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 queueing 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).

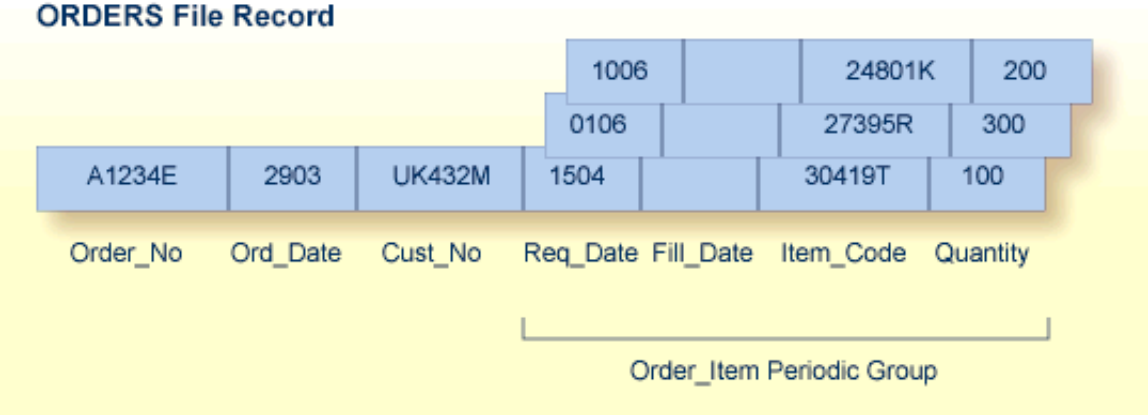
Each of these techniques is described in the following sections.

This chapter covers the following topics:

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



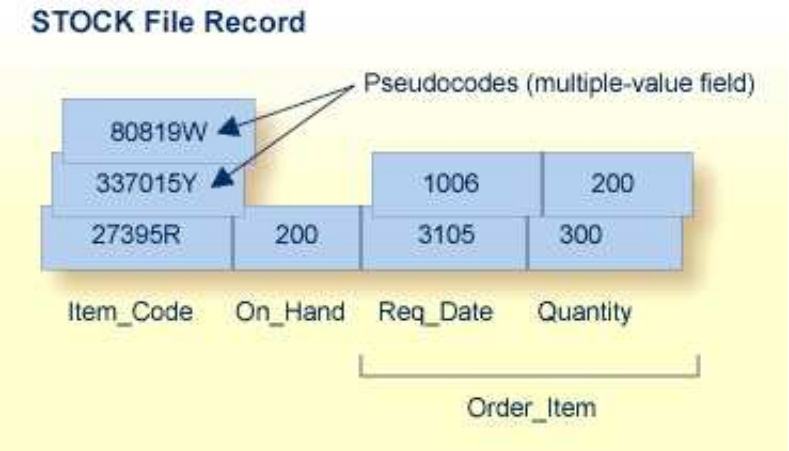
Example of a Periodic Group

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 and the related fields QUANTITY, REQ_DATE, and FILL_DATE, can specify up to 191 different items, quantity desired, and the date needed as well as when the order is actually filled. Each item/quantity/date needed/date filled group is called an "occurrence"; up to 191 occurrences per periodic group are possible.

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 labelled 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 191 values. 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 field is 191;
- The maximum number of occurrences of any one periodic group is 191;
- 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 3-byte ISNs can contain a maximum of 16,777,215 records; a file with 4-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

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 totalling 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.