Database Design

This part of the DBA task description contains information about and guidelines for database design. Topics discussed include performance, file structure, record design, efficient use of descriptors, use of the Adabas direct access method (ADAM), disk storage space techniques, database recovery and restart procedures, and security.

This information is organized under the following headings:

- Performance Control During System Design
- File and Record Design
- Data Access Strategies
- Disk Space Usage
- Adabas Security
- Recovery/Restart Design
- The Adabas Recovery Aid
- Multiclient Support

Performance Control During System Design

The performance of a system is measured by the time and computer resources required to run it. These may be important for the following reasons:

- Some system functions may have to be completed within a specified time frame;
- The system may compete for computer resources with other systems that have more stringent time constraints.

Performance may not be the most important objective. Trade-offs often need to be made between performance and

- flexibility;
- data independence;
- accessibility of information;
- audit and security considerations;
- currency of information;
- ease of scheduling and impact on concurrent users of the database; or

- disk space.

In some cases, performance may be a constraint to be met rather than an objective to be optimized. If the system meets its time and volume requirements, attention may be turned from performance to other areas.

This section covers the following topics:

- Methodology for Performance Control in System Design

Methodology for Performance Control in System Design

The need to achieve satisfactory performance may affect

- the design of the database;
- the options defined when first loading the database;
- the logic of application functions (for example, whether to use direct access or a combination of sequential accesses and sorts); and/or
- operation procedures and scheduling.

Performance requirements must be considered early in the system design process. The following procedure may be used as a basis for controlling performance:

1. Obtain from the users the *time* constraints for each major system function. These requirements are likely to be absolute; that is, the system must meet them.
2. Obtain the computer *resource* constraints from both the users and operations personnel and use the most stringent set.
3. Describe each function in terms of the logical design model specifying the
   - manner in which each record type is processed;
   - access path and the sequence in which records are required;
   - frequency and volume of the run;
   - time available.
4. *Decide* which programs are most performance critical. The choice may involve volumes, frequency, deadlines, and the effect on the performance or scheduling of other systems. Other programs may also have minimum performance requirements which may constrain the extent to which critical functions can be optimized.
5. *Optimize* the performance of critical functions by shortening their access paths, improving their logic, eliminating database features that increase overhead, and so on. On the first pass, an attempt should be made to optimize performance without sacrificing flexibility, accessibility of information, or other functional requirements of the system.
6. *Estimate* the performance of each critical function. If this does not give a satisfactory result, either a compromise between the time constraints and the functional requirements must be found or a hardware upgrade must be considered.

7. *Estimate* the performance of other system functions. Calculate the total cost and compare the cost and peak period resource requirements with the economic constraints. If the estimates do not meet the constraints, then a solution must be negotiated with the user, operations, or senior management.

8. If possible, *validate* the estimates by loading a test database and measuring the actual performance of various functions. The test database should be similar to the planned one in terms of the number of records contained in each file and the number of values for descriptors. In the test database, the size of each record is relatively unimportant except when testing sequential processing, and then only if records are to be processed in physical sequence.

File and Record Design

It is possible to design an Adabas database with one file for each record type as identified during the conceptual design stage. Although such a structure would support any application functions required of it and is the easiest to manipulate for interactive queries, it may not be the best from the performance point of view, for the following reasons:

- 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 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 and therefore can result in the overwriting of blocks needed for a later request.

For the above reasons, it may be advisable to reduce the number of Adabas files used by critical programs. The following techniques may be used for this procedure:

- Using multiple-value fields and periodic groups;
- Including more than one type of record in an Adabas file;
- Linking physical files into a single logical (expanded) file;
- Controlling data duplication (and the resulting high resource usage).

Each of these techniques is described in the following sections.

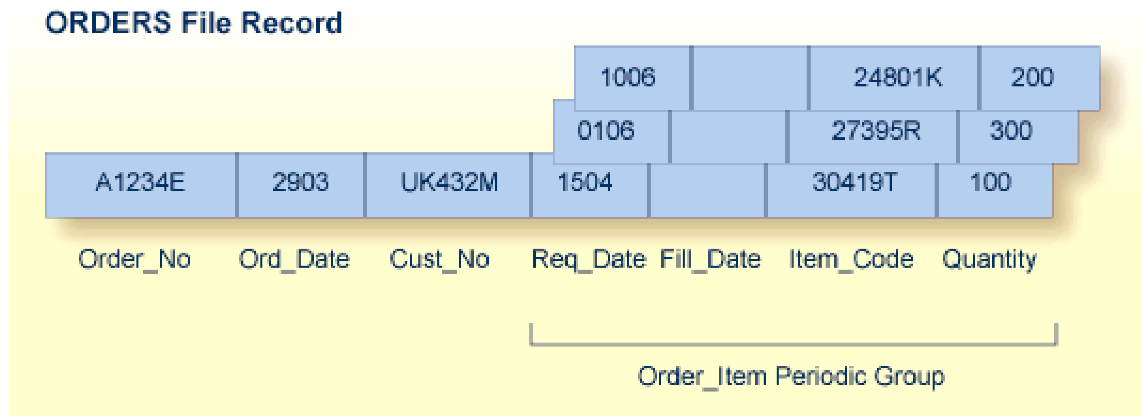
This section covers the following topics:

- Multiple-Value Fields and Periodic Groups
- Different Record Types in a Single Adabas File
- Linking Physical Files in a Single Logical File
- Data Duplication
- Adabas Record Design

Multiple-Value Fields and Periodic Groups

The simple example below shows the practical use of a periodic group:

Order Number	Order Date	Date Filled	Customer	Date Required	Item Code	Quantity
A1234E	29MAR	--	UK432M	10JUN	24801K	200
		--		15APR	30419T	100
		--		01JUN	273952	300



Example of a Periodic Group

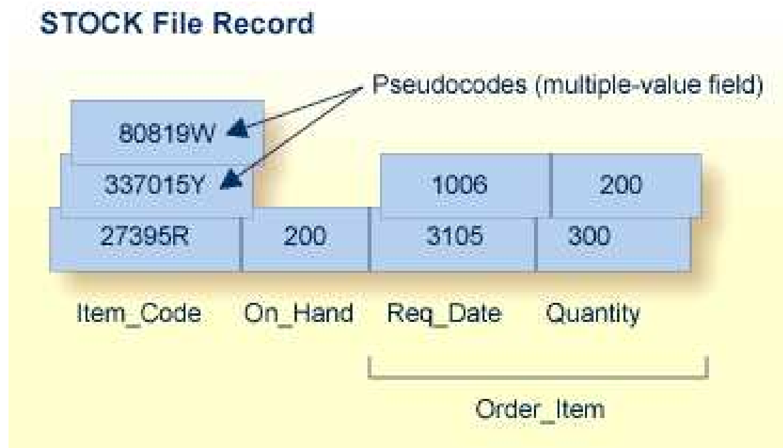
In the example shown in the table above, the order information in the table is shown converted to a record format in an Adabas file called ORDERS. Each order record contains a periodic group to permit a variable number of order items. In this case, the periodic group ORDER_ITEM, comprising the ITEM_CODE field (order item code) and the related fields QUANTITY (quantity desired), REQ_DATE (date required), and FILL_DATE (actual date the order was filled), can specify up to 65,534 different items in a given record. Each unique occurrence of the ORDER_ITEM periodic group is called an *occurrence*; up to 65,534 occurrences per periodic group are possible.

Note:
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.

The unique characteristic of the periodic group-the ability to maintain the order of occurrences-is the reason for choosing the periodic group structure. If a periodic group originally contained 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, discussed below.

Note also that the record format shown for the ORDERS file may not seem the most logical; however, fields most likely to contain nulls should be placed together and at the end of the record to save database space. The fields comprising periodic groups, therefore, are combined after the other fields in the record.

On the other hand, the ORDERS file record structure, while being appropriate for managing orders, may not as desirable when managing inventory. A stock control application for the items in the ORDER file can require a completely different record structure. These records are kept in a different database file called STOCK (see the figure below).



Example of a Multiple-Value Field

The record format in STOCK is more suitable to the applications required for stock management than the format in the ORDERS file. The record is designed to handle cases where an item is designated as a replacement for another that is no longer in the inventory. By allowing multiple values for the ITEM_CODE field, the current stock item can also be labeled with the numbers of discontinued items that the new item replaces, allowing references to the old items to automatically select the new replacement item. To do this, the ITEM_CODE field is defined as a multiple-value field.

For example, the items 80819W and 337015Y are no longer in stock; their item codes have become synonyms for the basic item 27395R. An application program that inquires about either discontinued item can first look through all ITEM_CODE values for the old code, and then refer to the first ITEM_CODE value in the multiple-value field to identify the replacement.

The ITEM_CODE field may contain from one to 65,534 values, depending on the settings for the file. Unlike a periodic group, however, the individual values in a multiple-value field do not keep positional integrity if one of the values is removed. For example, if the item 337015Y in the STOCK record shown above can no longer be ordered and the pseudocode is set to a null, 80819W automatically becomes the second occurrence under ITEM_CODE.

The following limits apply when using multiple-value fields or periodic groups:

- The maximum number of values of any multiple-value (MU) field can be up to 65,534. The actual number of occurrences allowed must be explicitly set 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.

- The maximum number of occurrences of any one periodic group (PE) can be up to 65,534. The actual number of occurrences allowed must be explicitly set 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.

- A periodic group cannot contain another periodic group.
- Depending on the compressed size of one occurrence, their usage can result in extremely large record sizes which may be larger than the maximum record size supported by Adabas.

Descriptors contained within a periodic group and subdescriptors or superdescriptors derived from fields within a periodic group cannot be used to control logical sequential reading or as a sort key in find and sort commands. In addition, specific rules apply to the ways in which search requests involving one or more descriptors derived from multiple-value fields and/or contained within a periodic group may be used. These rules are described in the Adabas Utilities documentation, ADACMP utility.

Different Record Types in a Single Adabas File

Another method of reducing the number of files is to store data belonging to two logical record types in the same Adabas file. The following example shows how a customer file and an order file might be combined. This technique takes advantage of Adabas null-value suppression.

```
Fields in the field definition table for the combined file:

Key, Record Type, Order Data, Order Item Data

Stored records:

Key Type Order Data*

Key Type * Order Item Data

  * indicates suppressed null values.
```

The key of an order item record could be order number plus line sequence number within this order.

This technique reduces I/O operations by allowing the customer and order record types to share control blocks and higher-level (UI) index blocks. Fewer blocks have to be read before processing of the file can start, and more space is left free in the buffer pool for other types of blocks.

The customer and order records can be grouped together in Data Storage, reducing the number of blocks that have to be read to retrieve all the orders for a given customer. If all the orders are added at the same time the customer is added, the total I/O operations required will also be reduced. If the orders are added later, they might not initially be grouped in this way but they can be grouped later by using the ADAORD utility.

The key must be designed carefully to insure that both customer and order data can be accessed efficiently. To distinguish different orders belonging to the same customer, the key for a customer record will usually have the null value of the suffix appended to it, as shown below:

```
A00231        000       Order header for order A00231
A00231        001       Order item 1
A00231        002       Order item 2
A00231        003       Order item 3
A00232        000       Order header for order A00232
A00232        001       Order item 1
```

A record type field is unnecessary if the program can tell whether it is dealing with a customer or order record by the contents of the key suffix. It may be necessary for a program to reread a record to read additional fields or to return all fields that are relevant to any of the record types.

Linking Physical Files in a Single Logical File

An Adabas file with three-byte ISNs can contain a maximum of 16,777,215 records; a file with four-byte ISNs can contain 4,294,967,294 records. 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 version 6 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 according to 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.

Adabas utilities do not always recognize expanded files; that is, some utility operations automatically perform their functions on all component files, and others recognize only individual physical files. See *Expanded Files* for more information.

Data Duplication

Data duplication can be part of database design in either of two ways:

- **Physical duplication:**

  In some cases, a few fields from a header record are required almost every time a detail record is accessed. For example, the production of an invoice may require both the order item data and the product description which is part of the product record. The simplest way to make this information quickly available to the invoicing program is to hold a copy of the product description in the order item data. This is termed physical duplication because it involves holding a duplicate copy of data which is already physically represented elsewhere-in this case, in the product record. Physical duplication can also be in effect if some fields from each detail record are stored as a periodic group in a header record.

  Physical duplication seldom causes much of a problem if it is limited to fields that are updated only infrequently. In the example above, the product description data rarely changes; the rule is: the less activity on duplicated fields, the better.

- **Logical duplication:**

  Assume a credit control routine needs the sum of all invoices present for a customer. This information can be derived by reading and totaling the relevant invoices, but this might involve random access of a large number of records. It can be obtained more quickly if it is stored permanently in a customer record that has been correctly maintained. This is termed *logical*

duplication because the duplicate information is not already stored elsewhere but is implied by the contents of other records.

Programs that update physically or logically duplicated information are likely to run more slowly because they must also update the duplicate copies. Logical duplication almost always requires duplicate updating because the change of any one record can affect data in other records. Logical duplication can also cause severe degradation in a TP environment if many users have to update the same record.

Adabas Record Design

Once an Adabas file structure has been determined, the next step is usually to define the fields for the file. The field definitions are entered as input statements to the ADACMP utility's COMPRESS function, as described in the Adabas Utilities documentation This section describes the performance implications of some of the options that may be used for fields.

The fields of a file should be arranged so that those which are read or updated most often are nearest the start of the record. This will reduce the CPU time required for data transfer by reducing the number of fields that must be scanned. Fields that are seldom read but are mainly used as search criteria should be placed last.

For example, if a descriptor field is not ordered first in the record 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.

Combining Fields

If several fields are always read and updated together, CPU time can be saved by defining them as one Adabas field. The disadvantages of combining fields in this way are:

- More disk space may be required since combining fields may reduce the possibilities for compression;
- It may be more difficult to manipulate such fields in query language programs such as SQL.

Using Field Groups

The use of groups results in more efficient internal processing of read and update commands. This is the result of shorter format buffers in the Adabas control block. Shorter format buffers, in turn, take less time to process and require less space in the internal format buffer pool.

Numeric Fields

Numeric fields should be loaded in the format in which they will most often be used. This will minimize the amount of format conversion required.

Fixed-Storage Option

The use of the fixed storage (FI) option normally reduces the processing time of the field but may result in a larger disk storage requirement, particularly if the field is contained within a periodic group. FI fields, like NU fields, should be grouped together wherever possible.

Data Access Strategies

This section describes various data access strategies available in Adabas. It covers the following topics:

- Efficient Use of Descriptors
- Collation Descriptor
- Superdescriptor
- Subdescriptor
- Phonetic Descriptor
- Hyperdescriptor
- File Coupling
- User-Assigned ISNs
- Using the ISN as a Descriptor
- ADAM Usage

Efficient Use of Descriptors

Descriptors are used to select records from a file based on user-specified search criteria and to control a logical sequential read process. The use of descriptors is thus closely related to the access strategy used for a file. Additional disk space and processing overhead are required for each descriptor, particularly those that are updated frequently. The following guidelines may be used in determining the number and type of descriptors to be defined for a file:

- If data in certain fields needs to be resequenced before processing on the field can continue, a collation descriptor can be defined.
- The distribution of values in the descriptor field should be such that the descriptor can be used to select a small percentage of records in the file;
- Additional descriptors should not be defined to further refine search criteria if a reasonably small number of records can be selected using existing descriptors;
- If two or three descriptors are used in combination frequently (for example, area, department, branch), a superdescriptor may be used instead of defining separate descriptors;
- If the selection criterion for a descriptor always involves a range of values, a subdescriptor may be used;
- If the selection criterion for a descriptor never involves the selection of null value, and a large number of null values are possible for the descriptor, the descriptor should be defined with the null-value suppression (NU) option;

- If a field is updated very frequently, it should normally not be defined as a descriptor;
- Files that have a high degree of volatility (large number of additions and deletions) should not contain a large number of descriptors.

Collation Descriptor

A collation descriptor is used to sort (collate) descriptor field values in a special sequence based on a user-supplied algorithm. An alpha or wide field can be defined as a parent field of a collation descriptor.

Special collation descriptor user exits are specified using the ADARUN parameter CDXnn (CDX01 through CDX08). The user exits are used encode the collation descriptor value or decode it back to the original field value. Each collation descriptor must be assigned to a user exit, and a single user exit may handle multiple collation descriptors.

Superdescriptor

A superdescriptor is a descriptor created from a combination of up to 20 fields (or portions of fields). The fields from which a superdescriptor is derived may or may not be descriptors. Superdescriptors are more efficient than combinations of ordinary descriptors when the search criteria involve a combination of values. This is because Adabas accesses one inverted list instead of several and does not have to ‘AND’ several ISN lists to produce the final list of qualifying records. Superdescriptors can also be used in the same manner as ordinary descriptors to control the logical sequence in which a file is read.

The values for search criteria that use superdescriptors must be provided in the format of the superdescriptor (binary for superdescriptors derived from all numeric fields, otherwise alphanumeric). If the superdescriptor format is binary, the input of the search value using an interactive query or report facility such as Natural may be difficult.

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

Subdescriptor

A descriptor that is derived from a portion of a field is called a subdescriptor. The field used to derive the subdescriptor may or may not be a descriptor. If a search criteria involves a range of values that is contained in the first ‘n’ bytes of an alphanumeric field or the last ‘n’ bytes of a numeric field, a subdescriptor may be defined from only the relevant bytes of the field. Using a subdescriptor allows the search criterion to be represented as a single value rather than a range. This results in more efficient searching, since Adabas does not need to merge intermediate ISN lists; the merged list already exists.

For example, assume an alphanumeric field AREA of 8 bytes, the first 3 of which represent the region and the last 5 the department. If only records for region ‘111’ are desired, a search criterion of ‘AREA = 11100000 thru 11199999’ would be required without a subdescriptor. If the first three bytes of AREA were defined as a subdescriptor, a search criterion equal to ‘REGION = 111’ can be specified.

Phonetic Descriptor

A phonetic descriptor may be defined to perform phonetic searches. Using a phonetic descriptor in a Find command returns all the records that contain similar phonetic values. The phonetic value of a descriptor is based on the first 20 bytes of the field value with only alphabetic values being considered (numeric values, special characters and blanks are ignored).

Hyperdescriptor

The hyperdescriptor option enables descriptor values to be generated 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.

Hyperdescriptors can be used to implement n-component superdescriptors, derived keys, or other key constructs. For more information about hyperdescriptors, see the documentation on User and Hyperexits, as well as the ADACMP utility description in the Adabas Utilities documentation.

File Coupling

Using a single Find command, file coupling allows the selection of records from one file that are related (coupled) to records containing specified values in a second file. For example, assume two files, CUSTOMER and ORDERS, that contain customer and order information, respectively. Each file contains the descriptor CUSTOMER_NUMBER, which is used as the basis for relating (coupling) the files.

- Physical Coupling
- Logical Coupling

Physical Coupling

The files are physically coupled using the ADAINV utility, which creates a pair of additional indices in the inverted list indicating which records in the CUSTOMER file are related (coupled) to records in the ORDERS file (that is, have the same customer number) and vice versa. Once the files have been coupled, a single Find command containing descriptors from either file may be constructed, for example:

```
FIND CUSTOMER WITH NAME = JOHNSON
             AND COUPLED TO ORDERS
             WITH ORDER-MONTH = JANUARY
```

Physical coupling may be useful for information retrieval systems in which file volatility is very low, or the additional overhead of the coupling lists is deemed insignificant compared with the ease with which queries may be formulated. It may also be useful for small files which are primarily query-oriented.

Physical coupling may involve a considerable amount of additional 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 requires additional disk space for the storage of the coupling indices. The space required depends on the number of records that are related (coupled). The best case is where the coupling descriptor is a unique key for one of the files. This means that only a few records in one file will be coupled to a given record in the other file. The worst case is when a many-to-many relationship exists between the files. This will result in a large number of records being coupled to other records in both files.

A descriptor used as the basis for coupling should normally be defined with the null suppression option so that records containing a null value are not included in the coupling indices.

See the Adabas Utilities documentation, the ADAINV utility, for additional information on the use of coupling.

Logical Coupling

A multifile query may also be performed by specifying the field to be used for interfile linkage in the search criteria. This feature is called logical coupling and does not require the files to be physically coupled.

Although this technique requires read commands, it is normally more efficient if the coupling descriptor is volatile because it does not require any physical coupling lists. It should also be noted that the user program need only specify the search criteria and the field to be used for the soft-coupling link. *Adabas performs all necessary search, read and internal list matching operations.*

User-Assigned ISNs

The user has the option of assigning the ISN of each record in a file rather than having this done by Adabas. This technique permits later data retrieval using the ISN directly rather than using the inverted list technique. This requires that the user develop his own method for the assigning a unique ISN to each record. The resulting ISNs must be within the range of the MINISN and MAXISN parameter values specified by the ADALOD utility when the file is loaded.

Using the ISN as a Descriptor

The user may store the ISN of related records in another record in order to be able to read the related records directly without using the Inverted Lists.

For example, assume an application which needs to read an order record and then find and read all customer records for the order. If the ISN of each customer record (for more than one customer per order, a multiple-value field could be used) were stored in the order record, the customer records could be read directly since the ISN is available in the order record.

Storing the customer record ISNs avoids having to issue a FIND command to the customer file to determine the customer records for the order. This technique requires that the field containing the ISNs of the customer records be established and maintained in the order record, and assumes that the ISN assignment in the customer file will not be changed by a file unload and reload operation.

ADAM Usage

The Adabas direct access method (ADAM) facility permits the retrieval of records directly from Data Storage without access to 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.

The ADAM key of each record must be a unique value. The ISN of a record may also be used as the ADAM key.

While accessing ADAM files is significantly faster, adding new records to and loading of ADAM files is slower than for standard files because successive new records will not generally be stored in the same block.

If an ADAM file is to be processed both randomly and in a given logical sequence, the logical sequential processing may be optimized by using the bit truncation feature of the ADALOD utility. This feature permits the truncation of a user-specified number of bits from the rightmost portion of each ADAM key value prior to its usage as input to the randomizing algorithm. This will cause records of keys with similar

leftmost values to be stored in the same Data Storage block.

It is important not to truncate too many bits, however, as this may increase the number of overflow records and degrade random access performance. The reason is, overflow records which cannot be stored in the blocks located using the ADAM key are stored elsewhere using the standard inverted list process; overflow records must also be located using the inverted list. The only other way to minimize overflow is to specify a relatively large file and padding factor size.

ADAM will generally use an average of 1.2 to 1.5 I/O operations (including an average of overflow records stored under Associator control in other blocks of the file), rather than the three to four I/O operations required to retrieve a record using the inverted lists. Overflow records are also retrieved using normal Associator inverted list references.

The variable factors of an ADAM file that affect performance are, therefore, the amount of disk space available (the more space available, the fewer the overflow records which need to be located with an inverted list), the number of bits truncated from the ADAM key, and the amount of record adding and update activity. The ADAMER utility may be used to determine the average number of I/O operations for various combinations of disk space and bit truncation. See the Adabas Utilities documentation for additional information.

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 section 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
- Ordinary Compression
- Null-Value Suppression

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.

Adabas Security

This section describes general considerations for database security and introduces the security facilities provided by Adabas and the Adabas subsystems. Detailed information about the facilities discussed in this section may be found in other parts of this documentation and in the Adabas Security documentation.

This section covers the following topics:

- Security Planning
- Password Security
- Security by Value
- Ciphering
- Using Adabas SAF Security
- Natural and Adabas Online System Security

Security Planning

Effective security measures must take account of the following:

- A system is only as secure as its weakest component. This may be a non-DP area of the system: for example, failure to properly secure printed listings;
- It is rarely possible to design a foolproof system. A security system will probably be breached if the gain from doing so is likely to exceed the cost;
- Security can be expensive. Costs include inconvenience, machine resources, and the time required to coordinate the planning of security measures and monitor their effectiveness.

The cost of security measures is usually much easier to quantify than the risk or cost of a security violation. The calculation may, however, be complicated by the fact that some security measures offer benefits outside the area of security. The cost of a security violation depends on the nature of the violation. Possible types of cost include

- loss of time while the violation is being corrected;
- penalties under privacy legislation, breach of contracts, and so on;
- damage to relationships with customers, suppliers, employees, and so on.

Password Security

Password security allows the DBA to control a user's use of the database by

- restricting the user to certain files;
- specifying for each file whether the user can access and update, or access only;
- preventing the user from accessing or updating certain fields while allowing access or update of other fields in the same file;
- restricting the user's view of the file to records that contain specified field values (for example, department code).

The DBA can assign a security level to each file and each field within a file. In the following table, *x/y* indicates the access/update security level. The value 0/0 indicates no security.

File	Fields	
1 (2/3)	AA (0/0)	BB (4/5)
2 (6/7)	LL (6/7)	MM (6/9)
3 (4/5)	XX (4/5)	YY (4/5)
4 (0/0)	FF (0/0)	GG (0/15)

A user must supply an appropriate password to access/update a secured file. In the following table, *x/y* indicates the password access/update authorization level.

	Passwords	
	ALPHA	**BETA**
File 1	2/3	4/5
File 2	0/0	6/7
File 3	4/5	0/0

Assuming the files, fields, and passwords shown in the above tables, the following statements are true:

- Password ALPHA
  - can access and update field AA in file 1, but not field BB;
  - can access and update all fields in file 3;

  - cannot access or update file 2.
- Password BETA
  - can access and update all fields in file 1;
  - can access all the fields in file 2 and can update field LL, but not field MM;
  - cannot access or update file 3.
- No password is required to access any field in file 4, or to update field FF.
- Field GG in file 4 can be read only. Its update security level is 15 and the highest possible authorization level is 14.

If password BETA can access a field that password ALPHA cannot, then password BETA can also access all the fields in the same file that password ALPHA can access. There is no way in which ALPHA can be authorized to access field AA but not field BB and password BETA to access BB but not AA. The same restriction applies to update (although not necessarily to the same combinations of fields or to the advantage of the same password). ALPHA could be permitted to update all the fields which BETA can update and some others which BETA cannot update.

This restriction does not apply to file-level security. For example, ALPHA can use file 3 but not file 2, and BETA can use file 2 but not file 3. When a record is being added to a file, Adabas only checks the update security level on those fields for which the user is supplying values. For example, the password ALPHA could be used to add a record to file 1 provided that no value was specified for field BB. This could represent the situation where, for example, a customer record is only to be created with a zero balance. For record deletion, the password provided must have an authorization level equal to or greater than the highest update security level present in the file. For example, an update authorization level of 9 is required to delete a record from file 2, and, it is not possible to delete records from file 4.

Security by Value

It is also possible to limit access/update fields within a file based on the contents of the field in the file. See the Adabas Security documentation for more information.

Ciphering

Adabas is able to cipher (encrypt) records when they are initially loaded into a file or when records are being added to a file. Ciphering makes it extremely difficult to read the contents of a copy of the database obtained from a physical dump of the disk on which the database is contained. Ciphering applies to the records stored in Data Storage only. No ciphering is performed for the Associator.

Using Adabas SAF Security

Adabas SAF Security, an Adabas add-on product, can be used with Software AG’s Com-plete and with the following non-Software AG security environments:

- CA-ACF2 (Computer Associates);

- CA-Top Secret (Computer Associates);
- RACF (IBM Corporation)

For more information about Adabas SAF Security, contact your Software AG representative.

Natural and Adabas Online System Security

The Natural Security system may also be used to provide extensive security provisions for Adabas/Natural users. See the Natural Security documentation for additional information.

Access to the DBA facility Adabas Online System (AOS) can also be restricted. AOS Security requires Natural Security as a prerequisite.

Recovery/Restart Design

This section discusses the design aspects of database recovery and restart. Proper recovery and restart planning is an important part of the design of the system, particularly in a database environment. Although Adabas provides facilities to perform both restart and recovery, the functions must be considered separately.

This section covers the following topics:

- Adabas Recovery
- Planning and Incorporating Recoverability
- Matching Requirements and Facilities
- Transaction Recovery
- End Transaction (ET) Command
- Close (CL) Command
- Reading ET Data
- System or Transaction Failure
- Limitations of Adabas Transaction Recovery
- Adabas Checkpoint Commands
- Exclusive File Control
- User Restart Data

Adabas Recovery

Recovery of database integrity has the highest priority; if a database transaction fails or must be cancelled, the effects of the transaction must be removed and the database must be restored to its exact condition before the transaction began.

The standard Adabas system provides transaction logic (called *ET logic*), extensive checkpoint/logging facilities, and transaction-reversing backout processing to ensure database integrity.

Restarting the database following a system failure means reconstructing the task sequence from a saved level before the failure, up to and including the step at which the failure occurred-including, if possible, successfully completing the interrupted operation and then continuing normal database operation. Adabas provides a recovery aid that reconstructs a recovery job stream to recover the database.

Recoverability is often an implied objective. Everyone assumes that whatever happens, the system can be systematically recovered and restarted. There are, however, specific facts to be determined about the level of recovery needed by the various users of the system. Recoverability is an area where the DBA needs to take the initiative and establish necessary facts. Initially, each potential user of the system should be questioned concerning his recovery/restart requirements. The most important considerations are

- how long the user can manage without the system;
- how long each phase can be delayed;
- what manual procedures, if any, the user has for checking input/output and how long these take;
- what special procedures, if any, need to be performed to ensure that data integrity has been maintained in a recovery/restart situation.

Planning and Incorporating Recoverability

Once the recovery/restart requirements have been established, the DBA can proceed to plan the measures necessary to meet these requirements. The methodology provided in this section may be used as a basic guideline.

1. A determination should be made as to the level and degree to which data is shared by the various users of the system.
2. The recovery parameters for the system should be established. This includes a predicted/actual breakdown rate, an average delay and items affected, and items subject to security and audit.
3. A determination should be made as to what, if any, auditing procedures are to be included in the system.
4. An outline containing recovery design points should be prepared. Information in this outline should include
    - validation planning. Validation on data should be performed as close as possible to its point of input to the system. Intermediate updates to data sharing the record with the input will make recovery more difficult and costly;
    - dumps (back-up copies) of the database or selected files;
    - user and Adabas checkpoints;
    - use of ET logic, exclusive file control, ET data;

- audit procedures.

5. Operations personnel should be consulted to determine if all resources required for recovery/restart can be made available if and when they are needed.

6. The final recovery design should be documented and reviewed with users, operations personnel, and any others involved with the system.

Matching Requirements and Facilities

Once the general recovery requirements have been designed, the next step is to select the relevant Adabas and non-Adabas facilities to be used to implement recovery/restart. The following sections describe the Adabas facilities related to recovery/restart.

Transaction Recovery

Almost all online update systems and many batch update programs process streams of input transactions which have the following characteristics:

- The transaction requires the program to retrieve and add, update, and/or delete only a few records. For example, an order entry program may retrieve the customer and product records for each order, add the order and order item data to the database, and perhaps update the quantity-on-order field of the product record.

- The program needs exclusive control of the records it uses from the start of the transaction to the end, but can release them for other users to update or delete once the transaction is complete.

- A transaction must never be left incomplete; that is, if it requires two records to be updated, either both or neither must be changed.

End Transaction (ET) Command

The use of the Adabas ET command

- ensures that all the adds, updates, and/or deletes performed by a completed transaction are applied to the database;

- ensures that all the effects of a transaction which is interrupted by a total or partial system failure are removed from the database;

- allows the program to store up to 2000 bytes of user-defined restart data (ET data) in an Adabas system file. This data may be retrieved on restart with the Adabas OP or RE commands. The restart data can be examined by the program or TP terminal user to decide where to resume operation;

- releases all records placed in hold status while processing the transaction.

Close (CL) Command

The Adabas CL command can be used to update the user's current ET data (for example, to set a user-defined *job completed* flag). Refer to the section *User Restart Data* for more information.

Reading ET Data

After a user restart or at the start of a new user or Adabas session, ET data can be retrieved with the OP command. The OP command requires a user ID, which Adabas uses to locate the ET data, and a command option to read ET data.

The RE command can also be used to read ET data for the current or a specified user; for example, when supervising an online update operation.

System or Transaction Failure

The autobackout routine is automatically invoked at the beginning of every Adabas session. If a session terminates abnormally, the autobackout routine removes the effects of all interrupted transactions from the database up to the most recent ET. If an individual transaction is interrupted, Adabas automatically removes any changes the transaction has made to the database. Each application program can explicitly back out its current transaction by issuing the Adabas BT command.

Limitations of Adabas Transaction Recovery

The transaction recovery facility recovers only the contents of the database. It does not recover TP message sequences, reposition non-Adabas files, or save the status of the user program.

It is not possible to back out the effects of a specific user’s transactions because other users may have performed subsequent transactions using the records added or updated by the first user.

Adabas Checkpoint Commands

Some programs cannot conveniently use ET commands because:

- the program would have to hold large numbers of records for the duration of each transaction. This would increase the possibility of a deadlock situation (Adabas automatically resolves such situations by backing out the transaction of one of the two users after a user-defined time has elapsed, but a significant amount of transaction reprocessing could still result), and a very large Adabas hold queue would have to be established and maintained;

- the program may process long lists of records found by complex searches; restarting part of the way through such a list may be difficult.

Such programs can use the Adabas checkpoint command (C1) to establish a point to which the file or files the program is updating can be restored if necessary.

Exclusive File Control

A user can request exclusive update control of one or more Adabas files. Exclusive control is requested with the OP command and will be given only if the file is not currently being updated by another user. Once exclusive control is assigned to a user, other users may read but not update the file. Programs that read and/or update long sequences of records, either in logical sequence or as a result of searches, may use exclusive control to prevent other users from updating the records used. This avoids the need for placing each record in hold status.

Exclusive control users may or may not use ET commands. If ET commands are not used, checkpoints can be taken by issuing a C1 command.

In the event of a system or program failure, the file or files being updated under exclusive control may be restored using the BACKOUT function of the ADARES utility. This utility is not automatically invoked and requires the Adabas data protection log as input. This procedure is not necessary if the user uses ET commands (see the section *Transaction Recovery*).

The following limitations apply to exclusive file control:

- Recovery to the last checkpoint is not automatic, and the data protection log in use when the failure occurred is required for the recovery process. This does not apply if the user issues ET commands.
- In a restart situation following a system failure, Adabas does not check nor prevent other users from updating files which were being updated under exclusive control at the time of the system interruption.

User Restart Data

The Adabas ET and CL commands provide an option of storing up to 2000 bytes of user data in an Adabas system file. One record of user data is maintained for each user. This record is overwritten each time new user data is provided by the user. The data is maintained from session to session only if the user provides a user identification (user ID) with the OP command.

The primary purpose of user data is to enable programs to be self-restarting and to check that recovery procedures have been properly carried out. The type of information which may be useful as user data includes the following:

- The *date and time* of the original program run and the time of last update. This will permit the program to send a suitable message to a terminal user, console operator, or printer to allow the user and/or operator to check that recovery and restart procedures have operated correctly. In particular, it will allow terminal users to see if any work has to be rerun after a serious overnight failure of which they were not aware.
- The *date of collection* of the input data.
- *Batch numbers*. This will enable supervisory staff to identify and allocate any work that has to be reentered from terminals.
- *Identifying data*. This data can be a way for the program to determine where to restart. For example, a program driven by a logical sequential scan needs to know the key value at which to resume.
- *Transaction number/input record position* . This may allow an interactive user or batch program to locate the starting point with the minimum of effort. Although Adabas returns a transaction sequence number for each transaction, the user also may want to maintain a sequence number because
    - after a restart, the Adabas sequence number is reset;
    - if transactions vary greatly in complexity, there may not be a simple relationship between the Adabas transaction sequence number and the position of the next input record or document;

  - if a transaction is backed out by the program because of an input error, Adabas does not know whether the transaction will be reentered immediately (it may have been a simple keying error) or rejected for later correction (if there was a basic error in the input document or record);

- *Other descriptive or intermediate data*; for example, totals to be carried forward, page numbers and headings of reports, run statistics.

- *Job/batch completed flag*. The system may fail after all processing has been completed but before the operator or user has been notified. In this case, the operator should restart the program which will be able to check this flag without having to run through to the end of the input. The same considerations apply to batches of documents entered from terminals.

- *Last job/program name*. If several programs must update the database in a fixed sequence, they may share the same user ID and use user data to check that the sequence is maintained.

A user's own data can be read with either the OP or RE command. User data for another user can be read by using the RE command and specifying the other user's ID. User data for all users can be read in logical sequential order using the RE command with a command option; in this case, user IDs are not specified.

The Adabas Recovery Aid

When a system failure disrupts database operation, the Adabas Recovery Aid can create a job stream that reconstructs the database to the point of failure.

The Recovery Aid combines the protection log (PLOG) and the archived database status from previous ADASAV operations with its own recovery log (RLOG) information to reconstruct the job sequence. The result is a reconstructed job statement string (recovery job stream) that is placed in a specially named output data set.

The two major parts of the Adabas Recovery Aid are the recovery log (RLOG) and the recovery aid utility ADARAI. The RLOG is formatted like other Adabas files, using ADAFRM, and then defined with the ADARAI utility.

The DBA must run the Recovery Aid utility, ADARAI, to

- define the RLOG and set up the Recovery Aid environment;
- display current RLOG information;
- create the recovery job stream.

This section covers the following topics:

- The Recovery Log (RLOG)
- Starting the Recovery Aid

The Recovery Log (RLOG)

The recovery log (RLOG) records the essential information that, when combined with the PLOG, is used by the ADARAI utility's RECOVER function to rebuild a job stream to recover and restore the database status up to the point of failure.

The RLOG information is grouped in generations, where each generation comprises the database activity between consecutive ADASAV SAVE, RESTORE (database) or RESTORE GCB operations. The RLOG holds a minimum of four consecutive generations, up to a maximum value specified when the RLOG is activated; the maximum is 32. If RLOG space is not sufficient to hold the specified number of generations, the oldest generation is overwritten with the newest in wraparound fashion.

The RLOG file is formatted like other database components by running the ADAFRM utility (SIZE parameter), and then defined using the PREPARE function of the Recovery Aid ADARAI utility (with the RLOGSIZE parameter). The space required for the RLOG file is approximately 10 cylinders of 3380 or equivalent device space.

The ADARAI PREPARE function must be performed just before the ADASAV SAVE run that begins the first generation to be logged. After ADARAI PREPARE is executed, all subsequent nucleus and utility jobs that update the database must specify the RLOG file. Of course, the RLOG file can be included in any or all job streams, if desired.

The RLOG file job statement should be similar to the following:

```
//DDRLOGR1 DD DISP=SHR,DSN=... .RLOGR1
```

Starting the Recovery Aid

The activity of the Recovery Aid and RLOG logging begins when the first ADASAV SAVE/RESTORE database or RESTORE GCB function is executed following ADARAI PREPARE.

All activity between the first and second ADASAV SAVE/RESTORE database or RESTORE GCB operations following the ADARAI PREPARE operation belongs to the first generation. When viewing generations with the ADARAI utility’s LIST function, generations are numbered relatively in ascending order beginning with the oldest generation.

For more detailed information on setting up the Recovery Aid, see *Restart and Recovery* in the Adabas Operations documentation and the ADARAI utility description in the Adabas Utilities documentation.

Multiclient Support

The Adabas multiclient feature stores records for multiple users or groups of users in a single Adabas file. This feature is specified at the file level. It divides the physical file into multiple logical files by attaching an owner ID to each record. Each user can access only the subset of records that is associated with the user’s owner ID. The file is still maintained as a single physical Adabas file.

The Adabas nucleus handles all database requests to multiclient files.

This section covers the following topics:

- The Owner Concept
- Superusers
- Program Compatibility

- Support for Soft Coupling
- Data and Index Structures
- Performance Considerations
- User Profile Table
- Possible Adabas Response Codes
- Utility Support for Multiclient Files

The Owner Concept

Each record in a multiclient file has a specific owner, which is identified by an internal owner ID attached to each record (for any installed external security package such as RACF or CA-Top Secret, a user is still identified by either Natural ETID or LOGON ID). 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.

The following table shows examples of the ETID/owner ID relationship.

<table>
<tr><th>ETID</th><th>Owner ID</th><th>Description</th></tr>
<tr><td>USER1</td><td>1</td><td rowspan="5">More than one user can use the same owner ID.

Here, USER1, USER2 and USER3 share the same owner ID and therefore the same records.</td></tr>
<tr><td>USER2</td><td>1</td></tr>
<tr><td>USER3</td><td>1</td></tr>
<tr><td>...</td><td></td></tr>
<tr><td>USER4</td><td>2</td></tr>
</table>

The relationship between the user ID and the owner ID is stored in the profile table in the Adabas checkpoint file. The DBA maintains the profile table using Adabas Online System/Basic Services (AOS), a prerequisite for the multiclient feature.

The relation between user ID and owner ID is 1:1 or n:1; either a single user or group of users can be assigned to one owner ID. Record isolation is always performed on the owner ID level.

The owner ID has a fixed length of up to 8 bytes (alphanumeric). The length is defined by the user during file creation; it can be changed only by unloading and reloading the multiclient file. Each owner ID must be less than or equal to the length assigned for the file; otherwise, a nonzero response code occurs. To avoid wasting space, make the owner ID no larger than necessary.

The following tables show an example of owner isolation for a group of eight file records.

ISN	Owner ID	Record	Discussion
1	1	data	Example for a physical Adabas file with records owned by different users
2	2	data	
3	1	data	
4	3	data	
5	2	data	
6	3	data	
7		- no data -	
8	1	data	

ISN	Record	ISN	Record
1	data	1	- no data -
2	- no data -	2	data
3	data	3	- no data -
4	- no data -	4	- no data -
5	- no data -	5	data
6	- no data -	6	- no data -
7	- no data -	7	- no data -
8	data	8	- no data -
File as seen by a user with an owner ID=1		File as seen by a user with an owner ID=2	

Superusers

A *superuser* owner ID provides access to all records in a multiclient file. A superuser owner ID begins with an asterisk (*). Adabas allows users with such an owner ID to match with any other owner ID, allowing the user to read all records in a file. More than one superuser owner IDs, each beginning with an asterisk and allowing identical privileges, can be defined for a multiclient file.

A superuser owner ID applies only to L*x* read commands and nondescriptor search (S*x*) commands. Descriptor search commands by a superuser return only the records having the superuser’s owner ID. Data records or index values stored by a superuser are labeled with the superuser’s owner ID.

Note:
If a superuser issues an L3 or L9 command, the value start option is ignored; that is, Adabas always starts at the very beginning of the specific descriptor.

Program Compatibility

No changes to an existing application program are needed to use it in a multiclient environment; however, a user ID must be supplied in the Additions 1 field of the Adabas control block of each open (OP) call made by a user who addresses a multiclient file. This allows Adabas to retrieve the owner ID from the checkpoint file. Otherwise, the application program neither knows nor cares whether a multiclient file or a standard Adabas file is being accessed.

Support for Soft Coupling

Multiclient support is provided for soft coupling.

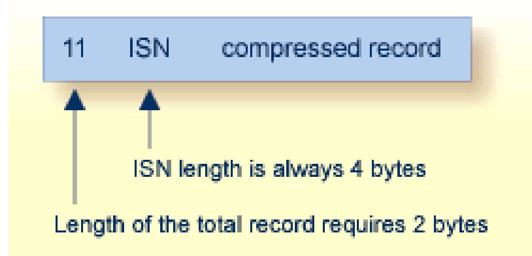
Data and Index Structures

The data and index structures of a multiclient file differ from those of standard Adabas files.

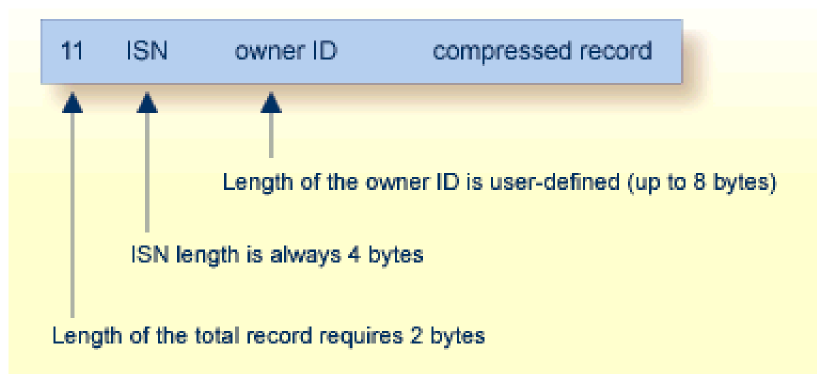
- Data Storage
- Associator

Data Storage

A Data Storage (DATA) record in a standard file has the following structure:



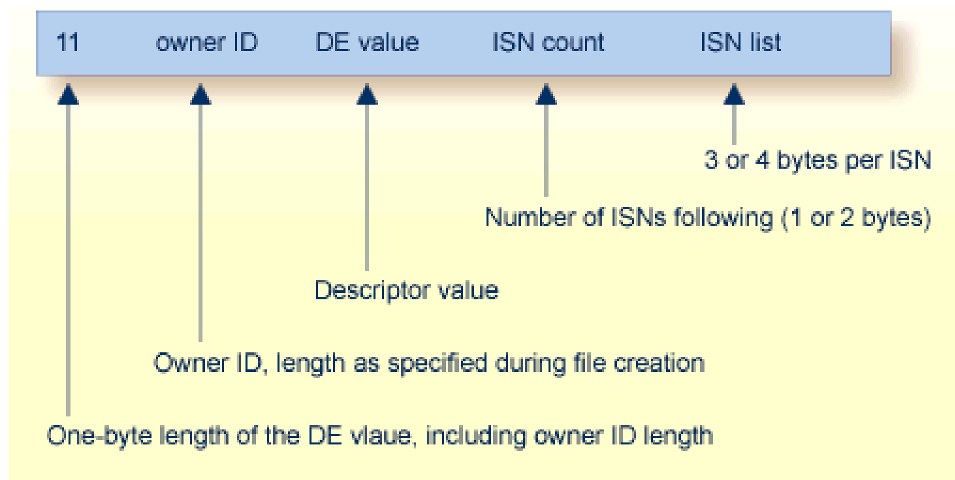
A Data Storage record in a multiclient file has the following structure:



Comparison of Normal and Multiclient Record Formats

Associator

Every normal index and upper index value for a multiclient file is prefixed by the owner ID:



Normal Index Element Structure

The tables below illustrate a multiclient index structure. If a single descriptor value points to more than one Data Storage record, Adabas stores this extended index value only once, followed by the list of ISNs. If the same descriptor value for different owner IDs is to be stored, then multiple entries are made in the index.

ISN	Owner ID	NAME	
1	1	SMITH	The field NAME is a descriptor.
2	2	SMITH	
3	1	SMITH	
4	3	JONES	
5	2	JONES	
6	3	HARRIS	
7		- not stored -	
8	1	HARRIS	

Owner ID	DE value	ISN count	ISN list	
1	HARRIS	1	8	This is the index for the descriptor NAME. The sort sequence of values is: owner ID | (DE-value)
1	SMITH	2	1,3	
2	JONES	1	5	
2	SMITH	1	2	
3	HARRIS	1	6	
3	JONES	1	4	

Notes:

1. Every type of descriptor is prefixed by the owner ID: simple descriptors, sub/superdescriptors, phonetic, and hyperdescriptors. The owner ID prefix is not counted as a parent field for super- and hyperdescriptors. The maximum number of parent fields is not affected.
2. The maximum length of a descriptor value, *including* the owner ID, is 253 bytes.
3. A superuser reading index values in L3/L9 sequence gets values in sorting order by owner ID: the values for the lowest owner ID first, then the values for the next higher owner ID, and so on. Values for each owner ID are sorted in ascending order.

Performance Considerations

The multiclient feature causes no added processing overhead for find (S1,S2), read-logical (L3) and histogram (L9) commands. The index structure permits specific record selection, and there is no postselection procedure in the Data Storage.

If the selection is done on the Data Storage, Adabas must read the record and check the owner ID. If the record's owner ID does not match the current user's owner ID, the record is skipped. This might slow down a read-physical (L2) and a read-by-ISN (L1 with I option) command or a nondescriptor search command.

User Profile Table

The owner ID is part of the user's profile record, which is stored in the Adabas profile table. The profile is maintained using the Adabas Online System. See the Adabas Online System documentation for more information.

Possible Adabas Response Codes

Calls to multiclient files can result in the following non-zero Adabas response codes, which indicate that an error has occurred:

Read and Update Operation	If a user tries to read or change a multiclient file's record using an owner ID that does not apply to the record, Adabas returns either response code 3 (ADARSP003) or 113 (ADARSP113), depending on the type of read or update operation.
Add Record Operation	If a user has an owner ID that is either blank or too long for the owner ID length assigned to the multiclient file, Adabas returns response code 68 (ADARSP068) if this owner tries to add a new record.
Blank or Missing Owner IDs	A user with a blank or missing owner ID receives response code 3 (ADARSP003) or 113 (ADARSP113) when trying to access a multiclient file.

Utility Support for Multiclient Files

In general, multiclient files are transparent to Adabas utility processing. Special functions of the ADALOD and ADAULD utilities support the migration of an application from a standard to a multiclient environment.

- The ADALOD Utility LOAD Function
- The ADALOD Utility UPDATE Function
- The ADAULD Utility
- The ADACMP Utility

The ADALOD Utility LOAD Function

Two ADALOD LOAD parameters LOWNERID and ETID support multiclient files. The parameters work together to define owner IDs and determine whether a file is a multiclient file.

LOWNERID specifies the length of the internal owner ID values assigned to each record for multiclient files.

Valid length values are 0-8. In combination with the ETID parameter, the LOWNERID parameter can be used to reload a standard file as a multiclient file, change the length of the owner ID for the file, or remove the owner ID from the records of a file.

If the LOWNERID parameter is not specified, the length of the owner ID for the input file (if any) remains the same.

ETID assigns a new owner ID to all records being loaded into a multiclient file, and must be specified if the input file contains no owner IDs; that is, the input file was not unloaded from a multiclient source file.

The following table illustrates the effects of LOWNERID and ETID settings.

LOWNERID Parameter Setting:	Owner ID Length in Input File:	
	0	2
0	Keep as non-multiclient file	Convert into a non-multiclient file
1	Set up multiclient file (ETID)	Decrease owner ID length
2	Set up multiclient file (ETID)	Keep owner ID length
3	Set up multiclient file (ETID)	Increase owner ID length
(LOWNERID not specified)	Keep as non-multiclient file	Keep as a multiclient file

Where this table indicates (ETID) in the "Owner ID Length...0" column, the ETID parameter must specify the user ID identifying the owner of the records being loaded. The owner ID assigned to the records is taken from the user profile of the specified user ID. In the "Owner ID Length...2" column the ETID parameter is optional; if ETID is omitted, the loaded records keep their original owner IDs.

Note:
If the ETID parameter is used, the ADALOD utility requires an active nucleus. The nucleus will translate the ETID value into the internal owner ID value.

The ADALOD Utility UPDATE Function

When executing the UPDATE function, ADALOD keeps the owner ID length previously defined for the file being updated. The owner IDs of the records being added are adjusted to the owner ID length defined for the file. The owner IDs of the loaded records or of any new records must fit into the existing owner ID space.

Examples:

```
ADALOD LOAD FILE=20,LOWNERID=2,NUMREC=0
```

-creates file 20 as a multiclient file. The length of the internal owner ID is two bytes, but no actual owner ID is specified. No records are actually loaded in the file (NUMREC=0).

```
ADALOD LOAD FILE=20,LOWNERID=2,ETID=USER1
```

-creates file 20 as a multiclient file and loads all supplied records for user USER1. The length of the internal owner ID is two bytes.

```
ADALOD UPDATE FILE=20,ETID=USER2
```

-performs a mass update to add records to file 20, a multiclient file. Loads all the new records for USER2.

The ADAULD Utility

The ADAULD utility unloads records from an Adabas file to a sequential output file. This output file can then be used as input to a subsequent ADALOD operation.

If a multiclient file is unloaded, the output file contains all the unloaded records with their owner IDs. This information can either be retained by the subsequent ADALOD operation, or be overwritten with new information by the ADALOD ETID parameter. Any differences in LOWNERID parameter values for the unloaded and newly loaded file are automatically adjusted by ADALOD.

The ETID parameter of ADAULD can be used to restrict UNLOAD processing to only the records owned by the specified user. If the ETID parameter is omitted, all records are unloaded. If the SELCRIT/SELVAL parameters are specified for a multiclient file, the ETID parameter must also be specified.

Example:

```
ADAULD UNLOAD FILE=20,ETID=USER1
```

-unloads all records owned by USER1 in physical sequence.

The ADACMP Utility

The ADACMP utility either compresses user data from a sequential input file into the Adabas internal structure, or decompresses Adabas data to a sequential user file. The COMPRESS function makes no distinction between standard and multiclient files, processing both in exactly same way. The DECOMPRESS function can decompress records selectively if the INFILE parameter specifies a multiclient file and a valid ETID value is specified.

The DECOMPRESS function skips the owner ID, if present. The output of a DECOMPRESS operation on a multiclient file contains neither owner ID nor any ETID information.

If the INFILE parameter specifies a multiclient file for the DECOMPRESS function, decompression can be limited to records for a specific user using the ETID parameter. ADACMP then reads and decompresses records for the specified user. If the ETID parameter is not specified when decompressing a multiclient file, all records in the file are decompressed.

Example:

```
ADACMP  DECOMPRESS  INFILE=20,ETID=USER1
```

-decompresses records which are owned by USER1 from file 20 to a sequential output file.