Adabas Design

Database systems often involve complex data structures and data handling procedures that can be designed and used only by persons with extensive knowledge and experience. Adabas has a remarkably simple structure by comparison, yet it provides significant advantages for operational efficiency, ease of design, definition, and database evolution.

This chapter covers the following topics:

- Adabas Entities
- Database Components
- Database Files
- Record and Field Definitions
- Spanned Records

Adabas Entities

In Adabas, a *field* is the smallest logical unit of information (e.g., current salary) that may be defined and referenced by the user. A *record* is a collection of related fields that make up a complete unit of information (e.g., all the payroll data for a single employee). A *file* is a group of related records that have the same format (with some exceptions; read *Multiple Record Types in One File*). A *database* is a group of related files.

- Adabas Limits
- Adabas Space Management

Adabas Limits

The table below shows the maximum number that mainframe Adabas supports for each entity:

Entity	Maximum
Databases	65,535
Blocks per database	2,147,483,646 using 4-byte RABNs
Files per database	the lower of 5,000 or the Associator block size minus one
Records per file	4,294,967,294 using 4-byte ISNs
Fields per record	3214
Uncompressed record length	depends on the operating system
Compressed record length	Data Storage block size *Spanned records*, supported in Adabas version 8 (or later), split a logical record into multiple physical records, each smaller than one Data Storage (DS) block. For more information, read *Spanned Record Support*.

Adabas Space Management

The disk storage space allocated to a single Adabas database is segmented into *logical* Adabas files. A certain part of the overall space within the database is allocated to each logical file. When the space is filled with records from the file, Adabas automatically allocates more space to the file from the common free space pool. This dynamic space allocation, together with the dynamic recovery of released space, allows Adabas databases to run without intervention for long periods of time.

The distribution of database space across disk drives can be controlled by physically segmenting it into multiple independent data sets. When all physical database space is filled, more data sets can be allocated dynamically, or the size of existing data sets can be increased so that new physical files can be loaded without reorganizing the entire database.

Database Components

To support the separation of data and access structures, the Adabas nucleus uses three database components:

- Data Storage for compressed data
- Associator for data management and retrieval
- Work, a scratch area for complex search criteria, etc.

This section describes each of these database components:

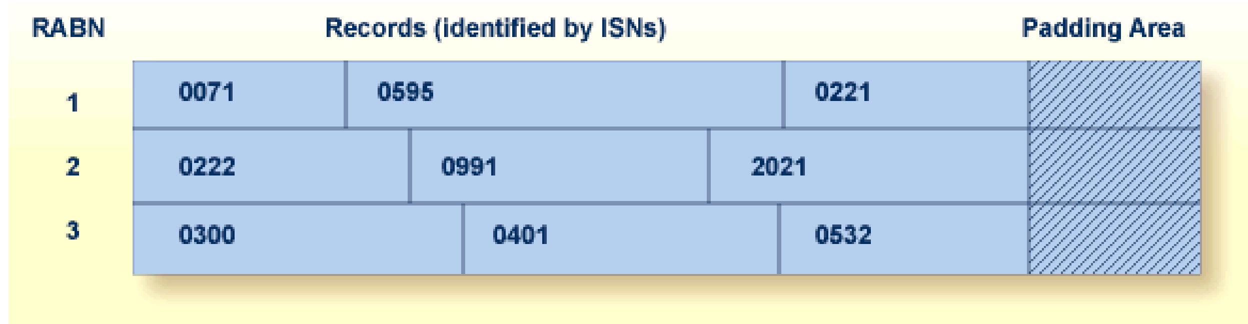
- Data Storage
- Associator
- Work
- Other Components

Data Storage

Data Storage is divided into blocks, each identified by a 3- or 4-byte relative Adabas block number, or RABN, that identifies the block's physical location relative to the beginning of the component. Data Storage blocks contain one or more physical records and a padding area to absorb the expansion of records in the block.

A logical identifier stored in the first four bytes of each physical record is the only control information stored in the data block. This internal sequence number or ISN uniquely identifies each record and never changes. When a record is added, it is assigned an ISN equal to the highest existing ISN plus one. When a record is deleted, its ISN is reused only if you instruct Adabas to do so. Reusing ISNs reduces system overhead during some searches and is recommended for files with records that are frequently added and deleted.

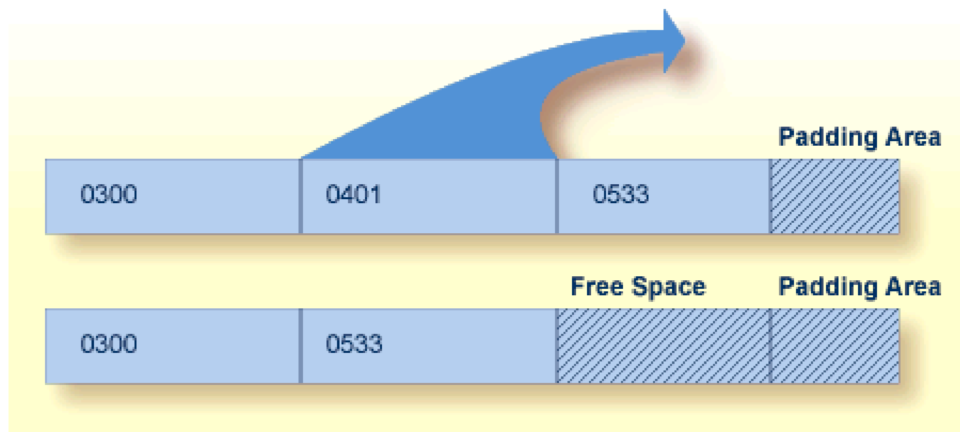
For each file, between 1-90 percent (default 10%) of each block can be allocated as padding based on the amount and type of updating expected. This reserved space permits records to expand without migrating to another block and thus helps to minimize system overhead.



- Free Space and Space Reusage
- Compression

Free Space and Space Reusage

If records become too large for their blocks, they migrate to new locations. When a record migrates or is deleted, free space is opened in the data block between the last record and the padding area. The following figure shows free space created when the record with ISN 0401 becomes too large for the block and migrates to another block:



You can instruct Adabas to reuse free space. Reusing space saves computer time, since Adabas then reads fewer physical blocks during searches. It is recommended for all files.

Compression

Data compression significantly reduces the amount of storage required. It also permits the transmission of more information per physical transfer, resulting in greater I/O efficiency.

Adabas retains data records in compressed form. Several compression options are supported:

- default compression;
- null suppression; and
- fixed format; and
- forward or prefix index compression.

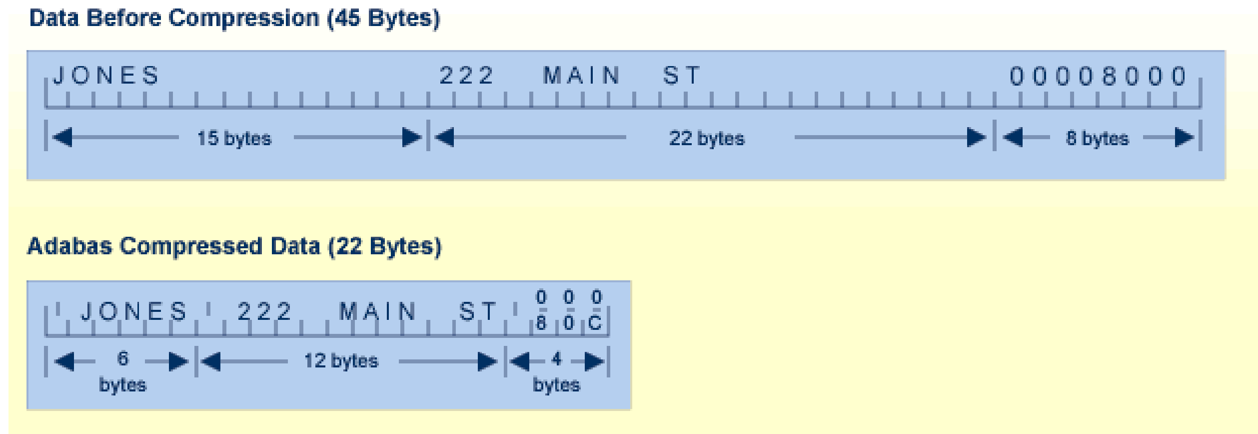
The first three options define and execute compression at the field level, with null suppression and fixed format compression added as field options.

The fourth option, forward or prefix index compression, compresses the descriptor values in the Associator's inverted list. It can be implemented at the file or the database level, in which case specific files can be set differently; the file-level setting overrides the database setting. The forward index compression option is set using the ADALOD utility and can be changed using the ADAORD utility. This compression option is more fully described in *Inverted Lists*.

The null suppression and fixed format options are added as field options and are discussed in *Data Compression Options FI and NU*.

Default compression deletes trailing blanks in alphanumeric fields and leading zeros in binary fields. An inclusive length byte (ILB) at the beginning of the field indicates the total number of stored bytes, including the ILB. Thus, if "Susan" is entered in a "first-name" field defined with a 20-character length and default compression, its stored size will be six bytes: five bytes for the letters of the name, plus one byte for the ILB. In addition, empty fields in a record are not stored; an empty field is replaced by a one-byte empty field counter (EFC). Adabas can store up to 63 contiguous empty fields in a single hexadecimal byte.

Many Adabas files require only 50% to 60% of the space used for the raw data. Even with the addition of approximately 25% for the access structures stored in the Associator, Adabas storage requirements are still less than those required for traditional file storage or for DBMSs that do not use data compression.



Associator

The Associator is an organizational unit used for storing the structures required to access data in Data Storage. It contains the following elements:

- Two *general control blocks (GCBs)* for the database. The GCBs provide information regarding the physical characteristics of the database, such as the database ID (DBID), the number of files loaded, the number of Associator, Data Storage, and Work extents, the Associator, Data Storage, and Work device types, system file information, Data Storage Space Table (DSST) extents, and the database version indicator.
- Individual *file control blocks (FCBs)* for each file. The FCBs identify the physical characteristics and associated RABNs of database files. The contents include the file name, file number, current file status, the ISN reuse settings, the space reuse settings, MINISN and MAXISN settings, the first free ISN, and the number of updates against the file. In addition, the first RABN, last RABN, and first unused RABN are stored in the FCB.
- All tables needed to control and maintain the database including a field definition table (FDT) for each file and coupling lists for physically coupled files. For more information about the FDT, read *Records and Field Definitions*. For more information about physically coupled files, read *Coupled Files*.
- An inverted list for each descriptor in each file of the database and an address converter for each file.
- If spanned records are used in a file, a secondary address converter for the file.

Inverted Lists

An inverted list, which is used to resolve Adabas search commands and read records in logical sequence, is built and maintained for each field in an Adabas file that is designated as a key field or *descriptor* (read *Descriptor Options DE, UQ, and XI*). It is called an *inverted list* because it is organized by descriptor value rather than by ISN. The list comprises the normal index (NI) and as many as 14 upper indexes (UI).

The normal index (NI) of the inverted list for a particular descriptor has an entry for each value. The entry contains the value itself, the number of records in which the value occurs, and the ISNs of those records.

To increase search efficiency, upper index (UI) levels are automatically created by Adabas as required, each level to manage the next lower level index. The first level UI, like the NI it manages, contains entries for only one descriptor in each index block. All other UI levels contain entries for all descriptors in each index block. UIs require a minimal amount of space: two blocks is the minimum.

Note:
The Adabas direct access method (ADAM) facility permits the retrieval of records directly from Data Storage without accessing the inverted lists. The Data Storage block number in which a record is located is calculated using a randomizing algorithm based on the ADAM key of the record. The use of ADAM is completely transparent to application programs and query and report writer facilities. See *Random Access Using the Adabas Direct Access Method (ADAM)* for more information.

The following figure shows a typical normal index for the descriptor CITY in a customer file.

Value	Count	ISNs
London	27	3 ...
New York	61	96 ...
Zurich	31	2 6 23 76 ...
...		

The example indicates that there are 31 records with the CITY Zurich (the ISNs of these records are 2,6,23,76...).

Forward (or 'front' or 'prefix') index compression removes redundant prefix information from index values. 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.
l	is the exclusive length of the remaining value including the p-byte.

For example:

Before Compression	After Compression
ABCDE	6 0 ABCDE
ABCDEF	2 5 F
ABCGGG	4 3 GGG
ABCGGH	2 5 H

The decision to compress index values 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.

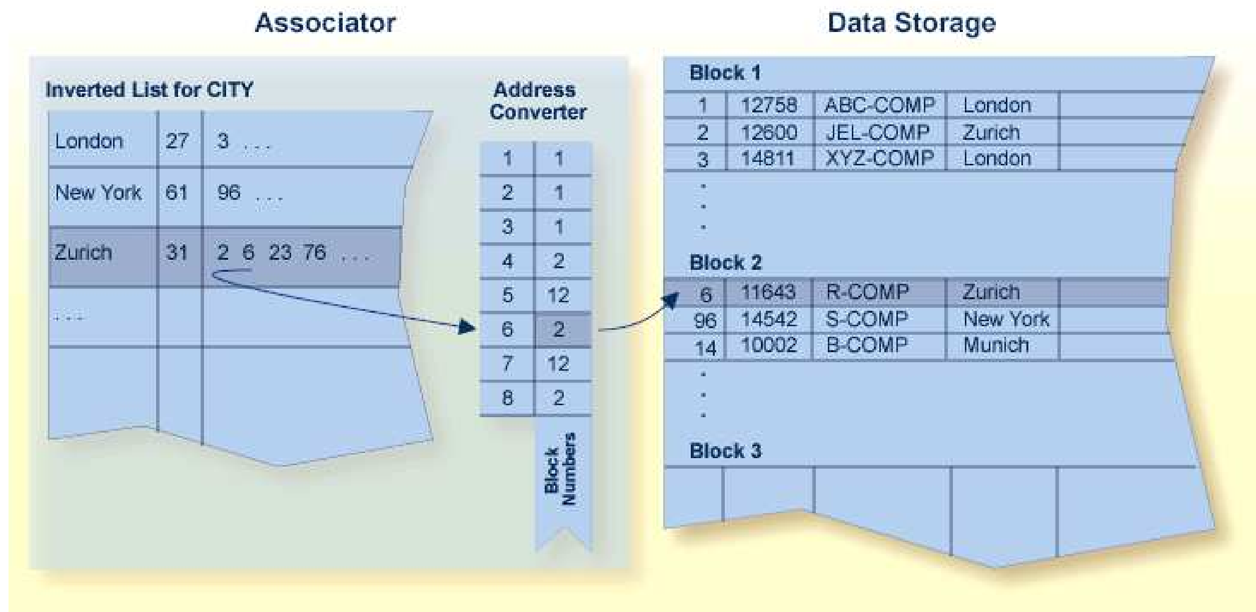
Even in a worst case scenario where the index values for a file do not compress well, a compressed index will not require more index blocks than an uncompressed index.

Address Converter

The address converter determines the physical location of a record. It is an index that maps the logical identifier of a record (that is, the ISN) to the relative Adabas block number (RABN) of the Data Storage block where the record is stored. If spanned records are used, a secondary address converter is used to map the secondary ISNs to the RABNs of the Data Storage blocks where the secondary records are stored. For more information about spanned records, read *Spanned Records*.

The address converter contains a list of RABNs in ISN order. Only the RABNs are actually stored in the address converter; the ISNs are identified by their relative position.

The following figure shows the relationship between an inverted list, the address converter, and Data Storage. For example, to determine the physical location of the record whose ISN is 6, Adabas uses the ISN as an index into the address converter. The sixth entry in the address converter is 2. Therefore, ISN 6 is located in physical block 2 in Data Storage for this file.



When a record moves or is deleted, Adabas updates the address converter automatically and transparently.

Since the ISN for a record never changes, and its physical block address is stored only in the address converter entry, the record itself may be moved in Data Storage with only one update to the address converter required and with no extension to the access path of the record.

Even if a record has many descriptors defined, the inverted list for each descriptor need not be modified because it contains ISNs.

This process explains how Adabas is able to perform simple and complex searches quickly and efficiently without storing pointer information in Data Storage.

Work

The Work area stores information in four parts:

Part	Stores . . .
1	data protection information required by the routines for autorestart and autobackout. Read *Backout, Recovery, and Restart* for more information.
2	intermediate results (ISN lists) of search commands.
3	final results (ISN lists) of search commands.
4	data related to two-phase commit processing.

Other Components

Sort and Temp Areas

Certain Adabas utilities (ADAINV, ADALOD) require two additional data sets, sort and temp, for sorting and intermediate storage of data. Certain functions of other utilities require the temp data set for intermediate storage.

The size of the temp and sort data sets varies according to the utility function to be executed. These data sets can be allocated during the job and then released, or permanent data sets can be allocated and reused.

Logs

Adabas uses the following optional logs:

- The Command log (CLOG) records information from the control block of each Adabas command that is issued. The CLOG provides an audit trail and can be used for debugging and for monitoring the use of resources. Single, dual, or multiple (2-8) data sets can be used (multiple data sets are recommended).

  Timestamps in an Adabas 8 command log created using the ADARUN CLOGLAYOUT=8 parameter are stored in machine time (GMT), whereas CLOGLAYOUT=5 timestamps are stored, as always, in local time. The LORECX record layout that describes the CLOGLAYOUT=8 command log includes a differential time field that stores the difference between machine time and local time at the time the CLOG record is written. This field allows you to calculate the local time of a command log record.

  Because of the difference in timestamp formats, we do not recommend that you mix or merge command logs created using different CLOGLAYOUT settings. This is especially true for Adabas nuclei in a cluster environment. For more information, read *CLOGLAYOUT : Command Logging Format* .

- The Protection log (PLOG) records before- and after-images of records and other elements when changes are made to the database. It is used to recover the database (up to the last completed transaction or ET) after restart. Single, dual, or multiple (2-8) data sets can be used (multiple data sets are recommended).

- The Recovery log (RLOG) records additional information that the Adabas Recovery Aid uses to construct a recovery job stream. Read the ADARAI utility discussion for more information.

Database Files

Each database contains system files and data files. A data file is generally created for each record structure required; that is, for each set of related fields identified.

Files are loaded into the database using the ADALOD utility. A file number must be unique in the database and not greater than the maximum file number defined for the database in the MAXFILES parameter. Checkpoint, security, trigger, and system files can have two-byte file numbers, but cannot be greater than 5000. Physically coupled files cannot include files with numbers greater than 255. File numbers are assigned by the user in any sequence.

This section describes the different types of database files:

- System Files
- Coupled Files
- Structuring Files to Enhance Performance

System Files

Adabas uses certain files to store system information. Using the ADALOD utility's FILE parameter, you can identify an Adabas *system* file as one of the following:

CHECKPOINT	Adabas checkpoint file
SECURITY	Adabas security file
SYSFILE	Adabas system file
TRIGGER	Adabas trigger file

Coupled Files

File coupling allows you to select, using a single search command, records from one file that are related (coupled) to records containing specified values in a second file.

- Physical Coupling
- Logical or Soft Coupling

Physical Coupling

Any two files with file numbers 255 or lower may be physically coupled if a common descriptor (read *Descriptor Options DE, UQ, and XI*) with identical format and length definitions is present in both files. A single file may be coupled with up to 18 other files, but only one coupling relationship may exist between any two files at any one time. A file may not be coupled to itself.

When files are coupled, coupling lists are created in the Associator for each file being coupled. File coupling is bidirectional rather than hierarchical in that two coupling lists are created for each coupling relationship with each list containing the ISNs that are coupled to the other file.

Once the physical coupling lists have been created, any key field in either file may be used within a search criteria.

Physical coupling may add a considerable amount of overhead if the files involved are frequently updated. The coupling lists must be updated if a record in either of the files is added or deleted, or if the descriptor used as the basis for the coupling is updated in either file.

Physical coupling may be useful for information retrieval systems in which

- files seldom change;
- the additional overhead of the coupling lists is insignificant compared with the increased ease of formulating queries; or

- files are small and primarily query-oriented.

Logical or Soft Coupling

Multiple files may also be queried by specifying the field to be used for interfile linkage in the search criteria. Adabas then performs all necessary search, read, and internal list matching operations.

This technique is called logical or soft coupling because it does not require the files to be physically coupled. Although logical coupling requires read commands, it is normally more efficient because it avoids the increased overhead of coupling lists.

Structuring Files to Enhance Performance

An Adabas database with one file for each record type supports any application functions required of it and is the easiest to manipulate for interactive queries, but it may not yield the best performance:

- As the number of Adabas files increases, the number of Adabas calls increases. Each Adabas call requires interpretation, validation and, in multiuser mode, supervisor call (SVC) and queuing overhead.

- In addition to the input/output (I/O) operations necessary for accessing at least one index, address converter, and Data Storage block from each file, the one-file-per-record-type structure requires buffer pool space. If sufficient buffer space is not available, blocks are overwritten that may be needed for a later request.

The number of Adabas files used by critical programs can be reduced by

- using multiple-value fields and periodic groups (read *Field Levels*);

- linking physical files into a single logical (expanded) file;

- including more than one type of record in an Adabas file;

- including records for more than one category of user in an Adabas (multiclient) file; and

- controlling data duplication and the resulting high resource usage.

This section describes the following topics:

- Expanded Files
- Multiple Record Types in One File
- Multiclient Files
- Controlled Data Redundancy

Expanded Files

If you have a large number of records of a single type, you may need to spread the records over multiple physical files.

To reduce the number of files accessed, Adabas allows you to link multiple physical files containing records of the same format together as a single logical file. This file structure is called an expanded file and the physical files comprising it are the component files. An expanded file can comprise up to 128 component files, each with a unique range of logical ISNs. An expanded file cannot exceed 4,294,967,294

records.

Note:
Since Adabas now supports larger file sizes and a greater number of Adabas physical files and databases, the need for expanded files has, in most cases, been removed.

Although an application program addresses the logical file (the address of the file is the number of the expanded file’s base component or anchor file), Adabas selects the correct component file based on the data in a field defined as the criterion field. The data in this field has characteristics unique to records in only one component file. When an application updates the expanded file, Adabas looks at the data in the criterion field in the record to be written to determine which component file to update. When reading expanded file data, Adabas uses the logical ISN as the key to finding the correct component file.

Multiple Record Types in One File

Multiple record types can be defined within a single physical record; each record type is a logical record composed of a subset of the fields defined for the file. Fields that do not belong to a given type are null-suppressed.

Record types can be identified to Adabas by

- defining a record type field with values to differentiate one type from another; or
- using values of an existing field to differentiate type; for example, to differentiate two types, a value of zero for a field common to both types might identify one type and any nonzero value for the same field might identify the other type.

Multiclient Files

Records for multiple users or groups of users can be stored in a single Adabas physical file defined as multiclient. The multiclient feature divides the physical file into multiple logical files by attaching an internal owner ID to each record.

The owner ID is assigned to a user ID. A user ID can have only one owner ID, but an owner ID can belong to more than one user. Each user can access only the subset of records that is associated with the user’s owner ID.

Note:
For any installed external security package such as RACF, CA-ACF2, or CA-Top Secret, a user is still identified by either Natural ETID or LOGON ID.

All database requests to multiclient files are handled by the Adabas nucleus.

Controlled Data Redundancy

Physical redundancy increases storage requirements but may also enhance performance and decrease complexity. For example, if a database stores customer and order information in a customer-orders file and product descriptions in an inventory file, and a program that generates invoices requires product descriptions in addition to customer-order data, it might enhance performance to store a duplicate copy of the product descriptions in the customer-orders file.

Logical redundancy also increases storage demands while decreasing complexity. It involves storing in one file the results of a process on data in another file; thus, the duplicate data is implied by the content of another file, rather than being physically stored in two places.

Physical and logical redundancy cause update programs to run more slowly. The duplicate updates required when changes in one file affect records in another file may degrade performance severely. Redundancy should be used only for static data or data that is updated rarely. You can control data redundancy by using multiple-value fields, periodic groups, and multiple record types within a file.

Record and Field Definitions

In Adabas, the record structure and the content of each field in a physical file are described in a Field Definition Table, or FDT, which is stored in the Associator. There is one FDT for each database file. The FDT is used by Adabas during the execution of Adabas commands to determine the logical structure and characteristics of any given field (or group) in the file.

Spanned records, supported in Adabas version 8 (or later), split a logical record into multiple physical records, each smaller than one Data Storage (DS) block. For more information, read *Spanned Record Support*.

This section covers the following topics:

- Record Structure and the FDT
- Field Levels and Group Fields
- System Fields
- Field Names
- Field Length and Data Format
- Field Options
- Special Fields and Descriptor Fields

Record Structure and the FDT

The FDT lists the fields of the file in physical record order, provides a quick index to the file’s records, and defines the file’s fields, subfields, superfields, and descriptors (including collation descriptors, subdescriptors, superdescriptors, hyperdescriptors, and phonetic descriptors). A minimum of one and a maximum of 3214 field definitions may be specified.

Information about each field includes the level, name, length, format, options, and special field and descriptor attributes.

```
FIELD DESCRIPTION TABLE

      I      I        I        I              I
 LEVEL I NAME I LENGTH I FORMAT I  OPTIONS     I         PARENT OF
      I      I        I        I              I
------I------I--------I--------I--------------I----------------------------I
      I      I        I        I              I                            I
   1  I  AA  I    8   I    A   I DE,UQ        I                            I
   1  I  AB  I        I        I              I                            I
   2  I  AC  I   20   I    A   I NU           I                            I
   2  I  AE  I   20   I    A   I DE           I  SUPERDE,PHONDE            I
   2  I  AD  I   20   I    A   I NU           I                            I
   1  I  AF  I    1   I    A   I FI           I                            I
   1  I  AG  I    1   I    A   I FI           I                            I
   1  I  AH  I    6   I    U   I DE           I                            I
   1  I  A2  I        I        I              I                            I
   1  I  AO  I    6   I    A   I DE           I  SUBDE,SUPERDE             I
   1  I  AQ  I        I        I PE           I                            I
   2  I  AR  I    3   I    A   I NU           I  SUPERDE                   I
   2  I  AS  I    5   I    P   I NU           I  SUPERDE                   I
   1  I  A3  I        I        I              I                            I
   2  I  AU  I    2   I    U   I              I  SUPERDE                   I
   2  I  AV  I    2   I    U   I NU           I  SUPERDE                   I
```

The order of the fields listed in the FDT determines the structure of the record and the efficiency of retrieval. The following factors should be considered when ordering fields:

- Fields that will be accessed frequently should be ordered first in the FDT. This technique reduces CPU time because Adabas does not have to read the whole record when retrieving a field.
- Fields that will frequently be accessed together should be assigned to a group field.
- Fields that will *always* be accessed together should be defined as a single field. This technique may inhibit compression and query language use; however, it decreases processing time by providing more efficient internal processing and shorter format buffers.
- If appropriate, fields that will frequently be empty should be ordered together in the FDT and set to use default compression or null suppression.
- Numeric fields should be loaded in the format in which they will be used most often.

Field Levels and Group Fields

When two or more consecutive fields in the FDT are frequently accessed together, you can reference them together by defining a group field. Other than its level and Adabas short name, a group field has no attributes defined. It immediately precedes its member fields in the FDT. A higher field level number is used to assign the member fields to the group field. Adabas supports up to seven field levels. User programs can access each member field individually, or all member fields together by referencing the group field.

For example, in the illustration of the Field Definition Table (FDT) in the section *Records and Field Definitions*, field AB is defined as a group field and assigned to level 1. Fields AC, AE, and AD are assigned to level 2, indicating that they belong to group field AB. The next field, AF, is assigned to level 1, indicating that it is not part of the AB group. User programs can access AC, AE, and AD individually, or together by referencing the group field AB.

A group field can be assigned as a *periodic group field* if it is comprised of fields that can have more than one value (for example, group field AQ in the figure).

System Fields

Adabas allows you to define *system fields* in your Adabas files.

A system field is a field in an Adabas file whose value is automatically set by the Adabas nucleus when records are inserted or updated on the file. Optionally, you can specify that some system field values only be set when records are inserted. System fields are fields that store information such as the job name of the user making the update, the eight-byte user ID of the user, the session ID of the user, or the time at which the update was made.

The value of a system field refers to an insert or update of an entire record. You cannot define a system field that refers to only a portion of a record.

Values for system fields are saved in the compressed storage record in the same manner that other Adabas fields are stored.

This section covers the following topics:

- Allowed Types of System Fields
- Defining System Fields
- System Fields as MU Fields
- System Field Rules
- System Field Processing by an Adabas Nucleus

Allowed Types of System Fields

System fields containing the following types of information can currently be defined in an Adabas file:

- Job name: The job name of the user inserting or updating a record.
- ETID: The eight-byte user ID of the user inserting or updating a record. This is the user ID set in the Additions 1 field of an OP (open) command for the user session.
- Session ID: The 28-byte user ID of the user inserting or updating a record.
- Session user: The last eight bytes of the 28-byte session ID or the user inserting or updating a record.
- Time: The date or date and time at which a record is inserted or updated.

Note:
Information about record deletion is not recorded in system fields for the simple reason that the value of the system field is itself deleted along with the record.

Defining System Fields

System fields may *not* be part of a periodic group. Otherwise, they can be simple fields or MU fields (although not MU fields in a periodic group).

System fields are defined in the same manner as other database fields using FNDEF definitions in ADACMP COMPRESS utility runs, except each system field definition must include an SY field option. If you only want the system field values updated when a record is inserted (and not when it is updated), you can also specify the CR field option in the system field definition. If the system field is a multiple occurrence (MU) field, it *cannot* be defined with the CR field option.

System fields can also be added to a file using the ADADBS NEWFIELD utility function. The same SY and CR field options are supported by ADADBS NEWFIELD.

The SY and CR options cannot be changed for a field using the ADADBS CHANGE utility function. In other words, a field defined as a system field cannot later be changed from a system field to a non-system field. However, a system field may be logically deleted. Note that system field values for logically deleted fields are still set when record insertions or updates occur; this ensures that the system field values are correct if they are later no longer logically deleted

For more information about the SY and CR field options, read *Field Options*.

System Fields as MU Fields

A system field cannot be part of a periodic group. In addition, if the system field is defined with the CR field option, it cannot also use the MU option. Likewise, if the system field is *not* defined with the CR option, it *must* be defined as an MU field.

If a system field is an MU (multiple occurrence) field, Adabas sets the field values on record insertion and record update in a specific way.

- When a record is inserted, an appropriate value is set in the first occurrence of the MU system field. The MU system field will contain only a single value (the value set in the first occurrence of the field).
- When a record is updated, an appropriate value is set in the first occurrence of the MU system field. Values held previously in occurrences 1 to N of the MU system field will be shifted to occurrences 2 to N+1. If the maximum number of MU system field values is reached for the file, the oldest value is dropped from the file. For example, if a file is defined with a maximum number of MU system field values set to 5 and 5 values are already present in the file, an update to the MU system field will drop the value in occurrence 5 of the field, shift the others down and insert the new value in occurrence 1.

The maximum number of MU system field values allowed in a file is stored in the File Control Block (FCB) for the file and is set when you load the file using ADALOD LOAD. Thereafter, the SYFMAXUV value can be modified using the ADADBS MODFCB utility function. This setting applies to all MU system fields in the file. For more information, read *MODFCB: Modify File Parameters* .

The following are examples of valid system fields. Note that the fourth example from the top is *not* an MU field, but is defined with the CR option.

```
01,ET,8,A,NU,MU,SY=OPUSER
01,SU,8,A,MU,NU,NV,SY=SESSIONUSER
01,SI,28,A,NU,NV,MU,SY=SESSIONID
01,D1,8,U,NU,DT=E(DATE),SY=TIME,CR
01,TI,14,U,MU,NU,DT=E(DATETIME),SY=TIME
01,TZ,14,U,MU,NU,DT=E(DATETIME),TZ,SY=TIME
01,Z3,8,A,MU,SY=JOBNAME
```

System Field Rules

The following rules apply to system fields:

1. The field format for JOBNAME, OPUSER, SESSIONID, and SESSIONUSER system fields must be A (alphanumeric).
2. The format and length of a TIME system field will be enforced based on the rules set for the date-time edit mask specified for the field.

   A date-time edit mask (DT field option) must be specified for time system fields. However, the DT field option is not valid for any other type of system field.
3. System fields cannot be a periodic group field (the PE field option *cannot* be specified), nor can it be a member of a periodic group.
4. A system field can be a descriptor (the DE field option can be specified) or a unique descriptor field (the UQ field option can be specified).
5. The system field may be the parent of a superdescriptor field or of a special descriptor.
6. A system field *cannot* be a long alpha or wide-character field or a large object field (the LA and LB field options *cannot* be specified).
7. Security-by-value is allowed for system fields.
8. The CR field option can only be specified if the SY field option is also specified for a field.
9. A system field can be defined to have a fixed storage length (the FI field option can be specified). If FI is specified, the field length must exactly match the lengths shown in the SY field option description. If it is not specified, any field length is allowed in the field definition; the length of the data stored for each field will match the lengths shown in the SY field option description.
10. A system field must have either the MU option *OR* the CR option specified (but not both). The MU and CR options are mutually exclusive.
11. Null values can be suppressed for a system field (the NU field option can be specified).
12. Alphanumeric system fields can be processed in the record buffer without being converted (the NV field option can be specified).
13. The value of a system field refers to the insert or update of an entire record. You cannot define a system field that refers to only a portion of a record.
14. Information about record deletion is not recorded in system fields for the simple reason that the value of the system field is itself deleted along with the record.
15. System fields are not required in a file (none can be specified). In addition, one or more system fields of the same type can be defined for a file.
16. The SY and CR options cannot be changed for a field using the ADADBS CHANGE utility function. In other words, a field defined as a system field cannot later be changed from a system field to a non-system field.

17. A system field can be logically deleted. Note that system field values for logically deleted fields are still set when record insertions or updates occur; this ensures that the system field values are correct if they are later no longer logically deleted.

System Field Processing by an Adabas Nucleus

When a record is inserted or updated in an Adabas database file with system fields, the system field values are set by the nucleus. If system fields are specified one or more times in the format buffers and record buffers of an insert or update command, the values passed by the user are ignored.

The following processing occurs by the Adabas nucleus for system fields:

1. If the system field is *not* defined with the CR option, the system field value is set by the nucleus when an insert command and when an update command is issued.
2. If the system field is defined with the CR option, the system field value is set by the nucleus when an insert command is issued, and is left as is when an update command is issued.

Field Names

A field is identified to Adabas by a two-character Adabas short name that must begin with an alphabetic character (either upper- or lowercase) and can be followed by a numeral or an alphabetic character (either upper-or lowercase) and must be unique within a file. The combinations E0-E9 are reserved and special characters are not allowed. Adabas assigns short names to fields automatically, although you can choose to assign them yourself. Adabas uses the short names internally and actually accesses fields by their short names.

Note:
Lowercase fields will not display correctly (they will be converted to uppercase) if you use the ADARUN parameter settings MSGCONSL=UPPER, MSGDRUCK=UPPER, or MSGPRINT=UPPER.

Field Length and Data Format

Field values are fixed or variable in length and can be in alphanumeric, binary, fixed-point, floating-point, packed/unpacked decimal, or wide character formats.

The length (expressed in bytes) and format (expressed as a one-character code) of a field define the standards (defaults) to be used by Adabas during command processing. They are used when the field is read/updated unless the user specifies an override.

If standard length is zero for a field, the field is assumed to be a variable-length field. Standard format must be specified for a field. The format specified determines the type of default compression to be performed on the field.

The maximum field lengths that may be specified depend on the format value:

Format	Format Description	Maximum Length
A	Alphanumeric (left-justified): see also the long alphanumeric (LA) option in *Long Alpha Option LA* and the large object (LB) option in *Large Object Option LB*	253 bytes
B	Binary (right-justified, unsigned/positive)	126 bytes
F	Fixed point (right-justified, signed, positive value in normal form; negative value in two's complement form)	8 bytes (always exactly 2, 4, or 8 bytes)
G	Floating point (normalized form, signed)	8 bytes (always exactly 4 or 8 bytes)
P	Packed decimal (right-justified, signed)	15 bytes
U	Unpacked decimal (right-justified, signed)	29 bytes
W	Wide character (left-justified): see also the long alphanumeric (LA) option in *Long Alpha Option LA*	253 bytes

Field Options

Field options are specified using two-character codes, which may be specified in any order, separated by a comma.

Code	Option	Read Section
CR	A system field will not be modified when updates occur to the record, but only when the record is first inserted.	*System Field Options SY and CR*
DE	The field is to be a descriptor (key).	*Descriptor Options DE, UQ, and XI*
DT	A date-time edit mask is specified for the binary, fixed point, packed decimal, or unpacked decimal field.	*Date-Time Edit Mask Option DT*
FI	The field is to have a fixed storage length; values are stored without an internal length byte, are not compressed, and cannot be longer than the defined field length.	*Data Compression Options FI and NU*
LA	An alphanumeric or wide-character, variable-length field may contain a value up to 16,381 bytes long.	*Long Alpha Option LA* and *Comparing LA and LB Fields*
LB	An alphanumeric field may contain up to 2,147,483,643 (about 2 GB) of data.	*Large Object Option LB* and *Comparing LA and LB Fields*
MU	The field may contain up to about 65,534 values in a single record.	*MU and PE Options and Field Types*

Code	Option	Read Section
NB	Trailing blanks should not be removed (compressed) from the LA or LB fields. Specification of this option requires the specification of NU or NC as well.	*Blank Compression Option NB*
NC	Field may contain a null value that satisfies the SQL interpretation of a field having no value; that is, the field's value is not defined (not counted).	*SQL Compatibility Options NC and NN*
NN	Field defined with NC option must always have a value defined; it cannot contain an SQL null (not null).	*SQL Compatibility Options NC and NN*
NU	Null values occurring in the field are to be suppressed.	*Data Compression Options FI and NU*
NV	An alphanumeric or wide-character field is to be processed in the record buffer without being converted.	*Encoding Conversion Option NV*
PE	This group field is to define consecutive fields (which may include one or more MU fields) in the FDT that repeat together (up to about 65,534 times) in a record.	*MU and PE Options and Field Types*
SY	The field is a system field.	*System Field Options SY and CR*
TZ	The date-time field value is presented in the user's local time and stored in UTC time, allowing for differences in time zones.	*Time Zone Option TZ*
UQ	The field is to be a unique descriptor; that is, for each record in the file, the descriptor must have a different value.	*Descriptor Options DE, UQ, and XI*
XI	For this field, the occurrence (index) number is to be excluded from the unique descriptor (UQ) option set for a periodic group (PE).	*Descriptor Options DE, UQ, and XI*

Descriptor Options DE, UQ, and XI

A *descriptor* is a search key. The DE option indicates that the field is to be a descriptor. The UQ option can only be specified if DE is also specified; it indicates that the DE field is to have a different (i.e., unique) value for each record in the file. If the UQ field is also an MU field or a field in a periodic group, the same value for the field may occur multiple times in the same record, but must be unique in different records. Entries are made in the Associator's inverted list for DE fields, adding disk space and processing overhead requirements.

Any field can be used within a selection criterion. When a field that is used extensively as a search criterion is defined as a descriptor (key), the selection process is considerably faster since Adabas is able to access the descriptor's values directly from the inverted list without reading any records from Data Storage.

A descriptor field can be used as a sort key in a search command, as a way of controlling a logical sequential read process (ascending or descending values), or as the basis for file coupling.

Any field and any number of fields in a file can be defined as descriptors. When a multiple-value field or a field in a periodic group is defined as a descriptor, multiple key values are generated for the record. Key searches may be limited to particular occurrences of a periodic group.

For descriptor fields that are part of a periodic group (PE field), the group index is considered part of the descriptor value in the index. This makes it possible to search for a value plus a group index. By default, a given value plus the group index of one occurrence of a record is considered different than the same value plus the different group index of a second record. Because the group indexes are different, these two occurrences do not violate the "uniqueness" criteria. If you want to eliminate the group index from the uniqueness criteria, use the XI option. The XI option is used for unique descriptors in periodic groups to exclude the occurrence (index) number from the definition of uniqueness.

Because the inverted list requires disk space and update overhead, the descriptor option should be used judiciously, particularly if the file is large and the field that is being considered as a descriptor is updated frequently. For instance, the inverted list for a periodic group used as a descriptor may be very large because each occurrence is stored.

A descriptor may be defined at the time a file is created, or later by using an Adabas utility. Because the definition of a descriptor is independent of and has no effect on the record structure, descriptors may be created or deleted at any time without the need for database restructuring or reorganization.

Note, however, that if a descriptor field is not ordered first in the record structure and logically falls past the end of the physical record, the inverted list entry for that record is not generated for performance reasons. To generate the inverted list entry in this case, it is necessary to unload short, decompress, and reload the file; or use an application program to reorder the field first for each record of the file.

A portion of a field may be defined as a *subdescriptor*; combinations of fields or portions thereof may be defined as a *superdescriptor*; a user-supplied algorithm may be the basis of a *collation descriptor* or *hyperdescriptor*; and a sounds-like encoding algorithm may be the basis of a *phonetic descriptor*, which may be customized for specific language requirements. Read *Special Field and Descriptor Attributes* for more information.

System Field Options SY and CR

A system field is a field in an Adabas file whose value is automatically set by the Adabas nucleus when records are inserted or updated on the file. Optionally, you can specify that some system field values only be set when records are inserted. A system field cannot be a PE field.

System fields containing the following types of information can currently be defined in an Adabas file:

- Job name: The job name of the user inserting or updating a record. When ADACMP COMPRESS is used to define this type of field, its field value will be the job name of the ADACMP COMPRESS job.

- ETID: The eight-byte user ID of the user inserting or updating a record. This is the user ID set in the Additions 1 field of an OP (open) command for the user session.
- Session ID: The 28-byte user ID of the user inserting or updating a record.
- Session user: The last eight bytes of the 28-byte session ID or the user inserting or updating a record.
- Time: The date or date and time at which a record is inserted or updated.

Use the SY field option to specify the type of information stored in a system field.

Use the CR option to indicate that the system field value should only be maintained when a record is inserted and not when it is updated. The CR field option can only be specified for fields defined with the SY field option, but cannot be specified for an MU field.

For complete information about system fields, read *System Fields*. For more information about the SY and CR field options, read *Field Options*.

Date-Time Edit Mask Option DT

DT assigns a date-time edit mask to a binary, fixed point, packed decimal, or unpacked decimal field. This option cannot be specified for fields of other formats.

The syntax of the DT option is:

```
DT=E(edit-mask-name)
```

Valid values for edit-mask-name substitutions are described in the following table. It also shows the required minimum field lengths for the different formats of fields that can specify the DT option; the length of the field must be large enough to store the date-time values. Detailed discussions of each edit mask is provided in *Date-Time Edit Mask Reference*.

Note:
In the table, "YYYY" represents the 4-digit year (1-9999), "MM" represents the 2-digit month (1-12), "DD" represents the 2-digit day of the month (1-31), "HH" represents the 2-digit hour (0-23), "II" represents the 2-digit minute within the hour (0-59), "SS" represents the 2-digit second within the minute (0-59), and "XXXXXX" represents the 6-digit microsecond within the second.

edit-mask-name	**Description**	**Minimum Field Length for Field Format**			
		B	**F**	**P**	**U**
DATE	The date field is in the format `Z'`*`YYYYMMDD`*`'`.	4	4	5	8
TIME	The time field is in the format `Z'`*`HHIISS`*`'`.	3	4	4	6
DATETIME	The date and time field is in the format `Z'`*`YYYYMMDDHHIISS`*`'`	6	8	8	14
TIMESTAMP	The date and time field is in the format `Z'`*`YYYYMMDDHHIISSXXXXXX`*`'`, with microsecond precision	--	--	11	20
NATTIME	The time field is in Natural T format (tenths of seconds since year zero)	6	8	7	13
NATDATE	The date field is in Natural D format (days since year zero)	3	4	4	7
UNIXTIME	The time field is in UNIX time_t type format (seconds since January 1, 1970)	4	4	6	10
XTIMESTAMP	The date and time field is in UNIX timestamp format, with microsecond precision, since January 1, 1970 (`UNIXTIME * 60**6 +` *`microseconds`*).	8	8	10	18

The following table contains some examples.

Example	**The field contains...**
`1,SD,8,U,DT=E(DATE)`	Numeric data in the form `Z'YYYYMMDD'`.
`1,TI,6,U,DT=E(TIME)`	Numeric time in the form `Z'HHIISSD'`
`1,DT,14,U,DT=E(DATETIME)`	A value composed of DATE and TIME
`1,TS,20,U,DT=E(TIMESTAMP)`	A value composed of DATETIME plus microseconds
`1,TT,7,P,DT=E(NATTIME)`	Natural T-format data (tenths of seconds since the year zero)
`1,DD,4,P,DT=E(NATDATE)`	Natural D-format data (days since the year zero)
`1,UU,4,F,DT=E(UNIXTIME)`	UNIX time_t-type data (seconds since January 1, 1970)
`1,XS,8,F,DT=E(XTIMESTAMP)`	UNIX time data (microseconds since January 1, 1970)

Time Zone Option TZ

The TZ field option identifies a date-time field that should be presented in the user's local time and stored in UTC time, allowing for differences in time zones. There is no specific syntax for the TZ field option as there are no parameters; simply specifying TZ in the field definition of a date-time field provides time zone support.

When TZ is specified, date-time values are converted and displayed in the user's local time, but are stored in *coordinated universal (UTC) time*. This allows users in different time zones to view the data in their individual local times, but still share the same data. Storing values in standardized UTC time makes them easily comparable.

Adabas uses the time zone data taken from the tz database, which is also called the *zoneinfo* or *Olson* database. The specific list of time zone names that Adabas supports in any given release can be found in the TZINFO member of the Adabas time zone library (ADA*vrs*.TZ00). For more information about the TZINFO member of the time zone library, read *Supported Time Zones*.

The TZ option can be specified in field definitions that use the following date-time edit masks:

- DATETIME
- TIMESTAMP
- NATTIME
- UNIXTIME
- XTIMESTAMP

You cannot use the TZ option in field definitions that use the DATE, TIME, or NATDATE date-time edit masks because the timezone offsets depend on the presence of both date and time values in the data.

Note that UNIXTIME and XTIMESTAMP fields are by definition based on the UTC; standard conversion routines will perform time zone handling outside of Adabas. In other words, the TZ option has no effect when reading or writing fields with the UNIXTIME or XTIMESTAMP edit mask.

However, when the DATETIME, NATTIME, and TIMESTAMP edit masks are set in the format buffer, the TZ option will convert the times to local time; otherwise they will be converted and returned as UTC times.

For example, if a date-time field is stored in UTC format is February 14, 2009, 16:00 hours, user A in time zone America/New_York will see the field displayed as February 14, 2009, 11:00 hours or 10:00 (UTC time minus 5 or 6 hours, depending on the differences in daylight savings time). Alternately, user G in time zone Europe/Berlin will see the field as February 14, 2009, 17:00 hours (UTC time plus 1).

For information on the conversions between date-time fields defined with the TZ option, read *Conversions Between Date-Time Representations for Fields with the TZ option*.

Data Compression Options FI and NU

Default data compression is described in the section *Compression*. At the field level, additional compression can be specified (null suppression option) or all compression can be disabled (fixed storage option).

Null suppression (NU) differs from default compression in that searches on descriptor fields defined with null suppression do *not* return records in which the descriptor field is empty.

Fields defined as fixed format (FI) do not include a length byte and are not compressed. This option actually saves storage space for one-byte fields or fields that are nearly always full (e.g., a field containing the social security number).

Encoding Conversion Option NV

Alphanumeric (A) or wide-character (W) format fields with the NV option are processed in the record buffer without being converted to or from the user.

The field has the characteristics of the file encoding; that is, the default blank:

- for A fields is always the EBCDIC blank (X'40'), and
- for W fields is always the blank in the file encoding for W format.

The NV option is used for fields containing data that cannot be converted meaningfully or should not be converted because the application expects the data exactly as it is stored.

The field length for NV fields is byte-swapped if the user architecture is byte-swapped.

Long Alpha Option LA

The long alphanumeric (LA) option can only be specified for variable-length alphanumeric or wide-character fields; i.e., A- or W-format fields having a length of zero. With the LA option, such an alphanumeric or wide-character field can contain a value up to 16,381 bytes long.

An alpha or wide field with the LA option is compressed in the same way as an alpha or wide field without the option. The maximum length that a field with LA option can actually have is restricted by the block size where the compressed record is stored.

In Adabas 8 (or later), the NB (no blank compression) option can be specified for LA fields to control blank suppression.

LA fields cannot also be defined with the LB field option. To assist you in determining whether to define a field as an LA or an LB field, read *Comparing LA and LB Fields*.

Large Object Option LB

The large object (LB) option can be specified for some fields to identify them as *large object* fields. LB fields can contain up to 2,147,483,643 bytes (about 2 GB) of data.

The format of an LB field must be "A" (alphanumeric) and its default field length must currently be defined as zero.

LB fields *cannot* be:

- Descriptors or parents of a special (phonetic, sub-, super-, or hyper-) descriptor.
- Defined with the FI or LA options.

To assist you in determining whether to define a field as an LA or an LB field, read *Comparing LA and LB Fields*.

- Specified in a search buffer or in format selection criteria in a format buffer.

LB fields may be:

- Defined with any of the following options: MU, NB, NC, NN, NU, or NV
- Part of a simple group or a PE group.

The presence of the NB (no blank compression) field option in the LB field definition indicates whether on not Adabas removes trailing blanks in LB fields containing characters.

LB fields containing both binary and character data are supported. An LB field defined with both the NV and NB options can store binary large object data, as Adabas will not modify binary LB fields in any way. The identical LB binary byte string that was stored is what is retrieved when the LB field is read. In addition, because LB fields containing binary values are defined with the NV and the NB options, Adabas will not convert LB field binary values according to some character code page nor will it cut off trailing blanks in LB fields containing binary values.

Note:
LB fields containing binary values are not defined using format B, because format B can imply byte swapping in some environments with different byte orders. Byte swapping does not apply to binary LB fields.

The following table provides some valid example of FDT definitions for LB fields:

FDT Specification	Description
`1,L1,0,A,LB,NU`	Field L1 is a null-suppressed, character, large object field
`1,L2,0,A,LB,NV,NB,NU,MU`	Field L2 is a null-suppressed, multiple-value, binary, large object field.

Commands dealing with LB fields must always be directed to the *base file* of a *LOB file group*. User commands against *LOB files* are rejected.

For information on getting started using LB fields, read *Large Object (LB) Files and Fields*.

Comparing LA and LB Fields

The following table comparing pertinent LA and LB field features may help you decide which to use when defining fields for your database.

Feature	LA Field Behavior	LB Field Behavior
Zero field length specification in format buffers	Two bytes in the corresponding record buffer area are used to store the actual length of the LA field.	Four bytes in the corresponding record buffer area are used to store the actual length of the LB field.

Feature	LA Field Behavior	LB Field Behavior
Data record storage	Alphanumeric and wide-character fields are stored within the compressed record. All long values must fit into the same compressed record. The maximum length of simple or spanned data records limits the number and lengths of long values that can be stored. This can be a problem if multiple long values are contained in a record.	Some LB field values (those larger than 253 bytes) are stored offline in a separate large object file (the LOB file) and only references to the LB field values in the LOB file are included in the data record. This allows for storing more long objects for a single data record than using normal or LA fields. However, the performance overhead at runtime and for file maintenance is increased for LB fields because of this behavior. Smaller LB field values (up to 253 bytes) are stored directly in the compressed record. This improves performance for small values, but also limits the number of small LB field occurrences that can be stored in the same compressed record.
Asterisk (*) field length notation in format buffers	Supported for LA fields of any length.	Supported for LB fields of any length.
Maximum length of any stored object does *not* exceed 16,381 bytes	Alphanumeric or wide-character LA field can be used. This avoids the overhead of LB fields, but limits the number of such fields that can be stored in a single record.	Alphanumeric LB field can be used.
Maximum length of any stored object exceeds 16,381 bytes	Not supported.	Supports objects with sizes larger than 16,381 bytes.
So many large objects that they will not fit in a single simple or spanned data record	Not supported.	Supports multiple large objects.

MU and PE Options and Field Types

Adabas supports two basic field types: elementary fields and multiple-value fields. An *elementary* field has only one value per record. *Multiple-value* (MU) fields can have 191 up to about 65,534 values, or occurrences, in a single record. The use of more than 191 MU fields or PE groups in a file must be explicitly allowed for a file (it is not allowed by default). This is accomplished using the ADADBS MUPEX function or the ADACMP COMPRESS MUPEX and MUPECOUNT parameters. Each multiple-value field has a *binary occurrence counter* (BOC) that stores the number of occurrences.

A *periodic (PE) group field* defines consecutive fields in the FDT that repeat together in a record. Like the members of a non-periodic group field, PE members immediately follow the PE group field, have a higher level number than the PE field, and can be accessed both individually and as a group. Each PE has a BOC that stores the number of occurrences.

A periodic group may be repeated 191 or up to about 65,534 times per record and may contain one or more multiple-value fields. The use of more than 191 MU fields or PE groups in a file must be explicitly allowed for a file (it is not allowed by default). This is accomplished using the ADADBS MUPEX function or the ADACMP COMPRESS MUPEX and MUPECOUNT parameters. Occurrences or values that are not used require no storage space.

Adabas thus supports four field types:

	Single Value per Record	**Multiple Values per Record**
Single Field	Elementary	MU
Multiple Fields	Group	PE

The actual limit to the number of occurrences of MU fields and PE groups in a file is derived from the maximum data storage record length (the ADALOD MAXRECL parameter), which defaults to the size of the data storage block minus 4.

The number of occurrences of each MU field or each PE group in a record can be increased from 191 to about 65,534 using the ADADBS MUPEX function or the ADACMP COMPRESS MUPEX and MUPECOUNT parameters. However, the actual limit is derived from the maximum Data Storage record length (the ADALOD MAXRECL parameter), which defaults to the size of the Data Storage block minus 4, the device type, and the file type (spanned or unspanned). All MU fields and PE groups and other fields must fit into one compressed record. If you are using spanned records (introduced with Adabas 8), more MU fields and PE groups can be stored.

In addition, subdescriptors and superdescriptor definitions can affect the number of MU fields or PE groups in the record. For example, if a superdescriptor is created as a combination of a PE group and one or more MU fields and the number of occurrences is high, performance and resource problems can occur.

Note:
Excessive use of extended MU and PE fields might cause performance and resource problems. These can result in a work storage overflow, resulting in response code 9 (ADARSP009). If this should happen, increase the ADARUN LP size for the database.

All MU fields and PE groups and other fields must fit into one compressed record. If you are using spanned records (introduced with Adabas 8), more MU fields and PE groups can be stored.

The following figure illustrates the four field types in a single record structure.

CUSTOMER NUMBER	FIRST NAME	LAST NAME	PE BOC	MU BOC	STREET	CITY	STATE	ZIP
19811	Laura	Cagnetti	3	1	118 Glade	Erie	PA	16509
				2	271 Larue	Cincinnati	OH	45211
					P.O. Box 88			
				2	733 Hall	Easton	PA	19014
					P.O. Box 7			

Elementary Field

Multiple Value Field

Group Field (Name)

Periodic Group (Address)

A PE field cannot be nested within another PE group. Nesting an MU field within a PE group, as shown in the figure above, is permitted but complicates programming by introducing a two-dimensional array. It also has implications for data access: when Adabas accesses the periodic group, it returns only the first occurrence of the MU for each occurrence of the PE returned.

The unique characteristic of the periodic group and the reason for choosing the periodic group structure is its ability to maintain the order of occurrences. If a periodic group originally contains three occurrences and the first or second occurrence is later deleted, those occurrences are set to nulls; the third occurrence remains in the third position. This contrasts with the way leading null entries are handled in multiple-value fields. The individual values in a multiple-value field do not retain positional integrity if one of the values is removed.

If a file has been established with extended MU or PE limits, you should not read the occurrence count of an MU field or PE group into a one-byte field in the record buffer. If you try, Adabas returns response code 55 (ADARSP055), subcode 9. Therefore, any application program that reads the occurrence count using an `xxC` element in the format buffer (for example, FB='MUC.' or FB='MUC,1,B.') must be changed to read the occurrence count into a field with two or more bytes (for example, FB='MUC,2,B.' or FB='MUC,4,B.').

Blank Compression Option NB

The NB option can be used with LA and LB fields to control blank compression. When specified, the NB option indicates that Adabas should *not* remove trailing blanks for the field; when not specified, Adabas removes trailing blanks when storing an alphanumeric or wide-character field value. If you specify the NB option for a field, you must also specify the NU or NC option for the field; NB processing requires the use of NC or NU as well.

Note:
Fields specified without the NB option can lead to differences in the stored and retrieved lengths of the fields. The retrieved length of a non-NB field is likely to be smaller than the length specified for the field when it is stored due to blank compression. This may matter if the value if not really a character string, but rather a binary value that happens to end with the character codes for a blank. Therefore, if you want the stored and retrieved lengths of a field to be the same, use the NB option.

Adabas does not allow the storing of a field with a length of zero unless the field has been defined with the NB option. One way to specify a length of zero for a large object (LOB) field L1 would be, for instance, via `FB='L1L,4,B,L1,*,A.'` and `RB=x'00000000'`. Adabas accepts this construction if field L1 has been defined with the NB-option (for example, `L1,0,A,LB,NU,NB`) and will return the length of zero if the field is later read. However, if the field has been defined *without* the NB option (for example, `L1,0,A,LB,NU`), Adabas rejects the attempt to store the zero-length value with response code 52 (ADARSP052), subcode 2. Without the NB option, an empty field should be stored as a single blank (for example, `FB='L1L,4,B,L1,*,A.'` and `RB=x'0000000140'`).

For examples of this, read *Read Operations, Length Indicators, and the NB (No Blank Compression) Option*.

SQL Compatibility Options NC and NN

Special data definition options are included in Adabas to accommodate Software AG's mainframe Adabas SQL Gateway (ACE) and other structured query language (SQL) database query languages that require SQL-compatible null representation.

A field designated with the NC (not counted) option may contain a null value that satisfies the SQL interpretation of a field having no value. An NC field containing a null means that *no field value has been entered*; that is, the field's value is not defined.

This undefined state differs from a null value assigned to a non-NC field for which no value has been specified: a non-NC field's null means the value in the field is either zero or blank, depending on the field's format.

The NN (not null) option can be specified only for NC-defined fields. It indicates that an NC field must always have a value defined; it cannot contain an SQL null. This ensures that the field cannot be left undefined when a record is either created or updated. The field value may be zero or blank, however.

Special Fields and Descriptor Fields

The FDT indicates whether a field is a parent field for a collation descriptor, subfield, superfield, subdescriptor, superdescriptor, hyperdescriptor, or phonetic descriptor. Information about any special fields and descriptors (collation descriptors, subdescriptors, subfields, superdescriptors, superfields, phonetic descriptors, and hyperdescriptors) in a file is maintained in the special descriptor table (SDT) part of the FDT.

```
  SPECIAL DESCRIPTOR TABLE

        I      I        I        I                       I                I
  TYPE  I NAME I LENGTH I FORMAT I        OPTIONS        I   STRUCTURE    I
        I      I        I        I                       I                I
 -------I------I--------I--------I-----------------------I----------------I
        I      I        I        I                       I                I
  SUPER I  H1  I    4   I    B   I DE,NU                 I AU (  1 -  2)  I
        I      I        I        I                       I AV (  1 -  2)  I
  SUB   I  S1  I    4   I    A   I DE                    I AO (  1 -  4)  I
  SUPER I  S2  I   26   I    A   I DE                    I AO (  1 -  6)  I
        I      I        I        I                       I AE (  1 - 20)  I
  SUPER I  S3  I   12   I    A   I DE,NU,PE              I AR (  1 -  3)  I
        I      I        I        I                       I AS (  1 -  9)  I
        I      I        I        I                       I                I
  PHON  I  PH  I        I        I                       I PH =PHON(AE)   I
        I      I        I        I                       I                I
  COL   I  Y1  I   20   I    W   I DE                    I CDX  8,PA      I
  COL   I  Y2  I   12   I    A   I DE,NU,PE              I CDX  1,AR      I
        I      I        I        I                       I                I
        I      I        I        I                       I                I
 -------------------------------------------------------------------------
```

Along with the name, length, format, and specified options of each special field and descriptor, this table provides the following information:

Column	Explanation
TYPE	COL Collation descriptor HYPER Hyperdescriptor PHON Phonetic descriptor SUB Subfield/subdescriptor SUPER Superfield/superdescriptor
STRUCTURE	The component fields and field bytes of the sub-, super-, or hyperdescriptor. Phonetic descriptors show the equivalent alphanumeric elementary fields. Collation descriptors show the associated collation descriptor user exit and the name of the parent field.

This section describes the special fields and descriptors:

- Collation Descriptor
- Hyperdescriptor
- Phonetic Descriptor
- Subfield / Superfield
- Subdescriptor
- Superdescriptor

Collation Descriptor

An alphanumeric or wide-character field can be defined as a parent field of a collation descriptor. A collation descriptor is used to sort field values in a special user-defined sequence. The LF command reports the collation descriptor field information.

A collation descriptor is assigned a collation descriptor user exit (1-8) which encodes the collation descriptor value and decodes it back to the original field value. The ADARUN parameter CDXnn is used to specify collation descriptor user exits.

Hyperdescriptor

The hyperdescriptor option can be used to generate descriptor values based on a user-supplied algorithm. Up to 31 different hyperdescriptors can be defined for a single physical Adabas database. Each hyperdescriptor must be named by an appropriate HEXnn ADARUN statement parameter in the job where it is used.

With hyperdescriptors, fuzzy matching is possible; i.e., retrieving data based on *similar* rather than on *exact* search criteria. Hyperdescriptors allow multiple virtual indexes, meaning that several different search index entries can be made for a single data field.

Hyperdescriptors can be used to implement n-component superdescriptors, derived keys, or other key constructs. Using hyperdescriptors, it is possible to develop applications that are simpler and more flexible than applications based on a strictly normalized relational structure.

One application area for hyperdescriptors is name processing. For example, the name SCHROEDER could be stored not only with the index SCHROEDER itself, but also with the virtual indexes SCHRODER, SCHRADER, or any other variation of the name. Thus, although only the name SCHROEDER is physically stored in the data area of the database, multiple search indexes exist to the data. If, subsequently, a search is made for the name SCHRODER, the record SCHROEDER will be found.

A more sophisticated application area for hyperdescriptors is fingerprint matching, in which typical characteristics of fingerprints can form the basis of a fuzzy matching algorithm; i.e., the original fingerprint is stored in the database, but any number of search indexes can be made to the fingerprint, based on an algorithm that allows small-scale deviations from the original.

Phonetic Descriptor

A phonetic descriptor may be defined and used to search for all records that contain similar phonetic values. The phonetic value of a descriptor is determined by an internal algorithm based on the first 20 bytes of the field value with only alphabetic values being considered (numeric values, special characters and blanks are ignored).

Subfield / Superfield

A portion of a field (subfield) or any combination of fields (superfield) may be defined as an elementary field (read *MU and PE Options and Field Types*). Subfields and superfields may be used for read operations only. They may only be changed by updating the original fields.

Subdescriptor

A subdescriptor is part of a single field used as a descriptor. The field from which the subdescriptor is derived may or may not be an elementary descriptor (read *Descriptor Options DE, UQ, and XI*. If a search criteria involves a range of values contained in the first n bytes of an alphanumeric field or the last n bytes of a numeric field, a subdescriptor may be defined using only the relevant bytes of the field. A subdescriptor allows you to increase the efficiency of a search by specifying a single value rather than a range of values.

For example, if the first two bytes of a five-byte field refer to a geographical region and you want to retrieve all records for region 11 without using a subdescriptor, you would have to search for all records in the range 11000-11999. If you define a subdescriptor comprising the first two bytes of the field, you could search for all records with 11 in the subdescriptor.

Superdescriptor

A superdescriptor combines all or parts of 2-20 fields. The fields from which the superdescriptor is derived may or may not be elementary descriptors. When search criteria involve values for a combination of fields, using a superdescriptor is more efficient than using a combination of several elementary descriptors.

For example, to search for customers by last name within regions, you could create a superdescriptor by combining the first two bytes (i.e., the geographical region indicator) of the five-byte customer number field and the entire customer last name field.

For complete information about defining superdescriptors, read *SUPDE: Superdescriptor Definition* in the ADACMP documentation .

Spanned Records

With Adabas 8, records can be spanned in a database. In the database, the logical record is split into a number of physical records, each part fitting into a single Data Storage (DS) block. Spanned records may be segmented at the field or byte level. The resulting physical records are each assigned individual ISNs. The first physical record is called the *primary record* and contains the beginning of the compressed record and is assigned a *primary ISN*. The remaining physical records are called *secondary records* and contain the rest of the data of the logical record. Secondary records are assigned *secondary ISNs*. These ISNs do not affect the user ISNs assigned when using the N2 command or the ISNs used when using the I option of the L1 command. If spanned records are used, a secondary address converter is used to map the secondary ISNs to the RABNs of the Data Storage blocks where the secondary records are stored.

A spanned record is comprised of one primary record and one or more secondary records. However, the number of segments in a spanned record is limited. The Adabas nucleus allows up to five physical records (one primary record and four secondary records) in a spanned record.

Spanned records are not directly visible to application programs. Applications always address spanned records via the primary ISN.

Spanned records are also supported in expanded Adabas files and in multi-client files.

Note:
Spanned record support must be explicitly allowed for a file. You can do this using the ADADBS RECORDSPANNING function or the SPAN parameter of ADACMP COMPRESS. For more

information, read the *Adabas Utilities Manual* documentation for the ADADBS and ADACMP utilities.

This section covers the following topics:

- Spanned Record Structure
- Allowing Spanned Records in Files
- Secondary Record Segmentation
- Padding Factors
- Spanned Record ISN Use
- ADARUN Parameters Affected
- Reporting on Spanned Records
- Securing Spanned Records

Spanned Record Structure

A spanned logical record is comprised of one or more physical records, including a single primary record and one or more secondary records. The number of records that comprise a spanned record is limited. The Adabas nucleus allows up to five physical records (one primary record and four secondary records) in a spanned record.

The primary and secondary records in a spanned record are connected using their ISNs. The header of each physical record contains the ISN of the current record, the ISN of the primary record, as well as the ISN of the next secondary record. In addition, the header indicates whether the current record is the primary record or a secondary record.

The header of each physical record also provides the length of the record -- even if it is a segmented record (in which case, it is the length of the segment).

Allowing Spanned Records in Files

Files can contain spanned records only if it has been explicitly requested via the SPAN parameter of ADACMP COMPRESS, the RECORDSPANNING function of ADADBS or the equivalent Adabas Online System function. The ADAREP database report and the Adabas Online System report functions indicate whether or not a file has been defined to allow spanned records.

The SPAN attribute of a file is retained in an ADAULD UNLOAD function. In other words, when a file is unloaded, deleted, and reloaded, its support for spanned records remains unchanged.

Similar rules hold for files that allow more than 191 MU or PE occurrences. For more information on identifying MU and PE occurrences greater than 191 in a compressed record, read *Identifying MU and PE Occurrences Greater Than 191 in Compressed Records*.

Secondary Record Segmentation

Secondary records are segmented either by field or by byte. For performance reasons, segmentation is done by field whenever possible. However, when any non-LB (large object) type field is larger than the data storage block size, the record is split at the byte level. If a field is larger than the remaining space in the data storage block, but smaller than the data storage block size, than the field is split at the field level and not at the byte level. The header of each secondary record indicates which type of segment record it is.

Padding Factors

Padding factors are generally ignored for spanned records, in an attempt to fully use the block. So it is frequently listed as zero on reports. The padding factor is only used in the last, short, segment of a spanned record.

Spanned Record ISN Use

Primary and secondary records are addressed by Adabas using address converters (AC). However, the primary address converter maps only the ISNs of primary records to the RABNs of their corresponding Data Storage blocks. If spanned records are used, a secondary address converter is used to map the secondary ISNs to the RABNs of the Data Storage blocks where the secondary records are stored. Therefore, spanned records have no affect on the index structure, since there is still only one index for each record.

Separate ISN ranges are maintained for primary and secondary ISNs. Wherever an ISN is stored or handled, it distinguishes between whether the action is for a primary or a secondary ISN.

All commands should be specified using the primary record’s ISN; secondary record ISNs are kept hidden and cannot be used. Physical sequential commands will automatically skip the secondary records in Data Storage. Read commands that specify secondary ISNs will receive an error (response code 113, ADARSP113).

The ISN of the primary records are included in TOPISN and MAXISN values. Secondary record ISNs are not. Secondary ISNs are included in the MINSEC and MAXSEC values instead. A file containing spanned records can be loaded by specifying an MINISN value, but the MINISN must refer only to a primary record ISN (never a secondary record ISN).

ADARUN Parameters Affected

The following ADARUN parameters may need to be changed to support files with spanned records.

- The number of ISNs in the hold queue per user (NISNHQ parameter) may need to be increased as the number of spanned records to be updated also increases.
- The length of the Adabas work pool (LWP) may also need to be increased since space is needed to store both the before and after image of the spanned record and to support several update threads running in parallel. Space may also be needed to accommodate larger descriptor value tables (up to 65,534 occurrences of descriptors in PE groups are permitted).
- The SRLOG parameter indicates how spanned records are logged to the protection logs (PLOGs). Complete or partial logging can occur.

For complete information on these and other ADARUN parameters, read *Adabas Initialization (ADARUN Statement)*.

Reporting on Spanned Records

Maximum record length statistics have no relevance with spanned files. Utilities that report on the maximum record length will now report that the statistics as "N/A" (not applicable). The FCB will contain high values in the maximum record length field for a file that is using spanned records.

Securing Spanned Records

Files containing spanned records can be ciphered and protected with security-by-value. If the primary record's ISN is referenced, all secondary segment records must be read, and therefore, processing is time-sensitive.