memory (in the process-local memory of a work process) if possible. If the space in the main memory is not sufficient, the components are sorted in the file system. For this purpose, the blocks are first sorted in the main memory and then written to the file system. Subsequently, these sorted blocks are reimported using a merge sort.

Sorting is a runtime-intensive statement regardless of whether the sorting is implemented in the main memory or in the file system. (Of course, the sorting in the file system is even more expensive than the sorting in the main memory.) Therefore, only sort if this is absolutely required by the application or, in the case of the standard table, if you can achieve runtime gains for the reading from these tables using the binary search. For example, it is possible to sort an internal standard table first by one key field and then by another one and to browse it using the binary search. In this case, the achieved runtime gains via the binary search are not canceled out by the increased effort of sorting. The sorting is only worthwhile if you can optimize a large number of subsequent read accesses this way. For a table with about 1,000 rows, a sorting process should be followed by at least 40 to 50 read accesses.

If the internal standard table is processed in such a way that a search access to a field is implemented alternately to a search access to another field, and consequently a sorting process for the respective resorting would be necessary for each search access, it would be counterproductive to carry out the sorting. In this case, only optimize one of the two search processes by means of a one-time sorting and a binary search. Optionally, you could consider the use of a second internal table, which acts as a secondary index (see Section 7.4.8, Secondary Indexes).

Note

The assignments in sorted tables could also require implicit sorting processes if these have a key that is different to the source table. These sorting processes are not evident in the ABAP trace directly because assignments are not assigned to events and are not recorded separately. The time required for these sorting processes is added to the net times of the calling modularization unit.

7.4.7 Copy Cost-Reduced or Copy Cost-Free Access

If you use the `LOOP ... WHERE` and `READ` statements, the results are *copied* to the work area. If you use the `MODIFY` statement, the changes are copied from the work area back to the table.

In case of `READ` and `MODIFY`, the costs for copying can be limited to the required fields. For this purpose, you must specify the `TRANSPORTING f1 f2 ...` addition. Then only the fields are copied, which are indicated after the addition. You can also avoid the costs for copying for `LOOP ... WHERE` and `READ` if you specify a `TRANSPORTING NO FIELDS`. In this case, the system fills only the corresponding system fields and no result is copied to the header or the work area. This is used to check whether a specific entry is available in an internal table. For `LOOP ... WHERE`, this access corresponds to a read access instead.

You can also avoid the costs for copying if the reference to a table row is copied to a reference variable or if the memory address of a row is assigned to a field symbol. Figure 7.7 illustrates this.

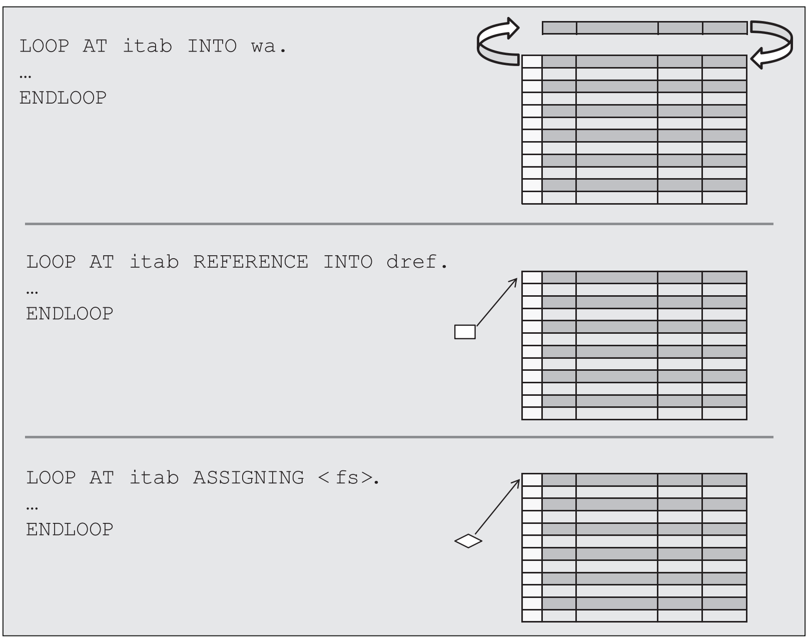


Figure 7.7 Schematic Diagram of the LOOP Variant

The first variant, `LOOP AT itab INTO wa`, copies the `itab` internal table row by row into the `wa` work area. If the row should be modified, you must copy it back using `MODIFY` (see Section 7.4.3, Modify).

The second variant, `LOOP AT itab REFERENCE INTO dref`, provides the memory address of each row — row by row — to the `dref` data reference variable.

The third variant, `LOOP AT itab ASSIGNING <fs>`, assigns the memory address of each row to the `<fs>` field symbol, again row by row.

The second and the third variant are more efficient due to the reduced cost for copying. For large datasets, the runtime can be reduced by means of these options. For very small datasets — when the internal tables have less than five rows and no excessively long rows (more than 5,000 bytes) — the regular copy process is faster because both the management of the data reference variable and the field symbols constitute a certain overhead for the system. In case of nested internal tables (internal tables in which a column of the row structure is another table), it is always worthwhile to use the copy-free techniques. If the changes to the row in the internal table should be written back, it pays off to use the copy-free techniques because you don't require the `MODIFY` command any longer.

The basic rule here is that the larger the dataset to be copied, the more worthwhile it is to use the copy-free techniques.

An access to *one* entry via `LOOP ... WHERE` or `READ` is suitable for wide rows (more than 1,000 bytes). If the read row should be modified and written back into the table (`MODIFY`), the copy-free access already pays off for shorter rows.

Real-Life Example — Transaction SE30, Tips & Tricks

In the Tips & Tricks under Internal Tables • Using the Assigning Command • Modifying a Set of Lines Directly, Transaction SE30 provides an example whose runtime you can measure.

7.4.8 Secondary Indexes

Up to and including Release 7.0 EhP1, internal tables cannot include secondary indexes. If you require efficient accesses via different fields, secondary indexes are implemented in the form of custom internal tables. In this process, an additional internal table is created for each secondary key, which includes a reference to the main table in addition to the field that represents the secondary key. This reference can be the position of the data record in the main table (only for index tables) or the key in the main table. But you can also define a separate unique number for it. All solutions entail additional memory requirement but allow for an efficient access via multiple key fields in return. When processing the internal table, you must ensure with utmost accuracy that the secondary index tables are maintained with every change of the main table. Generally, such a procedure is error prone because of its complexity and should be used in special situations only.

Real-Life Example — Transaction SE30, Tips & Tricks

In the Tips & Tricks under Internal Tables • Secondary Indices, Transaction SE30 provides an example whose runtime you can measure.

As of Release 7.0 EhP2 and 7.1, you are provided with secondary indexes which are described in Chapter 10.

7.4.9 Copy

Table sharing is another performance aspect that you should be aware of. For assignments and value transfers (import and export per value) of internal tables *of the same type*, whose row types don't contain a table type, only the internal administration information (table header) is transferred because of performance reasons. Figure 7.8 illustrates this.

Background: Internal Tables of the Same Type

Tables with the same structure are referred to as internal tables of the same type. Table sharing is possible between tables of the same type if the table in the target table has the same or a more generic type as the source table. The following combinations are possible, for example:

```
itab_standard = itab_sorted
itab_standard = itab_hashed
itab_sorted_with_nonunique_key = itab_sorted_with_unique_key
```

The sharing works for the same or a more general key of the target table (on the left-hand side of =).

In the following cases, the table sharing is not possible because the target table is not more generic than the source table:

`itab_sorted = itab_standard` (with same key definition)

`itab_sorted_with_unique_key = itab_sorted_with_nonunique_key` (with same key definition)

Table sharing is possible with any number of tables and cannot be influenced by the ABAP developer.

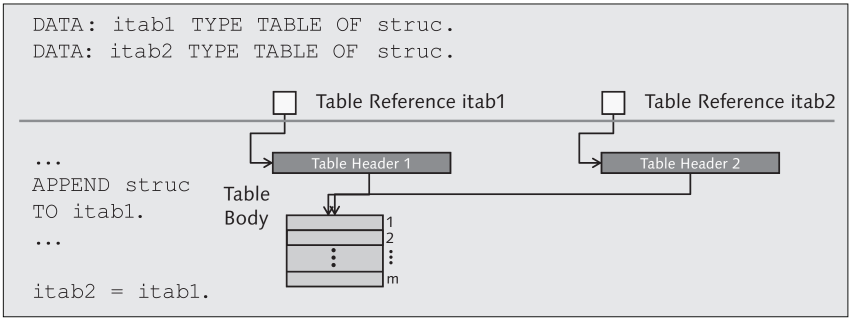


Figure 7.8 Table Sharing — Assignment

Table sharing is canceled if one of the internal tables involved in the sharing is modified. Only then does the actual copy process take place. Figure 7.9 shows the situation after the table sharing was canceled.

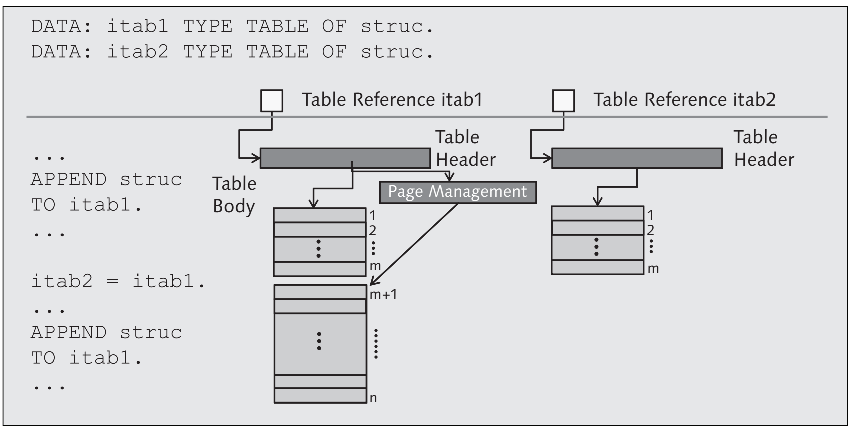


Figure 7.9 Canceled Table Sharing After Modification

The copy process after the cancellation of the table sharing (also referred to as *copy on write* or *lazy copy*) can result in situations that look "strange" at first:

For example, it can be possible that not sufficient memory is available if an entry of an internal table should be deleted because the table sharing can only be canceled in case of change accesses to one of the tables involved. Only then does the actual copy process take place. If sufficient memory is not available for the copy, then the short dump will inform you that not sufficient memory was available for executing the current operation (`DELETE`).

Another example is that a fast operation, such as an `APPEND` statement, can become eye-catching in the runtime measurement, because it has a considerably higher time than comparable operations. This may be due to the cancellation of the table sharing.

In principle, each changing access to an internal table can possibly cancel a previously existing table sharing. However, these are not additional but only deferred costs.

Change accesses to internal tables include the statements, `APPEND`, `INSERT`, `MODIFY`, `DELETE` but also assignments to fields or rows of tables implemented via data references or field symbols. A `DETACH` for shared objects also results in cancellation of table sharing. Likewise, the transfer of a table per value as a parameter of a method/function/form can cancel the sharing if the parameter is changed.

Table sharing is also displayed in the Debugger or in the Memory Inspector. In Figure 7.10 below the memory objects, the respective table headers point to the memory object. In this example, the internal tables, ITAB2A and ITAB1, are shared.

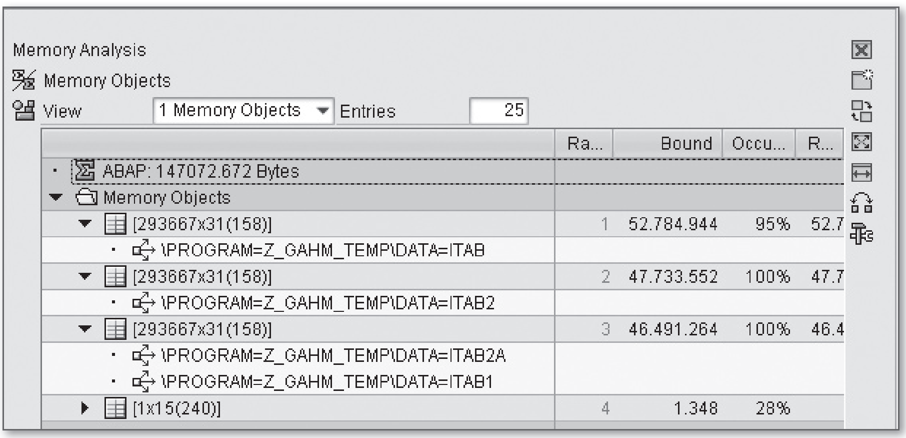
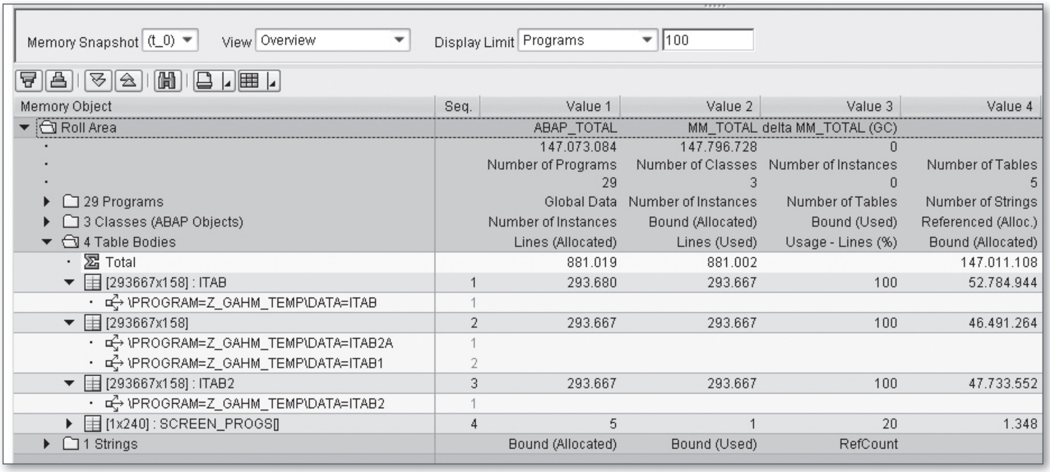


Figure 7.10 Table Sharing in the Debugger

In the Memory Inspector (see Figure 7.11), you can already view the name of the respective internal table next to the table bodies. Table bodies without a name (for example, the second table body in Figure 7.11) indicate shared tables. In this case, too, these are the internal tables, ITAB2A and ITAB1.



Memory Snapshot (t_0) View Overview Display Limit Programs 100

Memory Object	Seq.	Value 1	Value 2	Value 3	Value 4
Roll Area		ABAP_TOTAL	MM_TOTAL	delta MM_TOTAL (GC)	
·		147.073.084	147.796.728	0	
·		Number of Programs	Number of Classes	Number of Instances	Number of Tables
·		29	3	0	5
29 Programs		Global Data	Number of Instances	Number of Tables	Number of Strings
3 Classes (ABAP Objects)		Number of Instances	Bound (Allocated)	Bound (Used)	Referenced (Alloc.)
4 Table Bodies		Lines (Allocated)	Lines (Used)	Usage - Lines (%)	Bound (Allocated)
Total		881.019	881.002		147.011.108
[293667x158] : ITAB	1	293.680	293.667	100	52.784.944
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB	1				
[293667x158]	2	293.667	293.667	100	46.491.264
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB2A	1				
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB1	2				
[293667x158] : ITAB2	3	293.667	293.667	100	47.733.552
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB2	1				
[1x240] : SCREEN_PROGS[]	4	5	1	20	1.348
1 Strings		Bound (Allocated)	Bound (Used)	RefCount	

Figure 7.11 Table Sharing in the Memory Inspector

7.4.10 Nested Loops and Nonlinear Runtime Behavior

Inefficient accesses to internal tables have a particular impact in case of large datasets. The following little example shows a nested loop in which the respective orders of the customer are processed:

```
LOOP AT it_customers REFERENCE INTO dref_customer.
  LOOP AT it_orders REFERENCE INTO dref_order
  WHERE cnr = dref_customer->nr.
...
  ENDLOOP.
ENDLOOP.
```

Let's assume that the internal table, `it_customers`, has 1,000 entries. An average of two orders exists for each customer; consequently, the internal table, `it_orders`, has 2,000 entries. If these are standard tables, respectively, the two internal tables must be fully processed: the external table, `it_customers`, because no restriction exists and because all data records should be processed semantically; the internal table, `it_orders`, is restricted, but the corresponding entries for each customer cannot be searched efficiently. Therefore, the entire table, `it_orders`, must be browsed for the internal table. This is done for each entry of the external table, that is, 1,000 times in this example.

Let's assume that the external loop requires approximately 200 µs and the internal loop a total of 140,000 µs. If you now double the datasets, the runtime of *each* loop doubles as well because the two loops scale linearly with the number of entries. So, in case of 2,000 entries in the external table, `it_customers`, this results in ~400 μs and for 4,000 entries in the internal table, `it_orders`, in ~560.000 μs for all 2,000 runs. The internal table must be run through for each entry of the external table, but the system doesn't need to process all entries of the internal table for each external entry but only the two entries that belong to a customer.

As a result, the runtime is four times longer in case of a double dataset. The runtime behavior is not linear but quadratic. (The internal loop is twice as long as previously — scaled with n — and is executed twice as often as previously.)

In this case, the reason is an inefficient access to the inner internal table. To avoid this, you must optimize the access to the inner internal table. For a linear runtime behavior, the access to the inner internal table has to be constant, so the runtime doubles if the access frequency doubles. Because no unique key is possible in the previous example, you can achieve a logarithmic runtime behavior for the inner access using a sorted table. The sorted table allows for a binary search in the inner internal table and consequently ensures an efficient finding of the two suitable entries in the inner internal table for each entry of the outer table. The result of the entire code fragment is `O(n x log n)`.

At this point, a brief comparison to the nested loop join for databases (see Section 5.4.5 in Chapter 5): Like for the nested loop joins on databases, the number and the efficiency of the access to the internal table are significant for the optimization of nested loops.

Nonlinear runtime behavior is not always due to inefficient accesses to internal tables but can also result from a quadratic increase of the call frequency of an efficient access to an internal table, for example.

In general, the effects of nonlinear programming can be reduced by using smaller data packages. However, the packages should not be too small to not generate a too large overhead at other points (see Section 4.1.3 in Chapter 4).

Because in most cases only a small test dataset is available for the development of programs in the development system, it may occur that a nonlinear runtime behavior can only be discovered with difficulty because nested loops with small datasets only account for a smaller portion of the entire program runtime. For small test datasets, it often appears as if the program behaves linearly to the number of processed datasets.

To detect a nonlinear runtime behavior already during the development with small datasets, you must compare the runtime behavior at ABAP statement level. Here, the times for the accesses to internal tables with two variants — for example, with ten or with 100 data records to be processed — is measured and compared with one another using Transactions SE30 or ST12. This way, you can detect a nonlinear runtime behavior already with small datasets.

In Release 7.0 EhP1, there is no tool available that you can use to automatically implement this comparison. However, the following links of the SDN provide tools and descriptions of how you can automate such a comparison:

- Nonlinearity: The problem and background
  https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/5804
- A Tool to Compare Runtime Measurements: Z_SE30_COMPARE:
  https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/8277
- Report Z_SE30_COMPARE:
  https://www.sdn.sap.com/irj/sdn/wiki?path=/display/Snippets/Report%2bZ_SE30_COMPARE
- Nonlinearity Check Using the Z_SE30_COMPARE:
  https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/8367

In Release 7.00 EhP2, you can implement the comparison of trace files using the standard SAP means (see Chapter 10).

7.4.11 Summary

When you work with internal tables, the selection of the right table type and the access type is important.

Standard tables should only be used for small tables or tables that can be processed using the index accesses. For larger standard tables, ensure an efficient processing with the binary search if you want to process some parts of the table only. It can be used both for single record accesses and for mass accesses. If possible, the standard tables should be sorted in the same way or only sorted as often as absolutely necessary. For accesses to different key fields using the binary search, which requires a resorting, you must check whether the effort for sorting is justified (is amortized by the improved read accesses).

Sorted tables can be used uniquely or non-uniquely for most application cases. They are particularly useful for partially sequential processing in which the initial part of the table key is used, for example.

Hashed tables should only be used where the unique key access is the only type of access, particularly if you must process very large tables.

In general, internal tables should be filled using array operations if possible to avoid the overhead of single record accesses.

Wherever reasonable, reduce the copy costs for providing the results using `TRANSPORTING fieldlist/NO FIELDS` or completely avoid it by means of `ASSIGNING` or `REFERENCE INTO`. This is particularly essential for nested tables.

If possible, internal tables should not be too large to save the memory space of the SAP system.

Index

D

E

F

G

H

I

J

K

L

M

T

U

V

W