Disk Space Usage

The efficient use of disk space is especially important in a database environment since

- sharing data between several users, possibly concurrently and in different combinations, normally requires that a large proportion of an organization’s data be stored online; and
- some applications require extremely large amounts of data.

Decisions concerning the efficient usage of disk space must be made while considering other objectives of the system (performance, flexibility, ease of use). This section discusses the techniques and considerations involved in performing trade-offs between these objectives and the efficient usage of disk space.

This chapter covers the following topics:

- Data Compression
- Forward Index Compression
- Padding Factors

Data Compression

Each field may be defined to Adabas with one of three compression options:

- Fixed storage (FI), in which the field is not compressed at all. One-byte fields that are always filled (for example, *gender* in a personnel record) and alphanumeric or numeric fields with full values (for example, *personnel number*) should always be specified as fixed (FI) fields.
- Ordinary compression (the default) which causes Adabas to remove trailing blanks from alphanumeric fields and leading zeros from numeric fields;
- Null-value suppression, which includes ordinary compression and in addition suppresses the null value for a field. Adjacent null value fields are combined into a single value.

The following table illustrates how various values of a five-byte alphanumeric field are stored using each compression option.

Field Value	Fixed Storage	Ordinary Compression	Null-Value Suppression
ABCbb	ABCbb (5 bytes)	4ABC (4 bytes)	4ABC (4 bytes)
ABCDb	ABCDb (5 bytes)	5ABCD (5 bytes)	5ABCD (5 bytes)
ABCDE	ABCDE (5 bytes)	6ABCDE (6 bytes)	6ABCDE (6 bytes)
bbbbb	bbbbb (5 bytes)	2b (2 bytes)	* (1 byte)
X	X (1 byte)	2X (2 bytes)	2X (2 bytes)

The number preceding each stored value is an inclusive length byte (not used for FI fields). The asterisk shown under null-value suppression indicates a suppressed field count. This is a one-byte field which indicates the number of consecutive empty (suppressed) fields present at this point in the record. This field can represent up to 63 suppressed fields.

The compression options chosen also affect the creation of the inverted list for the field (if it is a descriptor) and the processing time needed for compression and decompression of the field.

Fixed Storage

Fixed storage indicates that no compression is to be performed on the field. The field is stored according to its standard length with no length byte. Fixed storage should be specified for small one- or two-byte fields that are rarely null, and for fields for which little or no compression is possible. Refer to the Adabas Utilities documentation the ADACMP utility, for restrictions related to the use of FI fields.

Ordinary Compression

Ordinary compression results in the removal of trailing blanks from alphanumeric fields and leading zeros from numeric fields. Ordinary compression will result in a saving in disk space if at least two bytes of trailing blanks or leading zeros are removed. For two-byte fields, however, there is no savings, and for one-byte fields, adding the length byte actually doubles the needed space. Such fields, and fields that rarely have leading or trailing zeros or blanks, should be defined with the fixed storage (FI) option to prevent compression.

Null-Value Suppression

If null-value suppression (NU) is specified for a field, and the field value is null, a one-byte empty field indicator is stored instead of the length byte and the compressed null value (see *Data Compression*). This empty field indicator specifies the number of consecutive suppressed fields that contain null values at this point in the record. It is, therefore, advantageous to physically position fields which are frequently empty next to one another in the record, and to define each with the null-value suppression option.

An NU field that is also defined as a descriptor is not included in the inverted lists if it contains a null value. This means that a find command referring to that descriptor will not recognize qualifying descriptor records that contain a null value.

This applies also to subdescriptors and superdescriptors derived from a field that is defined with null-value suppression. No entry will be made for a subdescriptor if the bytes of the field from which it is derived contain a null value and the field is defined with the null-value suppression (NU) option. No entry will be made for a superdescriptor if any of the fields from which it is derived is an NU field containing a null value.

Therefore, if there is a need to search on a descriptor for null values, or to read records containing a null value in descriptor sequence-for example, to control logical sequential reading or sorting-then the descriptor field should not be defined with the NU option.

Null-value suppression is normally recommended for multiple-value fields and fields within periodic groups in order to reduce the amount of disk space required and the internal processing requirements of these types of fields. The updating of such fields varies according to the compression option used.

If a multiple-value field value defined with the NU option is updated with a null value, all values to the right are shifted left and the value count is reduced accordingly. If all the fields of a periodic group are defined with the NU option, and the entire group is updated to a null value, the occurrence count will be reduced only if the occurrence updated is the highest (last) occurrence. For detailed information on the updating of multiple-value fields and periodic groups, see the Adabas Utilities documentation ADACMP utility, and the Adabas Command Reference documentation A1/A4 and N1/N2 commands.

Forward Index Compression

The forward (or ‘front’ or ‘prefix’) index compression feature saves index space by removing redundant prefix information from index values. The benefits are less disk space used, possibly fewer index levels used, fewer index I/O operations, and therefore greater overall throughput. The buffer pool becomes more effective because the same amount of index information occupies less space. Commands such as L3, L9, or S2, which sequentially traverse the index, become faster and the smaller index size reduces the elapsed time for Adabas utilities that read or modify the index.

Within one index block, the first value is stored in full length. For all subsequent values, the prefix that is common with the predecessor is compressed. An index value is represented by

```
<l,p,value>
```

where

- p is the number of bytes that are identical to the prefix of the preceding value; and
- l is the exclusive length of the remaining value including the p-byte.

For example:

Decompressed	Compressed
ABCDE	6 0 ABCDE
ABCDEF	2 5 F
ABCGGG	4 3 GGG
ABCGGH	2 5 H

Index compression is not affected by the format of a descriptor. It functions as well for PE-option and multiclient descriptors.

The maximum possible length of a compressed index value occurs for an alphanumeric value in a periodic group:

```
253 bytes for the proper value if no bytes are compressed
1 byte for the PE index
1 byte for the p-byte.
```

The total exclusive length can thus be stored in a single byte.

Adabas implements forward index compression at the file level. When loading a file (ADALOD), an option is provided to compress index values for that file or not. The option can be changed by reordering the file (ADAORD).

Adabas also provides the option of compressing all index values for a database as a whole. In this case, specific files can be set differently; the file-level setting overrides the database setting.

The decision to compress index values or not is based on the similarity of index values and the size of the file:

- the more similar the index values, the better the compression results.
- small files are not good candidates because the absolute amount of space saved would be small whereas large files are good candidates for index compression.

Even in a worst case scenario where the index values for a file do not compress well, a compressed index will not require more index blocks than an uncompressed index.

Padding Factors

A large amount of record update activity may result in a considerable amount of record migration, i.e., removal of the record from its current block to another block in which sufficient space for the expanded record is available. Record migration may be considerably reduced by defining a larger padding factor for Data Storage for the file when it is loaded. The padding factor represents the percentage of each physical block which is to be reserved for record expansion.

The padding area is not used during file loading or when adding new records to a file (this is not applicable for an ADAM file, since the padding factor is used if necessary to store records into their calculated Data Storage block). A large padding factor should not be used if only a small percentage of the records is likely to expand, since the padding area of all the blocks in which nonexpanding records are located would be wasted.

If a large amount of record update/addition is to be performed in which a large number of new values must be inserted within the current value range of one or more descriptors, a considerable amount of migration may also occur within the Associator. This may be reduced by assigning a larger padding factor for the Associator.

The disadvantages of a large padding factor are a larger disk space requirement (fewer records or entries per block) and possible degradation of sequential processing since more physical blocks will have to be read.

Padding factors are specified when a file is loaded, but can be changed when executing the ADADBS MODFCB function or the ADAORD utility for the file or database.